

CPS 104
Computer Organization
Lecture-7 :MIPS ISA and Assembler

Sep. 17, 1999

Dietolf (Dee) Ramm

<http://www.cs.duke.edu/~dr/cps104.html>

Overview of Today's Lecture:

- **Review: MIPS Assembly Language Programming Conventions.**
- **System calls**
- **Simple Assembly program constructs**
- **Data structures in assembler.**
- **The Stack**
- **Writing Functions in Assembler.**

MIPS: Software conventions for Registers

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)

System call

- System call is used to communicate with the system and do simple I/O.
- Load system call code into Register **\$v0**
- Load arguments (if any) into registers **\$a0**, **\$a1** or **\$f12** (for floating point).
- do: **syscall**
- Results returned in registers **\$v0** or **\$f0**.

code	service	Arguments	Result	comments
1	print int	\$a0		
2	print float	\$f12		
3	print double	\$f12		
4	print string	\$a0		(address)
5	read integer		integer in \$v0	
6	read float		float in \$f0	
7	read double		double in \$f0	
8	read string	\$a0=buffer, \$a1=length		
9	sbrk	\$a0=amount	address in \$v0	
10	exit			

Example: Arrays and pointer arithmetic using C++

```
#include <iostream.h>
int a[100]; // a is a static array
main()
{
    int *p = a;      // p points to a[0]
    int k;

    for (k = 0; k < 100; k++)
    {
        *p = k;      // set the entry value
        p++;        // go to the next array element
    }

    cout << "entry 3 = " << a[3] << endl;
        // we will use simple I/O in Assembler
}

```

Example1: Assembler

```

        .text                               # Code
        .align 2
        .globl main

main:
        subi   $sp, 40                       # MAIN program entrance
        sw     $ra, 36($sp)                  # \ Push the stack
        sw     $fp, 34($sp)                  # \ Save return address
        . . .

        la     $t0, a                        # p = a ; a static array
        sw     $t0, 32($sp)                  # store the address
        move   $t1, $0                       # k=0
L1:     sw     $t1, 0($t2)                    # *p = k
        addiu  $t0, $t0, 4                   # p++
        addi   $t1, $t1, 1                   # k++
        blt   $t1, 100, L1                  # if (k < 100) go to L1
        la    $a0, str                       # load string address
        li    $v0, 4                         #
        syscall                              # Print a string
        lw    $t0, 32($sp)                   # $t0 = a
        lw    $a0, 12($t0)                   # $a0 = a[3]
        li    $v0, 1                         #
        syscall                              # print integer
        . . .
```

Example-2: Array access via index

```
#include <iostream.h>
int a[100]; // a is a static array
main()
{
    int k;

    for (k = 0; k < 100; k++)
        a[k] = k;           // set the array values

    cout << "entry 3 = " << a[3] << endl;
    // we will use simple I/O in Assembler
}
```

Example-2: arrays index in assembler

```

        .text                # Code
        .align 2
        .globl main

main:   # MAIN procedure Entrance
        subiu   $sp, 40      # \ Push the stack
        sw     $ra, 36($sp)  # \ Save return address
        sw     $fp, 34($sp)  #
        la     $t0, a        # $t0 = a
        sw     $t0, 32($sp)  # store a's address
        move   $t1, $0       # k=0
L1:     mul    $t2, $t1, 4    # $t2 = 4*k
        addu   $t3, $t0, $t2 # $t3 = a + 4*k
        sw     $t1, 0($t3)   # a[k] = k
        addi   $t1, $t1, 1   # k++
        blt   $t1, 100, L1   # if (k < 100) go to L1
        la    $a0, str
        li    $v0, 4         # \
        syscall             # > Print a string
        lw    $t0, 32($sp)   # $t0 = a
        lw    $a0, 12($t0)   # $a0 = a[3]
        li    $v0, 1         #
        syscall             # print integer

```

● ● ●

Example-3: if (cond.) {•••}

The C++ code

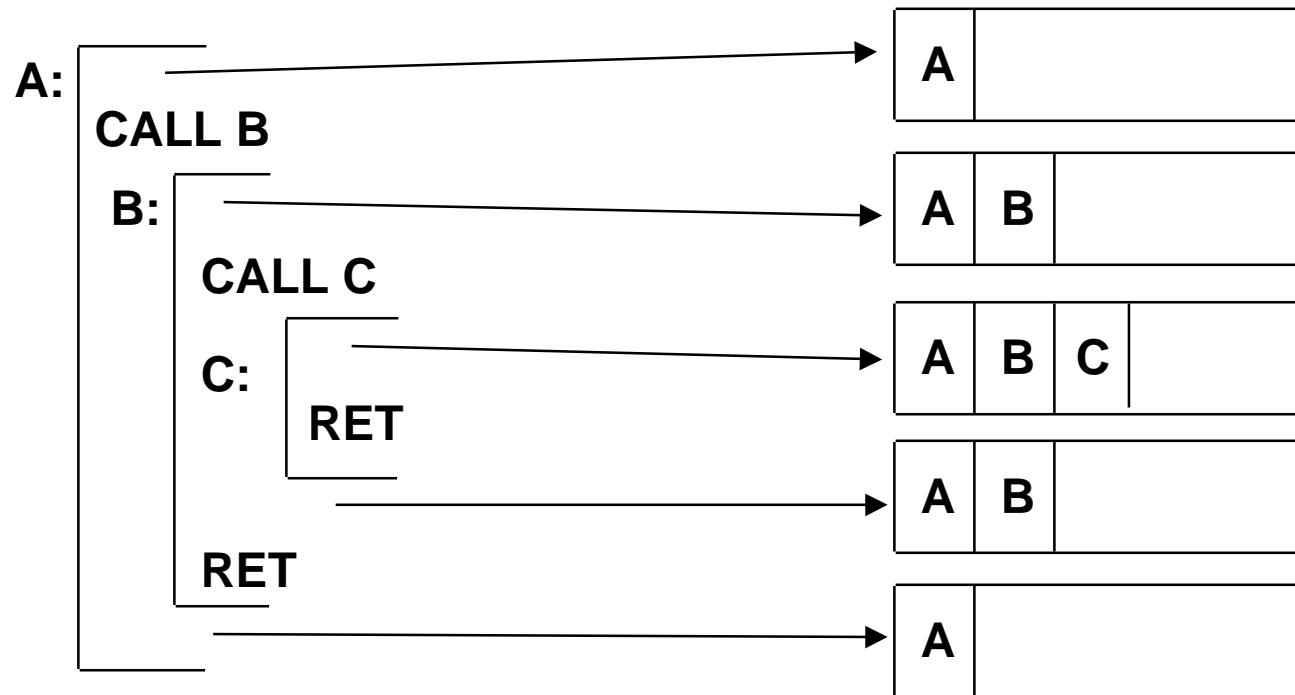
```
If(a < b){  
    temp = a;  
    a = b;  
    b = temp;  
}  
a = a + 5;  
•••
```

Assembler code

```
# assume $t1 = a; $t2=b; $t3=temp  
    ble    $t2, $t1, L1    # if b <= a goto L1  
    move   $t3, $t1        # temp = a  
    move   $t1, $t2        # a = b  
    move   $t2, $t3        # b = temp  
L1: addi   $t1, $t1, 5     # a += 5
```

Calls: Why Are Stacks So Great?

Stacking of Subroutine Calls & Returns and Environments:



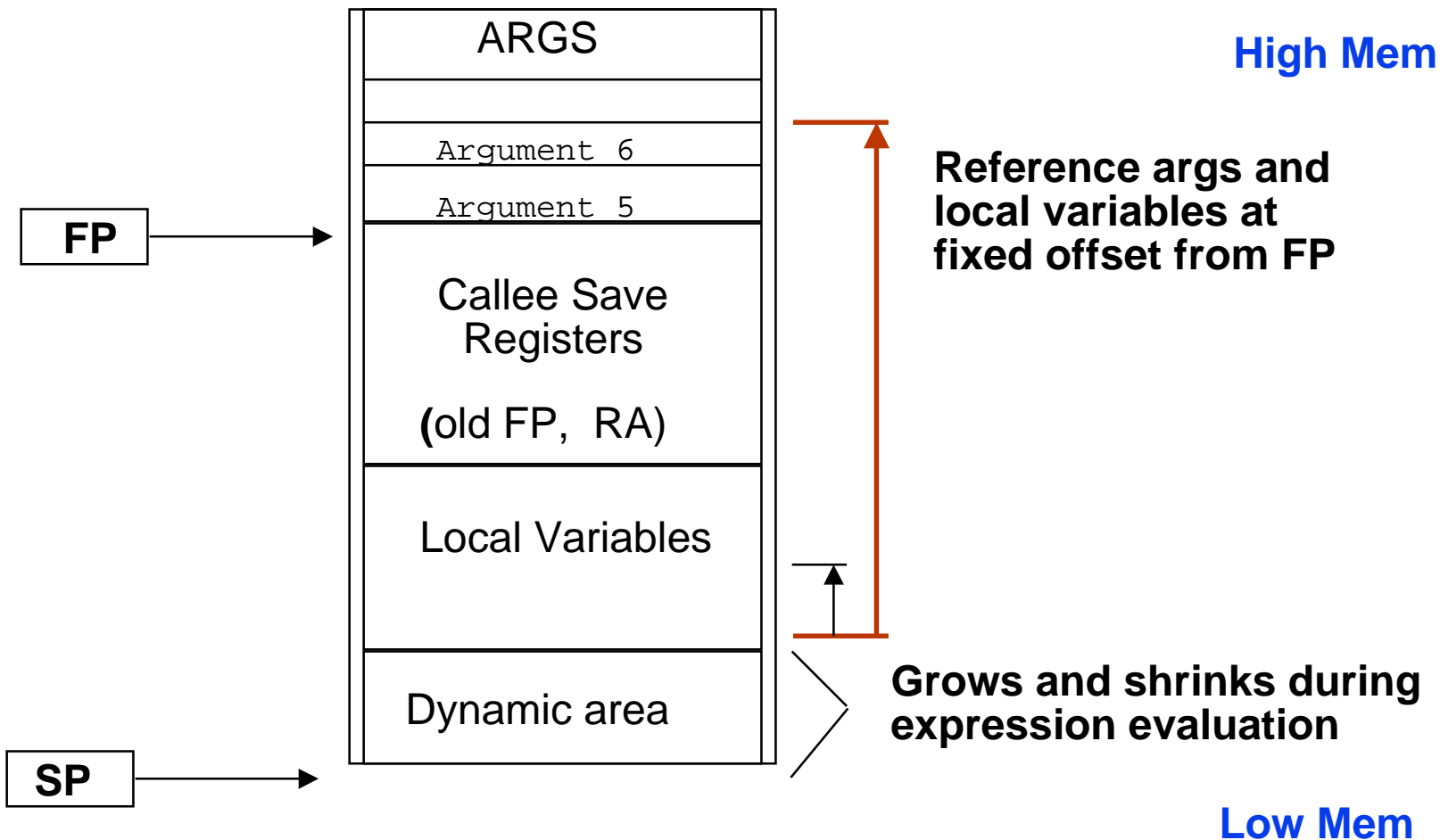
Some machines provide a memory stack as part of the architecture
(e.g., VAX)

Sometimes stacks are implemented via software convention
(e.g., MIPS)

Procedure Call (Stack) Frame

- Procedures use a frame in the stack to:
 - Hold values passed to procedures as arguments.
 - Save registers that a procedure may modify, but which the procedure's caller does not want changed. (ex: **\$s0 - \$s7**)
 - Save the procedure return address (**\$ra**), and frame pointer (**\$fp**)
 - provide space for local variables (variables with local scope)
 - Used to evaluate complex expressions.
- There are two special registers **\$sp** and **\$fp** that are used as special data reference
 - The stack pointer **\$sp** points to the top of the stack.
 - The **\$fp** points to the the frame beginning.

Call-Return Linkage: Stack Frames



- Many variations on stacks possible (up/down, last pushed / next)
- Block structured languages contain link to lexically enclosing frame
- **Compilers normally keep scalar variables in registers, not memory!**

MIPS/GCC Procedure Calling Conventions

Calling Procedure:

- Step-1: Pass the arguments:
 - The **first four arguments** are passed in registers `$a0-$a3`
 - Remaining arguments are pushed into the stack
 - ➔ (in reversed 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/GCC Procedure Calling Conventions (cont.)

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 `$ra` is saved if routine makes a call.
 - Registers `$a0-$a3` are saved if they are changed.
 - Registers `$s0-$s7` are saved if they are used.
- Step-3: Establish Frame pointer `$fp`
 - Add the stack `<frame size -4>` to the address in `$sp`
`addiu $fp, $sp, <frame-size> - 4`

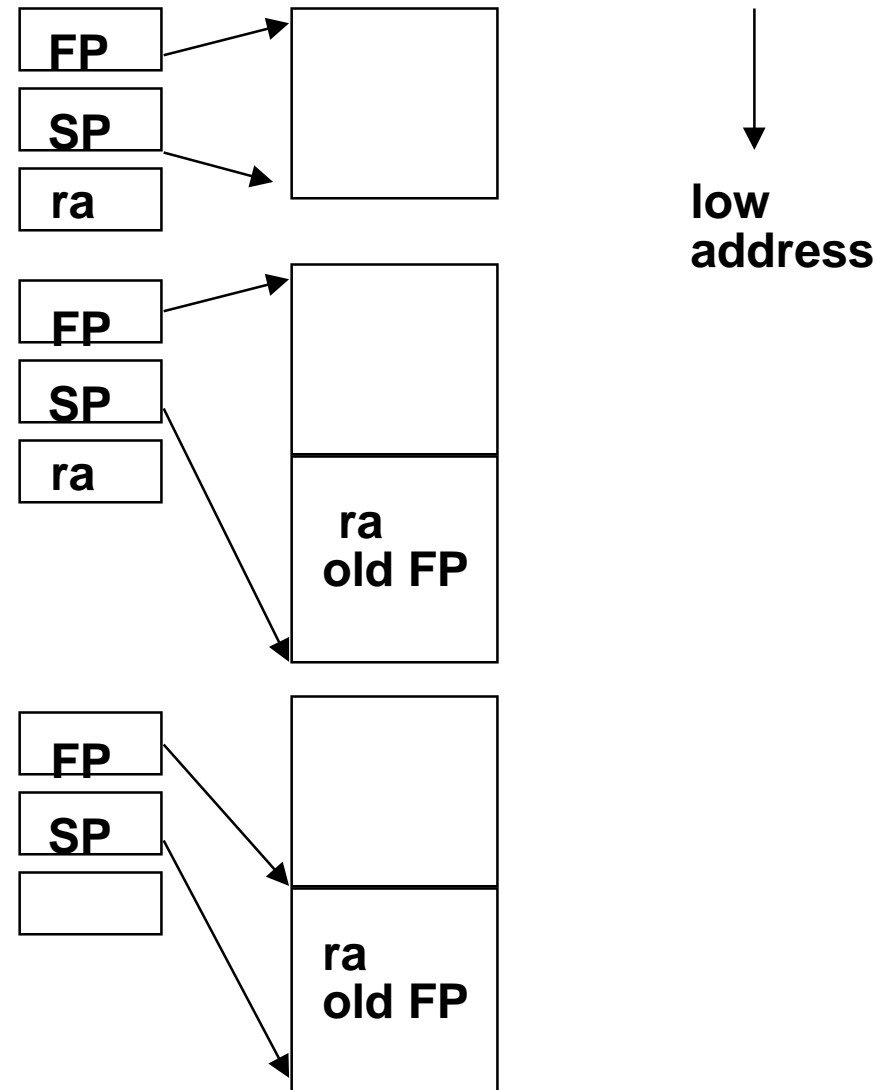
MIPS/GCC Procedure Calling Conventions (cont.)

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`

MIPS / GCC Calling Conventions

```
act:
    subiu $sp, $sp, 32
    sw    $ra, 20($sp)
    sw    $fp, 16($sp)
    f addiu $fp, $sp, 28
    . . .
    sw    $a0, 0($fp)
    ...
    lw    $ra, 20($sp)
    lw    $fp, 16($sp)
    addiu $sp, $sp, 32
    jr    $ra
```



First four arguments are passed in registers.

Example: Factorial

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

```

.text
.global main
main:
    subiu    $sp, $sp, 32    #stack frame size is 32 bytes
    sw      $ra,20($sp)     #save return address
    sw      $fp,16($sp)     #save frame pointer
    addu    $fp, $sp,32     # set frame pointer

    li      $a0,10         # load argument (10) in $a0
    jal     fact           #call fact
    la      $a0,LC         #load string address in $a0
    move    $a1,$v0        #load fact result in $a1
    jal     printf         # call printf

    lw      $ra,20($sp)     # restore $sp
    lw      $fp,16($sp)     # restore $fp
    addu    $sp, $sp,32     # pop the stack
    jr      $ra            # exit()

.rdata
LC:
.asciiz "The factorial of 10 is %d\n"

```

```
.text
fact:
```

```
    subiu    $sp,$sp,32      # stack frame is 32 bytes
    sw      $ra,20($sp)     #save return address
    sw      $fp,16($sp)     #save frame pointer
    addiu   $fp, $sp,28     # set frame pointer
```

```
    sw      $a0,0($fp)     # save argument(n)
    lw      $v0,0($fp)     # load n
    bgtz    $v0, L2        # if n>0 go to $L2
    li      $v0, 1         #
    j       L1             # return(1)
```

```
L2:
```

```
    lw      $v1, 0($fp)    # load n
    sub     $v0,$v1,1      # compute n-1
    move    $a0,$v0        # load argument (n-1) in $a0
    jal     fact           # call fact
    lw      $v1,0($fp)     # load n
    mul     $v0,$v0,$v1    # fact(n-1)*n
```

```
L1:
```

```
    lw      $ra,20($sp)    # restore $ra
    lw      $fp, 16($sp)   # restore $fp
    addiu   $sp,$sp,32     # pop the stack
    jr      $ra            #return
```

Example: Factorial

Stack

○ <code>ld \$ra</code> ○ <code>ld \$fp</code>	Main
○ <code>ld \$ra</code> ○ <code>ld \$fp</code> ○ <code>ld \$a0=10</code>	fact(10)
○ <code>ld \$ra</code> ○ <code>ld \$fp</code> ○ <code>ld \$a0=9</code>	fact(9)
○ <code>ld \$ra</code> ○ <code>ld \$fp</code> ○ <code>ld \$a0=8</code>	fact(8)
○ <code>ld \$ra</code> ○ <code>ld \$fp</code> ○ <code>ld \$a0=7</code>	fact(7)
○ <code>ld \$ra</code> ○ <code>ld \$fp</code> ○ <code>ld \$a0=6</code>	fact(6)
○ <code>ld \$ra</code> ○ <code>ld \$fp</code> ○ <code>ld \$a0=5</code>	fact(5)



Stack grows