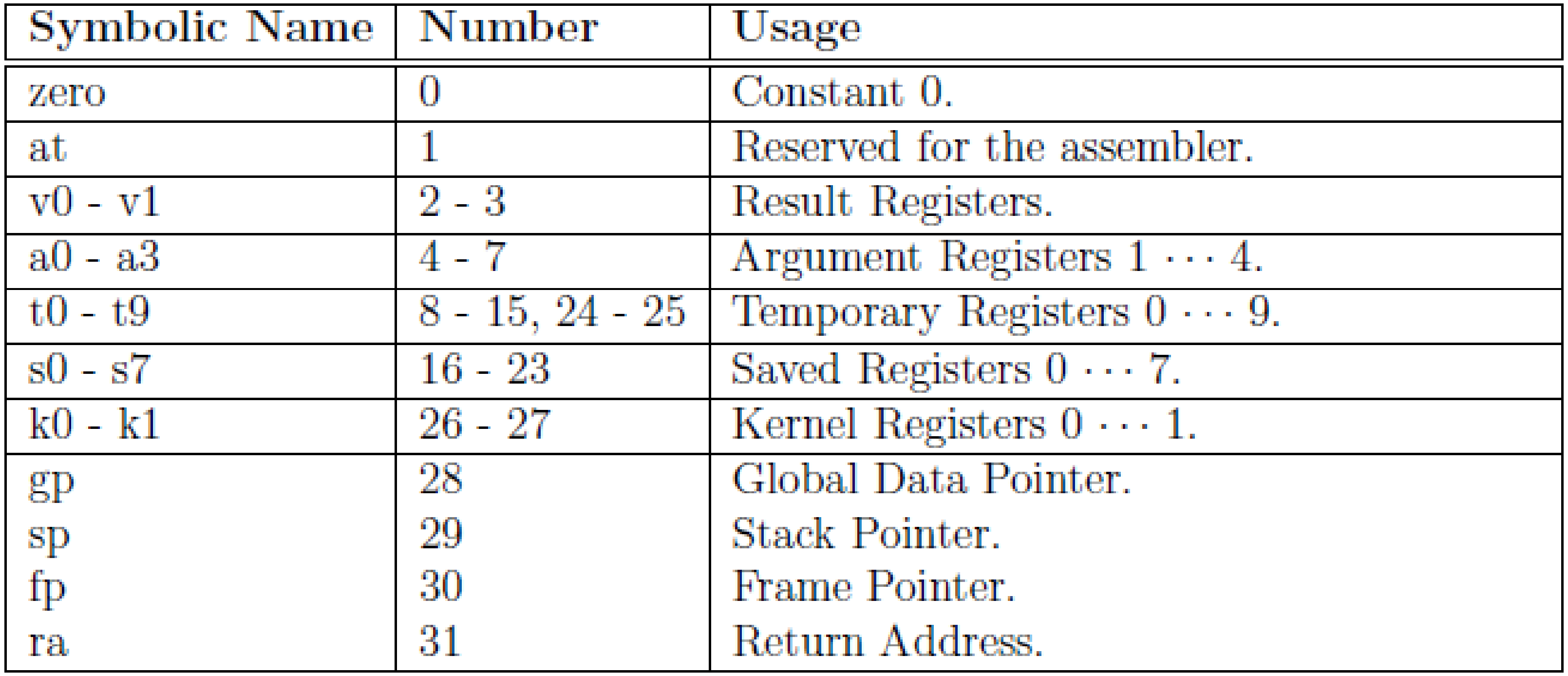
CS39001 COMPUTER ORGANIZATION AND ARCHITECTURE LAB

Debdeep Mukhopadhyay,
CSE, IIT Kharagpur

Procedures and Stack

MIPS General Purpose Registers

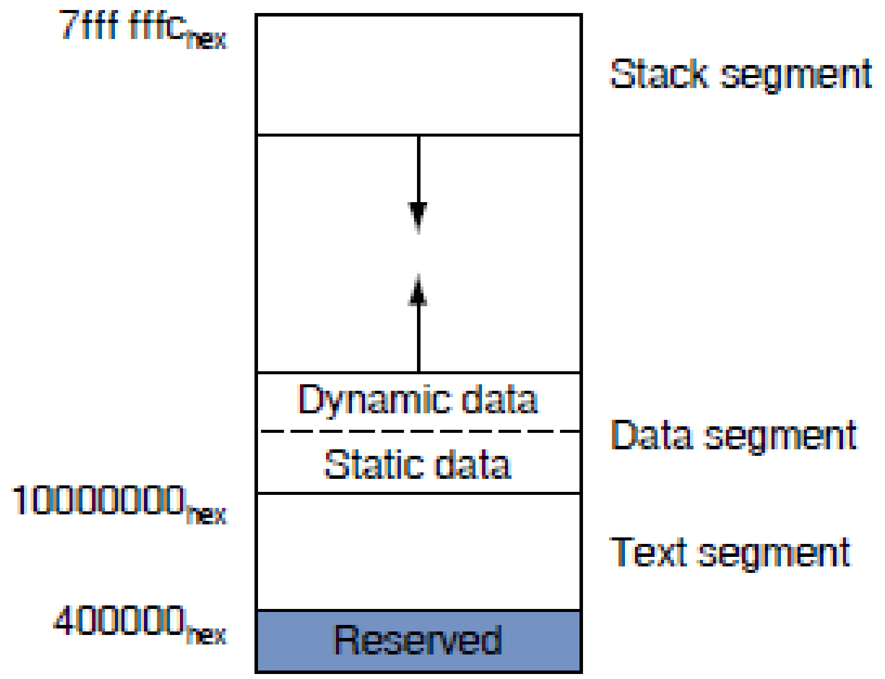


Symbolic Name	Number	Usage
zero	0	Constant 0.
at	1	Reserved for the assembler.
v0 - v1	2 - 3	Result Registers.
a0 - a3	4 - 7	Argument Registers 1 · · · 4.
t0 - t9	8 - 15, 24 - 25	Temporary Registers 0 · · · 9.
s0 - s7	16 - 23	Saved Registers 0 · · · 7.
k0 - k1	26 - 27	Kernel Registers 0 · · · 1.
gp	28	Global Data Pointer.
sp	29	Stack Pointer.
fp	30	Frame Pointer.
ra	31	Return Address.

System Calls

Service	System call Code (in $v0)	Arguments	Result
print_int	1	$a0=integer	
print_float	2	$f12=float	
Print_double	3	$f12=double	
Print_string	4	$a0=string address	
Read_int	5		Integer in $v0
Read_float	6		Float in $f0
Read_double	7		Double in $f0
Read_string	8	$a0=buffer address $a1=buffer size	
Sbrk	9		Address in $v0
exit	10		

Memory Layout



Procedures and Functions

- Receives a list of arguments.
- Performs computations based on these arguments.
- Functions usually return a single value.
- Procedures are also similar, but may return more than one value.

Types of parameter passing mechanisms

- Call by value
- Call by reference

Procedure Invocation in MIPS

- Two instructions:
  - jal (jump and link)
  - jr (jump register)
- The jal instruction is used to call a procedure.
- The jr instruction is used to return from a procedure.

The jal instruction

- Usage: jal proc_name

  transfer control to proc_name
- We need the return address to return from the called procedure.
  - Stored in $ra.
  - Is a J-type instruction

op=0000 11 **Jump Target Address**

31 26	25 **Offset** 0

The jal instruction (contd.)

- $ra=PC + 4
- Pseudo-direct addressing:
  - PC=PC[31:28]||offset << 2

Returning from a procedure

- jr $ra
  - reads the return address from the $ra register
  - returns the control to this address
- R-type instruction:

000000	11111	00000	00000	00000	
OpCode	Source register ($ra)	Unused	Unused	Unused	function jr=8

Parameter Passing

- Calling procedure first places all the parameters needed by the called procedure in mutually accessible storage area:
  - registers: register method, via \$a0, \$a1, \$a2, \$a3
  - memory: stack method
- In the register method, \$v0 and \$v1 are used to return results from the procedure.
- 4 parameters are passed by \$a0-3, remaining by stacks (spilled registers)

Leaf and non-leaf procedures

- Two major types of procedures.
- Leaf procedures do not call other procedures.
- Non-leaf procedures call other procedures.
- Leaf Procedures:
  - Simple leaf procedures may not need the stack if its local variables can fit in the caller saved registers $t0-9.
  - If not, the leaf procedure will have to use the stack.
- Non-leaf procedures:
  - Has to use the stack at least to store the return address.

Compute the sum of three integers

```
###Data Segment############
.data
number_prompt:
   .asciiz "Please enter three numbers: "
out_msg:
   .asciiz "\n The sum is: "
newline:
   .asciiz "\n"
```

Compute the sum of three integers

```
#####Code
    Segment############
  .text
  .globl main
main:
  la $a0,number_prompt
  li $v0,4
  syscall

  li $v0,5 #read 1st number into
    $a0
  syscall
  move $a0,$v0
```

```
li $v0,5 #read 2nd number into
  $a1
syscall
move $a1,$v0

li $v0,5 #read 3rd number into
  $a2
syscall
move $a2,$v0
```

Compute the sum of three integers

```
jal find_sum
   move $t0,$v0

   la $a0,out_msg
   li $v0,4
   syscall
```

```
move $a0,$t0
   li $v0,1
   syscall

   la $a0,newline
   li $v0,4
   syscall
exit:
    li $v0,10
    syscall
```

The procedure code

```
find_sum:
    move $v0,$a0
    addu $v0,$v0,$a1
    addu $v0,$v0,$a2
    #move $a0,$v0
    #li $v0,1
    #syscall
    #move $v0,$a0
    jr $ra
```

Consequences of the register method

- Pros:
  - easier and convenient for small number of variables.
  - faster, as operands are available in register than in memory.
- Cons:
  - A small number of arguments.
  - Registers are often used by the called procedure. So, it is needed to store these registers in the stack temporarily for free usage. Hence the above advantage of speed cannot be realized.

Stack implementation in MIPS

- LIFO (Last in First Out).
- MIPS memory layout has a memory from 0x10000000 to 0x7FFFFFFF is used for data and stack segments.
- Stack segment begins at the top end of the memory section (at 0x7FFFFFFF) and grows downward.
- The dynamic data grows area upward.
- This is an efficient way of using the unused area between the stack and the memory.
- Remember: When we push data into the stack, memory address decreases.

Stack Pointer

- Stack is a LIFO data structure.
- We need a pointer to the top of the stack: stack pointer.
- In Intel architectures, there is a special register called the esp register.
- For MIPS, r29, referred to as $sp, is used as the stack pointer.

PUSH and POP

- Push:

```
sub $sp, $sp, 4  #Decreasing
sw $a0, 0($sp) #sp creates space
```

- Pop:

```
lw $a0, 0($sp)
addu $sp, $sp, 4 #increase of sp means
                 #freeing 4 bytes of stack
```

Uses of the Stack

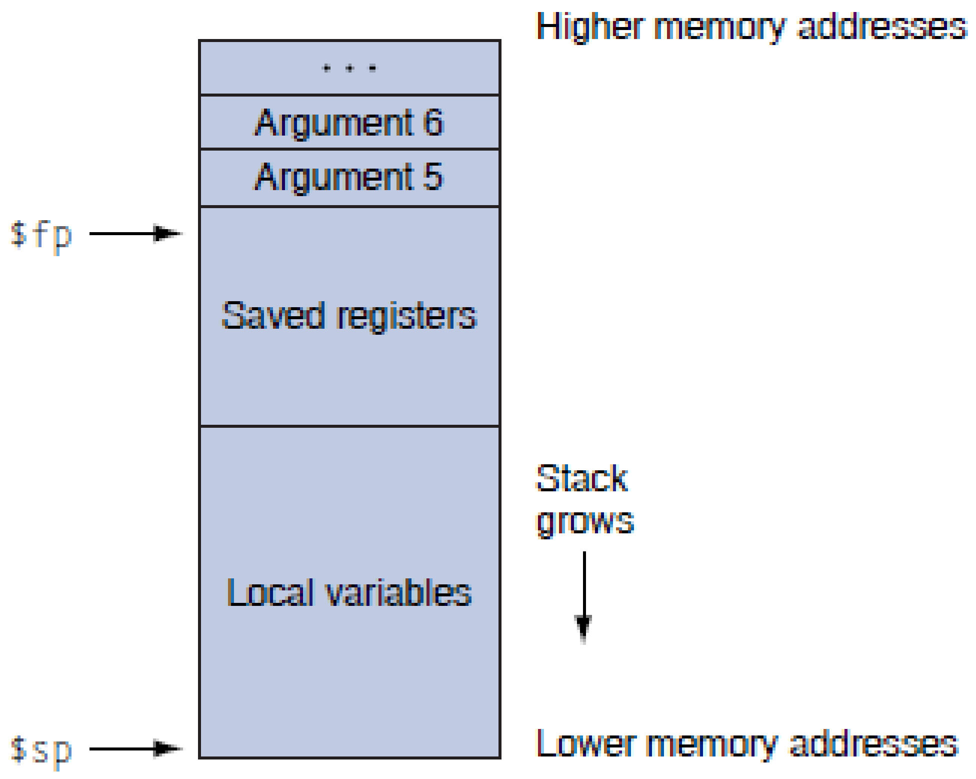
- Most of the bookkeeping associated with a call is centered around a block of memory called a **procedure call frame**.
- This memory is used for a variety of purposes:

  ■ To hold values passed to a procedure as arguments

  ■ To save registers that a procedure may modify, but which the procedure's caller does not want changed

  ■ To provide space for variables local to a procedure

Layout of a Stack Frame



Temporary storage of data

- Stack can be used as a scratch-pad to store data on temporary basis.
- There are some registers, called as the callee-saved, $s0-2, which the callee must save.

Usage:

```
test:
        sub $sp,$sp,12 #reserve 12 bytes of stack
        sw $s0,0($sp)
        sw $s1,4($sp)
        sw $s2,8($sp)
    #Now these registers are free for usage in the procedure
        lw $s0,0($sp)
        lw $s1,4($sp),
        lw $s2,8($sp)
        add $sp,$sp,12 #free space in stack
```

Transfer of Control

- Stack is used to transfer control.
- For non-leaf procedures, the return address ($ra) is stored in the stack.
- At the end, this is restored so that the control is transferred back to the caller properly.

An Example: Add 20 unsigned integers

```
###Data Segment###########
.data
number_prompt:
   .asciiz "Please enter the numbers: "
out_msg:
   .asciiz "\n The sum is: "
newline:
   .asciiz "\n"

#####Code Segment###########
   .text
   .globl main
main:
   la $a0,number_prompt
   li $v0,4
   syscall
```

An Example: Add 20 unsigned integers

```
li $t0,20
read_more:
  li $v0,5
  syscall
  sub $sp,$sp,4
  sw $v0,($sp)
  sub $t0,$t0,1
  bnez $t0,read_more

  jal find_sum
  move $t0,$v0
```

```
la $a0,out_msg
li $v0,4
syscall

move $a0,$t0
li $v0,1
syscall

la $a0,newline
li $v0,4
syscall
```

The function using stack

```
find_sum:
    li $v0,0
    li $t0,3
 sum_loop:
    lw $t1,($sp)
    add $sp,$sp,4
    add $v0,$v0,$t1
    sub $t0,$t0,1
    bnez $t0,sum_loop
    jr $ra
```

Preserving Calling Procedure States

- It is important to preserve the contents of the registers across a procedure call.
- Example:

```
        li $s0,50
    repeat:
        jal compute
        …
        sub $s0,$s0,1
        bnez $s0, repeat
```

The code is intended to read 50 numbers.

The $s0 maintains the number of remaining integers.

Suppose, $s0 is used by the procedure "compute"

Hence, this can lead to erroneous results.

Which registers should be saved and who will save?

- If the calling procedure saves, it needs to know the registers used by the called procedure.
- If the called procedure is modified, all the procedures that call them needs to be modified.
- Programs are longer. Each time a procedure is called one has to save and restore the registers.

So, the callee saves the registers.

- The called procedure is responsible for saving registers, and restoring them before returning to the calling procedure.
- But may be inefficient.
  - Assume a called procedure has 10 registers and none of them is used by the calling procedure.
  - Time and resource is wasted in saving all these registers.

Callee-saved and caller-saved

- Thus, MIPS divides the registers into two groups:
  - Caller saved: $t0-9. These registers can be over-written by the called procedure. It is the caller's onus to preserve them across procedure call.
  - Callee-saved: $s0-8. These registers, if used by the called procedure, must be preserved by the called procedure.

Fibonacci Sequence

```
###Data Segment###########
.data
number_prompt:
   .asciiz "Please enter a number n>0: "
error_msg:
   .asciiz "\n Invallid number! \n"
out_msg:
   .asciiz "Fib(n)= "
newline:
   .asciiz "\n"
```

Fibonacci Sequence

```
#####Code Segment###########
   .text
   .globl main
main:
   la $a0,number_prompt
   li $v0,4
   syscall

   li $v0,5
   syscall

   bgtz $v0,number_ok
   la $a0,error_msg
   li $v0,4
   syscall
   b main
```

```
number_ok:
   move $a0,$v0
   jal find_fib  #fib(n) stored in $v0
   move $t0,$v0

   la $a0,out_msg
   li $v0,4
   syscall

   move $a0,$t0
   li $v0,1
   syscall

exit:
    li $v0,10
    syscall
```

Fibonacci Sequence

```
find_fib:
#$v0 holds the current fib value
#$t0 holds the last fib value
#$t1 holds the next fib value
   li $v0,1 #if n=1 or 2
   ble $a0,2,fib_done

   li $t0,1

   loop:
      add $t1,$t0,$v0
        move $t0,$v0
        move $v0,$t1
        sub $a0,$a0,1
        bgt $a0,2,loop

fib_done:
   jr $ra
```

Non-leaf procedures need to save $ra

```
###Data Segment###########
.data
number_prompt:
  .asciiz "Please enter three numbers"
range_msg:
  .asciiz "\n The range is \n"
newline:
  .asciiz "\n"

#####Code Segment###########
  .text
  .globl main
main:
  la $a0,number_prompt
  li $v0,4
  syscall

  li $v0,5
  syscall
  move $a0,$v0
```

```
li $v0,5
  syscall
  move $a1,$v0

  li $v0,5
  syscall
  move $a2,$v0

  jal find_range
  move $t0,$v0

  la $a0,range_msg
  li $v0,4
  syscall

  move $a0,$t0
  li $v0,1
  syscall

  la $a0,newline
  li $v0,4
  syscall

exit:
   li $v0,10
   syscall
```

The procedures

```
##############################
find_range:

    sub $sp,$sp,4
    sw $ra,0($sp)

    jal find_min
    move $t0,$v0
    jal find_max
    sub $v0,$v0,$t0

    lw $ra,0($sp)
    add $s0,$sp,4
    jr $ra
##############################
```

```
find_min:
    move $v0,$a0
    ble $v0,$a1,min_a2
    move $v0,$a1
min_a2:
    ble $v0,$a2,minfound
    move $v0,$a2
minfound:
    jr $ra
##############################
find_max:
    move $v0,$a0
    bge $v0,$a1,max_a2
    move $v0,$a1
max_a2:
    bge $v0,$a2,maxfound
    move $v0,$a2
maxfound:
    jr $ra
```

Passing variable number of arguments

- Write a MIPS procedure to add a variable number of integers, ending the input if 0 is entered.

Example:

```
###Data Segment###########
.data
prompt:
   .asciiz "Please enter integers \n"
   .asciiz "Terminate inputs with 0\n"
sum_msg:
   .asciiz "\n The sum is \n"
newline:
   .asciiz "\n"
```

Program 1

```
#####Code Segment###########
   .text
   .globl main
main:
   la $a0,prompt
   li $v0,4
   syscall

   li $a0,0
readloop:
   li $v0,5
   syscall
   beqz $v0,exit_read
   subu $sp,$sp,4
   sw $v0,($sp)
   addu $a0,$a0,1
   b readloop
exit_read:
   jal sum
   move $t0,$v0

   la $a0,sum_msg
   li $v0,4
   syscall

   move $a0,$t0
   li $v0,1
   syscall

   la $a0,newline
   li $v0,4
   syscall

exit:
    li $v0,10
    syscall
```

Program 2

```
##########################
sum:
   li $v0,0

sum_loop:
   beqz $a0,done
   lw $t0,($sp)
   addu $sp,$sp,4
   addu $v0,$v0,$t0
   subu $a0,$a0,1
   b sum_loop
done:
   jr $ra
```

Passing an array

```
###Data Segment###########
.data
number_prompt:
   .asciiz "Please enter the numbers: "
sum_msg:
   .asciiz "\n The sum is: "
newline:
   .asciiz "\n"
.align 2
numbers:
   .space  40
```

```
#####Code Segment###########
   .text
   .globl main
main:
   la $a0,number_prompt
   li $v0,4
   syscall


   la $t0,numbers
   li  $t1,10

   read_loop:
         li $v0,5
         syscall
         sw $v0,0($t0)
          addu $t0,$t0,4
         subu $t1,$t1,1
          beqz $v0,exit_loop
          bnez $t1,read_loop
```

```
exit_loop:
          la $a0,numbers
          li $a1,9
          subu $a1,$a1,$t1
          jal sumarray

          move $s0,$v0

      la $a0,sum_msg
          li $v0,4
          syscall

          move $a0,$s0
          li $v0,1
          syscall

  li $v0,10
  syscall
```

```
#####################
  sumarray:
      li $v0,0
  loop:
     beqz $a1,exit_add_loop
     lw $t0,($a0)
     add $v0,$v0,$t0
     addu $a0,$a0,4
     subu $a1,$a1,1
     b loop
  exit_add_loop:
     jr $ra
```