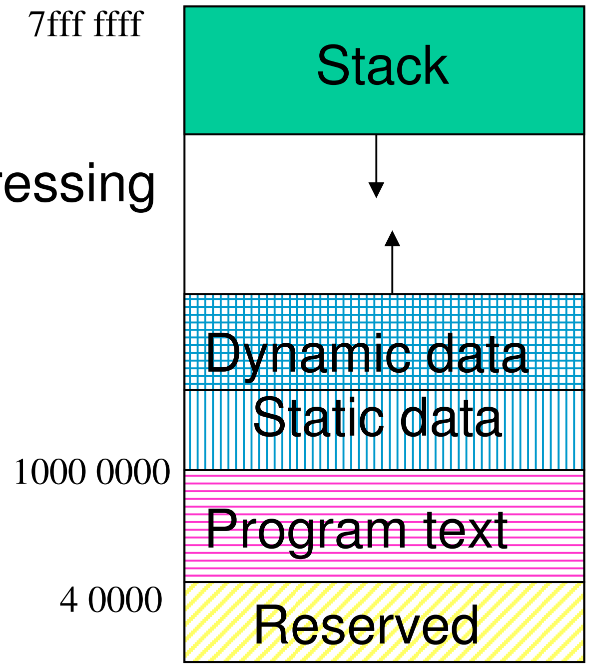
Program and memory layout

- By convention the layout is:
  - Note that only half of the addressing space is taken by user
    Other half is O.S.



Procedures

- Procedures/functions are the major program structuring mechanism
- Calling and returning form a procedure requires a protocol between *caller* and *callee*
- Protocol is based on conventions

Procedures/Functions -- Protocol

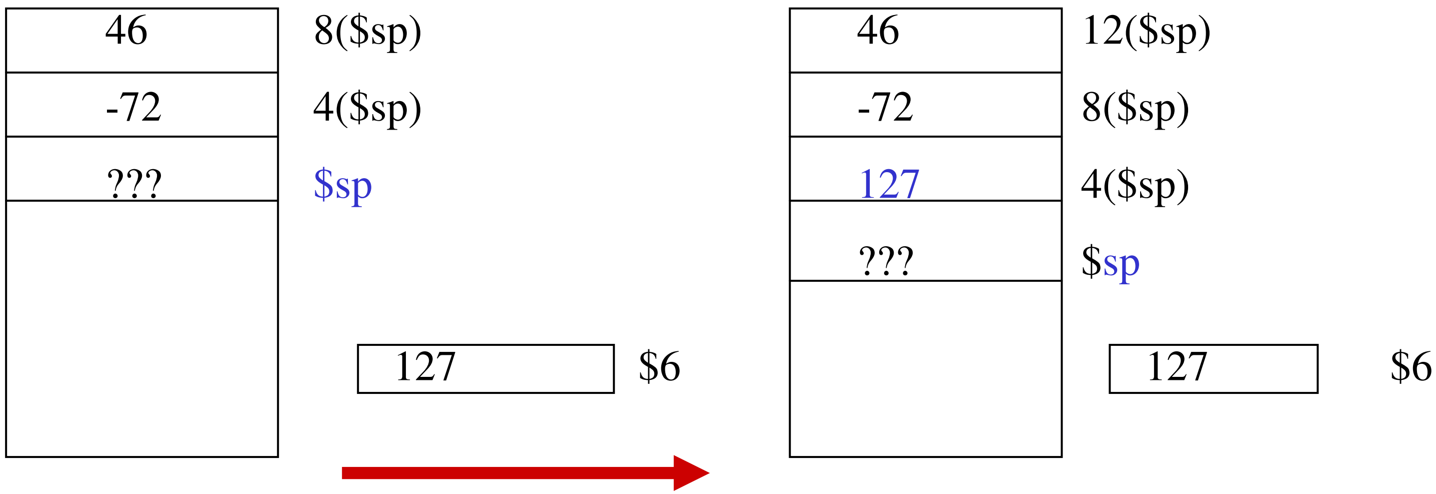
- Each machine (compiler?) has its own set of protocol(s)
- Protocol: combination of hardware/software
  – e.g., “jal” is hardware
  – use of register $29 as $sp is software
- Protocol: sequence of steps to be followed at each call and each return
  – controlled by hardware and/or software
- In RISC machines
  – hardware performs simple instructions
  – software (compiler/assembler) controls sequence of instructions

Program stack

- Each executing program (process) has a *stack*
- Stack = dynamic data structure accessed in a LIFO manner
- Program stack automatically allocated by O.S.
- At the start of the program, register $sp ($29 in MIPS) is automatically loaded to point to the first empty slot on top of stack
  - After that it will be your responsibility to manage $sp
- By convention, stack grows towards lower addresses
  - to allocate new space (i.e., when you *push*), decrement $sp
  - to free space on top of stack (*pop*), increment $sp

Push operation

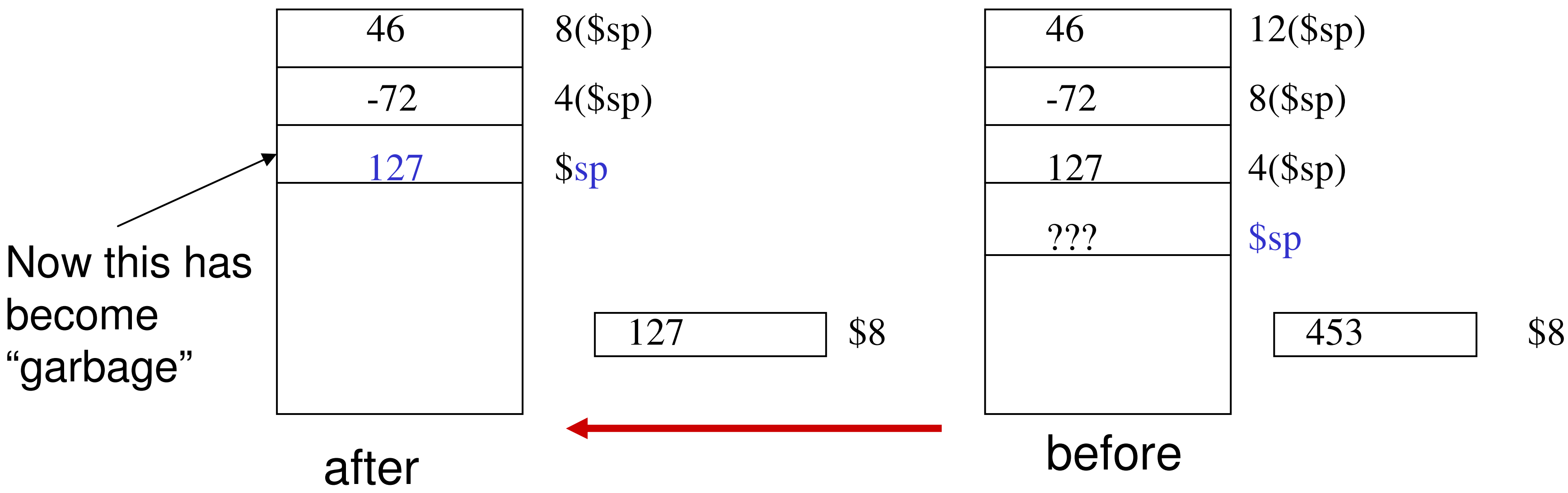
- *push* adds an item on top of stack
  - one instruction to manipulate the data, e.g. “sw $6,0($sp)”
  - one instruction to adjust the stack pointer e.g., “subu $sp,$sp,4”



before after

Pop operation

- *pop* removes the item on top of stack and stores it in a register
  - one instruction to adjust the stack pointer e.g., "addu $sp,$sp,4"
  - one instruction to manipulate the data, e.g. "lw $8,0($sp)"



Procedure call requirements (caller/callee)

- Caller must pass the return address to the callee
- Caller must pass the parameters to the callee
- Caller must save what is in *volatile* (registers) that could be used by the callee
- Callee must save the return address (in case it becomes a caller)
- Callee must provide (stack) storage for its own use
- Caller/callee should support recursive calls

Mechanism

- Registers are used for
  - passing return address in $ra

    jal   target
  - passing a small number of parameters (up to 4 in $a0 to $a3)
  - keeping track of the stack ($sp)
  - returning function values (in $v0 and $v1)
- Stack is used for
  - saving registers to be used by callee
  - saving info about the caller (return address)
  - passing parameters if needed
  - allocating local data for the called procedure

Procedure calls and register conventions

Register	Name	Function	Comment
$0	Zero	Always 0	No-op on write
$1	$at	Reserved for assembler	Don't use it
$2-3	$v0-v1	Expr. Eval/funct. Return	
$4-7	$a0-a3	Proc./func. Call parameters	
$8-15	$t0-t7	Temporaries; volatile	Not saved on proc. Calls
$16-23	$s0-s7	Temporaries	Should be saved on calls
$24-25	$t8-t9	Temporaries; volatile	Not saved on proc. Calls
$26-27	$k0-k1	Reserved for O.S.	Don't use them
$28	$gp	Pointer to global static memory	
$29	$sp	Stack pointer	
$30	$fp	Frame pointer	
$31	$ra	Proc./funct return address	

Who does what on a call (one sample protocol)

- Caller
  - Saves any volatile register (\$t0-\$t9) having contents that need to be kept
  - Puts up to 4 arguments in \$a0-\$a3
  - If more than 4 arguments, pushes the rest on the stack
  - calls with jal instruction
- Callee
  - saves \$ra on stack
  - saves any non-volatile register (\$s0-s7) that it will use

Who does what on return

- Callee
  - restores any non-volatile register ($s0-$s7) it has used
  - restores $ra
  - puts function results in $v0-$v1
  - adjusts $sp
  - returns to caller with “jr $ra”
- Caller
  - restores any volatile register it had saved
  - examines $v0-$v1 if needed

Example of a call sequence

- Assume 2 arguments in $t0 and $t3 and we want to save the contents of $t6 and $t7

```
move    $a0,$t0       #1st argument in $a0
move    $a1,$t3       #2nd argument in $a1
subu    $sp,$sp,8     #room for 2 temps on stack
sw      $t6,8($sp)    #save $t6 on stack
sw      $t7,4($sp)    #save $t7 on stack
jal     target
```

- Assume the callee does not need to save registers

```
target:  sw   $ra,0($sp)    #save return address
         subu $sp,$sp,4     # on stack
```

Return from the previous sequence

- The callee will have put the function results in $v0-$v1

```
addu    $sp,$sp,4       #pop
lw      $ra,0($sp)      #return address in $ra
jr      $ra             #to caller
```

- The caller will restore $t6 and $t7 and adjust stack

```
lw      $t6,8($sp)
lw      $t7,4($sp)
addu    $sp,$sp,8
```

Factorial, The Logic

- Logic of call:
  - Put argument in register a0 — li $a0,5
  - Call procedure — jal fact
  - Grab result from v0 — sw $v0, …
- Logic of Operation
  - Make a place on the stack
  - Save the argument and return address
  - Decrement the argument
  - Check if recursion is done
  - If not, recurse again
  - If so, compute the result into $v0
  - Grab return address
  - Restore stack
  - Return

$v0 = $a0*($a0-1)!

Factorial, The Code

```
fact:   subu $sp,$sp,8        #create 2 spaces on stack
        sw   $ra, 8($sp)      #push return address
        sw   $a0,4($sp)       #push argument
        addi $a0,$a0,-1       #decrement argument
        beqz $a0,base         #is recursion done?
        jal  fact             #recurse
        j    done             #v0 contains a0!
base:   li   $v0,1            #initialize v0
done:   lw   $a0,4($sp)       #get argument back
        mult $v0,$a0          #figure next term
        mflo $v0              #grab result & ready for return
        lw   $ra,8($sp)       #restore the return address
        addiu $sp,$sp,8       #pop
        jr   $ra              #return
```

$v0 = $a0*($a0-1)!