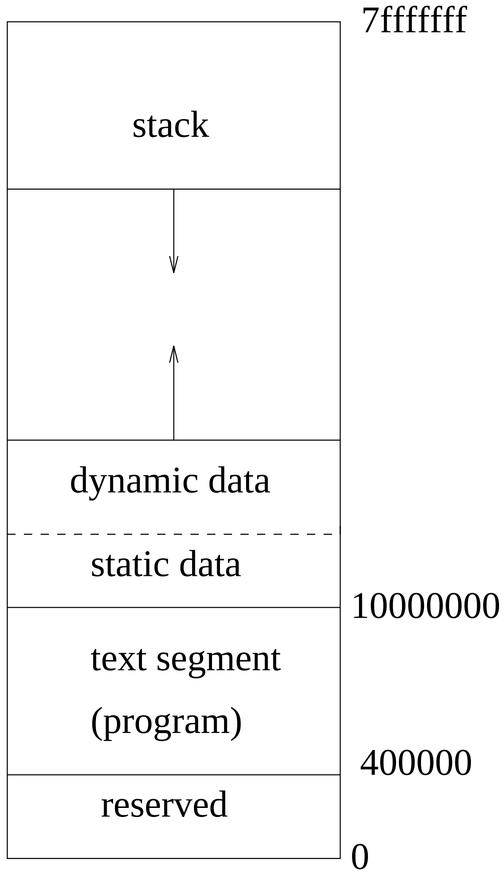
Program and memory data layout.

By convention, the layout is as follows:



Note that only half of the address space is taken by the "user".

The other half is for the O.S.

Procedures

Need some agreed upon protocol between caller and callee.

Each machine (compiler?) has its own set of protocol(s)

Protocol is usually a combination of hardware and software conventions

Protocol is a sequence of steps that have to be followed at each call (and return); this sequence is controlled by hardware and/or software

In RISC machines, the hardware performs simple instructions; the compiler/assembler programmer controls the sequence of instructions.

Program stack

Each executing program (process) has a stack

A stack is a dynamic data structure that is accessed in a *Last-In-First-Out* manner (LIFO)

The program stack is automatically allocated by the O.S.

Register $sp (register 29 on the MIPS) is automatically loaded to point to the first empty slot on top of the stack

By convention the stack grows towards *lower* addresses so BEWARE

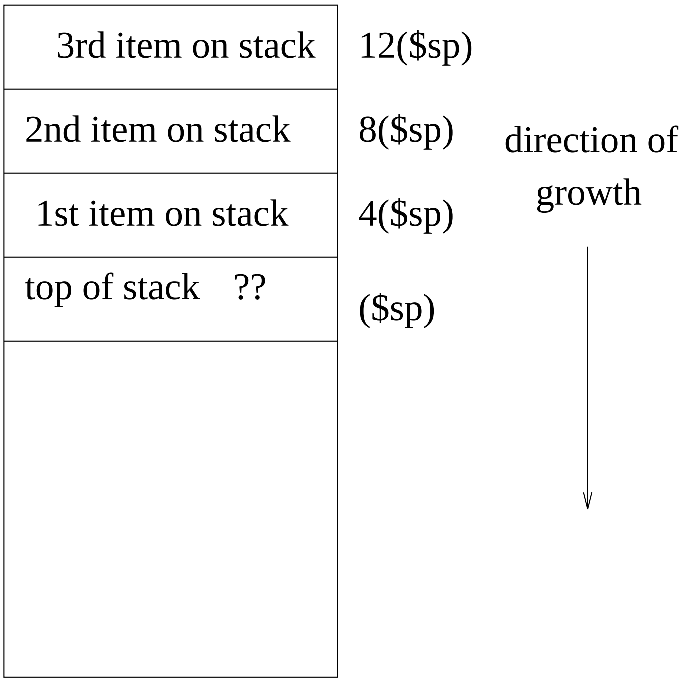
- to allocate new space on the stack, decrement $sp
- to free old space on the stack, increment $sp

Operations on a stack

For our purposes, we will use two operations on a stack

1. *push* adds an item on top of the stack
2. *pop* removes the item on top of stack and stores it

Example:



Operations on a stack - Push and Pop

Push and Pop require two instructions each:

- one to adjust the stack pointer
- one to manipulate the data

Example: Push the contents of register r6 on the stack

Opcode	Operands	Comments
sw	$6,0($sp)	#push contents of r6
subu	$sp,$sp,4	#allocate new word on stack

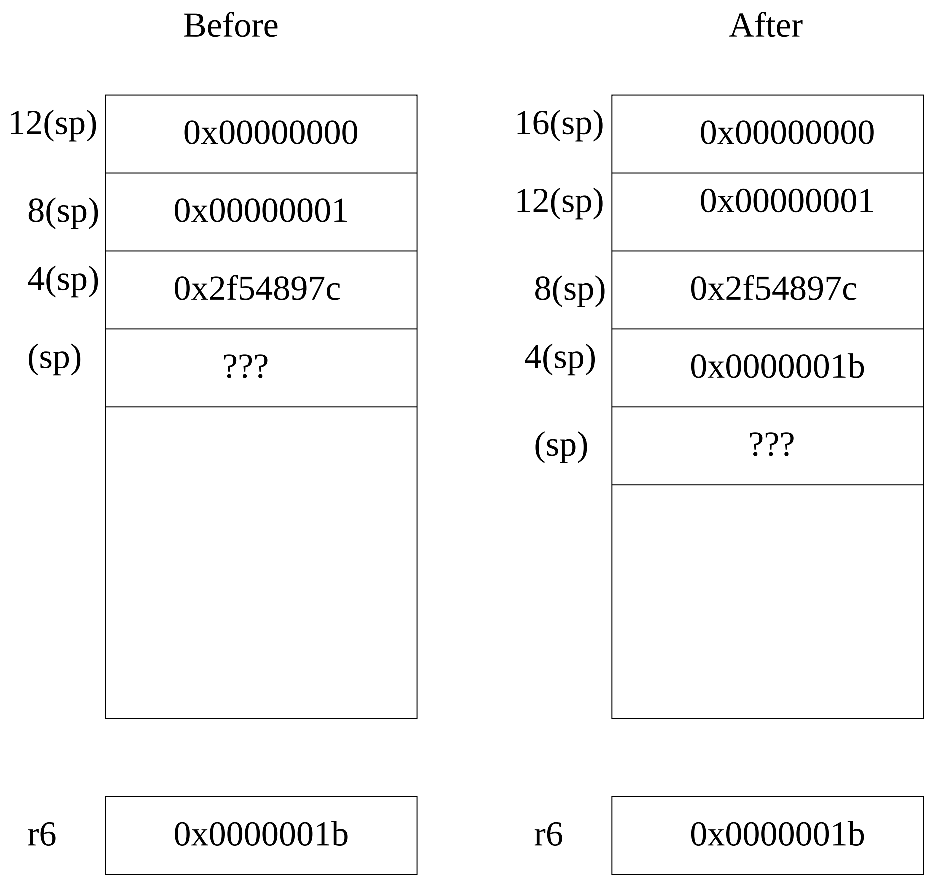
Example: Pop top of stack and put it in register r6

Opcode	Operands	Comments
addu	$sp,$sp,4	#adjust stack pointer
lw	$6,0($sp)	#pop contents in r6

Note 1. The way the "addu" and "subu" are written does not correspond to the 3 register format but the assembler understands and transforms them in "addui".

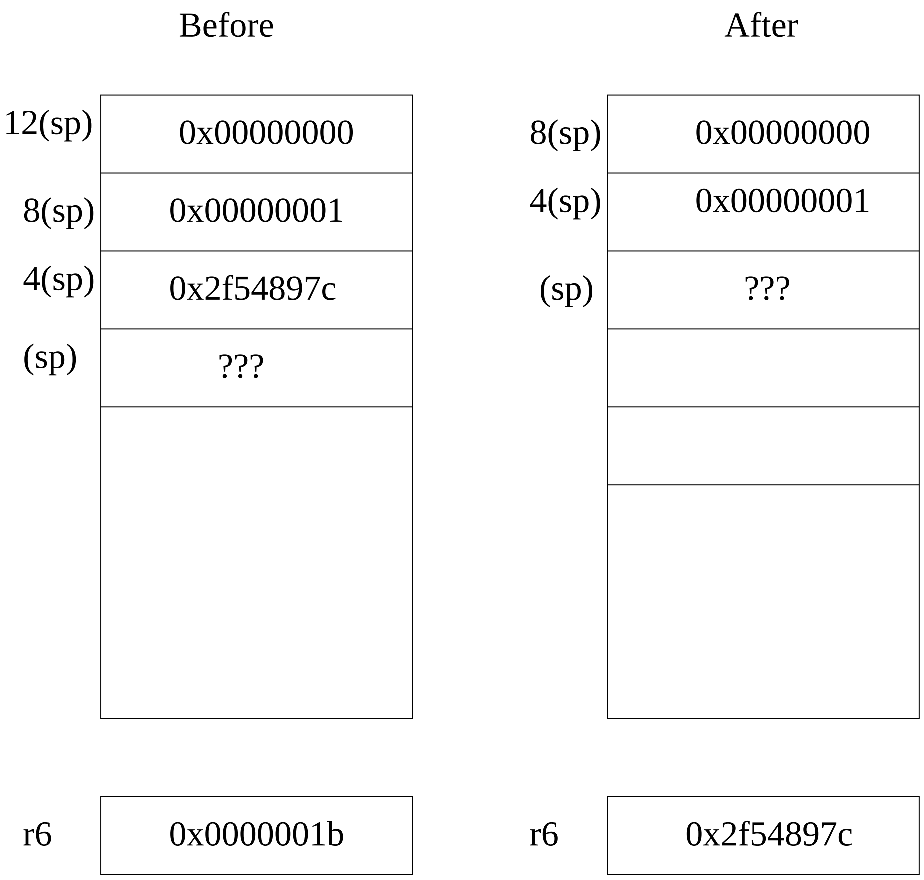
Push Example

Example: Push the contents of register r6 on the stack



Pop Example

Example: Pop top of stack and put it in register r6



Procedures

Procedures (functions – subroutines) are the major program structuring mechanisms

Calling and returning from a procedure requires a protocol

The protocol is a sequence of conventions, each of which takes a very small number of instructions.

Procedure Call Requirements (caller/callee)

1. Must pass to the callee the address to where to go back (caller)
2. Must pass parameters from caller to callee (caller)
3. Must save whatever is volatile and could be used by the callee (registers) (caller)
4. Must save its own return address (callee)
5. Must provide local storage for its own use (callee)
6. Should support recursive calls (caller/callee)

Mechanisms

Use of registers and use of the stack.

Registers are used for:

- passing the return address (jal $ra, target)
- passing a small number of parameters (up to 4, $a0 to $a3)
- keep track of stack ($sp)
- return function values (in $vo and $v1)

Stack is used for:

- save registers to be used by callee
- save info about the caller (its return address)
- pass parameters if needed
- allocate local data for the called procedure

Procedure calls and register conventions

Register	Name	Function
$2,3	$v0-v1	return values for functions
$4-7	$a0-a3	the first 4 parameters
$8-15	$t0-t7	temporaries;volatile;not saved on proc. call
$24-25	$t8-t9	
$16-23	$s0-s7	temporaries;should be saved on calls
$28	$gp	pointer to global static memory
$29	$sp	stack pointer
$30	$fp	frame pointer
$31	$ra	return address

Who does what on a call?

Depends on convention. For example:

Caller

1. Saves any volatile register ($t0-t9) whose contents it wants to keep
2. Puts up to four arguments in $a0-a3
3. If more than 4 arguments pushes them on the stack
4. Calls the callee with "JAL target" instruction

Callee

1. Saves $ra (for return address)
2. Saves any non-volatile temporary that it will use ($s0-s7)

Who does what on a return

Callee

1. Restores any non-volatile temporary that it has used (s0-s7)
2. Restores \$ra
3. For a function puts result in \$v0-v1
4. Adjusts \$sp
5. Returns to the caller with a "JR   \$ra" instruction

Caller

1. Restores any volatile register (\$t0-t9) whose contents it needs
2. If it had called a function, examines \$v0-v1 if needed

Example of a call sequence

Assume that we have 2 arguments in registers $t0 and $t3 and that we want to save $r14 and $r15 ($t6-t7).

```
        move  $a0, $t0       # first argument in $a0
        move  $a1, $t3       # second argument in $a1
        subu  $sp,$sp,8      # adjust stack pointer
        sw    $t6,8($sp)     # save $t6 on stack
        sw    $t7,4($sp)     # save $t7 on stack
        jal   targ           # call function
```

The first thing the callee will do is (assume that it does not have to save any $s register):

```
targ:   sw    $ra,0($sp)     # save return address
        subu  $sp,$sp,4      # adjust stack pointer
```

Return from example call sequence

Before returning the callee will have put results in \$v0-v1 if needed and then:

```
addu  $sp,$sp,4        # adjust stack pointer
lw    $ra,0($sp)       # Load return address in $ra
jr    $ra              # return
```

The caller will restore \$t6-t7 and adjust the stack appropriately

```
lw    $t6,8($sp)       # restore $t6
lw    $t7,4($sp)       # restore $t7
addu  $sp,$sp,8        # adjust stack pointer
```