Control structures practice problems

- branches and jumps are used to implement
  - conditional statements
  - loops
  - functions

More about functions

Leaf and non-leaf functions/procedures

- a leaf function does not call any other function or itself
- a non-leaf functions calls other functions or itself

If we call a function from within a function, $ra will get overwritten. Problem!

Solution: save $ra on the stack

We can also use the stack to store any variables we want.

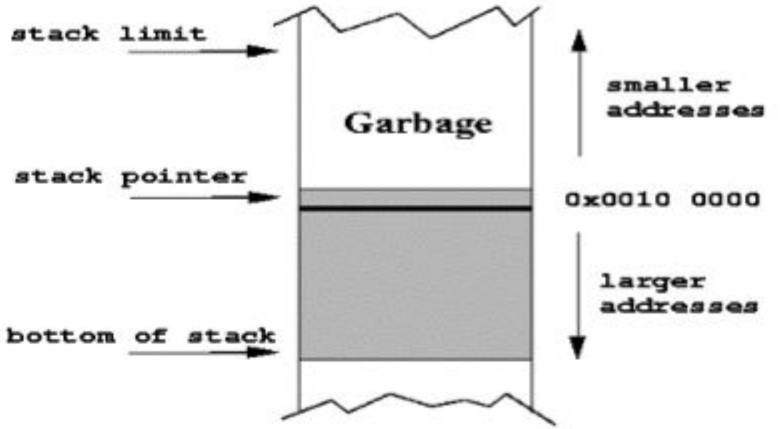
the stack

the stack is a contiguous section of memory

the stack pointer ($sp) points to the current top of the stack

when the stack is initialized, $sp points to the bottom of available stack memory

the stack grows upward



push and pop

PUSH copies a register to the stack

- used to save data on the stack

POP copies a value from the stack to a register

- used to retrieve data from the stack

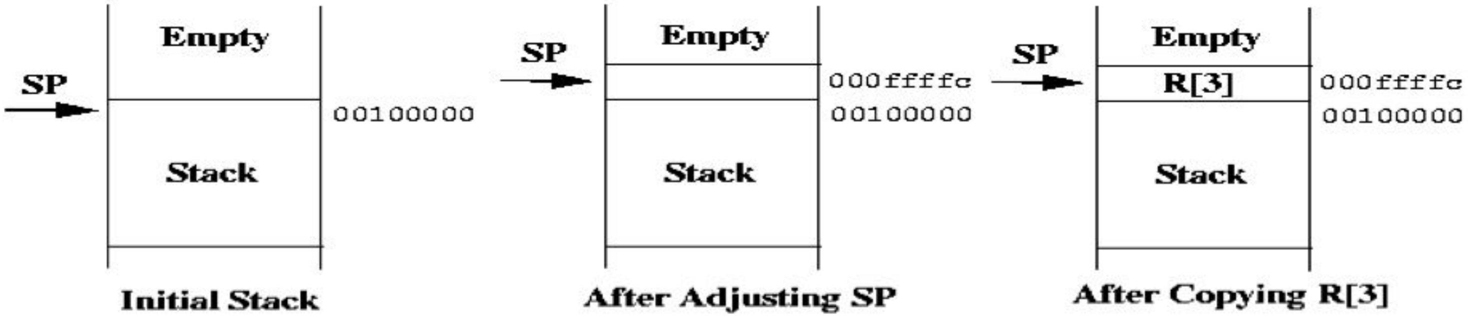
Many ISAs have PUSH and POP instructions

MIPS uses load and store instructions

push

This is how we push:

```
addi $sp, $sp, -4

sw $s3, ($sp)
```

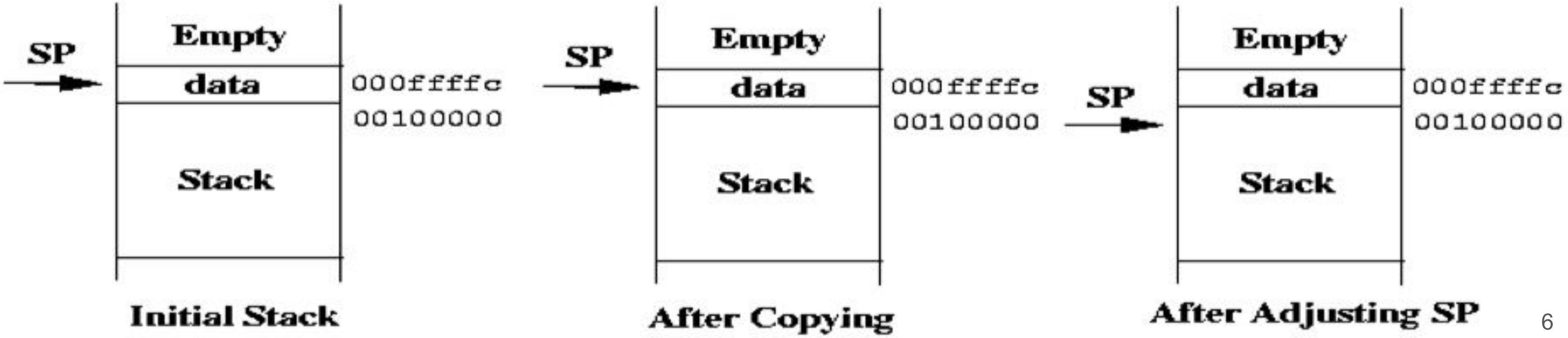


pop

This is how we pop:

```
lw $s3, ($sp)

addi $sp, $sp, 4
```

Notice that the data is still there in the now unused portion of the stack.



stacks and functions

In some architectures, each time a function is called, the entire state of the machine (registers) is stored on the stack

If you have recursive calls, you can run out of stack space - stack overflow!

MIPS reduces the likelihood of this by providing register conventions of what is saved across function calls

You need to use the stack in MIPS for non-leaf functions and if you want to pass in more than 4 arguments or pass out more than 2 arguments

leaf function example

Args in $a registers

f in $s0;

result in $v0

C Code:

```
int leaf (int g, h, i, j)
{
  int f;
  f = (g + h) - (i + j)
  return f;
}
```

```
leaf_example:
    addi  $sp, $sp, -4
    sw    $s0, 0($sp)        # Save $s0 on stack
    add   $t0, $a0, $a1
    add   $t1, $a2, $a3      # Procedure body
    sub   $s0, $t0, $t1
    add   $v0, $s0, $zero    # Result
    lw    $s0, 0($sp)
    addi  $sp, $sp, 4        # Restore $s0
    jr    $ra                # Return
```

```
# leaf function with arguments and push/pop
# function_leaf_args.asm
# purpose: demo how to use a and v registers, how to use stack

        .data
f:      .word   0
g:      .word   5
h:      .word   12
i:      .word   4
j:      .word   3

        .text
        lw      $a0, g          # load arguments
        lw      $a1, h
        lw      $a2, i
        lw      $a3, j

        li      $s0, -1         # put something in $s0 for demo purposes
        # put something in $t0 and $t1 to demonstrate that
        #    t registers are not preserved in a function call
        li      $t0, 9
        li      $t1, 9

        jal     calc
        sw      $v0, f          # save result from function call

exit:   li      $v0, 10
        syscall
```

```
calc:   # return (g + h) - (i + j)
        # g..j are in $a0..$a3
        # result is returned in $v0

        # push $s0 on stack
        addi    $sp, $sp, -4
        sw      $s0, ($sp)

        add     $t0, $a0, $a1
        add     $t1, $a2, $a3
        sub     $s0, $t0, $t1    # we could have used
        add     $v0, $s0, $zero

        # pop $s0 from stack
        lw      $s0, ($sp)
        addi    $sp, $sp, 4
        # return
        jr      $ra
```

saving multiple registers on the stack

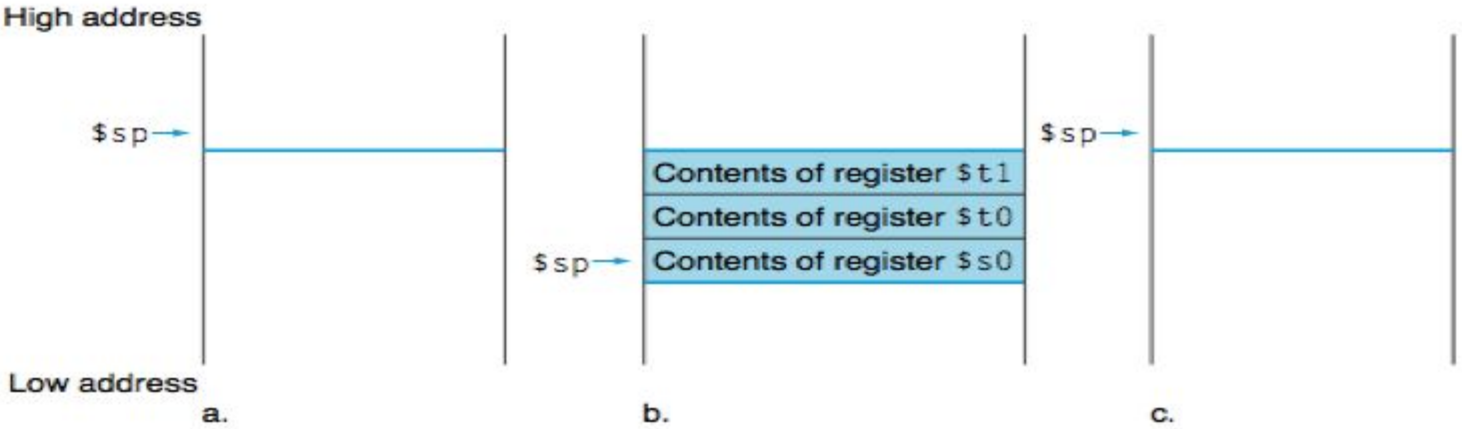


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

$t and $s registers

The following convention is used in MIPS:

- $t0 - $t9 are temporary registers that the called function is not required to save
- $s0 - $s7 are saved registers that the called function must preserve them by saving/restoring on the stack

This is a convention:

- not enforced by the assembler
- agreed upon by programmers to make code easier to read

byte operations

lb   rt, offset(rs)   # load a byte from address rs+offset to rt (sign extends)

lbu rt, offset(rs) # load a byte from address rs+offset to rt (no sign extend)

sb rt, offset(rs) # store register rt at address rs+offset

string copy example

C code:

```
void strcpy (char x[], char y[])
{
  int i = 0;
  while ((x[i]=y[i] != '\0')
    i ++ 1;
}
```

MIPS code:

```
strcpy:
     addi $sp, $sp, -4          # adjust stack for 1 item
     sw   $s0, 0($sp)           # save $s0
     add  $s0, $zero, $zero     # i = 0
L1:  add  $t1, $s0, $a1         # addr of y[i] in $t1
     lbu  $t2, 0($t1)           # $t2 = y[i]
     add  $t3, $s0, $a0         # addr of x[i] in $t3
     sb   $t2, 0($t3)           # x[i] = y[i]
     beq  $t2, $zero, L2        # exit loop if y[i] == 0
     addi $s0, $s0, 1           # i = i + 1
     j    L1                    # next iteration of loop
L2:  lw   $s0, 0($sp)           # restore saved $s0
     addi $sp, $sp, 4           # pop 1 item from stack
     jr   $ra                   # and return
```

```
# string copy example
# C equivalent - copies x to y
#         void strcpy (char x[], char y[]) {
#              int i = 0;
#              while ((y[i] = x[i]) != '\0\)
#                   i += 1;
#         }

.data

string1:          .asciiz "Hello"
string2:          .byte   6

.text

main:
          add      $s0, $zero, $zero                # s0 = i
          la       $s1, string1
          la       $s2, string2
L1:       add      $t1, $s1, $s0   # t1 = address of x[i]
          lbu      $t2, 0($t1)     # t2 = x[i]
          add      $t3, $s2, $s0   # t3 = address of y[i]
          sb       $t2, 0($t3)     # y[i] = x[i]
          beq      $t2, $zero, exit
          addi     $s0, $s0, 1
          j        L1

exit:
          li       $v0, 10
          syscall

```

form of a function

```
procedure:
              addi $sp, $sp, -12
Prologue  {   sw $ra, 8($sp)
              sw $s0, 4($sp)
              sw $s1, 0($sp)
              addi $s0, $zero, 50
    Body  {   addi $s0, $zero, -23
              add $a0, $s0, $s1
              jal procedure2
              lw $s1, 0($sp)
Epilogue  {   lw $s0, 4($sp)
              lw $ra, 8($sp)
              addi $sp, $sp, 12
              jr $ra
```

non-leaf functions

C code:

```
int fact (int n)
{
  if (n < 1) return f;
  else return n * fact(n-1);
}
```

MIPS code:

```
fact:
    addi $sp, $sp, -8     # adjust stack for 2 items
    sw   $ra, 4($sp)      # save return address
    sw   $a0, 0($sp)      # save argument
    slti $t0, $a0, 1      # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1    # if so, result is 1
    addi $sp, $sp, 8      #   pop 2 items from stack
    jr   $ra              #   and return
L1: addi $a0, $a0, -1     # else decrement n
    jal  fact             # recursive call
    lw   $a0, 0($sp)      # restore original n
    lw   $ra, 4($sp)      #   and return address
    addi $sp, $sp, 8      # pop 2 items from stack
    mul  $v0, $a0, $v0    # multiply to get result
    jr   $ra              # and return
```

```
# nonleaf example 1
# find factoral n!
# C implementation
#       int fact (int n) {
#               if (n < 1) return (1);
#               else return (n * fact(n-1);
#       }

.data
n:      .word   3
nfact:  .word   0

.text
main:   # load the data
        lw      $s0, nfact
        lw      $s1, n

        add     $a0, $s1, $zero  # copy data to arguments

        jal     fact    # call subroutine to find factorial
        sw      $v0, nfact

        j       exit

fact:   # subroutine to find the factorial of n
```

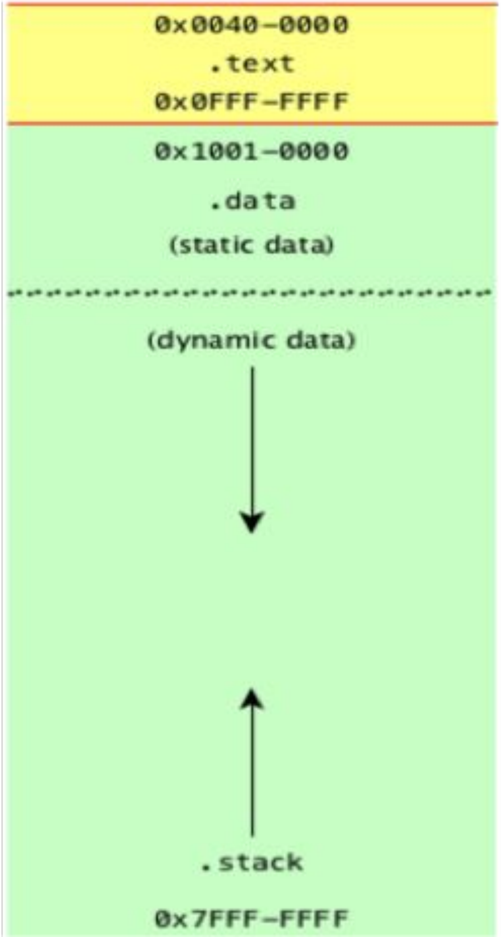
```
fact:   # subroutine to find the factorial of n

        # save registers on the stack:  $a0 and $ra
        addi    $sp, $sp, -8
        sw      $ra, 4($sp)
        sw      $a0, 0($sp)

        # factorial
        slti    $t0, $a0, 1      # test for n < 1
        beq     $t0, $zero, L1

        # base case n =0
        addi    $v0, $zero, 1    # return 1
        addi    $sp, $sp, 8      # pop 2 items off stack
        jr      $ra              # return to called

L1:     addi    $a0, $a0, -1     # decrement n
        jal     fact             # call fact again

        # restore variables off the stack: pop
        lw      $a0, 0($sp)      # restore n
        lw      $ra, 4($sp)      # restore return address
        addi    $sp, $sp, 8      # adjust sp

        # multiply and exit
        mul     $v0, $a0, $v0    # return n * fact(n-1)
        jr      $ra
        # end of calc

exit:
        li  $v0, 10
        syscall
```

MIPS static and dynamic variables

Data in the .data section is static data.

We can also allocate data at run time on the heap.

Notice that the stack and the heap grow towards each other.



memory management

In C, programmers allocate dynamic memory with malloc( ) and free it with the free( ) function.

A memory leak occurs if a program does not release the bytes on the heap when it is finished with it.

Another source of errors is releasing the memory before the program is finished with them.

These types of problems led to the development of Java and C++

arrays v. pointers: clearing an array

`clear1(int array[], int size) {`
`  int i;`
`  for (i = 0; i < size; i += 1)`
`    array[i] = 0;`
`}`
	`clear2(int *array, int size) {`
	`  int *p;`
	`  for (p = &array[0]; p < &array[size];`
	`       p = p + 1)`
	`    *p = 0;`
	`}`

`       move $t0,$zero    # i = 0`
`loop1: sll $t1,$t0,2      # $t1 = i * 4`
`       add $t2,$a0,$t1    # $t2 =`
`                          #   &array[i]`
`       sw $zero, 0($t2)   # array[i] = 0`
`       addi $t0,$t0,1     # i = i + 1`
`       slt $t3,$t0,$a1    # $t3 =`
`                          #   (i < size)`
`       bne $t3,$zero,loop1 # if (…)`
`                           # goto loop1`
	`       move $t0,$a0      # p = & array[0]`
	`       sll $t1,$a1,2     # $t1 = size * 4`
	`       add $t2,$a0,$t1   # $t2 =`
	`                         #   &array[size]`
	`loop2: sw $zero,0($t0)   # Memory[p] = 0`
	`       addi $t0,$t0,4    # p = p + 4`
	`       slt $t3,$t0,$t2   # $t3 =`
	`                         #(p<&array[size])`
	`       bne $t3,$zero,loop2 # if (…)`
	`                           # goto loop2`

arrays v. pointers

Array indexing involves:

- multiplying index by element size
- added to array base address

Pointers correspond directly to memory addresses.

Can avoid indexing complexity