Chapter 4:

Functions

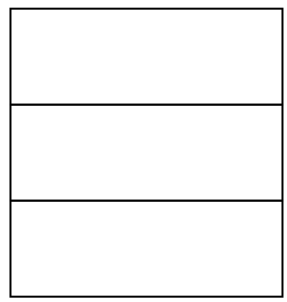
Topics:

The stack
Function call and return
Parameter handling
Register use conventions

Reading: Patterson and Hennessy
2.8, 2.13

The stack

a last-in-first-out (LIFO) data structure



Only the top item in the stack is “visible”.
Push: add an item to top of stack
Pop: remove an item from top of stack

Stack in memory

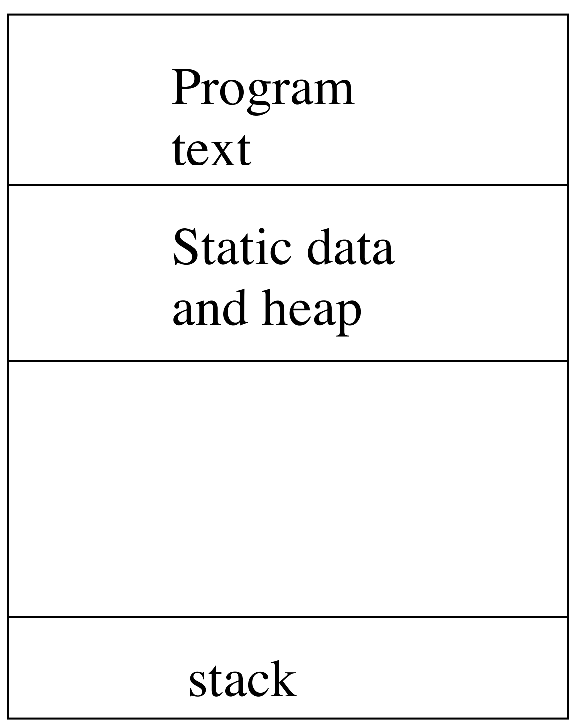
stack: array of 32-bit words

$29 or $sp: stack pointer
* contains address of the top item in stack

Stacks in most machines grow upward,
from high addresses to low addresses.

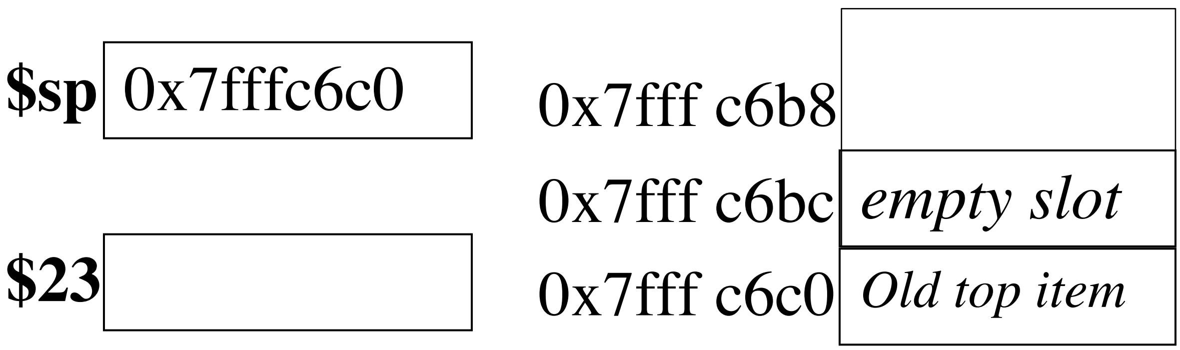
Push: subtract from $sp
Pop: add to $sp

How memory address space is usually allocated:

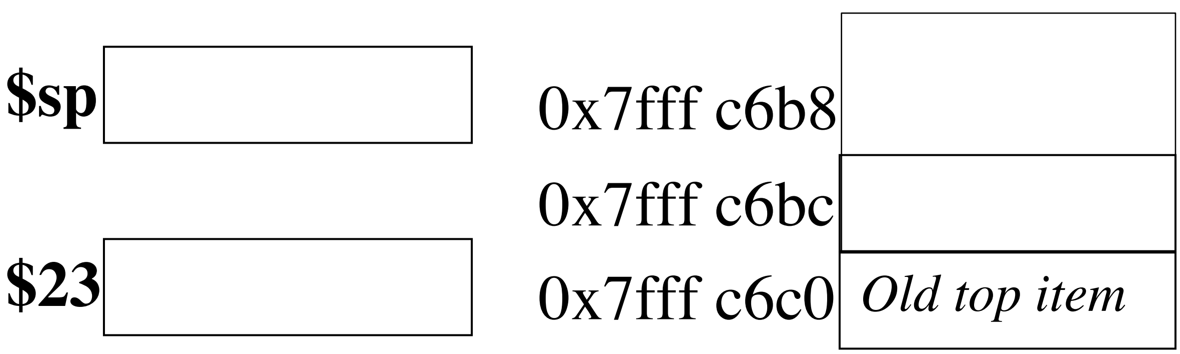


To push a 32-bit word from $23 to top of stack:

Before:



After:



Alternate way to push an item:

To pop item from top of stack into $19:

Before:

$sp 0x7fff c6bc

0x7fff c6b8 *empty slot*

0x7fff c6bc 0x89abcdef

$19

After:

$sp

0x7fff c6b8

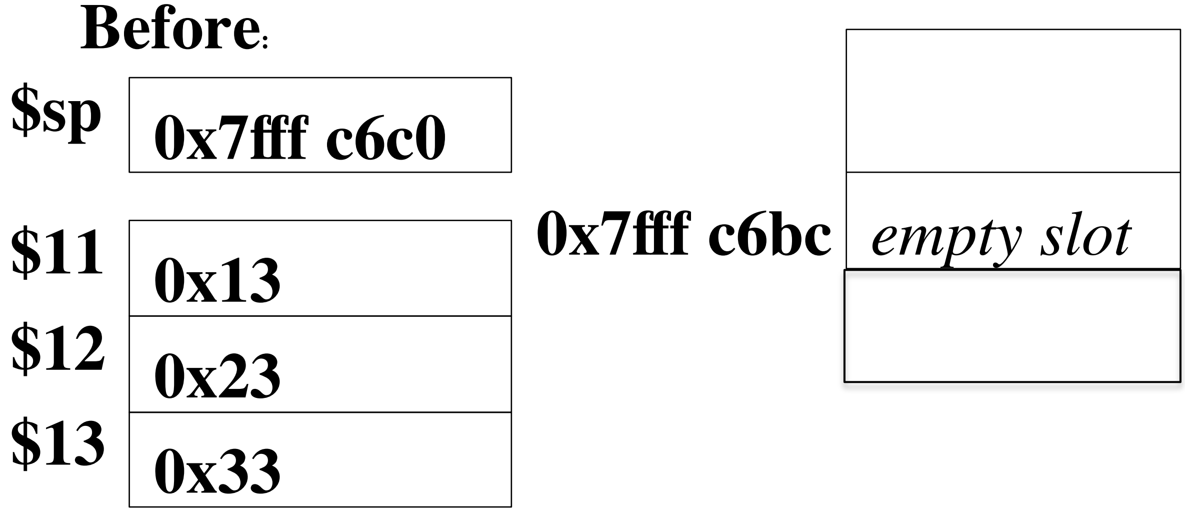
0x7fff c6bc

$19

Alternate way:

To push \$11, \$12, \$13 onto stack, in order:

Before:



after:

$sp	

0x7fff c6b0	
0x7fff c6b4	
0x7fff c6b8	
0x7fff c6bc	

$11, $12, $13 unchanged

To pop top items from stack into $13, $12, $11:

before:

$sp	0x7fffc6b4

0x7fff c6b0	
0x7fff c6b4	0x01020304
0x7fff c6b8	0x05060708
0x7fff c6bc	0x090a0b0c

after:

$sp	
$11	
$12	
$13	

0x7fff c6bc	*empty slot*

Basic mechanics of functions (review)

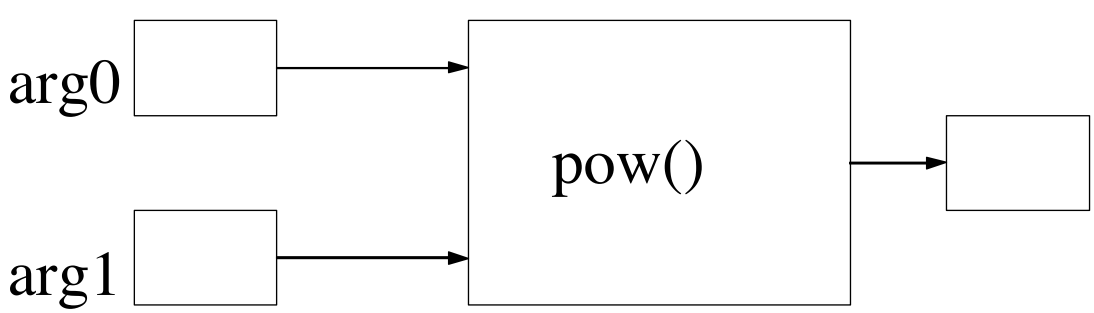
C/C++ program: compute x^p, y^q

```
int pow(int, int);

int main()
{
  int x,p,y,q;
  int result;
  result = pow(x,p);
```

[etc etc…more code not shown]

Function prototype for pow:

```
int pow(int, int);
```



With prototype for pow(), a caller has all the information needed to call pow().

* no need to see the code for pow()!

Caller code (main) and callee function (pow) can be written independently.
Caller and callee communicate through arguments and return values defined in prototype.

Arguments:
* “placeholders” for copies of data
* caller puts copies of its variables in argument placeholders
* callee function puts return value in another “placeholder”

Registers are used for arguments and return values.

Register use convention:
set of rules on how to use registers so software modules can communicate with each other properly

(register use conventions are part of the *call conventions* of a system)

In MIPS:
first four arguments in $a0-$a3 ($4-$7)

return values in $v0, $v1 ($2, $3)

Caller (main) puts correct values in $a0-$a3 before calling function.

Callee (pow) puts return value in $v0, $v1 when it ends.

Control flow of main and pow():

```
int main()
{
  int x,p,y,q;

  temp = pow(x,p);

  temp = pow(y,q);
}

int pow(int arg0, int arg1)
{


  return ?;
}
```

Each function call must remember return address (where to return to when function is done).

To call function pow() in MIPS,
use jump and link (jal) instruction.

Skeleton of MIPS version of C/C++ program:

```
main:

        jal     pow    # pow(x,p)

        jal     pow    # pow(y,q)

        li      $v0,10
        syscall


pow:    [code for pow() function]
```

Return address:
address after the jal instruction that we want to return to when function returns

jump and link:

jal label
* $31 (or $ra) = return address
* go to label [or, PC = address of label]

jump register:
jr R [R is any register]
* PC = contents of R

To return from function:

C/C++ prototype for pow():

```
int pow(int arg0, int arg1)
```

MIPS comment header for pow():

```
#    int pow(int arg0,int arg1)
#
#    a0        arg0
#    a1        arg1
#    v0        result
```

In C/C++ main program,
`result = pow(x,p);`
means:

1) copy x into placeholder arg0 ($a0)
2) copy p into placeholder arg1 ($a1)
3) call pow()
4) when pow() returns, copy return value from placeholder into variable result

In MIPS main program, “compiler” decides:

```
# x   $s0
# p   $s1
# result  $s2
```

C/C++ pow() function:

[(arg0) ^{arg1}]

```
int pow(int arg0, int arg1)
{
   int product = 1, i;









}
```

Note:
* pow() is written independently of main!
* main and pow() communicate through arguments and return value

Modularity: organize code into independent modules

Translate pow() into MIPS.
pow() does not call any function; it is a *leaf* procedure/function.
MIPS Rule: for leaf procedures, all local variables are allocated to $t? registers. (They look like temps! Why?)

Choose registers for all variables/placeholders:
* arg0, arg1 in $a0, $a1
* local variables in $t?

```
pow:
```

[Example 4.1 (C++ and MIPS versions):
http://unixlab.sfsu.edu/~whsu/csc256/PROGS/4.1.cpp
http://unixlab.sfsu.edu/~whsu/csc256/PROGS/4.1.s]

```
#
# CSc 256 Example 4.1: Power function
# Name: William Hsu
# Date: 6/22/2010
# Description:  Computes x^p, y^q

        .data
endl:   .asciiz "\n"

# x             $s0
# p             $s1
# result        $s2
# y             $s3
# q             $s4

        .text
main:   li      $s0, 3          # int x = 3;
        li      $s1, 4          # int p = 4;
        li      $s3, 5          # int y = 5;
        li      $s4, 6          # int q = 6;

        add     $a0, $s0, $0    #  result =
                                #  pow(x, p);
        add     $a1, $s1, $0
        jal     pow
        move    $s2, $v0

        move    $a0, $v0        #  cout <<
                                #  result <<
                                #  endl;
        li      $v0, 1
```

```
syscall

li      $v0, 4
la      $a0, endl
syscall

add     $a0, $s3, $0     #  result =
                         #  pow(y, q);
add     $a1, $s4, $0
jal     pow
move    $s2, $v0

move    $a0, $v0         #  cout <<
                         #  result <<
                         #  endl;
li      $v0, 1
syscall

li      $v0, 4
la      $a0, endl
syscall

li      $v0, 10
syscall                  #}
```

```
# int pow(int arg0, int arg1)
# arg0          $a0
# arg1          $a1
# return result in $v0

# Computes arg0 to the arg1-th power
# i             $t0
# product       $t1

pow:    li      $t1, 1    #  int product = 1;
        li      $t0, 0    #  for (int i=0;
                          #    i<arg1; i++) {
        bge     $t0, $a1, endpow
for:    mul     $t1, $t1, $a0
                          #    product *= arg0;
        addi    $t0, $t0, 1
        blt     $t0, $a1, for
                          #  }
endpow: move    $v0, $t1  #  return product;
        jr      $ra       #}
```

Nested function (poly())

C/C++ program:
compute polynomial $x^4 + x^3 + 1$, for x = 2 to 4

Define poly() function to compute polynomial:

```
int poly(int)
```

Skeleton of C/C++ program:

```
int main()
{
  int i;
  int result;

  for (i=2; i<=4; i++) {
    result = poly(i);
    cout << result << endl;
  }
}
```

poly() function calls pow():

```
int poly(int arg)
{
  int temp1,result;

  temp1 = pow(arg,4);
  result = pow(arg,3);
  result = temp1 + result + 1;
  return result;
}
```

Again, note that main, poly() and pow() are completely modular!

poly()'s return address will be in $ra. When poly calls pow, pow's return address will overwrite poly's return address in $ra.

Hence:

Skeleton for poly() so far:

```
poly: addi  $sp, $sp, -4
      sw    $ra, ($sp)


// do some useful work



      lw    $ra, ($sp)
      addi  $sp, $sp, 4
      jr $ra
```

Remember: when we write poly(), we don't see code for other functions. (Compilers work in a similar way when they compile functions; *interprocedural optimizations* not always possible.)

From C++ code:

```
temp1 = pow(arg,4);
result = pow(arg,3);
result = temp1 + result + 1;
```

Where to allocate temp1?
How about $t0?

Need to allocate temp1 to a register that is *safe*, i.e., will not be changed by pow().
These are the $s? registers (for *saved* registers).

Register use conventions: set of rules that determine how registers are used in a system.

In general, a function is written/translated independently from other functions.
Suppose we are writing a *caller* function; it calls a *callee* function.
What happens to registers after the callee returns?

```
caller:
      $s0 =
      $a0 =
      $v0 =
      $t0 =
      jal      callee   # callee()

      $s0 =
      $a0 =
      $v0 =
      $t0 =
```

MIPS register use convention determines:
* $a?, $v?, $t? registers not preserved across function calls
(What other register is not preserved?)
* $s? registers preserved across function calls

RULE (part of MIPS register use convention):

Suppose we are writing/translating function F().

If $s? is used as a local var. in F(),
* the old value of $s? must first be saved on the stack
* before the function returns it must restore all $s? registers that were saved

Back to poly(); we need to allocate temp1 to an $s? register (say, $s1).

Skeleton for poly() so far:

```
poly: addi  $sp, $sp, -8
      sw    $s1, ($sp)
      sw    $ra, 4($sp)


// do some useful work



      lw    $ra, 4($sp)
      lw    $s1, ($sp)
      addi  $sp, $sp, 8
      jr $ra
```

Note on scope of arguments (review)

There are various arguments floating around:
`poly()`'s argument `arg`
`pow()`'s arguments `arg0, arg1`

If it helps makes things less confusing:

```
int poly(int poly_arg)
{
...
  temp1 = pow(poly_arg,4);
  result = pow(poly_arg,3);
...
}
```

Note that all arguments share \$a0, \$a1, etc!
`poly()` will use \$a0 for `poly_arg`.

C/C++ prototype for poly():

```
int poly(int arg)
```

MIPS prototype/comment header for poly():

```
#        int poly(int arg)
#
#        a0      arg
#        v0      result
```

Note from C/C++ code:

```
  temp1 = pow(arg,4);
  result = pow(arg,3);
```

arg (in $a0) is used twice; $a0 needs to be the same after call to pow().

But register use convention says $a0 must be assumed to contain garbage after call to pow() (not preserved across function call!)

Hence:

For nested function calls, often need to make a copy of each argument (in this case, `poly_arg`); then reuse $a0 etc for `pow()`'s arguments.

Now ready to translate poly() into MIPS.

$a0 is arg, $v0 is result.

Choose:

$s0 contains copy of arg, $s1 contains temp1.

```
poly: addi  $sp, $sp, -12
      sw    $s1, ($sp)
      sw    $s0, 4($sp)
      sw    $ra, 8($sp)
```

```
      lw    $ra, 8($sp)
      lw    $s0, 4($sp)
      lw    $s1, ($sp)
      addi  $sp, $sp, 12
      jr $ra
```

Complete version of 4.2:
[Example 4.2 (C++ and MIPS versions):
http://unixlab.sfsu.edu/~whsu/csc256/PROGS/4.2.cpp
http://unixlab.sfsu.edu/~whsu/csc256/PROGS/4.2.s]

```
#
# CSc 256 Example 4.2: Poly function
# Name: William Hsu
# Date: 6/22/2010
# Description:  Computes x^4 + x^3 + 1, x from 2 to 4

        .data
endl:   .asciiz "\n"

# i       $s0
# 4       $s1

        .text
main:   li      $s1, 4
        li      $s0, 2          #  for (i=2; i<=4; i++) {
m_for:  move    $a0, $s0        #    result = poly(i);
        jal     poly

        move    $a0, $v0        #    cout << result << endl;
        li      $v0, 1
        syscall

        la      $a0, endl
        li      $v0, 4
        syscall

        addi    $s0, $s0, 1     #  }
        ble     $s0, $s1, m_for

        li      $v0, 10         #}
        syscall
```

```
# int poly(int arg)
# arg   $a0
# return result in $v0

# computes arg^4 + arg^3 + 1
# copy of arg   $s0
# temp1         $s1

poly:   addi    $sp, $sp, -12
        sw      $s1, ($sp)
        sw      $s0, 4($sp)
        sw      $ra, 8($sp)

        move    $s0, $a0
        li      $a1, 4          #  temp1 = pow(arg, 4);
        jal     pow
        move    $s1, $v0

        move    $a0, $s0        #  result = pow(arg, 3);
        li      $a1, 3
        jal     pow

        add     $v0, $v0, $s1   #  result = temp1 + result + 1;
        addi    $v0, $v0, 1

        lw      $ra, 8($sp)
        lw      $s0, 4($sp)
        lw      $s1, ($sp)
        addi    $sp, $sp, 12    #  return result;
        jr      $ra             #}
```

pow() not shown (same as in 4.1.s)

Summary of MIPS register use conventions

Set of rules agreed on by software developers to make sure software modules can communicate properly in a system.

$a0 - $a3 are arguments
$v0 - $v1 are return values
$a? ,$v?, $t? not preserved across function calls:

```
caller:
      $a? =
      jal      callee
      $a? =
```

must assume $a?, $v? change contents after any function call.

$t? are temporaries; also not preserved across function calls.
It is the responsibility of the caller to save $a?, $v?, $t?, if necessary [caller-saved]

$s? are saved values, also for temporaries

$s? registers are preserved across function calls:

```
caller:
     $s? =
     jal     callee
     $s? =
```

$s? is guaranteed to contain the same contents before and after any function call.

It is the responsibility of the callee to save and restore $s? registers that it uses [callee-saved]

A compiler will choose the correct type of register to use for each local variable.

Elaboration: importance of saving $s?

Let's take a look at main() in 4.2:

```
# i        $s0
# result   $v0
# 4        $s1
        .text
main: li   $s1, 4
      li   $s0, 2     #  for (i=2;
                      #  i<=4; i++) {
m_for:move $a0, $s0   #    result =
                      #    poly(i);
      jal  poly
                      #    print result
                      #  }

      [print result code not shown…]

      addi $s0, $s0, 1
      ble  $s0, $s1, m_for
```

Note we chose: i is $s0, result is $v0, 4 in $s1
Can i be in $t? instead?
Can 4 be in $t? instead?

4.2 works correctly because when we wrote poly(), we *carefully* saved and restored $s0 and $s1!

What if we forgot?

Variation: suppose we need to add 1 to result before printing it.

```
for (i=2; i<=4; i++) {
    result = poly(i);
    result = result + 1;
    cout << result << endl;
}
```

Can result be in $t? ?

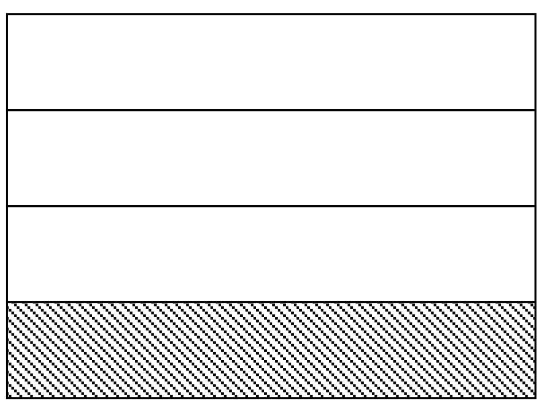
Call frame (stack frame, activation record)

a data structure that is created on the stack when a function is called.
When a function returns, it deletes its call frame from the stack.

`$sp`

address	contents
0x7fffc6b0	
0x7fffc6b4	
0x7fffc6bc	
0x7fffc6c0	

General format for call frame:



Note: leaf procedures have no call frames!

Call frames keep track of function calls that are currently *active*.

Example code:

```
main:            # int main() {

      jal F1     #  ?? = F1() * 2;
                 # }

# END OF MAIN #

F1:              # int F1() {

      jal F2     #  ?? = F2() …
                 # }
      jr  $ra

# END OF F1 #

F2:              # int F2() {

      jal F3     #  F3() …
                 # }
      jr  $ra

# END OF F2 #
# F3 is leaf procedure; not shown…
```

0) initial stack ($sp = 0x7fffc6c0)

address	contents
0x7fffc6c0	[main’s top item]

1) main calls F1; F1’s call frame is constructed on stack ($sp = 0x7fffc6b4):

<table>
<tr><td>address</td><td>contents</td></tr>
<tr><td>0x7fffc6b4</td><td rowspan="3">[F1’s call frame]</td></tr>
<tr><td>0x7fffc6b8</td></tr>
<tr><td>0x7fffc6bc</td></tr>
<tr><td>0x7fffc6c0</td><td>[main’s top item]</td></tr>
</table>

2) F1 calls F2; F2’s call frame is constructed on stack, above F1’s call frame ($sp = 0x7fffc6ac)

<table>
<tr><td>address</td><td>contents</td></tr>
<tr><td>0x7fffc6ac</td><td rowspan="2">[F2’s call frame]</td></tr>
<tr><td>0x7fffc6b0</td></tr>
<tr><td>0x7fffc6b4</td><td rowspan="3">[F1’s call frame]</td></tr>
<tr><td>0x7fffc6b8</td></tr>
<tr><td>0x7fffc6bc</td></tr>
<tr><td>0x7fffc6c0</td><td>[main’s top item]</td></tr>
</table>

3) F2 calls F3; F3 is leaf procedure, no call frame

4) F3 returns, back in F2

5) F2 returns; F2’s call frame is deleted from stack, back in F1:

<table>
<tr><td><code>address</code></td><td><code>contents</code></td></tr>
<tr><td><code>0x7fffc6b4</code></td><td rowspan="3"><code>[F1’s call frame]</code></td></tr>
<tr><td><code>0x7fffc6b8</code></td></tr>
<tr><td><code>0x7fffc6bc</code></td></tr>
<tr><td><code>0x7fffc6c0</code></td><td><code>[main’s top item]</code></td></tr>
</table>

6) F1 returns; F1’s call frame is deleted from stack, back in main:

address	contents
0x7fffc6c0	[main’s top item]

In general: the stack contains all the information needed to keep track of function calls. (Even recursion!)

Guidelines for MIPS functions, args in registers

In main program (or caller function):

1) put arguments into input registers ($a0 - $a3)
2) call function: jal func_name

In function code (callee code):

1) if function calls another function, save $31 (return address) on stack
2) if $s0-s8 used in current function as temporaries, save on stack
    [steps 1-2 create call frame]
    [if function calls another function, save $a? in $s? if necessary]
3) at this point:
arguments are in $a0-a3 (or $s?)
temporaries/local vars. are in $s0-s8 or $t0-t9

  perform computations

4) if function returns a result, make sure result is in $v0 or $v1
5) if $s0-$s8 saved on stack, restore old values to registers
6) if $31 saved on stack in step 1, restore old value to $31
7) return to main program (or calling function): jr $31

Back in main program (or calling function):

1) if function returns a result, result is in $v0 or $v1

Tracing poly.s:

before first jal poly (assume $sp = 0x7fff effc):

reg	contents	Mem address	contents
$pc		0x7fffefdc	
$sp		0x7fffefe0	
$a0		0x7fffefe4	
$a1		0x7fffefe8	
$v0		0x7fffefec	
$s0		0x7fffeff0	
$s1		0x7fffeff4	
$31		0x7fffeff8	

after first jal poly (poly(2)):

reg	contents	Mem address	contents
$pc		0x7fffefdc	
$sp		0x7fffefe0	
$a0		0x7fffefe4	
$a1		0x7fffefe8	
$v0		0x7fffefec	
$s0		0x7fffeff0	
$s1		0x7fffeff4	
$31		0x7fffeff8	

in poly(2), after sw $ra, 8($sp)

reg	contents	Mem address	contents
$pc		0x7fffefdc	
$sp		0x7fffefe0	
$a0		0x7fffefe4	
$a1		0x7fffefe8	
$v0		0x7fffefec	
$s0		0x7fffeff0	
$s1		0x7fffeff4	
$31		0x7fffeff8	

in poly(2), before first jal pow:

reg	contents	Mem address	contents
$pc		0x7fffefdc	
$sp		0x7fffefe0	
$a0		0x7fffefe4	
$a1		0x7fffefe8	
$v0		0x7fffefec	
$s0		0x7fffeff0	
$s1		0x7fffeff4	
$31		0x7fffeff8	

after first jal pow (pow(2,4)):

reg	contents	Mem address	contents
$pc		0x7fffefdc	
$sp		0x7fffefe0	
$a0		0x7fffefe4	
$a1		0x7fffefe8	
$v0		0x7fffefec	
$s0		0x7fffeff0	
$s1		0x7fffeff4	
$31		0x7fffeff8	

in pow(2,4), at label endpow:

reg	contents	Mem address	contents
$pc		0x7fffefdc	
$sp		0x7fffefe0	
$a0		0x7fffefe4	
$a1		0x7fffefe8	
$v0		0x7fffefec	
$s0		0x7fffeff0	
$s1		0x7fffeff4	
$31		0x7fffeff8	

pow(2,4) returns, after jr $ra (back in poly(2)):

reg	contents	Mem address	contents
$pc		0x7fffefdc	
$sp		0x7fffefe0	
$a0		0x7fffefe4	
$a1		0x7fffefe8	
$v0		0x7fffefec	
$s0		0x7fffeff0	
$s1		0x7fffeff4	
$31		0x7fffeff8	

back in poly(2), before second jal pow:

reg	contents	Mem address	contents
$pc		0x7fffefdc	
$sp		0x7fffefe0	
$a0		0x7fffefe4	
$a1		0x7fffefe8	
$v0		0x7fffefec	
$s0		0x7fffeff0	
$s1		0x7fffeff4	
$31		0x7fffeff8	

after poly(2) calls pow(2,3):

reg	contents	Mem address	contents
$pc		0x7fffefdc	
$sp		0x7fffefe0	
$a0		0x7fffefe4	
$a1		0x7fffefe8	
$v0		0x7fffefec	
$s0		0x7fffeff0	
$s1		0x7fffeff4	
$31		0x7fffeff8	

in pow(2,3), at label endpow:

reg	contents	Mem address	contents
$pc		0x7fffefdc	
$sp		0x7fffefe0	
$a0		0x7fffefe4	
$a1		0x7fffefe8	
$v0		0x7fffefec	
$s0		0x7fffeff0	
$s1		0x7fffeff4	
$31		0x7fffeff8	

pow(2,3) returns, after jr $ra (back in poly(2)):

reg	contents	Mem address	contents
`$pc`		`0x7fffefdc`	
`$sp`		`0x7fffefe0`	
`$a0`		`0x7fffefe4`	
`$a1`		`0x7fffefe8`	
`$v0`		`0x7fffefec`	
`$s0`		`0x7fffeff0`	
`$s1`		`0x7fffeff4`	
`$31`		`0x7fffeff8`	

back in poly(2), after add $sp,$sp,12:

reg	contents	Mem address	contents
`$pc`		`0x7fffefdc`	
`$sp`		`0x7fffefe0`	
`$a0`		`0x7fffefe4`	
`$a1`		`0x7fffefe8`	
`$v0`		`0x7fffefec`	
`$s0`		`0x7fffeff0`	
`$s1`		`0x7fffeff4`	
`$31`		`0x7fffeff8`	

poly(2) returns; after jr $ra, back in main:

reg	contents	Mem address	contents
$pc		0x7fffefdc	
$sp		0x7fffefe0	
$a0		0x7fffefe4	
$a1		0x7fffefe8	
$v0		0x7fffefec	
$s0		0x7fffeff0	
$s1		0x7fffeff4	
$31		0x7fffeff8	

Passing arrays as arguments

In most programming languages, arrays are passed by reference (not by value!)

Function str_len() returns length of string.

Code for main():

```
int str_len(char *);

int main() {
  char str[] = "abcde";
  int length;

  length = str_len(str);
```

str_len() is passed the base address of the array str[].

Code for str_len():

```
int str_len(char *arg) {
  char *ptr;
  int count = 0;

  ptr = arg;
  while (*ptr != 0) {
    count++;
    ptr++;
  }
  return count;
}
```

Excerpt of MIPS main program:

```
       .data
str:   .asciiz "abcde"

       .text

main:
       la     $a0,str    # length =
                         #  str_len(str);
       jal    str_len

       move   $s0,$v0
```

MIPS version of str_len():

```
#   int str_len(char *arg)
#
#   $a0      arg
#   $v0      length of string

str_len:
```

[C++/MIPS versions:
http://unixlab.sfsu.edu/~whsu/csc256/PROGS/4.3.cpp
http://unixlab.sfsu.edu/~whsu/csc256/PROGS/4.3.s]

Patterson and Hennessy Section 2.13: Bubble-sort example

Typical bubble-sort:

```
void sort (int v[], int n) {
  int i, j;

  for (i=0; i<n; i++) {
    for (j=i-1; j>=0 && v[j] > v[j+1];
      j-=1) {
      swap(v, j);
    }
  }
}

void swap(int v[], int k) {
  int temp;

  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

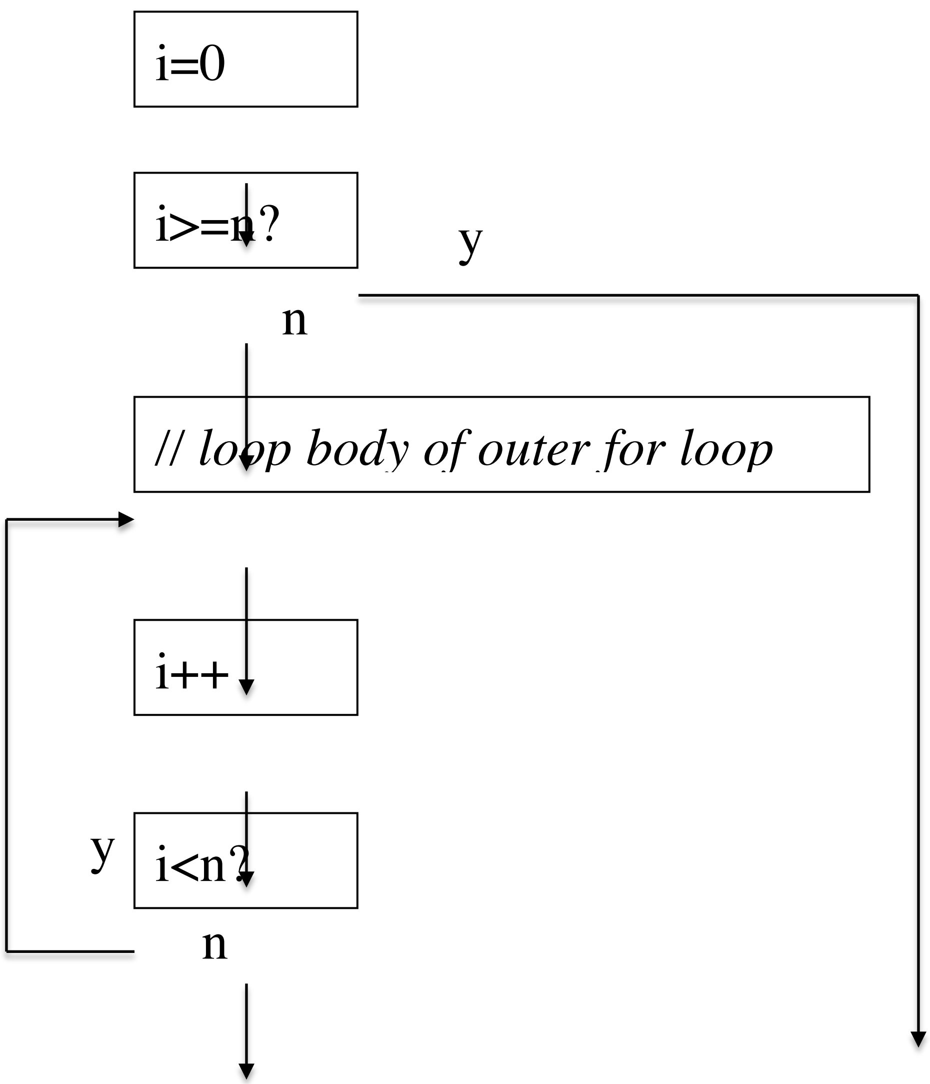
Translate swap(); leaf procedure, so can use $t? for all local variables.

Need to calculate &v[k]; also have &v[k+1]!

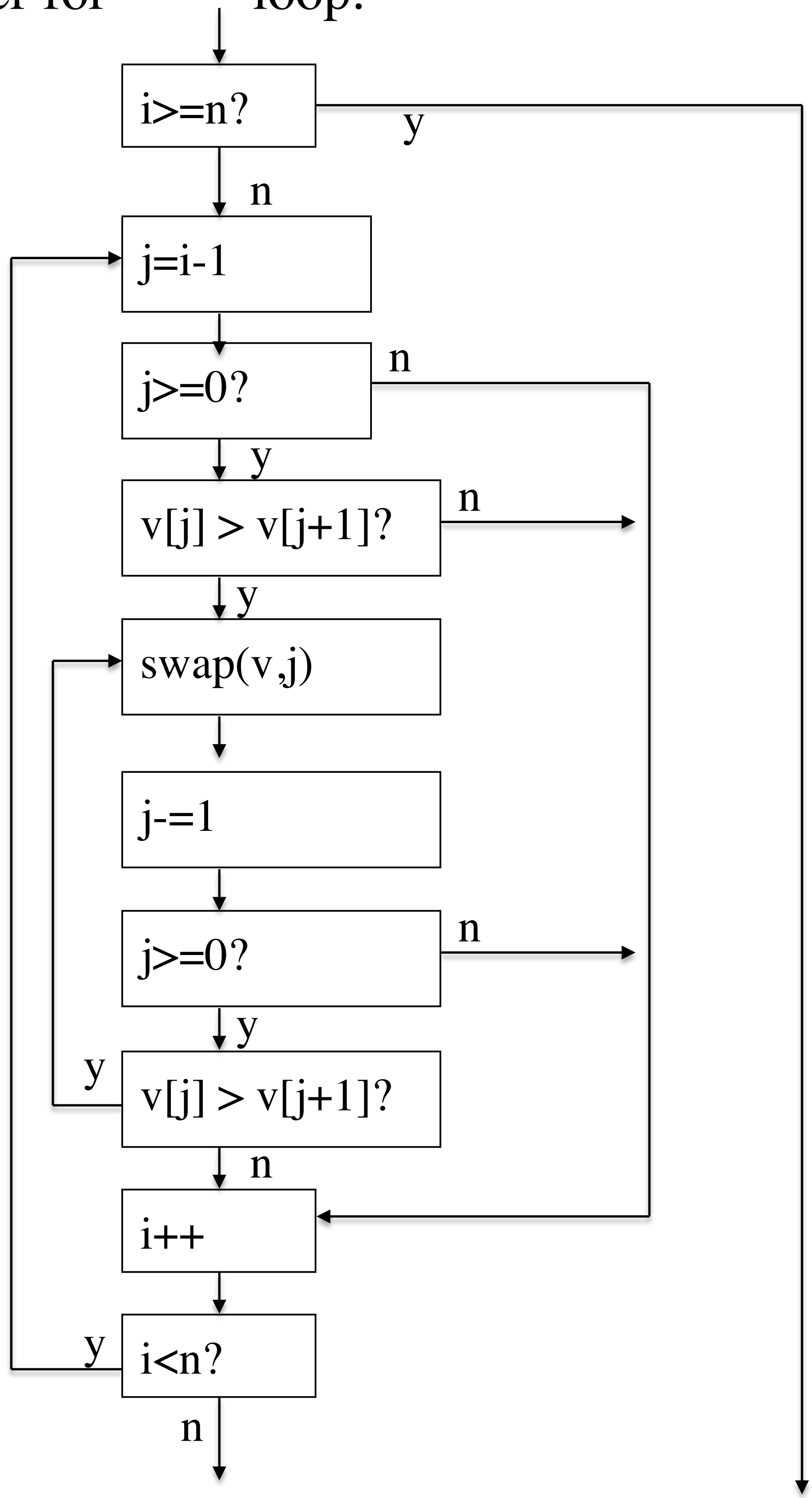
```
# temp   $t0
swap:                        # $t1=&v[k]



                             # temp=v[k];

                             # v[k]=v[k+1];


                             # v[k+1]=temp;


         jr $ra
```

Flowchart for sort()

Outer for loop:



Inner for loop:



Summary

Topics covered in this chapter:

Stack operations
Simple functions (leaf procedures)
Nested functions
Stack frames/activation records
Arrays as arguments