

MIPS Subroutines

CS31

Pascal Van Hentenryck



MIPS Subroutines

Functions in C, C++

Methods in Java

- A method is just a function with the object as the first parameter

Outline

- Branching and return
- Passing parameters
- Saving registers

Subroutines

Procedure without parameters

- Where to jump?
- Where to go back?

```
__start:      move    $s0,$s1
              ...
              {call subroutine gumbo}
              mul     $s2,$s5,$s7
              ...
              done

gumbo:       add     $s3,$s4,$s5
              ...
              {go back to where we came from}
```

Getting Back

MAL helps us quite a bit

```
jal label
```

- puts the address of the next instruction into register \$ra (return address)
- branches to label

This is easy to do in hardware since the PC contains the right address (or almost)

Example Subroutine

```
__start:    li    $s0, 7
            jal   verse1
            jal   refrain
            jal   verse2
            jal   refrain
            done

verse1:    ...
            ...
            jr   $ra

refrain:   ...
            ...
            jr   $ra
```

Could we do without `jal`?

Example Subroutine

```
start:      li    $s0, 7
            jal   verse1
            jal   refrain
            jal   verse2
            jal   refrain
            done

verse1:     ...
            ...
            jr    $ra

refrain:    ...
            ...
            jr    $ra
```

- What if `verse1` does a `jal`?
- What if it uses `$s0`?

The System Stack

(A Necessary Digression)

Sometimes we have to save data into memory:

- return addresses for nested subroutines
- register values if more than one subroutine wants to use the same register

It's inconvenient to have to anticipate exactly how much such storage we'll need and allocate memory to it explicitly.

Instead, we'll construct another way to allocate memory locations: the system stack.

Stacks in the Abstract

Stacks have two operations:

- push(item): add item to the top of the stack
- pop: remove the item from the top of the stack

push a

push b

push c

pop =>

pop =>

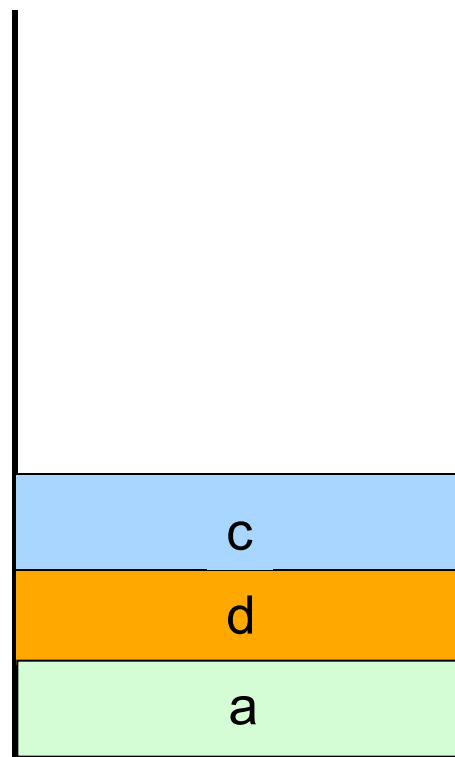
push d

push e

pop =>

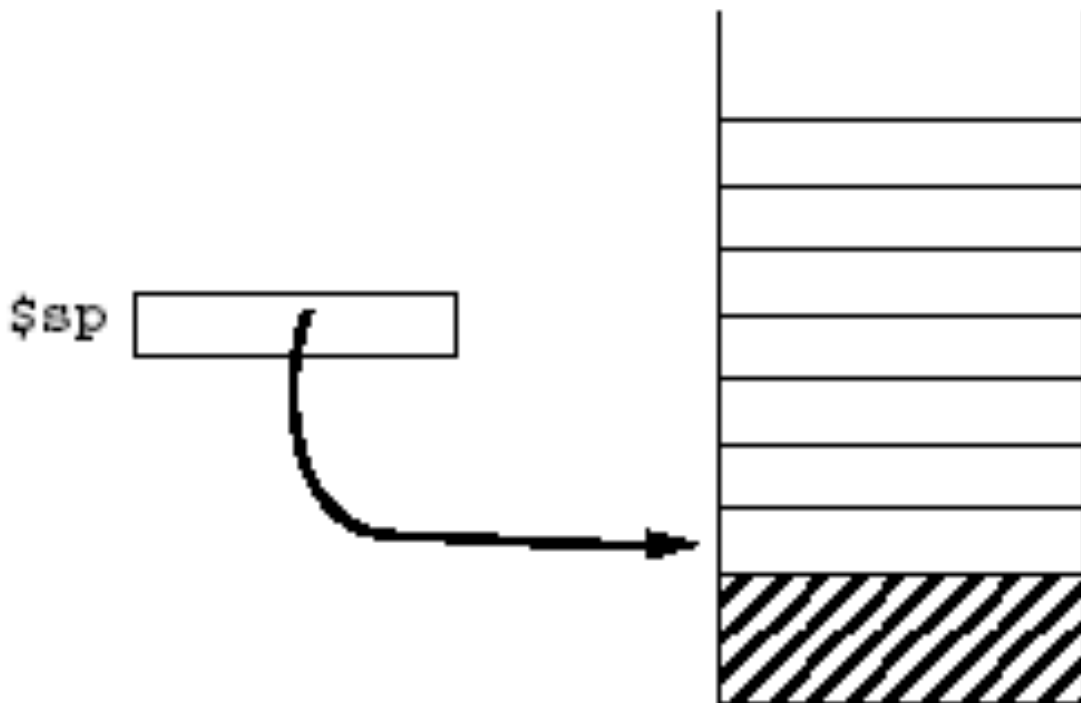
pop =>

pop =>



MIPS Stack

- the MIPS system stack is in memory (the same memory as your program and data)
- register `$sp` contains the stack pointer
- the stack grows in the direction of smaller addresses
- the stack pointer always contains the address of the next free location



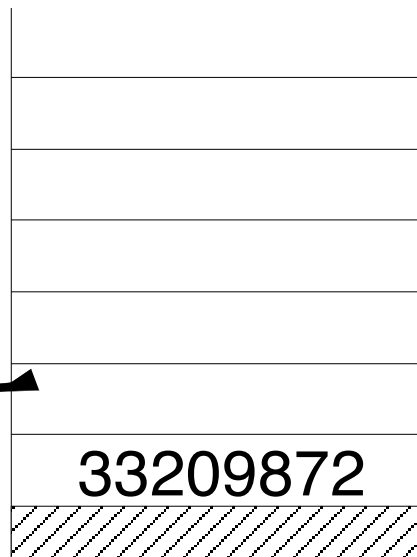
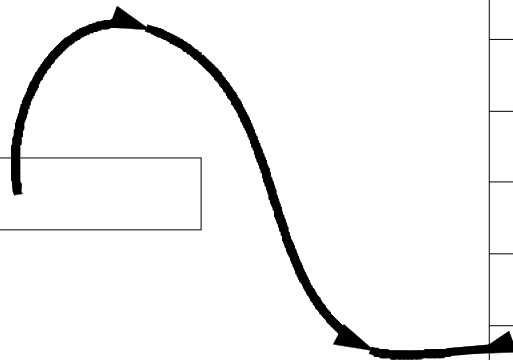
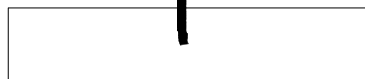
Pushing

To push a word onto the stack:

```
sub    $sp, $sp, 4  
li     $s0, 0x12345678  
sw     $s0, 4($sp)
```

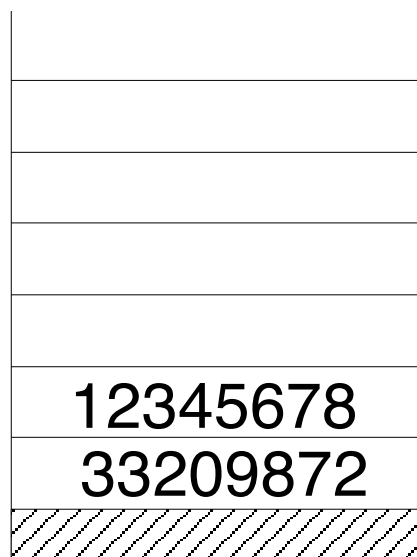
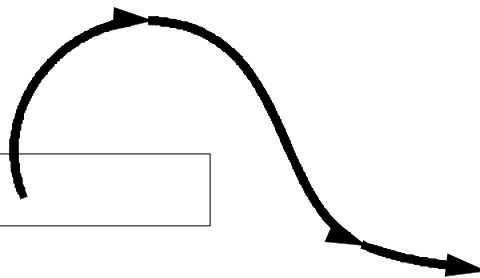
Before:

\$sp



After:

\$sp



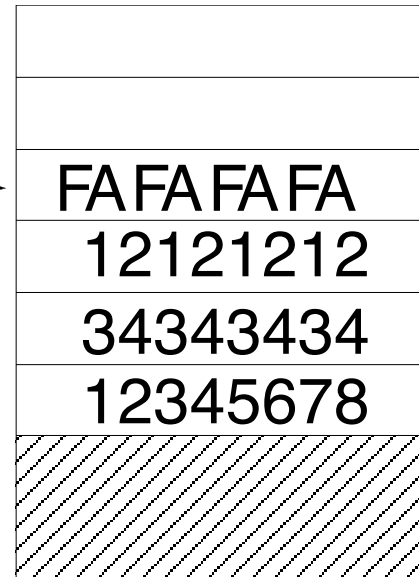
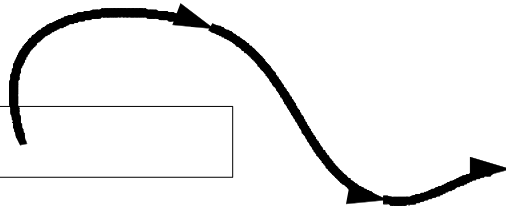
Popping

To pop a word off the stack:

```
lw      $s0, 4($sp)
add     $sp, $sp, 4
```

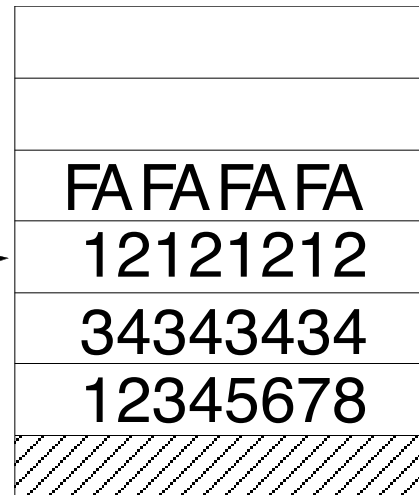
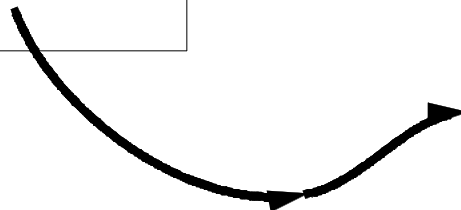
Before:

\$sp



After:

\$sp



Saving Return Addresses

At the beginning of every subroutine, push the return address on the stack. Then pop it at the end.

```
start:
    jal    mumbo
    ...
done

mumbo:
    sub    $sp, $sp, 4    # push ra
    sw     $ra, 4($sp)

    ...
    jal    jumbo
    ...
    lw     $ra, 4($sp)    # pop ra
    add    $sp, $sp, 4
    jr     $ra
```

Always do this!!

S and T Registers

```
__start:  
    lw    $t0,important_value  
    jal   mumbo  
    add   $t0,$t0,1  
    ...  
done
```

S and T Registers

There are two sets of general-purpose registers:

Saved registers: \$s0-\$s8

Temporary registers: \$t0-\$t9

Saved registers must be *preserved* across subroutine calls, so if you use one in a subroutine, you must restore its old value when you're done.

Temporary registers may *not* be *preserved* across subroutine calls.

What's wrong with this picture?

```
__start:
    lw     $t0, important_value
    jal   mumbo
    add   $t0, $t0, 1
    ...
done
```

Saving Registers

At the beginning of a subroutine (after saving the return address) save any s registers you are going to use. Restore them at the end.

```
# s1 will hold the GNP
# s2 will hold the avg. grease ratio
# t0 is used for calculation
```

jumbo:

```
sub    $sp, $sp, 4      # push ra
sw     $ra, 4($sp)
sub    $sp, $sp, 4      # save $s1
sw     $s1, 4($sp)
sub    $sp, $sp, 4      # save $s2
sw     $s2, 4($sp)
...
lw     $s2, 4($sp)      # restore $s2
add    $sp, $sp, 4
lw     $s1, 4($sp)      # restore $s1
add    $sp, $sp, 4
lw     $ra, 4($sp)     # pop ra
add    $sp, $sp, 4
jr     $ra
```

Nobody said assembly language wasn't tedious.

Saving Registers More Efficiently

We can make the previous example more efficient:

```
# s1: GNP
# s2: avg. grease ratio
# t0: used for calculation
```

jumbo:

```
sub    $sp, $sp, 12
sw     $ra, 12($sp) # push ra
sw     $s1, 8($sp)  # save s1
sw     $s2, 4($sp)  # save s2
...
lw     $s2, 4($sp)  # restore $s2
lw     $s1, 8($sp)  # restore $s1
lw     $ra, 12($sp) # pop ra
add    $sp, $sp, 12
jr     $ra
```

Passing Parameters: The Easy Way

Registers \$a0-\$a3 are reserved for passing parameters. They are *not preserved* across subroutine calls.

Registers \$v0-\$v1 are for returning results.

```
    # a0: one of the nums to be averaged
    # a1: another num to be averaged
    # v0: return the result
    # t0: calculation
average:
    add    $t0,$a0,$a1
    div    $v0,$t0,2
    jr     $ra
```

What if we need to call another subroutine?

Passing Parameters: The General Way

In nested subroutines, we may have to save parameter values on the stack.

Sometimes, we'll have too many parameters to fit into 4 registers, or too many return values.

General Answer: use the stack.

- Caller pushes parameters and space for results.
- Callee uses parameters and fills in results.
- Caller pops everything.

Parameters on the Stack

```
# average the values in $s0 and $s1, put the
# result in $s2
    sub    $sp,$sp,12      # space for rslt.
    sw    $s0,8($sp)      # push 1st param
    sw    $s1,12($sp)     # push 2nd param
    jal   average
    lw    $s2,4($sp)      # get result
    add   $sp,$sp,12
done

average:
    sub   $sp,$sp,4
    sw   $ra,4($sp)
    lw   $t0,12($sp)      # load 1st param
    lw   $t1,16($sp)     # load 2nd param
    add  $t0,$t0,$t1
    div  $t0,$t0,2
    sw   $t0,8($sp)      # store result
    lw   $ra,4($sp)      # pop ra
    add  $sp,$sp,4
    jr   $ra             # return
```

Passing Parameters by Reference

In the previous examples, we have passed actual values as parameters.

What if we want to pass an *array* as a parameter.

Rather than stuffing all those values on the stack, we can simply pass its *address* (the base address) as a parameter.

This is known as passing parameters *by reference*.

Array as a Parameter

Here's a subroutine that fills a one-dimensional array of words with a given value. Parameters are

```
addr:  address
len:   number of elements in the array
value: value to fill in array elements
```

and will be passed on the stack in that order.

```
        # t0: pointer incremented thru array
        # t1: address of end of array
        # t2: value
fill:   sub    $sp,$sp,4      # push ra
        sw    $ra,4($sp)
        lw    $t0,16($sp)    # get start array
        lw    $t1,12($sp)    # get len. (elts)
        mul   $t1,$t1,4      # len. in bytes
        add   $t1,$t1,$t0    # end addr.
        lw    $t2,8($sp)     # get value
loop:   bgt   $t0,$t1,done
        sw    $t2,($t0)      # asgn val to elt
        add   $t0,$t0,4      # incr. pointer
        b     loop
done:   lw    $ra,4($sp)
        add   $sp,$sp,4
        jr   $ra
```