MONTANA
STATE UNIVERSITY

Function Calls In MIPS Assembly

Beyond the LMC

MIPS

- In the last lecture we looked at an introduction to MIPS
- It was a little annoying that we had had to keep writing all that code out to just print a string, wasn't it?
- How could we fix that?

```
.text

main:
  li $v0, 4 # print string
  la $a0, enter
  syscall

  li $v0, 5 # read int
  syscall

  move $t0, $v0 # save to temp

  li $v0, 4 # print string
  la $a0, enter
  syscall

  li $v0, 5 # read int
  syscall

  bgt $t0, $v0, skip
  move $t0, $v0 # v0 is greater
 skip:

  li $v0, 4 # print string
  la $a0, max
  syscall

    li $v0, 1 # print int
  move $a0, $t0
  syscall

  li $v0, 10  # exit
  syscall

.data
enter:   .asciiz "Enter a number: "
max:     .asciiz "Max: "

```

Subroutines

- Let's take that first print string and re-arrrange it so that we jump to some logic to print the string
- We'll define the following calling convention:
    - $a1 holds the address of the string to print

```
main:
  la $a0, enter
  j print_str

  li $v0, 5 # read int
  syscall

  move $t0, $v0 # save to temp

  li $v0, 4 # print string
  la $a0, enter
  syscall

  li $v0, 5 # read int
  syscall

  bgt $t0, $v0, skip
  move $t0, $v0 # v0 is greater
 skip:

  la $a0, max
  li $v0, 4 # print string
  syscall

  li $v0, 1 # print int
  move $a0, $t0
  syscall

  li $v0, 10  # exit
  syscall

print_str:
  li $v0, 4 # print string
  syscall
```

Subroutines

- We'll label the area we pull the logic out to *print_str* and we'll jump to it
- Looks reasonable, so… what happens?

```
main:
  la $a0, enter
  j print_str

  li $v0, 5 # read int
  syscall

  move $t0, $v0 # save to temp

  li $v0, 4 # print string
  la $a0, enter
  syscall

  li $v0, 5 # read int
  syscall

  bgt $t0, $v0, skip
  move $t0, $v0 # v0 is greater
 skip:

  la $a0, max
  li $v0, 4 # print string
  syscall

  li $v0, 1 # print int
  move $a0, $t0
  syscall

  li $v0, 10  # exit
  syscall

print_str:
  li $v0, 4 # print string
  syscall
```

Subroutines

- Hmm, OK
- We just ran off the end of the program and exited
- We need some way to jump back to where the syscall was made...

```
main:
  la $a0, enter
  j print_str

  li $v0, 5 # read int
  syscall

  move $t0, $v0 # save to temp

  li $v0, 4 # print string
  la $a0, enter
  syscall

  li $v0, 5 # read int
  syscall

  bgt $t0, $v0, skip
  move $t0, $v0 # v0 is greater
 skip:

  la $a0, max
  li $v0, 4 # print string
  syscall

  li $v0, 1 # print int
  move $a0, $t0
  syscall

  li $v0, 10  # exit
  syscall

print_str:
  li $v0, 4 # print string
  syscall

```

Subroutines

- OK, let’s add a *return* label and jump back there after the sub-routine has completed
- It works!
- But can we reuse this sub-routine elsewhere?
    - No…
    - Return is hard-coded

```
main:
  la $a0, enter
  j print_str
return:

  li $v0, 5 # read int
  syscall

  move $t0, $v0 # save to temp

  li $v0, 4 # print string
  la $a0, enter
  syscall

  li $v0, 5 # read int
  syscall

  bgt $t0, $v0, skip
  move $t0, $v0 # v0 is greater
 skip:

  la $a0, max
  li $v0, 4 # print string
  syscall

  li $v0, 1 # print int
  move $a0, $t0
  syscall

  li $v0, 10  # exit
  syscall

print_str:
  li $v0, 4 # print string
  syscall
  j return
```

Subroutines

- OK, so let’s use a register to store the address that we want to jump back to!
- Note that rather than a jump (j) instruction we now use a jump register (jr) instruction

```
main:
  la $a0, enter
  la $ra, return1
  j print_str
return1:

  li $v0, 5 # read int
  syscall

  move $t0, $v0 # save to temp

  la $a0, enter
  la $ra, return2
  j print_str
return2:

  li $v0, 5 # read int
  syscall

  bgt $t0, $v0, skip
  move $t0, $v0 # v0 is greater
 skip:

  la $a0, max
  la $ra, return3
  j print_str
return3:

  li $v0, 1 # print int
  move $a0, $t0
  syscall

  li $v0, 10  # exit
  syscall

print_str:
  li $v0, 4 # print string
  syscall
  jr $ra
```

Subroutines

- Now we are getting somewhere: we can reuse this subroutine everywhere in our program
- But is this any better than what we had before?
  - It’s actually a bit longer

```
main:
  la $a0, enter
  la $ra, return1
  j print_str
return1:

  li $v0, 5 # read int
  syscall

  move $t0, $v0 # save to temp

  la $a0, enter
  la $ra, return2
  j print_str
return2:

  li $v0, 5 # read int
  syscall

  bgt $t0, $v0, skip
  move $t0, $v0 # v0 is greater
 skip:

  la $a0, max
  la $ra, return3
  j print_str
return3:

  li $v0, 1 # print int
  move $a0, $t0
  syscall

  li $v0, 10  # exit
  syscall

print_str:
  li $v0, 4 # print string
  syscall
  jr $ra
```

Subroutines

- Note the pattern:
  - Jump to some other location and then return to the *next* instruction
- This is such a common pattern that processors almost always have an instruction for it
- On MIPS this is the *jump and link* (jal) instruction

```
main:
  la $a0, enter
  la $ra, return1
  j print_str
return1:

  li $v0, 5 # read int
  syscall

  move $t0, $v0 # save to temp

  la $a0, enter
  la $ra, return2
  j print_str
return2:

  li $v0, 5 # read int
  syscall

  bgt $t0, $v0, skip
  move $t0, $v0 # v0 is greater
 skip:

  la $a0, max
  la $ra, return3
  j print_str
return3:

  li $v0, 1 # print int
  move $a0, $t0
  syscall

  li $v0, 10  # exit
  syscall

print_str:
  li $v0, 4 # print string
  syscall
  jr $ra
```

Subroutines

- The jump and link instruction will push the *next* instructions address into the $ra register and then jump
- Now we are cooking with gas!
- We've managed to pretty well encapsulate the function and make it read well in our assembly

```
main:
  la $a0, enter
  jal print_str

  li $v0, 5 # read int
  syscall

  move $t0, $v0 # save to temp

  la $a0, enter
  jal print_str

  li $v0, 5 # read int
  syscall

  bgt $t0, $v0, skip
  move $t0, $v0 # v0 is greater
 skip:

  la $a0, max
  jal print_str

  li $v0, 1 # print int
  move $a0, $t0
  syscall

  li $v0, 10  # exit
  syscall

print_str:
  li $v0, 4 # print string
  syscall
  jr $ra
```

Procedures

- OK, so far so good…
- But how do we return a value from a function?
- Another calling convention: we use $v0 to return a value
- Here we have a function *sum_to* that sums the numbers from 0 to the given argument

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  li $t0, 0
loop:
  blez $a0, done
  add $t0, $t0, $a0
  subi $a0, $a0, 1
  j loop
done:
  move $v0, $t0
  jr $ra
```

Procedures

- Note that there is nothing special about the label *sum_to*
- It's a procedure simply because we follow the calling conventions
  - Using $a0-3 for arguments
  - Using $v0-1 for return values

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  li $t0, 0
loop:
  blez $a0, done
  add $t0, $t0, $a0
  subi $a0, $a0, 1
  j loop
done:
  move $v0, $t0
  jr $ra
```

Procedures

- Note a few other things about our procedure
  - We use a temporary variable to keep track of the sum
  - We use the blez instruction - Branch if less than or equal to zero
  - Note that we have to move $v0 to $a0 before the system call to avoid stomping on it

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  li $t0, 0
loop:
  blez $a0, done
  add $t0, $t0, $a0
  subi $a0, $a0, 1
  j loop
done:
  move $v0, $t0
  jr $ra
```

Procedures

- So far, so good
- But what if we wanted to implement this as a *recursive* function?
- How would we do that?

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  li $t0, 0
loop:
  blez $a0, done
  add $t0, $t0, $a0
  subi $a0, $a0, 1
  j loop
done:
  move $v0, $t0
  jr $ra
```

Procedures

- So far, so good
- But what if we wanted to implement this as a *recursive* function?
- How would we do that?
- Here is a naive first attempt

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1
  jal sum_to
  add $t0, $t0, $v0
done:
  move $v0, $t0
  jr $ra
```

Procedures

- First move $a0 into a temp register
- If the value is 0, move it into the return value register and return
- If not, subtract one from $a0 and call sum_to again
- Add the result of that to the temp register and fall through to return

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1
  jal sum_to
  add $t0, $t0, $v0
done:
  move $v0, $t0
  jr $ra
```

Procedures

- This looks somewhat reasonable, but when we run it, we get an infinite loop
- Why?

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1
  jal sum_to
  add $t0, $t0, $v0
done:
  move $v0, $t0
  jr $ra
```

Procedures

- When we call sum_to recursively, we stomp on the initial value of $ra from the main: code sequence
- We can *never get it back*
- Gone forever!

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1
  jal sum_to
  add $t0, $t0, $v0
done:
  move $v0, $t0
  jr $ra
```

Procedures

- Additionally, every time we call sum_to, it is going to stomp on the $t0 register!
- Complete disaster here folks
- OK, so what do we need to do?

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1
  jal sum_to
  add $t0, $t0, $v0
done:
  move $v0, $t0
  jr $ra
```

Procedures

- *> The Stack has entered the chat…*
- We need a place to store values when function calls are made, and the stack is where we are going to do that

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1
  jal sum_to
  add $t0, $t0, $v0
done:
  move $v0, $t0
  jr $ra
```

Procedures

- The stack is just a place in memory that is allowed to grow and shrink according to your needs for a given function call
- All it is in a pointer/address
- You move the pointer *down* in memory to allocate some space for your data

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1
  jal sum_to
  add $t0, $t0, $v0
done:
  move $v0, $t0
  jr $ra
```

Procedures

- MIPS supports the stack concept with the $sp register, which points to the current stack address
- Let’s fix our code using $sp

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  subi $sp, $sp, 8

  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1

  sw   $t0, 4($sp) # save the temp value
  sw   $ra, 0($sp) # save the return address

  jal sum_to

  lw   $t0, 4($sp) # restore the temp value
  lw   $ra, 0($sp) # restore the return address

  lw $t0, 4($sp)
  add $t0, $t0, $v0
done:
  move $v0, $t0
  addi $sp, $sp, 8
  jr $ra
```

Procedures

- The first thing we need to do is *bump* the stack pointer *down*
  - The stack grows from the top of memory down on most platforms
- We are going to be storing two registers so we can restore them
  - $t0 - a temporary register
  - $ra - the return address

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  subi $sp, $sp, 8

  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1

  sw   $t0, 4($sp) # save the temp value
  sw   $ra, 0($sp) # save the return address

  jal sum_to

  lw   $t0, 4($sp) # restore the temp value
  lw   $ra, 0($sp) # restore the return address

  lw $t0, 4($sp)
  add $t0, $t0, $v0
done:
  move $v0, $t0
  addi $sp, $sp, 8
  jr $ra
```

Procedures

- MIPS is a 32 bit platform and both registers are 32 bit (1 word or 4 bytes) wide
- In MIPS, as on most platforms, memory is byte-addressed
  - That is, incrementing an address by 1 moves it to the next *byte*

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  subi $sp, $sp, 8

  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1

  sw   $t0, 4($sp) # save the temp value
  sw   $ra, 0($sp) # save the return address

  jal sum_to

  lw   $t0, 4($sp) # restore the temp value
  lw   $ra, 0($sp) # restore the return address

  lw $t0, 4($sp)
  add $t0, $t0, $v0
done:
  move $v0, $t0
  addi $sp, $sp, 8
  jr $ra
```

Procedures

- So, we need to move the stack pointer by a total of 8 bytes
  - 4 bytes for the 32-bit temporary value
  - 4 bytes for the 32-bit return address
- Hence, we subtract 8 from the current stack pointer

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  subi $sp, $sp, 8

  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1

  sw   $t0, 4($sp) # save the temp value
  sw   $ra, 0($sp) # save the return address

  jal sum_to

  lw   $t0, 4($sp) # restore the temp value
  lw   $ra, 0($sp) # restore the return address

  lw $t0, 4($sp)
  add $t0, $t0, $v0
done:
  move $v0, $t0
  addi $sp, $sp, 8
  jr $ra
```

Procedures

- OK, next change
- Before we make the recursive call to sum_to, we need to store the values of $t0 and $ra onto the stack
- We do this by calling *save word* (sw) with offsets from the current pointer

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  subi $sp, $sp, 8

  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1

  sw   $t0, 4($sp) # save the temp value
  sw   $ra, 0($sp) # save the return address

  jal sum_to

  lw   $t0, 4($sp) # restore the temp value
  lw   $ra, 0($sp) # restore the return address

  lw $t0, 4($sp)
  add $t0, $t0, $v0
done:
  move $v0, $t0
  addi $sp, $sp, 8
  jr $ra
```

Procedures

- The return address, $r0 is stored at offset 0 from the stack pointer
- The temporary value, $t0, is stored 4 bytes “higher” in memory
  - That’s what the funny 4($sp) syntax means

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  subi $sp, $sp, 8

  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1

  sw   $t0, 4($sp) # save the temp value
  sw   $ra, 0($sp) # save the return address

  jal sum_to

  lw   $t0, 4($sp) # restore the temp value
  lw   $ra, 0($sp) # restore the return address

  lw $t0, 4($sp)
  add $t0, $t0, $v0
done:
  move $v0, $t0
  addi $sp, $sp, 8
  jr $ra
```

Procedures

- Next we make the recursive call
- Then we *restore* the values (which, recall, have been stomped all over on by the recursive call) from the stack, using the *load word* (lw) instruction

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  subi $sp, $sp, 8

  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1

  sw   $t0, 4($sp) # save the temp value
  sw   $ra, 0($sp) # save the return address

  jal sum_to

  lw   $t0, 4($sp) # restore the temp value
  lw   $ra, 0($sp) # restore the return address

  lw $t0, 4($sp)
  add $t0, $t0, $v0
done:
  move $v0, $t0
  addi $sp, $sp, 8
  jr $ra
```

Procedures

- We use the same offsets that we used to save the values, and we should get the original (pre call) values back
- Now wait a second, Carson...
  - I see a register we are modifying and not saving…
  - Do you see it?

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  subi $sp, $sp, 8

  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1

  sw   $t0, 4($sp) # save the temp value
  sw   $ra, 0($sp) # save the return address

  jal sum_to

  lw   $t0, 4($sp) # restore the temp value
  lw   $ra, 0($sp) # restore the return address

  lw $t0, 4($sp)
  add $t0, $t0, $v0
done:
  move $v0, $t0
  addi $sp, $sp, 8
  jr $ra
```

Procedures

- What about the *$sp* register?
- We are modifying it
- Aren't recursive calls also modifying it?
- Yes, yes they are
- Look down a bit further, just before our return jump

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  subi $sp, $sp, 8

  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1

  sw   $t0, 4($sp) # save the temp value
  sw   $ra, 0($sp) # save the return address

  jal sum_to

  lw   $t0, 4($sp) # restore the temp value
  lw   $ra, 0($sp) # restore the return address

  lw $t0, 4($sp)
  add $t0, $t0, $v0
done:
  move $v0, $t0
  addi $sp, $sp, 8
  jr $ra
```

Procedures

- Note that we are bumping the stack pointer back up by the same amount that we bumped it down when we entered the procedure
- So, if our child call bumps the $sp pointer down (it will) then it will restore it before it comes back

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  subi $sp, $sp, 8

  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1

  sw   $t0, 4($sp) # save the temp value
  sw   $ra, 0($sp) # save the return address

  jal sum_to

  lw   $t0, 4($sp) # restore the temp value
  lw   $ra, 0($sp) # restore the return address

  lw $t0, 4($sp)
  add $t0, $t0, $v0
done:
  move $v0, $t0
  addi $sp, $sp, 8
  jr $ra
```

Procedures

- This is a calling convention
  - Some registers must be preserved by the caller
    - The temporary registers
    - The argument registers
    - The return registers
    - The return address
  - Some registers must be preserved by the callee
    - The stack pointer
    - $s0-s7

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  subi $sp, $sp, 8

  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1

  sw   $t0, 4($sp) # save the temp value
  sw   $ra, 0($sp) # save the return address

  jal sum_to

  lw   $t0, 4($sp) # restore the temp value
  lw   $ra, 0($sp) # restore the return address

  lw $t0, 4($sp)
  add $t0, $t0, $v0
done:
  move $v0, $t0
  addi $sp, $sp, 8
  jr $ra
```

Procedures

- In this call we see both
  - We save $ra and $t0 before making the recursive call
    - Caller responsibility
  - We restore $sp before we return from an invocation
    - Callee responsibility

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  subi $sp, $sp, 8

  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1

  sw   $t0, 4($sp) # save the temp value
  sw   $ra, 0($sp) # save the return address

  jal sum_to

  lw   $t0, 4($sp) # restore the temp value
  lw   $ra, 0($sp) # restore the return address

  lw $t0, 4($sp)
  add $t0, $t0, $v0
done:
  move $v0, $t0
  addi $sp, $sp, 8
  jr $ra
```

Procedures

- This is how the stack works and how you can have a *StackOverflow*
    - You just bumped the stack pointer too many times and ran out of memory :)

```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  subi $sp, $sp, 8

  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1

  sw   $t0, 4($sp) # save the temp value
  sw   $ra, 0($sp) # save the return address

  jal sum_to

  lw   $t0, 4($sp) # restore the temp value
  lw   $ra, 0($sp) # restore the return address

  lw $t0, 4($sp)
  add $t0, $t0, $v0
done:
  move $v0, $t0
  addi $sp, $sp, 8
  jr $ra
```

Procedures

- This may seem like a lot but it’s not that bad if you step through it a few times
- Use MARS to do so, it becomes clear what’s going on
- *IT’S JUST CODE*

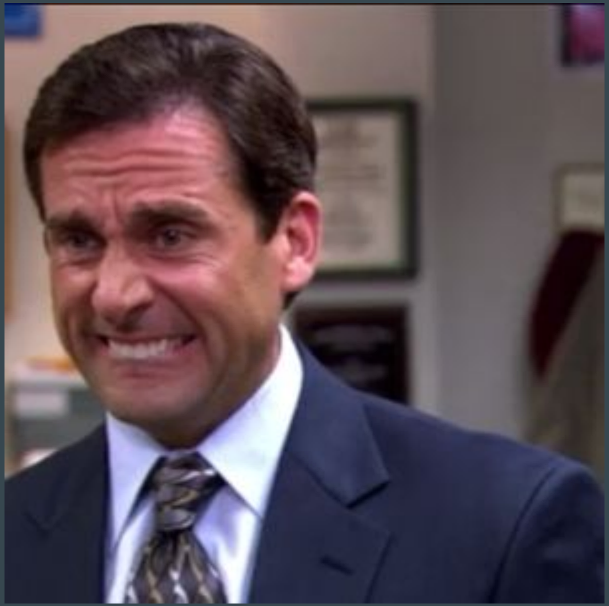
```
.text

main:
  li $a0, 100
  jal sum_to

  move $a0, $v0 # save the value
  li $v0, 1 # print int
  syscall

  li $v0, 10  # exit
  syscall

sum_to:
  subi $sp, $sp, 8

  move $t0, $a0
  blez $a0, done
  subi $a0, $a0, 1

  sw   $t0, 4($sp) # save the temp value
  sw   $ra, 0($sp) # save the return address

  jal sum_to

  lw   $t0, 4($sp) # restore the temp value
  lw   $ra, 0($sp) # restore the return address

  lw $t0, 4($sp)
  add $t0, $t0, $v0
done:
  move $v0, $t0
  addi $sp, $sp, 8
  jr $ra
```

MIPS

- OK, today we went through the painful process of learning how function calls work on the MIPS platform
- We skipped some stuff
  - Using the stack to pass additional arguments to a function
  - Using the frame pointer ($fp) which is related to the stack pointer
  - Using the global pointer ($gp) which can be used to allocate heap memory for a process
- Thankfully we have compilers to take care of all this for us
  - Thank your local compiler developer
- Next up...

MIPS

- x86 assembly



MONTANA
STATE UNIVERSITY