

Lecture 4

- HW #1 due on Monday Oct 6th (to be done individually)
- Today:
 - Finish up control flow
 - Functions in MIPS

Slides adapted from Josep Torrellas, Craig Zilles, and Howard Huang

1

An Example Function: Factorial

```
int fact(int n) {
    int i, f = 1;
    for (i = n; i > 0; i--)
        f = f * i;
    return f;
}

fact:
    li $t0, 1
    move $t1,$a0 # set i to n
loop:
    blez $t1,exit # exit if done
    mul $t0,$t0,$t1 # build factorial
    addi $t1, $t1,-1 # i--
    j loop
exit:
    move $v0,$t0
```

2

Register Correspondences

- | | | |
|-------------|---------|------------------------------|
| ▪ \$zero | \$0 | Zero |
| ▪ \$at | \$1 | Assembler temp |
| ▪ \$v0-\$v1 | \$2-3 | Value (return from function) |
| ▪ \$a0-\$a3 | \$4-7 | Argument (to function) |
| ▪ \$t0-\$t7 | \$8-15 | Temporaries |
| ▪ \$s0-\$s7 | \$16-23 | Saved Temporaries Saved |
| ▪ \$t8-\$t9 | \$24-25 | Temporaries |
| ▪ \$k0-\$k1 | \$26-27 | Kernel (OS) Registers |
| ▪ \$gp | \$28 | Global Pointer Saved |
| ▪ \$sp | \$29 | Stack Pointer Saved |
| ▪ \$fp | \$30 | Frame Pointer Saved |
| ▪ \$ra | \$31 | Return Address Saved |

3

Functions in MIPS

- We'll talk about the 3 steps in handling function calls:
 1. The program's flow of control must be changed.
 2. Arguments and return values are passed back and forth.
 3. Local variables can be allocated and destroyed.
- And how they are handled in MIPS:
 - New instructions for calling functions.
 - Conventions for sharing registers between functions.
 - Use of a stack.

4

Control flow in C

- Invoking a function changes the control flow of a program twice.
 1. **Calling** the function
 2. **Returning** from the function
- In this example the `main` function calls `fact` twice, and `fact` returns twice—but to *different* locations in `main`.
- Each time `fact` is called, the CPU has to remember the appropriate **return address**.
- Notice that `main` itself is also a function! It is called by the operating system when you run the program.

```
int main()
{
    ...
    t1 = fact(8);
    t2 = fact(3);
    t3 = t1 + t2;
    ...
}

int fact(int n)
{
    int i, f = 1;
    for (i = n; i > 1; i--)
        f = f * i;
    return f;
}
```

5

Control flow in MIPS

- MIPS uses the jump-and-link instruction `jal` to call functions.
 - The `jal` saves the return address (the address of the *next* instruction) in the dedicated register `$ra`, before jumping to the function.
 - `jal` is the only MIPS instruction that can access the value of the program counter, so it can store the return address `PC+4` in `$ra`.

`jal Fact`

- To transfer control back to the caller, the function just has to jump to the address that was stored in `$ra`.

`jr $ra`

- Let's now add the `jal` and `jr` instructions that are necessary for our factorial example.

6

Data flow in C

- Functions accept **arguments** and produce **return values**.
- The **blue** parts of the program show the actual and formal arguments of the fact function.
- The **purple** parts of the code deal with returning and using a result.

```
int main()
{
    ...
    t1 = fact(8);
    t2 = fact(3);
    t3 = t1 + t2;
    ...
}

int fact(int n)
{
    int i, f = 1;
    for (i = n; i > 1; i--)
        f = f * i;
    return f;
}
```

7

Data flow in MIPS

- MIPS uses the following conventions for function arguments and results.
 - Up to four function arguments can be “passed” by placing them in argument registers **\$a0-\$a3** before calling the function with jal.
 - A function can “return” up to two values by placing them in registers **\$v0-\$v1**, before returning via jr.
- These conventions are not enforced by the hardware or assembler, but programmers agree to them so functions written by different people can interface with each other.
- Later we'll talk about handling additional arguments or return values.

8

A note about types

- Assembly language is **untyped**—there is no distinction between integers, characters, pointers or other kinds of values.
- It is up to **you** to “type check” your programs. In particular, make sure your function arguments and return values are used consistently.
- For example, what happens if somebody passes the **address** of an integer (instead of the integer itself) to the fact function?

9

The big problem so far

- There is a big problem here!
 - The main code uses **\$t1** to store the result of fact(8).
 - But **\$t1** is also used within the fact function!
- The subsequent call to fact(3) will overwrite the value of fact(8) that was stored in \$t1.

10

Nested functions

- A similar situation happens when you call a function that then calls another function.
- Let's say A calls B, which calls C.
 - The arguments for the call to C would be placed in \$a0-\$a3, thus **overwriting** the original arguments for B.
 - Similarly, **jal C** overwrites the return address that was saved in \$ra by the earlier **jal B**.

```
A: ...
    # Put B's args in $a0-$a3
    jal B     # $ra = A2
A2: ...
```

```
B: ...
    # Put C's args in $a0-$a3,
    # erasing B's args!
    jal c     # $ra = B2
B2: ...
    jr $ra   # where does
             # this go???
```

```
C: ...
    jr $ra
```

11

Spilling registers

- The CPU has a limited number of registers for use by all functions, and it's possible that several functions will need the same registers.
- We can keep important registers from being overwritten by a function call, by saving them before the function executes, and restoring them after the function completes.
- But there are two important questions.
 - Who is responsible for saving registers—the caller or the callee?
 - Where exactly are the register contents saved?



12

Who saves the registers?

- Who is responsible for saving important registers across function calls?
 - The caller knows which registers are important to it and should be saved.
 - The callee knows exactly which registers it will use and potentially overwrite.
- However, in the typical “black box” programming approach, the caller and callee do not know anything about each other’s implementation.
 - Different functions may be written by different people or companies.
 - A function should be able to interface with any client, and different implementations of the same function should be substitutable.
- So how can two functions cooperate and share registers when they don’t know anything about each other?

13

The caller could save the registers...

- One possibility is for the *caller* to save any important registers that it needs before making a function call, and to restore them after.
- But the caller does not know what registers are actually written by the function, so it may save more registers than necessary.
- In the example on the right, *frodo* wants to preserve *\$a0*, *\$a1*, *\$s0* and *\$s1* from *gollum*, but *gollum* may not even use those registers.

```
frodo: li $a0, 3
       li $a1, 1
       li $s0, 4
       li $s1, 1

       # Save registers
       # $a0, $a1, $s0, $s1
       jal gollum

       # Restore registers
       # $a0, $a1, $s0, $s1
       add $v0, $a0, $a1
       add $v1, $s0, $s1
       jr $ra
```

14

...or the callee could save the registers...

- Another possibility is if the *callee* saves and restores any registers it might overwrite.
- For instance, a *gollum* function that uses registers *\$a0*, *\$a2*, *\$s0* and *\$s2* could save the original values first, and restore them before returning.
- But the callee does not know what registers are important to the caller, so again it may save more registers than necessary.

```
gollum: # Save registers
        # $a0 $a2 $s0 $s2
        li $a0, 2
        li $a2, 7
        li $s0, 1
        li $s2, 8
        ...

        # Restore registers
        # $a0 $a2 $s0 $s2
        jr $ra
```

15

...or they could work together

- MIPS uses conventions again to split the register spilling chores.
- The *caller* is responsible for saving and restoring any of the following **caller-saved registers** that it cares about.

\$t0-\$t9 *\$a0-\$a3* *\$v0-\$v1*

In other words, the callee may freely modify these registers, under the assumption that the caller already saved them if necessary.

- The *callee* is responsible for saving and restoring any of the following **callee-saved registers** that it uses. (Remember that *\$ra* is “used” by *jal*.)

\$s0-\$s7 *\$ra*

Thus the caller may assume these registers are not changed by the callee. – *\$ra* is tricky; it is saved by a callee who is also a caller.

- Be especially careful when writing nested functions, which act as both a caller and a callee!

16

Register spilling example

- This convention ensures that the caller and callee together save all of the important registers—*frodo* only needs to save registers *\$a0* and *\$a1*, while *gollum* only has to save registers *\$s0* and *\$s2*.

```
Frodo: li $a0, 3
       li $a1, 1
       li $s0, 4
       li $s1, 1

       # Save registers
       # $a0 and $a1
       jal gollum

       # Restore registers
       # $a0 and $a1
       add $v0, $a0, $a1
       add $v1, $s0, $s1
       jr $ra

gollum: # Save registers
        # $s0 and $s2
        li $a0, 2
        li $a2, 7
        li $s0, 1
        li $s2, 8
        ...

        # Restore registers
        # $s0 and $s2
        jr $ra
```

17

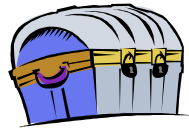
How to fix factorial

- In the factorial example, main (the caller) should save two registers.
 - \$t1* must be saved before the second call to *fact*.
 - \$ra* will be implicitly overwritten by the *jal* instructions.
- But *fact* (the callee) does not need to save anything. It only writes to registers *\$t0*, *\$t1* and *\$v0*, which should have been saved by the caller.

18

Where are the registers saved?

- Now we know who is responsible for saving which registers, but we still need to discuss where those registers are saved.
- It would be nice if each function call had its own private memory area.
 - This would prevent other function calls from overwriting our saved registers—otherwise using memory is no better than using registers.
 - We could use this private memory for other purposes too, like storing local variables.

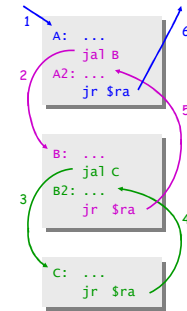


19

Function calls and stacks

- Notice function calls and returns occur in a stack-like order: the most recently called function is the first one to return.

- Someone calls A
- A calls B
- B calls C
- C returns to B
- B returns to A
- A returns



- Here, for example, C must return to B before B can return to A.

20

Stacks and function calls

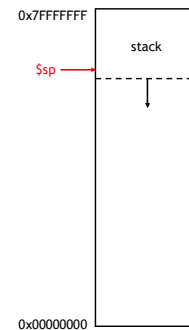
- It's natural to use a **stack** for function call storage. A block of stack space, called a **stack frame**, can be allocated for each function call.
 - When a function is called, it creates a new frame onto the stack, which will be used for local storage.
 - Before the function returns, it must pop its stack frame, to restore the stack to its original state.
- The stack frame can be used for several purposes.
 - Caller- and callee-save registers can be put in the stack.
 - The stack frame can also hold local variables, or extra arguments and return values.



21

The MIPS stack

- In MIPS machines, part of main memory is reserved for a stack.
 - The stack grows downward in terms of memory addresses.
 - The address of the top element of the stack is stored (by convention) in the "stack pointer" register, `$sp`.
- MIPS does not provide "push" and "pop" instructions. Instead, they must be done explicitly by the programmer.



22

Pushing elements

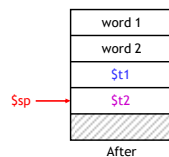
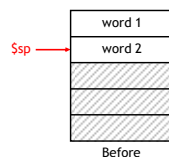
- To **push** elements onto the stack:
 - Move the stack pointer `$sp` down to make room for the new data.
 - Store the elements into the stack.
- For example, to push registers `$t1` and `$t2` onto the stack:

```
sub $sp, $sp, 8
sw $t1, 4($sp)
sw $t2, 0($sp)
```

- An equivalent sequence is:

```
sw $t1, -4($sp)
sw $t2, -8($sp)
sub $sp, $sp, 8
```

- Before and after diagrams of the stack are shown on the right.



23

Accessing and popping elements

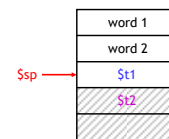
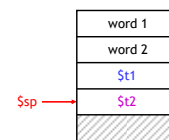
- You can access any element in the stack (not just the top one) if you know where it is relative to `$sp`.
- For example, to retrieve the value of `$t1`:

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

- You can **pop**, or "erase," elements simply by adjusting the stack pointer upwards.
- To pop the value of `$t2`, yielding the stack shown at the bottom:

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

- Note that the popped data is still present in memory, but data past the stack pointer is considered invalid.



24

Summary

- Today we focused on implementing function calls in MIPS.
 - We call functions using `jal`, passing arguments in registers `$a0-$a3`.
 - Functions place results in `$v0-$v1` and return using `jr $ra`.
- Managing resources is an important part of function calls.
 - To keep important data from being overwritten, registers are saved according to conventions for `caller-save` and `callee-save` registers.
 - Each function call uses stack memory for saving registers, storing local variables and passing extra arguments and return values.
- Assembly programmers must follow many conventions. Nothing prevents a rogue program from overwriting registers or stack memory used by some other function.