Chapter 2b: Writing Functions in Assembly Language

Gene Cooperman

Contents

1 Functions in Assembly **1**
1.1 Introduction 1
1.2 Register Naming Conventions and Aliases 1
1.3 The Stack Frame 5
1.4 Key Principles Summary 6
1.5 Practice Exercise 7

1 Functions in Assembly

1.1 Introduction

In the previous assembly lecture, you learned about basic instructions, flow control, and arrays. Now we'll explore two crucial concepts that enable modular programming in assembly: **stacks** and **subroutines** (functions).

These mechanisms allow you to:

- Break programs into reusable pieces (subroutines/functions)
- Manage temporary data efficiently
- Handle recursive function calls
- Pass parameters and return values between functions

1.2 Register Naming Conventions and Aliases

Real assembly languages use **register aliases** that indicate their conventional purpose. This ensures code from different sources can work together.

1.2.1 Common Register Aliases

Here's the standard register convention (shared by MIPS and RISC-V):

Alias	MIPS Number	RISC-V Number	Purpose	Who Saves?
`zero`	$0	x0	Always contains 0	N/A (hardwired)
`a0-a3`	$4-$7	x10-x13	Function arguments 1-4	Caller
`a4-a7`	-	x14-x17	Function arguments 5-8 (RISC-V)	Caller
`v0-v1`	$2-$3	-	Function return values (MIPS)	-

Alias	MIPS Number	RISC-V Number	Purpose	Who Saves?
`a0-a1`	-	x10-x11	Function return values (RISC-V)	-
`t0-t9`	$8-$15, $24-$25	x5-x7, x28-x31	Temporary registers	Caller
`s0-s7`	$16-$23	x8-x9, x18-x27	Saved registers	Callee
`ra`	$31	x1	Return address	Varies
`sp`	$29	x2	Stack pointer	Callee
`fp`	$30	x8	Frame pointer	Callee

1.2.2 MIPS vs RISC-V: Key Differences

While MIPS and RISC-V share similar design philosophies, they differ in some instruction syntax and conventions:

Operation	MIPS	RISC-V
call	`jal`	`jal` (or `call`)
return	`jr $ra`	`ret` (or `jalr zero, ra, 0`)
load	`lw ...`	`lw ...`
store	`sw ...`	`sw ...`
return value register	`$v0-$v1`	`a0-a1`

Key difference: MIPS uses separate registers for return values (`v0-v1`), while RISC-V reuses the argument registers (`a0-a1`) for return values. This makes RISC-V slightly more uniform but requires careful attention to when registers switch roles. Note also that MIPS uses "$" to introduce a register, and RISC-V does not.

1.2.3 How Call and Return Work

Here's a visual representation of what happens during a function call:

```
Caller:
    ...
    ...
    call myfnc  -------+
    ...         <---+  |
    ...             |  |
                    |  |
Callee:             |  |
    myfnc:      <---+--+
    ...             |
    ...             |
    return     -----+
```

When the caller executes `call myfnc`, two things happen:

1. The `ra` (return address) register is set to point to the next instruction after `call myfnc`
2. The program jumps to the address of the `myfnc` label

When the callee executes `return`, it jumps to the address stored in the `ra` register, returning control to the caller.

Implementation details:

In MIPS and RISC-V, `call myfnc` is often coded as `jal myfnc`, which stands for **"jump and link"**: - The **"link"** refers to the `ra` register that stores the return address (the link back to the caller) - The **"jump"** changes the **PC** (Program Counter) to point to the address of `myfnc`

Similarly, `return` is implemented as `jr ra` in MIPS (jump register) or `ret` in RISC-V, which jumps to the address stored in the `ra` register.

1.2.4 Complete Function Call Example

Here's a complete example using generic operations that work for both MIPS and RISC-V.

NOTE: Here, we add one more assembly instruction for MIPS/RISC-V to our repertoire. It is `li register, integer`. This stands for *load immediate*, where an *immediate* in assembly is an integer. There is a fuller explanation in Chapter 2c (2c-assembly-in-practice.pdf):
From Generic to Real Assembly Instructions.

```
main:
    li t0, 42           # t0 = 42
    move a0, t0         # Set up argument
    call addOne         # Call the function

    li v0, 0            # MIPS: v0; RISC-V: a0
    return              # Return from main

addOne:
    addi a0, a0, 1      # a0 = a0 + 1
    move v0, a0         # MIPS: v0; RISC-V: a0
    return              # Return to caller
```

1.2.5 The Problem with Nested Function Calls

What happens when one function calls another function? Consider this example:

```
main:
    li t0, 42           # t0 = 42
    move a0, t0         # Set up argument
    call addTwo         # Call addTwo function

    li v0, 0            # MIPS: v0; RISC-V: a0
    return              # Return from main

addTwo:
    call addOne         # First call to addOne
    move a0, v0         # MIPS: v0; RISC-V: a0
    call addOne         # Second call to addOne
    return              # Return to caller

addOne:
    addi a0, a0, 1      # a0 = a0 + 1
    move v0, a0         # MIPS: v0; RISC-V: a0
    return              # Return to caller
```

This code creates an infinite loop! Here's why:

1. When `main` calls `addTwo`, the `ra` register is set to point to an address inside `main` (the instruction after the call)
2. When `addTwo` calls `addOne`, the `ra` register is **overwritten** to point to an address inside `addTwo`
3. When `addOne` returns, it jumps back to `addTwo` (correct so far)
4. When `addTwo` calls `addOne` again, `ra` is overwritten once more
5. When `addTwo` tries to return, `ra` still points to an address inside `addTwo`, not back to `main`
6. The program is now stuck in an infinite loop, unable to return to `main`

The problem is that the `ra` register can only hold **one** return address at a time. When a function makes a nested call, it loses its own return address! This is why we need the **stack** to save the `ra` register before making nested function calls.

1.2.6 Fixing Nested Function Calls with a Stack

We can fix the infinite loop problem by using a stack with call frames. **Call frames** are *data frames for pending function calls*. We will save the `ra` register in the call frame of the callee on the stack when we make nested calls.

MIPS and RISC-V provide a dedicated register, sp, that always points to the the address of the latest call frame. Here's the corrected version:

```
main:
    li t0, 42           # t0 = 42
    move a0, t0         # Set up argument
    call addTwo         # Call addTwo function

    li v0, 0            # MIPS: v0; RISC-V: a0
    return              # Return from main (But use 'exit(0)' in practice.)

addTwo:
    # Prologue - save ra on the stack
    addi sp, sp, -4     # Allocate 4 bytes (1 word) on stack
    store ra, 0(sp)     # Save return address

    call addOne         # First call to addOne
    move a0, v0         # MIPS: v0; RISC-V: a0
    call addOne         # Second call to addOne

    # Epilogue - restore ra from the stack
    load ra, 0(sp)      # Restore return address
    addi sp, sp, 4      # Deallocate stack frame

    return              # Return to caller

addOne:
    addi a0, a0, 1      # a0 = a0 + 1
    move v0, a0         # MIPS: v0; RISC-V: a0
    return              # Return to caller
```

Why this fixes the problem:

1. When `main` calls `addTwo`, `ra` points back to `main`
2. In the **prologue**, `addTwo` saves this return address on the stack before making any nested calls

3. When `addTwo` calls `addOne`, `ra` is overwritten to point inside `addTwo` (this is fine - we saved the original!)
4. `addOne` returns to `addTwo`
5. When `addTwo` calls `addOne` again, `ra` is overwritten again
6. `addOne` returns to `addTwo` a second time
7. In the **epilogue**, `addTwo` restores the original `ra` value from the stack, which points back to `main`
8. Now when `addTwo` returns, it correctly returns to `main`

The stack allows us to save and restore the return address, enabling proper nested function calls. This is a fundamental pattern: **any function that calls another function must save its `ra` register first**.

In general, a call frame serves two important purposes:

1. It prevents a callee function from overwriting registers needed by the caller function - as we saw with the `ra` register, but this applies to other registers as well.
2. A function may have too many local variables or even local arrays that don't fit among the 32 registers available in the CPU. In these cases, the "overflow" must be stored in the call frame on the stack, and then loaded into registers as needed during execution.

1.3 The Stack Frame

For complex functions, we create a **call frame on the stack** - a region of the stack dedicated to that function's local variables and saved registers.

1.3.1 Stack Frame Structure

```
|                  |
+==================+  <- Stack pointer (sp) for newest call frame
|  Saved ra        |  <- Return address
+------------------+
|  Saved registers |  <- Preserved registers
+------------------+
|  Local variables |  <- Function's local data
+==================+  <- Previous stack pointer (sp)
|  Previous frame  |
|                  |
```

1.3.2 Extended Example in RISC-V Assembly, with Many Registers

Next, we will show a more realistic example, using the registers a0-a1, ra, s0-s1, t0. This program is written in RISC-V assembly language. You can paste this into the RARS simulator for RISC-V and run it.

Notice that we have two arguments (a0 and a1 registers). Also, RISC-V assembly, by convention, places the return value of a function in the a0 register. And we also use the registers s0, s1 and t0. All three act as local variables in the function `sum_of_squares()`. But we follow the conventions of **temporary registers** (t0) and **saved registers** (s0, s1).

- **Saved registers (s0, s1):** A saved register should retain its current value even if our function calls a new function that will also use the saved registers. In order to follow this convention, a saved register should always be saved in the prolog and restored in the epilog. And this code also has an example of a temporary register.

- **Temporary registers (t0, etc.):** A temporary register is "cheaper" in the sense that we don't have to save and restore it. But this means that a temporary register will not retain its original value after calling a new function, unless we explicitly save it in our call frame just before calling the new function, and then restore it later.

```
# C functions: int sum_of_squares(x, y) {return square(x) + square(y);}
#              int square(x) {return x*x;}
    li a0, 3
    li a1, 4
    call sum_of_squares  # This should return 25 (3^2 + 4^2)
    li a7, 1         # Print Integer is syscall number 1 in our emulator
                     # square() already returned its value in a0
    ecall            # Execute syscall for Print Integer
    li a7, 10        # exit() is syscall number 0
    li a0, 0         # Pass 0 as argument of exit() syscall
    ecall            # Execute syscall for exit(0)

square:              # No prolog/epilog needed.  This doesn't call a fnc.
    mul a0, a0, a0   # a0 - a0*a0
    ret
sum_of_squares:
    # PROLOG: Save ra (return address) and s0, s1 (saved registers)
    addi sp, sp, -12 # Allocate a 12-byte call frame on the stack
    sw ra, 0(sp)     # Save return address
    sw s0, 4(sp)     # Save s0 in case the caller of this fnc uses it
    sw s1, 8(sp)     # Save s1 in case the caller of this fnc uses it

    mv s1, a1        # Move a1 to saved register, in case square() uses a1
    # Call square(a0)
    call square      # RISC-V: a0 = square(x); MIPS: v0 = square(x)
    # We need to save return value of square(x), before calling square(y)
    mv s0, a0        # RISC-V: mv s0, a0; MIPS: move s0, v0
    mv a0, s1        # a0 is the argument for square; RISC-V: mv a0, s1
    call square      # Call square(y); return value is a0 (RISC-V) or v0 (MIPS)
    add t0, s0, a0   # RISC-V: add t0, s0, a0; MIPS: add t0, s0, v0
    mv a0, t0        # return val: RISC-V: mv a0, t0; MIPS: move v0, t0

    # EPILOG: Resw registers and return
    lw ra, 0(sp)     # Restore return address
    lw s0, 4(sp)     # Restore s0 in case the caller of this fnc uses it
    lw s1, 8(sp)     # Restore s1 in case the caller of this fnc uses it
    addi sp, sp, 12  # Deallocate the call frame
    ret
```

1.4 Key Principles Summary

1. **The stack enables modular programming** - It allows functions to save and restore state
2. **Push/pop are LIFO** - Last In, First Out order must be maintained
3. **jal and jr implement function calls** - jal saves the return address, jr jumps back
4. **Use calling conventions** - Follow standard rules for parameter passing and return values
5. **Save what you modify** - If a function uses a register, save it first
6. **Prologue and epilogue** - Every function needs setup and cleanup code

7. **Stack frames enable recursion** - Each call gets its own space on the stack

Understanding stacks and subroutines reveals how all high-level function calls work at the machine level. Every time you call a function in C or Java, the compiler generates assembly code that manages the stack, saves registers, passes parameters, and handles returns - exactly as you've learned here!

1.5 Practice Exercise

Try implementing these functions in assembly:

1. `max(a, b)` - Returns the maximum of two numbers
2. `sum_array(array, size)` - Returns the sum of all elements in an array
3. `fibonacci(n)` - Returns the nth Fibonacci number (try both recursive and iterative versions!)

Each of these will help you practice subroutine calls, parameter passing, and stack management.