Chapter 2: Assembly Language

Gene Cooperman

Copyright © 2026 Gene Cooperman, gene@ccs.neu.edu
This text may be copied as long as the copyright notice remains and no text is modified.

Contents

1 Introduction to Assembly Programming **2**
1.1 Variables: r0, r1, ..., r31 . . . 2
1.2 Assignment statement . . . 2
1.3 Control Flow ("if-else", etc.) . . . 4
1.4 Arrays . . . 5
1.5 Functions . . . 6
1.6 Extra comments . . . 6

2 Functions in Assembly **6**
2.1 Introduction . . . 6
2.2 Register Naming Conventions and Aliases . . . 7
2.3 The Stack Frame . . . 10
2.4 Key Principles Summary . . . 12
2.5 Practice Exercise . . . 13

3 Practical Assembly: Real Instructions and Conventions **13**
3.1 Introduction . . . 13
3.2 Practicing with an Assembly Simulator . . . 13
3.3 From Generic to Real Assembly Instructions . . . 14
3.4 The Three Uses of Labels . . . 15
3.5 System Calls (Syscalls) . . . 16
3.6 Assembler Directives . . . 17
3.7 Summary . . . 20
3.8 APPENDIX: Other Complete Examples: . . . 20

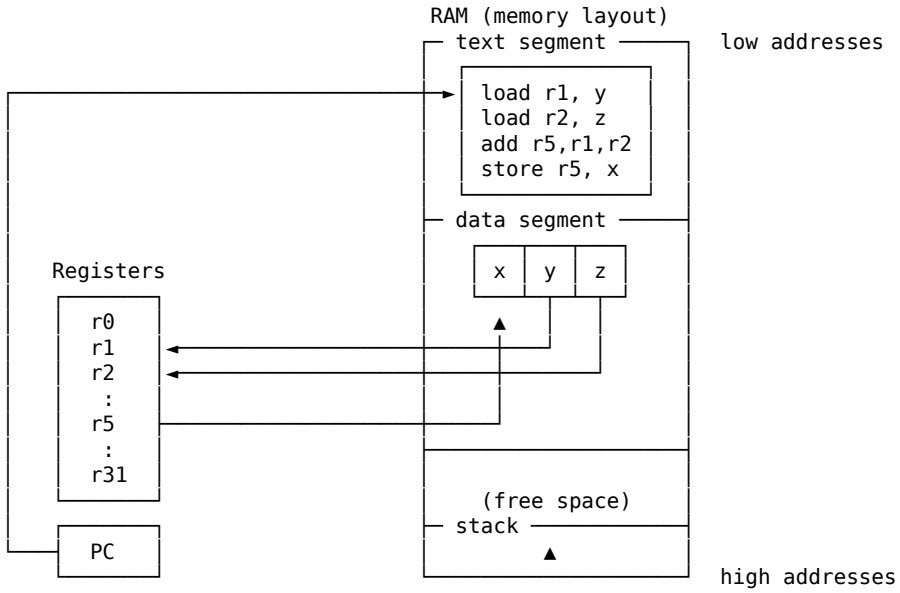
4 APPENDIX: Details and Examples of Assembly Programming **24**
4.1 Registers: The CPU's Hardware Variables . . . 25
4.2 Instruction Families . . . 25
4.3 Putting It Together: Examples . . . 26
4.4 Variables and Memory: From C to Assembly . . . 27
4.5 Flow Control: If-Else, While, and For Loops . . . 29
4.6 Working with Arrays . . . 33
4.7 Key Takeaways . . . 37

1 Introduction to Assembly Programming

This exposition will rely on your knowledge of high-level language. The goal of this chapter is to treat "Assembly" as if it is just one more high-level language. As with any new, high-level languages, we will first describe the syntax of variables, then control flow (if-else, etc.), then arrays, and finally functions. This section will not be the final, complete description of Assembly Language. But it will give you enough of an understanding so that in just a few pages of study, you can start to write real assembly programs. The following sections will then fill out the details of what a sophisticated assembly program looks like.

1.1 Variables: r0, r1, ..., r31

After compiling source code to assembly language, the variables of the source code will be stored either in the data segment (global variables) or the stack (local variables), just as you would have seen in a first class on programming. But a modern CPU typically has only 32 variables, called registers, and stored in the hardware of the CPU. So, the compiler generates assembly language instructions that load a source code variable to a CPU register from its home in RAM: `x`, `y` and `z` in the data segment in our example. It is possible to do operations such as add, subtract, etc., only within the CPU, using its CPU registers. Afterward, a register can be stored back to its home in RAM. Below, see the example diagram illustrating the source code statement: `x = y + z;`



For now, we will assume a typical 32-bit CPU. In this case, the default size for a "box" (register, int variable, or single assembly instruction) is 32 bits or 4 bytes. Notice that assembly uses the terms "load" and "store" because we implicitly imagine ourselves as sitting on top of the CPU registers and "loading from RAM" or "storing to RAM".

1.2 Assignment statement

Now that our assembly language has variables, let's again visit the assembly code from the previous diagram, which was compiled from the source code "`x=y+z;`". Here is the way that it will appear in your assembly file. (An assembly file will often have type `.s` in Linux, but sometimes `.S` in Linux or `.asm` in Windows.) Notice that we use "`#`" as the comment character in assembly. There's a reason that Python (and bash shell scripts) use "`#`" for comments.

```
    load r1, y
    load r2, z
    add r5, r1, r2  # x = y + z
    store r5, x
```

But of course, we have not initialized these variables. So, y and z will have "garbage". Let's initialize y and z. For this, we need to tell the assembler which lines are code and should be in the text segment and which lines refer to the data segment.

```
.text
    load r1, y
    load r2, z
    add r5, r1, r2  # x = y + z
    store r5, x
.data
 x: .word 0  # int x; (but .word requires an initial value)
 y: .word 17 # int y = 17;
 z: .word 12 # int z = 12;
```

Now we have told the assembler to create an x box, a y box, and a z box. These are global variables, since they are in the data segment. (Local variables would be stored in a call frame in the stack segment.) We have also initialized y and z. (NOTE: Most assemblers do not care about indentation.)

The x, y, and z in the data segment are called "labels". A *label* is a constant address where the given assembly instruction or data will be located.

Next, an assembler usually executes two passes. First, it reads the assembly code, and fills out a *Symbol Table* with the addresses for all labels (i.e., symbols). Below, is an example symbol table.

Symbol (variable, fnc, etc.)	Address
x	*data* + 0x0
y	*data* + 0x4
z	*data* + 0x8

Note, the reference to *data*, the beginning of the data segment. This is so the data segment (and hence the address *data*) can be loaded into memory at an address to be chosen later. In fact, many CPUs reserve a special register, "gp" (global data pointer), that points to the beginning of the data segment. This will be initialized when it is decided at what address to load the data segment.

Then in the second pass, the assembler will write the assembly instructions and data into memory. So, if the assembler sees a reference to x, then in the second pass, the assembler will look up the address of x in the symbol table, and then replace the x in the instruction by the address of x. It is because of this two-pass architecture that the assembler doesn't care whether you place the ".text" first or ".data" first.

Finally, we sometimes need to write "`x=y;`" or "`x=42;`" or "`x=y+42;`" in assembly. These are written as:

```
    move r5, r1
    li r5, 42
    addi r5, r1, 42
```

In fact, the first two instructions are *pseudo-instructions*, which the assembler translates into variations of the original "`add`" or an instruction "`addi`" (add immediate), where "immediate" is assembly for a "small constant", that can be embedded in the assembly instruction word itself.

```
    add r5, r1, zero
```

```
addi r5, zero, 42
addi r5, r1, 42
```

("zero", above, is a hardwired "zero register" that can never be altered. Many CPUs set r0 to be the zero register.)

1.3 Control Flow ("if-else", etc.)

Assembly does not have if-else, while, and loop. Instead, it only has "goto". Long ago, programmers would frequently use the goto command, and the C language still supports "goto" — but maybe your instructor never told you about "goto", because he or she was hoping to stop you from using it. Now is your chance to use it again.

In fact, long ago in computer history, there was a famous paper, "Goto Considered Harmful". That paper advocated the new-fangled idea of *structured programming*: using if-else, while and loop *instead* of goto. Almost all modern programs use structured programming. (But there is at least one common exception: programming a particular state machine (finite state automaton).)

So, instead of writing:

```
if (x < y) { x = y; }
```

we will write:

```
if (x >= y) { goto done; }
x = y;
done:
```

Next, let's translate this to assembly:

```
    bge r5, r1, done  # x is r5 and y is r1
    move r5, r1  # x is r5 and y is r1
done:
```

In fact, assembly supports several variations of "goto":

Assembly	C equivalent	Meaning
`b done`	`goto done`	unconditional branch
`blt r1, r2, done`	`if (r1 < r2) goto done`	"branch on less than"
`ble r1, r2, done`	`if (r1 <= r2) goto done`	"branch on less than or equal"
`bgt r1, r2, done`	`if (r1 > r2) goto done`	"branch on greater than"
`bge r1, r2, done`	`if (r1 >= r2) goto done`	"branch on greater than or equal"
`bgtz r1, done`	`if (r1 > 0) goto done`	"branch on greater than zero"

Other common variants are: beq, bne, bltz, blez, bgez, beqz and bnez.

NOTE: Because a CPU circuit should be small and fast, they don't necessarily define all of the assembly instructions above. For example,

```
bge r5, r1, done
```

and

```
ble r1, r5, done
```

do the same thing. So, many CPUs implement only one of the above, and the assembler then translates the remaining pseudo-instruction to the other assembly operation that is actually supported by the CPU.

1.4 Arrays

Recall from the discussion of C pointers in the file "A-SYS-PROG-C-pointers" that an array name can be thought of as a constant pointer. For that reason, the original C compiler did not directly implement arrays. Instead it translated something like "x[2]" to "*(x+2)".

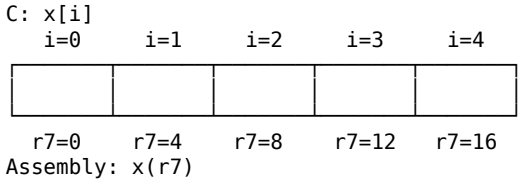
Assembly language also uses that same strategy. Suppose we want to execute "`r8=x[2]`" (load register `r8` with the array element `x[2]`). Let's assume that we have an array in our data segment, occupying 16 bytes:

```
.data
  x: .space 16
```

We need to load the address of `x` in the data segment into a register, before we can load `x[2]` itself.

```
    li r7, 8    # Here, "la" means "load address" of x
    lw r8, x(r7)
```

In `x(r7)`, `x` is the base address of the array "x", and we add the offset `r7` (with value 8 (two 4-byte words), to arrive at the address "x+8". Then load it into `r8` ("`r8 = *(x+2)`" in C, where 2 words equal 8 bytes), which is the same as "`r8 = x[2]`".



Note that most assemblers will accept two forms for "`lw`", according to whether we wish to reference an array element or a scalar (e.g., int).
Array element:

```
    lw r8, 8(r7) # for arrays: r7 is the base address and 8 is the byte offset
    lw r8, x(r7) # for arrays: x is the base address and r7 is the byte offset
```

Scalar:

```
    lw r8, x # for scalars: load address of label x; equivalent to x(zero)
```

And here is how we might initialize a register to set up "`x[2]`".

```
    la r7, x     # Here, "la" means "load address" of x
    lw r8, 8(r7) # r7 is the "base address" and 8 is the "offset"

    li r7, 8     # Here, "la" means "load address" of x
    lw r8, x(r7) # x is the base address and "r7" is the "offset"
```

And how does the assembler know how to assemble "`lw r8, x`" into binary? For this, we must remember the symbol table. The variable `x` is actually the name of a symbol, and the symbol table allows us to look up the address of `x`. We then load that address into `r8`.

Note that in order to execute "`la r7, x`" at runtime, the CPU never has to touch RAM. Instead, the assembler has translated this into something like the following instruction in binary: "load the address *data*+0 into r7". The address "*data*+0" was found as a lookup in the assembler's symbol table.

Let's now increment and store back:

```
.data
 x: .space 16
.text
    la r7, x      # Here, "la" means "load address" of x
```

```
lw r8, 8(r7)
addi r8, r8, 1
sw r8, 8(r7)
```

1.5 Functions

(For a more complete description, see "Functions in Assembly".)

In order to implement functions, we must decide how to define a function, and how to call it:
a. **Define foo(x,y) -> x+y**
b. **Call foo(x,y)**

a. **Define foo(x,y) -> x+y**

Special argument registers: `a0`, ..., `a3` The caller places the arguments here and receives the return value in register `v0`. CHECK MIPS and v0 section. MIPS uses v0 and v1, and RISC re-uses a0 and a1 for doubles. In the "Common Register Aliases" indicate the return value for RISC-V

```
move r2, a0 # x
move r3, a1 # y
add v0, r2, r3  # Assume v0 is a special register for the return value
ret # Return (Some assemblers require "ret ra".)
```

b. **Call foo**

```
call foo
```

or

```
jal foo # jump-and-link foo
```

HERE, WE DO: i. link to pc+4 into ra ii. jump to foo

NOTE: This scheme works well until we want to define two functions, `foo` and `bar`, and to have `foo` call `bar`. Then, when we call `foo`, `ra` is set to the outer return address. Then, when we call `bar`, `ra` is set to the inner return address. Then `bar` returns to `foo`. *But then we have a problem:* We want to return from `foo`, and we have lost the outer return address. We will solve this problem (and therefore the problem of recursive functions) in the Section on **Fixing Nested Function Calls with a Stack**.

1.6 Extra comments

POSSIBLY 3 types of words: assembly instruction, scalar data, address (pointer)

2 Functions in Assembly

2.1 Introduction

In the previous section, 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

We will use generic names that find their counterparts in most assembly instruction sets. We will also occasionally show examples in which we translate between our generic names in assembly and the real names in two particular assembly instruction sets: RISC-V and MIPS. This is analogous to when we study algorithms. We first use pseudo-code to express the algorithm *conceptually,* and we later translate the pseudo-code to the implementation language of our choice (e.g., C, C++ or Java).

2.2 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.

2.2.1 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)	-
`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 (same as s0)	Frame pointer	Callee

(Note that in the above table, RISC-V uses t0-t6 and s0-s11.)

2.2.2 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`)
ret	`jr $ra`	`ret` (or `jalr zero, ra, 0`)
load	`lw ...`	`lw ...`
store	`sw ...`	`sw ...`
return value register	`$v0-$v1`	`a0-a1`

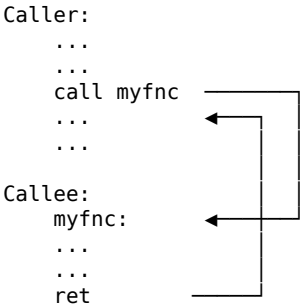
(There are two return value registers in case we want to return a value of type "double".)

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 "$" to introduce a register, and RISC-V does not. For the sake of writing just one version of the code, we will omit "$".

2.2.3 How Call and Return Work

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



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.

2.2.4 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 later in this chapter, in the section **From Generic to Real Assembly Instructions**.

```
main:
    li t0, 42           # t0 = 42
    move a0, t0         # Set up argument
    call addOne         # Call the function
                        # MIPS: return value in v0; RISC-V: a0

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

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

2.2.5 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
                        # MIPS: return value in v0; RISC-V: a0

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

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

addOne:
    addi a0, a0, 1      # a0 = a0 + 1
    move v0, a0         # MIPS: v0; RISC-V: a0
    ret                 # 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.

2.2.6 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 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
                        # MIPS: return value in v0; RISC-V: a0

    li v0, 0            # MIPS syscall argument: v0; RISC-V arg.: a0
```

```
    ret                 # Return from main (But use 'exit(0)' in practice.)

addTwo:
    # Prolog - 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

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

    ret                 # Return to caller

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

ADD DIAGRAM HERE ABOUT MEMORY LAYOUT WITH ADDRESS FROM 0 TO 4GB, AND WHY WE NEED -4 FOR THE STACK CALL FRAME, AND HOW A CALL FRAME CAN BE VIEWED AS AN ARRAY OF REGISTERS TO BE SAVED. HENCE, WE USE 0(sp) [THE ARRAY NOTATION] AS IF THE sp IS THE BASE ADDRESS OF AN ARRAY OF WORDS SAVED IN THAT CALL FRAME.

Why this fixes the problem:

1. When `main` calls `addTwo`, `ra` points back to `main`
2. In the **prolog**, `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 **epilog**, `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:

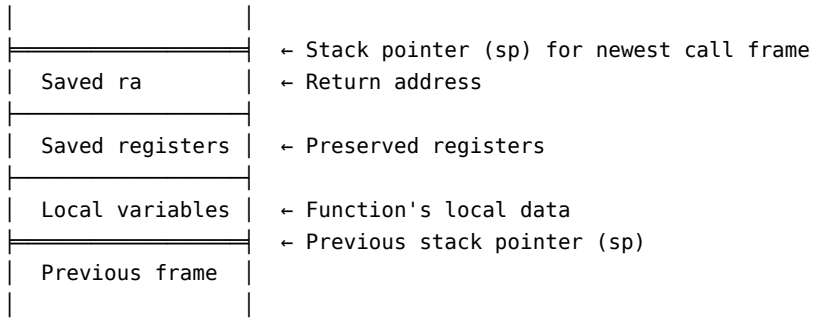
1. It prevents a callee function from overwriting registers needed by the caller function - as we saw with the `ra` register, but this applies to other registers as well.
2. A function may have too many local variables or even local arrays that don't fit among the 32 registers available in the CPU. In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

2.3 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.

2.3.1 Stack Frame Structure

MERGE THIS INTO NEW MEMORY LAYOUT DIAGRAM



2.3.2 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
                     # sum_of_squares() already returned its value in a0
    ecall            # Execute syscall for Print Integer
    li a7, 10        # exit() is syscall number 10
    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: Restore 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
```

2.4 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 (or commonly, call and ret for RISC-V)
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 (but t0-t9 are *temporary registers*, and so an exception)
6. **Prolog and epilog** - Every function normally 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!

2.5 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.

3 Practical Assembly: Real Instructions and Conventions

3.1 Introduction

In the previous sections, we used generic assembly instructions to teach fundamental concepts. Now let's transition to real assembly programming by learning: - Actual instruction names in MIPS and RISC-V - Register naming conventions and their purposes - The three uses of labels in assembly - System calls for interacting with the operating system - Common assembler directives

3.2 Practicing with an Assembly Simulator

Before diving into the details, it's important to know that you can practice assembly programming using free simulators. There are excellent Java-based simulators available for both MIPS and RISC-V assembly languages:

MARS (MIPS Assembler and Runtime Simulator)
URL: https://computerscience.missouristate.edu/mars-mips-simulator.htm

RARS (RISC-V Assembler and Runtime Simulator)
URL: https://github.com/TheThirdOne/rars

These simulators provide a complete Integrated Development Environment (IDE) for assembly programming. Key features include:

- **Register Display:** View all register values in real-time as your program executes
- **Memory Display:** Inspect RAM contents at any address, watching how data changes
- **Single-Stepping:** Execute one instruction at a time to observe exactly what each instruction does
- **Breakpoints:** Set breakpoints to pause execution at specific instructions
- **Continue Execution:** Run the program at full speed until it hits a breakpoint or completes
- **Reverse Execution:** Step backward to undo the last instruction - incredibly useful for debugging!

The typical workflow is straightforward: open a new file in the editor, write your assembly code, save it to disk, then go to the "Run" menu and click "Assemble" to convert your code to machine instructions, and finally click "Go" to execute your program.

These tools make learning assembly much easier because you can see exactly what's happening inside the CPU as each instruction executes. We highly recommend downloading one of these simulators and trying out the examples in this section.

3.3 From Generic to Real Assembly Instructions

3.3.1 Instructions We've Covered

In the earlier sections, we used generic names for clarity. Here are the more common real-world equivalents:

lw (Load Word) - More common than generic "load":

```
lw r1, 0(r2)        # Load word from address (r2 + 0) into r1
lw r1, 8(r2)        # Load word from address (r2 + 8) into r1
```

sw (Store Word) - More common than generic "store":

```
sw r1, 0(r2)        # Store word from r1 to address (r2 + 0)
sw r1, 12(r2)       # Store word from r1 to address (r2 + 12)
```

li (Load Immediate) - Load a constant value into a register:

```
li r1, 42           # r1 = 42 (constant)
li r2, 0x1000       # r2 = 4096 (hex constant)
```

move or **mov** - Copy value from one register to another:

```
move r2, r1         # r2 = r1 (copy)
# or
mov r2, r1          # Same thing (syntax varies by architecture)
```

3.3.2 Instruction Comparison: MIPS vs RISC-V

Here's a comparison of common instructions in MIPS and RISC-V assembly:

Operation	MIPS	RISC-V	Description
Load word	`lw`	`lw`	Load 32-bit word from memory
Store word	`sw`	`sw`	Store 32-bit word to memory
Load address	`la`	`la`	Load address of label (pseudo-instruction)
Load immediate	`li`	`li`	Load constant into register (pseudo-instruction)
Move register	`move`	`mv`	Copy from one register to another
Add	`add`	`add`	Add two registers
Add immediate	`addi`	`addi`	Add register and constant
Subtract	`sub`	`sub`	Subtract two registers
Multiply	`mul`	`mul`	Multiply two registers
Divide	`div`	`div`	Divide two registers
AND	`and`	`and`	Bitwise AND
OR	`or`	`or`	Bitwise OR
Branch equal	`beq`	`beq`	Branch if two registers are equal
Branch not equal	`bne`	`bne`	Branch if two registers are not equal
Branch less than	`blt`	`blt`	Branch if first < second
Branch greater/equal	`bge`	`bge`	Branch if first >= second
Jump	`j`	`j`	Unconditional jump to label
Jump and link	`jal`	`jal`	Call function (save return address)
Jump register	`jr`	`jr`	Return from function (with ra)
System call	`syscall`	`ecall`	Invoke operating system service

Note: Some instructions like `li`, `la`, and `move/mv` are pseudo-instructions - the assembler translates them into one or more real instructions.

3.4 The Three Uses of Labels

In assembly language, labels serve three distinct purposes:

1. Variable names in the data segment
2. Branch/jump targets in the code (text segment)
3. Function names (entry points) in the code (text segment)

3.4.1 Variable Names in the Data Segment

Labels mark the location of variables and data:

```
.data
count:      .word 0         # Label 'count' refers to this variable
message:    .asciz "Hello" # Label 'message' refers to this string
array:      .word 1, 2, 3   # Label 'array' refers to this array

.text
    lw t0, count            # Access the variable 'count'
    la a0, message          # Load address of string 'message'
```

3.4.2 Targets for Conditional Branches

Labels mark positions in code where branches can jump:

```
.text
    li t0, 5
    li t1, 10

    blt t0, t1, less_than   # If t0 < t1, jump to 'less_than'

    # t0 >= t1
    li a0, 0
    j done

less_than:                  # Label marks the branch target
    li a0, 1

done:                       # Label marks end of if-else
    # Continue program...
```

3.4.3 Function Names

Labels mark the entry point of functions:

```
.text
main:                       # Label 'main' is the function name
```

```
    li a0, 5
    jal factorial           # Call function 'factorial'
    j exit

factorial:                  # Label 'factorial' is the function name
    # Function implementation...
    jr ra

exit:
    li v0, 10               # Exit syscall
    syscall
```

Summary: Labels are simply named locations in memory - for data, for branch targets, or for function entry points. The assembler replaces each label with its actual memory address.

3.5 System Calls (Syscalls)

Programs interact with the operating system through **system calls** for tasks like I/O, file operations, and memory allocation.

3.5.1 MIPS System Calls

Instruction: `syscall`

Register Convention: - `v0` - Syscall number (which service to invoke) - `a0-a3` - Arguments to the syscall - `v0` - Return value (for syscalls that return data)

Example: Print an integer in MIPS

```
li v0, 1            # Syscall 1 = print integer
li a0, 42           # Argument: the integer to print
syscall             # Invoke OS
```

Example: Read an integer in MIPS

```
li v0, 5            # Syscall 5 = read integer
syscall             # After this, v0 contains the input
move s0, v0         # Save input to s0
```

3.5.2 RISC-V System Calls

Instruction: `ecall` (Environment Call)

Register Convention: - `a7` - Syscall number (which service to invoke) - `a0-a5` - Arguments to the syscall - `a0` - Return value (for syscalls that return data)

Example: Print an integer in RISC-V

```
li a7, 1            # Syscall 1 = print integer
li a0, 42           # Argument: the integer to print
ecall               # Invoke OS
```

Example: Read an integer in RISC-V

```
li a7, 5            # Syscall 5 = read integer
ecall               # After this, a0 contains the input
mv s0, a0           # Save input to s0
```

3.5.3 System Call Comparison

Architecture	Instruction	Syscall Number Register	Argument Registers	Return Value Register
MIPS	`syscall`	`v0`	`a0-a3`	`v0`
RISC-V	`ecall`	`a7`	`a0-a5`	`a0`

Note: In RISC-V, notice that `a0` is used both for the first argument AND for the return value. In MIPS, `v0` holds the syscall number before the call and the return value after.

3.5.4 Common Syscall Numbers

While syscall numbers vary by operating system, here are some common examples, as used by the MARS and RARS simulator. Real Linux may differ. Also, MARS/RARS ignores the a0 exit code for "Exit".

Service	MIPS v0	RISC-V a7	Arguments	Returns
Print integer	1	1	a0 = integer	-
Print string	4	4	a0 = string address	-
Read integer	5	5	-	v0/a0 = integer
Exit	10	10	a0 = exit code	-

3.6 Assembler Directives

Beyond `.data`, `.text`, and `.word`, assemblers support many directives for organizing your program.

3.6.1 Common Directives

- `.data`, `.text` - Segment markers
- `.word`, `.byte`, `.half` - Data allocation
- `.ascii`, `.asciz` - Strings (`.asciz` appends a null char as terminator)
- `.space` - Uninitialized buffers
- `.float`, `.double` - Floating-point values
- `.globl` - Export labels for linking
- `.align` - Memory alignment

NOTE: In the MARS simulator for MIPS, they define `.asciiz` instead of `.asciz`. RARS and MARS both define `.ascii`.

3.6.2 Segment Directives

.data - Begin data segment (variables and constants):

`.data`

.text - Begin code segment (instructions):

`.text`

3.6.3 Data Allocation Directives

.word - Allocate and initialize 4-byte word(s):

```
x:      .word 42        # Single word initialized to 42
array:  .word 1, 2, 3   # Array of three words
```

.byte - Allocate and initialize byte(s):

```
flag:   .byte 1         # Single byte
chars:  .byte 'A', 'B'  # Multiple bytes
```

.half or **.short** - Allocate 2-byte half-word(s):

```
small:  .half 100       # 2-byte value
```

.space - Allocate uninitialized space (in bytes):

```
buffer: .space 100      # Reserve 100 bytes (not initialized)
```

3.6.4 String Directives

.ascii - Store ASCII string without null terminator:

```
msg:    .ascii "Hello"  # 5 bytes: 'H', 'e', 'l', 'l', 'o'
```

.asciz - Store ASCII string with null terminator:

```
msg:    .asciz "Hello" # 6 bytes: 'H', 'e', 'l', 'l', 'o', '\0'
```

The **"z"** in `.asciz` means append an **ASCII zero** (the null character `\0`) at the end of the string. This is the standard C-style string format, where strings are terminated with a zero byte.

3.6.5 Floating-Point Directives

.float - Allocate single-precision (4-byte) floating-point:

```
pi:     .float 3.14159
```

.double - Allocate double-precision (8-byte) floating-point:

```
e:      .double 2.71828
```

3.6.6 Code Organization Directives

.globl or **.global** - Make a label visible to other files (for linking):

```
.globl main         # 'main' can be called from other files

main:
    # Program starts here
```

.align - Align the next data/instruction to a power-of-2 boundary:

```
.align 2            # Align to 2^2 = 4-byte boundary (word-aligned)
.align 3            # Align to 2^3 = 8-byte boundary (double-word-aligned)
```

3.6.7 Example: Complete Program with Directives

```
# Complete program demonstrating common directives

.data
    .align 2                        # Word-align data section
    prompt:     .asciz "Enter a number: "
    result_msg: .asciz "Result: "
    newline:    .asciz "\n"

    number:     .word 0             # Storage for input
    array:      .word 1, 2, 3, 4, 5 # Array of 5 integers

    buffer:     .space 100          # 100-byte buffer
    pi_value:   .float 3.14159      # Floating-point constant

.text
    .globl main                     # Make main visible to linker

main:
    # Print prompt
    li v0, 4                        # MIPS: syscall 4 = print string
    la a0, prompt
    syscall

    # Read integer
    li v0, 5                        # MIPS: syscall 5 = read integer
    syscall
    move s0, v0                     # Save input

    # Double the input
    add s1, s0, s0                  # s1 = s0 * 2

    # Print result message
    li v0, 4
    la a0, result_msg
    syscall

    # Print result value
    li v0, 1                        # MIPS: syscall 1 = print integer
    move a0, s1
    syscall

    # Print newline
    li v0, 4
    la a0, newline
```

```
syscall

# Exit
li v0, 10                       # MIPS: syscall 10 = exit
syscall
```

3.7 Summary

3.7.1 Key Instructions

- `lw`/`sw` for loading/storing words
- `li` for loading constants
- `move`/`mv` for copying between registers
- `la` for loading addresses

3.7.2 Register Conventions

- **Arguments:** `a0-a3` (MIPS), `a0-a7` (RISC-V)
- **Return values:** `v0-v1` (MIPS), `a0-a1` (RISC-V)
- **Temporaries:** `t0-t9` (MIPS), `t0-t6` (RISC-V) (caller-save)
- **Saved:** `s0-s7` (MIPS), `s0-s11` (RISC-V) (callee-save)
- **Return address:** `ra`
- **Stack pointer:** `sp`

3.7.3 Three Uses of Labels

1. Variable names in data segment
2. Branch/jump targets in code
3. Function entry points

3.7.4 System Calls

- **MIPS:** Use `syscall`, syscall number in `v0`, args in `a0-a3`, return in `v0`
- **RISC-V:** Use `ecall`, syscall number in `a7`, args in `a0-a5`, return in `a0`

These practical details bridge the gap between conceptual assembly and real code. While MIPS and RISC-V have slight differences, the underlying principles are universal across all assembly languages!

3.8 APPENDIX: Other Complete Examples:

The following examples are suitable for copy-pasting into a file, and then opening it into the RARS simulator and executing.

3.8.1 Complete Example: Factorial Function

Let's implement a recursive factorial function: `n! = n × (n-1)!`

C code:

```c
int factorial(int n) {
    if (n <= 1) {
        return 1;
    }
    return n * factorial(n - 1);
}

int result = factorial(5);  // result = 120
```

Assembly:

```
# int factorial(int n)
# Recursive implementation
# Argument:  n = a0
# Returns:   n! in a0

.data
msg_n:      .asciz "n = "
msg_result: .asciz "n! = "
newline:    .asciz "\n"
    .text
    .globl main

main:
    addi    sp, sp, -8          # allocate stack space
    sw      ra, 0(sp)           # save return address
    sw      s0, 4(sp)           # save s0
    li      s0, 5               # n = 5
    # Print "n = 5\n"
    li      a7, 4               # ecall 4: print string
    la      a0, msg_n
    ecall
    li      a7, 1               # ecall 1: print integer
    mv      a0, s0
    ecall
    li      a7, 4               # ecall 4: print string
    la      a0, newline
    ecall
    # Call factorial(5)
    mv      a0, s0              # n = 5
    call    factorial            # a0 = factorial(5) = 120
    mv      s0, a0              # save result in s0
    # Print "n! = 120\n"
    li      a7, 4
    la      a0, msg_result
    ecall
    li      a7, 1
    mv      a0, s0
    ecall
    li      a7, 4
    la      a0, newline
    ecall
```

```
    # Restore saved registers
    lw      ra, 0(sp)
    lw      s0, 4(sp)
    addi    sp, sp, 8
    # Exit
    li      a7, 10              # ecall 10: exit
    ecall

factorial:
    addi    sp, sp, -8          # allocate stack space
    sw      ra, 0(sp)           # save return address
    sw      s0, 4(sp)           # save s0
    mv      s0, a0              # save n in s0
    # Base case: if n <= 1, return 1
    li      t0, 1
    ble     a0, t0, .Lbase
    # Recursive case: return n * factorial(n - 1)
    addi    a0, a0, -1          # a0 = n - 1
    call    factorial            # a0 = factorial(n - 1)
    mul     a0, s0, a0          # a0 = n * factorial(n - 1)
    j       .Lreturn
.Lbase:
    li      a0, 1               # return 1
.Lreturn:
    lw      ra, 0(sp)           # restore return address
    lw      s0, 4(sp)           # restore s0
    addi    sp, sp, 8           # deallocate stack space
    ret
```

3.8.2 Tracing the Factorial Execution

When we call `factorial(5)`, here's what happens:

1. Call `factorial(5)` - pushes frame, saves ra and n=5
2. Since 5 > 1, call `factorial(4)` - pushes new frame, saves ra and n=4
3. Since 4 > 1, call `factorial(3)` - pushes new frame, saves ra and n=3
4. Since 3 > 1, call `factorial(2)` - pushes new frame, saves ra and n=2
5. Since 2 > 1, call `factorial(1)` - pushes new frame, saves ra and n=1
6. Base case! Return 1
7. Return to `factorial(2)`: 2 × 1 = 2, return 2
8. Return to `factorial(3)`: 3 × 2 = 6, return 6
9. Return to `factorial(4)`: 4 × 6 = 24, return 24
10. Return to `factorial(5)`: 5 × 24 = 120, return 120

Each call creates its own stack frame, allowing the recursion to work correctly!

3.8.3 Complete Example: Computing Power (x^n)

Let's write a non-recursive function that computes x raised to the power n.

C code:

```c
int power(int x, int n) {
    int result = 1;
    for (int i = 0; i < n; i++) {
        result = result * x;
    }
    return result;
}

int p = power(2, 8);  // p = 256
```

```
.data
msg_x:      .asciz "x = "
msg_n:      .asciz "n = "
msg_result: .asciz "result = "
newline:    .asciz "\n"

    .text
    .globl main

# int power(int x, int n)
# Arguments: x = a0, n = a1
# Returns:   result in a0

main:
    addi    sp, sp, -12         # allocate stack space
    sw      ra, 0(sp)           # save return address
    sw      s0, 4(sp)           # save s0 (will hold x)
    sw      s1, 8(sp)           # save s1 (will hold n)

    li      s0, 2               # x = 2
    li      s1, 8               # n = 8

    # Print "x = 2\n"
    li      a7, 4               # ecall 4: print string
    la      a0, msg_x
    ecall
    li      a7, 1               # ecall 1: print integer
    mv      a0, s0
    ecall
    li      a7, 4               # ecall 4: print string
    la      a0, newline
    ecall

    # Print "n = 8\n"
    li      a7, 4
    la      a0, msg_n
    ecall
    li      a7, 1
    mv      a0, s1
    ecall
    li      a7, 4
    la      a0, newline
```

```
    ecall

    # Call power(2, 8)
    mv      a0, s0              # x = 2
    mv      a1, s1              # n = 8
    call    power               # a0 = power(2, 8) = 256

    mv      s0, a0              # save result in s0

    # Print "result = 256\n"
    li      a7, 4
    la      a0, msg_result
    ecall
    li      a7, 1
    mv      a0, s0
    ecall
    li      a7, 4
    la      a0, newline
    ecall

    # Restore saved registers
    lw      ra, 0(sp)
    lw      s0, 4(sp)
    lw      s1, 8(sp)
    addi    sp, sp, 12

    # Exit
    li      a7, 10              # ecall 10: exit
    ecall

power:
    li      t0, 1               # result = 1
    ble     a1, zero, .Ldone    # if n <= 0, skip loop

.Lloop:
    mul     t0, t0, a0          # result = result * x
    addi    a1, a1, -1          # n--
    bnez    a1, .Lloop          # loop while n != 0

.Ldone:
    mv      a0, t0              # move result into return register
    ret
```

4 APPENDIX: Details and Examples of Assembly Programming

(This section again starts from beginning concepts, in a longer version of the previous section on Introduction to Assembly Programming*.)*

Assembly language directly translates to machine code, which is understood natively by a CPU chip. When you write assembly code, you're working at the lowest level of programming that's still human-readable.

An **assembler** takes your human-readable assembly instructions and assembles them into binary instructions that are then executed by the CPU. An assembler is similar to a compiler, except that most assembly instructions are directly converted into their binary representation. Often, one assembly instruction is converted into one 4-byte instruction, called a **word**.

In fact, when you compile C code, the compiler first compiles your source code to assembly, and then assembles it into binary. (Some compilers skip the intermediate step and compile directly to binary, but the principle remains the same.)

Throughout this chapter, we assume you are already comfortable programming in a high-level language such as C, and we build on that experience: wherever possible, each new assembly concept is related back to the equivalent idea in C or Java.

We proceed in three stages. In Section 1, we introduce the basics of assembly using a *generic* assembly syntax, so that we can concentrate on the underlying concepts rather than the quirks of any one processor. Section 2 adds how function calls and functions are implemented. Section 3 then takes the RISC-V (and MIPS) assembly languages and translates our generic assembly into a real assembly language — complete with a simulator for running assembly programs, system calls, and the assembler directives that describe the data and text memory segments.

4.1 Registers: The CPU's Hardware Variables

A typical, modern CPU has 32 hardware variables, built directly into the chip and called **registers**. They are the CPU's own variables, and they might be named `r0`, `r1`, `r2`, ..., `r31`. Because they are physically part of the CPU, registers are extremely fast to access — far faster than RAM.

Registers are where computation actually happens. Consider an assignment statement in a high-level language, with an arithmetic or logical expression on its right-hand side. To carry out that statement, the variables appearing in it are assigned to registers, and the arithmetic or logic is then performed on those registers.

A program in a high-level language, however, may have far more than 32 variables. This is why assembly also includes **load** and **store** instructions. The variables of the original source code have their permanent *home* in RAM: global variables live in the **data segment** of memory, and local variables live on the **stack**. One *loads* a variable from RAM into a register before executing an assignment statement, and one *stores* the variable from its register back to its home in RAM once it is no longer being used.

4.2 Instruction Families

Assembly instructions are typically organized into families of related operations. Let's explore two fundamental families:

4.2.1 1. Arithmetic and Logic Operations

These instructions perform computations directly on register values. Common operations include:

- Arithmetic: `add`, `sub`, `mul`, `div`
- Logic: `and`, `or`, `not`, `xor`

Examples: - `add r3, r1, r2` — Add the values in registers `r1` and `r2`, store the result in `r3` - `mul r5, r2, r4` — Multiply values in `r2` and `r4`, store result in `r5`

Note that `r3` (or any destination register) can be the same as one of the source registers: - `add r1, r1, r2` — Add `r2` to `r1`, storing the result back in `r1` (equivalent to `r1 += r2` in C)

Immediate Operations:

The "i" suffix stands for **immediate**, meaning the instruction uses a small constant value: - `addi r3, r1, 5` — Add the constant 5 to `r1`, store result in `r3` - `muli r2, r2, 10` — Multiply `r2` by 10, store result back in `r2`

Why must the constant be small? Because the constant, along with the register numbers, must all be encoded immediately in a single 4-byte word. There simply isn't room for large constants!

4.2.2 2. Load and Store Operations

These instructions move data between registers and RAM (main memory).

Important perspective: Always imagine you are sitting on top of the CPU. From this viewpoint: - **Load** means: load data **from** RAM **into** a register (bringing data to where you sit) - **Store** means: store data **into** RAM **from** a register (sending data away from where you sit)

Examples: - `load r1, addressInRam` — Read a word from RAM at the given address into register `r1` - `store r1, addressInRam` — Write the word in register `r1` to RAM at the given address

4.3 Putting It Together: Examples

Let's see how these instructions work in practice. Remember, in these examples we're using simplified syntax for clarity.

4.3.1 Example 1: Simple Arithmetic

```
# Compute: r3 = (r1 + r2) * 10
add r3, r1, r2      # r3 = r1 + r2
muli r3, r3, 10     # r3 = r3 * 10
```

In C, this would be equivalent to:

```c
r3 = (r1 + r2) * 10;
```

4.3.2 Example 2: Working with Memory

```
# Load two values from memory, add them, store the result
load r1, 1000       # Load word from RAM address 1000 into r1
load r2, 1004       # Load word from RAM address 1004 into r2
add r3, r1, r2      # r3 = r1 + r2
store r3, 1008      # Store result to RAM address 1008
```

In C, if we had pointers:

```c
int* p1 = (int*)1000;
int* p2 = (int*)1004;
int* p3 = (int*)1008;
*p3 = *p1 + *p2;
```

4.3.3 Example 3: Computing an Expression

```
# Compute: result = (a + 5) * (b - 3)
# Assume 'a' is in r1, 'b' is in r2

addi r3, r1, 5      # r3 = a + 5
addi r4, r2, -3     # r4 = b - 3  (or use 'subi r4, r2, 3' if available)
mul r5, r3, r4      # r5 = r3 * r4
```

In C:

```
result = (a + 5) * (b - 3);
```

4.3.4 Example 4: Array Access Pattern

```
# Load array[0] and array[1], compute their sum
# Assume array base address is 2000

load r1, 2000       # r1 = array[0]
load r2, 2004       # r2 = array[1] (next word, 4 bytes later)
add r3, r1, r2      # r3 = array[0] + array[1]
store r3, 2008      # store result at array[2]
```

In C:

```
int array[3];
array[2] = array[0] + array[1];
```

4.4 Variables and Memory: From C to Assembly

In C or Java, when you declare a variable, the compiler automatically allocates memory for it:

```
int a = 5;
int b = 10;
int x;
```

You don't need to worry about *where* in RAM these variables are stored - the compiler handles that for you.

In assembly, we achieve something similar using a **data segment** (typically marked with `.data`). Instead of hardcoding memory addresses like `3000` or `3004`, we use **labels** as symbolic names for memory locations. The assembler then assigns actual addresses to these labels.

Here's how we declare variables in assembly:

```
.data                   # Begin data segment
a:      .word 5         # Declare variable 'a', initialize to 5
b:      .word 10        # Declare variable 'b', initialize to 10
c:      .word 7         # Declare variable 'c', initialize to 7
x:      .word 0         # Declare variable 'x', initialize to 0
```

The `.word` directive tells the assembler to allocate one word (4 bytes) of memory. The label (like `a:` or `x:`) gives that memory location a name we can use in our code.

Now, instead of using numeric addresses, we can reference variables by name. And we need to declare this section as `.text`, so that these instructions are placed in the text segment, and not in the data segment.

```
.text
    load r1, a              # Load the value of 'a' into r1
    load r2, b              # Load the value of 'b' into r2
    add r3, r1, r2          # r3 = a + b
    store r3, x             # Store result in 'x'
```

This is much more readable and maintainable! The assembler will replace `a`, `b`, and `x` with their actual memory addresses during assembly.

4.4.1 Example 5: The Quadratic Formula (Revised with Labels)

Let's implement the quadratic formula using proper variable labels: $x = (-b + \sqrt{b^2 - 4ac})/2a$

First, we declare our variables in the data segment:

```
.data
a:      .word 2         # Coefficient a
b:      .word 5         # Coefficient b
c:      .word 3         # Coefficient c
x:      .word 0         # Result x
```

Now the code section with our calculations:

```
.text                   # Begin code segment
# Quadratic formula: x = (-b + sqrt(b^2 - 4ac)) / 2a
# Load the coefficients
load r1, a              # r1 = a
load r2, b              # r2 = b
load r3, c              # r3 = c

# Compute b^2
mul r4, r2, r2          # r4 = b * b = b^2

# Compute 4ac
addi r5, r0, 4          # r5 = 4 (assuming r0 is always 0, or use muli if avail.)
mul r5, r5, r1          # r5 = 4 * a
mul r5, r5, r3          # r5 = 4a * c = 4ac

# Compute b^2 - 4ac (the discriminant)
sub r6, r4, r5          # r6 = b^2 - 4ac

# Compute sqrt(b^2 - 4ac)
sqrt r7, r6             # r7 = sqrt(r6) ; but most assembly doesn't provide sqrt

# Compute -b
sub r8, r0, r2          # r8 = 0 - b = -b (assuming r0 = 0)

# Compute -b + sqrt(b^2 - 4ac)
add r9, r8, r7          # r9 = -b + sqrt(b^2 - 4ac)

# Compute 2a
add r10, r1, r1        # r10 = a + a = 2a (or use muli r10, r1, 2)
```

```
# Compute the final result: (-b + sqrt(...)) / 2a
div r11, r9, r10        # r11 = r9 / r10 = x

# Store the result
store r11, x            # Store x using the label, not a numeric address
```

Note: If your assembly language lacks a `sqrt` instruction, you would pass the argument to an assembly subroutine (function), which would compute the square root and pass back the result into the destination register. We'll cover subroutines in a later section.

Compare this to the C version:

```
int a = 2;
int b = 5;
int c = 3;

float x = (-b + sqrt(b*b - 4*a*c)) / (2*a);
```

The key advantage of using labels is **readability and maintainability**. If you need to reorganize your data segment or add new variables, you don't have to manually recalculate and update all the numeric addresses throughout your code. The assembler does that work for you, just like a compiler does for high-level languages.

4.5 Flow Control: If-Else, While, and For Loops

In C and Java, you use `if`, `while`, and `for` to control program flow. Assembly doesn't have these high-level constructs. Instead, it uses **branch** and **jump** instructions, along with **comparison** instructions.

4.5.1 The Building Blocks

First, we need instructions to compare values and branch based on the result:

Branch instructions (conditional jumps):

- `beq r1, r2, label` — Branch if Equal: jump to label if r1 == r2
- `bne r1, r2, label` — Branch if Not Equal: jump to label if r1 != r2
- `blt r1, r2, label` — Branch if Less Than: jump to label if r1 < r2
- `ble r1, r2, label` — Branch if Less than or Equal: jump to label if r1 <= r2
- `bgt r1, r2, label` — Branch if Greater Than: jump to label if r1 > r2
- `bge r1, r2, label` — Branch if Greater than or Equal: jump to label if r1 >= r2

Unconditional jump: - `j label` — Jump unconditionally to label (like `goto` in C)

4.5.2 Translating If-Else

Consider this C code:

```
if (x > 5) {
    y = 10;
} else {
    y = 20;
}
```

In assembly:

```
    load r1, x          # Load x
    addi r2, r0, 5      # r2 = 5
    ble r1, r2, else_branch   # If x <= 5, jump to else

    # Then branch (x > 5)
    addi r3, r0, 10     # r3 = 10
    store r3, y         # y = 10
    j end_if            # Skip the else branch

else_branch:
    # Else branch (x <= 5)
    addi r3, r0, 20     # r3 = 20
    store r3, y         # y = 20

end_if:
    # Continue with rest of program
```

Key insight: We branch to the `else` when the condition is **false** (when x <= 5), not when it's true!

4.5.3 Translating While Loops

Consider this C code:

```
int i = 0;
while (i < 10) {
    sum = sum + i;
    i = i + 1;
}
```

In assembly:

```
    addi r1, r0, 0      # r1 = i = 0
    load r2, sum        # r2 = sum
    addi r3, r0, 10     # r3 = 10

while_loop:
    bge r1, r3, end_while   # If i >= 10, exit loop

    # Loop body
    add r2, r2, r1      # sum = sum + i
    addi r1, r1, 1      # i = i + 1

    j while_loop        # Jump back to loop condition

end_while:
    store r2, sum       # Store final sum
```

Pattern: 1. Check condition at the top 2. Branch out if condition is false 3. Execute loop body 4. Jump back to condition check

4.5.4 Translating For Loops

Consider this C code:

```c
for (int i = 0; i < 10; i++) {
    sum = sum + i;
}
```

A `for` loop is really just a `while` loop with initialization and increment built in:

```
    addi r1, r0, 0      # i = 0 (initialization)
    load r2, sum        # r2 = sum
    addi r3, r0, 10     # r3 = 10

for_loop:
    bge r1, r3, end_for     # If i >= 10, exit loop

    # Loop body
    add r2, r2, r1      # sum = sum + i

    # Increment
    addi r1, r1, 1      # i++

    j for_loop          # Jump back to condition check

end_for:
    store r2, sum       # Store final sum
```

4.5.5 Example: Finding the Maximum of Two Numbers

C code:

```c
int a = 15;
int b = 23;
int max;

if (a > b) {
    max = a;
} else {
    max = b;
}
```

Assembly:

```
.data
a:      .word 15
b:      .word 23
max:    .word 0

.text
    load r1, a          # r1 = a
    load r2, b          # r2 = b

    ble r1, r2, b_is_max    # If a <= b, jump to b_is_max

    # a is max
    store r1, max       # max = a
```

```
    j done

b_is_max:
    store r2, max       # max = b

done:
    # Continue program
```

4.5.6 Example: Sum of First N Numbers

C code:

```c
int n = 10;
int sum = 0;
int i = 1;

while (i <= n) {
    sum = sum + i;
    i = i + 1;
}
```

Assembly:

```
.data
n:      .word 10
sum:    .word 0

.text
    load r1, n          # r1 = n
    addi r2, r0, 0      # r2 = sum = 0
    addi r3, r0, 1      # r3 = i = 1

loop:
    bgt r3, r1, end_loop    # If i > n, exit loop

    add r2, r2, r3      # sum = sum + i
    addi r3, r3, 1      # i = i + 1

    j loop              # Repeat

end_loop:
    store r2, sum       # Store final sum
```

4.5.7 Key Principles

1. **Labels mark positions** in code where you can jump to
2. **Branch instructions compare and jump:** Use instructions like `blt r1, r2, label` to compare two registers and branch based on the result
3. **Invert the logic:** Branch when the condition is **false** to skip code blocks
4. **Always jump back** at the end of a loop to recheck the condition
5. **Think in gotos:** High-level control flow compiles down to labels and jumps

Understanding flow control in assembly helps you appreciate what compilers do automatically. Every `if`, `while`, and `for` in your C code becomes a series of comparisons, branches, and jumps!

4.6 Working with Arrays

Arrays are fundamental data structures, and understanding how they work in assembly reveals what's happening behind the scenes in C and Java.

4.6.1 Understanding Array Indexing in Assembly

Most assembly languages support an addressing mode written as:

```
load r1, offset(r2)     # Load from address: (r2 + offset)
store r1, offset(r2)    # Store to address: (r2 + offset)
```

This introduces a new addressing mode, "constant(register)". Here's how to think about this:

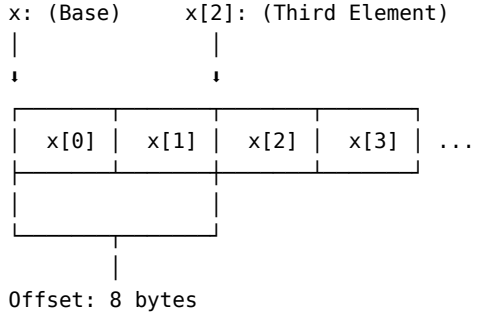
- **register** (r2) contains the **base address** - where the array begins in RAM
- **offset** (offset) is the **byte offset** from that base address
- The actual memory address accessed is: base address + offset

For example, if r2 contains address 2000:

- `load r1, 0(r2)` loads from address 2000 (first element)
- `load r1, 4(r2)` loads from address 2004 (second element, 4 bytes later)
- `load r1, 8(r2)` loads from address 2008 (third element)

Since each word is 4 bytes, to access array[i], we use offset = i × 4.

Now, let's assume that the array x begins at address 2000 in the data segment, and assume that we have loaded the address x into register r2. Let's re-examine `load r1, 8(r2)`, but this time visually.



In this example, x is the base address and 8 is the offset. In assembly language, we might load `x[2]` into `r1` as follows.

```
la r2, x            # Load the base address, x, into r2, using la (load address)
load r1, 8(r2)      # Load from address (r2 + 8), where 8 is the offset
```

4.6.2 Understanding Array Indexing: From C to Assembly

In C, when you write `x[i]` for an integer array, you're accessing the element at index `i`. Let's break down what this really means:

Array names are addresses: In C, an array name like `x` can be viewed as a constant pointer - it's the address where the array begins in memory.

Array indexing is pointer arithmetic: The expression `x[i]` is actually equivalent to `*(x + i)` in pointer notation. This means: - Start at address `x` - Add `i` (the compiler knows to multiply by the element size) - Dereference to get the value

4.6.3 Translating Array Access to Assembly

Let's see how `x[i]` translates to assembly. Consider loading `x[i]` into register `r2`:

In C:

```
int x[10];
int i = 3;
int value = x[i];
```

A naive assembly translation might look like:

```
load r1, i          # r1 = i
muli r1, r1, 4      # r1 = i * 4 (convert index to byte offset)
load r2, x(r1)      # r2 = x[i]  (pseudo-instruction)
```

However, `load r2, x(r1)` is typically a **pseudo-instruction**. It looks simple, but the assembler actually decomposes it into multiple real instructions:

```
la r6, x            # r6 = address of x (load address, not value!)
load r1, i          # r1 = i
muli r1, r1, 4      # r1 = i * 4 (convert index to byte offset)
add r6, r6, r1      # r6 = address of x + (i * 4)
load r2, 0(r6)      # r2 = value at address in r6
```

Let's understand each step:

1. `la r6, x` - **Load Address**: This copies the *address* of `x` (not its value) into `r6`. If `x` starts at memory address 2000, then `r6` becomes 2000.
2. `load r1, i` - Load the index value into `r1`
3. `muli r1, r1, 4` - Multiply by 4 because each integer is 4 bytes. This converts the index to a byte offset.
4. `add r6, r6, r1` - Add the offset to the base address, giving us the address of `x[i]`
5. `load r2, 0(r6)` - Load the value from that address

4.6.4 The `la` (Load Address) Instruction

The `la` instruction is crucial for working with arrays:

```
la r1, x        # r1 = address of x (NOT the value at x)
load r1, x      # r1 = value stored at x (loads from RAM)
```

Key difference:

- `la` (load address) - Copies a constant address into a register. No RAM access needed!
- `load` (load word) - Reads a value from RAM at the given address

Think of it this way:

- `la r1, x` is like `r1 = &x` in C (address-of operator)
- `load r1, x` is like `r1 = x` in C (value access)

4.6.5 Constant Array Indexing: A Simpler Case

When the index is a compile-time constant, like `x[2]`, the translation is simpler:

C code:

```
int x[10];
int value = x[2];
```

Assembly:

```
la r1, x            # r1 = address of x
load r2, 8(r1)      # r2 = x[2] (offset = 2 * 4 = 8 bytes)
```

Since the index is constant (2), we can compute the byte offset at compile time ($2 \times 4 = 8$), and use it directly in the `offset(register)` addressing mode.

4.6.6 Declaring Arrays in the Data Segment

Here's how we declare an array in assembly:

```
.data
myarray:    .word 17, 24, 3, 5, 18, 11, 6, 47, 22, 4
arraysize:  .word 10
```

This creates an array of 10 integers in consecutive memory locations. The label `myarray` points to the first element.

4.6.7 Complete Example: Variable vs Constant Indexing

Let's compare both approaches:

```
.data
x:      .word 10, 20, 30, 40, 50
i:      .word 3

.text
# Example 1: Constant index - x[2]
la r1, x            # r1 = address of x
load r2, 8(r1)      # r2 = x[2] (offset 8 = 2*4)
                    # r2 now contains 30

# Example 2: Variable index - x[i] where i=3
la r1, x            # r1 = address of x
load r3, i          # r3 = i = 3
muli r3, r3, 4      # r3 = 3 * 4 = 12 (byte offset)
add r1, r1, r3      # r1 = address of x[3]
load r4, 0(r1)      # r4 = x[3]
                    # r4 now contains 40
```

4.6.8 Example: Bubble Sort Algorithm

Let's implement bubble sort on our array. Bubble sort repeatedly steps through the array, compares adjacent elements, and swaps them if they're in the wrong order.

C code:

```c
int myarray[] = {17, 24, 3, 5, 18, 11, 6, 47, 22, 4};
int n = 10;

// Bubble sort
for (int i = 0; i < n - 1; i++) {
    for (int j = 0; j < n - i - 1; j++) {
        if (myarray[j] > myarray[j + 1]) {
            // Swap
            int temp = myarray[j];
            myarray[j] = myarray[j + 1];
            myarray[j + 1] = temp;
        }
    }
}
```

Assembly implementation:

```
.data
myarray:    .word 17, 24, 3, 5, 18, 11, 6, 47, 22, 4
arraysize:  .word 10

.text
    la r1, myarray      # r1 = base address of array
    load r2, arraysize  # r2 = n = 10
    addi r3, r0, 0      # r3 = i = 0 (outer loop counter)

outer_loop:
    addi r4, r2, -1     # r4 = n - 1
    bge r3, r4, done    # If i >= n - 1, sorting done

    addi r5, r0, 0      # r5 = j = 0 (inner loop counter)

inner_loop:
    # Compute n - i - 1
    sub r6, r2, r3      # r6 = n - i
    addi r6, r6, -1     # r6 = n - i - 1

    bge r5, r6, end_inner   # If j >= n - i - 1, exit inner loop

    # Compute byte offset for array[j]: j * 4
    add r7, r5, r5      # r7 = j * 2
    add r7, r7, r7      # r7 = j * 4 (byte offset)

    # Load array[j] and array[j+1]
    add r8, r1, r7      # r8 = address of array[j]
    load r9, 0(r8)      # r9 = array[j]
```

```
    load r10, 4(r8)     # r10 = array[j+1]

    # Compare array[j] and array[j+1]
    ble r9, r10, no_swap    # If array[j] <= array[j+1], no swap needed

    # Swap array[j] and array[j+1]
    store r10, 0(r8)    # array[j] = array[j+1]
    store r9, 4(r8)     # array[j+1] = temp (original array[j])

no_swap:
    addi r5, r5, 1      # j++
    j inner_loop        # Repeat inner loop

end_inner:
    addi r3, r3, 1      # i++
    j outer_loop        # Repeat outer loop

done:
    # Array is now sorted!
```

4.6.9 Understanding the Array Indexing

The trickiest part is computing the byte offset. Since each integer is 4 bytes: - Array[0] is at offset 0 - Array[1] is at offset 4 - Array[2] is at offset 8 - Array[j] is at offset j × 4

In the code above, we compute `j × 4` by doubling twice:

```
add r7, r5, r5      # r7 = j + j = j * 2
add r7, r7, r7      # r7 = (j * 2) + (j * 2) = j * 4
```

Alternatively, if we had a shift instruction:

```
sll r7, r5, 2       # Shift left logical by 2 bits: r7 = j * 4
```

Or with multiply:

```
muli r7, r5, 4      # r7 = j * 4
```

4.6.10 Key Takeaways for Arrays

1. **Arrays are contiguous memory** - elements are stored one after another
2. **Base address + offset** - use `offset(register)` to access array elements
3. **Calculate byte offsets** - for word arrays, element i is at offset i × 4
4. **Load address vs load value** - `la` gets the address, `load` gets the value
5. **Manual index arithmetic** - you must compute offsets yourself, unlike C's automatic `array[i]` translation

Working with arrays in assembly shows you exactly what the compiler does when you write `array[i]` in C!

4.7 Key Takeaways

1. **Assembly is low-level:** You work directly with the CPU's native instructions
2. **Registers are precious:** You only have a fixed number (often 32). Use them wisely
3. **Operations are simple:** Each instruction does one small thing (add two numbers, load one word, etc.)

4. **Memory access is explicit:** Unlike high-level languages, you must explicitly load/store between RAM and registers
5. **Labels in assembly:** A label in the data segment is usually the address of a word. A label in the text segment is usually the target of a conditional branch. (And we will soon see that it may also be the name of a function.)
6. **Types in assembly:** Assembly does not have types in the sense of C, C++ or Java. A word might contain bits representing an int, a float, or even four char's.
7. **Values versus Addresses in assembly:** Assembly does distinguish between the value of a word and the address of a word. In `load x` and `store x`, we are referring to the *value* at the address `x`. In `la x`, we are referring to the *address* of `x` (similar to &x in the C language). (And we will soon see that if a label is the name of a function, then we are referring to the *address* in the text segment where the function begins.)
8. **Instruction names vary:** While we used generic names like `add` and `load`, your particular assembly language (x86, ARM, MIPS, RISC-V, etc.) may use different names for the same concepts
9. **Complex expressions require planning:** Breaking down formulas like the quadratic equation shows how high-level languages do a lot of work behind the scenes

The beauty of assembly is its simplicity and directness. The challenge is managing all the details that high-level languages handle for you automatically!