Chapter 2a: Introduction to Assembly Language

Gene Cooperman

Contents

1 Introduction to Assembly Programming

1.1 What is Assembly Language?

Assembly language is the language that 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 or Java 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.)

1.2 Registers: The CPU's Hardware Variables

In assembly programming, the CPU has a fixed number of hardware variables built directly into the chip, called **registers**. Think of registers as the CPU's personal, ultra-fast storage locations. A common design is to have 32 registers, typically named `r0`, `r1`, `r2`, ... `r31`.

Unlike variables in C or Java that live in RAM, registers are physically part of the CPU itself, making them extremely fast to access. However, there are only a limited number of them, so you must use them carefully.

1.3 Instruction Families

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

1.3.1 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!

1.3.2 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

1.4 Putting It Together: Examples

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

1.4.1 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:

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

1.4.2 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:

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

1.4.3 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);
```

1.4.4 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];
```

1.5 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.

1.5.1 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)
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)

# 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
addi 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 lecture.

Compare this to the C version:

```c
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.

1.6 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.

1.6.1 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)

1.6.2 Translating If-Else

Consider this C code:

```c
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!

1.6.3 Translating While Loops

Consider this C code:

```c
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

1.6.4 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
```

1.6.5 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
```

1.6.6 Example: Sum of First N Numbers

C code:

```
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
```

1.6.7 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!

1.7 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.

1.7.1 Understanding Array Addressing 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:

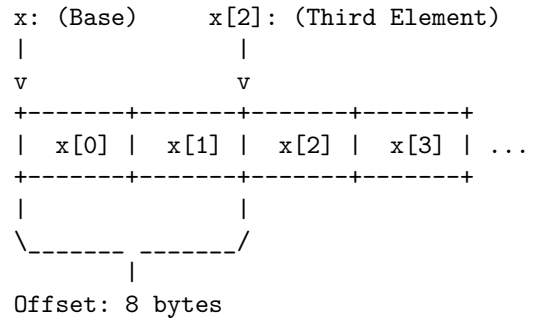
- **register** (r2) contains the **base address** - where the array begins in RAM
- **offset** (offset) is the **byte offset** from that base address
- The actual memory address accessed is: base address + offset

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 $\times$ 4.

Now, let's assume that we have set `x: .word 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 register 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
```

1.7.2 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

1.7.3 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
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

1.7.4 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)

1.7.5 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.

1.7.6 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.

1.7.7 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
```

1.7.8 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
    addi r7, r5, r5     # r7 = j * 2
    addi 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!
```

1.7.9 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 $\times$ 4

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

```
addi r7, r5, r5     # r7 = j + j = j * 2
addi 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
```

1.7.10 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 $\times$ 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!

1.8 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!