Chapter 2c: Assembly Programming in Practice

Gene Cooperman

Contents

1 Practical Assembly: Real Instructions and Conventions

1.1 Introduction

In previous lectures, 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

1.2 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

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 lecture.

1.3 From Generic to Real Assembly Instructions

1.3.1 Instructions We've Covered

In earlier lectures, 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)
```

1.3.2 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

Operation	MIPS	RISC-V	Description
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`	`jalr` (with `x0`)	Return from function
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.

1.4 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)

1.4.1 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'
```

1.4.2 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...
```

1.4.3 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.

1.5 System Calls (Syscalls)

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

1.5.1 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
```

1.5.2 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
```

1.5.3 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.

1.5.4 Common Syscall Numbers

While syscall numbers vary by operating system, here are some common examples:

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	-

1.6 Assembler Directives

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

1.6.1 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`.

1.6.2 Segment Directives

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

```
.data
```

.text - Begin code segment (instructions):

```
.text
```

1.6.3 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)
```

1.6.4 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.

1.6.5 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
```

1.6.6 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)
```

1.6.7 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
```

1.7 Summary

1.7.1 Key Instructions

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

1.7.2 Register Conventions

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

1.7.3 Three Uses of Labels

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

1.7.4 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!

1.8 APPENDIX: Other Complete Examples:

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

1.8.1 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
```

1.8.2 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)`: $2 \times 1 = 2$, return 2
8. Return to `factorial(3)`: $3 \times 2 = 6$, return 6

9. Return to `factorial(4)`: $4 \times 6 = 24$, return 24
10. Return to `factorial(5)`: $5 \times 24 = 120$, return 120

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

1.8.3 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
```

Assembly:

```
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
```