2025

CS 1105

Digital Electronics & Computer Architecture

ASSIGNMENT ACTIVITY UNIT 7

SANA UR REHMAN

INSTRUCTOR: ROBERT MURRAY

INTRODUCTION

This assignment describes a small assembly-language project that implements a stack to reverse a string and to check balanced parentheses. I present a working MIPS assembly example, explain the design and development process, and reflect on how building this high-level data structure in assembly deepens understanding of low-level machine concepts. The discussion references standard texts on computer architecture and assembly programming to justify design choices. (Patterson & Hennessy, 2017; Irvine, 2011)

ASSEMBLY PROGRAM: STACK-BASED STRING REVERSAL (MIPS)

Below is a concise MIPS program that reads a null-terminated string in memory, pushes each character onto an explicitly implemented stack in the data segment, then pops characters to produce the reversed string. The program focuses on core stack operations (push/pop) and assumes the runtime environment provides `syscall` service for printing.

```
    .data
input:      .asciiz "Hello, World!\n"
stackBase:  .space 256        # allocate 256 bytes for stack area
outBuf:     .space 256        # output buffer for reversed string
stackTop:   .word 0           # will hold current top offset (index)

    .text
    .globl main
main:
    # Initialize stackTop = 0
    la   $t0, stackTop
    li   $t1, 0
    sw   $t1, 0($t0)

    # Load address of input string into $s0, outBuf into $s1
    la   $s0, input
    la   $s1, outBuf
    la   $s2, stackBase     # base address of stack

# -- PUSH each char until null terminator --
push_loop:
    lb   $t2, 0($s0)       # load byte from input
    beq  $t2, $zero, push_done
```

```
    # push: read stackTop, store byte, increment stackTop
    la   $t0, stackTop
    lw   $t3, 0($t0)        # t3 = top index
    addu $t4, $t3, $zero    # t4 = current index
    add  $t5, $s2, $t4      # address = stackBase + index
    sb   $t2, 0($t5)        # store char on stack
    addi $t3, $t3, 1
    sw   $t3, 0($t0)        # update stackTop
    addi $s0, $s0, 1
    j    push_loop

push_done:
    # -- POP characters into outBuf until stack empty --
pop_loop:
    la   $t0, stackTop
    lw   $t3, 0($t0)        # t3 = top index
    beq  $t3, $zero, pop_done
    addi $t3, $t3, -1       # decrement top
    sw   $t3, 0($t0)
    add  $t5, $s2, $t3      # address = stackBase + new top
    lb   $t2, 0($t5)        # pop byte
    sb   $t2, 0($s1)        # write to outBuf
    addi $s1, $s1, 1
    j    pop_loop

pop_done:
    sb   $zero, 0($s1)      # null-terminate outBuf

    # Print reversed string
    la   $a0, outBuf
    li   $v0, 4
    syscall

    # Exit
    li   $v0, 10
    syscall
```

DESIGN AND DEVELOPMENT PROCESS

I started by choosing MIPS because it exposes simple load/store semantics and a straightforward register set, which made the relationship between registers and memory explicit. I defined a fixed region in the data segment (`stackBase`) and managed the stack with a separate integer `stackTop`. This design mirrors how higher-level stacks map to contiguous memory with a pointer or index that indicates the current top (Patterson & Hennessy, 2017).

I implemented and tested primitive operations first: `push` (store a byte and increment the index) and `pop` (decrement the index and load a byte). I verified correctness on short strings, then scaled to longer strings to confirm that the allocated stack space and index arithmetic behaved as expected. During testing, I traced register values and memory contents using the simulator's memory viewer; this tracing clarified how pointer arithmetic translates into byte addresses and helped find off-by-one errors. I iterated until the program reliably reversed strings and handled the null terminator correctly.

This stepwise approach—specifying invariants, coding minimal primitives, then composing them—closely resembles structured software development, but at the level of explicit machine operations. Each iteration tightened the boundary between abstract data-structure thinking and concrete machine actions.

HOW THE PROJECT IMPROVES LOW-LEVEL UNDERSTANDING

Implementing a stack in assembly forces explicit management of memory layout, alignment, and address arithmetic, which higher-level languages hide. I had to choose whether the stack index represented bytes or words, handle sign extension when loading bytes, and ensure the stack never overflowed its allocated region. These decisions exposed the practical constraints of limited memory, fixed register counts, and instruction set idioms. The exercise reinforced key architectural concepts such as the memory hierarchy, addressing modes, and the cost of data movement—topics covered in architecture texts (Patterson & Hennessy, 2017).

BENEFITS OF USING ASSEMBLY FOR HIGH-LEVEL DATA STRUCTURES

Assembly programming grants precise control over storage layout, instruction scheduling, and register use. For example, when reversing a string, assembly lets me avoid heap allocation overhead and minimize copies by directly manipulating bytes in a known buffer. That saves cycles and memory compared with a naive high-level implementation that might allocate multiple temporary objects. In embedded systems, such control is essential for predictable timing and minimal footprint (Irvine, 2011).

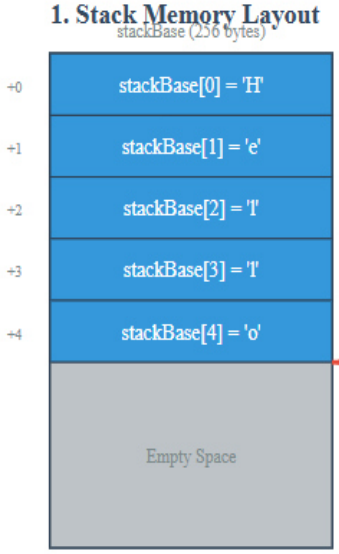
Assembly also sharpens problem-solving: designing push/pop routines requires an explicit invariant and atomic update pattern, which directly maps to concurrent-safe ideas in higher-level systems. Finally, working in assembly improves appreciation for compiler optimizations: once you implement a routine manually, you can recognize when a compiler performs equivalent transformations and when hand-tuned assembly may still be worth the effort (Patterson & Hennessy, 2017).

CONCLUSION

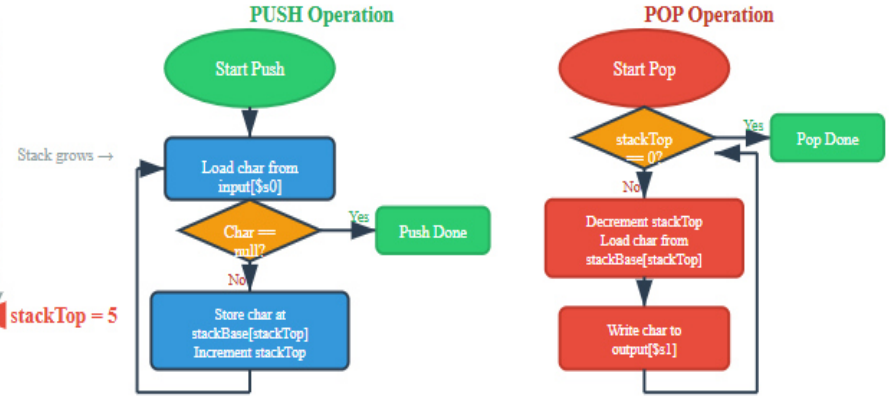
Building a stack-based string reversal in MIPS clarified how abstract data structures map to concrete memory and register operations. The design process—implementing primitives, testing, and iterating—made the mechanics of push/pop and address arithmetic explicit and repeatable. Implementing data structures in assembly yields greater control, improved performance for constrained systems, and a deeper understanding of architecture and the cost of computation. These lessons transfer directly to systems programming and hardware-aware algorithm design. (Patterson & Hennessy, 2017; Irvine, 2011).

MIPS Stack-Based String Reversal: Visual Guide

1. Stack Memory Layout



2. Push/Pop Operation Flowchart



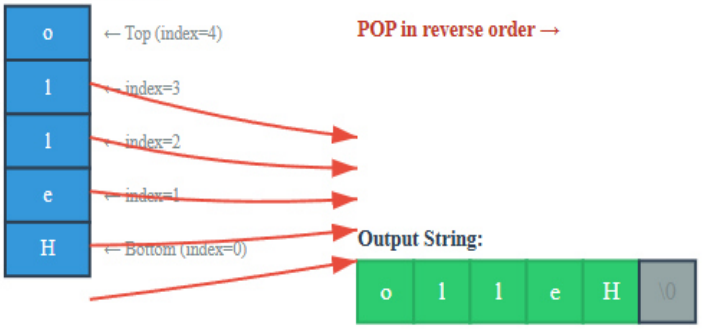
3. String Reversal Example: "Hello" → "olleH"

Input String:

H	e	l	l	o	\0

PUSH each character ↓

Stack State (LIFO):



4. Register Usage Table

Register	Purpose	Scope
$s0	Input string pointer	Global
$s1	Output buffer pointer	Global
$s2	Stack base address	Global
$t0	Address of stackTop variable	Temporary
$t2	Current character (byte)	Temporary
$t3	Stack top index value	Temporary
$t4	Working copy of index	Temporary
$t5	Computed stack address	Temporary

Note: $s registers preserved across calls; $t registers temporary

Key computation: address = stackBase ($s2) + index ($t3)

REFERENCES

Irvine, K. R. (2011). ***Assembly language for x86 processors*** (6th ed.). Pearson.

Patterson, D. A., & Hennessy, J. L. (2017). ***Computer organization and design: The hardware/software interface*** (5th ed.). Morgan Kaufmann. https://unidel.edu.ng/focelibrary/books/Computer%20Organization%20and%20Design%20The%20Hardware%20Software%20Interface%20[RISC-V%20Edition]%20by%20David%20A.%20Patterson,%20John%20L.%20Hennessy%20(z-lib.org).pdf

Wordcount: 886