ASM PROJECT

Mini Command-Line Shell

GROUP 63

E/22/014 & E/22/034

CO1020: Computer Systems Programming

2025 / 07 / 26

Table of Contents

Introduction

In this project, we built a basic command-line shell using **ARM32 Assembly Language**.

The story behind the project is that, after a worldwide disaster, our group finds an ARM32-based embedded computer. This computer is the only one left working, and we need to create a shell (a command-line interface) to communicate with it. Since no high-level language is available, we used **pure ARM32 Assembly** to build this shell.

The shell can understand simple commands typed by the user. It supports four built-in commands (hello, help, exit, clear) and two custom commands (hex, add).

This project helped us understand low-level programming, input handling, and memory management using assembly.

Commands of the Shell

The shell supports six commands such as :

I. Basic Commands - 4
II. Custom-made Commands – 2

Basic Commands

Command	Description
hello	Shows the message "Hello World!"
help	Lists all available commands
exit	Exits the shell with a goodbye note
clear	Clears the terminal screen

Custom Commands

Command	Description
hex <number>	Converts a decimal number into hexadecimal and prints it. (Example: hex 15 → 0xF)
add <num1> <num2>	Adds two decimal numbers and shows the result. (Example: add 10 5 → 15)

Implementation Details

This project was completed in one file called *shell.s*, using **ARM32 Assembly Language**. We followed the instructions step-by-step according to the given 3 tasks.

Below is the breakdown of how each part of the code was implemented.

Task 1 : Base Shell Architecture

This task focused on building the basic structure of the shell.

1. Used a Single File

   All code was written inside *shell.s*.

2. Created All Sections

   - *.data* : for constant strings and command names
   - *.bss* : for buffers (input and temporary)
   - *.text* : for the main program logic

3. Defined Entry Point

   The program starts at the main label.

4. Basic Shell Loop

   The main loop does:

   - Print the prompt " *shell>* "
   - Read input from user (up to 99 characters)
   - Remove the newline character
   - Call check_commands to identify the command
   - Repeat the loop

5. Allocated Buffers

    - *input_buffer* : stores what user types
    - *temp_buffer* : helps with printing numbers

6. Used System Calls

    - Linux syscall *write (r7 = 4)* was used for printing.
    - Linux syscall *read (r7 = 3)* was used for getting input.

Task 2 : Command Handling Logic

This part involved matching and running the right command.

1. Command Matching with String Comparison

    Function *compare_strings* was written to check if two strings are equal. It compares character by character, up to a given length.

2. Conditional Branching

    After comparing each command, we use *cmp* and *bne* to go to the next check.

```
cmp r0, #1
bne check_next_command
```

3. Reusable Functions for Each Command

    Each command has its own function:

    - *do_hello* : prints "Hello World!"
    - *do_help* : shows all commands
    - *do_exit* : prints goodbye and exits
    - *do_clear* : clears the screen

4. Preserved Registers

All functions follow proper ARM rules:

- Use *push {lr}* or *push {r4, r5, ...}* to save registers.
- Use *pop {pc}* to return safely.
- This prevents breaking the main loop or overwriting data accidentally.

5. Shell Loop Maintained

   After running any command, the code goes back to *main_loop* and repeats. The shell never crashes or exits unless the *exit* command is used.

Task 3: Custom Commands

We created two custom commands that go beyond basics:

1. hex <number>

   This command takes a number and converts it to hexadecimal.

   Steps:

   I. Skip *"hex "* (4 bytes).
   II. Parse the number using *get_number.*
   III. If valid:
      - Print *"0x"* using syscall write.
      - Call *print_hex_number* to convert and print digits.
         - ❖ Uses bitwise operations : *AND* , *LSR* to extract hex digits.
         - ❖ Reverses the digit order before printing.
         - ❖ Uses *hex_digits* lookup table to convert 0–15 → *'0'* - *'F'*.

   IV. If invalid:
      - Print *"Unknown command"*

2. add <num1> <num2>

   This command adds two numbers and prints the sum.

   Steps:

   I. Skip *"add "* (4 bytes).
   II. Read the first number using *get_number*.
   III. Use *skip_first_number* to move past the first number and spaces.
   IV. Read second number using *get_number*.
   V. If both numbers are valid:
      - Add them and call *print_number* to print result.
         - ❖ Converts integer to decimal string.
         - ❖ Uses *divide_by_ten* to get digits one by one.
         - ❖ Reverses them before printing
   VI. If any number is invalid:
      - Print *"Unknown command"*

Supporting Functions

All helpers are well-organized and used in multiple places :

Functions	Used for
compare_strings	Compares two strings up to given length
get_number	Converts ASCII to integer, returns -1 if invalid
skip_first_number	Skips first number and spaces in input
print_number	Converts number to characters and prints
print_hex_number	Converts number to hexadecimal and prints
divide_by_ten	Divides number by 10 using subtraction loop

All of them preserve the stack and registers properly.

Design Decisions

This section explains why we made certain choices in the code.

hex and *add* Commands

We picked these two because:

- *hex* shows how to work with bitwise operations and character lookup, which is common in embedded programming.
- *add* shows how to deal with parsing, multiple arguments, and basic math in assembly.

They are good examples of:

- Input processing
- Error handling
- String to number conversion
- Number formatting and output

Error Handling

We kept error handling simple but effective:

- If the command is not found → print *"Unknown command"*
- If the number is invalid (like letters instead of digits) → return *-1* from *get_number*
- If either input in *add* or *hex* is bad → fall to the error handler

Key logic used:

```
cmp r0, #-1
beq error_label
```

This makes the code strong against invalid inputs but still small and easy to understand.

Reusability of Code

To reduce repeated code, we reused the same logic in different places:

- *compare_strings* is used in all command checks
- *get_number* is reused in both *hex* and *add*
- *print_number* and *print_hex_number* are separated, so we don't duplicate printing logic
- *skip_first_number* helps keep parsing clean and avoids copying input multiple time

Register Safety

One of the most important things in assembly is to not break other parts of the code.

So :

- Every function that uses temporary registers (like *r4*, *r5*, etc.) saves them using *push* at the start
- Before returning, *pop* is used to restore them
- All leaf functions return with *pop {pc}* or equivalent

This shows proper register usage and understanding of calling conventions.

Challenges & Solutions

In this project, we faced some real problems while working with ARM32 assembly. Since assembly is a low-level language, even small tasks like reading input or printing numbers require careful thinking. Below are the main problems we faced and how we solved them:

1. Removing newline from input

Problem :

When the user types something and presses **Enter**, the system adds a newline character (*\n*, ASCII 10) at the end of the input. For example, if the user types *hello*, it becomes *hello\n* in memory. This extra newline causes command matching to fail, because *"hello\n"* is not equal to *"hello".*

Our Solution :

After using the *read* system call, the number of characters read is stored in register *r0*. We reduced this count by 1:

```
sub r0, r0, #1
```

Then we added a null terminator (*\0*) at the end of the string by doing:

```
mov r2, #0
strb r2, [r1, r0]
```

This replaces the newline with a null character and turns the input into a proper C-style string (ending with 0), which works well with our comparison function.

2. Parsing two numbers in one line

Problem :

For the *add* command, the user types two numbers in one line like this:

```
add 10 25
```

We had to read the first number (*10*) and then move forward to read the second one (*25*). This is not easy in assembly because there's no built-in function like in C.

Our Solution :

First, we used our *get_number* function to read the first number.

Then, we used a new function called *skip_first_number*:

- It moves the pointer forward until it finds a space.
- Then it skips all the spaces until it reaches the second number.

Now, we can use *get_number* again to read the second number.

This two-step system lets us read both numbers correctly from one string.

3. Converting numbers to strings

Problem :

We can easily get a number like *15* into a register, but we can't print it directly. The system call *write* can only print characters, not numbers.

Our Solution :

We wrote two separate functions:

- *divide_by_ten* : This function divides a number by 10 using subtraction and returns:
    - ❖ Quotient → the new number
    - ❖ Remainder → the current digit
- *print_number* : This function uses *divide_by_ten* in a loop.
    - ❖ Each remainder is turned into a character (like *5* → *'5'*)
    - ❖ The characters are stored in *temp_buffer*
    - ❖ After storing all digits, it prints them in reverse order.

This works for any number, even large ones like 1032.

4. Printing digits in reverse order

Problem :

When converting a number like *123*, we get digits in the reverse order: first *3*, then *2*, then *1*. But we want to print them as *123*.

Our Solution :

We saved each digit into a buffer (*temp_buffer*) starting from index 0. Then we used a loop to print the characters **from the last to the first**.

This way, we reverse the digits before printing:

```
sub r6, r6, #1
add r1, r5, r6
mov r2, #1
swi 0
```

This gave us the correct number output, in the right order.

5. Handling unknown or invalid input

Problem :

If the user typed an unknown command like *goodbye* or gave bad input like *hex ABC*, the shell would not know what to do.

Our Solution :

- We used *compare_strings* to check each known command.
- If no match was found, we used a final fallback:

```
unknown_command:
    mov r0, #1                      @ Print unknown command message
    ldr r1, =unknown_msg
    mov r2, #unknown_msg_len
    mov r7, #4
    swi 0

    pop {pc}                       @ Return to main loop
```

- For bad number input (non-numeric characters), *get_number* returns -*1*.
- If we see -*1*, we immediately go to the same error message.

This way, we handled all incorrect inputs with a simple and consistent message:

```
Unknown command
```

AI Tool Usage

We used **ChatGPT** as a **learning and support tool** during the development of this project. Here is how we used it and how we made sure our final code was fully correct.

We Used AI For :

1. Understanding assembly syntax :
   We asked how to use instructions like *SWI, PUSH*, etc.

2. Learning system calls :
   We used AI to understand how Linux syscalls like *read, write*, and *exit* work in ARM32.

3. Algorithm design ideas :
   We discussed how to convert numbers, how to print digits, and how to reverse buffers.

4. Debugging help :
   When our output was wrong or GDB showed a register mismatch, we asked AI what could be wrong.

We never copied full code from AI. Instead, we:

1. Wrote our own code by hand, based on our understanding.
2. Checked the logic step by step
3. Tested each part separately, for example:

   - Checked *get_number* with multiple inputs like *"123"*, *"a12", " "*
   - Tested *add* with positive and zero values
   - Checked hex output for edge values like *0, 255, 4095*

AI was used like a teaching assistant, not a code generator. The full code was written and debugged manually.

Individual Contributions

This project was completed by a group of two members. Both members were involved in the development and documentation process. The work was divided across different areas of the shell, and each member focused on specific tasks, with regular discussions and joint testing.

Member 1 – (E/22/014)

Task	Work Done
Shell Setup	- Created .data, .bss, and .text sections - Designed the prompt (shell>) and input buffer system - Implemented the shell loop and input handling using Linux syscalls
Command Matching	- Wrote compare_strings function - Implemented logic for checking each command using branching
Basic Commands	- Fully developed hello, help, exit, and clear commands - Handled output formatting and ANSI escape sequences
Custom Commands	- Designed and coded both hex and add commands - Wrote supporting functions: get_number, skip_first_number, print_hex_number, print_number
Error Handling	- Built error detection for invalid input in hex and add - Managed fallback message for unknown or invalid commands
Testing & Debugging	- Tested all commands thoroughly - Used GDB to debug registers and string positions - Confirmed buffer safety and correct number outputs
Documentation	- Wrote the majority of the report - Organized structure by tasks and added deep explanations - Described design decisions, challenges, and solutions in clear language

Member 2 – (E/22/034)

Task	Work Done
Code Review & Support	- Reviewed code written for hex, add, and helper functions - Helped verify stack usage and syscall parameters
Command Summary & Testing	- Helped test command cases (valid, invalid, edge cases) - Ensured shell loop continues smoothly after each command
Design Discussions	- Contributed to command selection ideas - Took part in planning structure for reusability and clarity
Final Touches	- Assisted in final review of code formatting and comments - Reviewed report draft and contributed feedback

Joint Tasks

- Decided on custom commands and command structure
- Verified output formats (e.g., *"0x"* prefix, newlines, spacing)
- Final testing of the complete shell with various test cases
- Ensured proper register usage (*push/pop*) in all functions
- Completed the final review of the report for clarity and correctness

Conclusion

We have successfully created a simple command-line shell using ARM32 assembly language.

The shell supports six commands such as :

I. Basic Commands - 4
II. Custom-made Commands – 2

By doing this project, we learned many things:

- How to use system calls to read and write data.
- How to manage registers and the stack correctly.
- How to convert numbers between decimal and hexadecimal.
- How to process user input step by step.
- How to debug assembly programs using tools like GDB.

At first, it was hard to write everything in assembly. But we kept improving the code and fixed many problems. In the end, we made a working shell that can handle different commands and errors. This project helped us understand how computers run programs at a low level.

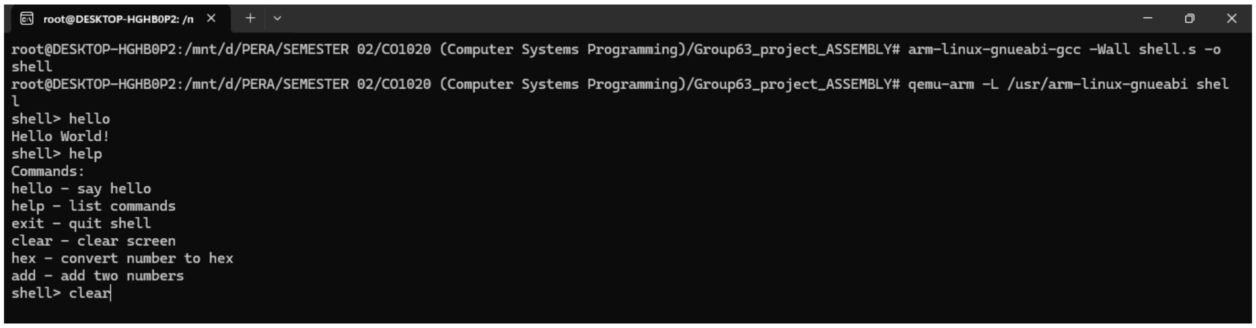
We believe this shell can be used as a base to add more features in the future.

References

1. ARM Limited, 2005. ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition. [online] Available at: https://developer.arm.com/documentation/ddi0406/latest [Accessed 27 July 2025].
2. Ganssle, J., 2000. The Art of Designing Embedded Systems. 2nd ed. Burlington: Newnes.
3. Peradeniya University, Department of Computer Engineering, 2025. CO1020: Assembly Project – Mini Command-Line Shell. [Course Document] University of Peradeniya. Provided via LMS. [Accessed 27 July 2025].
4. Linux, 2025. Linux System Call Table for ARM EABI. [online] Available at: https://chromium.googlesource.com/chromiumos/docs/+/HEAD/constants/syscalls.md#arm-eabi [Accessed 27 July 2025].
5. Patterson, D.A. and Hennessy, J.L., 2013. Computer Organization and Design: The Hardware/Software Interface. 5th ed. San Francisco: Morgan Kaufmann.

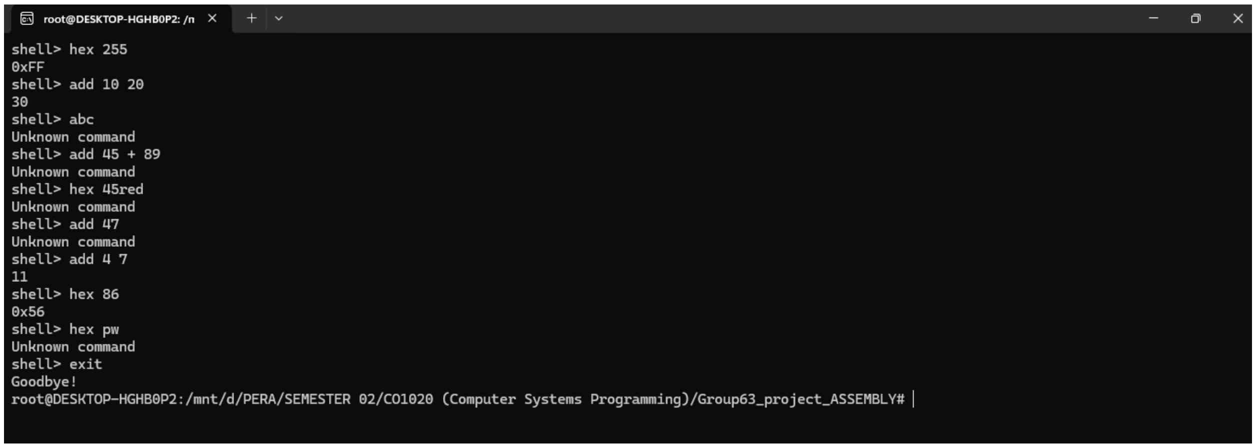
Appendices

We tested our shell using different commands. These are some example results:



```
root@DESKTOP-HGHB0P2:/mnt/d/PERA/SEMESTER 02/CO1020 (Computer Systems Programming)/Group63_project_ASSEMBLY# arm-linux-gnueabi-gcc -Wall shell.s -o
shell
root@DESKTOP-HGHB0P2:/mnt/d/PERA/SEMESTER 02/CO1020 (Computer Systems Programming)/Group63_project_ASSEMBLY# qemu-arm -L /usr/arm-linux-gnueabi shel
l
shell> hello
Hello World!
shell> help
Commands:
hello - say hello
help - list commands
exit - quit shell
clear - clear screen
hex - convert number to hex
add - add two numbers
shell> clear
```

```
shell>
```



```
shell> hex 255
0xFF
shell> add 10 20
30
shell> abc
Unknown command
shell> add 45 + 89
Unknown command
shell> hex 45red
Unknown command
shell> add 47
Unknown command
shell> add 4 7
11
shell> hex 86
0x56
shell> hex pw
Unknown command
shell> exit
Goodbye!
root@DESKTOP-HGHB0P2:/mnt/d/PERA/SEMESTER 02/CO1020 (Computer Systems Programming)/Group63_project_ASSEMBLY#
```