Lab 1 :

Basic assembly programming for RISC-V

This lab is a mini-guide that shows how to write tiny RISC-V assembly programs, explains the key instructions, and then assembles/runs them on a Linux-based RISC-V board (like your K1/MUSE Pi Pro with Bianbu/Debian), or an equivalent compatible RISC-V board.

RISC-V (RV64) uses a **clean register set** and a simple calling convention:

- **Integer argument/return registers:** `a0–a7` (`x10–x17`).
  - Function args go in `a0..a7`, return value in `a0`.
- **Temporary registers:** `t0–t6 (x5–x7, x28–x31)`.
- **Saved registers:** `s0–s11 (x8–x9, x18–x27)`. Preserve them across calls if you use them.
- **Zero register:** `x0` is always `0`.
- **Program counter jumps:** `jal` (call), `ret` (return), conditional branches like `beq, bne, blt`.

Common instructions we can use immediately:

- `addi rd, rs1, imm` — add immediate
- `add rd, rs1, rs2` — add registers
- `li rd, imm` — load small constant (assembler pseudo-op)
- `la rd, symbol` — load address of a label (pseudo-op)
- `ld rd, offset(rs1) / sd rs2, offset(rs1)` — load/store 64-bit
- `mv rd, rs` — move (pseudo-op)
- Branches: `beq, bne, blt, bge`
- Calls/returns: `jal ra, func, ret`
- Linux syscalls: put args in `a0..`, **syscall** number in `a7`, then `ecall`

1.1 Minimal code with `exit` (pure syscall, no `libc`)

Below we present a minimal assembly program that performs a single operation (an **exit** call).

The code is assembled using the `as` command with the debug option **`-g`**. The resulting object file is then linked with the `ld` command.

Program execution is started and inspected within the **debugger** `gdb`.

Note the debugger commands:

`b _start` - to move (**branch**) the execution starting address to label **`_start`**

`r` - to **start** the execution (**`run`**)

`s` - to **run** one **`step`** – next instruction

`i r` - to give the **information** about the state of general **registers**

```
-------------------------------------------------------------------------------------------
# file: exit0.s
    .section .text
    .globl _start
_start:
    li   a0, 0        # exit status = 0
    li   a7, 93       # SYS_exit on RISC-V Linux
    ecall             # make the syscall
-------------------------------------------------------------------------------------------
% as exit0.s -g -o  exit0.o
% ld exit0.o -o exit0

% gdb exit0
..
Reading symbols from exitcall...
(gdb) b _start
Breakpoint 1 at 0x100b0: file exit0.s, line 5.
(gdb) r
Starting program: /home/bako/PLabs/PLabs/book_lab0/exit0

Breakpoint 1, _start () at exit0.s:5
5         li   a0, 0        # exit status = 0
(gdb) s
6         li   a7, 93       # SYS_exit on RISC-V Linux
(gdb) s
7         ecall             # make the syscall
(gdb) i r
ra             0x2aaaad733e   0x2aaaad733e
sp             0x3ffffff880   0x3ffffff880
gp             0x2aaab69558   0x2aaab69558
tp             0x3ff7d7eb80   0x3ff7d7eb80
t0             0xffffffffffffffff   -1
t1             0x2aaaac2a8c   183252036236
t2             0xffffffffffffffff   -1
fp             0x3fffff49f0   0x3fffff49f0
s1             0x3ff7fd2670   274743502448
a0        0x0      0
a1             0x3ff7fd2670   274743502448
a2             0x3ffffff8c8   274877905096
a3             0x3ff7fd2678   274743502456
a4             0x0      0
a5             0x0      0
a6             0x0      0
a7        0x5d    93
s2             0x3fffff45f0   274877859312
..
t3             0x3ff7e1e530   274741716272
t4             0x62616c5f6b6f6f62   7089066445538029410
t5             0x30746978652f30     13638795222396720
t6             0x2aaab727fc   183252756476
pc            0x100b8 0x100b8 <_start+8>
-------------------------------------------------------------------------------------------
```

To do

Analyze the above program and the **debugger** output: **`a0, a7, pc`** .

1.2 Add two integers and print the result (uses `printf` from `libc`)

In this example we define **`main`**, pass arguments in **`a0..`**, call **`printf`**, then return.

```
# file: add_printf.s
    .section .rodata
fmt: .asciz "a + b = %ld\n"

    .section .text
    .globl _start
_start:
    li   a0, 7          # a = 7
    li   a1, 5          # b = 5
    add  a2, a0, a1     # a2 = a + b
    # Set up printf(fmt, a2)
    la   a0, fmt        # a0 = &fmt (1st arg)
    mv   a1, a2         # a1 = sum  (2nd arg)
    call printf         # jal ra, printf
    li   a0, 0          # return 0 from main
   ret
```

Key ideas: note the use of gcc compiler – necessary to import `libc`

- With **`libc`**, define **`_start`**.
- First two args to **`printf`** go in **`a0, a1`**.
- **`call printf`** is a convenient **alias** for **`jal ra, printf`**.

```
gcc -nostartfiles add_printf.s -o add_printf
% ./add_printf
a + b = 12
```

1.3 Sum an array in a loop and print the sum

This shows simple pointer arithmetic, loads, branches, and a **loop** counter.

```
# file: sum_array.s
    .section .rodata
arr:    .quad 10, 20, 30, 40, 50      # 5 elements (64-bit each)
nval:   .quad 5
fmt2:   .asciz "sum = %ld\n"

    .section .text
    .globl _start
_start:
    # a0 = base pointer to arr
    la   a0, arr
    # a1 = number of elements (n)
    la   t0, nval
    ld   a1, 0(t0)
    # a2 = running sum, a3 = loop index i
    li   a2, 0
    li   a3, 0
loop:
    bge  a3, a1, done        # if (i >= n) break
    slli t1, a3, 3           # t1 = i * 8 (bytes per 64-bit)
    add  t2, a0, t1          # t2 = &arr[i]
    ld   t3, 0(t2)           # t3 = arr[i]
    add  a2, a2, t3          # sum += arr[i]
    addi a3, a3, 1           # i++
    j    loop
done:
    # printf("sum = %ld\n", a2)
    la   a0, fmt2
    mv   a1, a2
    call printf
    li   a0, 0
    ret
```

```
% gcc -nostartfiles sum_array.s -o sum_array
% ./sum_array
sum = 150
```

1.4 Big example - 3×3 Matrix Multiply (C=A×B), 64-bit integers

$$\underset{A}{\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix}} \times \underset{B}{\begin{bmatrix}9&8&7\\6&5&4\\3&2&1\end{bmatrix}} = \underset{C}{\begin{bmatrix}30&24&18\\84&69&54\\138&114&90\end{bmatrix}}$$

In this example we introduce:

- **Nested loops** with `bge, addi, slli` (indexing).
- Proper use of **saved registers** (`s0–s2`) and a **stack** frame: `sp` (**6*8=48 bytes**).
- Pointer arithmetic and `ld/sd`.
- Calling `printf` with `RV64 ABI`.

```
    .section .rodata
fmt_row:    .asciz "%4ld %4ld %4ld\n"

A:
    .quad 1, 2, 3
    .quad 4, 5, 6
    .quad 7, 8, 9
B:
    .quad 9, 8, 7
    .quad 6, 5, 4
    .quad 3, 2, 1
    .section .bss
    .align 3                # 8-byte align
C:
    .skip 8*9               # space for 3x3 int64 result
    .section .text
    .globl _start
_start:
    # Prologue: save ra, s0-s4. Keep stack 8-byte aligned (48 bytes total).
    addi sp, sp, -48
    sd   ra, 40(sp)
    sd   s0, 32(sp)
    sd   s1, 24(sp)
    sd   s2, 16(sp)
    sd   s3, 8(sp)
    sd   s4, 0(sp)
    # Base pointers in callee-saved regs
    la   s0, A
    la   s1, B
    la   s2, C
    li   s3, 3              # N = 3 (matrix dimension), callee-saved
    # ---- Compute C = A * B (scalar) ----
    li   t0, 0              # i
outer_i:
    bge  t0, s3, done_compute
    li   t1, 0              # j
outer_j:
    bge  t1, s3, next_i
    li   t2, 0              # sum
    li   t3, 0              # k

inner_k:
    bge  t3, s3, store_C
    # A[i][k]
    mul  t5, t0, s3         # t5 = i*3
    add  t5, t5, t3         # t5 = i*3 + k
    slli t5, t5, 3          # *8 bytes
    add  t6, s0, t5
    ld   t4, 0(t6)          # t4 = A[i][k]
    # B[k][j]
    mul  t5, t3, s3         # t5 = k*3
    add  t5, t5, t1         # t5 = k*3 + j
    slli t5, t5, 3
```

```
    add  t6, s1, t5
    ld   t5, 0(t6)           # t5 = B[k][j]
    mul  t4, t4, t5          # t4 = A[i][k]*B[k][j]
    add  t2, t2, t4          # sum += t4
    addi t3, t3, 1
    j    inner_k

store_C:
    # C[i][j] = sum
    mul  t5, t0, s3
    add  t5, t5, t1
    slli t5, t5, 3
    add  t6, s2, t5
    sd   t2, 0(t6)
    addi t1, t1, 1
    j    outer_j

next_i:
    addi t0, t0, 1
    j    outer_i

done_compute:
    # ---- Print C row by row ----
    li   s4, 0               # row counter in callee-saved reg
print_rows:
    bge  s4, s3, finish
    mul  t5, s4, s3
    slli t5, t5, 3           # byte offset = (row*3)*8
    add  t6, s2, t5
    ld   a1, 0(t6)
    ld   a2, 8(t6)
    ld   a3, 16(t6)
    la   a0, fmt_row
    call printf              # uses caller-saved regs only; s3/s4 preserved
    addi s4, s4, 1
    j    print_rows

finish:
    li   a0, 0
    call exit                # exit(0) — end cleanly regardless of ra
    # (No fallthrough; exit() does not return)
    # Epilogue, in case exit() were replaced with 'ret'
    ld   ra, 40(sp)
    ld   s0, 32(sp)
    ld   s1, 24(sp)
    ld   s2, 16(sp)
    ld   s3, 8(sp)
    ld   s4, 0(sp)
    addi sp, sp, 48
    ret
-------------------------------------------------------------------------------------
% gcc -nostartfiles matmult3x3.s -o matmult3x3
% ./matmult3x3
  30   24   18
  84   69   54
 138  114   90
-------------------------------------------------------------------------------------
```

To do - study in details

Goal: Compute **C=A×B** for two 3×3 `int64` matrices and print the result (three rows).

Register usage map (at a glance)

Callee-saved (survive `printf`)

- `s0` – base address of **A**
- `s1` – base address of **B**
- `s2` – base address of **C**
- `s3` – constant **N = 3** (matrix dimension)
- `s4` – **row counter** for the print loop

Temporaries (caller-saved; used inside compute loops only)

- `t0` – loop `i` (rows of **A**/**C**)
- `t1` – loop `j` (cols of **B**/**C**)
- `t2` – **`sum`** accumulator for `C[i][j]`
- `t3` – loop `k` (inner product index)
- `t4` – scratch for loaded value/product
- `t5` – scratch for indices/offsets
- `t6` – scratch pointer (effective address)

Call/return & args

- `ra` – return address (saved/restored)
- `a0–a3` – **`printf` args** when printing a row

Data layout & addressing

- Row-major matrices (int64): element size = **8 bytes**.
- Index → byte offset:
  - `idx = row * N + col`
  - `offset = idx << 3 (shift-left by 3 = ×8)`

Examples used in the code:

- `A[i][k]`
  `offset = ((i*N) + k) << 3`
- `B[k][j]`
  `offset = ((k*N) + j) << 3`
- `C[i][j]`
  same formula

Loop structure (pseudocode):

```
for (i = 0; i < 3; ++i) {
  for (j = 0; j < 3; ++j) {
    sum = 0;
    for (k = 0; k < 3; ++k) {
      sum += A[i][k] * B[k][j];
    }
    C[i][j] = sum;
  }
}
// print C, one row at a time
for (row = 0; row < 3; ++row) {
  printf("%4ld %4ld %4ld\n", C[row][0], C[row][1], C[row][2]);
}
```

Assembly mapping:

- Outer loops: `i` → `t0`, `j` → `t1`, inner `k` → `t3`
- **Bound** checks use `bge <counter>, s3, <exit_label>`
- Offsets built with `mul` by `s3` then `slli` by `3 (?)`
-

Key instruction patterns (with intent)

- **Prologue/Epilogue**
- Keep stack **16-byte aligned** across calls; save/restore `ra` and all `s*` used.

```
addi sp, sp, -48    # multiple of 16
sd   ra, 40(sp)     # ... save s0-s4 ...
...
ld   ra, 40(sp)
addi sp, sp, 48
```

```
ret
--------------------------------------------------------------------------------
```

Loop compare against constant N (in callee-saved **s3**)

```
bge t0, s3, done_compute
```

Row-major element address

```
mul  t5, t0, s3          # row * N
add  t5, t5, t3          # + k (or + j)
slli t5, t5, 3           # * 8 bytes
add  t6, s0, t5          # base + offset
```

Inner product

```
ld   t4, 0(t6)           # load A[i][k] or B[k][j]
mul  t4, t4, t5          # product
add  t2, t2, t4          # accumulate sum
```

Print row (safe across **printf**)

```
ld   a1, 0(t6)           # C[row][0]
ld   a2, 8(t6)           # C[row][1]
ld   a3, 16(t6)          # C[row][2]
la   a0, fmt_row
call printf              # s3/s4 preserved
```