

# RISC-V assembly

## 1

Arguments go in registers a0 through a7

13 is stored in `a2`, see the code:

```
1  printf("%d %d\n", f(8)+1, 13);
2  24:  4635                li   a2,13
3  26:  45b1                li   a1,12
4  28:  00000517           auipc  a0,0x0
5  2c:  7a850513           addi   a0,a0,1960 # 7d0 <malloc+0xea>
6  30:  00000097           auipc  ra,0x0
7  34:  5f8080e7           jalr   1528(ra) # 628 <printf>
```

## 2

at line

```
1  26:  45b1                li   a1,12
```

it seems that the inline optimization already computed `f(8) + 1 = 12`

## 3

see main:

```
1  000000000000001c <main>:
2
3  void main(void) {
4  1c:  1141                addi   sp,sp,-16
5  1e:  e406                sd    ra,8(sp)
6  20:  e022                sd    s0,0(sp)
7  22:  0800                addi   s0,sp,16
8  printf("%d %d\n", f(8)+1, 13);
9  24:  4635                li   a2,13
10 26:  45b1                li   a1,12
11 28:  00000517           auipc  a0,0x0
12 2c:  7a850513           addi   a0,a0,1960 # 7d0 <malloc+0xea>
13 30:  00000097           auipc  ra,0x0
14 34:  5f8080e7           jalr   1528(ra) # 628 <printf>
15 exit(0);
16 38:  4501                li   a0,0
17 3a:  00000097           auipc  ra,0x0
18 3e:  276080e7           jalr   630(ra) # 2b0 <exit>
```

at 30:

$ra = pc = 0x0000000000000001c + 30 = 0x0000000000000003a$

location of `printf` =  $1528 + ra = 0x00000000000000632$

## 4

ra should be the next instruction to be executed after `jalr`, so it is  
 $0x0000000000000001c + 38 = 0x0000000000000042$

## 5

if we run the following program:

```
1  int main() {
2      unsigned int i = 0x00646c72;
3      printf("H%x Wo%s", 57616, &i);
4  }
```

res: He110 world

If RISC-V is big endian, we need to reverse the order of bits within each byte.

so for 57616

## 6

whatever value that happens to be on the stack above 3 will be printed.  
assembly code:

```
1      .file "testprint.c"
2      .option nopic
3      .attribute arch, "rv64i2p0_m2p0_a2p0_f2p0_d2p0_c2p0"
4      .attribute unaligned_access, 0
5      .attribute stack_align, 16
6      .text
7      .section .rodata
8      .align 3
9      .LC0:
10     .string "x=%d y=%d"
11     .text
12     .align 1
13     .globl main
14     .type main, @function
15 main:
16     addi sp,sp,-16
17     sd ra,8(sp)
18     sd s0,0(sp)
19     addi s0,sp,16
20     li a1,3
21     lui a5,%hi(.LC0)
22     addi a0,a5,%lo(.LC0)
23     call printf
24     li a5,0
25     mv a0,a5
26     ld ra,8(sp)
27     ld s0,0(sp)
28     addi sp,sp,16
29     jr ra
30     .size main, .-main
```

```
31 .ident "GCC: () 10.2.0"
32
```

since we do not set value for a2, so a2 is a random value and whatever value stored in that address will be printed, if that is valid address.

## code explain

.file "testprint.c"

- This directive indicates the name of the source file being compiled (in this case, testprint.c).

.option nopic

- This tells the assembler not to generate position-independent code (PIC). PIC is typically used for shared libraries, and since it's not required here, this option disables it.

.attribute arch, "rv64i2p0\_m2p0\_a2p0\_f2p0\_d2p0\_c2p0"

- This specifies the architecture and features of the RISC-V target. It indicates a 64-bit RISC-V machine (rv64), with certain extensions enabled (e.g., integer, multiplication, floating point, etc.).

.attribute unaligned\_access, 0

- This specifies that unaligned memory access is not allowed. In RISC-V, memory accesses should generally be aligned to avoid performance penalties.

.attribute stack\_align, 16

- This ensures that the stack pointer is aligned to 16-byte boundaries, which is required for efficient data access and compatibility with certain instructions.

.text

- This starts the section of the program that contains executable instructions (code).

.section .rodata

- This starts the section for read-only data (such as string literals).

.align 3

- This ensures that the .LC0 string is aligned to a 3-byte boundary, which is a common alignment for string literals.

.LC0: .string "x=%d y=%d"

- This defines a string constant (used in printf) to format the output. It will print two integers, labeled x and y.

.text

- This indicates the beginning of another text section. The next part of the code will be executable instructions.

.align 1

- Aligns the following code to a 1-byte boundary.

.globl main

- This makes the main function globally visible, meaning it can be called from other modules or linked into the final program.

.type main, @function

- This declares the symbol main as a function.

main:

- This is the label marking the entry point for the main function.

addi sp, sp, -16

- This instruction adjusts the stack pointer (sp) to allocate 16 bytes of space on the stack.

sd ra, 8(sp)

- This saves the return address (ra) to the stack at position sp + 8.

sd s0, 0(sp)

- This saves the value of register s0 to the stack at position sp.

addi s0, sp, 16

- This sets the s0 register to point to the stack space allocated for local variables. It's used for managing the stack frame.

li a1, 3

- This loads the immediate value 3 into register a1, which is used as an argument to printf.

lui a5, %hi(.LC0)

- This loads the upper 20 bits of the address of the string .LC0 (the format string) into register a5.

addi a0, a5, %lo(.LC0)

- This adds the lower 12 bits of the address of .LC0 to a5 to get the full address and stores it in register a0. This will be passed to printf.

call printf

- This calls the printf function to print the formatted string x=%d y=%d with the value of a1 (which is 3), though the second argument is not loaded here, so only 3 would be printed.

## Backtrace

---

core function is in printf.c:

```
1 void backtrace()
2 {
3     printf("backtrace:\n");
4     uint64 fp = r_fp();
5     uint64 top = PGROUNDUP(fp);
6     uint64 bottom = PGROUNDDOWN(fp);
7
8     while(fp >= bottom && fp <= top) {
9         // location of return address in user virtual memory = fp - 8
10        // we need to print the M(fp - 8) - 8, which is the address of calling
function
11        uint64 ra = *(uint64 *)(fp - 8);
12
13        printf("%p\n", ra - 8);
14
15        fp = *(uint64 *)(fp - 16);
16    }
17 }
18
```

result:

```

1  $ bttest
2  backtrace:
3  0x0000000080002148
4  0x0000000080001fac
5  0x0000000080001c96
6  $ QEMU: Terminated
7  $ addr2line -e kernel/kernel
8  0x0000000080002148
9  /home/toadjiang/Code/OS_study/xv6-labs-2021/kernel/sysproc.c:73
10 0x0000000080001fac
11 /home/toadjiang/Code/OS_study/xv6-labs-2021/kernel/syscall.c:139
   (discriminator 1)
12 0x0000000080001c96
13 /home/toadjiang/Code/OS_study/xv6-labs-2021/kernel/trap.c:67
14 ^Ax?:0

```

note that when we call `sys_sleep()`, we are in the kernel stack in supervisor mode with kernel satp. So VM is directly mapped to PM.

We can get return address `ra` at `Memory[fp-8]` and the previous frame pointer `fp` at `Memory[fp-16]`, and we repeatedly print `ra - 8`, which is the instruction jump to the next routine, are update fp with previous fp.