

## 1 RISC-V Instruction Translation

1.1 In this question, translate the following RISC-V instructions into their binary and hexadecimal values.

a) `addi s1 x0 -24` = 0b\_\_\_\_\_

= 0x\_\_\_\_\_

b) `sh s1 4(t1)` = 0b\_\_\_\_\_

= 0x\_\_\_\_\_

1.2 In this question, translate the following hexadecimal values into RISC-V instructions.

a) 0xFE05 0CE3 = \_\_\_\_\_

b) 0x2345 54B7 = \_\_\_\_\_

1.3 Given the following RISC-V code and instruction addresses, translate the `jal` and `bne` instructions (you'll need your RISC-V reference sheet!) and determine the value of `R[ra]` during the execution of `loop`.

```

    loop:
0x002CFF00:    add t1, t2, t0        0x00538333

0x002CFF04:    jal ra, foo        _____

0x002CFF08:    bne t1, zero, loop    _____
                ...
    foo:
0x002CFF2C:    jr ra            R[ra] = _____

```

## 2 RISC-V Addressing

We have several *addressing modes* to access memory (immediate not listed):

- a) Base displacement addressing adds an immediate to a register value to create a data memory address (used for `lw`, `lb`, `sw`, `sb`).
- b) PC-relative addressing uses the PC and adds the immediate value of the instruction to create an instruction address (used by branch and jump instructions).
- c) Register Addressing uses the value in a register as an instruction address. For instance, `jalr`, `jr`, and `ret`, where `jr` and `ret` are just pseudoinstructions that get converted to `jalr`.

2.1 What is the range of 32-bit instructions that can be reached from the current PC using a single branch instruction? Note that RISC-V branch instructions must support branching to 16-bit “compressed” instructions (enabled via an optional RISC-V extension).

2.2 What is the maximum range of 32-bit instructions that can be reached from the current PC using a jump instruction?

### 3 Two-Pass Assembly

Consider the following assembly code. Assume that `printf` exists in the C standard library and that `msg` exists at an unknown address in the `.data` section.

| Address | Assembly                           |
|---------|------------------------------------|
| .data   | msg: .string "Hello World"         |
| .text   |                                    |
| 0x0C    | add t0, x0, x0                     |
| 0x10    | addi t1, x0, 4                     |
| 0x14    | loop: beq t0, t1, end              |
| 0x18    | addi a0, a0, 1                     |
| 0x1C    | la a0, msg # load address of `msg` |
| 0x20    | jal ra, printf                     |
| 0x24    | n: addi t0, t0, 1                  |
| 0x28    | j loop                             |
| 0x2C    | end: ret                           |

3.1 This code is output from the \_\_\_\_\_ (Compiler, Assembler, Linker, or Loader) and \_\_\_\_\_ (may / may not) contain pseudoinstructions.

3.2 Assume we are using a two-pass assembler. Fill out the symbol table after the first pass (top-to-bottom) of the assembler. Not all lines may be used. The order of entries in the table do not matter.

| Symbol Table |         |
|--------------|---------|
| Label        | Address |
|              |         |
|              |         |
|              |         |
|              |         |
|              |         |

3.3 After the first pass of the assembler, which of the instructions do not have their addresses fully resolved?

- 3.4 After the second pass of the assembler but before the linker, which of the instructions do not have their addresses fully resolved?