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 1111 1110 1000 0000 0000 0100 1001 0011

1.1

= 0x FE800493

1.1

b) `sh s1 4(t1)` = 0b 0000 0000 1001 0011 0001 0010 0010 0011

1.1

= 0x 00931223

1.1

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

a) `0xFE05 0CE3` = beq a0 x0 -8

1.2

b) `0x2345 54B7` = lui s1 0x23455

1.2

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             0x028000EF
0x002CFF08:     bne t1, zero, loop      0xFE031CE3
                ...
      foo:
0x002CFF2C:     jr ra          R[ra] =  0x002CFF08
```

1.3

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).

从PC-4096到PC+4094 字节

32位指令范围：PC-1024到PC+1022 指令

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

共524287个32位指令，范围从PC-262144到PC+262142

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 (Compiler, Assembler, Linker, or Loader)and **may** (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**
loop	0x14
n	0x24
end	0x2C
msg	未知

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

beq, la, jal

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

la, jal