1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 The main job of the assembler is to generate optimized machine code.

False. That’s the job of the compiler. The assembler is mostly replacing pseudoinstructions and resolving offsets.

1.2 The object files produced by the assembler are only moved, not edited, by the linker.

False. The linker needs to relocate all absolute address references.

1.3 The destination of all jump instructions is completely determined after linking.

False. Jumps relative to registers are only known at run-time. Otherwise, you would not be able to call a function from different call sites.

2 CALL

The following is a diagram of the CALL stack detailing how C programs are built and executed by machines:
2.1 What is the Stored Program concept and what does it enable us to do?

It is the idea that instructions are just the same as data, and we can treat them as such. This enables us to write programs that can manipulate other programs!

2.2 How many passes through the code does the Assembler have to make? Why?

Two, one to find all the label addresses and another to convert all instructions while resolving any forward references using the collected label addresses.

2.3 Describe the six main parts of the object files outputed by the Assembler (Header, Text, Data, Relocation Table, Symbol Table, Debugging Information).

- Header: Size and position of other parts
- Text: The machine code
- Data: Binary representation of any data in the source file
- Relocation Table: Identifies lines of code that need to be “handled” by Linker (jumps to external labels (e.g. lib files), reference to static data)
- Symbol Table: List of file labels and data that can be referenced across files
- Debugging Information: Additional information for debuggers

2.4 Which step in CALL resolves relative addressing? Absolute addressing?

Assembler, Linker
3  Assembling RISC-V

Let’s say that we have a C program that has a single function `sum` that computes
the sum of an array. We’ve compiled it to RISC-V, but we haven’t assembled the
RISC-V code yet.

```assembly
.import print.s  # print.s is a different file
.data
.array: .word 1 2 3 4 5
.text
.sum:   la t0, array
       li t1, 4
       mv t2, x0
.loop:  blt t1, x0, end
        slli t3, t1, 2
        addi t3, t0, t3
        lw t3, 0(t3)
        add t2, t2, t3
        addi t1, t1, -1
        j loop
.end:   mv a0, t2
.jal ra, print_int  # Defined in print.s
```

3.1 Which lines contain pseudoinstructions that need to be converted to regular RISC-V
instructions?

5, 6, 7, 14, 15.

`la` becomes the `auipc` and `addi` instructions.

`li` becomes an `addi` instruction here (e.g. `li t0, 4 → addi t0, x0, 4`).

`mv` becomes an `addi` instruction (i.e. `mv rd, rs → addi rd, rs, 0`).

`j` becomes a `jal` instruction (e.g. `j loop → jal x0, loop`).

3.2 For the branch/jump instructions, which labels will be resolved in the first pass of
the assembler? The second?

Note: This answer assumes that the assembler goes from top to bottom. The
answer changes if it goes in reverse.

`loop` (in `j loop`) will be resolved in the first pass since it’s a backward reference.
Since the assembler will have kept note of where `end` is in the first pass, it will
resolve `end` in `blt t1, x0, end` in the second pass.

Let’s assume that the code for this program starts at address `0x00061C00`. The
code below is labelled with its address in memory (think: why is there a jump of 8
between the first and second lines?).

There’s a jump of 8 because `la` is a pseudoinstruction that gets translated to two
regular RISC-V instructions!
4 RISC-V Procedures

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

1. Base displacement addressing adds an immediate to a register value to create a memory address (used for lw, lb, sw, sb).
2. PC-relative addressing uses the PC and adds the immediate value of the instruction (multiplied by 2) to create an address (used by branch and jump instructions).

3. Register Addressing uses the value in a register as a memory address. For instance, jalr, jr, and ret, where jr and ret are just pseudoinstructions that get converted to jalr.

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

The immediate field of the branch instruction is 12 bits. This field only references addresses that are divisible by 2, so the immediate is multiplied by 2 before being added to the PC. Therefore, the branch immediate can move PC in the range of $[-2^{12}, 2^{12} - 1]$ bytes. If we’re in a version of RISC-V that has 2-byte instructions, then this corresponds to a range of $[-2^{11}, 2^{11} - 1]$ instructions. The instructions we use, however, are 4 bytes so they reside at addresses that are divisible by 4 not 2. Therefore, we can only reference half as many 4-byte instructions as before, and the range of 4-byte instructions is $[-2^{10}, 2^{10} - 1]$.

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

The immediate field of the jump instruction is 20 bits. Similar to above, this immediate is multiplied by 2 before added to the PC to get the final address. Since the immediate is signed, we have a range of $[-2^{20}, 2^{20} - 1]$ bytes, or $[-2^{19}, 2^{19} - 1]$ 2-byte instructions. As we actually want the number of 4-byte instructions, we actually can reference those within $[-2^{18}, 2^{18} - 1]$ instructions of the current PC.

4.3 Given the following RISC-V code (and instruction addresses), fill in the blank fields for the following instructions (you’ll need your RISC-V green card!).

```
1 0x002cff00: loop: add t1, t2, t0 |________|________|________|________|________|__0x33__|
2 0x002cff04: jal ra, foo |__________________________|_________________|__0x6F__|
3 0x002cff08: bne t1, zero, loop |________|________|________|________|________|__0x63__|
4 ...
5 0x002cff2c: foo: jr ra ra = __________________________
```

```
1 0x002cff00: loop: add t1, t2, t0 | 0 | 5 | 7 | 0 | 6 | 0x33 |
2 0x002cff04: jal ra, foo | 0 | 0x14 | 0 | 0 | 1 | 0x6F |
3 0x002cff08: bne t1, zero, loop | 1 | 0x3F | 0 | 6 | 1 | 0xC | 1 | 0x63 |
4 ...
5 0x002cff2c: foo: jr ra ra = _______ 0x002cff08_______
```