1 CALL

The following is a diagram of the CALL stack detailing how C programs are built and executed by machines:

C program: foo.c
   Compiler
   Assembly program: foo.a
      Assembler
      Object Code: foo.o
         Linker
            Executable a.out (Machine Language)
               Loader
                  Memory

1.1 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!

1.2 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.

1.3 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
CALL, RISC-V Procedures

- 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

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

Assembler, Linker

2 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
```

2.1 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`).

2.2 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!

<table>
<thead>
<tr>
<th>Address</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00061C00</td>
<td>sum: la t0, array</td>
</tr>
<tr>
<td>0x00061C08</td>
<td>li t1, 4</td>
</tr>
<tr>
<td>0x00061C0C</td>
<td>mv t2, x0</td>
</tr>
<tr>
<td>0x00061C10</td>
<td>blt t1, x0, end</td>
</tr>
<tr>
<td>0x00061C14</td>
<td>slli t3, t1, 2</td>
</tr>
<tr>
<td>0x00061C18</td>
<td>addi t3, t0, t3</td>
</tr>
<tr>
<td>0x00061C1C</td>
<td>lw t3, 0(t3)</td>
</tr>
<tr>
<td>0x00061C20</td>
<td>add t2, t2, t3</td>
</tr>
<tr>
<td>0x00061C24</td>
<td>addi t1, t1, -1</td>
</tr>
<tr>
<td>0x00061C28</td>
<td>j loop</td>
</tr>
<tr>
<td>0x00061C2C</td>
<td>end: mv a0, t2</td>
</tr>
<tr>
<td>0x00061C30</td>
<td>jal ra, print_int</td>
</tr>
</tbody>
</table>

What is in the symbol table after the assembler makes its passes?

<table>
<thead>
<tr>
<th>Label</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>0x00061C00</td>
</tr>
</tbody>
</table>

Normally, one would assume that both the loop and end labels would be included in the symbol table—and that’s perfectly valid answer given that an isolated assembler would have no way to tell the difference between the three labels.

However, we stated at the beginning of this problem that this file is compiled from C code. If we have an integrated compiler, assembler, and linker (e.g. gcc), then it will know from the compilation phase which labels are for functions and which ones aren’t. As such, it will only put the function labels in the symbol table since those are the only ones that other files can reference.

What’s contained in the relocation table?

array and print_int.

Since array is defined in the static portion of memory, there’s no way the assembler could know where it will be located (relative to the program counter) until the program actually executes. We recall that the static portion of memory is above the code portion of memory. Since we haven’t linked other files with this one yet (that’s done in the linker phase!), we don’t know how much code we’ll have, so we don’t know where the static portion of memory will begin! Also, other files may declare
items in static memory, and the assembler won’t know how these are specifically ordered when the program is finally loaded.

Similarly, print_int is defined in a different file, so the assembler doesn’t know where it will be in the final executable. That will be decided in the linking stage.

3 RISC-V Addressing

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.

3.1 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 \([-2^{10}, 2^{10} - 1]\).

3.2 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.

3.3 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 = __________________________
4 Writing RISC-V Functions

Write a function \texttt{sumSquare} in RISC-V that, when given an integer \( n \), returns the summation below. If \( n \) is not positive, then the function returns 0.

\[
\sum_{i=0}^{n-1} i^2 = n^2 + (n - 1)^2 + (n - 2)^2 + \ldots + 1^2
\]

For this problem, you are given a RISC-V function called \texttt{square} that takes in a single integer and returns its square.

First, let’s implement the meat of the function: the squaring and summing. We will be abiding by the caller/callee convention, so in what register can we expect the parameter \( n \)? What registers should hold \texttt{square}’s parameter and return value? In what register should we place the return value of \texttt{sumSquare}?  

```
add s0, a0, x0  # Set s0 equal to the parameter n
add s1, x0, x0  # Set s1 (accumulator) equal to 0
loop: beq s0, x0, end  # Branch if s0 reaches 0
  add a0, s0, x0  # Set a0 to the value in s0, setting up
  # args for call to function square
jal ra, square  # Call the function square
  add s1, s1, a0  # Add the returned value into s1
addi s0, s0, -1  # Decrement s0 by 1
jal x0, loop  # Jump back to the loop label
end: add a0, s1, x0  # Set a0 to s1 (desired return value)
```

Since \texttt{sumSquare} is the callee, we need to ensure that it is not overriding any registers that the caller may use. Given your implementation above, write a prologue and epilogue to account for the registers you used.

```
prologue: addi sp, sp -12  # Make space for 3 words on the stack
  sw ra, 0(sp)  # Store the return address
  sw s0, 4(sp)  # Store register s0
  sw s1, 8(sp)  # Store register s1

epilogue: lw ra, 0(sp)  # Restore ra
  lw s0, 4(sp)  # Restore s0
  lw s1, 8(sp)  # Restore s1
  addi sp, sp, 12  # Free space on the stack for the 3 words
  jr ra  # Return to the caller
```