1 Problem 1: Getting Started with the LC-2199

In this homework, you will be using the LC-2199 ISA to complete a Tower of Hanoi move-counting function. Before you begin, you should familiarize yourself with the available instructions, the register conventions and the calling convention of LC-2199. Details can be found in the section, Appendix A: LC-2199 Instruction Set Architecture, at the end of this document.

The `assembly` folder contains several tools for you to use:

- `assembler.py`: a basic assembler that can take your assembly code and convert it into binary instructions for the LC-2199.
- `lc2199.py`: the ISA definition file for the assembler, which tells assembler.py the instructions supported by the LC-2199 and their formats.
- `lc2199-sim.py`: A simulator of the LC-2199 machine. The simulator reads binary instructions and emulates the LC-2199 machine, letting you single-step through instructions and check their results.

To learn how to run these tools, see the `README.md` file in the `assembly` directory.

Before you begin work on the second problem of the homework, try writing a simple program for the LC-2199 architecture. This should help you familiarize yourself with the available instructions.

We have provided a template, `mod.s`, for you to use for this purpose. Try writing a program that performs the `mod` operation on the two provided arguments. A correct implementation will result in a value of 2.

You can use the following C code snippet as a guide to implement this function:

```
int mod(int a, int b) {
  int x = a;
  while (x >= b) {
    x = x - b;
  }
  return x;
}
```

There is no turn-in for this portion of the assignment, but it is **recommended** that you attempt it in order to familiarize yourself with the ISA.

2 Problem 2: Tower of Hanoi

For this problem, you will be implementing the missing portions of the program that calculates the minimum number of moves to solve the Tower of Hanoi problem for `n` disks.

Tower of Hanoi involves three vertical rods and a set of varying sized disks, which can slide onto any rod. The disks are initially stacked on one of the rods in ascending order of size, with the largest disk on the bottom and the smallest on top, thus making a conical shape. The objective of this puzzle is to migrate the tower of disks completely to another rod, under the rule that only individual disks may be moved at once, and no disks may be placed on smaller disks.

You will be finishing a **recursive** implementation of the Tower of Hanoi minimal moves calculator program that follows the LC-2199 calling convention. Recursive functions always obtain a return address through the function call and return to the callee using the return address.

You must use the stack pointer ($sp) and frame pointer ($fp) registers as described in the textbook and lecture slides.

Here is the C code for the Tower of Hanoi minimal moves calculator you have been provided:

```
int minimumHanoi(int n) {
  if (n == 1)
    return 1;
  else
    return (2 * minimumHanoi(n - 1)) + 1;
}
```

Note that this C code is just to help your understanding and does not need to be exactly followed. However, your assembly code implementation should meet all of the given conditions in the description.

Open `hanoi.s` file in the assembly directory. This file contains an implementation of the Tower of Hanoi minimal moves calculator program that is missing significant portions of the calling convention. Near the bottom of the `hanoi.s` we have provided multiple numbers that you can use to test your homework. They are located at labels `testNumDisks1`, `testNumDisks2`, `testNumDisks3`. Be sure to use these provided integers by loading them from the labels into registers. None of the numbers provided and tested will be lower than 1.

Complete the program by implementing the various missing portions of the LC-2199 calling convention. Each location where you need to implement a portion of the calling convention is marked with a TODO label as well as a short hint describing the portion of the calling convention you should be implementing.

Please note that we will be testing your implementation for multiple different instances, so please do not attempt to hardcode your solutions.

3 Problem 3: Short Answer

Please answer the following question in the file named `answers.txt`:

1. The LC-2199 instruction set contains an instruction called `jalr` that is used to jump to a location while saving a return address. However, this functionality could be emulated using a combination of other instructions available in the ISA. Describe a sequence of other instructions in the LC-2199 ISA that you may use to accomplish the functionality of `jalr`.

   For the purpose of this question, you may assume the target address is represented with the label `<target>` which can be accessed using the 20 bits reserved for an offset or immediate value in the LC-2199 ISA.

4 Deliverables

- `hanoi.s`: your assembly code from Section 2
- `answers.txt`: your answer to the problem from Section 3

Submit these files to **Gradescope** before the assignment deadline.

The TAs should be able to type `python assembler.py -i lc2199 --sym hanoi.s` and then `python lc2199-sim.py hanoi.bin` to run your code. If you cannot do this with your submission, then you have done something wrong.

5 Appendix A: LC-2199 Instruction Set Architecture

The LC-2199 is a simple, yet capable computer architecture. The LC-2199 combines attributes of both ARM and the LC-2200 ISA defined in the Ramachandran & Leahy textbook for CS 2200.

The LC-2199 is a **word-addressable**, **32-bit** computer. **All addresses refer to words**, i.e. the first word (four bytes) in memory occupies address 0x0, the second word, 0x1, etc.

All memory addresses are truncated to 16 bits on access, discarding the 16 most significant bits if the address was stored in a 32-bit register. This provides roughly 64 KB of addressable memory.

5.1 Registers

The LC-2199 has 16 general-purpose registers. While there are no hardware-enforced restraints on the uses of these registers, your code is expected to follow the conventions outlined below.

Table 1: Registers and their Uses

Register Number	Name	Use	Callee Save?
0	$zero	Always Zero	NA
1	$at	Assembler/Target Address	NA
2	$v0	Return Value	No
3	$a0	Argument 1	No
4	$a1	Argument 2	No
5	$a2	Argument 3	No
6	$t0	Temporary Variable	No
7	$t1	Temporary Variable	No
8	$t2	Temporary Variable	No
9	$s0	Saved Register	Yes
10	$s1	Saved Register	Yes
11	$s2	Saved Register	Yes
12	$k0	Reserved for OS and Traps	NA
13	$sp	Stack Pointer	No
14	$fp	Frame Pointer	Yes
15	$ra	Return Address	No

1. **Register 0** is always read as zero. Any values written to it are discarded. Regardless of what is written to this register, it should always output zero.
2. **Register 1** is used to hold the target address of a jump. It may also be used by pseudo-instructions generated by the assembler.
3. **Register 2** is where you should store any returned value from a subroutine call.
4. **Registers 3 - 5** are used to store function/subroutine arguments. **Note:** registers 2 through 8 should be placed on the stack if the caller wants to retain those values. These registers are fair game for the callee (subroutine) to trash.
5. **Registers 6 - 8** are designated for temporary variables. The caller must save these registers if they want these values to be retained.
6. **Registers 9 - 11** are saved registers. The caller may assume that these registers are never tampered with by the subroutine. If the subroutine needs these registers, then it should place them on the stack and restore them before they jump back to the caller.
7. **Register 12** is reserved for handling interrupts. While it should be implemented, it otherwise will not have any special use on this assignment.

8. **Register 13** is the everchanging top of the stack stack; it keeps track of the top of the activation record for a subroutine.
9. **Register 14** is the anchor point of the activation frame. It is used to point to the first address on the activation record for the currently executing process.
10. **Register 15** is used to store the address a subroutine should return to when it is finished executing.

5.2 Instruction Overview

The LC-2199 supports a variety of instruction forms, only a few of which we will use for this project. The instructions we will implement in this project are summarized below.

Table 2: LC-2199 Instruction Set

	31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6	5 4	3 2 1 0
ADD	0000	DR	SR1	unused		SR2
NAND	0001	DR	SR1	unused		SR2
ADDI	0010	DR	SR1	immval20		
LW	0011	DR	BaseR	offset20		
SW	0100	SR	BaseR	offset20		
BR	0101	unused		offset20		
JALR	0110	RA	AT	unused		
HALT	0111	unused				
SKPEQ	1000	SR1	SR2	unused		0000
SKPGT	1000	SR1	SR2	unused		0001
LEA	1001	DR	unused	PCoffset20		
SLL	1010	DR	SR1	unused	00	SR2
SRL	1010	DR	SR1	unused	01	SR2
SRA	1010	DR	SR1	unused	10	SR2
ROR	1010	DR	SR1	unused	11	SR2

5.2.1 Conditional Branching

Branching in the LC-2199 ISA is a bit different than usual. We have the series of isntructions known as the Skip Instructions, or SKP. These instructions use the comparison operators, comparing the values of two source registers. If the comparisons are true (for example, with the SKPGT instruction, if SR1 > SR2), then we skip over the next line of code – we increment PC by 1 (remember that at the time of execution, PC has already been incremented by 1, so this is an additional increment).

Note: These SKP instructions all have the same opcode and use IR[2:0] to determine the comparison type. Recall the following. Bit 0 is used to check equality between SR1 and SR2. Bit 1 is used to check if SR1 is less than SR2. Bit 2 is used to negate the condition. We have given you some examples in section 3.1.4, so try and work out the rest on your own.

5.3 Detailed Instruction Reference

5.3.1 ADD

Assembler Syntax

```
ADD    DR, SR1, SR2
```

Encoding

31–28	27–24	23–20	19–4	3–0
0000	DR	SR1	unused	SR2

Operation

```
DR = SR1 + SR2;
```

Description

The ADD instruction obtains the first source operand from the SR1 register. The second source operand is obtained from the SR2 register. The second operand is added to the first source operand, and the result is stored in DR.

5.3.2 NAND

Assembler Syntax

```
NAND   DR, SR1, SR2
```

Encoding

31–28	27–24	23–20	19–4	3–0
0001	DR	SR1	unused	SR2

Operation

```
DR = ~(SR1 & SR2);
```

Description

The NAND instruction performs a logical NAND (AND NOT) on the source operands obtained from SR1 and SR2. The result is stored in DR.

> **HINT:** A logical NOT can be achieved by performing a NAND with both source operands the same. For instance,
>
> `NAND DR, SR1, SR1`
>
> ...achieves the following logical operation: $DR \leftarrow \overline{SR1}$.

5.3.3 ADDI

Assembler Syntax

```
ADDI   DR, SR1, immval20
```

Encoding

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0010	DR	SR1	immval20

Operation

```
DR = SR1 + SEXT(immval20);
```

Description

The ADDI instruction obtains the first source operand from the SR1 register. The second source operand is obtained by sign-extending the immval20 field to 32 bits. The resulting operand is added to the first source operand, and the result is stored in DR.

5.3.4 LW

Assembler Syntax

```
LW    DR, offset20(BaseR)
```

Encoding

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0011	DR	BaseR	offset20

Operation

```
DR = MEM[BaseR + SEXT(offset20)];
```

Description

An address is computed by sign-extending bits [19:0] to 32 bits and then adding this result to the contents of the register specified by bits [23:20]. The 32-bit word at this address is loaded into DR.

5.3.5 SW

Assembler Syntax

```
SW    SR, offset20(BaseR)
```

Encoding

31–28	27–24	23–20	19–0
0100	SR	BaseR	offset20

Operation

```
MEM[BaseR + SEXT(offset20)] = SR;
```

Description

An address is computed by sign-extending bits [19:0] to 32 bits and then adding this result to the contents of the register specified by bits [23:20]. The 32-bit word obtained from register SR is then stored at this address.

5.3.6 BR

Assembler Syntax

```
BR  offset20
```

Encoding

31–28	27–20	19–0
0101	unused	offset20

Operation

```
PC = incrementedPC + offset20
```

Description

A branch is unconditionally taken. The PC will be set to the sum of the incremented PC (since we have already undergone fetch) and the sign-extended offset[19:0].

5.3.7 JALR

Assembler Syntax

```
JALR   RA, AT
```

Encoding

31–28	27–24	23–20	19–0
0110	RA	AT	unused

Operation

```
RA = PC;
PC = AT;
```

Description

First, the incremented PC (address of the instruction + 1) is stored into register RA. Next, the PC is loaded with the value of register AT, and the computer resumes execution at the new PC.

5.3.8 HALT

Assembler Syntax

```
HALT
```

Encoding

31–28	27–0
0111	unused

Description

The machine is brought to a halt and executes no further instructions.

5.3.9 SKPEQ

Assembler Syntax

```
SKPEQ  SR1, SR2
```

Encoding

31–28	27–24	23–20	19–4	3–0
1000	SR1	SR2	unused	0000

Operation

```
if (SR1 == SR2) {
    PC = incrementedPC + 1
}
```

Description

The incremented PC is further incremented by 1 if SR1 is equal to SR2.

5.3.10 SKPGT

Assembler Syntax

```
SKPGT  SR1, SR2
```

Encoding

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4	3 2 1 0
1000	SR1	SR2	unused	0001

Operation

```
if (SR1 > SR2) {
    PC = incrementedPC + 1
}
```

Description

The incremented PC is further incremented by 1 if SR1 is greater than SR2.

5.3.11 LEA

Assembler Syntax

```
LEA    DR, label
```

Encoding

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1001	DR	unused	PCoffset20

Operation

```
DR = PC + SEXT(PCoffset20);
```

Description

An address is computed by sign-extending bits [19:0] to 32 bits and adding this result to the incremented PC (address of instruction + 1). It then stores the computed address into register DR.

5.3.12 SLL

Assembler Syntax

```
SLL    DR, SR1, SR2
```

Encoding

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6	5 4	3 2 1 0
1010	DR	SR1	unused	00	SR2

Operation

```
DR = SR1 << SR2;
```

Description

The value stored in SR1 is logically left shifted by the value stored in SR2, and the result is stored in DR.

5.3.13 SRL

Assembler Syntax

```
SRL    DR, SR1, SR2
```

Encoding

31–28	27–24	23–20	19–6	5–4	3–0
1010	DR	SR1	unused	01	SR2

Operation

```
DR = SR1 >> SR2;
```

Description

The value stored in SR1 is logically right shifted by the value stored in SR2, and the result is stored in DR.

5.3.14 SRA

Assembler Syntax

```
SRA    DR, SR1, SR2
```

Encoding

31–28	27–24	23–20	19–6	5–4	3–0
1010	DR	SR1	unused	10	SR2

Operation

```
DR = SR1 >> SR2;
```

Description

The value stored in SR1 is arithmetically right shifted by the value stored in SR2, and the result is stored in DR. **NOTE THE DIFFERENCE BETWEEN SHIFTS: Logical right shift will fill the resulting space with 0s, while arithmetic right shift sign-extends the MSB**.

> **HINT:** Like there is a component to add or subtract, there is also a component that performs shifts. Try to find and play around with this component to see how it can be helpful in your project.

5.3.15 ROR

Assembler Syntax

```
ROR    DR, SR1, SR2
```

Encoding

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6	5 4	3 2 1 0
1010	DR	SR1	unused	11	SR2

Operation

```
DR = (SR1 >> SR2) | (SR1 << (32 - SR2));
```

Description

Bits in SR1 are "rotated" by SR2 times as if the left and right ends of the register were joined. The values that are shifted into the left will be the values that are shifted off from the right during the right-shift operation.