

# Computer Systems and Networks

Prof. Forsyth

## Homework 2 - Assembly Review

Due: Sep 10<sup>th</sup> 2024

## 1 Problem 1: Getting Started with the LC-4000

In this homework, you will be using the LC-4000 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-4000. Details can be found in the section, Appendix A: LC-4000 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-4000.
- `lc4000.py`: the ISA definition file for the assembler, which tells `assembler.py` the instructions supported by the LC-4000 and their formats.
- `lc4000-sim.py`: A simulator of the LC-4000 machine. The simulator reads binary instructions and emulates the LC-4000 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-4000 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-4000 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-4000 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-4000 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. Provide a sequence of other instructions in the LC-4000 ISA that you may use to accomplish the functionality of `jalr`. And describe why your solution works.

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-4000 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.

## 5 Local Debugging

To debug locally, you can use commands below.

```
python3 assembler.py -i lc4000 hanoi.s --bin hanoi.bin --sym hanoi.sym
python3 lc4000-sim.py hanoi.bin
```

To learn more about the python files used here, read Appendix B.

## 6 Appendix A: LC-4000 Instruction Set Architecture

The LC-4000 is a simple, yet capable computer architecture. The LC-4000 combines attributes of both ARM and the LC-2200 ISA defined in the Ramachandran & Leahy textbook for CS 2200.

The LC-4000 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.

### 6.1 Registers

The LC-4000 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. **Note:** for the purposes of this project, you must implement the zero register. 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; 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.

## 6.2 Instruction Overview

The LC-4000 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-4000 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																															
NAND	0001																															
ADDI	0010																															
LW	0011																															
SW	0100																															
BEQ	0101																															
JALR	0110																															
HALT	0111																															
BLT	1000																															
LEA	1001																															
MIN	1010																															
MAX	1010																															

### 6.2.1 Conditional Branching

Branching in the LC-4000 ISA is slightly different than usual. We have a set of branching instructions including both BEQ and BLT, which offer the ability to branch upon a certain condition being met. These instructions use comparison operators, comparing the values of two source registers. If the comparisons are true (for example, with the BLT instruction, if  $SR1 < SR2$ ), then we will branch to the target destination of  $incrementedPC + offset20$ .

For MIN, if  $SR1 < SR2$ , we branch to the series of microstates that stores the value in SR1 to DR. Otherwise, we branch to a series of microstates that stores the value in SR2 to DR. For MAX, we do the opposite.

## 6.3 Detailed Instruction Reference

### 6.3.1 ADD

#### Assembler Syntax

ADD 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
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.

### 6.3.2 NAND

#### Assembler Syntax

NAND 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
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. The following assembly:

NAND DR, SR1, SR1

achieves the following logical operation:  $DR \leftarrow \overline{SR1}$ .

### 6.3.3 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.

### 6.3.4 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.



**6.3.5 SW****Assembler Syntax**

SW SR, 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
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.

**6.3.6 BEQ****Assembler Syntax**

BEQ SR1, SR2, offset20

**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
0101				SR1				SR2				offset20																			

**Operation**

```
if (SR1 == SR2) {
    PC = incrementedPC + offset20
}
```

**Description**

A branch is taken if SR1 is equal to SR2. If this is the case, 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].

### 6.3.7 JALR

#### Assembler Syntax

JALR AT, RA

#### 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
0110	AT				RA				unused																						

#### Operation

RA = PC;

PC = AT;

#### Description

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

### 6.3.8 HALT

#### Assembler Syntax

HALT

#### 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
0111				unused																											

#### Description

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

### 6.3.9 BLT

#### Assembler Syntax

BLT SR1, SR2, offset20

#### 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				offset20																						

#### Operation

```
if (SR1 < SR2) {
    PC = incrementedPC + offset20
}
```

#### Description

A branch is taken if SR1 is less than SR2. If this is the case, 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].

**6.3.10 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 in the register DR.

**6.3.11 MIN****Assembler Syntax**

MIN     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										0		SR2											

**Operation**

```
if (SR1 < SR2) {
    DR = SR1
} else {
    DR = SR2
}
```

**Description**

The minimum is computed between the values in both source registers. It then stores the minimum value in the register DR.

**6.3.12 MAX****Assembler Syntax**

MAX 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																1	SR2		

**Operation**

```
if (SR1 > SR2) {  
    DR = SR1  
} else {  
    DR = SR2  
}
```

**Description**

The maximum is computed between the values in both source registers. It then stores the maximum value in the register DR.

## 7 Appendix B: Assembler and Simulator

### 7.1 LC-4000 Assembler and Simulator

To aid in testing your assembly code, we have provided an assembler and simulator for the LC-4000 architecture. The assembler supports converting text `.s` files into either binary (for the simulator) or hexadecimal (for pasting into ) formats.

The assembler and simulator run on any version of Python 3+. An instruction set architecture definition file is required along with the assembler. The LC-4000 assembler definition is included. If you have issues with Python, run the commands in the Docker container as it has all the dependencies for using these files.

- **assembler.py**: the main assembler program
- **lc4000.py**: the LC-4000 assembler definition
- **lc4000-sim.py**: the LC-4000 simulator program

**Note:** At any point, if using the `python3` command does not work for you, try replacing it with the `python` command.

We highly recommend reading through these files, and understanding the tools available to help you debug and complete this homework.

### 7.2 Using the Assembler

Example usage to assemble **hanoi.s**:

```
python3 assembler.py -i lc4000 hanoi.s --sym hanoi.sym --bin hanoi.bin
```

This should create **hanoi.sym** and **hanoi.bin** files. These files can then be loaded into the simulator.

### 7.3 Using the Simulator

You can use the simulator to explore how your processor is expected to behave, step through code instruction-by-instruction, and examine the values of registers and memory at each stage of the program.

Example usage to simulate **hanoi.bin** (generated by the assembler):

```
python3 lc4000-sim.py hanoi.bin
```

Then type “help” at the prompt for a list of available commands:

```
(sim) help
```

Welcome to the simulator text interface. The available commands are:

<code>r[un] or c[ontinue]</code>	resume execution until next breakpoint
<code>s[tep]</code>	execute one instruction
<code>b[reak] &lt;addr or label&gt;</code>	view and set breakpoints
	ex) 'break'
	ex) 'break 0x3'
	ex) 'break main'
<code>print &lt;lowaddr&gt;-&lt;highaddr&gt;</code>	print the values in memory between <lowaddr> and <highaddr>
	ex) 'print 0x20-0x30'
<code>q[uit]</code>	exit the simulator

This can help examine register values at different points of a program. You can also do this by uploading the hex into and running the program.

## 7.4 Assembler Pseudo-Ops

In addition to the syntax described in the LC-4000 ISA reference, the assembler supports the following pseudo-instructions:

- `.fill`: fills a word of memory with the literal value provided
- `.word`: an alias for `.fill`

For example: `mynumber: .fill 0x500` places 0x500 at the memory location labeled `mynumber`.