

Computer Systems and Networks

Prof. Ramachandran, Prof. Daglis, Prof. Sarma

Homework 2 - Assembly Review & Register Conventions

Due: **January 30th 2025** @ 11:59 PM EST

1 Problem 1: Getting Started with the LC-3200

In this homework, you will be using the **LC-3200** ISA to complete a recursive fibonacci function. This is the same series denoted by

$$F_n = 0, 1, 1, 2, 3, 5, 8, 13, \dots$$

where $F_n = F_{n-1} + F_{n-2}$. Before you begin, you should familiarize yourself with the available instructions, the register conventions and the calling convention of LC-3200. Details can be found in the section, Appendix A: LC-3200 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 machine code for the LC-3200.
- **LC3200.isa**: the ISA definition file for the assembler, which tells assembler.py the instructions supported by the LC-3200 and their formats.
- **simulator.py**: A simulator of the LC-3200 machine. The simulator reads binary instructions and emulates the LC-3200 machine, letting you single-step through instructions and check their results.

Before you begin work on the second problem of the homework, try writing a simple program for the LC-3200 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 submission for this portion of the assignment, but it is **recommended** that you attempt it to familiarize yourself with the ISA. It will be significantly easier than the Fibonacci function that you will implement.

2 Problem 2: Fibonacci

For this problem, you will be implementing the missing portions of the program that calculates the n^{th} number of the Fibonacci sequence.

2.1 Background

In mathematics, the Fibonacci sequence is a sequence in which each element is the sum of the two elements that precede it. Numbers that are part of the Fibonacci sequence are known as Fibonacci numbers. They appear in biological settings, such as branching in trees, the arrangement of leaves on a stem, the fruit sprouts of a pineapple, the flowering of an artichoke, and the arrangement of a pine cone's bracts, though they do not occur in all species.

2.2 Requirements

You will be finishing a **recursive** implementation of the Fibonacci number calculator program that follows the LC-3200 calling convention.

- Your implementation must be recursive. You are NOT permitted to implement the function iteratively.
- You must use the stack pointer (\$sp) and frame pointer (\$fp) registers as described in the textbook and lecture slides.
- You should adhere to the calling convention discussed in lecture and lab in regards to saving values to the stack and retrieving values from the stack.
- Store the return value of the function in \$v0

We understand that the sharp among you will notice that Fibonacci written this way isn't the most time efficient way of generating a solution (since we are repeating the same subproblem a bunch of times). But the purpose of this assignment is to become comfortable with LC-3200 and certain conventions associated with it. If we see an iterative implementation of Fibonacci, we will take off points.

Reminder: DO NOT implement Fibonacci with an iterative function, i.e. looping.

2.3 Pseudocode

Here is the C code for the Fibonacci calculator program you have been asked to implement.

```
int fibonacci(int n) {
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

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 the conditions given in the description.

2.4 Getting Started

Open `fibonacci.s` file in the assembly directory. This file contains an implementation of the Fibonacci number calculator program that is missing significant portions of its implementation. Near the bottom of the `fibonacci.s` file we have provided multiple values that you can use to test your homework. They are located at labels `testFibInput1`, `testFibInput2`, `testFibInput3`. Be sure to use these provided integers by loading them from the labels into registers. None of the numbers provided and tested will be less than 1.

Complete the program by implementing the various missing portions of the LC-3200 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.

2.5 Tips & Hints

1. Apart from initializing the stack, please do not edit `main`'s functionality.
2. You do not need to save the return address before jumping to `fibonacci` in `main`.
3. Follow the `TODOs` to figure out where to insert your implementation.
4. Implement your assembly code analogous to the pseudocode provided to keep it simple.
5. We do not explicitly tell you what to save and where to save as per the calling convention. Feel free to reference lab slides and lecture slides.
6. Use the simulator in the section Using the Simulator as an interactive debugger to debug your code locally rather than spamming the autograder.

3 Problem 3: Short Answer

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

The LC-3200 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-3200 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-3200 ISA.

4 Deliverables

- `fibonacci.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 the following commands.

```
python3 assembler.py fib.s -i LC3200.isa fib.s
python3 simulator.py fib.bin
```

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

6 Appendix A: LC-3200 Instruction Set Architecture

The LC-3200 is a simple, yet capable computer architecture. The LC-3200 ISA is based off the LC-2200 ISA defined in the Ramachandran & Leahy textbook for CS 2200 with a few additional instructions. We highly recommend reading the textbook to understand how an ISA works.

The LC-3200 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 66 KB of addressable memory.

6.1 Registers

The LC-3200 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 always outputs zero.
2. **Register 1** is used to hold the target address of a jump. Pseudo-instructions generated by the assembler may also be used.
3. **Register 2** is where you should store any returned value from a subroutine call. A quick way to check if **pow.s** runs correctly to verify if this register's value is correct.
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 the subroutine never tampered with these registers. 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.

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-3200 supports a variety of instruction forms, only a few of which we will use for this homework. The instructions we may use in this homework are summarized below.

Table 2: LC-3200 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																						SR2
NAND	0001									DR																						SR2
ADDI	0010									DR																						immval20
LW	0011									DR																						offset20
SW	0100									SR																						offset20
BEQ	0101									SR1																						offset20
JALR	0110									AT																						RA
HALT	0111																															unused
BGT	1000									SR1																						SR2
LEA	1001									DR																						offset20
TERN	1010									DR																						SR2
COLS	1011									DR																						SR2
SWAP	1100									SR1																						SR2

6.2.1 Conditional Branching

Branching in the LC-3200 ISA is slightly different than usual. With BEQ and BGT, we offer the ability to branch upon a certain condition being met. These instructions uses comparison operators, comparing the values of two source registers. If the comparisons are true (for example, with the BEQ instruction, if $SR1 == SR2$), then we will branch to the target destination of $incrementedPC + offset20$.

For COLS, if $SR1 == 0$, 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.

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.

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 BGT**Assembler Syntax**

BGT 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 greater 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 TERN**Assembler Syntax**

TERN 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																SR2					

Operation

```
if (DR != 0) {
    DR = SR1
} else {
    DR = SR2
}
```

Description

The ternary operator is evaluated by setting DR to SR1 if DR is not 0, and to SR2 otherwise.

6.3.12 COLS**Assembler Syntax**

COLS 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
1011				DR				SR1				unused																SR2			

Operation

```
if (SR1 == 0) {  
    DR = SR2  
} else {  
    DR = SR1  
}
```

Description

The null coalesce operator checks whether the data in SR1 is zero (i.e. null). If SR1 is zero, then DR is set to SR2. If SR1 is not zero, then DR is set to SR1

6.3.13 SWAP**Assembler Syntax**

SWAP 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
1100				SR1				SR2				unused																			

Operation

```
TempA = SR1  
TempB = SR2  
SR2 = TempA  
SR1 = TempB
```

Description

The data contained in the registers SR1 and SR2 are swapped.

7 Appendix B: Assembler and Simulator

7.1 LC-3200 Assembler and Simulator

To aid in testing your processor, we have provided an assembler and simulator for the LC-3200 architecture. The assembler supports converting text `.s` files into either binary (for the simulator) or hexadecimal (for pasting into CircuitSim) 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-3200 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
- **LC3200.isa**: the LC-3200 assembler definition
- **simulator.py**: the LC-3200 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 project.

Note: The simulator is NOT an autograder to test your files. Rather, it is a tool built to assist with the expected functionality of the ISA. In order to locally debug your LC-3200 implementation, follow the instructions below to load the **pow.hex** file into CircuitSim.

7.2 Using the Assembler

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

```
python3 assembler.py pow.s -i LC3200.isa
```

This should create **pow.hex** and **pow.bin** files. To use assembled code in CircuitSim, we need it in hexadecimal. You can then open the resulting **pow.hex** file in your favorite text editor. In CircuitSim, double-click into the RAM subcircuit from your datapath. Then right-click on your RAM, select “Edit Contents”, and copy-and-paste the contents of **pow.hex** into this window. Make sure to click into the subcircuit from your datapath, **not the tabs at the top of the window**, otherwise the changes will not persist once you leave the subcircuit.

Do not use the Open or Save buttons in CircuitSim, as it will not recognize the format.

Tip: Clear the contents of the RAM before pasting to ensure that they are properly pasted.

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 **pow.bin** (generated by the assembler):

```
python3 simulator.py pow.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:
```

r[un] or c[ontinue]	resume execution until next breakpoint
s[tep]	execute one instruction
b[reak] <addr or label>	view and set breakpoints
	ex) 'break'
	ex) 'break 0x3'
	ex) 'break main'
print <lowaddr>-<highaddr>	print the values in memory between <lowaddr> and <highaddr>
	ex) 'print 0x20-0x30'
q[uit]	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 CircuitSim and running the program.

7.4 Assembler Pseudo-Ops

In addition to the syntax described in the LC-3200 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**.