Computer Systems and Networks

Prof. Forsyth

Homework 2 - Assembly & Calling Convention

Due: **September 4^{th} 2025** @ 11:59 PM EDT

1 Problem 1: Getting Started with the LC-4200

In this homework, you will be using the **LC-4200** ISA to complete a recursive fibonacci function. This is the same series denoted by

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

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-4200. Details can be found in the section, Appendix A: LC-4200 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-4200.
- `LC-4200.isa`: the ISA definition file for the assembler, which tells assembler.py the instructions supported by the LC-4200 and their formats.
- `simulator.py`: A simulator of the LC-4200 machine. The simulator reads binary instructions and emulates the LC-4200 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-4200 architecture. This should help you familiarize yourself with the available instructions.

You have been 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 $\mathtt{n}^{\text{th}}$ 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-4200 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.
- Due to this function having a single argument, do not save $a (argument) registers directly onto the stack.
- Store the return value of the function in $v0

The sharp among you will notice that Fibonacci written this way isn't the most time efficient way of generating a solution (since the same subproblem is being repeated a bunch of times). But the purpose of this assignment is to become comfortable with LC-4200 and certain conventions associated with it. If you submit an iterative implementation of Fibonacci, you may lose a significant amount of 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 Calling Convention

The LC-4200 ABI includes, but is not limited to, the following specification. For additional clarification on the specific registers and semantics of the LC-4200 ABI, refer to the appendix or the lecture and lab slides.

1. Registers designated as callee-saved and caller-saved should be saved in the appropriate section of the code.
2. All registers should be saved in the correct order on the stack.
3. The stack pointer should always point to valid data on the stack.
4. When allocating space for registers on the stack, the amount of reserved stack space used should be minimized while still adhering to the convention specified by the ABI.
5. The frame pointer must be saved upon function calls.

2.5 Getting Started

Open `fib.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 `fib.s` file you have been provided values in the labels `testFibInput1`, `testFibInput2`, `testFibInput3` that you can use to test your homework. 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-4200 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.

Your implementation will be tested with several different inputs; do not attempt to hardcode your solutions.

2.6 Tips & Hints

1. Apart from initializing the stack, 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 analagous to the pseudocode provided to keep it simple.
5. Feel free to reference lab slides and lecture slides for more details on the LC-2200/LC-4200 ABI
6. Use the simulator in the section Appendix B: Assembler and Simulator from Project 1 as an interactive debugger to debug your code locally rather than spamming the autograder.
7. Appendix A: LC-4200 Instruction Set Architecture from Project 1 has been attached for your reference and includes a detailed description of the registers and the ISA.
8. If you do not handle the base case properly, you could end up with a **StackOverflow Error** which might manifest as the autograder crashing since instructions may get overwritten by data with illegal opcodes.

3 Problem 3: Short Answer

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

1. The LC-4200 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-4200 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-4200 ISA.

2. Why do we distinguish between $s and $t registers? Is there an advantage to using either in different situations. Give your answer in roughly 2 to 4 sentences.

4 Deliverables

- `fib.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 LC-4200.isa
python3 simulator.py fib.bin
```

To learn more about the python files used here, read Appendix B: Assembler and Simulator.

6 Appendix A: LC-4200 Instruction Set Architecture

The LC-4200 is a simple, yet capable computer architecture. The LC-4200 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. It may also provide further hints on how to implement some of the instructions

The LC-4200 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 262 KB of addressable memory.

6.1 Registers

The LC-4200 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 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. 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. 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-4200 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-4200 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	
BEQ	0101	SR1	SR2	offset20	
JALR	0110	AT	RA	unused	
HALT	0111	unused			
BGT	1000	SR1	SR2	offset20	
LEA	1001	DR	unused	offset20	
XORI	1010	DR	SR1	immval20	
CLMP	1011	DR	SR1	unused	SR2

6.2.1 Conditional Branching

Branching in the LC-4200 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 CLMP, if DR < SR1, we branch to the series of microstates that stores the value in SR1 to DR. Otherwise, we check whether DR > SR2, and if so branch to the series of microstates that stores the value in SR2 to DR.

HINT: You can reuse microstates for different instructions for a more efficient microcode!

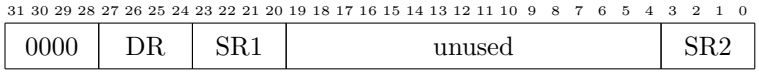
6.3 Detailed Instruction Reference

6.3.1 ADD

Assembler Syntax

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

Encoding



Operation

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

Description

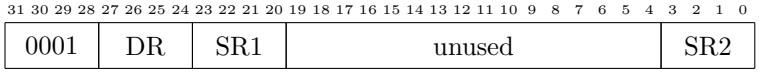
The ADD instruction obtains the first source operand from the SR1 register. The second source operand is obtained from the SR2 register. The second operand is added to the first source operand, and the result is stored in DR.

6.3.2 NAND

Assembler Syntax

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

Encoding



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.

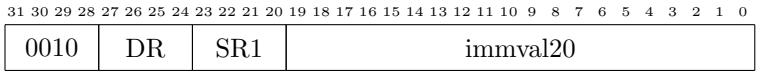
> **HINT:** A logical NOT can be achieved by performing a NAND with both source operands the same. The following assembly:
>
> `NAND DR, SR1, SR1`
>
> achieves the following logical operation: $DR \leftarrow \overline{SR1}$.

6.3.3 ADDI

Assembler Syntax

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

Encoding



Operation

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

Description

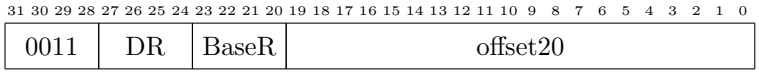
The ADDI instruction obtains the first source operand from the SR1 register. The second source operand is obtained by sign-extending the immval20 field to 32 bits. The resulting operand is added to the first source operand, and the result is stored in DR.

6.3.4 LW

Assembler Syntax

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

Encoding



Operation

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

Description

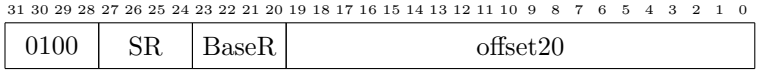
An address is computed by sign-extending bits [19:0] to 32 bits and then adding this result to the contents of the register specified by bits [23:20]. The 32-bit word at this address is loaded into DR.

6.3.5 SW

Assembler Syntax

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

Encoding



Operation

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

Description

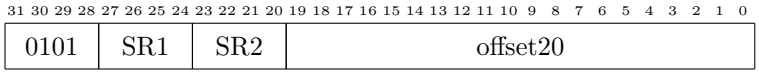
An address is computed by sign-extending bits [19:0] to 32 bits and then adding this result to the contents of the register specified by bits [23:20]. The 32-bit word obtained from register SR is then stored at this address.

6.3.6 BEQ

Assembler Syntax

```
BEQ  SR1, SR2, offset20
```

Encoding



Operation

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

Description

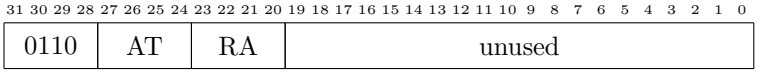
A branch is taken if SR1 is equal to SR2. If this is the case, the PC will be set to the sum of the incremented PC (since we have already undergone fetch) and the sign-extended offset[19:0].

6.3.7 JALR

Assembler Syntax

```
JALR   AT, RA
```

Encoding



Operation

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

Description

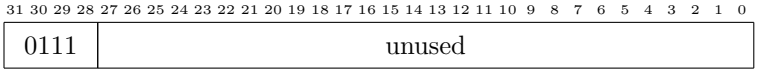
First, the incremented PC (address of the instruction + 1) is stored in register RA. Next, the PC is loaded with the value of register AT, and the computer resumes execution at the new PC.

6.3.8 HALT

Assembler Syntax

```
HALT
```

Encoding



Description

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

> **Note:** The autograder will not run if this instruction is not implemented correctly.

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 XORI

Assembler Syntax

```
XORI    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
1010	DR	SR1	immval20

Operation

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

Description

XORI takes the bitwise xor of SR1 and SEXT(immval20) and stores it in DR.

6.3.12 CLMP

Assembler Syntax

```
CLMP    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 (DR < SR1) {
    DR = SR1;
} else if (DR > SR2) {
    DR = SR2;
} else {
    // do nothing
}
```

Description

If DR is outside of the range [SR1, SR2], it is moved the least amount such that it will be in said range.

7 Appendix B: Assembler and Simulator

7.1 LC-4200 Assembler and Simulator

To aid in testing your processor, we have provided an assembler and simulator for the LC-4200 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-4200 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
- **LC-4200.isa**: the LC-4200 assembler definition
- **simulator.py**: the LC-4200 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-4200 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 LC-4200.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-4200 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`.