1 Problem 1: Getting Started with the RAMA-2200

In this homework, you will be using the RAMA-2200 ISA to complete a Fibonacci function. Before you begin, you should familiarize yourself with the available instructions, the register conventions and the calling convention of RAMA-2200. Details can be found in the section, Appendix A: RAMA-2200 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 RAMA-2200.
- `rama2200.py`: the ISA definition file for the assembler, which tells assembler.py the instructions supported by the RAMA-2200 and their formats.
- `rama2200-sim.py`: A simulator of the RAMA-2200 machine. The simulator reads binary instructions and emulates the RAMA-2200 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 RAMA-2200 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:

```c
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: Fibonacci Number

For this problem, you will be implementing the missing portions of the Fibonacci Number program we have provided for you.

You'll be finishing a **recursive** implementation of the Fibonacci Number program that follows the RAMA-2200 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 Fibonacci Number algorithm you have been provided:

```
int fib(int n) {
  if (n <= 1) {
    if (n < 0) {
      n = 0;
    }
    return n;
  }
  else {
    return fib(n-1) + fib(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 of the given conditions in the description.

Open the `fib.s` file in the `assembly` directory. This file contains an implementation of the Fibonacci Number program that is missing significant portions of the calling convention.

Complete the given Fibonacci number subroutine by implementing the various missing portions of the RAMA-2200 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.

There are some important restrictions / reminders for this assignment.

1. Store parameters to *s* registers during setup stage of RAMA-2200 calling convention (there will be an instruction in the code).
2. Do NOT use stack to store local variables.

Please note that we will be testing your implementation for multiple values of n, so please do not attempt to hardcode your solutions. We will be testing for all possible values of n, where n is an integer.

3 Problem 3: Short Answer

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

1. The RAMA-2200 instruction set contains an instruction called `goto` that is used to go to a certain location within the code, specified by a label. However, this functionality could be emulated using a combination of other instructions available in the ISA. Describe a sequence of other instructions in the RAMA-2200 ISA that you may use to accomplish the functionality of `goto`.

4 Deliverables

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

Submit these files to Canvas before the assignment deadline.

The TAs should be able to type `python assembler.py -i rama2200 --sym fib.s` and then `python rama2200-sim.py fib.bin` to run your code. If you cannot do this with your submission, then you have done something wrong.

5 Appendix A: RAMA-2200 Instruction Set Architecture

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

The RAMA-2200 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 RAMA-2200 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 your anchor on the stack. It keeps track of the top of the activation record for a subroutine.
9. **Register 14** 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 RAMA-2200 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: RAMA-2200 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	
GOTO	0101	0000	unused	PCoffset20	
JALR	0110	RA	AT	unused	
HALT	0111	unused			
SKP	1000	mode	SR1	unused	SR2
LEA	1001	DR	unused	PCoffset20	

5.2.1 Conditional Branching

Conditional branching in the RAMA-2200 ISA is provided via two instructions: the SKP ("skip") instruction and the GOTO ("unconditional branch") instruction. The SKP instruction compares two registers and skips the immediately following instruction if the comparison evaluates to true. If the action to be conditionally executed is only a single instruction, it can be placed immediately following the SKP instruction. Otherwise a GOTO can be placed following the SKP instruction to branch over to a longer sequence of instructions to be conditionally executed.

5.3 Detailed Instruction Reference

5.3.1 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.

5.3.2 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. 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 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.

5.3.6 GOTO

Assembler Syntax

```
GOTO    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
0101	0000	unused	PCoffset20

Operation

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

Description

The program unconditionally branches to the location specified by adding the sign-extended PCoffset20 field to the incremented PC (address of instruction + 1). **In other words, the PCoffset20 field specifies the number of instructions, forwards or backwards, to branch over.**

5.3.7 JALR

Assembler Syntax

```
JALR   RA, AT
```

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	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 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.

5.3.9 SKP

Assembler Syntax

```
SKPE   SR1, SR2
SKPLT  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	mode	SR1	unused	SR2

mode is defined to be 0x0 for SKPE, and 0x1 for SKPLT.

Operation

```
if (MODE == 0x0) {
    if (SR1 == SR2) PC = PC + 1;
} else if (MODE == 0x1) {
    if (SR1 < SR2) PC = PC + 1;
}
```

Description

The SKP instruction compares the source operands SR1 and SR2 according to the rule specified by the mode field. For mode 0x0, the comparison succeeds if SR1 equals SR2. For mode 0x1, the comparison succeeds if SR1 is less than SR2.

If the comparison succeeds, the incremented PC (address of instruction + 1) is incremented again, for a resulting PC of (address of instruction + 2). **This effectively skips the immediately following instruction.**. If the comparison fails, the program continues execution as normal.

5.3.10 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 into register DR.