1 Problem 1: Getting Started with the GT-2200

In this homework, you will be using the GT-2200 ISA to program a GCD function. Before you begin, you should familiarize yourself with the available instructions, the register conventions and the calling convention of GT-2200. Details can be found in the section, Appendix A: GT-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 GT-2200.
- `gt2200.py`: the ISA definition file for the assembler, which tells assembler.py the instructions supported by the GT-2200 and their formats.
- `gt2200-sim.py`: A simulator of the GT-2200 machine. The simulator reads binary instructions and emulates the GT-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.

To get started, we're going to write a simple implementation to preform the `mod` (modulo) operation. The `mod` operation (`a % b`) takes two integers a and b, and returns the remainder after dividing $\frac{a}{b}$.

Many ISAs provide a `mod` instruction to perform this calculation in hardware, but the GT-2200 is not one of those. Fortunately limited cases of mod are easy to implement in software, as shown with this small snippet of C code:

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

Hint: To implement subtraction, you'll need to re-visit your knowledge of 2's complement. A logical NOT can be implemented in GT-2200 using the NAND instruction, as explained in the reference.

Edit the text file called `mod.s`. Write the necessary GT-2200 assembly instructions within this file to calculate `28 mod 13` using the algorithm above. Your code only needs to support non-negative integers a and b, $b \neq 0$. **Store your result in the $v0 register.** For this problem, you do not have to follow any calling convention.

Run your code in the simulator. If your code works correctly, the `$v0` register should contain the number 2. Before you call it finished, try a few other inputs within the constraints specified in the previous paragraph to make sure your program is robust.

Submit your mod.s file to T-Square along with the file on the next page.

If you have trouble figuring out how to start, try writing a simpler program first, such as one that adds two numbers. The "step" function in the assembler is very helpful for debugging issues, as it lets you run one instruction at a time and check the results.

2 Problem 2: GCD Program

With your mod program working, we'll now tackle the complete GCD (Greatest Common Divisor) algorithm.

For GCD, you'll be writing a **recursive** implementation that follows the LC-2200 calling convention. **Iterative solutions will receive no credit.** 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 GCD algorithm you will be implementing:

```
int gcd(int a, int b) {
  if (b == 0) {
    return a;
  }
  return gcd(b, a % b);
}
```

Open the `gcd.s` file in the `assembly` directory. This file contains the basic skeleton for a recursive implementation of the GCD program.

1. Complete the code below the `main` label to initialize the stack pointer (`$sp`) to point at `0xFFFFFF`. You should use the provided "stack" to load this value into the register.
2. Write a function in GT-2200 to compute `gcd(a, b)`. Your routine is required to follow the GT-2200 calling convention. It should work for any inputs a and b, where $a \geq 0$, $b \geq 0$, and $a \geq b$.

To implement the `mod` operation in the GCD code, you'll need to integrate the code you wrote in Problem 1: Getting Started with the GT-2200. You may write this code as a proper subroutine (using the calling convention) and call it from your `gcd` routine, or you may include it directly or branch to it.

Once complete, test your code with a variety of inputs, as we will do when grading your assignment. As before, your code only has to work for $a \geq b$, where a and b are non-negative.

3 Deliverables

- `mod.s` your MOD assembly code from Problem 1: Getting Started with the GT-2200
- `gcd.s` your GCD assembly code, from Problem 2: GCD Program

Upload both of these files directly to T-Square before the assignment deadline.

Make sure that mod.s and gcd.s are in a UNIX-readable format (no DOS/Windows nonsense).

The TAs should be able to type `python assembler.py -i gt2200 --sym gcd.s` and then `python gt2200-sim.py gcd.bin` to run your code. If you cannot do this with your submission, then you have done something wrong.

4 Appendix A: GT-2200 Instruction Set Architecture

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

The GT-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 24 bits on access, discarding the 8 most significant bits if the address was stored in a 32-bit register. This provides roughly 67 MB of addressable memory.

4.1 Registers

The GT-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	Reserved for the Assembler	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 a general purpose register. You should not use it because the assembler will use it in processing pseudo-instructions.
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. Don't worry about using this register.
10. **Register 15** is used to store the address a subroutine should return to when it is finished executing.

4.2 Instruction Overview

The GT-2200 supports a variety of instruction forms, only a few of which we will use for this homework. The instructions we will use in this homework are summarized below.

Table 2: GT-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
ADDI	0001	DR	SR1	immval20	
NAND	0010	DR	SR1	unused	SR2
SKP	0011	mode	SR1	unused	SR2
GOTO	0100	0000	unused	PCoffset20	
LEA	0101	DR	unused	PCoffset20	
LW	1000	DR	BaseR	offset20	
SW	1001	SR	BaseR	offset20	
JALR	1100	AT	RA	unused	
HALT	1111	unused			

4.2.1 Conditional Branching

Conditional branching in the GT-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.

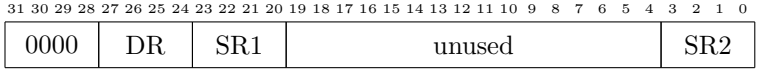
4.3 Detailed Instruction Reference

4.3.1 ADD

Assembler Syntax

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

Encoding



Operation

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

Description

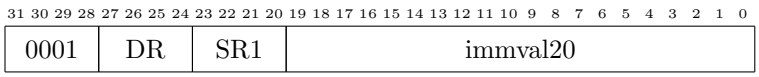
The ADD instruction obtains the first source operand from the SR1 register. The second source operand is obtained from the SR2 register. The second operand is added to the first source operand, and the result is stored in DR.

4.3.2 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.

4.3.3 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	
0010	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.

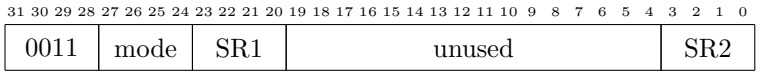
> **HINT:** A logical NOT can be achieved by performing a NAND with both source operands the same. For instance,
>
> `NAND DR, SR1, SR1`
>
> ...achieves the following logical operation: $DR \leftarrow \overline{SR1}$.

4.3.4 SKP

Assembler Syntax

```
SKPE    SR1, SR2
SKPGT   SR1, SR2
```

Encoding



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

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 as signed two's-complement integers 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 greater than SR2.

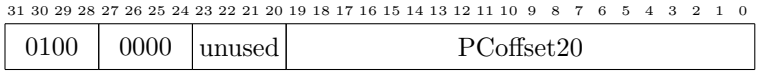
If the comparison succeeds, the incremented PC (address of instruction + 1) is incremented again, for a resulting PC of (address of instruction + 2). **This effectively "skips" the immediately following instruction.** If the comparison fails, the program continues execution as normal.

4.3.5 GOTO

Assembler Syntax

```
GOTO   LABEL
```

Encoding



Operation

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

Description

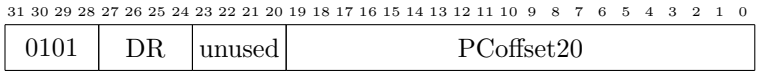
The program unconditionally branches to the location specified by adding the sign-extended PCoffset20 field to the incremented PC (address of instruction + 1). **In other words, the PCoffset20 field specifies the number of instructions, forwards or backwards, to branch over.**

4.3.6 LEA

Assembler Syntax

```
LEA    DR, label
```

Encoding



Operation

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

Description

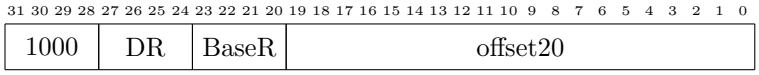
An address is computed by sign-extending bits [19:0] to 32 bits and adding this result to the incremented PC (address of instruction + 1). This instruction effectively performs the same computation as the GOTO instruction, but rather than performing a branch, merely stores the computed address into register DR.

4.3.7 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.

4.3.8 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
1001	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.

4.3.9 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
1100	AT	RA	unused

Operation

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

Description

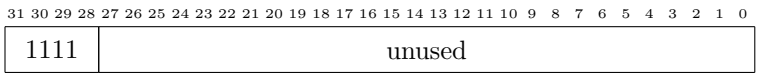
First, the incremented PC (address of the instruction + 1) is stored into register RA. Next, the PC is loaded with the value of register AT, and the computer resumes execution at the new PC.

4.3.10 HALT

Assembler Syntax

```
HALT
```

Encoding



Description

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