Problem 1: Assembly Programming Warmup

In this problem, you will be introduced to the 16 bit LC (Little Computer) 2200 assembly language. You will learn the syntax and semantics that underlie each of the supported operations. Although our instruction set is not as extensive as MIPS (Millions of Instructions Per Second) or x86, it is still able to solve a multitude of problems. In a LC-2200 computer, the word size is two bytes (16 bits) and there are 16 registers. We restrict memory to be addressable by words. Please see the diagram of the Datapath for the LC 2200 Processor.

Register Conventions

Although the registers are for general-purpose use, we shall place restrictions on their use for the sake of convention and all that is good on this earth. Here is a table of their names and uses:

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	$pr	Frame Pointer	Yes
15	$ra	Return Address	No

Register 0 This register is always read as zero. Any values written to it are discarded.

Register 1 is a general purpose register, you should not use it because the assembler will use it in processing pseudo-instructions.

Register 2 is where you should store any returned value from a subroutine call.

Registers 3 to 5 are used to pass arguments into subroutines.

Registers 6 to 8 are used to store temporary values. Note that 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 (the subroutine) to trash.

Registers 9 to 11 are the 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 palce them on the stack and restore them before they jump back to the caller's code.

Register 12 is used to handle interrupts (something we'll get to in a few weeks).

Register 13 is your anchor on the stack. It keeps track of the top of the activation record for some subroutine.

Register 14 is used to point to the first address on the activation record for the currently executing process. You do not need to worry about using this register.

Register 15 is used to store the address a subroutine should return to when it is finished executing. It is only supposed to be used by the JALR (Jump And Link Register) command.

Instruction Formats

There are four types of instructions: R-Type (Register Type), I-Type (Immediate value Type), J-Type (Jump Type), and S-Type (Stack Type).

Here is the instruction format for R-Type instructions (ADD, NAND):

Bits	15 - 13	12 - 9	8 - 5	4 - 1	0
Purpose	opcode	RX	RY	RZ	Unused

Here is the instruction format for I-Type instructions (ADDI, LW, SW, BEQ):

Bits	15 - 13	12 - 9	8 - 5	4 - 0
Purpose	opcode	RX	RY	2's Complement Offset

Here is the instruction format for J-Type instructions (JALR):

Bits	15 - 13	12 - 9	8 - 5	4 - 0
Purpose	opcode	RX	RY	Unused (all 0s)

Here is the instruction format for S-Type instructions (SPOP):

Bits	15 - 13	12 - 2	1 - 0
Purpose	opcode	Unused (all 0s)	Control Code

Symbolic instructions follow the same layout. That is, the order of the registers and offset fields align with the order given in the instruction format, ie. instructions in assembly are written as:
instruction RX, RY, RZ **or**
instruction RX, [optional offset](RY).

Table 2: Assembly Language Instruction Descriptions

Name	Type	Example	Opcode	Action
add	R	add $v0, $a0, $a2	000	Add contents of RY with the contents of RZ and store the result in RX.
nand	R	nand $v0, $a0, $a2	001	NAND contents of RY with the contents of RZ and store the result in RX.
addi	I	addi $v0, $a0, 7	010	Add contents of RY to the contents of the offset field and store the result in RX.
lw	I	lw $v0, 0x07($sp)	011	Load RX from memory. The memory address is formed by adding the offset to the contents of RY.
sw	I	sw $a0, 0x07($sp)	100	Store RX into memory. The memory address is formed by adding the offset to the contents of RY.
beq	I	beq $a0, $a1, done	101	Compare the contents of RX and RY. If they are the same, then branch to address PC + 1 + Offset, where PC is the address of the beq instruction. **Memory is word addressed**. Note that if you use a label in a BEQ instruction, it will jump to the relative offset of the label.
jalr	J	jalr $at, $ra	110	First store PC + 1 in RY, where PC is the address of the jalr instruciton. Then branch to the address in RX. If RX = RY, then the processor will store PC + 1 into RY and end up branching to PC + 1.
spop	S	spop 0	111	Perform the action as determined by the control code, which is the last two bits (control code = 0 tells the processor to halt).

LC 2200 provides a number of pseudo-instructions.

Table 3: Assembly Language Pseudo-Instructions

Name	Example	Action
halt	halt	Emits a spop 0 to halt the processor.
la	la $a0 MyLabel	Loads the address of a label into a register.
noop	noop	No operation, does nothing. It actually does add $zero, $zero, $zero.
.word	.word 32, .word MyLabel	Fills the memory location it is located with a given value or the address of the label.

[**0 points**]Play around with the simulator. Try writing some simple programs to copy values from one register to another or to load/store values from memory. You should get familiar with the syntax for the assembler. When you extract the archive, you will have access to two different assemblers and two simulators: `gt16as` and `lcas.pl` (`lcas.pl` is the perl version of `gt16as` and is still being worked on). These files are located in hw1/gt16/ and hw1/lcasm, respectively. **You must tell linux that these files are executables. Please run `chmod a+x *` in hw1/gt16/ and hw1/lcasm/ in order to run these programs**. The simulator comes with a help option to explore all the different comamands. You might want to bring up the assembler executables to the top-level directory, hw1.

Here is the suggested workflow for writing and running your assembly programs on the simulator:

1. Edit and save your assembly file with your favorite text editor.
2. Assemble your code by running `./gt16as myFile.s`. If this operation is successful, you will have a file called 'myFile.lc'.
3. You can run your .lc file with the simulator by typing `./gt16text myFile.lc` (lctext.pl is the perl version of gt16text). Some useful commands are 'r' for run, 'q' for quit, 'break [line # or label]', and 'help'.

*** We recommend using the gt16 versions, as they will be what is used for grading. There is one difference between the assemblers/simulators. gt16 calls the Frame Pointer $pr and lcasm calls it $fp.**

Problem 2: Factorial Test Program

In this problem, you have to use the LC 2200 assembly language to write a simple program.

1. [**30 points**] Define a procedure calling convention for the LC-2200 assembly language. Your answer should have enough detail so that someone else could write a procedure (or procedure call) to be used as part of another program. You can come up with your own convention, but we recommend basing your convention description off of the one shown in class and described above. Be sure to explicitly address the following standard issues:
   (a) [**10/30 points**] Define how registers are used. Which registers are used for what?
   (b) [**10/30 points**] Define how the stack is accessed. What does the stack pointer point to? In which way does the stack grow in terms of memory addresses?
   (c) [**10/30 points**] Define the mechanics of the call, including what the caller does to initiate a procedure call, what the callee does at the beginning of a procedure, what the callee does at the end of a procedure to return to the caller, and what the caller does to clean up after the procedure returned.
2. [**70 points**] Write a function in LC-2200 to compute `factorial(num)`. Your function is required to follow the calling convention you established above. It should work for non-negative integers, but you don't have to handle detecting/handling integer overflow. **YOUR FUNCTION MUST BE RECURSIVE, PURELY ITERATIVE SOLUTIONS WILL NOT RECIEVE ANY CREDIT! YOU MUST USE THE STACK AND STACK POINTER TO IMPLEMENT RECURSION FOR FULL CREDIT!** We recommend making a helper multiply function. Multiply can be done iteratively. Feel free to ask us questions in our office hours, on Piazza, or in the weekly recitation. Make sure that fact.s is in a UNIX-readble format (no DOS/Windows nonsense).

Deliverables

Turn in **ALL** of your files in T-Square in **.zip** or **.tar.gz** format.

- `answers.txt`, which has your answers for Problem 2, question 1.
- `fact.s`, which has your assembly code.
- `gt16as`, the gt16 assembler, either in its original directory or the root hw directory
- `gt16text`, the gt16 simulator, either in its original directory or the root hw directory

The TAs should be able to type `./gt16as fact.s` and then `./gt16text fact.lc` to run your code. If you cannot do this with your submission, then you have done something wrong.

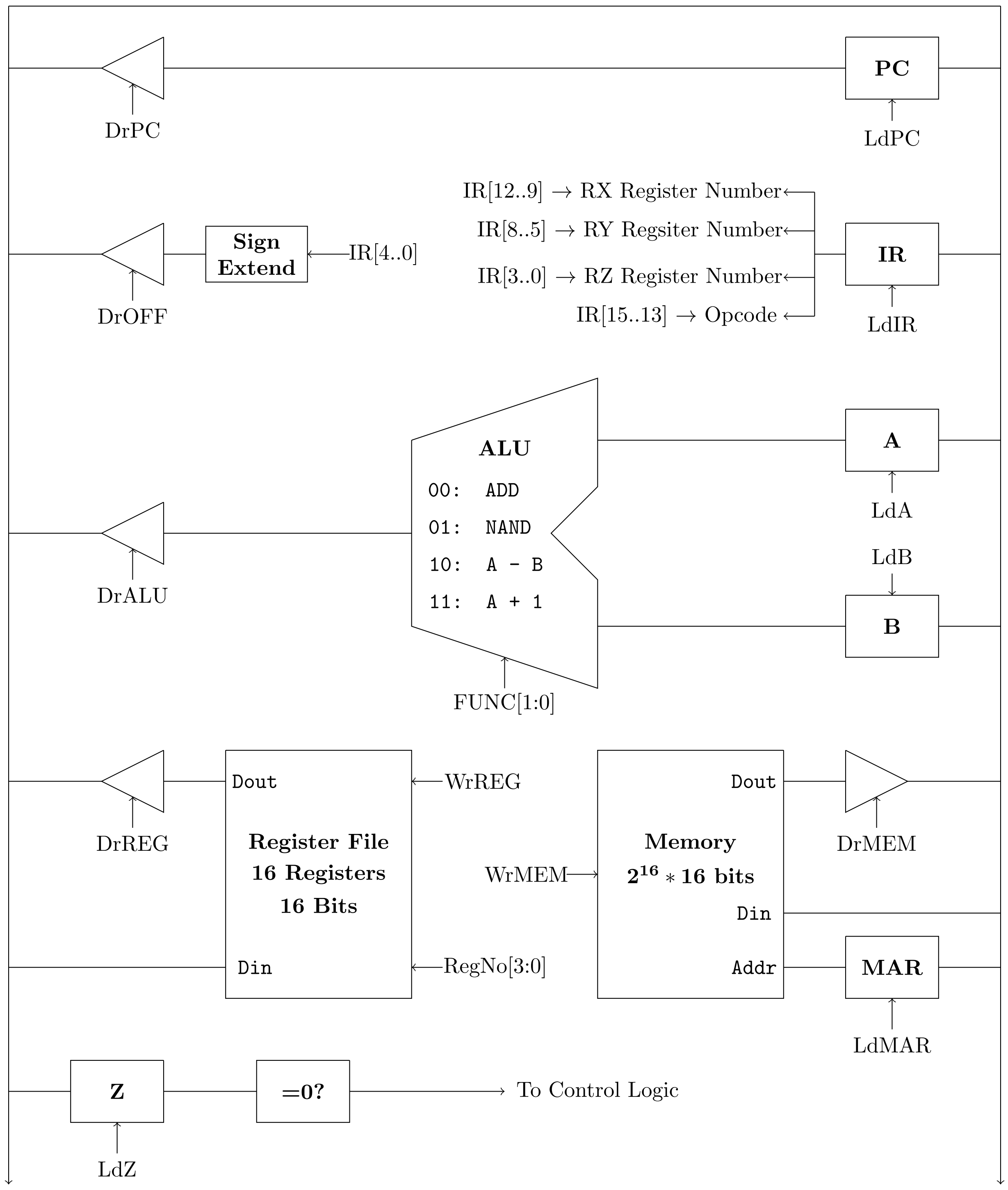

Figure 1: Datapath for the LC 2200 Processor

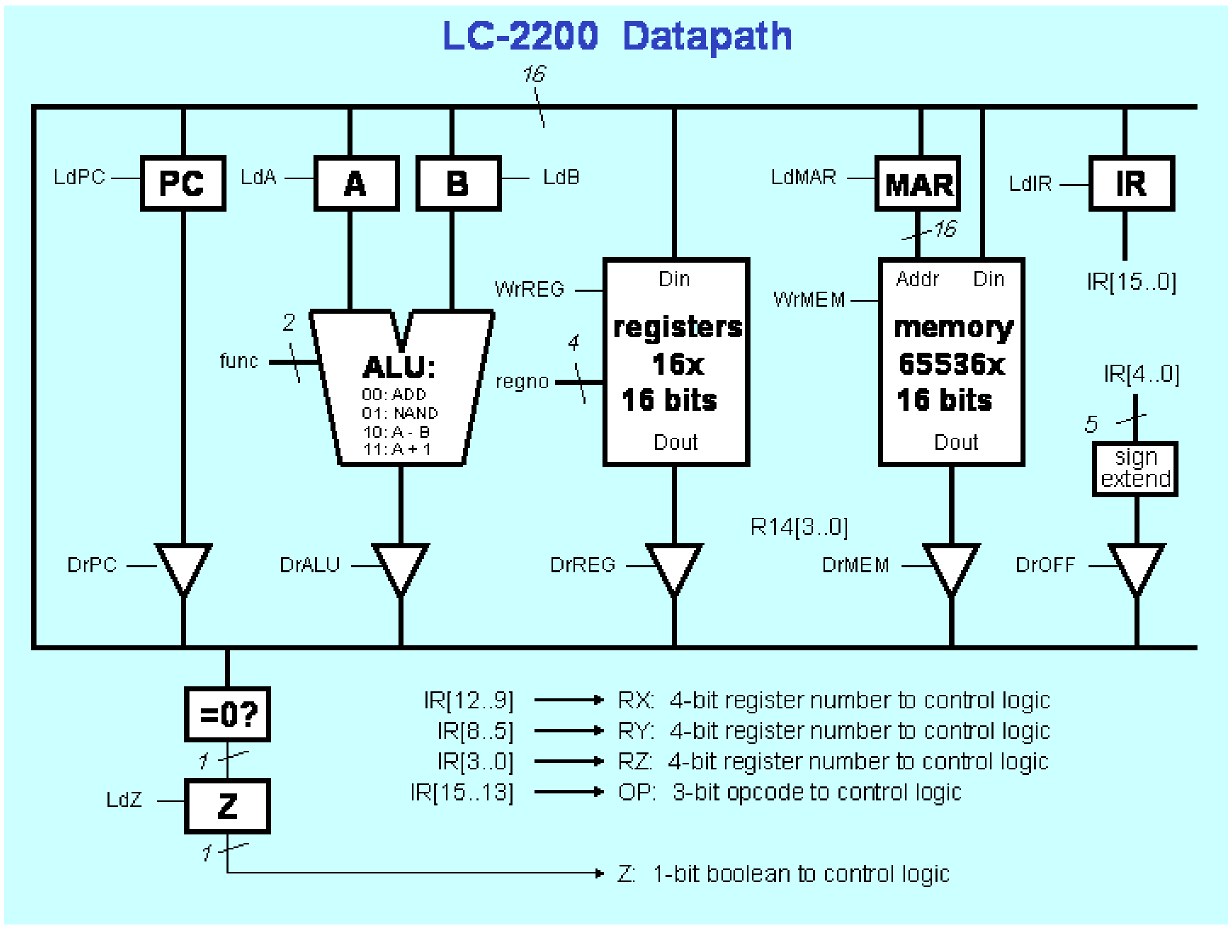

Figure 2: Second Diagram for the LC 2200 Processor Datapath