ICSI 404 – Assignment 2: ALU Design

The second programming assignment builds on the first to complete an ALU design. Now that we have created a long-word simulator with bitwise logical operations and bit-shift capabilities, we can **extend it to simple logic-based arithmetic processing**, namely addition and subtraction. We need to remember that logic-based addition (subtraction is only addition of the 2's complement of the integer subtracted) works solely on the bit-vector patterns – without any concern whether the operands are unsigned or signed. The only difference is made in the interpretation of the bit-vector and implementation of the overflow logic for the unsigned or signed cases.

The other aspect of ALU design is that we need to decide upon **a set of arithmetic and logic operations** the ALU will perform on the operands (long-words in our case). We also need a few **encoded control bits** as a chooser for the ALU operation to be triggered. As noted in our discussions about assembly and machine instructions, these control bits come from either the opcode field of the machine instruction or the secondary function part, in case multiple operations are grouped to share an opcode. To keep it simple, all instructions using the ALU will have distinct opcodes in our design.

For our 32-bit machine, the `ALU` class implementation **must follow these specifications**:

1. The `ALU` class must be **a new class that does not inherit from (i.e., extend) any other class**.

2. A ALU object **must be instantiated** before it can perform an operation. In other words, you should not plan on developing ALU as a static class. There are two reasons behind this choice. First, most modern general-purpose CPUs have multiple ALUs to exploit instruction level parallelism. Thus, you may like to extend your code later to emulate a more featureful CPU with multiple ALU instances. The second and more compelling reason is that the following boolean flags must be maintained for an ALU instance. The values of these flags are to be automatically updated after every operation, based on the specifications below:
    - **The zero flag (ZF)** should be set to 1 (true) when the ALU operation results in a long-word valued 0 (i.e., all bits are cleared). It should not matter whether the operation in arithmetic (add/subtract) or logical (including bit shifts).
    - **The negative flag (NF)** should be set to 1 (true) when the ALU operation results in a long-word that is considered negative under 2's complement signed interpretation (i.e., MSB is negative). It should not matter whether the operation in arithmetic (add/subtract) or logical (including bit shifts).
    - **The carry-out flag (CF)** should be set to 1 (true) when ALU operation results in a carry-out from the MSB position. This flag is meant to indicate an overflow resulting from unsigned addition, but this flag must be computed for every add/subtract operation as the machine opcode only works on bit-vector without considering their interpretation.
    - **The overflow flag (OF)** should be set to 1 (true) when ALU operation results in an overflow condition based on 2's complement arithmetic. Recall that for ripple-carry addition, this is indicated by a mismatch (XOR) in the carry-in and carry-out bits in the MSB position. While we expect none of the logical operations to set this bit, there is only one exception. The left shift operation, with shift amount 1-bit, is often substituted for a multiplication with 2. This

is particularly useful for shift-and-add multiplication. When such a multiply-by-2 operation flips the sign of the result, it clearly is an overflow, and the overflow flag must be set. While the same situation is also possible for multiple-bit left-shift (k-bit shift is multiplying with 2^k), the overflow flag is not to be set there as one cannot pinpoint after exactly how many bit shifts causes the overflow.

The exact way how to store these four flags is left to you. In the simplest implementation, you may like to keep four boolean variables. However, real CPU hardware dedicate a whole register (as long as the machine word) to store all necessary flags, including the four just described. This register goes by different names such as **flag register**, **status register** or **program status word (PSW)**. Some bits of the PSW may be left unused. So, another option is to keep a long-word for the PSW among the data field of the ALU object and use only the least significant four bits (0-3). This second approach mimics the hardware implementation more closely. Since it is easy to forget which bit is used for which flag, use of `enum` and the `ordinal` method of its constant instances may come handy.

3. The **ripple carry adder logic**, as discussed in class, **must be used** for both addition and subtraction of long-words. In other words, **you may not use Java's built-in add/subtraction** by extracting the value of the operands from the representative bit-vectors using `getSigned` or `getUnsigned` methods, and then injecting the result back using `set` method. The bit-by-bit addition logic makes use of Boolean XOR (since we are using `boolean` type for a bit). You may implement this any way you like. While you can compose '&&' and '||' operators to create logical XOR, there are simpler alternatives such as making use of the bitwise '^' operator or simply comparing two Boolean values using '!=' . In any case, you need a Boolean argument (apart from the two long-words) serving as the carry-in to the LSB (which is '0' for addition and '1' for subtraction, assuming the bits are flipped for the operand being subtracted). The ripple-carry adder method should look somewhat like:

```
private LongWord rippleCarryAdd(LongWord a, LongWord b,
                                boolean cin)
```

Note that this method should be **responsible for setting the CF and OF flag bits**. Also, it should be made private for internal use by the ALU when the control bits trigger an add/subtract operation. The ALU's public interface should only involve control codes and necessary operands as inputs and the result as well as flag bits as outputs.

4. You **must** create the public interface for your class by implementing the following methods:
   a) At creation, the flag bits, or the status register (depending on your implementation choice) must be cleared (initialized to '0').
   b) Implement the **accessor(s)** based on your implementation of the condition flags. You may either have an accessor for each of the flags (e.g., getZF, getNF, getCF and getOF) or have a single accessor for the whole program status word (getStatus). When your CPU needs to check the Boolean value of a flag (in the next assignment), it needs to extract the correct bit in the second case. The same applies to your tester program in the current one.
   c) Implement the only **mutator** which performs the ALU operation based on the given control code. The prototype may look like

```
public LongWord operate(int code, LongWord op1, LongWord op2)
```

Note that this mutator method not only returns the result of the ALU operation as a long-word, but also has an appropriate side-effect on the condition flags. The control codes must be exactly as the following table – the machine opcodes will not match otherwise:

Operation	Machine opcode	ALU code	OP1	OP2	ZF	NF	CF	OF
AND	1000	000 (0)	Operand1	Operand2	if zero	if -ive	X	X
OR	1001	001 (1)	Operand1	Operand2	if zero	if -ive	X	X
XOR NOT	1010 1010	010 (2) 010 (2)	Operand1 Operand	Operand2 11…1 (-1)	if zero if zero	if -ive if -ive	X X	X X
ADD	1011	011 (3)	Operand1	Operand2	if zero	if -ive	Cout(31)	Overflow
SUB	1100	100 (4)	Operand1	Operand2	if zero	if -ive	Cout(31)	Overflow
SLL	1101	101 (5)	Operand	amount	if zero	if -ive	X	amontt=1 and sign-flip
SRL	1110	110 (6)	Operand	amount	if zero	X	X	X
SRA	1111	111 (7)	Operand	amount	if zero	if -ive	X	X

In all methods, you **must validate inputs** where appropriate.

You must provide a test file (TestALU.java) that implements void `runTests` method and call it from your main, along with your existing tests. As with the other tests, these tests must be independent of each other and there must be reasonable coverage. You cannot reasonably test all of the billions of possible combinations, but you can test a few representative samples.

In case you have updated the source code in your LongWord.java file, please make sure to include the compatible updated version.

You must submit buildable .java files for credit.

Rubric	Poor	OK	Good	Great
Comments	None/Excessive (0)	“What” not “Why”, few (5)	Some “what” comments or missing some (7)	Anything not obvious has reasoning (10)
Variable/Function naming	Single letters everywhere (0)	Lots of abbreviations (5)	Full words most of the time (8)	Full words, descriptive (10)
Unit Tests	None (0)	Partial Coverage (7)	All methods covered, needs more cases (13)	All methods/cases covered (20)
Status code accessor	None (0)	Some implemented (5)	All implemented, some fail (7)	All implemented, all tests pass (10)
ALU Control Codes	None(0)	Some implemented (5)	Implemented, wrong (7)	Implemented, correct (10)
ALU logical (AND/OR/XOR)	None(0)	Some implemented (5)	Implemented, wrong (7)	Implemented, correct (10)
ALU Shifts (SLL/SRL/SRA)	None(0)	Some implemented (5)	Implemented, wrong (7)	Implemented, correct (10)
Ripple carry adder	None(0)	Some implemented (5)	Implemented, wrong (7)	Implemented, correct (10)
ALU add/subtract (call to ripple-carry)	None(0)	Some implemented (5)	Implemented, wrong (7)	Implemented, correct (10)