CS2630: Computer Organization
Project 1
MiniMa: (mini) assembler written in Java

Goals for this assignment
- Translate MAL instructions to TAL, and TAL to binary
- Resolve the addresses of branch and jump labels
- Build an assembler

Before you start

This project is fairly involved, so start early. To be prepared, you should:
- read the Week 3 readings and notes
- complete the "Stored Programs" activity
- complete Quiz 3.

Read the document as soon as you can and ask questions in Debug Your Brain/discussion board/office hours.

It is possible to get MiniMa working one phase at a time (there are 3 phases), so you can pace yourself.

If you are working on this homework as a group of 2 students, then
- submit only one copy to ICON “Project 1”
- by Sep 13, add yourself to the same Project 1 Group:
  https://uiowa.instructure.com/courses/65872/groups#tab-5475

Setup

You can clone (or download) the project from https://github.uiowa.edu/cs2630-assignments/minima.git

If you are using NetBeans, setup instructions are here:
https://piazza.com/class/j6l8nniwzot5ys?cid=37

If you are using IntelliJ, setup instructions are here:
https://piazza.com/class/j6l8nniwzot5ys?cid=38
A note on collaboration: We encourage you and your partner to create a github.uiowa.edu repository to collaborate on your code more effectively. If you do so, you must mark your repository Private. Repositories marked Public will be considered an intent to cheat by sharing code with other students.

Introduction

In this project, you will be writing some components of a basic assembler for MIPS, called MiniMa (mini MIPS assembler). You will be writing MiniMa in Java.

The input to MiniMa is an array of Instruction objects. The Instruction class has several fields indicating different aspects of the instruction. MiniMa does not include a parser to turn a text file into Instruction objects, so you will write programs using an InstructionFactory that creates Instructions.

```
public class Instruction {
    public final ID instruction_id;  // id indicating the instruction
    public final int rd;              // register number destination
    public final int rs;              // register number source
    public final int rt;              // register number secondary source
    public final int immediate;       // immediate, may use up to 32 bits
    public final int jump_address;    // jump address (not used, so it is always 0)
    public final int shift_amount;    // shift amount (not used, so it is always 0)
    public final String label;       // label for line
    public final String branch_label; // label used by branch or jump instructions
}
```

MiniMa has three basic phases for translating a MIPS program into binary. The next three sections describe these phases. The section after that, “What you need to do”, will describe your job in Project 1.

1. Convert MAL to TAL

In this phase, MiniMa converts any pseudo instructions into TAL instructions. Specifically, MiniMa creates a new output array of Instruction objects and stores the TAL instructions into it in order. For any true instruction in the input, MiniMa just copies it from the input to the output. For any pseudo instruction, MiniMa writes 1-3 real instructions into the output.

Examples:

Ex a)
label2: addu $t0,$zero,$zero

This instruction will be provided to you as the Instruction object:

```
Instruction_id  rd  rs  rt  imm  jump_address  shift_amount  label  branch_label
addu           8   0   0   0     0               0       "label2"  0
```
Because this instruction is already a TAL instruction, you will just copy it into the output array.

Ex b)  
```
blt $s0, $t0, label3
```

This pseudo instruction, will be provided to you as the instruction object:

<table>
<thead>
<tr>
<th>Instruction_id</th>
<th>rd</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
<th>jump_address</th>
<th>shift_amount</th>
<th>label</th>
<th>branch_label</th>
</tr>
</thead>
<tbody>
<tr>
<td>blt</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>&quot;label3&quot;</td>
</tr>
</tbody>
</table>

Note that label="" because the line the instruction was on is unlabeled.

This instruction is a pseudo instruction, so we must translate it to TAL instructions. In this case:
```
slt $at,$s0,$t0
bne $at,$zero,label3
```

Which you will represent with the following two Instruction objects.

<table>
<thead>
<tr>
<th>Instruction_id</th>
<th>rd</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
<th>jump_address</th>
<th>shift_amount</th>
<th>label</th>
<th>branch_label</th>
</tr>
</thead>
<tbody>
<tr>
<td>slt</td>
<td>1</td>
<td>16</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>bne</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>&quot;label3&quot;</td>
</tr>
</tbody>
</table>

We used $at (the assembler register) to store the result of the comparison. Since MIPS programmers are not allowed to use $at themselves, we know we can safely use it for passing data between generated TAL instructions.

IMPORTANT: notice that branch instructions do NOT have an Immediate in phase 1. Rather, they specify the target using branch_label. In phase 2, the branch_label will get translated into the correct immediate.

You must also make sure that you detect I-type instructions that use an immediate using more than the bottom 16 bits of the immediate field and translate them to the appropriate sequence of instructions.

This is just the end of the Part 1 background information. There are no tasks to do yet, so keep reading!

---

2. Convert labels into addresses
This phase converts logical labels into actual addresses. This process requires two passes over the instruction array.

- Pass one: find the mapping of labels to the PC where that label occurs
- Pass two: for each instruction with a non-zero branch_label (jumps and branches) calculate the appropriate address using the mapping.

**Example**

before phase2: branch target for branch instructions indicated using branch_label field

<table>
<thead>
<tr>
<th>Address</th>
<th>Label</th>
<th>Instruction</th>
<th>Important instruction field values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400000</td>
<td>label1</td>
<td>addu $t0,$t0,$t1</td>
<td>label=&quot;label1&quot;</td>
</tr>
<tr>
<td>0x00400004</td>
<td></td>
<td>ori $t0,$t0,0xFF</td>
<td></td>
</tr>
<tr>
<td>0x00400008</td>
<td>label2</td>
<td>beq $t0,$t2,label1</td>
<td>label=&quot;label2&quot;, branch_label=&quot;label1&quot;</td>
</tr>
<tr>
<td>0x0040000C</td>
<td></td>
<td>addiu $t1,$t1,-1</td>
<td></td>
</tr>
<tr>
<td>0x00400010</td>
<td>label3</td>
<td>addiu $t2,$t2,-1</td>
<td>label=&quot;label3&quot;</td>
</tr>
</tbody>
</table>

after phase2: branch target for branch instructions indicated using immediate field

<table>
<thead>
<tr>
<th>Address</th>
<th>Label</th>
<th>Instruction</th>
<th>Important instruction field values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400000</td>
<td>label1</td>
<td>addu $t0,$t0,$t1</td>
<td></td>
</tr>
<tr>
<td>0x00400004</td>
<td></td>
<td>ori $t0,$t0,0xFF</td>
<td></td>
</tr>
<tr>
<td>0x00400008</td>
<td>label2</td>
<td>beq $t0,$t2,-3</td>
<td>immediate = -3</td>
</tr>
<tr>
<td>0x0040000C</td>
<td></td>
<td>addiu $t1,$t1,-1</td>
<td></td>
</tr>
<tr>
<td>0x00400010</td>
<td>label3</td>
<td>addiu $t2,$t2,-1</td>
<td></td>
</tr>
</tbody>
</table>

This is just the end of the Part 2 background information. There are no tasks to do yet, so keep reading!

3. Translate instructions to binary

This phase converts each Instruction to a 32-bit integer using the MIPS instruction encoding, as specified by the MIPS reference card. We will be able to test the output of this final phase by using MARs to translate the same input instructions and compare them byte-for-byte.

To limit the work you have to do, MiniMa only needs to support the following instructions:

**Instruction**

- addiu (whether it is MAL depends on imm)
- addu
- or
- beq
- bne
- slt
- lui
- ori (whether it is MAL depends on imm)
- blt (always MAL)
- bge (always MAL)
This is just the end of the Part 3 background information. Your tasks begin in the next section.

What you need to do

1. (10 points) You will complete the implementation of phase 1 by modifying the file Phase1.java.

```java
/* Translates the MAL instruction to 1-3 TAL instructions
 * and returns the TAL instructions in a list
 * mals: input program as a list of Instruction objects
 * returns a list of TAL instructions (should be same size or longer than input list)
 */
public static List<Instruction> mal_to_tal(List<Instruction> mals)
```

If a MAL Instruction is already in TAL format, then you should just copy that Instruction object into your output list. **You should not change input instructions.**

- If you need to copy an instruction verbatim, then use Instruction.copy.
- If you need to create a new instruction with different fields set, use the InstructionFactory **not** the Instruction constructor (see AssemblerTest.test1() for an example of using the InstructionFactory).

If a MAL Instruction is a pseudo-instruction, such as blt, then you should create the TAL Instructions that it translates to in order in the buffer and return the number of instructions.

You must check I-type instructions for the case where the immediate does not fit into 16 bits and translate it to lui, ori, followed by the appropriate r-type instruction. Remember: the 16-bit immediate check does not need to be done on branch instructions because they do not have immediates in phase 1 (see phase 1 description above).

Use the following translations for pseudo instructions. These translations are the same as MARS uses.

1. Instruction passed to mal_to_tal: addiu r1,r2,Immediate  # when immediate is too large!
   =>
   Instructions returned from mal_to_tal:
   lui $at,Upper 16-bit immediate
   ori $at,$at,Lower 16-bit immediate
   addu r1,r2,$at

   The above formula shown for addiu also applies to ori.
   Note that lui will never be given an immediate too large because it is not well-defined for more than 16 bits (MARS also disallows lui with >16-bit immediate, try it).
II. Instruction passed to mal_to_tal:
blt r1,r2,gohere
=>
Instructions returned from mal_to_tal:
slt $at,r1,r2
bne $at,$zero,gohere

III. Instruction passed to mal_to_tal:
bge rs,rt,hello
=>
Instructions returned from mal_to_tal:
slt $at,r1,r2
beq $at,$zero,hello

2. (10 points) You will complete the implementation of phase 2 by implementing the 2-pass address resolution in a function called resolve_addresses.

/* Returns a list of copies of the Instructions with the
 * immediate field of the instruction filled in
 * with the address calculated from the branch_label.
 * The instruction should not be changed if it is not a branch instruction.
 * unresolved: list of instructions without resolved addresses
 * first_pc: address where the first instruction will eventually be placed in memory
 */
public static List<Instruction> resolve_addresses(List<Instruction> unresolved, int first_pc)

Using our example from the phase 2 description:

<table>
<thead>
<tr>
<th>Address</th>
<th>Label</th>
<th>Instruction</th>
<th>Important instruction field values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400000</td>
<td>label1</td>
<td>addu $t0,$t0,$t1</td>
<td>label=&quot;label1&quot;</td>
</tr>
<tr>
<td>0x00400004</td>
<td></td>
<td>ori $t0,$t0,0xFF</td>
<td></td>
</tr>
<tr>
<td>0x00400008</td>
<td>label2</td>
<td>beq $t0,$t2,label1</td>
<td>label=&quot;label2&quot;, branch_label=1</td>
</tr>
<tr>
<td>0x0040000C</td>
<td></td>
<td>addiu $t1,$t1,-1</td>
<td></td>
</tr>
<tr>
<td>0x00400010</td>
<td>label3</td>
<td>addiu $t2,$t2,-1</td>
<td>label=&quot;label3&quot;</td>
</tr>
</tbody>
</table>

The first_pc argument is the address where the first instruction in unresolved would be written to memory after phase 3. Using the above example, resolve_addresses would be called with first_pc=0x00400000.

Refer to the earlier description of phase 2 for how to calculate the immediate field.
3. (10 points) You will complete the implementation of phase 3 by implementing the function `translation_instruction`.

```java
/* Translate each Instruction object into
* a 32-bit number.
* * tals: list of Instructions to translate
* * returns a list of instructions in their 32-bit binary representation
*/
public static List<Integer> translate_instructions(List<Instruction> tals)
```

This function produces an encoding of each R-type or I-type instruction. Refer to the MIPS reference sheet for format of the 32-bit format.

**Make sure to set unused fields to the following defaults**
- int field is 0
- String field is ""

Requiring these default values is just a current limitation of the test code.

**Running and testing your code**

The three phases are run on a test case by running the JUnit test file AssemblerTest.java. The provided test, `test1`, will run each of the 3 phases in order. Each phase is followed by a check that the output is correct up to that point. If the test fails, JUnit will produce a useful error message.

You can add your own tests to AssemblerTest.java. Use `test1` as an example; notice that it uses a helper function to actually run the tests. TIP: you can calculate the expected output of Phase 3 by assembling your test program in MARS and looking at the Code column in the Text Segment.

(5 points) You must add at least one additional test to AssemblerTest.java. The test should not be of a trivially similar MIPS program to the one in `test1`. You must test things that `test1` doesn’t cover, such as
- other input instructions
- I-type instructions that do not exceed 16 bits
- other label positions

You are responsible for testing your assembler beyond `test1`. We will use more tests during grading.
Note that the MiniMa does not currently have a parser, so you must provide the input program as a sequence Instruction objects created using the InstructionFactory (for an example, see the list called input in test1() of AssemblerTest.java).

What to submit

For full credit your MiniMa implementation must compile and run. You should not depend on modifications to Instruction.java or add additional java files.

Required:
- Phase1.java (method implemented)
- Phase2.java (method implemented)
- Phase3.java (method implemented)
- AssemblerTest.java (at least one new test case added)

Good job!

Once you have completed this project, if you added support for the rest of the MIPS instructions you could replace the assembler of MARS (i.e., the code behind this button).