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

Due February 16, 2016, 11:59pm

Goals for this assignment

- Be able to translate MAL instructions to TAL
- Be able to translate TAL to binary
- Learn about how an assembler calculates addresses for branches

Before you start

This project is fairly involved, so start early. 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"**
- **submit ICON "Project 1 Team assignments" by Feb 9**

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, a language you are already comfortable with, so that you can focus your attention on translation of MIPs programs.

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 in terms of Instruction objects.

```java
public class Instruction {
    int instruction_id;   // id indicating the instruction
    int rd;              // register number
    int rs;              // register number
    int rt;              // register number
    int immediate;       // immediate, may use up to 32 bits
    int jump_address;    // jump address  (not used, so it is always 0)
    int shift_amount;    // shift amount (not used, so it is always 0)
    int label_id;        // 0=no label on this line; nonzero is a unique id
    int 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:

4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes
Instruction_id	Rd	Rs	Rt	Imm	Jump_addr	Shift_amt	Label_id	Branch_label
2	8	0	0	0	0	0	2	0

Note that label_id=2 because the line the instruction was on was labeled with label2.

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:

4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes
Instruction_id	Rd	Rs	Rt	Imm	Jump_addr	Shift_amt	Label_id	Branch_label
100	0	16	8	0	0	0	0	3

Note that label_id=0 because the line the instruction was on is unlabeled. Branch_label=3 because the instruction is a branch with target "label3".

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.

4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes
Instruction_id	Rd	Rs	Rt	Imm	Jump_addr	Shift_amt	Label_id	Branch_label
8	1	16	8	0	0	0	0	0
4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes
Instruction_id	Rd	Rs	Rt	Imm	Jump_addr	Shift_amt	Label_id	Branch_label
6	0	1	0	0	0	0	0	3

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.

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

Address	Label	Instruction	Important instruction field values
0x00400000	label1:	addu $t0,$t0,$t1	label_id=1
0x00400004		ori $t0,$t0,0xFF	
0x00400008	label2:	beq $t0,$t2,label1	label_id=2, branch_label=1, imm=0
0x0040000C		addiu $t1,$t1,-1	

0x00400010	label3:	addiu $t2,$t2,-1	label_id=3

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

Address	Label	Instruction	Important instruction field values
0x00400000	label1:	addu $t0,$t0,$t1	
0x00400004		ori $t0,$t0,0xFF	
0x00400008	label2:	beq $t0,$t2,-3	imm = -3
0x0040000C		addiu $t1,$t1,-1	
0x00400010	label3:	addiu $t2,$t2,-1	

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.

Here are the ID numbers for the `instruction_id` field of the Instruction objects.
IMPORTANT: these IDs are MiniMa's internal encoding for the type of an Instruction object. `Instruction_id` is not the same as the opcode or func fields of binary MIPS instructions.

MiniMa only needs to support the following instructions

instruction_id (in the Instruction object)	Instruction
1	addiu (might be pseudo instruction)
2	addu
3	or
5	beq
6	bne
8	slt
9	lui
10	ori (might be pseudo instruction)
100	blt (always a pseudo instruction)
101	bge (always a pseudo instruction)

What you need to do

1. (10 points) You will complete the implementation of phase 1 by modifying the file `Phase1.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 should not change input instructions. If you need to copy an instruction in any phase, then use Instruction.copy.**

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.

   I.   Instruction passed to `mal_to_tal` argument `instr`:
`addiu rt,rs,Immediate`   # when Imm is too large!
=>
Instructions written to `buffer`:
`lui $at,Upper 16-bit immediate`
`ori $at,$at,Lower 16-bit immediate`
`addu rt,rs,$at`   // we are referring to rt above, for addu's rd field

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` argument `instr`:
`blt rs,rt,labelx`
=>
Instructions written to `buffer`:
`slt $at,rs,rt`
`bne $at,$zero,labelx`

and `mal_to_tal` returns 2

   III.   Instruction passed to `mal_to_tal` argument `instr`:

```
bge rs,rt,labelx
=>
```
Instructions written to buffer:
```
slt $at,rs,rt
beq $at,$zero,labelx
```

and `mal_to_tal` returns 2

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:

Address	Label	Instruction	Important instruction field values
0x00400000	label1:	addu $t0,$t0,$t1	label_id=1
0x00400004		ori $t0,$t0,0xFF	
0x00400008	label2:	beq $t0,$t2,label1	label_id=2, branch_label=1, imm=0
0x0040000C		addiu $t1,$t1,-1	
0x00400010	label3:	addiu $t2,$t2,-1	label_id=3

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`.

```
/* 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 encoding of each R-type or I-type instruction. Refer to the MIPS reference sheet for format of the 32-bit format. Again, note that the **opcode** used in the 32-bit representation comes from the MIPS reference sheet, and it is completely different from the our assembler's internal `instruction_id`.

Be sure that unused fields are set to 0.

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.

(5 points) You must add at least one additional test to AssemblerTest.java. Significantly different means you must test things that test1 doesn't cover, such as other input instructions and I-type instructions that do not exceed 16 bits.

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 (see the list called `input` in test1).

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
- Phase2.java
- Phase3.java
- AssemberTest.java