Project 1

MIPS Assembler Programming in Java

Acknowledgment: Adapted from Brandon Mayers (Univ. of Iowa)

Goals

- Learn how to translate MAL (MIPS assembly language) instructions to TAL (true assembly language)
- Learn how to translate TAL to machine code (binary)
- Learn how an assembler calculates addresses for (labeled) branches

Before you start

This project is fairly involved, so start early. Read the document as soon as you can and ask questions in TA office hours. It is possible to get it working one phase at a time (there are 3 phases), so you can pace yourself. Download the assembler code skeleton here.

Assembler Skeleton

In this project, you will be writing some components of a basic assembler for MIPS. You will be writing it in Java, a language you are already comfortable with, so that you can focus your attention on the translation of MIPS programs.

The input to your assembler is an array of Instruction objects. The Instruction class has several fields indicating different aspects of the instruction. Note it does not include a parser to turn a text file into Instruction objects, so you will write programs in terms of Instruction objects.

```
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
}
```

The assembler 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.

Phase 1: Convert MAL to TAL

In this phase, the assembler converts any pseudo instructions into TAL instructions. Specifically, a new output array of Instruction objects is created to store the TAL instructions into it in the original order. For any true instruction in the input, you just need to copy the instruction from the input to the output. For any pseudo instruction, you will need to translate it into 1-3 real instructions and store them in the output. You may refer to the full list of MIPS pseudo-instructions here.

Examples:

a) label2: addu $t0, $0, $0

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

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 label 2. Because this instruction is already a TAL instruction, you will just copy it into the output array.

b) blt $s0, $t0, label3

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

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, $0, label3
```

Those TAL instructions you will be represented by the following two Instruction objects.

Instruction_id	Rd	Rs	Rt	Imm	Jump_addr	Shift_amt	Label_id	Branch_label
8	1	16	8	0	0	0	0	0
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.

Phase 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 Phase 2: 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 Phase 2: 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	

Phase 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 used as internal encoding for the type of an Instruction object. Instruction_id is not the same as the **opcode** or **funct** fields of binary MIPS instructions.

The assembler 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) 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 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 immediate in phase 1 (see phase 1 description above).

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

```
1) 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).

```
2) 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

3) 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. You will complete the implementation of phase 2 by implementing the 2- pass address resolution in a function called **resolve_addresses** (Phase2.java).

```
/* 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. You will complete the implementation of phase 3 by implementing the function **translation_instruction** (Phase3.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 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 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.

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 assembler skeleton 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 implementation must compile and run correctly. **You should not depend on modifications to Instruction.java** or add additional java files!

You need to modify the following files:

```
Phase1.java
Phase2.java
Phase3.java
AssemberTest.java
```

How to run your tests

If you are not familiar with Junit tests, you can follow those steps:

1. go to https://github.com/junit-team/junit4/wiki/download-and-install and download junit.jar and add it to your project folder. (for example: **junit-4.13-beta-2.jar**)

2. go to https://code.google.com/archive/p/hamcrest/downloads and download the hamcrest-all jar and add it to your project folder (for example: **hamcrest-all-1.3.jar**)

Now when you compile and test, you just need to include these libraries on your classpath. So, to compile, do (inside the project directory):

```
$ javac -cp .:hamcrest-all-1.3.jar:junit-4.13-beta-2.jar *.java
```

To run the tests, you need to use one of the JUnit runners and pass AssemblerTest as test class:

```
$ java -cp .:hamcrest-all-1.3.jar:junit-4.13-beta-2.jar
org.junit.runner.JUnitCore AssemblerTest
```