CS2630: Computer Organization
Homework 3
Miniaa: (mini) ARM assembler written in C

Table of Contents

1 Goals for this assignment

- Check whether constants in an ARM program are valid
- Resolve the addresses of branch labels
- Use bitwise operations to build 32-bit encodings of instructions
- Build an assembler for ARM

2 Before you start

This project is fairly involved, so start early. To be prepared, you should:

- review the readings from Ch 6
- complete the prelab/lab on bitwise operations
- complete the prelab/lab on machine code
- complete the *Stored Programs* inquiry activity

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

The testing infrastructure is designed so that it is possible to test Miniaa's three phases completely independently. We've designed it this way so you can complete Miniaa gradually. You can get a little bit working at a time in whatever order you want.

3 Hacking the rubric

See the assignment on ICON for the rubric. Notice that non-compiling programs and programs that don't pass any tests **earn no more than 50 points (an F)**. Therefore, we cannot stress enough that you should approach this project incrementally. Consider the following stories:

- **Team 0xBA51C:** "We got a few tests working each day, running the code frequently to check. We ended up passing all tests except a couple tricky cases." Grade A- ($\geq$ 90 %).
- **Team 0xCOFFEE**: "We wrote all functionality in one all-night coding fever. We are pretty sure we covered every single case for all 3 phases." Deeply in denial, they never

compiled or ran their code until the due date. When they did, they found it didn't even compile. They didn't have time to fix it and had to turn it in as is. Grade: F (≤ 50%).

4 Setup

You can git clone the project from
https://research-git.uiowa.edu/cs2630-assignments/miniaa.git
or you can download it from this link

Log into the research-git.uiowa.edu website using your hawkid and password.

4.1 Compiling and running the code

You must have libcheck installed. We've already installed it for you on the remote Linux lab computers (i.e., the ones you connect to with FastX or ssh).

```
make AssemblerTest
./AssemblerTest
```

You should see the unit tests run and fail.

```
…
AssemblerTest.c:79:F:Core:test9Phase3:0: Assertion 'expectedP3[i] == translated[i]' failed
AssemblerTest.c:79:F:Core:test10Phase3:0: Assertion 'expectedP3[i] == translated[i]' failed
AssemblerTest.c:79:F:Core:test11Phase3:0: Assertion 'expectedP3[i] == translated[i]' failed
Phase 3: # of failed: 7
```

4.2 Recommendation: version control

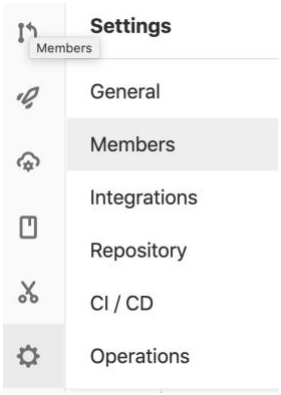
We encourage (but not require) you to create a https://research-git.uiowa.edu/ repository to work on your code more efficiently. Example steps to set up:

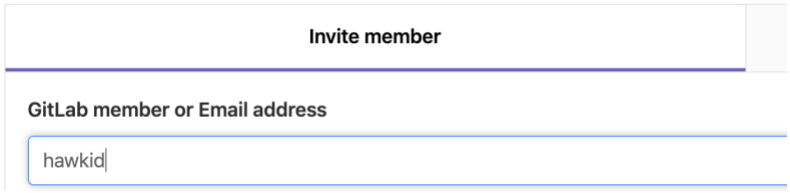
1. create a **New project**, e.g., called cs2630-hw3. **You must set your Gitlab repository as *Private***. Repositories marked ***Internal*** or ***Public*** will be considered an intent to cheat by sharing code with other students.

2. You can clone from the starter code, then push, with these commands.
   - git clone https://research-git.uiowa.edu/cs2630-assignments/miniaa
   - git remote rm origin
   - git remote add origin https://research-git.uiowa.edu/***hawkid***/cs2630-hw3
   - git push origin main

4.3 If you are working in a pair

We encourage you and your partner to create a GitLab repository to make collaboration easier. One partner should make the ***private*** repo and invite the other student.





5 Introduction

In this project, you will be writing some components of a basic assembler for ARM, called Miniaa (mini ARM assembler). You will be writing Miniaa in C.

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

```
struct Instruction {
    enum ID instruction_id;   // id indicating the instruction
    enum COND cond;
    uint8_t rd;            // register number destination
    uint8_t rn;            // register number source
    uint8_t rm;            // register number secondary source
    uint8_t rs;            // register number shift amount
    int32_t immediate;     // immediate, may use up to 32 bits
    uint8_t shift_amount;  // shift amount
    char label[MAX_LABEL_LENGTH]; // label of the line this Instruction appears on
    char branch_label[MAX_LABEL_LENGTH];  // label used by branch or jump instructions
};
```

Miniaa has three basic phases for translating a ARM program into binary. The next three sections describe these phases. The section after that, “What you need to do”, will describe your job in this project.

5.1 Phase 1: Check for valid constants

In this phase, Miniaa checks for valid constants. It copies the instructions into a new list. Each instruction is either a) unmodified because it has no constant, b) unmodified because its constant is valid, c) modified to make the constant work, or d) replaced with “the invalid instruction” to indicate the constant was invalid.

5.1.1 Example of case (a)

```
label2: b label8
```

This instruction will be provided to you as the Instruction object. Notice that unused values are 0 for numbers.

Instruction_id	cond	rd	rn	rm	rs	imm	shift_amount	label	branch_label
b	AL	0	0	0	0	0	0	"label2"	“label8”

Because this instruction has no constant, you will just copy the instruction into the output list.

5.1.2 Example of case (b)

```
hello: subeq R7, R8, #4
```

This instruction will be provided to you as the Instruction object. Notice that unused strings are the empty string.

Instruction_id	cond	rd	rn	rm	rs	imm	shift_amount	label	branch_label
subi	EQ	7	8	0	0	4	0	"hello"	“”

Because this instruction has a constant you must check that can it can be encoded into an 8-bit immediate (imm8) and an even right rotation (rot). In this case, it can. Therefore you will copy the instruction into the output list.

5.1.3 Example of case (c)

```
sublt R0, R12, #-7
```

Instruction_id	cond	rd	rn	rm	rs	imm	shift_amount	label	branch_label
subi	LT	0	12	0	0	-7	0	""	“”

Because this instruction has a constant you must check that can it can be encoded into an 8-bit immediate (imm8) and an even right rotation (rot). In this case, it cannot. But since this is a sub instruction, we could try replace it with an add instruction with the constant negated. We now need to check if 7 can be encoded. In this case, it can. Therefore you will put the following instruction into the output list.

Instruction_id	cond	rd	rn	rm	rs	imm	shift_amount	label	branch_label
addi	LT	0	12	0	0	7	0	""	“”

5.1.4 Example of case (d)

```
orr R1, R15, #0x8811
```

Instruction_id	cond	rd	rn	rm	rs	imm	shift_amount	label	branch_label
ori	AL	1	15	0	0	0x8811	0	""	“”

This instruction has a constant, so we need to check if that constant can be encoded. In this case, there is no way to encode it by rotating an 8-bit immediate. Therefore, we need to put the invalid instruction into the output list in its place.

Instruction_id	cond	rd	rn	rm	rs	imm	shift_amount	label	branch_label
invalid	AL	0	0	0	0	0	0	""	“”

IMPORTANT: notice that branch instructions have Immediate=0 in phase 1. Instead, they specify the target using branch_label. In phase 2, the branch_label will get translated into the correct immediate and the branch_label will be made the empty string.

This is just the end of the Phase 1 background information. There are no tasks to do yet.

5.2 Phase 2: Convert labels into addresses

This phase converts logical labels into actual addresses.

5.2.1 Example

This example is made entirely out of branch instructions only so that we can fit a lot into a single example. Your program of course might have other instructions, too.

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

Address	**Label**	**Instruction**	**Important instruction field values**
0x00400000	a:	b a	label="a", branch_label=”a”
0x00400004		bne f	branch_label=”f”
0x00400008	c:	blt f	label="c", branch_label="f"
0x0040000C	d:	blt c	label="d", branch_label="c"
0x00400010		bge f	branch_label="f"
0x00400014	f:	b d	label="f", branch_label="d"

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

Address	Label	Instruction	Important instruction field values
0x00400000	a:	b a	imm = -2
0x00400004		bne f	imm = 2
0x00400008	c:	blt f	imm = 1
0x0040000C	d:	blt c	imm = -3
0x00400010		bge f	imm = -1
0x00400014	f:	b d	imm = -4

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

5.3 Phase 3: Translate instructions to binary

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

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

Instruction id	notes
lsl	only with shift by a constant
add	two register operands
addi	(add) register and a constant as operands
sub	two register operands
subi	(sub) register and a constant as operands
or	(orr) two register opeands
ori	(orr) register and a constant as operands
cmp	only two register operands
b	
invalid	used by Phase 1 to indicate an instruction was invalid

This is just the end of the Part 3 background information. Your tasks begin in the next section.

6 What you need to do

6.1 Phase 1 tasks

You will complete the implementation of phase 1 by modifying the file `Phase1.c`. **Do not modify** Phase1.h.

```
/* Checks if each instruction has a valid constant
```

```
 *         (representable as imm8 with an even right rotation)
 * Will change an ADDI to SUBI (or vice versa) if negating the constant
 * would allow it to achieve the above constraint
 *
 * unchecked: list of instructions that need to get checked
 * checked: an empty list. When the function returns it will be full of
 *      copies of okay instructions, adds turned to subs and vice versa when needed,
 *      and invalid instructions replacing instructions that had errors.
 */
void check_errors(struct ArrayList *unchecked, struct ArrayList *checked);
```

If an instruction doesn't have a constant, then you should just copy that Instruction object into the checked list.

If an instruction does have a constant, then you need to check that it would be representable in 8 bits and a rotation.

- For a description of various cases, see Section 5.1 above
- We've provided a function rotl32c, whose signature you can find in rotl32c.h. It rotates a 32-bit number ***left*** by a specified number of bits. You might find this useful in creating your algorithm for checking the constants.
- If you need to change the instruction (as in case 5.1.3), then you'll need to create an instruction with the correct values of its fields.
- If the constant is invalid, then put into the checked list an "invalid instruction". You can use INVALID() from InstructionFactory.h to create this instruction.

6.1.1 How do I create an instruction?

To create an Instruction with specific fields, see the functions in InstructionFactory.h. Find examples of those functions' usage in AssemblerTest.c. Alternatively, you could modify an existing Instruction object that is close to the one you want, by accessing its fields. We ***do not*** recommend that you call newInstruction() from Instruction.h. We've already provided all the functions you'll likely need in InstructionFactory.h.

6.1.2 What is an enum?

You'll see in Instruction.h that the instruction id and cond are enum types. An enum is basically a finite set of named values. For example, in the set of ID is add, addi, sub, subi, and the rest of the instructions. Example of checking if an instruction object is an add instruction:

#include "Instruction.h"

```
struct Instruction ins;
…
if (ins.instruction_id == add) {
      …
}
```

If you prefer, switch statements also work well with enums.

6.1.3 Tips

If in doubt about a case of translation there are things you can do before asking for help.
1) examine the test cases, and
2) assemble a test program in ARMSim# to see what it produces

You *only* need to support the instructions that are in Instruction.h.

6.2 Phase 2 tasks

You will complete the implementation of phase 2 by implementing `resolve_addresses` in Phase2.c. **Do not modify** Phase2.h.

```
/* 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. And the branch_label turned to empty
string.
 *
 * The instruction should not be changed if it is not a branch instruction
 *
 * unresolved: input program as a list, whose branch instructions don't have resolved
immediate
 * first_pc: address where the first instruction of the program will eventually be placed in
memory
 * resolved: an empty list; after the function returns, resolved contains the resulting
instructions
 */
void resolve_addresses(struct ArrayList *unresolved, uint32_t first_pc, struct ArrayList
*resolved);
```

Try inventing an algorithm on paper first. You might use the example in 5.2.1. You'll quickly notice that some branches occur after the label they refer to and some occur before. Therefore, a single-pass algorithm (one that goes over the unresolved list only once) cannot solve this problem.

Therefore, we recommend you use a doubly-nested loop algorithm (easier, recommended) OR a two-pass algorithm with a data structure to keep track of the labels (harder but doable).

Examine the test cases in AssemblerTest.c for additional input/output examples.

6.2.1 How do I create a new branch instruction?

To create a branch instruction with a particular immediate and a branch_label that is the empty string, use the Bimm function from InstructionFactory.h. Alternatively you can modify the branch instruction you get from unresolved by changing its immediate and branch_label fields.

6.3 Phase 3 tasks

You will complete the implementation of phase 3 by implementing the function `translate_instruction` in Phase3.c. **Do not modify** Phase3.h.

```
/* Translate each Instruction object into
 * a 32-bit number.
 *
 * complete: list of Instructions to translate
 * machineCode: an array that is at least as long as the number of elements in complete;
 *              after the function returns, contains the instructions in 32-bit binary
representation
 *
 */
void translate_instructions(struct ArrayList *complete, uint32_t machineCode[]);
```

This function produces an encoding of each kind of instruction. Refer to the textbook chapter 6.4 and Appendix B for information about instruction encoding.

The instructions we are supporting (those in Instruction.h's enum ID) span these formats

- Data-processing instructions with a register operand
- Data-processing instructions with a constant operand
- LSL instruction with a constant shift amount (Data-processing instruction)
- CMP instruction with two register operands (Data-processing instruction)
- Branch instruction

The only cond codes you need to be concerned with are those in the enum COND in Instruction.h.

Make sure to set unused fields to 0. This default value is *not* required by the ARM architecture, but it is required by the provided Miniaa test code. You'll also notice that ARMSim# chooses to use 0 for unused fields.

Examine the test cases in AssemblerTest.c for additional input/output examples. You should also use ARMSim# to help you check translations you are unsure of.

6.3.1 How do I generate the 32-bit number I need?

Use the instruction formats found in Inquiry: Stored Programs, DDCA Ch 6.4, and DDCA Appendix B. Use the bitwise operations you learned about in the previous lab assignment to build a 32-bit number with bits in the right place.

6.3.2 How do I compute the rot and imm8?

You may assume that all constants that make it to Phase 3 **are valid and able to encoded**. What is your on-paper algorithm for determining rot and imm8? Make use of the skills you learned about in the bitwise operations lab assignment.

It might also be helpful to use a similar algorithm to what you do in Phase 1 to check the constant, except this time you need to use it to determine a rot and imm8. Remember, rotate.h has a function that may be helpful (rotate *left*).

7 General implementation help

- .h files are *header files*, which **declare** symbols (functions, structs, enums) that are **defined** elsewhere in a .c file. The idea of putting declarations in an .h file and definitions in a .c file is roughly analogous to putting declarations in an interface and definitions in a class in Java.
- If you need a function or struct declared in a .h file, then put the following at the top of your .c file.

#include “filename.h”

For example, if your code needed to refer to struct Instruction, then put
#include “Instruction.h”

- We’ve structured the skeleton code so that you **only need to modify…**
  - Phase1.c
  - Phase2.c
  - Phase3.c
  - AssemblerTest.c (optional but recommended to add tests)

8 Running and testing your code

The three phases are run on several test inputs by …

```
make
./AssemblerTest
```

The provided test cases separately test the three Phases. Therefore, you'll have good evidence for a correct Phase when all tests named by that Phase # are passing.

You can add your own tests to AssemblerTest.c. Use an existing test as a template. If you don't want to manually calculate what `phase3_expected` should be for your test, you can let ARMSim# do it for you: load your input program in ARMSim# and look at the machine code next to the source code.

You should add additional test cases to AssemblerTest.c. Find corner cases the tests do not cover:

- Other combinations of add/addi/sub/subi/or/ori with different constants (for Phase 1, Phase 3)
- Other combinations of label and branch positions (for Phase 2)

You are responsible for testing your assembler beyond the given tests. We will use additional tests during grading.

9 Testing tips

When you run the tests, you'll get a message like.

```
AssemblerTest.c:38:F:Core:test1Phase2:0: Assertion 'size(expectedP2) ==
size(resolved_tals)' failed
```

This says that test1Phase2 made an assertion that was false. Note that these assertions are actually in the functions testHelperPhase1,2,3, and all test results are printed at the end of the program.

Note that we've provided functions printInstructions and printMachineCodes, which are called within the testHelper functions. They print more information about cases that fail. Of course if multiple tests are failing, it will be harder to tell where the instruction came from. Feel free to modify the testing code to give you more information. When grading, we'll use our own copy of AssemblerTest.c

Alternatively, you could use gdb to look at the variable values. Since we are using the unit testing framework check.h, we need to run GDB slightly differently, adding CK_FORK=no before the command.

```
CK_FORK=no gdb ./AssemblerTest
Reading symbols from ./AssemblerTest...
```

If you want to isolate one *specific* test, one method is you can set a breakpoint at the test desired (need _fn at the end of it because of how the unit testing names functions).

```
(gdb) b test1Phase2_fn
Breakpoint 1 at 0x402d85: file AssemblerTest.c, line 119.
```

Then run the program. It will stop at the breakpoint. Now you can set a breakpoint at the test helper. And continue running up to that point.

```
(gdb) run
Breakpoint 1, test1Phase2_fn at AssemblerTest.c:119
(gdb) b phase2TestHelper
Breakpoint 2 at 0x402837: file AssemblerTest.c, line 51.
(gdb) c
Continuing.
Breakpoint 2, testHelperPhase2 (expectedP1=0x40c370, expectedP2=0x40c560)
```

```
    at AssemblerTest.c:51
```

Now you can display any desired variables that are in scope. For example,

```
(gdb) display size(expectedP1)
2: size(expectedP1) = 7
(gdb) display size(expectedP2)
3: size(expectedP2) = 0
```

10 What to submit

For credit your Miniaa implementation *must* at least compile and be runnable. **You should not depend on modifications to** files other than those you are submitting:

Required:

- Phase1.c (method implemented)
- Phase2.c (method implemented)
- Phase3.c (method implemented)

Both partners are responsible for **double-double-checking** that 3 files submitted to ICON are the version of the files that you intended.

11 Good job!

Once you have completed this project, if you added support for the rest of the ARM instructions and the dot (.) directives you could replace the assembler of ARMSim#.

The only major piece we've excluded is the ***parser***, which turns text files of ARM code into our internal representation of instructions: a list of Instruction objects.