COMS12200 Labs

Week 16: Writing an assembler for the simple machine

Simon Hollis (simon@cs.bris.ac.uk) Document Version 1.2

Build-a-comp module lab format

This worksheet is intended to be attempted in the corresponding lab session, where there will be help available and we can give feedback on your work.

- The work represented by the lab worksheets will accumulate to form a portfolio: whether complete or not, archive everything you do via https://wwwa.fen.bris.ac.uk/COMS12200/ since it will partly form the basis for assessment.
- You are not necessarily expected to finish all work within the lab. itself. However, the emphasis is on you to catch up with anything left unfinished.
- In build-a-comp labs, you will **work in pairs** to complete the worksheets for the first part of the course. In later labs, with more complex assignments, you will work in bigger groups.
- To accommodate the number of students registered on the unit, the 3 hour lab session is split into two 1.5 hour halves.
- **You should only attend one session**, 9am-10:30am OR 10:30am-12pm. **The slot to which you have been allocated is visible on your SAFE progress page for COMS12200.**

Lab overview: Writing an assembler for our machine

Our little machine is now finished. Rather than changing it any more, we will instead concentrate on writing an assembler for it. As a reminder, an *assembler* is a program that takes the description of a program in *assembly language* and converts it to its machine-code equivalent.

Why write an assembler?

Our little machine has only a few short instructions, but already writing a program for it is a hassle. One of the big problems is that human brains think in words and concepts and most computing machines use binary as an input.

So, when we want to think "Increment r0", we actually have to write "00000000". Our brains are just not set up to do this efficiently.

An assembler allows us to write our programs using mnemonics and higher-level input, whilst automating the mapping to machine code. With a finished assembler, writing and running programs will be much faster ☺

As a final incentive, in future you will be able to automatically download assembled programs to the machines.

Today's task

Using your favourite programming language (e.g. C, Java, Python…), write an assembler for the little machine's ISA. Here is a reminder of the ISA:

Instruction	Op-code	Meaning
INC r	000 r_i 0000	r_i <- r_i + 1
DEC r	001 r_i 0000	r_i <- r_i - 1
JNZ [address]	0100 $a_3a_2a_1a_0$	if (r_0 != 0) then PC <- address ; else PC <- PC + 1
JNEG [address]	0110 a3a2a1a0	if (r0 < 0) then PC <- address ; else PC <- PC + 1
STR r, [address]	100 r_i $a_3a_2a_1a_0$	MEM[address] <- r_i
LDR r, [address]	101 r_i $a_3a_2a_1a_0$	r_i <- MEM[address]

Assembler Overview

An assembler performs the following operations:

1. Parsing the input file
2. Resolving labels
3. Transforming instructions to machine-code
4. Writing an output file

The definitions of the stage requirements now follow:

Assembler Input

- The input to the assembler will be assembly language, using the given instructions and their mnemonics.
- One instruction will be entered per line
- Whitespace is ignored and indentation is not important
- Comments may exist in the input and should be ignored
- Comments can only exist on a line with no instructions
- Comments are prefixed by a semi-colon (';')
- Label declarations appear on a separate line and are prefixed by a colon (':')
- Labels point to the following instruction
- Instructions using labels also prefix the label with a colon

e.g. an input line may be:

```
; this is a comment. The next line is a label

:foo

INC r1

JNZ :foo

; the line above used a label

JNZ 2

; the above line points to itself and is an infinite loop
```

In the above example, the label "foo" points to the "INC r1" instruction. The JNZ destination will therefore point to the same instruction and address 0 if the above code is at the start of a program.

Assembler Output

The output from the assembler will be a hex file, in the following format: *<instruction><newline>.*

e.g. for the above program:

```
10

40

42
```

You may find it helpful for debugging purposes to have an assembler option that also outputs the addresses e.g.

```
0: 10

1: 40

2: 42
```

Two-pass parsing for label resolution

You should already have the programming skills to perform steps 1,3 & 4 of the assembler, but label resolution requires a little more explanation.

Your input file may have labels declared and used in any order. Thus, it is prudent to use a two-pass parsing strategy, *as outlined in lectures*.

1. Create an empty table with two columns, textual label and absolute address.
2. Open and parse your input file for labels only (i.e. detect only lines with labels on. Keep track of the instruction addresses that they point to.
3. When a label is encountered, add to your table.
4. Once the file has been parsed, re-open and parse a second time. This time, ignore any label declarations. Map instructions without address labels to output machine code directly. If any have address labels, look these up in your table and translate to absolute addresses.

e.g. after the first pass on the above example, your table will look like:

Label	Address
"foo"	0

Loading your file into ModuleSim

Now you have a hex file, you can automatically load it into Modulesim to initialise your RAM there!

1. Add a RAM to your design
2. Click "Simulation->Load Hex File"
3. Choose your hex file and load it in.

The RAM will now be initialised to your hex values ☺

OK! You now have all the knowledge you need to create your assembler. Go forth and assemble!