Introduction to Assemblers and assembly language

With specifics of Thumb V1 Assembly

Simon Hollis

COMS12200

What is assembly language

- Assembly language is a close-to-the-hardware programming language.
- There is normally a separate one for each target processor architecture.
- It consists of mnemonics and explicit argument and location information.
- Generally, a single assembly instruction relates to a single native instruction on an architecture.

What is assembly language?

- It is also the input to a program called an assembler.
- Assemblers take assembly language as an input and produce architecture-specific machine code.
- Machine code is in binary format and is directly runnable on the hardware.

Why is it useful?

- Assembly language is very useful in certain scenarios:
  - Understanding the hardware and what is going on.
  - Understanding the timing and requirements of code fragments.
  - Directing explicit resource allocations.
  - Performing optimisation of small code sections.
  - Interacting with normally hidden architectural aspects.

Why not use it all the time?

Here’s just four good reasons:

- It’s too low level for general program writing.
- Consider the power of a single assembly line vs that of e.g. C
- A human shouldn’t have to insert labels, addresses and immediates by hand.
- Readability and maintainability is very bad in large assembly fragments.

What does assembler look like?

- You’ve seen example assembler syntax already:
  - `ADD r0, r1, r2`
  - `ADD r0, #1`
- These are some examples of register machine assembler (specifically ARM Thumb syntax).

Labels

Labels are the way of naming code locations in assembler. Here's how they work in ARM syntax:

```
Loop                    Non-indented labels
        ADD r0, #1
        SUB r1, #1
        BNE Loop        Label use as Branch destination
Infinite                Non-indented labels
        B Infinite      Label use as Branch destination
```

Labels

- How does that work?
- After all, we can’t just pass text to the processor – it needs concrete addresses.
- The secret is in the assembler program.
- The assembler translates the labels into real addresses.
  - It does this though a *two-pass* methodology.

Assembler usage

AN INTRODUCTION TO THUMB ASSEMBLER

Thumb assembler format

- In this section, we will see specifics relating to the Thumb assembler language.
  - *(this will come in rather useful for your assignments too, so don't be shy with questions!)* ☺

General format

Thumb assembler instructions take:

1. An instruction.
2. Zero, one, two or three other arguments.
   - Arguments may be registers, immediates or addresses.

Thumb format terminology

When discussing assembler instructions, we use some mnemonics:

- Rm – 1st source argument
- Rn – 2nd source argument
- Rd – result destination
- Rt – result destination
- Rdn – result destination and 1st source
- ‘i’ / immediate – constant / address

Examples

Let's take the `ADD` instruction as an example.

The single instruction can take multiple formats:

1. Add with immediate (implicit 1st reg):
   - `ADD rmn, i8`
   - `ADD r0, #3`
2. Add registers
   - `ADD rd, rm, rn`
   - `ADD r0, r1, r2`

Examples

Let's take the `ADD` instruction as an example.

The single instruction can take multiple formats:

3. Add to stack pointer and update it
   - `ADD sp, i8`
   - `ADD sp, #8`
4. Add sp/pc and immediate, place in register
   - `ADD rd, pc, i8`
   - `ADD r0, pc, #12`

Zoom in on arguments

Various instruction arguments may be specified:

- Immediate, preceded by ‘#’
- Register, by name (e.g. ‘r0’, ‘lr’)
- Shifts (e.g. `LSL #2` to shift left two)
- Addresses, surrounded by square brackets
  - E.g. `[r1]` means treat the contents of `r1` as an address
  - `[44]` means access address `44`.

Zoom in on arguments

The length of an argument differs from instruction to instruction

- Registers: either implicit or 3 or 4 bits
- Immediates: from 5—23 bits

Memory instructions

Thumb V1 has two basic, plus some specialised memory access instructions

1. Load:
   - `LDR <dest reg>, [<address>]`
   - E.g. `LDR r0, [r1]` ; load contents of memory at r1's address into r0
   - `LDR r0, [44]` ; load contents of memory[44] into r0

Memory instructions

Thumb V1 has two basic, plus some specialised memory access instructions

2. Store

- STR <source reg>, [<address>]
- E.g. STR r0, [r1] ; store contents of r0 in r1’s address
- STR r0, [44] ; store contents of r0 in address 44

PUSH and POP

- We saw `PUSH` and `POP` last time
  - Enable efficient stack-based access
  - Are dedicated instructions with implicit destination/source in the Thumb V1
- They can take multiple registers as arguments
  - E.g. `PUSH {r0, r1, r2}` ; push 3 regs
  - `PUSH {r0-r2}` ; same
  - `PUSH {r0-r2, r4}` ; push 4 regs

Assembler translation

When running over an input assembly file, a key feature of an assembler is to map the input to the 'best' output instruction.

i.e. select the closest match or most efficient.

Sometimes, the argument length is considered.

Assembler translation

Take the ADD instruction we just saw.

- There are multiple options for its formatting.
- In the ISA, each will be represented by a *different* ADD instruction.
- In this course, we will specify the different op-codes with unique names
- However, the input assembly code only specifies ‘ADD’. Format is extracted from arguments.

Assembler translation

Translation example for `ADD` variants.

Assembler input	Selected format	16 bit instruction in binary
ADD rmn, #i	ADDI rmn, #i8	<u>00110</u>rdnrdnrdni8i8i8i8i8i8i8i8
ADD rd, rm, rn	ADDR rd, rm, rn	<u>0001100</u>rmrmrmrnrnrnrdrdrd
ADD sp, #i	ADDISP #i7	<u>101100000</u>i7i7i7i7i7i7i7
ADD rd, pc, #i	ADDPCI rd, #i8	<u>10100</u>rdrdrdi8i8i8i8i8i8i8i8

Op-codes are <u>underlined</u>

TWO-PASS ASSEMBLING

Overview of an assembler

- I will give examples about how an assembling program
  - Parses input
  - Matches mnemonics
  - Creates internal data structures
  - Produces machine code output

Translation examples

Using the assembler ‘aasm’ installed at:

/home/staff/simon/COMS12200/aasm/aasm

We will see some assembly examples.

Summary

- An ‘assembler’ is a program that translates *assembly language* into machine code.
- Machine code has its own architecture-specific formatting and many assembly languages mirror this.
- Assembly is not always as one-to-one mapping.
- Multiple output formats may exist.