# COMP15111 Notes

Chris Williamson, Todd Davies

October 21, 2013

## Contents

# 1 Lecture 1: Introduction

## 1.1 A Computational Model

The simplest, earliest, commonest, most important computational model is the **Von-Neumann Imperative Procedural Computer Model**

According to this model, a computer can:

1. Store information
2. Manipulate the stored information
3. Make decisions depending on the stored information

## 1.2 Simple View Of A Computer

$$Memory \Leftrightarrow Bus \Leftrightarrow Processor$$

### 1.2.1 Memory

Memory is a set of locations which can hold information, such as numbers(or programs). Each memory location has a unique (numerical) address, and there are typically thousands of millions of different locations. There are various ways of depicting memory; a common one is a 'hex dump' that often looks something like this:

| Address | Values (8 bit numbers) | Characters |
|---|---|---|
| 00000000 | 48 65 6c 6c 6f 0a | Hello. |

Run the command *hexdump* to generate hexdumps.

Each item that is in the memory has a unique address.

### 1.2.2 Bus

A bus is a bidirectional communication path. It is able to transmit addresses and numbers between components inside the computer.

### 1.2.3 Processor

The processor obeys a sequence of instructions, commonly referred to as a program. Historically the processor was often referred to as a CPU, however, this is inappropriate nowadays since typical processors consist of several processing cores.

## 1.3 Three-address instructions

Every kind of processor has a different set of instructions, real world examples include: Pentium, ARM and others

Each three-address instruction:

1. Copies the values from any two memory locations and sends them to the processor (source operands)
2. Copies some operation e.g. adds the copied numbers together
3. Copies the result back from the processor into a third memory location (destination operand)

For example, if we wanted to convert the Java code $sum = a + b$; into a three-address instruction we would:

1. Identify the two *source operands*: $a$ holds 2, $b$ holds 3
2. Perform the *operation*: $2 + 3 = 5$
3. Let the variable *sum* equal the answer 5. This is the *destination operand*

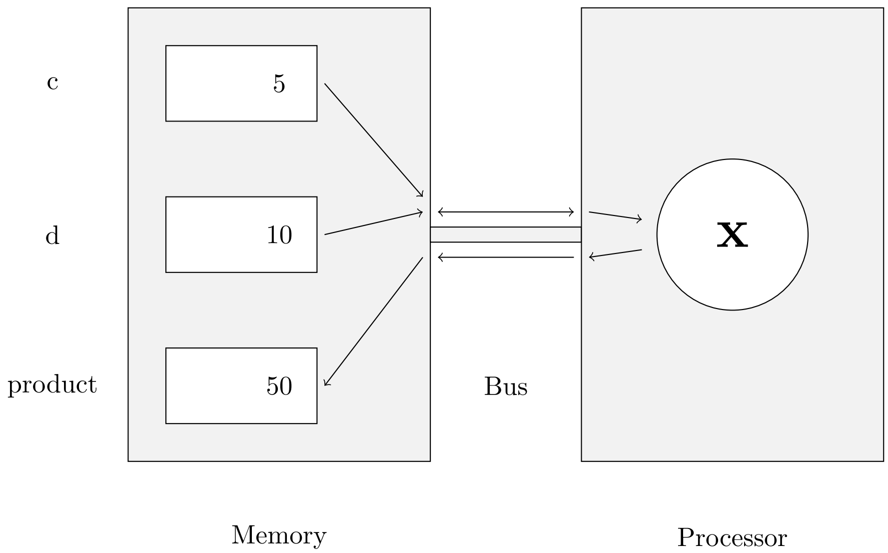
### 1.3.1 Three address example

**Question:** Convert the Java code *product = c * d;* into the three-address style and draw a two box view of it.

First we need to re-write the Java code in the three-address style:

$$product \leftarrow c * d$$

Now we can draw the box view of it:




### 1.3.2 Memory bottleneck

Most processors can process instructions faster than they can be fed by memory. Each instruction in the three-address cycle requires four memory cycles:

1. Fetch the instruction
2. Read the first operand
3. Read the second operand
4. Write the result to memory

Each of these memory cycles could take hundreds of processor clock cycles to complete, and so in this time the processor would be doing nothing. However, most modern processors employ a *cache* to temporarily store commonly accessed memory locations, and so avoid some of the memory cycles.

## 1.4 Registers

Registers are very small amounts of storage build into a processor. Since they are inside the processor data doesn't need to be transferred over the bus, and so they are very fast. Registers are used instead of the main memory which speeds up program execution.

Each register can only hold one value and each processor will only generally have a few dozen registers (e.g. ARM has sixteen).

### 1.5 Instruction Styles

#### 1.5.1 One address

The one address style can only use up to one memory location in each instruction, all other operands must be registers. An example may be:

$$R1 \leftarrow R0 + memory\ location$$

#### 1.5.2 Load-store

The load-store style cannot perform operations on memory locations at all. Instead, values from memory must be loaded into a registers before the operation takes place and then the operation can be performed on the registers. Following the operation, the result must be stored back into memory again.

$$R1 \leftarrow memory\ location R1 \leftarrow R0 + R1 memory\ location \leftarrow R1$$

This means that we need extra instructions to do stuff with memory locations:

1. **Load** the value from memory into a register before the operation.
2. **Store** the value in the register back to memory after the operation.

For example, the Java code *Sum = a + b + c;* would be run as:

| | | | |
|---|---|---|---|
| R1 | $\leftarrow$ | a | (i.e. load from a) |
| R2 | $\leftarrow$ | b | (i.e. load from b) |
| R3 | $\leftarrow$ | R1 + R2 | (i.e. a+b) |
| R4 | $\leftarrow$ | c | (i.e. load from c) |
| R5 | $\leftarrow$ | R3 + R4 | (i.e. (a+b)+c) |
| Sum | $\rightarrow$ | R5 | (i.e. store to sum) |

You can see that the load-store style favours lots of very simple, very fast instructions.

# 2 Lecture 2

Computers obey programs which are sequences of instructions. Instructions are coded as values in memory. The sequences are held in memory adjacent memory locations. Values in memory can be interpreted as:

- Numbers (in several different ways)
- Instructions

- Text
- Colours
- Music
- Anything you want

Values are often represented as numbers for convenience.

## 2.1 Assembly Language

Assembly language is a means of representing machine instructions in a human readable form.
Each type of processor has its own assembly language but they typically have a lot in common:

- A mnemonic specifies the type of operation
- A destination a register on this case
- And one or more sources also registers
- Possibly with a comment too

## 2.2 ARM instructions

ARM has many instructions but we only need three categories:

- Memory operations
- Processing operations
- Control flow

Memory operation move data between the memory and the registers. Processing operations perform calculations using value already in registers. Control flow instructions are used to make decisions, repeat operations etc.

## 2.3 ARM memory instructions

Memory operations load a register from the memory or store a register value to the memory. e.g. LDR R1, a means: R1 $\leftarrow$ a
e.g. STR R5, sum means: R5 $\rightarrow$ sum (i.e. sum $\leftarrow$ R5)

*a* and *sum* are aliases for the addresses of memory locations.

## 2.4 ARM processing instructions

Processing operations such as addition, subtraction, multiplication.

e.g. ADD R3, R1, R2 means: R3 $\leftarrow$ R1 + R2

## 2.5 ARM control instructions

Fundamentally, these are branches to other code sequences. Often, branches are made conditional to allow decisions to be made.

e.g. B somewhere means: branch to somewhere
e.g. BEQ elsewhere means: branch to elsewhere IF previous result was equals
e.g. BNE wherever means: branch to wherever IF previous result was not equal

## 2.6 Stored programs and the Program Counter

A computer can make decisions, and choose which instructions to obey next depending upon the results of those decisions. How? First we need to see how the sequence of instructions is controlled. Von-Neumann Model: memory holds both instructions and numbers - **stored program Program Counter** (PC) register: holds the address of the memory location containing the next instruction to b obeyed (executed).

ARM uses register 15 as its PC

## 2.7 Fetch-Execute Cycle

Start with PC containing the address of (the memory location holding) the first instruction of a program.

Repeatedly:

1. **Fetch**: copy the instruction, pointed to by the PC, from memory and set PC to point to the next instruction
2. **Execute**: obey the instruction (exactly as before)

ARM:

1. 'Resets' to (starts at) address 00000000
2. Instructions each occupy 4 memory locations, so PC increases by 4 in each fetch

## 2.8 Decision Making

Linear sequences of instructions are limiting. To make a decision, the computer must change (or not) to a different sequence of instructions.

e.g. a 1 pound discount on items worth 20 pounds or more. Decision: compare the total and 20 pounds to see if it is larger, then depending on result, either perform action or not. Action: subtract 1 pound from the total.

Computers have no intelligence, so spell out details. **Formalise**: if total $\geq 20$ pounds then subtract 1 pound from total
**Rewrite**: if total $< 20$ pounds then dont
**subtract** 1 pound from total
Encode: as ARM instructions

## 2.9 Compare And Branch