From last time

In the context of a typical Von-Neumann computer, explain the purpose of: the Memory; the processor; and the Bus between them. (3 marks)

What are the similarities and the differences between computer memory (RAM) and processor registers? (3 marks)

What are the similarities and the differences between the “3-address” and the “Load-Store” instruction styles? (3 marks)
COMP15111: Introduction to Architecture

Lecture 2: ARM

Dr. Javier Navaridas

School of Computer Science, University of Manchester

Autumn 2016
Overview & Learning Outcomes

Looking at a real processor: ARM

Looking at (tiny part of) a “real” application

How the computer runs the application

Where the application is, in the computer

How the computer “knows” which instruction to obey next

How a computer can “make a decision”

Some ARM instructions
load-store style

\[ \text{sum} = a + b + c; \]

becomes:

\begin{align*}
\text{register1 } &\leftarrow a & (\text{i.e. load from } a) \\
\text{register2 } &\leftarrow b & (\text{i.e. load from } b) \\
\text{register3 } &\leftarrow \text{register1} + \text{register2} & (\text{i.e. } a+b) \\
\text{register4 } &\leftarrow c & (\text{i.e. load from } c) \\
\text{register5 } &\leftarrow \text{register3} + \text{register4} & (\text{i.e. } (a+b)+c) \\
\text{sum } &\leftarrow \text{register5} & (\text{i.e. store to sum})
\end{align*}

One load or store instruction per variable (a, b, c, sum)
One arithmetic instruction per operation (+, +)

Lots of very simple, very fast instructions – no wasted effort!
load-store style

Registers carry values from memory, to (and between) the operations, and then back to memory
How computers work – an overview

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.
An example program

```
3852406804 E59F1014 ldr r1, a
3852410900 E59F2014 ldr r2, b
3766562818 E0813002 add r3, r1, r2
3852419088 E59F4010 ldr r4, c
3766702084 E0835004 add r5, r3, r4
3851374604 E58F500C str r5, sum
4009754626 EF000002 svc 2
```

“object code” – moderately difficult for a human to interpret!

May be easier in a different numerical base (e.g. base 16)

Definitely easier in mnemonic form – “source code”
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:

```
add r3, r1, r2 ; assembly code example
```

- A mnemonic – specifies the type of operation
- A destination – a register in this case
- ... and one or more sources – also registers
- Possibly with a comment too

... although other classes of instructions exist (more, later)
For most of this course-unit we will be looking at a typical load-store **ISA** (Instruction Set Architecture), the ARM.

“Acorn RISC Machine” (1983-6)
“Advanced RISC Machine” (1990)

RISC = “Reduced Instruction-Set Computer” (late 1970s)

2010 figures → 6.1 billion sold → ∼200/s
Mostly *embedded* or in portable devices such as smartphones (95% of smartphones, market)
ARM instructions

ARM has many instructions but we only need three categories:

- Memory operations
- Processing operations
- Control flow

Memory operations move data between the memory and the registers

Processing operations perform calculations using values already in registers

Control flow instructions are used to make decisions, repeat operations etc.
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

(ARM has some other memory transfer instructions, some of which will be met later.)
ARM processing instructions

Processing operations, such as **ADD**ition, **SUB**traction, **MUL**tiplication ...

- **ADD** R3, R1, R2  means: R3 ← R1 + R2

- **SUB** R3, R3, R2

- **MUL** R3, R3, R3

- **MOV** R3, R4

(ARM has some other data processing instructions, some of which will be met later.)
ARM control instructions

Fundamentally, these are Branches to other code sequences.

Often, branches are made conditional to allow decisions to be made.

e.g. \texttt{B somewhere} means: branch to ‘somewhere’

\begin{itemize}
\item e.g. \texttt{BEQ elsewhere} means: branch to ‘elsewhere’
  \hspace{1cm} IF ‘previous result’ was ‘equals’
\item e.g. \texttt{BNE wherever} ?
\end{itemize}

(ARM has a rich set of flow control instructions, some of which will be met later.)
Labels and Addresses

When an instruction says: \( \text{B somewhere} \)

The ‘somewhere’ part is a label which is an alias for the address of a memory location.

Instructions are stored in memory in the same way as data. You get to choose what’s in any location.

Hopefully, ‘somewhere’ is the start of another sequence of instructions – otherwise the result is called a “crash”!

There is the option of a label before any mnemonic:

\[ \text{B somewhere ; Jump from here ... ... somewhere ADD R0, R1, R2 ; ...to here} \]
sum = a + b + c;

LDR R1, a  (i.e. load R1 from a)
LDR R2, b  (i.e. load R2 from b)
ADD R3, R1, R2  (i.e. R3 ← a+b)
LDR R4, c  (i.e. load R4 from c)
ADD R5, R3, R4  (i.e. R5 ← (a+b)+c)
STR R5, sum  (i.e. store R5 to sum)

To stop the computer at the end of the instruction sequence:
SVC 2  (Treat as ‘magic’ for now!)
(depending on the System Software we use)
Question

What are the contents of the registers (R0, R1, R2) and variables (anne, tom, fred) after each of the following instructions is obeyed:

```
LDR R0, anne
STR R0, fred
LDR R1, tom
ADD R2, R0, R1
STR R2, anne
SVC 2
```

The variables start with these values:
anne is 23
tom is 45
fred is 10
Answer

R0  R1  R2  anne  tom  fred
?    ?    ?    23    45    10

LDR R0, anne

STR R0, fred

LDR R1, tom

ADD R2, R0, R1

STR R2, anne

SVC 2
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 be obeyed (executed).

ARM uses register 15 as its PC
Fetch-Execute Cycle

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

Repeatedly:

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

**ARM**:

- ‘Resets’ to (starts at) address 00000000
- Instructions each occupy 4 memory locations, so PC increases by 4 in each fetch
Question

Assuming the first instruction is at address 0, what is the address of each instruction, and what will be the contents of the PC register after each instruction is obeyed?

```
0:  LDR R0, anne
    STR R0, fred
    LDR R1, tom
    ADD R2, R0, R1
    STR R2, anne
    SVC 2
```
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 don’t subtract 1 pound from total.

**Encode**: as ARM instructions.
Compare and Branch

Decision takes two steps:

- **compare** the two values – \texttt{CMP}
- use result of comparison in a **conditional** instruction – e.g. \texttt{BLT} (a **branch**)

\texttt{CMP}: compares its two operands
Processor records if \((\text{Op1} - \text{Op2})\) is positive, negative or zero.

Branch instructions (\texttt{B, BLT} etc.) change PC
Operand is address of the start of the new sequence of instructions – a **label** (e.g. \texttt{BLT nodiscount})

**unconditional branch** always changes PC: \texttt{B} \ldots
**conditional branch** maybe changes PC, depending on previous comparison: e.g. \texttt{BLT} \ldots \equiv \text{Branch if Less Than}
Example conditions

These are the commonest conditions encountered:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Execute if previous comparison . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>EQual</td>
</tr>
<tr>
<td>NE</td>
<td>Not Equal</td>
</tr>
<tr>
<td>GE</td>
<td>signed Greater than or Equal</td>
</tr>
<tr>
<td>LT</td>
<td>signed Less Than</td>
</tr>
<tr>
<td>GT</td>
<td>signed Greater Than</td>
</tr>
<tr>
<td>LE</td>
<td>signed Less than or Equal</td>
</tr>
<tr>
<td>AL or blank</td>
<td>ALways execute</td>
</tr>
</tbody>
</table>

e.g. BEQ, BNE, BGE, BLT, BGT, BLE, B (or BAL)

ARM supports all these and others (described later)
Decision example

```
LDR R0, total ; add next to total
LDR R1, next
ADD R2, R0, R1
STR R2, total

LDR R3, minimum ; if total<20 pounds
CMP R2, R3
BLT nodiscount ; then don't discount

LDR R4, discount ; deduct 1 pound from total
SUB R5, R2, R4
STR R5, total

nodiscount ; whichever happened
SVC 2 ; stop

total DEFW 1534 ; data DEFinitions
next DEFW 105 ; (start values)
minimum DEFW 2000 ; Values in pence
discount DEFW 100
```
We need to allocate memory locations for “total” and “next”.

We can use DEFW (DEFine Word) – a *pseudo operation*

- puts a value into memory *before* the program runs (e.g. the starting values for “total” and “next”)
- looks like an instruction but it isn’t meant for execution
- the assembler uses the next available place
- the label lets us *find* the allocated location

We can also put constants “100” and “2000” into memory locations (“discount” and “minimum”)

**Don’t execute them!**

The processor won’t recognise what you ‘meant’
It will interpret these as instructions . . . doing ?????
### Addresses: name → number

<table>
<thead>
<tr>
<th>address</th>
<th>content (as mnemonics)</th>
<th>content (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>@0</td>
<td>LDR R0, total</td>
<td>@44</td>
</tr>
<tr>
<td>@4</td>
<td>LDR R1, next</td>
<td>@48</td>
</tr>
<tr>
<td>@8</td>
<td>ADD R2, R0, R1</td>
<td>@52</td>
</tr>
<tr>
<td>@12</td>
<td>STR R2, total</td>
<td>@56</td>
</tr>
<tr>
<td>@16</td>
<td>LDR R3, minimum</td>
<td>@64</td>
</tr>
<tr>
<td>@20</td>
<td>CMP R2, R3</td>
<td>@68</td>
</tr>
<tr>
<td>@24</td>
<td>BLT nodiscount</td>
<td>@70</td>
</tr>
<tr>
<td>@28</td>
<td>LDR R4, discount</td>
<td>total</td>
</tr>
<tr>
<td>@32</td>
<td>SUB R5, R2, R4</td>
<td>DEFW 105</td>
</tr>
<tr>
<td>@36</td>
<td>STR R5, total</td>
<td>minimum</td>
</tr>
<tr>
<td>@40</td>
<td>nodiscount SVC 2</td>
<td>DEFW 1534</td>
</tr>
<tr>
<td>@44</td>
<td>total DEFW 1534</td>
<td></td>
</tr>
<tr>
<td>@48</td>
<td>next DEFW 105</td>
<td></td>
</tr>
<tr>
<td>@52</td>
<td>minimum DEFW 2000</td>
<td></td>
</tr>
<tr>
<td>@56</td>
<td>discount DEFW 100</td>
<td></td>
</tr>
</tbody>
</table>
The Memory Map

The memory contains values:
  - Some values you have defined
  - Many values are undefined (i.e. could be anything)

The *meaning* of the values is a matter of *interpretation*:
  - Some values are instructions
  - Some values are data
  - There are different ways of defining the values
    - e.g.  `ADD R1, R2, R3` used for instructions
    - e.g.  `DEFW 100` used for data
  - The computer only ‘knows’ what you tell it
    - If the PC points at it then it’s an instruction whether you meant it or not
Write ARM instructions to do the following:

If a is greater than b, then subtract b from a.

i.e. if $a > b$ then $a = a - b$;
Write ARM instructions to calculate \( a = (b \times c) - (b + c); \)

(hint: try to avoid loading b or c more than once)
Write ARM instructions to put zero into a variable e.g. \( a = 0; \)

(hint: you could use a memory location that contains zero, or try to think of an arithmetic instruction that always gives an answer of zero)
Summary of key points

A real processor: ARM (Load-Store ISA, 16 registers)

Some ARM instructions: LDR, STR, ADD, SUB, MUL, SVC

Assembly language as a representation

How the computer runs the application: “fetch-execute cycle”

Where the application is in the computer: “stored program”

How the computer “knows” which instruction is next: PC reg
– Unconditional Branch: B

How a computer can “decide”: Compare, Conditional branch
– CMP, BEQ, BGE, BGT, BLE, BLT, BNE etc.
– invert condition to avoid doing action
Your Questions
Glossary

ARM
RISC
ISA
Load (LDR) instruction
Store (STR) instruction
Stored program computer
Fetch-Execute cycle
Fetch
Execute
PC register
Compare (CMP) instruction
Unconditional Branch (B) instruction
Conditional Branch (BLT etc.) instruction
Label
Pseudo-instruction
DEFW
SVC instruction
For next time

Describe in detail what happens when the following ARM program is obeyed. At each step, clearly describe the movement of information (both numbers and instructions) between the memory (RAM) and the processor, and how the values in the memory and in registers R0, R1, R2 and R15 (PC) change. Assume that the program starts at memory location 0. (5 marks)

```
B    d
a DEFW 23
b DEFW 45
c DEFW 10
d LDR R0, a
 STR R0, c
 LDR R1, b
 ADD R2, R1, R0
 STR R2, a
 SVC 2; terminate program
```
Exam Questions

**What** is a PC register used for in an ARM computer, and **why** is it used? (2 marks)

What are the similarities, and the differences, between:

- ARM instructions `LDR R1, a` **and** `STR R1, a`
- ARM instructions `ADD R1, R2, R3` **and** `SUB R1, R2, R3`
- ARM instructions `CMP R2, R3` **and** `SUB R1, R2, R3`
- ARM instructions `B next` **and** `BLT next`

(8 marks)