From last time

How is a bit used to represent a simple decision, like the answer to “Is this question hard”? (1 mark)

Without using a calculator, and briefly explaining how you do it, convert the decimal number 97 to binary, and then from binary to octal and to hexadecimal. (3 marks)
COMP15111: Introduction to Architecture
Lecture 4: ARM assembly programming

Dr. Javier Navaridas

School of Computer Science, University of Manchester

Autumn 2016
Overview & Learning Outcomes

ARM design decisions: flexibility v. simplicitly
- how are numbers stored in memory?
- why does the ARM’s PC register change in steps of 4?
- how are ARM instructions stored in memory?

What is in an Assembly Language?
- Instructions
- Instructions that don’t really exist
- Not instructions at all
How many bits might we want to use?

Instruction set design trade-offs:
- include everything that might be useful?
- include as little as possible?

What sizes of bit-patterns will be easy to access?
- any size: 1-bit, 2-bit, 3-bit, … ?
- a few fixed sizes?

Basic kinds of values:
- characters = 8 bits (byte)
- integers, addresses = 16? 32? 64? bits (word)
- instructions = ?
What does ARM do?

A **byte** is 8 bits
– used for characters (becoming outdated?)

A **word** is 4 bytes (32 bits)
– used for integers, addresses, instructions

A **halfword** is 2 bytes (16 bits)
– specialist use: ignored in this module

A **doubleword** is 8 bytes (64 bits)
– also ignored in this module

**All addresses are in byte units**

Words must be *aligned*
i.e. a word address must be a multiple of 4
LDR and STR instructions

Word (32-bit):

\texttt{STR} – All 32 bits of register copied to memory
\texttt{LDR} – 32 bits of memory copied to register

Byte (8-bit):

\texttt{STRB} – bottom 8 bits of register copied to memory
\texttt{LDRB} – 8 bits of memory copied to bottom of register; upper 24 bits of register zeroed

(There are some other loads and stores but they are not of particular interest here.)
Endianness

In what order are the bytes of a word? e.g. put 0x12345678 into word 0
What do we see in bytes 0, 1, 2 and 3?

The **least significant** byte of word 0 will contain 0x78
- is this byte 0? (**little-endian**)
- is this byte 3? (**big-endian**)

ARM computers can be configured to use either
– we use little-endian:

<table>
<thead>
<tr>
<th>byte 3</th>
<th>byte 2</th>
<th>byte 1</th>
<th>byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12</td>
<td>0x34</td>
<td>0x56</td>
<td>0x78</td>
</tr>
</tbody>
</table>

Least significant byte at smallest address.
Questions

<table>
<thead>
<tr>
<th>byte 23</th>
<th>byte 22</th>
<th>byte 21</th>
<th>byte 20</th>
<th>word 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x87</td>
<td>0x65</td>
<td>0x43</td>
<td>0x21</td>
<td></td>
</tr>
</tbody>
</table>

Q: what is in Register 0 after: `LDRB R0, 22`

Q: then what is in word 20 after: `STRB R0, 21`
Address range

How many different bytes and words can an ARM access?

An ARM address is 32 bits
All addresses are in byte units

so:
$2^{32}$ (≈ 4 billion) different bytes
$2^{32}/4$ (≈ 1 billion) different words

This is the address range of the architecture.

This does not mean you always have that much memory physically present!
Instruction encoding: example – branch

All ARM instructions occupy one 32-bit word.

This is one example:

<table>
<thead>
<tr>
<th>4</th>
<th>4</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>1010</td>
<td>offset</td>
</tr>
</tbody>
</table>

- First 4 bits: condition (e.g. 1011 = LT, 1110 = AL)
  $2^4$ different conditions (only 15 used – complete list later)
- Next 4 bits: operation = Branch
- Remaining 24 bits: where to branch to
  i.e. $2^{24}$ (16M) out of $2^{32}$ (4G) different addresses

Can’t jump anywhere (with this one instruction)
How do we get the best $2^{24}$ different locations?

– on ARM, an instruction is always a word, so an instruction address must be a multiple of 4, so actual address used = $4 \times (24\text{-bit number in instruction})$ (i.e. $2^{24}$ out of $2^{30}$ different instruction addresses)

– Branch destinations usually nearby
  • treat 24-bit number as offset
  • forward and backward branches, so offset is signed
  • i.e. address = PC + $4 \times (24\text{-bit offset in instruction})$

We can branch backwards or forwards for $\sim 2^{23}$ instructions.

ARM Quirk: The offset in the instruction is from two instructions ahead of the branch.
  • You shouldn’t really care
  • The assembler will work this out for you
Example BLT from Cash-till program

... 
LDR R3, minimum
CMP R2, R3
BLT nodiscount ← this instruction
LDR R4, discount
SUB R5, R2, R4 ← ARM actually offsets from here
STR R5, total

nodiscount SVC 2
total DEFW 1534
minimum DEFW 2000
discount DEFW 100

- Offset is +4 instructions (+16 bytes)
- ARM codes this as “+2”
Literal operands – “#”

‘Literal’ (or ‘immediate’) operands are values which are contained in the instructions.

Examples:

- MOV R2, #100 ; Put the value ‘100’ into R2
- SUB R5, R5, #1 ; Decrement R5 by 1

It is impossible to code all $2^{32}$ possible values and leave space for the rest of an instruction!

ARM dedicates 12 bits of some instructions for literal values in a useful but complicated way.

- MOV R2, #512 can be assembled
- MOV R2, #257 can’t!

Most of the time the values you want will be available so you don’t need to worry about the details.
Cash-till program

LDR R0, total
LDR R1, next
ADD R2, R0, R1
STR R2, total
LDR R3, minimum
CMP R2, R3
BLT nodiscount
LDR R4, discount
SUB R5, R2, R4
STR R5, total

Literals are fixed values

If the values are small enough, we can put them directly into instructions, rather than fetching them from memory.

What is “small enough”? Try it and see if it works!

nodiscount
SVC 2

total DEFW 1534
next DEFW 105
minimum DEFW 2000
discount DEFW 100
Literals – “#”

LDR R3, minimum
CMP R2, R3
...
minimum DEFW 2000
→
CMP R2, #2000

LDR R4, discount
SUB R5, R2, R4
...
discount DEFW 100
→
SUB R5, R2, #100

Q: what would SUB R5, R2, 100 mean?
Supervisor Calls (SVC)

Sometimes there is a need to communicate with the ‘operating system’ to implement ‘special’ functions such as Input and Output (I/O).

This is done using predefined SuperVisor Calls, e.g: \texttt{SVC 2}

The parameter can be used to define the function and, often, other values are passed in registers.

These actions depend on software – \textit{they are not a function of the ARM processor itself}.

These calls were formerly called ‘SoftWare Interrupt’ and the mnemonic \texttt{SWI} is still widely seen.

This means exactly the same thing.
Supervisor Calls in the lab.

The lab. simulates a very primitive operating system to provide some simple I/O.

For the purposes of the lab, we have defined:

- **SVC 0** = output a character;
- **SVC 1** = input a character;
- **SVC 2** = stop;
- **SVC 3** = output string;
- **SVC 4** = output integer in decimal

An ARM’s true execution mechanism of the SVC depends on more complications than we want in this module.

- You will not see the SVC code
- You can treat these as ‘magic’ for now
Pseudo-Instructions

A pseudo-instruction is an idealised instruction which doesn’t exist and may result in something more complex.

This may be multiple instructions and/or the use of extra store locations.

The assembler works out the actual (minimal) code etc.

Most of the time you don’t care!

You may see something ‘unexpected’ if you look at the disassembled instruction.
Negative literals

Technically, ARM doesn’t support negative literals.

The assembler will try and compensate for this, e.g.:

\[
\begin{align*}
\text{ADD} & \ R0, \ R1, \ #-1 \quad \Rightarrow \quad \text{SUB} \ R0, \ R1, \ #1 \\
\text{CMP} & \ R2, \ #-10 \quad \Rightarrow \quad \text{CMN} \ R2, \ #10 \\
\text{MOV} & \ R3, \ #-3 \quad \Rightarrow \quad \text{MVN} \ R3, \ #2
\end{align*}
\]

- \text{CMN} \text{ is ‘CoMpare Negative’}
- \text{MVN} \text{ is ‘MoVe Not’}

But you \text{don’t need to know the details}. 
LDR reg, =number

e.g. LDR R0, =123
means load R0 with the literal number “123”

If possible this becomes:
    MOV    R0, #123

If the number is ‘large’, this becomes:
    LDR    R0, const

    ... 

const DEFW 123

Watch out!
− “LDR” now has more than one meaning
− 1 pseudo-instruction → several words in memory
ADR(L) register, address

ADR puts an address in a register

ADR R1, val
...
val DEFW 1234

puts the address of the variable “val” in R1

Assembler is only allowed to use 1 word of memory (1 instruction)
e.g.  ADD R1, PC, #offset
... but is not able to encode any arbitrary address

ADRL is used in exactly the same way as ADR but works for any address

Assembler may use up to 4 instructions (memory words)!
Directives: DEF...

Directives are non-instructions which control the behaviour of an assembler.
Some of these ‘plant’ data in the memory, others do not.
DEFinitions allow data items to be defined

**DEFW** num – reserves a word of store and puts the initial value “num” in it.
It can be used for several words:
square_table  DEFW 0, 1, 4, 9, 16, 25
where the label is associated with the first (lowest) address.

**DEFB** ... – reserves byte(s) of store and puts the initial value(s) in. It can be useful for strings, e.g.
string       DEFB ``Hello'', 0

**DEFS** size, fill – reserves a block of store of “size” bytes all initialised to “fill” (fill can be omitted – values undefined)
Directives: ALIGN, ORIGIN, ENTRY, EQU

These directives control the assembly process.

ALIGN – leave any blank bytes needed so next item starts on a word boundary – useful after ‘DEFB’ for example

ORIGIN \textit{addr} – put the following code (or data) starting at address “\textit{addr}” (default 0)

ENTRY – this is the starting point of the program (initial PC – default 0)

\textit{label EQU expression} – allows you to define your own names for values (usually literals)
This can make code easier to read and maintain. E.g.
discount \textit{EQU} 100

\ldots

\texttt{SUB R5, R2, \#discount}
Question

```
ORIGIN 0x1000
block DEFS 16, 0xFF
one DEFW 999
two DEFW @10
ORIGIN 0x1100
DEFW 0
ENTRY
LDR R0, =$100001
ADRL R1, block
LDR R2, block
LDR R3, two
ADRL R4, one
SVC 2
```

At what address does this program start execution? When it stops, what values are in R0 to R4 (state whether decimal, hex etc.)
Answer
Summary of key points

ARM design decisions: flexibility v. simplicity
– how are numbers stored in memory?
– why does the ARM’s PC register change in steps of 4?
– how are ARM instructions stored in memory?

What is in an Assembly Language?
– Instructions
– Instructions that don’t really exist
– Not instructions at all
– ADR ADRL DEFB DEFS ALIGN ENTRY ORIGIN EQU

Some more ARM instructions: LDRB STRB CMN MVN
Your Questions
Glossary

Word
Addressing unit
Little-endian
Big-endian
Literal
Instruction field
Offset
Byte-offset
Word-offset
Pseudo-Instruction
ADR pseudo-instruction
Assembler Directive
DEFB directive
ALIGN directive
For next time

Explain how integer values are represented on the ARM. (3 marks)

Explain why Pseudo Instructions are used in ARM assembly language. (2 marks)

Explain why only certain literal values can be used in ARM instructions. (2 marks)
Exam Questions

Describe the ARM ADRL pseudo instruction & explain the circumstances in which it must generate more than one real instruction. (4 marks)

Explain the function of the ORIGIN and ENTRY directives in ARM assembly language. (2 marks)

The ARM representation of an arithmetic instruction is:
- 4 bits: Conditional-execution code
- 2 bits: 00
- 1 bit: Determines the form of the Source Operand
- 4 bits: Operation Code
- 1 bit: Set condition codes
- 4 bits: Destination register
- 4 bits: Source register
- 12 bits: Source operand

A particular SUB instruction is represented by 0xE04EC00F: What are the destination and the source registers?
What 4-bit Operation Code represents a SUB instruction?
Are the condition codes updated by this instruction? (4 marks)
Reading

http://homepages.cwi.nl/~robertl/mash/128vs32