From last time

What are the differences between these three ARM instructions: (3 marks) 

ADD R1, R2, R3  
ADDS R1, R2, R3 and ADDEQ R1, R2, R3

Explain how using “S” (as for ADDS above) can sometimes help to make a code sequence shorter and faster, giving an example. (3 marks)
From last time – ctd.

Explain how using conditional instructions can sometimes help to make a code sequence shorter and faster, giving an example. (3 marks)
COMP15111: Introduction to Architecture
Lecture 7: Addresses and Addressing

Dr. Javier Navaridas

School of Computer Science, University of Manchester

Autumn 2016
Overview & Learning Outcomes

Addressing Modes

Direct Addressing

Register Indirect Addressing

Base + Offset Addressing

Example – printing a String

ARM’s Indirect Addressing Modes
What are ‘Addressing Modes’?

When a processor references memory it needs to produce an address.

The address needs the same number of bits as the memory address – i.e. 32 in the case of ARM

There are numerous mechanisms for generating addresses which are usually referred to as “addressing modes”.

An exhaustive list is difficult to impossible to define
Not all processors support ‘all’ addressing modes

Here are some typical ones, referred to ARM.
Direct Addressing

Direct addressing is a mode where the address is simply contained within the instruction:

To do this requires an instruction longer than the address size.

Pentium can do this – its instruction length varies
ARM can’t – 32-bit address doesn’t fit within 32-bit instruction.
ARM ‘Direct Addressing’

So far, we assumed direct addresses in LDR/STR instructions (numbers or names translated to numbers by the assembler).

e.g. Java: \[\text{int } a, b, c; \ldots a = b + c;\]

⇒ ARM: \[
\begin{align*}
\text{LDR } R0, & \ b \\
\text{LDR } R1, & \ c \\
\text{ADD } R0, & \ R0, \ R1 \\
\text{STR } R0, & \ a
\end{align*}
\]

This \textit{looks like direct addressing} – the address, given as a number/label in the instruction encoding, is used directly to access memory.

On ARM it’s ‘faked’ by the assembler as a pseudo-instruction.
Problems with Direct Addressing

**Java**: useful for getting a single integer or character but not very flexible if we want several consecutive data. E.g. String – get each char in turn

**ARM**: both instructions and addresses are 32 bits, but the instruction also specifies operation, register etc., so it can’t contain every possible address.

Solution: allow a register to contain an address, use the address in the register to do loads and stores.

This is **Register Indirect Addressing**
Register Indirect Addressing

The address is held in a register

It takes only a few bits to select a register (4 bits in the case of ARM... R0-R15)

A register can (typically) hold an arbitrary address (32 bits in the case of ARM)
ARM Register Indirect Addressing

ARM has register indirect addressing

e.g. loading a register from a memory location:

```
LDR R0, b
```

could be done using register indirect addressing:

```
ADR R2, b ; move the address of b into R2
LDR R0, [R2] ; use address in R2 to fetch the value of b
```

This is still a bit limited – addresses are:

– range limited (within ADR ‘instruction’)
– fixed

but:

– `ADRL` pseudo-op allows larger range (at a price)
– having addressed a variable once it is often used again
– variables are usually ‘near’ each other
Address Arithmetic

We can operate on registers, so we can:
– store/load/move addresses
– do arithmetic to calculate addresses

Rather than use e.g. extra ADD instructions, we often use **Base + Offset Addressing**
– address addition done within the operand

We have actually been using this all along:
Base = PC register
Immediate Offset Addressing

The address is calculated from a register value and a literal.

The register specifier is just a few bits.

The offset can be ‘fairly small’.

With one register ‘pointer’ any of several variables in nearby addresses may be addressed.
ARM Immediate Offset Addressing

ARM allows offsets of 12 bits in \texttt{LDR/STR}

This may be added \textit{or} subtracted, for example:

\begin{verbatim}
LDR   R0, [R1, #8]
STR   R3, [R6, #-0x240]
LDR   R7, [R2, #short_constant]
\end{verbatim}

This provides a range of $\pm \sim 4$ Kbytes around a ‘base’ register

In practice, adequate for most purposes.
ARM Register Offset Addressing

ARM also allows offsets using a second register

This may be added or subtracted, for example:

\[
\begin{align*}
\text{LDR} & \quad R0, [R1, R2] \\
\text{STR} & \quad R3, [R6, -R4]
\end{align*}
\]

A few examples of this will occur later.
In particular, this is useful for access to arrays (something to anticipate later in the semester)
PC + Offset Addressing

```
start  LDR  R0, b
       LDR  R1, c
       ADD  R0, R0, R1
       STR  R0, a
       SVC  2

a     DEFW 0
b     DEFW 0
c     DEFW 0
```
each address = base (in PC) + small offset (in instruction)

e.g. LDR  R0, b
PC = address of instruction + 8 (i.e. ADD . . .)
    (the ‘+ 8’ is an ARM peculiarity)
so offset = 4 words = 16 bytes
so instruction is actually: LDR  R0, [PC, #16]
also LDR  R1, [PC, #16] and STR  R0, [PC, #0]
Question: Cash-till program

What are the offsets for these LDR and STR instructions?

```
LDR R3, minimum
CMP R2, R3
BLT nodiscount
LDR R4, discount
SUB R5, R2, R4
STR R5, total

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

```
STR R5, total ;
LDR R4, discount ;
LDR R3, minimum ;
```
Byte Offset from any Register

e.g.

```
ADRL R0, fred       ; R0 = address of fred
LDR  R1, [R0]       ; R1 = value of fred
LDR  R2, [R0,#-4]   ; R2 = value from (address of fred - 4)
ADD  R3, R1, R2
STR  R3, [R0,#4]    ; change value at (address of fred + 4)
```

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>fred-4</td>
<td>123</td>
<td>⇒ R2</td>
</tr>
<tr>
<td>R0 → fred</td>
<td>456</td>
<td>⇒ R1</td>
</tr>
<tr>
<td>fred+4</td>
<td>579</td>
<td>⇐ R3 ⇐ R1 + R2</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
</tbody>
</table>
e.g. Strings

Java:
String message = "Hello";
...
System.out.print (message);
→ (nearly)

ARM:
message DEFB "Hello", 0
  ALIGN ; Get onto next word boundary
  ...
  ADRL R0, message
  SVC 3

but suppose we wanted to output each character individually:
for (int i = 0; i < message.length(); i++)
  System.out.print (message.charAt(i));
Accessing characters: LDRB, STRB

Characters are **Bytes** in this example

```
ADRL R1, message
LDRB R0, [R1]       ; fetch 1st byte
SVC 0               ; output 'H'
LDRB R0, [R1,#1]    ; fetch 2nd byte
SVC 0               ; output 'e'
LDRB R0, [R1,#2]    ; fetch 3rd byte
SVC 0               ; output 'l'

. . . .
```

Not very general – should be more like:

while not at end of string
get next character from the string
output it
“get next character from the string” (1)

for (int i = 0; ; i++)
    System.out.print(message.charAt(i));

Use a second register (e.g. R2) to hold “i” around loop:

    MOV R2, #0         ; int i = 0
    ADRL R1, message 
    ...
    LDRB R0, [R1, R2] ; message.charAt(i) – address = R1 + R2
    ADD R2, R2, #1     ; i++
    SVC 0              ; System.out.print
    ...

byte address changes by 1 each time around loop
“get next character from the string” (1a)

Part way through execution …
“get next character from the string” (2)

Optimisation: avoid using both R1 and R2 for addresses

Actually change the address in R1

ADRL R1, message
...
LDRB R0, [R1]
ADD R1, R1, #1
SVC 0
...

“get next character from the string” (3)

Optimisation: avoid using  ADD R1, R1, #1

Change the address in R1 using “post-indexed” operand form

A special addressing mode that changes the address register after the fetch (or after the store, for an STR instruction)

ADRL R1, message
.
.
LDRB R0, [R1], #1 ; fetch [R1], then R1 ← R1+1
SVC 0
.
.
“while not at end of string”

Every Java String knows how long it is. But we only marked the end of the ARM string with a zero.

...  
LDRB R0, [R1], #1
CMP R0, #0
BEQ end  
...

(BTW: this way of storing strings, as used in the COMP15111 labs, is based on the C programming language, not Java)
Whole loop

while not at end of string 
get next character from the string 
output it 

ADRL R1, message

loop LDRB R0, [R1],#1
    CMP R0, #0
    BEQ end
    SVC 0
    B loop

end . . .
Question: Optimise using Conditional Instructions

e.g. $\text{ADD} \rightarrow \text{ADDEQ}$
ARM’s Indirect Addressing Modes

2 possibilities for each of:
- address used: Register, or Register + offset
- final register value: unchanged, or + offset

(Rn is Base Register) | Address = Rn | Address = Rn + offset
---|---|---
Rn unchanged | \([Rn]\) **indirect**
e.g. \([R1]\) | \([Rn, offset]\)
offset
e.g. \([R1, R2]\) or \([PC, \#offset]\)

Rn \(\Leftarrow\) Rn + offset | \([Rn], offset\) **post-indexed**
e.g. \([R1], \#1]\) | \([Rn, offset]\)! **pre-indexed**

possibilities for offset:
\#number or \#-number or Register or -Register

(or offset can be shifted – next lecture)
Examples

(simple) indirect:
LDR R0, [R1] ; R0 ⇐ value at [R1], R1 unchanged

(base +) offset:
LDR R0, [R1,#4] ; R0 ⇐ value at [R1+4], R1 unchanged
LDR R0, [R1,R2] ; R0 ⇐ value at [R1+R2], R1 unchanged

post-indexed:
LDR R0, [R1],#4 ; R0 ⇐ value at [R1], R1 ⇐ R1+4
LDR R0, [R1],R2 ; R0 ⇐ value at [R1], R1 ⇐ R1+R2

pre-indexed:
LDR R0, [R1,#4]! ; R0 ⇐ value at [R1+4], R1 ⇐ R1+4
LDR R0, [R1,R2]! ; R0 ⇐ value at [R1+R2], R1 ⇐ R1+R2

(Lots more examples in next few lectures!)
Summary of key points

Addressing: Load & Store ARM instructions only

Direct Addressing & Register Indirect Addressing

Base + Offset Addressing

ARM’s Indirect Addressing Modes
  – (simple) indirect
  – (base +) offset
  – pre/post-indexed

Example – printing a String
Your Questions
Glossary

Address
Addressing
Direct addressing
(Register) Indirect addressing
Offset
Base register
(Base +) Offset addressing
PC + Offset addressing
Byte offset
String
Post-indexed addressing
Pre-indexed addressing
For next time

Explain the difference between direct and indirect addressing. (2 marks)

Why would it be difficult to use direct addressing in ARM Load (LDR) and Store (STR) instructions? (2 marks)

Initially, R1=0x11aa, R2=0x22bb, R3=0x33cc, R4=0x44dd, R5=0x1000
– what value is in R5 after each instruction?
– what values are stored where? (4 marks)

```
STR R1, [R5]
STR R2, [R5, #4]
STR R3, [R5], #4
STR R4, [R5, #4]!
```
Exam Questions

Describe PC relative addressing & explain why it is of limited use to access large amounts of data in an ARM program. Explain how more general base plus offset addressing can be used to solve this problem. (4 marks)

Explain indirect addressing as implemented in the ARM processor, giving an example. (2 marks)

Explain base plus offset addressing as implemented in the ARM processor, giving an example. (2 marks)

Explain what is meant by “pre-indexed” and “post-indexed” addressing as implemented in the ARM processor, giving an example of each. (3 marks)
Exam Questions ctd.

This ARM code accesses a block of data starting at address “data”. What does it do? Rewrite it as a loop using a suitable addressing mode. (4 marks)

ADRL R1, data
MOV R2, #0
LDRB R0, [R1, #1]
ADD R2, R2, R0
LDRB R0, [R1, #2]
ADD R2, R2, R0
LDRB R0, [R1, #4]
ADD R2, R2, R0
LDRB R0, [R1, #8]
ADD R2, R2, R0
LDRB R0, [R1, #16]
ADD R2, R2, R0
LDRB R0, [R1, #32]
ADD R2, R2, R0