From last 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)

```c
STR R1, [R5]
STR R2, [R5, #4]
STR R3, [R5], #4
STR R4, [R5, #4]!
```
COMP15111: Introduction to Architecture
Lecture 8: Case Study – Strings, Tables

Dr. Javier Navaridas

School of Computer Science, University of Manchester

Autumn 2016
Overview & Learning Outcomes

Accessing linear data structures

– (C) Strings

– Translation tables

– More complex data tables

More features of the ARM instruction set
– Shifting and Rotation
Strings

Java strings consist of the characters and a length.

```java
String message = "Hello";
...
System.out.print (message);
```

C strings consist of the characters in bytes, terminated by a byte containing the value 0 (not character ‘0’)

```
'H' 'e' 'l' 'l' 'o' 0
```

These ARM labs use the C convention.

```assembly
message DEFB "Hello", 0
ALIGN
ADRL R0, message
SVC 3
```
String.length

**Java:**

```java
int length = message.length();
for (i=0 ; i’s character != 0; i++)
    ;
lenght = i;
```

**ARM, but always incrementing i:**

```assembly
ADRL R1, message
MOV R2, #0 ; R2 = i
again LDRB R0, [R1,R2]
ADD R2, R2, #1
CMP R0, #0
BNE again
SUB R2, R2, #1
STR R2, length
```
Question: rewrite to use conditional instructions

(e.g. $\text{SUB} \rightarrow \text{SUBGT}$)

so don’t increment and then decrement i (R2) at the end of the loop
Original version (again) – getting rid of R2

Addressing: Base in R1, Offset in R2

```assembly
ADRL R1, message
MOV R2, #0 ; R2 = i
LDRB R0, [R1,R2]
ADD R2, R2, #1
CMP R0, #0
BNE again
SUB R2, R2, #1
STR R2, length
```

post-indexed: e.g. STR R3, [R4], #1
Question: use post-indexed and save a register

(e.g. STR R3, [R4], #1)

(but not using conditional instructions)
**String.indexOf**

**Java:**
```java
int find = message.indexOf('e');
```

**ARM, using post-indexed**

```
ADRL R1, message
again LDRB R0, [R1],#1
CMP R0, #'e' ; look for 'e' instead of 0
BNE again
ADRL R0, message
SUB R0, R1, R0
SUB R0, R0, #1
STR R0, find
```
Question: what if character isn’t in string?

String.indexOf() returns –1 if it gets to the end of the string without finding the character.
Shift operations

**Shift** operations move the position of bits in a word.

Left shift operations increase the significance of each bit:

<table>
<thead>
<tr>
<th>Input:</th>
<th>x7</th>
<th>x6</th>
<th>x5</th>
<th>x4</th>
<th>x3</th>
<th>x2</th>
<th>x1</th>
<th>x0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left shifted by one place</td>
<td>x6</td>
<td>x5</td>
<td>x4</td>
<td>x3</td>
<td>x2</td>
<td>x1</td>
<td>x0</td>
<td>0</td>
</tr>
</tbody>
</table>

A one place left shift is like multiplying by 2

Right shift operations decrease the significance of each bit:

<table>
<thead>
<tr>
<th>Input:</th>
<th>x7</th>
<th>x6</th>
<th>x5</th>
<th>x4</th>
<th>x3</th>
<th>x2</th>
<th>x1</th>
<th>x0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right shifted by one place</td>
<td>0</td>
<td>x7</td>
<td>x6</td>
<td>x5</td>
<td>x4</td>
<td>x3</td>
<td>x2</td>
<td>x1</td>
</tr>
</tbody>
</table>

A one place right shift is like dividing by 2

Most processors provide some form of shift operation.
Rotation operations

Rotations are similar to shifts except the bits that ‘fall off’ one end of the word reappear at the other end.

<table>
<thead>
<tr>
<th>Input:</th>
<th>(x_7)</th>
<th>(x_6)</th>
<th>(x_5)</th>
<th>(x_4)</th>
<th>(x_3)</th>
<th>(x_2)</th>
<th>(x_1)</th>
<th>(x_0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left rotated by one place</td>
<td>(x_6)</td>
<td>(x_5)</td>
<td>(x_4)</td>
<td>(x_3)</td>
<td>(x_2)</td>
<td>(x_1)</td>
<td>(x_0)</td>
<td>(x_7)</td>
</tr>
</tbody>
</table>

\[\leftarrow\]

<table>
<thead>
<tr>
<th>Input:</th>
<th>(x_7)</th>
<th>(x_6)</th>
<th>(x_5)</th>
<th>(x_4)</th>
<th>(x_3)</th>
<th>(x_2)</th>
<th>(x_1)</th>
<th>(x_0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right rotated by one place</td>
<td>(x_0)</td>
<td>(x_7)</td>
<td>(x_6)</td>
<td>(x_5)</td>
<td>(x_4)</td>
<td>(x_3)</td>
<td>(x_2)</td>
<td>(x_1)</td>
</tr>
</tbody>
</table>

\[\Rightarrow\]
ARM shift operations

ARM does not have shift/rotate instructions, but can shift/rotate one of the inputs (register) of any instruction.

The second operand in data or address calculations may be shifted by an arbitrary number of places.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>...meaning</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSL n</td>
<td>Logical Shift Left</td>
<td>Make bits $2^n \times$ more significant least significant bit(s) zeroed</td>
</tr>
<tr>
<td>LSR n</td>
<td>Logical Shift Right</td>
<td>Make bits $2^n \times$ less significant most significant bit(s) zeroed</td>
</tr>
<tr>
<td>ASR n</td>
<td>Arithmetic Shift Right</td>
<td>Make bits $2^n \times$ less significant most significant bit(s) from sign bit</td>
</tr>
<tr>
<td>ROR n</td>
<td>ROtate Right</td>
<td>Like LSR except bits that ‘fall off’ reappear at the MS positions</td>
</tr>
</tbody>
</table>

Where ‘n’ can be a literal or register value
ARM shift operations (ctd.)

Examples:

\[
\begin{align*}
\text{MOV} & \quad \text{R0, R0, LSL \ #1} \quad ; \quad \text{R0} \leftarrow 2 \times \text{R0} \\
\text{MOV} & \quad \text{R1, R1, LSR \ #2} \quad ; \quad \text{R1} \leftarrow \text{R1}/4 \\
\text{ADD} & \quad \text{R0, R0, R0, LSL \ #2} \quad ; \quad \text{R0} \leftarrow 5 \times \text{R0}
\end{align*}
\]

Applications:

- Chiefly in arithmetic – e.g. ‘long’ division code
- Indexing into structures such as tables
- Moving bits for communications/coding
- ...
Questions:

Why two right shifts but only one left shift?

Why no rotate left?
xterm colours

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>30;47</td>
<td>30;41</td>
<td>30;42</td>
<td>30;43</td>
<td>30;44</td>
<td>30;45</td>
<td>30;46</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:30</td>
<td>1:30;47</td>
<td>1:30;41</td>
<td>1:30;42</td>
<td>1:30;43</td>
<td>1:30;44</td>
<td>1:30;45</td>
<td>1:30;46</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>37;40</td>
<td>37;41</td>
<td>37;42</td>
<td>37;43</td>
<td>37;44</td>
<td>37;45</td>
<td>37;46</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:37</td>
<td>1:37;47</td>
<td>1:37;41</td>
<td>1:37;42</td>
<td>1:37;43</td>
<td>1:37;44</td>
<td>1:37;45</td>
<td>1:37;46</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>31;47</td>
<td>31;40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:31</td>
<td>1:31;47</td>
<td>1:31;40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>32;47</td>
<td>32;40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:32</td>
<td>1:32;47</td>
<td>1:32;40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>33;47</td>
<td>33;40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:33</td>
<td>1:33;47</td>
<td>1:33;40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>34;47</td>
<td>34;40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:34</td>
<td>1:34;47</td>
<td>1:34;40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>35;47</td>
<td>35;40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:35</td>
<td>1:35;47</td>
<td>1:35;40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>36;47</td>
<td>36;40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:36</td>
<td>1:36;47</td>
<td>1:36;40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

http://www.pixelbeat.org/docs/terminal_colours/
xterm colour-codes

0: black
1: red
2: green
3: yellow
4: blue
5: magenta
6: cyan
7: grey

+ 030 = text colour, + 040 = background colour

(octal)
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>0: #000000 (black)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>#FF0000 (red)</td>
<td>1: #FF0000 (red)</td>
</tr>
<tr>
<td>2</td>
<td>#00FF00 (green)</td>
<td>2: #00FF00 (green)</td>
</tr>
<tr>
<td>3</td>
<td>#FFFF00 (yellow)</td>
<td>3: #FFFF00 (yellow)</td>
</tr>
<tr>
<td>4</td>
<td>#0000FF (blue)</td>
<td>4: #0000FF (blue)</td>
</tr>
<tr>
<td>5</td>
<td>#FF00FF (magenta)</td>
<td>5: #FF00FF (magenta)</td>
</tr>
<tr>
<td>6</td>
<td>#00FFFF (cyan)</td>
<td>6: #00FFFF (cyan)</td>
</tr>
<tr>
<td>7</td>
<td>#AAAAAA (grey)</td>
<td>7: #AAAAAA (grey)</td>
</tr>
</tbody>
</table>
Table of 24-bit (hexadecimal) RGB colour-codes

0: #000000 (black)
1: #FF0000 (red)
2: #00FF00 (green)
3: #FFFF00 (yellow)
4: #0000FF (blue)
5: #FF00FF (magenta)
6: #00FFFF (cyan)
7: #AAAAAA (grey)

To convert ansi colour to RGB colour:

(bottom 3 bits of) xterm-colour = row of table to use
ARM code for table look-up

; R0 = (bottom 3 bits of) xterm colour-code i.e. ansi colour
ADRL R1, table
MOV R2, #4 ; 1 word = 4 bytes
MUL R0, R0, R2
LDR R3, [R1, R0] ; R3 = RGB colour-code
...
table DEFW 0x000000 ; black
   DEFW 0xFF0000 ; red
   DEFW 0x00FF00 ; green
   DEFW 0xFFFF00 ; yellow
   DEFW 0x0000FF ; blue
   DEFW 0xFF00FF ; magenta
   DEFW 0x00FFFF ; cyan
   DEFW 0xAAAAAAA ; grey
Address calculation

ADRL R1, table
MOV  R2, #4
MUL  R0, R0, R2
LDR  R3, [R1, R0]

Often want to convert a number to a word-offset

Can “shift” a register in most instructions (i.e. $2^n$ or $/2^n$)

ADRL R1, table
LDR  R3, [R1, R0, LSL #2]

“Logical Left Shift” the value in R0 up 2 bits (i.e. $2^2$)
Including colour-names

“magenta\0” = 8 characters (2 words)

| Table | DEFW 0x000000 | DEFB "black", 0, 0, 0 | DEFW 0xFF0000 | DEFB "red", 0, 0, 0, 0, 0 | DEFW 0x00FF00 | DEFB "green", 0, 0, 0 | DEFW 0xFFFF00 | DEFB "yellow", 0, 0 | DEFW 0x0000FF | DEFB "blue", 0, 0, 0, 0 | DEFW 0xFF00FF | DEFB "magenta", 0 | DEFW 0x00FFFF | DEFB "cyan", 0, 0, 0, 0 | DEFW 0xAAAAAA | DEFB "grey", 0, 0, 0, 0 |
Table look-up

ADRL R1, table
MOV R2, #12 ; colour-code + name = 12 bytes
MUL R0, R0, R2
LDR R3, [R1,R0]

ARM arithmetic instructions can also use shifts on the last register.

Q: what do the “ADD” and “MOV” instructions below do?

ADRL R1, table
ADD R0, R0, R0,LSL #1
MOV R0, R0, LSL #2
LDR R3, [R1,R0]
Question: What does this code do?

assume R0 = bottom 3 bits of an xterm colour-code

ADRL  R1, table
ADD   R0, R0, R0, LSL #1
MOV   R0, R0, LSL #2
LDR   R0, [R1, R0]!
SVC   4
ADD   R0, R1, #4
SVC   3

print integer in R0
print string (address in R0)
Summary of key points

Accessing linear data structures

– (C) Strings

– Translation Tables

More features of the ARM instruction set

– Shifts \{LSL, LSR, ASR, ROR\}
Your Questions
Glossary

String
C String
ALIGN
String length
Data table
Translation table
Table look-up
Shift, Rotate
Logical Shift Left, LSL
Logical Shift Right, LSR
Arithmetic Shift Right, ASR
Rotate Right, ROR
What is the effect on R0 (in terms of an equivalent multiplication) of each of the following ARM instructions?

```
ADD R0, R0, R0, LSL #3
RSB R0, R0, R0, LSL #2
```

Write down an **ADD** instruction that has the same effect as the **RSB** instruction above.

(4 marks)

Explain, with the aid of an example, how pre-indexed or post-indexed addressing can be used to improve the efficiency of a program which uses offset addressing to access the characters of a string in a loop. (4 marks)
Exam Question

The following ARM code prints, in hexadecimal format, a value in register R1.

```
MOV R2, #8
loop  MOV R0, R1,LSR #28
      CMP R0, #9
      BHI letter
      ADD R0, R0, #'0'
      SVC 0
      B    next
letter ADD R0, R0, #'A'−10
      SVC 0
next   MOV R1, R1,LSL #4
      SUBS R2, R2, #1
      BNE  loop
```
ctd.

Explain the purpose of each line of code. (10 marks)

Note: it is not enough to state the effect of each instruction. You should also indicate how that instruction contributes to the overall goal. For example, the correct explanation of the first instruction

MOV R2, #8

is not just: “Move the value 8 into R2” but rather: “Register R2 is used as a loop counter. It is initialised to the value 8 because a 32-bit value is printed as 8 hexadecimal digits.”

Rewrite this code taking advantage of conditionally executed instructions to remove unnecessary branches. (Note that the LS condition is the opposite of the HI condition.) (10 marks)