From last 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)
COMP15111: Introduction to Architecture

Lecture 5: Integer Arithmetic

Dr. Javier Navaridas

School of Computer Science, University of Manchester

Autumn 2016
Overview & Learning Outcomes

Java $\rightarrow$ ARM: more complex integer expressions

Making best use of registers

More about ARM arithmetic instructions (& more ARM instructions)

Conditionals & status flags
Integer Arithmetic

**Registers**: R0, R1, ..., R15

Can’t be used for more than one thing at once! (e.g. R15 = PC)

Needed everywhere – Reuse when safe

Why doesn’t ARM provide more registers?

- space within 32-bit instructions
- space on silicon (originally)
- more → slower

Modern compiler: analyse program to discover best use for registers
\[ a = b + c + d + e; \]

Flow of values:

\[ ( (b + c) + d ) + e \]

Need 1 “STR” instruction to store result “a”
Need 4 “LDR” instructions to load “b”, “c”, “d”, “e”
Need 3 “ADD” instructions for 3 “+”s
How many registers?

Can change sequence to use different numbers of registers:

7 registers:

LDR R0, b
LDR R1, c
LDR R2, d
LDR R3, e
ADD R4, R0, R1 ;b+c
ADD R5, R4, R2 ;+d
ADD R6, R5, R3 ;+e
STR R6, a

2 registers:

LDR R0, b
LDR R1, c
ADD R0, R0, R1 ;b+c
LDR R1, d
ADD R0, R0, R1 ;+d
LDR R1, e
ADD R0, R0, R1 ;+e
STR R0, a
Sequence of actions

b → R0
  ↓
     +
    ↓
   R4
  ↓
   +
  ↓
 R5
  ↓
   +
  ↓
 R6
  ↓
a

d → R1
  ↓
 R2
  ↓
 e → R3

b ← R0
  ↓
   +
  ↓
   R0
  ↓
   +
  ↓
   R1
  ↓
 a
Several statements

\[
a = b + c + d + e; \\
e = a \times b \times c \times d;
\]

5 registers:

\[
\begin{align*}
LDR & \ R0, \ b \\
LDR & \ R1, \ c \\
LDR & \ R2, \ d \\
LDR & \ R3, \ e \\
ADD & \ R4, \ R0, \ R1 \ ;b+c \\
ADD & \ R4, \ R4, \ R2 \ ;+d \\
ADD & \ R4, \ R4, \ R3 \ ;+e \\
STR & \ R4, \ a \\
MUL & \ R3, \ R4, \ R0 \ ;a\times b \\
MUL & \ R3, \ R3, \ R1 \ ;*c \\
MUL & \ R3, \ R3, \ R2 \ ;*d \\
STR & \ R3, \ e
\end{align*}
\]

2 registers:

\[
\begin{align*}
LDR & \ R0, \ b \\
LDR & \ R1, \ c \\
ADD & \ R0, \ R0, \ R1 \ ;b+c \\
LDR & \ R1, \ d \\
ADD & \ R0, \ R0, \ R1 \ ;+d \\
LDR & \ R1, \ e \\
ADD & \ R0, \ R0, \ R1 \ ;+e \\
STR & \ R0, \ a \\
LDR & \ R1, \ b \\
MUL & \ R0, \ R0, \ R1 \ ;a\times b \\
LDR & \ R1, \ d \\
MUL & \ R0, \ R0, \ R1 \ ;*d \\
STR & \ R0, \ e
\end{align*}
\]
Reusing variables

Short sequences: using fewer registers is (normally) just as fast

Longer sequences: tend to reuse variables, so try to keep them in registers

Extra LDR instructions (loading “b”, “c”, “d” twice) increase
  • space used: more instructions
  • time taken: more instructions to obey, more memory accesses for the variables, more memory accesses for the extra instructions
  • energy cost: more activity ⇒ reduced battery life
Sub-expressions

More complex expressions may need an extra register for a sub-expression

\[ a = (b \times c) + (d \times e); \]

keep “b*c” in R0, evaluate “d*e” using R1 and R2.

```
LDR R0, b
LDR R1, c
MUL R0, R0, R1 ; b * c
LDR R1, d
LDR R2, e
MUL R1, R1, R2 ; d * e
ADD R0, R0, R1 ; +
STR R0, a
```
Question

\[ a = (b + c) \times (d + e) \times (b + d); \]
a = b - e + c * d;

‘natural” order:

LDR R0, b
LDR R1, e
SUB R0, R0, R1 ;b-e
LDR R1, c
LDR R2, d
MUL R1, R1, R2 ;c*d
ADD R0, R0, R1 ; +
STR R0, a

(b - e) + (c * d)
uses 3 registers

reordered:

LDR R0, c
LDR R1, d
MUL R0, R0, R1 ;c*d
LDR R1, b
ADD R0, R0, R1 ;+b
LDR R1, e
SUB R0, R0, R1 ;-e
STR R0, a

( (c * d) + b ) - e
uses 2 registers

Same space and time – but can use R2 for something else
Rearranging expressions

Musn’t change its value.

Limited register size makes this trickier.

e.g. 3 very large (positive) numbers:
\[(a - b) + c : a-b small, ok to add to c\]
\[(a + c) - b : a+c might be too big for register\]

Problems common when calculations allow fractional parts (float)

Possible problems with integer division, e.g.:

\[
\frac{3}{2} + \frac{5}{2} \Rightarrow 3 \\
\frac{3 + 5}{2} \Rightarrow 4
\]
Division

(original) ARM does not have a built-in division instruction
Must obey many instructions instead

Someone else writes this code, and we use it
(a bit like using “System.out.println” in Java)

Use BL to call a (pre-defined) method (lectures 9/10)
“arithmetic” operations (ADD, SUB, CMP etc.) can use short literals instead of the last register

e.g. ADD R0, R1, #100

MUL can’t!

STR and LDR don’t – why not?
Expressions

Literals can be any *expression* which the assembler can evaluate (and code as a literal!).

\[ y = x + 1 + 2; \]

\[
\begin{align*}
\text{ldr r0, x} \\
\text{add r0, r0, #1} \\
\text{add r0, r0, #2} \\
\text{str r0, y}
\end{align*}
\]

\[
\begin{align*}
\text{ldr r0, x} \\
\text{add r0, r0, #(1 + 2)} \\
\text{str r0, y}
\end{align*}
\]

This makes more sense with *symbols*:

\[
\begin{align*}
\text{ldr r0, x} \\
\text{add r0, r0, #k1} \\
\text{add r0, r0, #k2} \\
\text{str r0, y}
\end{align*}
\]

\[
\begin{align*}
\text{ldr r0, x} \\
\text{add r0, r0, #(k1 + k2)} \\
\text{str r0, y}
\end{align*}
\]

Where \( \{k1, k2\} \) are *constants*
### ARM data operations – the ‘complete’ set

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Dest</th>
<th>Op1</th>
<th>Op2</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADD</strong></td>
<td>Rd</td>
<td>Rn</td>
<td>Rm/#</td>
<td>$Rn + Rm$</td>
</tr>
<tr>
<td><strong>ADC</strong></td>
<td>Rd</td>
<td>Rn</td>
<td>Rm/#</td>
<td>$Rn + Rm + c$</td>
</tr>
<tr>
<td><strong>SUB</strong></td>
<td>Rd</td>
<td>Rn</td>
<td>Rm/#</td>
<td>$Rn - Rm$</td>
</tr>
<tr>
<td><strong>RSB</strong></td>
<td>Rd</td>
<td>Rn</td>
<td>Rm/#</td>
<td>$Rm - Rn$</td>
</tr>
<tr>
<td><strong>SBC</strong></td>
<td>Rd</td>
<td>Rn</td>
<td>Rm/#</td>
<td>$Rn - Rm - \overline{c}$</td>
</tr>
<tr>
<td><strong>RSC</strong></td>
<td>Rd</td>
<td>Rn</td>
<td>Rm/#</td>
<td>$Rm - Rn - \overline{c}$</td>
</tr>
<tr>
<td><strong>CMP</strong></td>
<td>--</td>
<td>Rn</td>
<td>Rm/#</td>
<td>$Rn - Rm$</td>
</tr>
<tr>
<td><strong>CMN</strong></td>
<td>--</td>
<td>Rn</td>
<td>Rm/#</td>
<td>$Rn + Rm$</td>
</tr>
<tr>
<td><strong>MOV</strong></td>
<td>Rd</td>
<td>--</td>
<td>Rm/#</td>
<td>$Rm$</td>
</tr>
<tr>
<td><strong>MVN</strong></td>
<td>Rd</td>
<td>--</td>
<td>Rm/#</td>
<td>$\overline{Rm}$</td>
</tr>
<tr>
<td><strong>AND</strong></td>
<td>Rd</td>
<td>Rn</td>
<td>Rm/#</td>
<td>$Rn &amp; Rm$</td>
</tr>
<tr>
<td><strong>BIC</strong></td>
<td>Rd</td>
<td>Rn</td>
<td>Rm/#</td>
<td>$Rn &amp; \overline{Rm}$</td>
</tr>
<tr>
<td><strong>ORR</strong></td>
<td>Rd</td>
<td>Rn</td>
<td>Rm/#</td>
<td>$Rn \mid Rm$</td>
</tr>
<tr>
<td><strong>EOR</strong></td>
<td>Rd</td>
<td>Rn</td>
<td>Rm/#</td>
<td>$Rn \hat{\mid} Rm$</td>
</tr>
<tr>
<td><strong>TST</strong></td>
<td>--</td>
<td>Rn</td>
<td>Rm/#</td>
<td>$Rn &amp; Rm$</td>
</tr>
<tr>
<td><strong>TEQ</strong></td>
<td>--</td>
<td>Rn</td>
<td>Rm/#</td>
<td>$Rn \hat{\mid} Rm$</td>
</tr>
</tbody>
</table>
ARM data operations

There are 16 ($= 2^4$) ‘regular’ data operations on an ARM (other processors have similar sets of operations)

<table>
<thead>
<tr>
<th>cond</th>
<th>00</th>
<th>I</th>
<th>Op</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Rm/#</th>
</tr>
</thead>
</table>

Of these, some are rarely used: the **emboldened** ones are by far the most useful, especially at this stage.

- Some (e.g. ‘MOV’) only use one source
- Some (e.g. ‘CMP’) have no destination (?!)

‘MUL’ – and some other related multiplication instructions – are coded differently which imposes other restrictions, particularly in forbidding literals.

ARM has no inbuilt ‘divide’: this must be done in software. (integer division is rare in ‘real-world’ programs)
Status Flags

In addition to its 16 registers \{R0-R15\} ARM has some 1-bit status flags

Only four need concern us:

- **Negative**: a previous result was negative (or not)
- **Zero**: a previous result was zero (or not)
- **Carry**: a previous add/subtract generated a carry
- **Overflow**: a previous add/subtract went ‘out of range’

Many processors (e.g. Pentium) have a similar flag set

In ARM: (Current) Program Status Register (CPSR)

Note also the Saved Program Status Register (SPSR) holds a copy of CPSR when dealing with exceptions
Status Flags

Flags are used to direct conditional execution... branches and other instructions can be made conditional just append the condition (see table) to the mnemonic

Flags are altered by (e.g.) CMP ...
... or any data operation by appending ‘S’ to the mnemonic.

```
SUBS R0, R1, R2 ; Like CMP but keep result
```

Carry is also used for (e.g.) extended arithmetic:
```
ADDS R0, R0, R2 ; Add LS words & set flags
ADC R1, R1, R3 ; Add MS words with carry
```

64-bit add using register pairs: $R1:R0 := R1:R0 + R3:R2$

This allows operations on arbitrary-size variables!
### Conditions

‘Success’ is calculated from the state of the flags

<table>
<thead>
<tr>
<th>AL or blank</th>
<th>ALways</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>EQual</td>
</tr>
<tr>
<td></td>
<td>Not Equal</td>
</tr>
<tr>
<td>GE</td>
<td>signed Greater than or Equal</td>
</tr>
<tr>
<td></td>
<td>signed Less Than</td>
</tr>
<tr>
<td>LT</td>
<td>signed Greater Than</td>
</tr>
<tr>
<td></td>
<td>signed Less than or Equal</td>
</tr>
<tr>
<td>GT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>LE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>HS or CS</td>
<td>unsigned Higher or Same, Carry Set</td>
</tr>
<tr>
<td>LO or CC</td>
<td>unsigned LOwer, Carry Clear</td>
</tr>
<tr>
<td>HI</td>
<td>unsigned HIgher</td>
</tr>
<tr>
<td>LS</td>
<td>unsigned Lower or Same</td>
</tr>
<tr>
<td>MI</td>
<td>MInus (negative)</td>
</tr>
<tr>
<td>PL</td>
<td>PLus (positive or zero)</td>
</tr>
<tr>
<td>VS</td>
<td>oVerflow Set</td>
</tr>
<tr>
<td>VC</td>
<td>oVerflow Clear</td>
</tr>
</tbody>
</table>
Other processing examples

**RSB (Reverse Subtract)**

```
SUB R0, R1, R2;  R0 = R1 - R2
RSB R0, R1, R2;  R0 = R2 - R1
```

**So what?**
- We can use a literal as the first operand:
  ```
  RSB R1, R0, #0 ; R1 = 0-R0 = -R0
  ```

**MLA (Multiply and Add)**

```
MLA R10,R11,R12,R13 ; R10 = (R11 * R12) + R13
```

**Variant of MUL:** can only use registers as operands
Question

ARM code for: \[ a = (2 - b) \times (c - 2); \]
Question

ARM code for: \( a = (b \times 2) + (c \times -2) \);
Summary of key points

Java → ARM: more complex integer expressions
– may need extra register e.g. \((a \times b) + (c \times d)\)
– rearrange expression to simplify it e.g. \((y \times z) + x\)

Making best use of registers:
– look at bigger chunks of Java
– keep variables in registers

More about ARM arithmetic instructions:
– no division!
– more about short literals
  RSB, MOV
  (CMN, MVN usually handled by assembler)
– MLA
Your Questions
Glossary

Expression
Sub-expression
Rearranging an expression
RSB instruction
CMN instruction
MOV instruction
MLA instruction
Explain the similarities & differences between the ARM instructions \texttt{SUB R0, R1, R2} & \texttt{RSB R0, R1, R2} (2 marks)

Explain the similarities & differences between the ARM instructions \texttt{CMP R0, R1} & \texttt{CMN R0, R1} (2 marks)

Explain why it can be useful to have an “RSB” operation as well as an “SUB” operation in the ARM instruction set. Give an example to illustrate your answer. (2 marks)
Exam Questions

The integer variables $w$, $x$, $y$, $z$ are in memory and can be accessed by simple load or store instructions. Give ARM code for each of the following blocks of assignment statements, making your code as efficient as possible.

a) Give separate code for each statement (i.e. loading variables from memory at the start and storing the result at the end).

b) Give code for all the statements in the block taken together.

c) Compare the efficiency of your answers to parts (a) and (b), in terms of the number of instructions, memory accesses, and registers used.

block 1:

$$y = y - x; \quad x = y \times x; \quad z = -x; \quad y = 3 - y; \quad z = y + x; \quad x = z \times y;$$

block 2:

$$w = (w \times z) - y; \quad y = y + 1; \quad x = 2 - x; \quad z = (z - y) \times w;$$