From last time

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)
From last time ctd.

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)
COMP15111: Introduction to Architecture
Lecture 9: Methods

Dr. Javier Navaridas

School of Computer Science, University of Manchester

Autumn 2016
Overview & Learning Outcomes

What happens during a Java method call

A simple but naïve solution

A better solution: use a Stack

Some more ARM instructions and registers
Java Method declaration and calls

Declare method in one place:
int age (int day, int month, int year) {
    
}

Use it from several places (possibly with different parameters):
myage = age (myday, mymonth, myyear);
...
agenextyear = age (myday, mymonth, myyear-1);
...
if (age (myday, mymonth, myyear) == age (1, 1, myyear-1))...

Animation: method calls & returns
Sequence of events during a method call/return

Call:
(1) process method arguments
(2) remember where to restart
(3) branch to method

Inside method:
(4) use method parameters
(5) perform body of method
(6) set result
(7) branch back to restart point

Back at restart point, just after (3):
(8) use result
Call and method need to agree:

(1) + (4) : where to put method arguments/parameters

(2) + (7) : where to put restart point

(5) : how to use registers safely

(6) + (8) : where to put result
Use registers as much as possible

Result: e.g. R0

Parameters: e.g. R0 = day, R1 = month, R2 = year

Restart point: R14 (LR – Link Register)

e.g. last part of call to method:

```
ADR LR, restart ; (2) restart point
B  age           ; (3) branch to method
restart STR R0, myage ; (8) use result
```

method ends with:

```
MOV PC, LR       ; (7) return to restart point
```

Special BL “Branch-and-Link” instruction

e.g. BL age does same as (2) + (3) above
Complete (simple) example

```c
myage = age (myday, mymonth, myyear);
...
int age (int day, int month, int year) {
    ...
}
```

```assembly
    LDR R0, myday ; (1) pass arguments
    LDR R1, mymonth ; ”
    LDR R2, myyear ; ”
    BL age ; (2,3) call method, save return point
    STR R0, myage ; (8) use result

    age . . . ; use R0, R1, R2 (4) parameters
    . . . ; to calculate (5) body
    . . . ; result in R0 (6) result
    MOV PC, LR ; (7) branch back
```
Question: ARM code for this

```c
int add (int x, int y) {
    return x+y;
}
```

```
a= add(c, d);
...
```
How to use registers safely (5)

Save any other registers used by method (e.g. just after start), Restore at end (e.g. just before return)

“callee saved” (v. “caller saved”)

e.g.

age    STR R4, temp
      ...
      method body can use R4
      ...
      LDR R4, temp
      MOV PC, LR
      temp  DEFW 0
What if one method calls another?

```java
int age (int day, int month, int year) {
    ...
    System.out.println("trace " + n);
    ...
}
```

Need to save LR and parameter registers

```assembly
age  STR LR, temp1
...
STR R0, temp2
... (put println’s argument in R0)
BL println
LDR R0, temp2
...
LDR PC, temp1
temp1 DEFW 0
temp2 DEFW 0
```
More problems

Recursion
a method can call any method, including itself
→ maybe one “temp” per register won’t be enough

Efficiency
space occupied by all those “temp”s mostly unused at any given moment during run-time

We need another way to save and restore registers

Method calls and returns are nested (like brackets)
...so register saves and restores are also nested
...so what we need is something that works the same way
A stack is like a very heavy pile of books:

– you can take anything off the top of the pile (“pop”)

– you can add anything to the top of the pile (“push”)

– you can’t move anything into/out of the pile except at the top!

This is simple, and exactly what we need for method calls

LIFO = last-in, first-out
Using a Stack on ARM

On most computers:
– stacks start at a large address,
– which decreases as the stack grows

ARM has:
– stack-pointer register (SP, R13)
– 4 different ways of using a stack!

We will use commonest version:
– SP = address of top word on stack (stack-top)
– decrease SP to push, increase to pop

Push: e.g. `STR R0, [SP,#-4]!` (pre-indexed)
Pop: e.g. `LDR R0, [SP],#4` (post-indexed)
int age (int day, int month, int year) {
...
    System.out.println("trace " + n);
...
}

(same example, better answer)
age STR LR, [SP,#-4]!
...
    STR R0, [SP,#-4]!
    ... (put println's argument in R0)
BL println
LDR R0, [SP],#4
...
    LDR PC, [SP],#4

We only use extra stack locations when needed, whereas temp1 and temp2 always allocated
Question: What happens – what is output?

main   ADR   SP,stack
BL     print
SVC    2

print  STR   LR, [SP,#-4]!
MOV    R0, #'?'
BL     print2
SVC    0
LDR    PC, [SP],#4

print2 SVC    0
MOV    R0, #'@'
SVC    0
MOV    PC, LR

DEFS  400

stack
Summary of key points

What happens during a Java method call

A simple but naïve solution

A better solution: use a Stack

Some more ARM instructions (BL) and registers (LR, SP)
Your Questions
Glossary

Method declaration
Method call
Method entry
Method exit/return
Method result
Method parameter
Method argument
LR register
BL instruction
Recursion
Stack (LIFO)
Pop
Push
SP register
For next time

Explain what the ARM instruction BL does, and how control is returned to the calling code when the method finishes. If the method itself calls another method, how does this affect the return? (4 marks)

Why is extra memory (i.e. a stack) useful in the implementation of method calls? Why do we use a stack rather than fixed memory locations? (4 marks)

A stack is implemented using SP and pre-/post-indexing instructions. Give code for the 'push' and 'pop' operations and explain the value in SP after each operation. (4 marks)
A “leaf method” is a method that doesn’t call any other method. Explain why ARM code for a leaf method can be simpler than the code for a “non-leaf” method (i.e. a method that does call another method). Give example instruction sequences for both kinds of methods as part of your explanation. (6 marks)

A method “main” calls a second method “work”. Method “work” needs to use registers R0, R1 & R2, which are also used by “main”. What code is necessary at entry to and exit from “work” to preserve the values in the registers for use by “main”? (4 marks)

During method calls, arguments can be passed either in registers or on the stack. Explain both mechanisms, giving example code. Explain the advantages and disadvantages of each mechanism. (8 marks)
There are two alternative mechanisms for preserving the contents of a register across a method call (and return):

– “caller saved” is when a register is saved and restored by the method making the call, in case the method being called uses it.
– “callee saved” is when a register is saved and restored by the method being called, in case the method making the call uses it.

Give example code for both mechanisms. Can you think of any circumstances where one mechanism or the other would be more efficient? (8? marks)