January 2005 Question 1

a) Explain the concept of a Pseudo Instruction in ARM assembly language. (2 marks)

The complexity of coding of various aspects of the ARM instruction set makes it difficult to know if a particular real instruction is valid (e.g. whether a PC relative mode is possible). The assembler therefore has idealised pseudo instructions which, when assembled, may translate to more than one real instruction.

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

The ADRL instruction is used to get the address of labelled data into a register. If this is possible using a PC relative mode, the planted instruction is just an ADD or SUB to/from the PC to the register. However, if the literal cannot be represented, it is necessary to plant a constant within the range of PC relative addressing and then load this to the register.

c) What is bytecode in the context of Java and how does it get executed by a real processor? (4 marks)

The Java compiler does not produce real machine instructions. It produces instead, bytecode, which is a stack based (zero address) instruction set. This has the advantages of portability and compactness. The simplest way to execute bytecode is with a software interpreter. However, performance considerations have led to the development of dynamic compilation virtual machines. Here bytecode is initially interpreted but, if monitoring shows that the code is being heavily executed, the bytecode is translated to native machine code.

d) (d) [1] 3-address instruction style: each instruction has 3 full operands (2 source, 1 destination) and an operation: a=b+c; a=a+d
[1] store-load instruction style: each operation has 3 register operands (2 source, 1 destination) and there are extra instructions to load and store registers: r1=b; r2=c; r3=d; r1=r1+r2; r1=r1+r3; a=r1
[e] [1] bit = single piece of information (0 or 1) - e.g. 0 = "no", 1 = "yes"
[1] a byte is 8 bits, so there are 2^8 different possible patterns e.g. 01000001 = "A", 01000010 = "B", etc. (ASCII, Unicode)
[1] a word is 16/32/64 bits - the size of a register/integer/address
ARM 32 bits: 2^32 patterns, 0...0 = 0, 0...01 = 1, 0...010 = 2 etc.
[f] [1] repeatedly divide by 2, noting remainders (in reverse order)
97/2=48r1, 48/2=24r0, 24/2=12r0, 12/2=6r0, 6/2=3r0, 3/2=1r1, 1/2=0r1
[1] replace groups of bits by digits: octal (in 3s) = 141, hex (in 4s) = 61
[g] [1] (i) sit in loop asking device if it is ready - e.g. if nothing else to do (not normally)
[1] (ii) do something else, wait for peripheral to alert CPU, so it pauses the something else and deals with the peripheral e.g. slow peripherals
[1] (iii) peripheral can pass data block to/from memory, interrupts at end of block - e.g. fast peripherals (discs etc.)

January 2005 Question 2

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

In direct addressing, memory addresses can exist directly in a machine instruction. In indirect addressing, a register is used as the source of address.

b) Explain base plus offset addressing as implemented in the ARM processor. (2 marks)

Base plus offset addressing is a form of indirect addressing where the address in memory is formed by the contents of a register together with a positive or negative offset which is a literal in the instruction.

c) Describe PC relative addressing and discuss 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)
The literal value in an ARM instruction is limited to 8 bits but can be shifted by various powers of two. In general, this cannot succeed in addressing elements of a block of data which is greater than a few k and within a few k of the current program counter. General base plus offset addressing can use any register to hold the origin of a block of data which can be anywhere in memory and can be changed and indexed as necessary.

d) Explain the concept of auto indexing with the aid of an example. (4 marks)
Auto indexing is an addressing mode where a register is used as source of address, but the register can be updated to its current value plus or minus a literal either before or after the access is made. This allows structures to be accessed and indexed in a single instruction. An array example would be appropriate.

e) Discuss the circumstances in which the Link Register (LR) in the ARM processor can be used in a subroutine call without the aid of a stack. You should include an explanation of why a stacking LR is necessary in many circumstances. (4 marks)
The Link Register holds the return address following a Branch and Link (BL) subroutine call. However, if that subroutine in turn makes a further BL call, the Link Register will get overwritten. As subroutine calls can in general be nested to any depth, we must have a general mechanism to remember a chain of return addresses. Putting them on a software stack solves this problem.

f) A subroutine is passed two integer parameters which it adds together and returns the result. Assuming that the subroutine does place the Link Register on the stack, sketch the ARM code required for a simple implementation of the subroutine and draw diagrams of the resulting stack structure. (4 marks)

name STR LR, [SP,#-4]!
LDR R0, [SP,#4]
LDR R1, [SP,#8]
ADD R0, R0, R1
LDR PC, [SP],#12

Stack is:
p2,p1 on call,
p2,p1,LR following the push and
empty on return

January 2005 Question 3

(a)
[1] stored program: program instructions held in memory, along with data
[1] fetch-execute cycle: fetch each instruction in turn from memory, increment PC, decode it, obey it and start again

PC=0, fetch "B start" from memory to CPU, PC+=4
obey: PC=16
PC=16, fetch "LDR R0,one", PC+=4
obey: fetch word one @4 from memory, R0=23
PC=20, fetch "STR R0, three", PC+=4
obey: store 23 in word three@12
PC=24, fetch "LDR R1, two", PC+=4
obey: fetch word two @8 from memory, R1=45
PC=28, fetch "ADD R2, R1, R0", PC+=4
obey: R2=23+45
PC=32, fetch "STR R2, one", PC+=4
obey: store 68 in word one@4
PC=36, fetch "SWI 2", PC+=4
obey: ask operating system to stop program

[1 for PC and PC+=]
[1 for fetch instructions]
[1 for fetch/store operands, including changes to words one and three]
[1 for use of R0/R1/R2]

(b)
[1] assembler: assembly code to binary; compiler: HLL to assembler/binary
[1] assembler: essentially 1 to 1; compiler: 1 to many, optimisation

[1] lexical analysis: group characters into words, drop spaces and comments
[1] syntactic analysis: group words into statements etc.
[1] semantic analysis: check identifiers (& types) declared/used correctly
[1] code generation = output translation

(c)

(ii)

depends somewhat on what they assume for part (i), so the marking has
to be flexible - they can get the mark if they have done it in (i)
[1] keep variables in registers - but remember to load at start and store
at end
[1] conditional arithmetic instructions - probably not making much
difference, as the then and else bodies are too big
[1] CMP #0 -> _S or Branch to branch or any other little improvement

Old compulsory exam questions

(a) (i) 0x5A (ii) 0132
(b) 147
(c) (i) 0x20 (ii) 0x5F
(d) pushes R0, R3, R4, R5 onto stack, SP decremented by 16
(e) 25
(f) ADDS does the ADD (doubling R0) and then compares the result with zero, leaving the result of the comparison in the CPSR register, ready for a subsequent conditional operation to use (e.g. BNE or ADDLT)

(g) (i) R0 = 0x36 (ii) R0 = 0x48

(h) see (f) above

(i) 0x65

(j) The 4 instructions do:

memory word 0x1000 = 0x11aa, R5 unchanges
memory word 0x1004 = 0x22bb, R5 unchanged
memory word 0x1000 = 0x33cc, R5 = 0x1004
R5 = 0x1008, memory word 0x1008 = 0x44dd
so byte 0x1000 = 0xcc, byte 0x1001 = 0x33, byte 0x1002 = 0x00, byte 0x1003 = 0x00
byte 0x1004 = 0xbb, byte 0x1005 = 0x22, byte 0x1006 = 0x00, byte 0x1007 = 0x00
byte 0x1008 = 0xdd, byte 0x1009 = 0x44, byte 0x100a = 0x00, byte 0x100b = 0x00

(k) ROM is 0x00000000 to 0x00000000 + 2^{3} \times 2^{20} - 1 i.e. 0x00800000 -1 i.e. 0x007FFFFF
so RAM is 0x00800000 to 0x00800000 + 2^{6} \times 2^{20} - 1 i.e. 0x04800000 -1 i.e. 0x047FFFFF

(l) 4 2 1 4 1 4 4 12

<table>
<thead>
<tr>
<th>Cond</th>
<th>00</th>
<th>OpCode</th>
<th>00</th>
<th>Rn</th>
<th>Rd</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>00</td>
<td>09</td>
<td>0010</td>
<td>0</td>
<td>1110</td>
<td>1100</td>
</tr>
</tbody>
</table>

(i) R12 (ii) R14 (iii) 0010 (iv) no (S=0)

January 2002 Question 2

a) 27

1st instruction does: R0 = R0 + (R0 \times 2^{3}) i.e. *9
2nd instruction does: R0 = - R0 + (R0 \times 2^{2}) i.e. *3

b) Only certain literal values can be used in ARM instructions:

i) limited number of bits in a 32-bit instruction

ii) by shifting the literal within a 32-bit number

iii) R2 = !(0xff \times 2^{something}) – you might guess that the “something” was 6, or 12, or 32-6 or 32-12 e.g. 32-12 gives !(0xff00000) = 0xf00fffff

c) i) we did this example in the lectures!

g = CMP R0, R1
    SUBGT R0, R0, R1
    SUBLT R1, R1, R0
    BNE gcd

ii) in some cases, makes code more compact and/or faster

iii) code is 4 words instead of 7, and takes 4 instructions per loop instead of 5

January 2003 Question 2

a) see previous question part (cii)

b) HexOut is a label which identifies the subroutine.

Register R2 is used as a loop counter and is initialised to the value 8 because there are 8 hexadecimal digits to be printed

HexOut MOV R2, #8

loop is a label that indicates the start of a loop in the routine

make R0 hold the (next) most significant hex digit from R1

loop MOV R0, R1,LSR #28
If R0 is 0 to 9
    CMP R0, #9
    BHI ascii
then convert numbers 0-9 to characters "0"-"9"

ADD R0, R0, #"0"
and print it
    SWI  0
and skip the "else" part
    B  next
else convert numbers 10-15 to characters "A"-"F"
    ascii  ADD  R0, R0, #"A"-10
and print it
    SWI  0
Either way, now remove the most significant hex digit from R1, and put the
next most significant digit at the top of R1
    next  MOV  R1, R1,LSL #4
decrement the loop counter
    SUBS  R2, R2, #1
and if it is not yet zero, go back to the start of the loop
    BNE  loop
otherwise exit the routine, going back to where it was called from
    MOV  PC, LR

c) HexOut  MOV  R2, #8
    loop  MOV  R0, R1,LSR #28
            CMP  R0, #9
            ADDLO  R0, R0, #"0"
            ADDHI  R0, R0, #"A"-10
            SWI  0
            MOV  R1, R1,LSL #4
            SUBS  R2, R2, #1
            BNE  loop
            MOV  PC, LR

January 2004 Question 2
a) A stack is something you can add values (push) to, or take values (pop) from, but only at one end. (etc. – perhaps a little example)
b) decrement SP to push, increment to pop, SP points to current top of stack
c) STR  R0, [SP, #-4]! to push R0, LDR  R0, [SP], #4 to pop R0.
d) LDR/STR can only load/store on register at a time. LDM/STM can load/store any or all registers in one instruction, but only in a fixed order.
f) On entry:  STMFD  SP!, {R0-R2, LR}
On exit:  LDMFD  SP!, {R0-R2, PC}

January 2004 Question 3 - This is all bookwork, from lecture 15 (interrupts and peripherals)