

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING  
EXAMINATIONS 2011

EEE Part II: MEng, BEng and ACGI

Corrected Copy

(2 (6)

## **INTRODUCTION TO COMPUTER ARCHITECTURE**

Monday, 6 June 2.00 pm

Time allowed: 1:30 hours

**There are THREE questions on this paper.**

**Answer ALL questions.**

**Question 1 carries 40% of the marks. Questions 2 and 3 carry equal marks (30% each).**

Any special instructions for invigilators and information for candidates are on page 1.

Examiners responsible:

First Marker(s): Clarke, T.

Second Marker(s): Demir, Y.

**Special instructions for invigilators**

*The sheet Exam Notes 2011 should be distributed with the Examination Paper.*

**Special instructions for students**

*The prefix &, or suffix <sub>(16)</sub>, introduces a hexadecimal number, e.g: &1C0, 1C0<sub>(16)</sub>.*

*Unless otherwise specified negative numbers are represented in two's complement.*

*Unless otherwise specified machine addressing is little-endian.*

*The sheet Exam Notes 2011, as published on the course web pages, is provided and contains reference material.*

*Answer ALL the questions.*

## The Questions

1.

a)

- (i) Calculate the 32 bit IEEE-754 representation of  $-31.25_{(10)}$ , writing your answer as 8 hexadecimal digits.
- (ii) The value &FF is stored in an N bit register. State the two's complement signed value of the register in the two cases  $N=16$  and  $N=8$ .

[8]

b)

A write-through direct mapped cache has line (block) length of 16 32-bit words, and contains of 512 lines. State the total data storage size of the cache (not counting tag memory) in bytes, and the tag, index and select fields for this cache in a 32 bit ARM address.

[8]

c)

Write a fragment of ARM assembly code which implements in as few instructions as possible:

$R3 := R1 + R2 - R3 + 111$

[8]

d)

For each of the ARM instructions below, summarise the datapath transfer that occurs by giving one of the 4 options in Figure 1.1. In cases A & B only, specify what is the destination register. Note that in the ARM architecture PC is a data register.

- (i) **MOV R0,R1**
- (ii) **STRB R0, [R1]**
- (iii) **LDR R0, [R1,R2]**
- (iv) **B START**

[8]

e)

State single ARM instructions which perform the following integer operations on unsigned 32 bit operands, giving a 32 bit result, rounding down, and ignoring overflow.

- (i)  $R10 := R10 + R9 * 4096$
- (ii)  $R3 := R3 + R3 / 8$

[8]

|   |                                       |
|---|---------------------------------------|
| A | data register(s) => data register     |
| B | memory unit location => data register |
| C | data register => memory unit location |
| D | no datapath transfer                  |

*Figure 1.1. Datapath transfers*

2. Each code fragment (a) - (c) below executes with all condition codes and registers initially 0, and memory locations as in *Figure 2.1*. State the values of R0-R3, and the condition codes, after execution of the code fragment. Write your answers using as a template a copy of the table in *Figure 2.3*, deleting the example row labelled (x) which indicates the required format of your answer. Each answer must be written in hexadecimal except the condition codes, which must be in binary, as indicated in row (x).

a) Code as in *Figure 2.2a*.

[10]

b) Code as in *Figure 2.2b*.

[10]

c) Code as in *Figure 2.2c*. Note that code fragment execution terminates directly after the BGE instruction is condition false executed.

[10]

| Location (word) | Value     |
|-----------------|-----------|
| &100            | &04030201 |
| &104            | &08070605 |
| > &104          | &0        |

*Figure 2.1*. Memory locations

```
MOV R0, #1
MVN R1, R0
ADDS R2, R0, R1
SBC R3, R0, R1
ADC R0, R0, R0
```

(a)

```
MOV R10, #&100
LDR R0, [R10], #4
LDR R1, [R10], #4
ORR R2, R0, R1
ANDS R3, R0, R1
```

(b)

```
MOV R2, #3
MOV R1, #&100
L1 LDRB R0, [R1, #1]!
ORR R3, R0, R3, lsl #8
SUBS R2, R2, #1
BGE L1
```

(c)

*Figure 2.2*. Code fragments

|     | R0 | R1    | R2        | R3 | NZCV |
|-----|----|-------|-----------|----|------|
| (x) | &0 | &1020 | &FFFFFFFF | &C | 0110 |
| (a) |    |       |           |    |      |
| (b) |    |       |           |    |      |
| (c) |    |       |           |    |      |

*Figure 2.3*. Template for answers

3.

- a) Explain the precise operation of the **LDMED** & **STMED** instructions in the subroutine **PARITY** displayed in *Figure 3.1*. Indicate the function of R13, and how the transfers in these instructions implement the subroutine.

[6]

- b) The code in *Figure 3.2* which calls the **PARITY** subroutine is executed from **START** to **FINISH** (not including the unspecified instruction after label **FINISH**). Using the instruction timings in the exam notes, work out the total number of cycles used during this execution where:

- (i) Conditional branch **BEQ** is condition true executed.  
(ii) Conditional branch **BEQ** is condition false executed.

Explain, with reference the ARM hardware, the given timing for the **STMED** instruction.

[8]

- c) By computing the value of R2 LSB throughout execution of the **PARITY** subroutine, or otherwise, show how the Z status bit on subroutine exit relates to the bits of R1 on entry.

[8]

- d) The computation from **START** to **FINISH** can be optimised as follows:
- Use R1 to calculate the **PARITY** result, overwriting R1 in the process. Other than the change to R1 *the subroutine must have the same specification as before*.
  - Eliminate the conditional branch instruction in the calling code.

Rewrite the code in *Figure 3.1* and *Figure 3.2* with these changes in such a way that the value of R1 at **FINISH** remains the same as in the original code.

[8]

```

; R1: input
; Z status bit: output
; no register is changed.
PARITY  STMED   R13!, {R2,R14}
        MOV     R2, R1
A       EOR     R2, R2, R2, ror #1
        EOR     R2, R2, R2, ror #2
        EOR     R2, R2, R2, ror #4
B       ANDS    R2, R2, #1
        LDMED   R13!, {R2,R15}

```

*Figure 3.1.* **PARITY** subroutine

```

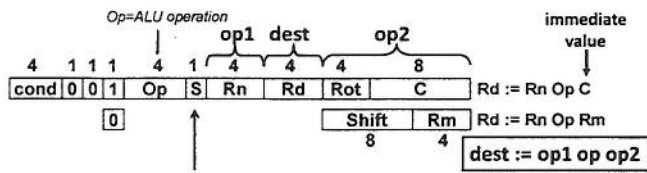
START   BL      PARITY
        BEQ     FINISH
        ORR     R1, R1, #&100
FINISH

```

*Figure 3.2.* Code calling the **PARITY** subroutine



### Data processing (ADD,SUB,AND,CMP,MOV, etc)



S bit = 1 => status bits are written  
S bit = 0 => status bits unchanged

The second operand, Op2, is either a constant C or register Rm

Assume Shift=0, Rot=0, for unshifted Rm or immediate C

|   |                      |
|---|----------------------|
| C | 1 => carry           |
| V | 1 => signed overflow |
| N | 1 => negative        |
| Z | 1 => zero            |

| Op-codes |                           |
|----------|---------------------------|
| AND      |                           |
| ANDEQ    | EQ,NE,... => Condition    |
| ANDS     | S => set status on result |
| ANDEQS   | note position of S        |

| Op   | Assembly      | Operation           | Pseudocode           |
|------|---------------|---------------------|----------------------|
| 0000 | AND Rd,Rn,op2 | Bitwise logical AND | Rd := Rn AND op2     |
| 0001 | EOR Rd,Rn,op2 | Bitwise logical XOR | Rd := Rn XOR op2     |
| 0010 | SUB Rd,Rn,op2 | Subtract            | Rd := Rn - op2       |
| 0011 | RSB Rd,Rn,op2 | Reverse subtract    | Rd := op2 - Rn       |
| 0100 | ADD Rd,Rn,op2 | Add                 | Rd := Rn + op2       |
| 0101 | ADC Rd,Rn,op2 | Add with carry      | Rd := Rn + op2 + C   |
| 0110 | SBC Rd,Rn,op2 | Subtract with carry | Rd := Rn - op2 - C   |
| 0111 | RSC Rd,Rn,op2 | Reverse sub with C  | Rd := op2 - Rn - C   |
| 1000 | TST Rn,op2    | set NZ on AND       | Rn AND op2           |
| 1001 | TEQ Rn,op2    | set NZ on EOR       | Rn EOR op2           |
| 1010 | CMP Rn,op2    | set NZCV on -       | Rn - op2             |
| 1011 | CMN Rn,op2    | set NZCV on +       | Rn + op2             |
| 1100 | ORR Rd,Rn,op2 | Bitwise logical OR  | Rd := Rn OR op2      |
| 1101 | MOV Rd,op2    | Move                | Rd := op2            |
| 1110 | BIC Rd,Rn,op2 | Bitwise clear       | Rd := Rn AND NOT op2 |
| 1111 | MVN Rd,op2    | Bitwise move invert | Rd := NOT op2        |

### Data Processing Op2

ADD r0, r1, op2  
MOV r0, op2

#### Examples

ADD r0, r1, r2  
MOV r0, #1  
CMP r0, #1  
EOR r0, r1, r2, lsr #10  
RSB r0, r1, r2, asr #3

| Op2          | Conditions                                     | Notes                                                                                                                 |
|--------------|------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| Rm           |                                                | r15=pc, r14=lr, r13=sp                                                                                                |
| #imm         | imm = s rotate 2r<br>(0 ≤ s ≤ 255, 0 ≤ r ≤ 15) | Assembler will translate negative values changing op-code as necessary<br>Assembler will work out rotate if it exists |
| Rm, shift #s | (1 ≤ s ≤ 31)                                   | rrx always writes carry<br>ror writes carry if S=1<br>shifts do not write carry                                       |
| Rm, rrx #1   | shift => lsr,lsr,asr,asr,ror                   |                                                                                                                       |
| Rm, shift Rs | shift => lsr,lsr,asr,asr,ror                   | shift by register value (takes 2 cycles)                                                                              |

### Multiply in detail

- MUL,MLA were the original (32 bit LSW result) instructions
  - Why does it not matter whether they are signed or unsigned?
- Later architectures added 64 bit results

Register operands only  
No constants, no shifts

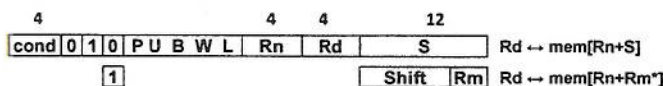
NB d & m must be different for MUL, MLA

#### ARM3 and above

|                      |                       |                            |
|----------------------|-----------------------|----------------------------|
| MUL rd, rm, rs       | multiply (32 bit)     | Rd := (Rm*Rs)[31:0]        |
| MLA rd, rm, rs, rn   | multiply-acc (32 bit) | Rd := (Rm*Rs)[31:0] + Rn   |
| UMULL rh, rl, rm, rs | unsigned multiply     | (Rh:Rl) := Rm*Rs           |
| UMLAL rh, rl, rm, rs | unsigned multiply-acc | (Rh:Rl) := (Rh:Rl) + Rm*Rs |
| SMULL rh, rl, rm, rs | signed multiply       | (Rh:Rl) := Rm*Rs           |
| SMLAL rh, rl, rm, rs | signed multiply-acc   | (Rh:Rl) := (Rh:Rl) + Rm*Rs |

ARM7DM core and above (64 bit multiply result)

### Data transfer (to or from memory LDR,STR)



| Bit in word | 0                                 | 1                                                   |
|-------------|-----------------------------------|-----------------------------------------------------|
| P           | use base register addressing [Rn] | use indexed or offset address [Rn+Rm], [Rn+S]       |
| U           | subtract offset [Rn-S]            | add offset [Rn+S]                                   |
| B           | Word                              | Byte                                                |
| W           | leave Rn unchanged if P=1         | write indexed or offset address back into Rn if P=1 |
| L           | Store                             | Load                                                |

NB - P=0, W=1 is not allowed

If P=0, W=0, write offset address back into Rn. For no change to Rn & no offset use P=0,W=0,S=0

### Data Transfer Instructions

|        |                            |
|--------|----------------------------|
| LDR    | load word                  |
| STR    | store word                 |
| LDRB   | load byte                  |
| STRB   | store byte                 |
| LDREQB | NB B is after EQ condition |
| STREQB |                            |

|     |                          |                                                              |
|-----|--------------------------|--------------------------------------------------------------|
| LDR | r0, [r1]                 | ; register-indirect addressing                               |
| LDR | r0, [r1, #offset]        | ; pre-indexed addressing (base + offset)                     |
| LDR | r0, [r1, #offset]!       | ; pre-indexed, auto-indexing (base + offset + writeback)     |
| LDR | r0, [r1], #offset        | ; post-indexed, auto-indexing (change Rn after)              |
| LDR | r0, [r1, r2]             | ; register-indexed addressing (base + reg)                   |
| LDR | r0, [r1, r2, lsr #shift] | ; scaled register-indexed addressing (base + reg * 2^shift)  |
| LDR | r0, address_label        | ; PC relative addressing (pc+8 is read, offset calculated)   |
| ADR | r0, address_label        | ; load PC relative address (pc+8 is read, offset calculated) |

LDMED r13!, {r0-r4,r6,r6}; !=> write-back to address register  
 STMFA r13, {r2}; no write-back  
 STMEQIB r2!, {r5-r12}; note position of EQ or other condition  
 higher reg nos go to/from higher mem addresses  
 [E|F][A|D] Empty|Full, Ascending|Descending  
 [I|D][A|B] Increment|Decrement, After|Before

| Name                 | Stack | Other |
|----------------------|-------|-------|
| pre-increment load   | LDMED | LDMIB |
| post-increment load  | LDMFD | LDMIA |
| pre-decrement load   | LDMEA | LDMDB |
| post-decrement load  | LDMFA | LDMDA |
| pre-increment store  | STMFA | STMIB |
| post-increment store | STMEA | STMIA |
| pre-decrement store  | STMPD | STMDB |
| post-decrement store | STMED | STMDA |

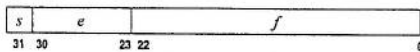
## Instruction Timing

Exact instruction timing is very complex and depends in general on memory cycle times which are system dependent. The table below gives an approximate guide.

| Instruction                                                     | Typical execution time (cycles) |
|-----------------------------------------------------------------|---------------------------------|
| Any instruction, with condition false                           | 1                               |
| data processing (except register-valued shifts)                 | 1 (+3 if Rd = R15)              |
| data processing (register-valued shifts):<br>MOV R1, R2, lsl R3 | 2 (+3 if  Rd  = R15)            |
| LDR, LDRB, STR, STRB                                            | 4 (+3 more if Rd = R15)         |
| LDM (n registers)                                               | n+3 (+3 more if Rd = R15)       |
| STM (n registers)                                               | n+3                             |
| B, BL                                                           | 4                               |
| Multiply                                                        | 7-14                            |

$$x = (-1)^s 2^{e-127} f$$

## IEEE 754



## Shadow registers

- FIQ mode: R8 - R14 shadowed
- IRQ, SVC, abort, UND modes: R13 - R14 shadowed
- Return from interrupt: set status bits to restore CPSR, so use SUBS to set PC equal to stored return address with offset & restore CPSR.
- R13 is SP

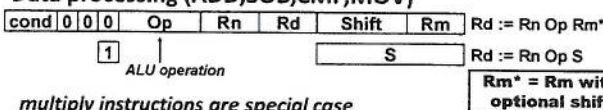
|               |                          |
|---------------|--------------------------|
| BL, SWI       | R14 = Return address     |
| IRQ, FIQ, UND | R14 = Return address + 4 |
| Abort         | R14 = Return address + 8 |

| Exception                                       | Mode  | Vector address |
|-------------------------------------------------|-------|----------------|
| Reset                                           | SVC   | 0x00000000     |
| Undefined instruction                           | UND   | 0x00000004     |
| Software interrupt (SWI)                        | SVC   | 0x00000008     |
| Prefetch abort (instruction fetch memory fault) | Abort | 0x0000000C     |
| Data abort (data access memory fault)           | Abort | 0x00000010     |
| IRQ (normal interrupt)                          | IRQ   | 0x00000018     |
| FIQ (fast interrupt)                            | FIQ   | 0x0000001C     |

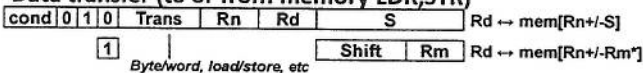
| Cond |       | Condition                 | Status Bits                |
|------|-------|---------------------------|----------------------------|
| 0000 | EQ    | Equal                     | Z set                      |
| 0001 | NE    | Not equal                 | Z clear                    |
| 0010 | CS/HS | Unsigned ≥ (High or Same) | C set                      |
| 0011 | CC/LO | Unsigned < (Low)          | C clear                    |
| 0100 | MI    | Minus (negative)          | N set                      |
| 0101 | PL    | Plus (positive or 0)      | N clear                    |
| 0110 | VS    | Signed overflow           | V set                      |
| 0111 | VC    | No signed overflow        | V clear                    |
| 1000 | HI    | Unsigned > (High)         | C set and Z clear          |
| 1001 | LS    | Unsigned ≤ (Low or Same)  | C clear OR Z set           |
| 1010 | GE    | Signed ≥                  | N equals V                 |
| 1011 | LT    | Signed <                  | N is not equal to V        |
| 1100 | GT    | Signed >                  | Z clear and N equals V     |
| 1101 | LE    | Signed ≤                  | Z set and N not equal to V |
| 1110 | AL    | Always                    | any                        |
| 1111 | NV    | Never (do not use)        | none                       |

## Machine Instruction Overview (1)

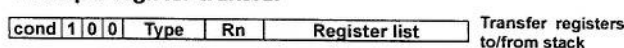
### Data processing (ADD, SUB, CMP, MOV)



### Data transfer (to or from memory LDR, STR)

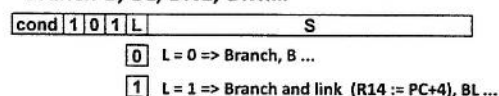


### Multiple register transfer

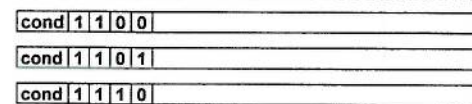


## Overview (2)

### Branch B, BL, BNE, BMI...

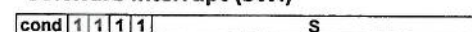


PC := PC+8+4\*S  
 S is sign extended  
 NB +8 because of  
 pipelining.



coprocessor  
 interface

### Software Interrupt (SWI)



Simulate hardware  
 interrupt: S is  
 passed to handler  
 in swi mode

## Introduction to Computer Architecture - Answers 2011

All questions are compulsory, marks are 40/30/30 (or 20/15/15 on whole paper) for Q1/Q2/Q3.

Questions will be slightly easier than in previous papers, due to lack of choice.

### Answer to Question 1

Q1 is an easy question testing basic knowledge.

1.

a)

- (i) &C1FA0000
- (ii)  $N = 8 \Rightarrow -1, N = 16 \Rightarrow 255$

[8]

b)

size =  $16 * 4 * 512 = 32768$  bytes  
 select =  $\log_2(16 * 4) = 6$  (includes LS 2 bits byte select) A5:0  
 index = 9 bits: A14:6  
 tag = A31:15

[8]

c)

**SUB R3, R1, R3**  
**ADD R3, R3, R2**  
**ADD R3, R3, #111**

**Many other equivalents. Note 1<sup>st</sup> line must include R3**

[8]

d)

- (i) A (R0)
- (ii) C
- (iii) B (R0)
- (iv) B

[8]

(e)

ADD R10, R10, R9, lsl #12  
 ADD R3, R3, R3, lsr #3

[8]



## Answer to Question 2

*This question tests ability to understand and analyse operation of ARM assembly code in detail. It requires accuracy and comprehensive understanding of the instructions, but is straightforward.*

- (a) tests understanding of two's complement arithmetic, and condition codes
- (b) test understanding of memory addressing modes & logical operations
- (c) tests understanding of shift, byte transfer, addressing modes & simple loops

| Location | Value     |
|----------|-----------|
| &100     | &04030201 |
| &104     | &08070605 |
| > &10C   | &0        |

Figure 2.1. Memory locations

```
MOV R0, #1
MVN R1, R0
ADDs R2, R0, R1
SBC R3, R0, R1
ADC R0, R0, R0
```

(a)

```
MOV R10, &100
LDR R0, [R10], #4
LDR R1, [R10], #4
ORR R2, R0, R1
ANDs R3, R0, R1
```

(b)

```
MOV R2, #3
MOV R1, #&100
L1 LDRB R0, [R1, #1]!
ORR R3, R0, R3, lsl #8
SUBs R2, R2, #1
BGE L1
```

(c)

Figure 2.2. Code fragments

2 marks each box

|     | R0       | R1       | R2       | R3       | NZCV |
|-----|----------|----------|----------|----------|------|
| (a) | 2        | FFFFFFFE | FFFFFFF  | 2        | 1000 |
| (b) | 04030201 | 08070605 | 08070605 | 00030201 | 0000 |
| (c) | 8        | 104      | FFFFFFF  | 03020108 | 1000 |

Figure 2.3. Template for answers

**Answer to Question 3**

*This more difficult question tests knowledge of the ARM subroutine implementation, as well as instruction timing, and ability to manipulate code.*

a)

STM/LDM instructions save/restore data on a downwards growing empty stack pointer stack. R13 is the stack pointer. In this case R2,R14 are saved, R2,R15 are restored - the restore operation thus causes the subroutine return by loading PC with the return address previously in R14.

[6]

b)

(i)  $4+5+1+1+1+1+1+(5+3)+4 = 26$  cycles

(ii) 24 cycles

STMED takes longer because:

- It is a random access memory operation which slows down the memory unit (precise reason here not needed).
- It has destination PC which breaks the prefetch/decode pipeline.

[8]

c)

R2(0) has values: R1(0)->xor R1(1:0)->xor R1(3:0) -> xor R1(7:0)

Z=1 iff R2(0)=0 => xor of R1(7:0) is 0.

[8]

d)

**PARITY**

; note MOV & LDM are not needed

**A**      EOR      R1, R1, R1, ror #1

EOR      R1, R1, R1, ror #2

EOR      R1, R1, R1, ror #4

**B**      ANDS    R1, R1, #1

MOV      R15, R14 ; note MOV not LDM

**START**    MOV      R2,R1

BL      PARITY

ORRNE    R1, R2, #&100 ; note optimisation

**FINISH**

[8]