Final

Paper Number(s): **E1.9A**

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2009

ISE Part I: MEng, BEng and ACGI

Corrected Copy

**INTRODUCTION TO COMPUTER ARCHITECTURE AND SYSTEMS (PART A)**

Monday, 1 June 2:00 pm

Time allowed: 1:30 hours

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

**Question 1 is compulsory and carries 40% of the marks.**

**Answer Question 1 and two others from Questions 2-4 which 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): Demiris, Y.

**Special information for invigilators:**

*The booklet Exam Notes 2009 should be distributed with the Examination Paper.*

**Information for candidates:**

*The prefix &, or suffix $_{(16)}$, introduces a hexadecimal number, e.g: &1C0, $1C0_{(16)}$.*

*The booklet Exam Notes 2009, as published on the course web pages, is provided and contains reference material.*

*Question 1 is compulsory and carries 40% of marks. Answer only TWO of the Questions 2-4, which carry equal marks.*

# The Questions

1. **[Compulsory]**

(a) Perform the following numeric conversions:

(i) $-251_{(10)}$ into 9 bit signed octal

(ii) "abcA" into 32 bit hexadecimal ASCII codes with the first character stored in the least significant byte.

(iii) $-22.25_{(10)}$ into IEEE-754 floating point. Give your answer in *hexadecimal*.

[8]

(b) A CPU with 60 MHz clock frequency executes one instruction per cycle using a pipeline of length of 5, and has a pipeline stall every 10 cycles. Calculate the average throughput of the CPU in millions of instructions per second (MIPS), assuming stall time = pipeline length. What would be the throughput of this CPU with no pipeline?

[8]

(c) State the value in *decimal* of registers R0-R5 at the end of the ARM assembly code fragment in *Figure 1.1* assuming that at the start of the code fragment R$n$ = $n$ ($n$ = 0, 1,....14).

[12]

(d)

(i) State the unsigned and signed number ranges for an $n$ bit binary number.

(ii) State in the ARM architecture what is the Boolean expression on condition codes **N,Z,C,V** for unsigned arithmetic overflow.

(iii) State in the ARM architecture what is the Boolean expression on condition codes **N,Z,C,V** that will implement the condition R0 > R1 after instruction **CMP R0,R1** assuming R0, R1 are signed.

[12]

```
RSBS    R0, R7, R6
ADD     R1, R6, R7, lsr #1
EOR     R2, R7, R8
SBC     R3, R3, R3
BIC     R4, R10, #3
MOV     R5, R12, lsl #10
```

*Figure 1.1*

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 value of R0-R3, the condition codes, and any *changed* memory locations, after execution of the code fragment. Write your answers using as a template a copy of the table in *Figure 2.3* omitting the 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, this format illustrated 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*. [10]

| Location | Value |
|---|---|
| &100 | &11121314 |
| &104 | &10203040 |
| &108 | &01020304 |
| &10C | &80706050 |
| > &10C | &0 |

*Figure 2.1* - memory locations

```
MOV     R10, #&100
MOV     R11, #4
LDR     R0, [R10,R11]
LDR     R1, [R10,#8]!
LDRB    R2, [R10],#1
LDRB    R3, [R10]
STRB    R10, [R11,R11]
```

(a)

```
MOV     R0, #&108
MOV     R1, #&200
LDMDA   R0!, {R2,R3}
STMIB   R1, {R2,R3}
```

(b)

```
MOV     R0, #1
MOV     R1, R0, rol #10
EORS    R2, R1, R0, ror #1
ADC     R3, R0, R0
SUBS    R0, R0, R0
```

(c)

*Figure 2.2* - code fragments

| | R0 | R1 | R2 | R3 | NZCV | Memory |
|---|---|---|---|---|---|---|
| (x) | 0 | &1020 | &FFFFFFFF | &C | 0110 | $mem_8$[&120] = &10<br>$mem_{32}$[&300]=&FFFF0000 |
| (a) | | | | | | |
| (b) | | | | | | |
| (c) | | | | | | |

*Figure 2.3* - template for answers

3. The ARM code in *Figure 3.1* sets R1 to a value which depends on R2, R3, R4.

(a) Give code examples from the execution of this code to show how an ARM instruction that enters the FETCH stage of the pipeline may be either not executed, condition-true executed, or condition-false executed.

[6]

(b) If initially R1 = *x1*, R2 = *x2*, R3 = *x3*, R4 = *x4,* state concisely, using pseudo-code with one or more if-then-else statements, what is the final value to which R1 is set.

[8]

(c) If R2,R3,R4 are initially 0 trace through the execution of the ARM7 code in *Figure 3.1* illustrating in a diagram the instructions occupying each stage of the pipeline in every cycle.

[8]

(d) State what are the quickest and slowest paths through the code in *Figure 3.1* when executed on the ARM7, giving in each case the code execution time in machine cycles. Instruction timing may be found in the Exam Notes booklet.

[8]

```
A     CMP R2, #0
B     ADDGE R1, R1, R2
C     SUBLT R1, R1, R2
D     BEQ X
E     ADD R1, R1, R3
F     B Y
X     ADD R1, R1, R4
Y
```

*Figure 3.1*

4. This question relates to the ARM assembler code fragment LOOP in *Figure 4.1*.

(a) If R0 has non-negative value $n$ at the start of LOOP, calculate as a function of $n$ the number of iterations of LOOP. Discuss what happens if $n$ is negative.

[6]

(b) The instructions with labels C & D test ARM condition codes. State in each case which instruction sets the condition codes which are tested, and therefore what is the condition on data in R1 for each of these instructions to be condition-true executed.

[8]

(c) Suppose that just before the instruction with label A is executed, the bits of R1 & R3 are denoted X(31:0) and Y(31:0). Using these bit designations, determine the value of each bit of R3 just after the instruction with label D is executed.

[8]

(d) At the start of LOOP, R5 = $p$, R6 = $q$. Determine the addresses of all memory locations loaded and stored in the $i$th iteration of LOOP, where $i$ ranges from 1 upwards, as a function of $i$, $p$ and $q$. Hence state concisely what is the change in memory locations made by this code when it is executed with R0 = $n$.

[8]

```
LOOP
        LDR     R1, [R5],#4
        LDR     R3, [R6]
A       BIC     R3, R3, #&80000003
B       ANDS    R4, R1, #&80000001
C       EORMI   R3, R3, R4
D       ORRNE   R3, R3, #2
        STR     R3, [R6], #4
        SUBS R0, R0, #1
        BGE LOOP
```

*Figure 4.1*

E2.19/E1.9A

# EXAM NOTES 2009

## Introduction to Computer Architecture (EE2)

## Introduction to Computer Architecture and Systems (ISE)

# Memory Reference & Transfer Instructions

```
LDR     load word
STR     store word
LDRB    load byte
STRB    store byte

LDREQB ; note position
; of EQ
STREQB
```

```
LDMED r13!,{r0-r4,r6,r6} ; ! => write-back to register
STMFA r13, {r2}
STMEQIB r2!,{r5-r12} ; note position of EQ
; higher reg nos go to/from higher mem addresses always
[E|F][A|D] empty/full, ascending/descending
[I|D][A|B]  incr/decr,after/before
```

```
LDR     r0, [r1]                ; register-indirect addressing
LDR     r0, [r1, #offset]       ; pre-indexed addressing
LDR     r0, [r1, #offset]!      ; pre-indexed, auto-indexing
LDR     r0, [r1], #offset       ; post-indexed, auto-indexing
LDR     r0, [r1, r2]            ; register-indexed addressing
LDR     r0, [r1, r2, lsl #shift] ; scaled register-indexed addressing
LDR     r0, address_label       ; PC relative addressing
ADR     r0, address_label       ; load PC relative address
```



# Conditions Binary Encoding

| Opcode [31:28] | Mnemonic extension | Interpretation | Status flag state for execution |
|---|---|---|---|
| 0000 | EQ | Equal / equals zero | Z set |
| 0001 | NE | Not equal | Z clear |
| 0010 | CS/HS | Carry set / unsigned higher or same | C set |
| 0011 | CC/LO | Carry clear / unsigned lower | C clear |
| 0100 | MI | Minus / negative | N set |
| 0101 | PL | Plus / positive or zero | N clear |
| 0110 | VS | Overflow | V set |
| 0111 | VC | No overflow | V clear |
| 1000 | HI | Unsigned higher | C set and Z clear |
| 1001 | LS | Unsigned lower or same | C clear or Z set |
| 1010 | GE | Signed greater than or equal | N equals V |
| 1011 | LT | Signed less than | N is not equal to V |
| 1100 | GT | Signed greater than | Z clear and N equals V |
| 1101 | LE | Signed less than or equal | Z set or N is not equal to V |
| 1110 | AL | Always | any |
| 1111 | NV | Never (do not use!) | none |



# ARM Data Processing Instructions Binary Encoding

| Opcode [24:21] | Mnemonic | Meaning | Effect |
|---|---|---|---|
| 0000 | AND | Logical bit-wise AND | Rd := Rn AND Op2 |
| 0001 | EOR | Logical bit-wise exclusive OR | Rd := Rn EOR Op2 |
| 0010 | SUB | Subtract | Rd := Rn - Op2 |
| 0011 | RSB | Reverse subtract | Rd := Op2 - Rn |
| 0100 | ADD | Add | Rd := Rn + Op2 |
| 0101 | ADC | Add with carry | Rd := Rn + Op2 + C |
| 0110 | SBC | Subtract with carry | Rd := Rn - Op2 + C - 1 |
| 0111 | RSC | Reverse subtract with carry | Rd := Op2 - Rn + C - 1 |
| 1000 | TST | Test | Scc on Rn AND Op2 |
| 1001 | TEQ | Test equivalence | Scc on Rn EOR Op2 |
| 1010 | CMP | Compare | Scc on Rn - Op2 |
| 1011 | CMN | Compare negated | Scc on Rn + Op2 |
| 1100 | ORR | Logical bit-wise OR | Rd := Rn OR Op2 |
| 1101 | MOV | Move | Rd := Op2 |
| 1110 | BIC | Bit clear | Rd := Rn AND NOT Op2 |
| 1111 | MVN | Move negated | Rd := NOT Op2 |

**Op-codes**

**AND**
**ANDEQ**
**ANDS**
**ANDEQS**

**S=> set flags**



# Data Processing Operand 2

**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 r3
```

| Op2 | Conditions | Notes |
|---|---|---|
| **Rm** | | |
| **#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**<br>**Rm, rrx #1** | (1 ≤ s ≤ 31)<br>shift => lsr,lsl,asr,asl,ror | rrx always sets carry<br>ror sets carry if S=1<br>shifts do not set carry |
| **Rm, shift Rs** | shift => lsr,lsl,asr,asl,ror | shift by register value (takes 2 cycles) |

## Multiply Instructions

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

- Note that some multiply instructions have 4 register operands!
  - + Multiply instructions must have **register** operands, **no immediate constant**
  - + Multiplication by small constants can often be implemented more efficiently with data processing instructions – see Lecture 10.

**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 rl, rh, rm, rs** | **unsigned multiply** | **(Rh:Rl) := Rm*Rs** |
| **UMLAL rl, rh, rm, rs** | **unsigned multiply-acc** | **(Rh:Rl) := (Rh:Rl)+Rm*Rs** |
| **SMULL rl,rh,rm,rs** | **signed multiply** | **(Rh:Rl) := Rm*Rs** |
| **SMLAL rl,rh,rm,rs** | **signed multiply-acc** | **(Rh:Rl) :=(Rh:Rl)+Rm*Rs** |

**ARM7DM core and above**



## Exceptions & Interrupts

| Exception | Return |
|---|---|
| SWI or undefined instruction | MOVS pc, R14 |
| IRQ, FIQ, prefetch abort | SUBS pc, r14, #4 |
| Data abort (needs to rerun failed instruction) | SUBS pc, R14, #8 |

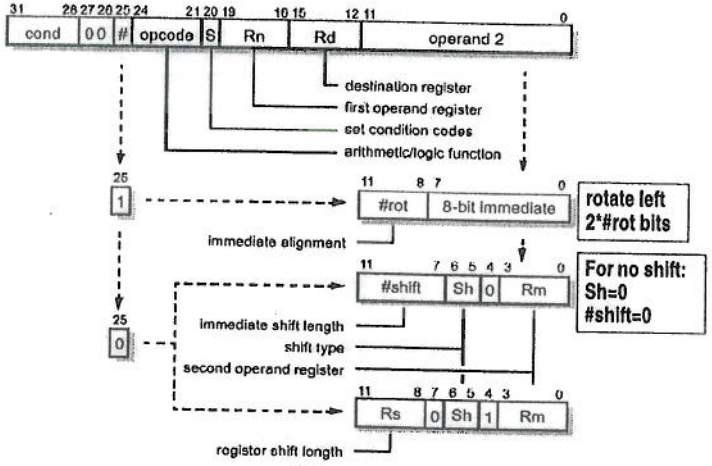
| Exception Mode | Shadow registers |
|---|---|
| SVC,UND,IRQ,Abort | R13, R14, SPSR |
| FIQ | as above + R8-R12 |

*(0x introduces a hex constant)*

| 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 |

R2.6

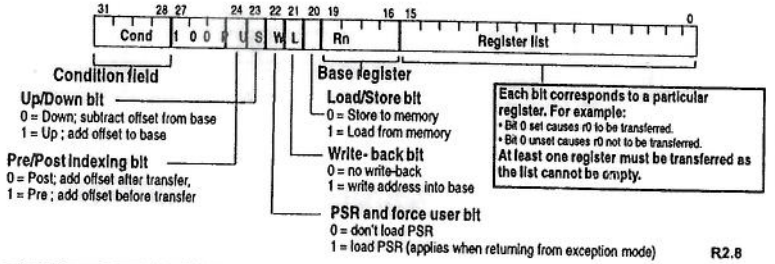
## Data Processing Instruction Binary Encoding




R2.7

## Multiple Register Transfer Binary Encoding

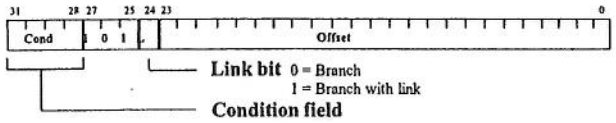
- The Load and Store Multiple instructions (LDM / STM) allow betweeen 1 and 16 registers to be transferred to or from memory.




R2.8

# Branch Instruction Binary Encoding

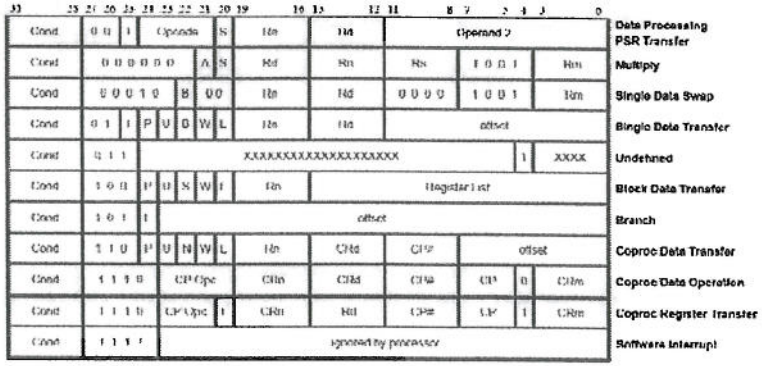
- Branch : `B{<cond>} label`
- Branch with Link : `BL{<cond>} sub_routine_label`




- The offset for branch instructions is calculated by the assembler:
  - + By taking the difference between the branch instruction and the target address minus 8 (to allow for the pipeline).
  - + This gives a 26 bit offset which is right shifted 2 bits (as the bottom two bits are always zero as instructions are word – aligned) and stored into the instruction encoding.
  - + This gives a range of ± 32 Mbytes.



# Instruction Set Overview




# ARM 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) *(If instruction condition is TRUE – otherwise 1 cycle)* |
|---|---|
| Any instruction, with condition false | 1 |
| data processing (all except when shift is by number equal to value of register) | 1 |
| data processing (register-valued shifts) | 2 |
| LDR,LDRB | 4 |
| STR,STRB | 4 |
| LDM (n registers) | n+3 (+3 if PC is loaded) |
| STM (n registers) | n+3 |
| B, BL | 4 |
| Multiply | 7-14 (varies with architecture & operand values) |

# ASCII Codes

NB – b7 = 0

b(3:0) / b(6:4)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | BS | HT | LF | VT | FF | CR | SO | SI |
| 1 | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS | RS | US |
| 2 | SPC | ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / |
| 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 4 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 5 | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| 6 | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 7 | p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ~ | DEL |

Paper Number(s): **E1.9B**

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2009

ISE Part I: MEng, BEng and ACGI

**INTRODUCTION TO COMPUTER ARCHITECTURE AND SYSTEMS (PART B)**

**OPERATING SYSTEMS**

Monday, 1 June 3.30 pm

Time allowed: 1:00 hour

Corrected Copy

Q1

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

**Answer ONE question.**

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

Examiners responsible:

First Marker(s): Demiris, Y.K.
Second Marker(s): Bouganis, C.

## The Questions
## Answer ONLY ONE of the following two questions

1.

(a) Describe the Shortest-Job-First (SJF) and Shortest-Remaining-Job-First (SRJF) process scheduling algorithms, describe their differences, and list the advantages and disadvantages of each algorithm. [3]

(b) Consider the following set of processes, with their corresponding duration and arrival times:

| Process | Arrival time (ms) | Duration (ms) |
|---|---|---|
| A | 0 | 3 |
| B | 3 | 4 |
| C | 4 | 6 |
| D | 5 | 3 |
| E | 6 | 2 |

Show the order of execution (including timing information) of the processes if the scheduler implements the scheduling algorithms below. For each of the algorithms calculate the average waiting time, and the average turnaround time.

(i) Shortest job first (SJF) [3]
(ii) RR (round robin) with a time quantum of 3ms [3]
(iii) RR with a time quantum of 1ms [3]

(c) In the "readers-writers" problem, a set of processes are accessing a block of shared data (e.g. a library catalogue): the reader processes only read shared data, while writer processes only write shared data. The following conditions are in place:

- A writer process can write items in the shared data block only if there are no other writer processes that are accessing the block.
- A reader process can read the shared data only if no writer is accessing them; more than one reader can read the shared data at the same time.
- Readers have priority: once a single reader has started accessing the shared data, readers can retain control of the shared data block as long as there is at least one reader in the act of reading.

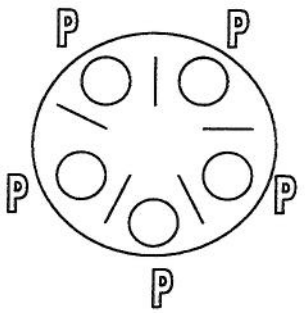
Using semaphores to ensure that the conditions above hold, provide Pascal procedures for the reader and writer processes. Declare and properly initialise all semaphores and other variables that you will use. The data type *Semaphore*, and the standard semaphore primitives *init(Sem, number), wait(Sem),* and *signal(Sem)* are available. You may assume that the following procedures are also available: produce_item, write_item, read_item, consume_item. [5]

(d) In the context of memory management, describe the three options as to when address binding can occur, and describe their relative advantages & disadvantages. [3]

2.

(a) In the context of page replacement algorithms, describe the Optimal Page Replacement and Least-Recently-Used (LRU) page replacement algorithms, and list their advantages and disadvantages. [2]
Subsequently, describe how you can implement the LRU algorithm using
(i) counters [2]
(ii) a stack [2]

(b) In the "dining philosophers" synchronization problem, five philosophers (P) need to eat, with only 5 chopsticks available to them, arranged as shown in the figure below. Eating requires two chopsticks; a philosopher can only pick one chopstick at a time, and only chopsticks located next to him/her.




Describe how can you ensure that a deadlock will not occur in this situation, and explain why your solution guarantees that. [3]

(c) In the context of the four necessary conditions for a deadlock to occur, describe how we can ensure that the "hold and wait" condition cannot occur, and describe the resulting disadvantages. [4]

(d) In the context of a memory paging system, consider the following scenario:

- You have three available frames
- The reference string is 6-1-1-4-5-2-4-5-6-2

Starting with empty frame contents, show the sequence of frame contents after each request, and count the number of page faults for each of the following page replacement algorithms:

(i) Optimal-page replacement [2]
(ii) First in First Out replacement [2]
(iii) LRU (Least Recently Used) page replacement [2]

(e) Specify the difference between a weak and a strong semaphore. [1]

*A=analysis, D=design, C=calculated solution using taught theory, B=bookwork*

***NB - marking will be 1 mark for every 2% indicated on question paper (max 50 marks) so marks here are half of marks on question paper.***

## Solution to Question 1

**36 minutes for the question => 9 minutes each part**

(a)
i) $405_{(8)}$
ii) $61626341_{(16)}$ (NB assume little-endian so LSB is bits (7:0) of the word)
iii) $C1B20000_{(16)}$ s=1, e=131, m=(1.)011001
**[1 mark each, except iii 2 marks]**

[4C]

(b)
T=60M/(1+0.1*5)=40MIPS. With no pipeline there is no stall, but stages happen sequenctially so 60M/5 = 12MIPS

[4C]

(c)
**R0= -1,**
**R1= 6+7/2= 9,**
**R2= 15,**
**R3= -1 (no carry from RSBS),**
**R4= 8,**
**R5= 12*1024 = 12288**

[6C]

d)
(i) $0\text{-}2^n\text{-}1$ (unsigned), $-2^{n-1}$ - $2^{n-1}$-1 (signed)
(ii) C
(iii) either result is positive & no overflow, or result is negative & overflow:
(!N).!V + N.V).!(Z) or
!(N⊕V).!Z etc

[6A]

## Solution to Question 2

**27 minutes for the question**

*This tests ability to understand low-level operation of ARM assembler instructions*

**For each part, deduct 1 mark for each column wrong down to minimum of 0 marks. Assume mem[] = $mem_{32}$[].**

| | r0 | r1 | r2 | r3 | NZCV | Memory |
|---|---|---|---|---|---|---|
| a) | &10203040 | &01020304 | &04 | &03 | n/a | $mem_8$[&8] = &108 |
| b) | &100 | &200 | &10203040 | &01020304 | n/a | $mem_{32}$[&204] = &10203040<br>$mem_{32}$[&208] = &01020304 |
| c) | &0 | &400 | &80000400 | &2 | 0110 | n/a |

Note ;
consequent errors allowed (e.g. data wrong in memory write)
In b) r2/r3 swapped => 1 mark

[5A+5A+5A]

## Solution to Question 3

27 minutes for the question

*This questions tests understanding of instruction execution in the ARM architecture.*

(a)

Not executed but FETCHED: instruction X if F is executed
condition-true executed: A
condition-false executed: B if R2 < 0 (signed)

[3B]

b)

if x2 > 0 (signed) then R1 := x1+x2+x3 else
if x2 < 0 (signed) R1 := x1 - x2 +x3
if x2 = 0 then R1 := x1+x2+ x4

[4A]

c)

| FETCH | A | B | C | D | E | F | wait | X | Y | ... |
|---|---|---|---|---|---|---|---|---|---|---|
| DECODE | | A | B | C | D | E | stall | stall | X | Y |
| EXECUTE | | | A | B | C | D | stall | stall | stall | X |

[4A/B]

d)

quickest: A,B,C,D,X (8 cycles)
slowest: A,B,C,D,E,F (9 cycles)

[4A]

## Solution to Question 4

*This question tests whether the student understands the ARM bit manipulation & conditional execution.*

**27 minutes for the question**

a)
n + 1 iterations (loop for values of n down to and including 0).If n is negative it will loop once.

[3A]

b) Instruction at B sets codes for both C & D. C & V are not set, N & Z are set based on the value of R3 AND &80000001 (bitwise AND).
Hence C is executed if R1(31) is 1<=> R1 negative, D if R1(31) or R1(0) is 1.

[4A]

c) There are three bits of R3 which can change: 31, 1 and 0.
Instrictions A, C & D may change these as follows

| | 31 | 1 | 0 | |
|---|---|---|---|---|
| A | 0 | 0 | 0 | |
| C | R1(31) | 0 | R1(31).R1(0) | *change if R1(31)=true* |
| D | R1(31) | R1(31)+R1(0) | R1(31).R1(0) | *change if R1(31) or R1(0)=true* |

Hence
R3(31) = R1(31)
R3(1) = R1(31)+R1(0)
R3(0) = R1(31).R1(0)
All other bits of R3 stay the same.
(NB R1=X, R3=Y)

[4A]

d)
In the first iteration the addresses are:
R1 load: p
R3 load: q
R3 store: q
Each iteration these both advance by 4, so we have
R1 load p+4(i-1)
R3 load and store q+4(i-1)
this continues for n+1 iterations i= 1 to n+1

Hence n+1 words [q, q+4,...,q+4n] have bits 31,1,0 modified as per part (c) by the corresponding bits of corresponding words in [p, p+4,...,p+4n]. If the two memory areas overlap with q > p the changes to q affect subsequent p locations.

[4A]

# E1.9 – section B: Operating Systems
# Sample model answers to exam questions 2009

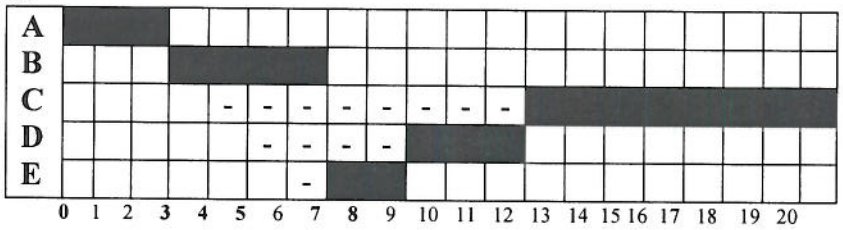
***Question 1***
(a) [bookwork]

SJF Algorithm description: CPU is allocated to the process that has the shortest next CPU burst
Advantages: Provably optimal in that it gives the minimum average waiting time for a given set of processes
Disadvantages: Knowing the length of next CPU burst is difficult; frequently this is predicted utilising previous lengths as estimates, or is user-specified. Also, it can result in long waits for long processes

SRJF Algorithm description: Variation of SJF, *adding preemption*; CPU again is allocated to the process that has the shortest next CPU burst; if a *new* process comes along that has a shorter CPU burst than the remaining one in the running process, the running process is removed and the new process is allocated the CPU.
Advantages: Allows new short jobs to get a good service; Good handling of interactive processes since it results in a short response time
Disadvantages: Same as SJF's, plus, this algorithm can be particularly bad for long processes (leading to *starvation* **[3]**

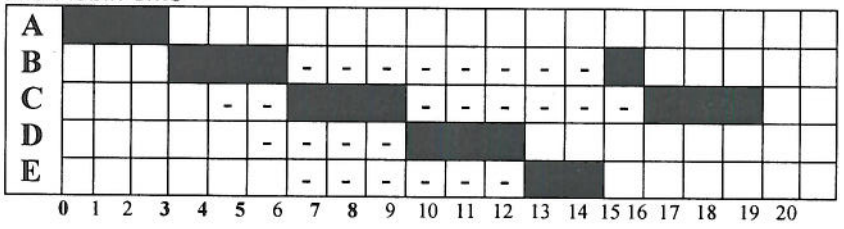
**(b) New Computed Example**

*SRJF*



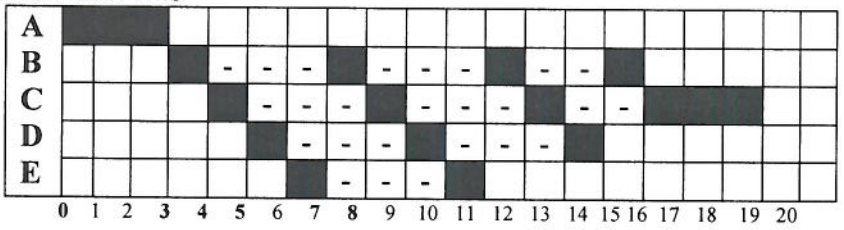

Avg waiting time: (0+0+8+4+1) / 5 = 2.6 ms, Avg turnaround time: (3+4+14+7+3)/5 = 6.2 ms **[3]**

*Round Robin 3ms*




Avg waiting time: (0+8+8+4+6) / 5 = 5.2 ms, Avg turnaround time: (3+12+14+7+8)/5 = 8.8 ms **[3]**

*Round Robin 1ms*




Avg waiting time: (0+8+8+6+3) / 5 = 5 ms, Avg turnaround time: (3+12+14+9+5)/5 = 8.6 ms **[3]**

(c) [Bookwork]

Readers-writers problem

```
var mutex, wrt: Semaphore;
var read_count: int;

init(mutex, 1);  init(wrt,1);
read_count=0;
```

*Writer process:*

```
procedure writer()
begin
  while(TRUE) do
    begin
      produce_item;
      wait(wrt);
      write_item;
      signal(wrt);
    end;
end;
```

*Reader process:*

```
procedure reader()
begin
  while(TRUE) do
    begin
        Wait(mutex);
        read_count=read_count+1;
        if read_count = 1 then wait(wrt);
        signal(mutex);
        get_item;
        wait(mutex);
        read_count = read_count – 1;
        if read_count = 0 then signal(wrt);
        signal(mutex);
        consume_item;
    end;
end;
```

(d) Binding of instructions and data to memory addresses can be done at:

*Compile time:* Absolute code can be generated at this stage, if it is known in advance where will the process reside in memory (e.g. MSDOS .com programs). No requirement for special hardware; easy to implement.
*Load time:* Relocatable code can be generated if it is not known in advance where will the code reside. Memory addresses are *relative* (to the beginning of the program), replaced by absolute ones during loading. No requirement for special hardware
*Execution time:* performed if a process can be moved during its execution from one memory segment to another (e.g. swapping) --- requires special hardware.

***QUESTION 2:***

(a)
Optimal page replacement: replace page that will not be used for the longest period of time.

- Advantages: Lowest page-fault rate of all algorithms – can be used to evaluate relative performance of other page replacement algorithms.
- Disadvantages: Difficult to impossible to implement since we need to know the stream of requested pages in advance.

Least-recently used: replace the page that has not been used for the longest time

- Advantages: Good performance
- Disadvantages: Not the easiest to implement. Implementations below: **[2]**

[Counters] A global counter gets updated with every memory reference; each page has a counter associated with it. When a reference to a page is made, the contents of the global counter are copied to the associated counter. LRU algorithm searches through the pages and picks the one with the lowest counter value. **[2]**

[Stack] A stack of page numbers is kept; whenever a page is referenced, it is removed from the stack and placed on the top. Therefore, the top of the stack is the most recently used page, bottom of the stack is the LRU page. **[2]**

(b)

Impose ordering on chopsticks, assigning a unique number (between 1 and 5) to each of them. A dining philosopher must acquire the lower number chopstick first before taking the higher one.
Deadlock now impossible since philosophers 1 and 5 will compete for one of the chopsticks, and whoever acquires it first, will eat, release the chopstick, allowing the other one to complete. **[3]**

(c)

*Condition 2 - Hold and wait*: Make sure that if a process requests a resource, it does not hold any other ones. We can do this in two different ways;
- Require that each process requests and gets allocated all the resources it needs before it begins execution
- OR, require that a process release all resources that it holds before requesting a new one.

Attacking condition 2 is sufficient, but
- Can result into low resource utilisation (processes holding resources for much longer than they need them for)
- Can result into starvation (e.g. for processes that need several popular resources) **[4]**

(d) [new computed example]

Optimal page replacement algorithm (6 page faults)

| | **6** | **1** | **1** | **4** | **5** | **2** | **4** | **5** | **6** | **2** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Frame1** | **6** | 6 | | 6 | 6 | 2 | | | 2 | |
| **Frame2** | **-** | 1 | | 1 | 5 | 5 | | | 5 | |
| **Frame3** | **-** | - | | 4 | 4 | 4 | | | 6 | |

(the last replacement is not important; any will do) **[2]**

FIFO page replacement algorithm (6 page faults)

| | **6** | **1** | **1** | **4** | **5** | **2** | **4** | **5** | **6** | **2** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Frame1** | **6** | 6 | | 6 | 5 | 5 | | | 5 | |
| **Frame2** | **-** | 1 | | 1 | 1 | 2 | | | 2 | |
| **Frame3** | **-** | - | | 4 | 4 | 4 | | | 6 | |

**[2]**

LRU (Least recently used) page replacement algorithm (7 page faults)

| | **6** | **1** | **1** | **4** | **5** | **2** | **4** | **5** | **6** | **2** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Frame1** | **6** | 6 | | 6 | 5 | 5 | | | 5 | 5 |
| **Frame2** | **-** | 1 | | 1 | 1 | 2 | | | 6 | 6 |
| **Frame3** | **-** | - | | 4 | 4 | 4 | | | 4 | 2 |

**[2]**

(e) [Bookwork]

A queue is used to hold processes waiting (and are blocked) on that semaphore. If the processes in a queue are released on a FIFO fashion (the process waiting the longest is released from the queue first) the semaphore is called a *strong* semaphore. A semaphore that does not specify the order in which processes are removed from the queue is a *weak* semaphore. **[1]**