Master - June 06

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

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2006

ISE Part I: MEng, BEng and ACGI

PRINCIPLES OF COMPUTING AND SOFTWARE ENGINEERING (PART A)
INTRODUCTION TO COMPUTER ARCHITECTURE

Friday 9th June 2006 2:00pm

Corrected Copy

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).

This exam is **closed book**

Time allowed: 1:30 hours.

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

Examiners responsible:

First Marker(s): Clarke, T.
Second Marker(s): Constantinides, G.

Special information for invigilators:

The booklet Exam Notes 2006 should be distributed with the Examination Paper.

Information for candidates:

The booklet Exam Notes 2006, 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) 8 bit two's complement $8B_{(16)}$ into a decimal number
(ii) Unsigned $8FFF_{(16)}$ into a decimal number.
(iii) $-13_{(10)}$ into 8 bit sign and magnitude (write your answer in hexadecimal).
(iv) $-111_{(10)}$ into 12 bit two's complement binary.

[8]

b) Derive the IEEE-754 representations for (i) 1.125 and (ii) 9×2^{10}. In each case, state what is the absolute numeric difference between these numbers and the nearest distinct numbers that can be represented in IEEE754.

[8]

c) Assume that R0, R1 contain *unsigned* 32 bit numbers and R2, R3 contain two's complement *signed* 32 bit numbers. Write efficient ARM assembly code fragments that implement the following pseudo-code statements:

(i) If R0 > R1 then R2 := 1 else R5 := R6 but with bits 3,4,5 set to 0.
(ii) If R2 > R3 then R5 := $2000_{(10)}$ else R5 := –R3

[8]

d) Figure 1.1 shows a fragment of ARM assembly code program. Explain what the sequence of instructions `A,B,C,D` implements in the two cases:

(i) R8 = 0
(ii) R8 = 1

Thereby deduce the function of the loop.

[Note that ADDCS & ADCS are not the same!]

[8]

e) Using the instruction timing information at the end of the Exam Notes 2006 booklet, and ignoring the instructions before `LOOP` in Figure 1.1, determine the speed of the loop in words written to memory per cycle.

[8]

```
        ADR   R2, ANUM
        ADR   R3, BNUM
        ADR   R4, CNUM
        MOV   R6, #10
        MOV   R8, #0
LOOP    LDR   R0, [R2],#4
        LDR   R1, [R3],#4
A       CMP   R8, #1
B       ADCS  R0, R0, R1
C       MOVCS R8, #1
D       MOVCC R8, #0
        STR   R0, [R4],#4
        SUBS  R6, R6, #1
        BNE   LOOP
```

Figure 1.1

2. Let a, b be 32 bit numbers and a_1, b_1, a_0, b_0 be the top and bottom 16 bits respectively of a, b such that:

(2.1) $a = a_0+2^{16}a_1$
(2.2) $b = b_0+2^{16}b_1$

The 64 bit product of a and b can be expressed in terms of four 16×16→32 bit products as follows:

(2.3) $(a_0+2^{16}a_1)\times(b_0+2^{16}b_1) = 2^{32}(a_1\times b_1) + 2^{16}(a_0\times b_1+a_1\times b_0)+(a_0\times b_0)$

Identity (2.3) may be used to compute an unsigned 32×32→64 bit multiply in ARM assembly code using four applications of the ARM 32×32→32 bit MUL instruction.

Assume that the two 32 bit multiplicands, a and b, are initially in R0 and R1; and the top and bottom 32 bits of the result, c_1 and c_0, will be stored in R3, R2 respectively.

a) Write ARM assembly code that computes a_0, a_1, b_0, b_1 from a, b.

[6]

b) Write ARM assembly code that sets c_1 and c_0 to $a_1\times b_1$ and $a_0\times b_0$ respectively. Why is this helpful?

[6]

c) Write ARM assembly code which computes in a register the sum of products:

$z = a_0\times b_1+a_1\times b_0$.

Add any resulting carry into c_1, c_0 with the appropriate weighting required by (2.3).

[6]

d) Write code that adds the bits of z to c_1, c_0 as is required to implement (2.3).

[6]

e) Write additional instructions which turn the above assembly code into a subroutine, leaving unchanged all registers excepting R3 & R2, and using a stack in which R13 points to the lowest word address containing a stacked data word. You need not copy your preceding code but must state precisely where the additional instructions are placed in relation to the previous code.

[6]

3.

A new CPU architecture, ARM-LONGPIPE, implements the ARM ISA, using a different hardware pipeline and branch prediction strategy from the current (ARM7) architecture. The time lost through pipeline stalling when a branch is incorrectly predicted, together with the likelihood that any given branch is correctly predicted, and hence executes in only one clock cycle, is shown in Figure 3.1.

a) Assuming that 30% of all ARM instructions executed are branches, and all other ARM instructions are executed at the rate of one per clock cycle, determine the average number of instructions executed per clock cycle in the two architectures.

[10]

b) Detail the sequence of data-path operations during the execution stage, number 3, of the ARM7 pipeline. Give one possible assignment of these data-path operations to pipeline stages 5 & 6 of the LONGPIPE architecture. The two architectures, implemented in identical technology, utilise the times given in Figure 3.2 for each of their pipeline stages. State the minimum clock period for each architecture and hence, under the assumptions of part (a), determine the average instruction rates in MIPS (millions of instructions per second) of the two architectures when clocked at the maximum possible frequency.

[10]

c) Demonstrate, giving assembly code to illustrate your answer, how conditional instruction execution in the ARM7 architecture can be used to speed up IF-THEN-ELSE pseudo-code by eliminating pipeline stalls.

[10]

Architecture	**Pipeline stall time (cycles)**	**Correct branch prediction probability**
ARM7	3	0.3
ARM-LONGPIPE	6	0.9

Figure 3.1 – pipeline characteristics

Stage	**ARM7**	**ARM-LONGPIPE**
1	3.5ns	1.0ns
2	3ns	0.7ns
3	2ns	1.1ns
4		1.1ns
5		1.0ns
6		1.1ns

Figure 3.2 - pipeline stage times in nanoseconds

4.

a) An ARM processor has a 32 bit memory data bus connected to a direct-mapped cache with 8 lines each of 16 bytes (4 words of 32 bits). You may assume that all cache memory access is word-based. Detail which bits of the ARM memory address correspond to the cache *tag*, *index* and *select* fields.

[10]

b) The processor from part (a) issues data memory word read operations to a sequence of addresses as shown below:

0x0, 0x4, 0x8, 0xC, 0x10, 0x14, 0x18, 0x1C

Which of these data memory read operations lead to memory misses, assuming that initially all cache lines are invalid (V=0)? How many words are read from main memory into the cache?

[10]

c) Figure 4.1 shows an ARM assembly code program. Compute the sequence of memory read or write addresses (ignoring instruction fetch). Explain in words what function the program implements. The program is run on ARM processors with write-through direct-mapped caches of size:

(i) 4 lines each of 2 words
(ii) 4 lines each of 4 words

In each case, assuming initially invalid cache lines, and again ignoring instruction fetch, determine the hit rate of the cache for memory reads. State, giving reasons, whether in these cases the hit rate for memory writes affects performance.

[10]

```
        MOV   R2, #&1000
        MOV   R3, #&2020
        MOV   R10, #5
LOOP    LDR   R0, [R2,#4]!
        STR   R0, [R3,#4]!
        SUBS  R10, R10, #1
        BNE   LOOP
```

Figure 4.1

Handout for
Paper E1.9A / E2.19

EXAM NOTES 2006

Introduction to Computer Architecture
Principles of Computing

Key to Tables	
{cond}	Refer to Table **Condition Field {cond}**
<Oprnd2>	Refer to Table **Oprnd2**
{field}	Refer to Table **Field**
S	Sets condition codes (optional)
B	Byte operation (optional)
H	Halfword operation (optional)
T	Forces address translation. Cannot be used with pre-indexed addresses
<a_mode1>	Refer to Table **Addressing Mode 1**
<a_mode2>	Refer to Table **Addressing Mode 2**
<a_mode3>	Refer to Table **Addressing Mode 3**
<a_mode4>	Refer to Table **Addressing Mode 4**
<a_mode5>	Refer to Table **Addressing Mode 5**
<a_mode6>	Refer to Table **Addressing Mode 6**
#32_Bit_Immed	A 32-bit constant, formed by right-rotating an 8-bit value by an even number of bits

Operation		Assembler	S updates	Action	Notes
Move	Move	MOV{cond}{S} Rd, <Oprnd2>	N Z C	Rd:= <Oprnd2>	
	NOT	MVN{cond}{S} Rd, <Oprnd2>	N Z C	Rd:= 0xFFFFFFFF EOR <Oprnd2>	
	SPSR to register	MRS{cond} Rd, SPSR		Rd:= SPSR	*Architecture 3, 3M and 4 only*
	CPSR to register	MRS{cond} Rd, CPSR		Rd:= CPSR	*Architecture 3, 3M and 4 only*
	register to SPSR	MSR{cond} SPSR{field}, Rm		SPSR:= Rm	*Architecture 3, 3M and 4 only*
	register to CPSR	MSR{cond} CPSR{field}, Rm		CPSR:= Rm	*Architecture 3, 3M and 4 only*
	immediate to SPSR flags	MSR{cond} SPSR_f, #32_Bit_Immed		SPSR:= #32_Bit_Immed	*Architecture 3, 3M and 4 only*
	immediate to CPSR flags	MSR{cond} CPSR_f, #32_Bit_Immed		CPSR:= #32_Bit_Immed	*Architecture 3, 3M and 4 only*
ALU	Arithmetic				
	Add	ADD{cond}{S} Rd, Rn, <Oprnd2>	N Z C V	Rd:= Rn + <Oprnd2>	
	with carry	ADC{cond}{S} Rd, Rn, <Oprnd2>	N Z C V	Rd:= Rn + <Oprnd2> + Carry	
	Subtract	SUB{cond}{S} Rd, Rn, <Oprnd2>	N Z C V	Rd:= Rn - <Oprnd2>	
	with carry	SBC{cond}{S} Rd, Rn, <Oprnd2>	N Z C V	Rd:= Rn - <Oprnd2> - NOT(Carry)	
	reverse subtract	RSB{cond}{S} Rd, Rn, <Oprnd2>	N Z C V	Rd:= <Oprnd2> - Rn	
	reverse subtract with carry	RSC{cond}{S} Rd, Rn, <Oprnd2>	N Z C V	Rd:= <Oprnd2> - Rn - NOT(Carry)	
	Negate				
	Multiply	MUL{cond}{S} Rd, Rm, Rs	N Z	Rd:= Rm * Rs	*Not in Architecture 1*
	accumulate	MLA{cond}{S} Rd, Rm, Rs, Rn	N Z	Rd:= (Rm * Rs) + Rn	*Not in Architecture 1*
	unsigned long	UMULL{cond}{S} RdHi, RdLo, Rm, Rs	N Z	RdHi:= (Rm*Rs)[63:32] RdLo:= (Rm*Rs)[31:0]	*Architecture 3M and 4 only*
	unsigned accumulate long	UMLAL{cond}{S} RdHi, RdLo, Rm, Rs	N Z	RdLo:=(Rm*Rs)+RdLo RdHi:=(Rm*Rs)+RdHi+ CarryFrom((Rm*Rs)[31:0]+RdLo))	*Architecture 3M and 4 only*
	signed long	SMULL{cond}{S} RdHi, RdLo, Rm, Rs	N Z	RdHi:= signed(Rm*Rs)[63:32] RdLo:= signed(Rm*Rs)[31:0]	*Architecture 3M and 4 only*
	signed accumulate long	SMLAL{cond}{S} RdHi, RdLo, Rm, Rs	N Z	RdHi:=signed(Rm*Rs)+RdHi+ CarryFrom((Rm*Rs)[31:0]+RdLo))	*Architecture 3M and 4 only*
	Compare	CMP{cond} Rd, <Oprnd2>	N Z C V	CPSR flags:= Rn - <Oprnd2>	
	negative	CMN{cond} Rd, <Oprnd2>	N Z C V	CPSR flags:= Rn + <Oprnd2>	
	Logical				
	Test	TST{cond} Rn, <Oprnd2>	N Z C	CPSR flags:= Rn AND <Oprnd2>	
	Test equivalence	TEQ{cond} Rn, <Oprnd2>	N Z C	CPSR flags:= Rn EOR <Oprnd2>	Does not update the V flag
	AND	AND{cond}{S} Rd, Rn, <Oprnd2>	N Z C	Rd:= Rn AND <Oprnd2>	
	EOR	EOR{cond}{S} Rd, Rn, <Oprnd2>	N Z C	Rd:= Rn EOR <Oprnd2>	
	ORR	ORR{cond}{S} Rd, Rn, <Oprnd2>	N Z C	Rd:= Rn OR <Oprnd2>	
	Bit Clear	BIC{cond}{S} Rd, Rn, <Oprnd2>	N Z C	Rd:= Rn AND NOT <Oprnd2>	
	Shift/Rotate				See Table **Oprnd2**

Operation		Assembler	Action	Notes
Branch	Branch	`B{cond} label`	R15:= address	
	with link	`BL{cond} label`	R14:=R15, R15:= address	
	and exchange instruction set	`BX{cond} Rn`	R15:=Rn, T bit:= Rn[0]	*Architecture 4 with Thumb only* Thumb state; Rn[0] = 0 ARM state; Rn[0] =1
Load	Word	`LDR{cond} Rd, <a_mode1>`	Rd:= [address]	
	with user-mode privilege	`LDR{cond}T Rd, <a_mode2>`		
	Byte	`LDR{cond}B Rd, <a_mode1>`	Rd:= [byte value from address] Loads bits 0 to 7 and sets bits 8-31 to 0	
	with user-mode privilege	`LDR{cond}BT Rd, <a_mode2>`		
	signed	`LDR{cond}SB Rd, <a_mode3>`	Rd:= [signed byte value from address] Loads bits 0 to 7 and sets bits 8-31 to bit 7	*Architecture 4 only*
	Halfword	`LDR{cond}H Rd, <a_mode3>`	Rd:= [halfword value from address] Loads bits 0 to 15 and sets bits 16-31 to 0	*Architecture 4 only*
	signed	`LDR{cond}SH Rd, <a_mode3>`	Rd:= [signed halfword value from address] Loads bits 0 to 15 and sets bits 16-31 to bit 15	*Architecture 4 only*
	Multiple			
	Block data operations			
	Increment Before	`LDM{cond}IB Rd{!}, <regs>{^}`	Stack manipulation (pop)	! sets the W bit (updates the base register after the transfer ^ sets the S bit
	Increment After	`LDM{cond}IA Rd{!}, <regs>{^}`		
	Decrement Before	`LDM{cond}DB Rd{!}, <regs>{^}`		
	Decrement After	`LDM{cond}DA Rd{!}, <regs>{^}`		
	Stack operations	`LDM{cond}<a_mode4> Rd{!},<registers>`		! sets the W bit (updates the base register after the transfer
	and restore CPSR	`LDM{cond}<a_mode4> Rd{!}, <registers+pc>`		
	User registers	`LDM{cond}<a_mode4> Rd, <registers>^`		
Store	Word	`STR{cond} Rd, <a_mode1>`	[address]:= Rd	
	with user-mode privilege	`STRT{cond} Rd, <a_mode2>`		
	Byte	`STRB{cond} Rd, <a_mode1>`	[address]:= byte value from Rd	
	with user-mode privilege	`STRBT{cond} Rd, <a_mode2>`		
	Halfword	`STR{cond}H Rd, <a_mode3>`	[address]:= halfword value from Rd	*Architecture 4 only*
	Multiple			
	Block data operations			
	Increment Before	`STM{cond}IB Rd{!}, <registers>{^}`	Stack manipulation (push)	! sets the W bit (updates the base register after the transfer
	Increment After	`STM{cond}IA Rd{!}, <registers>{^}`		^ sets the S bit
	Decrement Before	`STM{cond}DB Rd{!}, <registers>{^}`		
	Decrement After	`STM{cond}DA Rd{!}, <registers>{^}`		
	Stack operations	`STM{cond}<a_mode5> Rd{!}, <regs>`		
	User registers	`STM{cond}<a_mode5> Rd{!}, <regs>^`		
Swap	Word	`SWP{cond} Rd, Rm, [Rn]`		*Not in Architecture 1 or 2*
	Byte	`SWP{cond}B Rd, Rm, [Rn]`		*Not in Architecture 1 or 2*
Coprocessors	Data operations	`CDP{cond} p<cpnum>, <op1>, CRd, CRn, CRm, <op2>`		*Not in Architecture 1*
	Move to ARM reg from coproc	`MRC{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>`		
	Move to coproc from ARM reg	`MCR{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>`		
	Load	`LDC{cond} p<cpnum>, CRd, <a_mode6>`		
	Store	`STC{cond} p<cpnum>, CRd, <a_mode6>`		
Software Interrupt		`SWI #24_Bit_Value`		24-bit immediate value

Addressing Mode 1	
Immediate offset	[Rn, #+/-12_Bit_Offset]
Register offset	[Rn, +/-Rm]
Scaled register offset	[Rn, +/-Rm, LSL #shift_imm]
	[Rn, +/-Rm, LSR #shift_imm]
	[Rn, +/-Rm, ASR #shift_imm]
	[Rn, +/-Rm, ROR #shift_imm]
	[Rn, +/-Rm, RRX]
Pre-indexed offset	
Immediate	[Rn, #+/-12_Bit_Offset]!
Register	[Rn, +/-Rm]!
Scaled register	[Rn, +/-Rm, LSL #shift_imm]!
	[Rn, +/-Rm, LSR #shift_imm]!
	[Rn, +/-Rm, ASR #shift_imm]!
	[Rn, +/-Rm, ROR #shift_imm]!
	[Rn, +/-Rm, RRX]!
Post-indexed offset	
Immediate	[Rn], #+/-12_Bit_Offset
Register	[Rn], +/-Rm
Scaled register	[Rn], +/-Rm, LSL #shift_imm
	[Rn], +/-Rm, LSR #shift_imm
	[Rn], +/-Rm, ASR #shift_imm
	[Rn], +/-Rm, ROR #shift_imm
	[Rn, +/-Rm, RRX]

Addressing Mode 2	
Immediate offset	[Rn, #+/-12_Bit_Offset]
Register offset	[Rn, +/-Rm]
Scaled register offset	[Rn, +/-Rm, LSL #shift_imm]
	[Rn, +/-Rm, LSR #shift_imm]
	[Rn, +/-Rm, ASR #shift_imm]
	[Rn, +/-Rm, ROR #shift_imm]
	[Rn, +/-Rm, RRX]
Post-indexed offset	
Immediate	[Rn], #+/-12_Bit_Offset
Register	[Rn], +/-Rm
Scaled register	[Rn], +/-Rm, LSL #shift_imm
	[Rn], +/-Rm, LSR #shift_imm
	[Rn], +/-Rm, ASR #shift_imm
	[Rn], +/-Rm, ROR #shift_imm
	[Rn, +/-Rm, RRX]

Addressing Mode 3 - Signed Byte and Halfword Data Transfer	
Immediate offset	[Rn, #+/-8_Bit_Offset]
Pre-indexed	[Rn, #+/-8_Bit_Offset]!
Post-indexed	[Rn], #+/-8_Bit_Offset
Register	[Rn, +/-Rm]
Pre-indexed	[Rn, +/-Rm]!
Post-indexed	[Rn], +/-Rm

Addressing Mode 6 - Coprocessor Data Transfer	
Immediate offset	[Rn, #+/-(8_Bit_Offset*4)]
Pre-indexed	[Rn, #+/-(8_Bit_Offset*4)]!
Post-indexed	[Rn], #+/-(8_Bit_Offset*4)

Oprnd2	
Immediate value	#32_Bit_Immed
Logical shift left	Rm LSL #5_Bit_Immed
Logical shift right	Rm LSR #5_Bit_Immed
Arithmetic shift right	Rm ASR #5_Bit_Immed
Rotate right	Rm ROR #5_Bit_Immed
Register	Rm
Logical shift left	Rm LSL Rs
Logical shift right	Rm LSR Rs
Arithmetic shift right	Rm ASR Rs
Rotate right	Rm ROR Rs
Rotate right extended	Rm RRX

Field		
Suffix	Sets	
_c	Control field mask bit	(bit 3)
_f	Flags field mask bit	(bit 0)
_s	Status field mask bit	(bit 1)
_x	Extension field mask bit	(bit 2)

Condition Field {cond}	
Suffix	Description
EQ	Equal
NE	Not equal
CS	Unsigned higher or same
CC	Unsigned lower
MI	Negative
PL	Positive or zero
VS	Overflow
VC	No overflow
HI	Unsigned higher
LS	Unsigned lower or same
GE	Greater or equal
LT	Less than
GT	Greater than
LE	Less than or equal
AL	Always

Addressing Mode 4			
Addressing Mode		Stack Type	
IA	Increment After	FD	Full Descending
IB	Increment Before	ED	Empty Descending
DA	Decrement After	FA	Full Ascending
DB	Decrement Before	EA	Empty Ascending

Addressing Mode 5			
Addressing Mode		Stack Type	
IA	Increment After	EA	Empty Ascending
IB	Increment Before	FA	Full Ascending
DA	Decrement After	ED	Empty Descending
DB	Decrement Before	FD	Full Descending

ARM REFERENCE NOTES

2005/2006

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
```

R2.2

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

R2.3

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

R2.4

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 (0 ≤ s ≤ 255, 0 ≤ r ≤ 15)	Assembler will translate negative values changing op-code as necessary Assembler will work out rotate if it exists
Rm, shift #s **Rm, rrx #1**	(1 ≤ s ≤ 31) shift => lsr,lsl,asr,asl,ror	rrx always sets carry ror sets carry if S=1 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, MULA

ARM3 and above

MUL rd, rm, rs	multiply (32 bit)	Rd := (Rm*Rs)[31:0]
MULA 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

Assembly Directives

```
SIZE     EQU    100                ; defines a numeric constant
BUFFER   %      200                ; defines bytes of zero initialised storage
         ALIGN                     ;forces next item to be word-aligned
MYWORD   DCW    &80000000          ;defines word of storage
MYDATA   DCD    0,1,&ffff0000,&12345 ;defines one or more words of storage
TEXT     =      "string", &0d, &0a, 0 ; defines one or more bytes of storage. Each
                                  ; operand can be string or number in range 0-255
         LDR r0, =numb             ; assembles to instructions that set r0 to immediate
                                  ; value numb – numb may be too large for a MOV operand
                                  ;
```

NB:
& prefixes hex constant: &3FFF
Case does not matter anywhere (except inside strings)

Exceptions & Interrupts

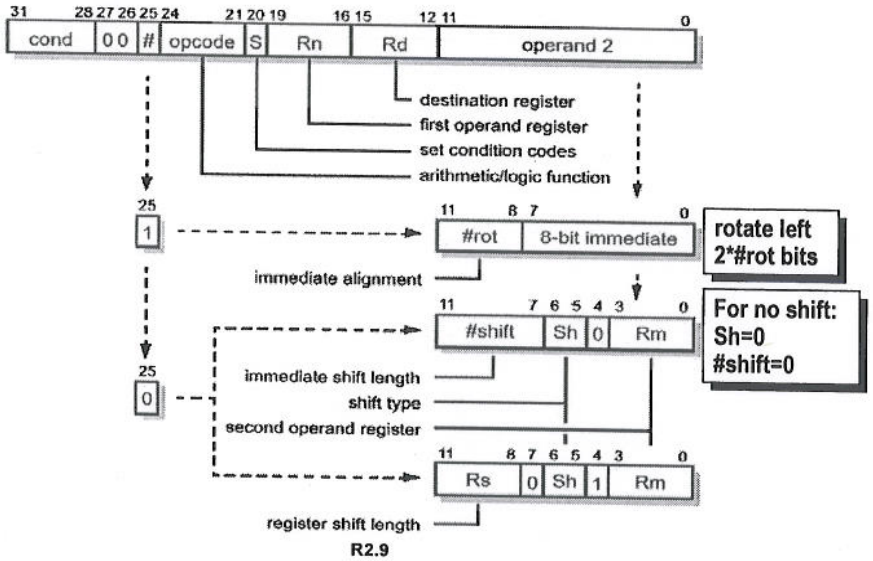
Exception	Return
SWI or undefined instruction	MOVS pc, R14
IRQ, FIQ, prefetch abort	SUBS pc, r14, #4
Data abort (needs to rerun failed instruction)	SUBS pc, R14, #8

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

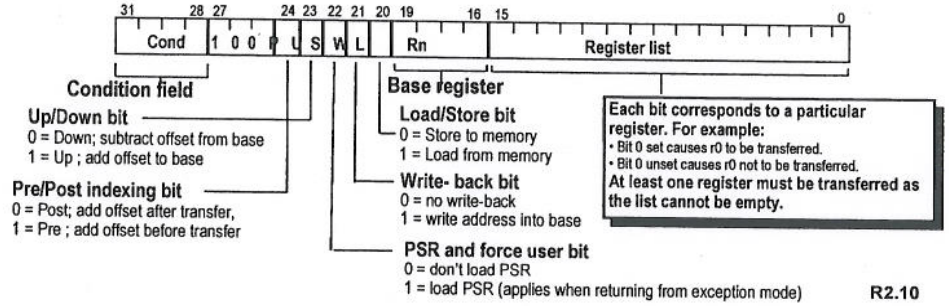
Data Processing Instruction Binary Encoding



R2.9

Multiple Register Transfer Binary Encoding

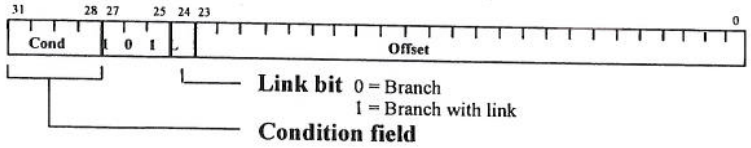
- The Load and Store Multiple instructions (LDM / STM) allow betweeen 1 and 16 registers to be transferred to or from memory.



R2.10

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.

R2.11

Instruction Set Overview

31 28	27 26 25	24 23 22 21	20 19 16	15 12	11 8	7 5	4 3 0	
Cond	0 0 I	Opcode	S Rn	Rd	Operand 2			Data Processing PSR Transfer
Cond	0 0 0 0 0 0	A S	Rd	Rn	Rs	1 0 0 1	Rm	Multiply
Cond	0 0 0 1 0	B 0 0	Rn	Rd	0 0 0 0	1 0 0 1	Rm	Single Data Swap
Cond	0 1 I	P U B W L	Rn	Rd	offset			Single Data Transfer
Cond	0 1 1	XXXXXXXXXXXXXXXXXXX				I	XXXX	Undefined
Cond	1 0 0	P U S W L	Rn	Register List				Block Data Transfer
Cond	1 0 1	L	offset					Branch
Cond	1 1 0	P U N W L	Rn	CRd	CP#	offset		Coproc Data Transfer
Cond	1 1 1 0	CP Opc	CRn	CRd	CP#	CP	0 CRm	Coproc Data Operation
Cond	1 1 1 0	CP Opc L	CRn	Rd	CP#	CP	1 CRm	Coproc Register Transfer
Cond	1 1 1 1	ignored by processor						Software Interrupt

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)
Any instruction, with condition false	1
data processing (all except register-valued shifts)	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)

Exam master

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

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2006

ISE Part I: MEng, BEng and ACGI

PRINCIPLES OF COMPUTING AND SOFTWARE ENGINEERING (PART B)

OPERATING SYSTEMS

Friday 9th June 2006 2:00pm

There are TWO questions on this paper.

Answer ONE question.

This exam is CLOSED BOOK

Time allowed: 1 hour

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

Examiners responsible:

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

E1.9 – SECTION B: Operating Systems [1 hour]

Answer ONLY ONE of the following two questions

QUESTION 1:

(a) Describe the MQS (Multi-level Queue Scheduling) process scheduling algorithm and list its advantages and disadvantages. [4]

(b) In the "producer-consumer" problem, two processes are communicating through a buffer that can hold ***0*** to ***n*** items. The producer process continuously produces items and places them in the buffer, while the consumer continuously fetches items from the buffer and consumes them. The following conditions are in place:

i. The producer process can only place items in the buffer if there is space available in the buffer – it blocks otherwise.
ii. The consumer process can fetch items from the buffer only if there are items available – it blocks otherwise.
iii. Mutual exclusion is required: the producer and the consumer processes cannot access the buffer at the same time.

Using semaphores to ensure that the conditions above hold, provide Pascal procedures for the producer and consumer processes. Declare and properly initialise all semaphores you 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, get_item, consume_item, and that the constant ***"n"*** has been declared. [7]

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

- You have three available frames
- The reference string is 2-5-1-2-7-5-7-3-5-1

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 [4]
ii. LRU (Least Recently Used) page replacement [4]

(d) In the context of memory management, describe the concept of "thrashing". [1]

QUESTION 2:

(a) In the context of the client-servel model of operating system organisation, describe the microkernel-based organisation and list its advantages and disadvantages. [3]

(b) Consider the following set of processes, with their corresponding duration, arrival times, and priority levels [*higher numbers indicate higher priority*]:

Process	Arrival time (ms)	Duration (ms)	Priority level
A	0	3	6
B	3	5	3
C	5	1	9
D	8	5	10

Show the order of execution (including timing information) of the processes if the scheduler implements the following scheduling algorithms:

(i) Shortest-remaining job first (SRJF) [3]
(ii) Priority-scheduling without preemption [3]
(iii) Priority-scheduling with pre-emption [3]

For each of the algorithms calculate the average waiting time, and the average turnaround time.

(c) Specify the steps taken by *banker's algorithm* for dynamically avoiding deadlocks; specify the algorithm's main weakness. [3]

(d) In the context of memory management, describe the scheme known as "*paging*" and list its advantages and disadvantages. Describe how virtual memory can be implemented through "*demand paging*". [4]

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

Master Copy –

Solutions for E1.9/E2.19 (A) June 06

Solution to Question 1

Part a) is taught material. Parts b & c are novel applications of taught material. This question tests whether the students understand number representations, and the differences between signed and unsigned numbers. Part c) also tests whether they can use ARM instruction conditions efficiently.

36 minutes for the question => 7 minutes each part

Each of the 5 parts has 8 marks.

a)
i) $8B_{(16)}$= -($75_{(16)}$)=-117
ii) $8FFF_{(16)}$ = 36863
iii) 8D
iv) $-111_{(10)}$=$-6F_{(16)}$=91 → $1111\ 1001\ 0001_{(2)}$

b)
seee eeee emmm mmmm mmmm mmmm mmmm mmmm
$1.125*2^0$ -> sign=0, exp = 7F, mant = 300000 -> 3F900000
$9*2^{10}=1.125*2^{13}$. sign=0, exp=8C, mant=300000 -> 46100000
Difference= $2^{-23} * 2^{exp} = 2^{-23}$ or 2^{-10}.

c) R0,R1 unsigned, R2, R3 signed (NB signed = two's complement signed unless otherwise specified)

(i) If R0 > R1 then R2 := 1 else R5 := R6 with bits 3,4,5 set to 0.

```
CMP R0, R1
MOVHI R2, #1
BICLS R2, R3, #&38
```

(ii) If R2 > R3 then R5 := $2000_{(10)}$ else R5 := -R3

```
CMP R2, R3
MOVGT R5, #2000
RSBLE R5, R3, #0  //NB MVN will not do as it inverts bits
```

d) if R8 >= 1, R0 := R0+R1+1, else R0 := R0 + R1, set R0= carry from the addition. Hence the loop performs multiple-word addition from vectors [R2], [R3] to vector [R4].

e) 5X1 (other) + 3X4 (load/store) +1X4 (branch) = 21 cycles for 1 word. so:
0.05 words written per cycle. (strictly, 207 cycles for 210 words since last interation is 3 cycles faster due to no branching it and BNE.

Solution to Question 2

This question leads through the implementation of a novel assembly language program which tests understanding of ARM data processing & multiply instructions.

An optimal solution is shown below. Note that without using the exclusive or extraction of the two 16 bit halves of a register would require a 16 bit constant set in a register, or two shifts. This optimal extraction method has not been taught and is difficult to devise so a less efficient solution will not be highly penalised.

27 minutes for the question => 5.4 minutes each part
Each of the 5 parts has 6 marks.

```
        ; register allocation:
        ;R0 x, R1 y, R3:R2 x*y, R4:a0, R5:a1, R6:b0, R7:b1

Part e: see also below
        STMFD R13!, {R4-R7,R10,11,R14}
Part a:
        ; a1:a0 := a
        MOV   R5, R0, lsr #16
        EOR   R4, R0, R5 lsl #16
        ; b1:b0 := b
        MOV   R7, R1, lsr #16
        EOR   R6, R1, R7 lsl #16
Part b:         ;  use R2.R3 since these components make up part of the result sum
                ; 2^32 factor aligns with top 32 bits (R3) and 1x factor aligns with R2.
                ; compute constituent 16X16 products
        MUL   R2, R4, R6      ; store in correct result word
        MUL   R3, R5, R7      ; store in correct result word
        MUL   R10, R4, R7     ; temporary
        MUL   R11, R5, R6     ; temporary: can't use MLA since this gives no carry
Part c:
                ; add the two top X bottom products
        ADDS  R10, R10, R11
                ; add carry from this addition to result with correct weight
        ADDCS R3, R3, #&10000
                ; finally add the top X bottom product sum onto the result
Part d:
        ADD   R3, R3, R10, lsr #16
        ADD   R2, R2, R10, lsl #16
Part e: (see also above)
        LDMFD R13!,{R4-7,R10, R11, R15}    ; could omit push/pop r14/r15 and replace
                                           ; here with MOV r15, r14
```

Solution to Question 3

Parts a and b are new applications of taught theory: the question re LONGPIPE pipeline stages requires students to have a deeper understanding what they have been taught. Part c requires understanding and analysis of ARM ISA.

27 minutes for the question

a)
[10 marks]
ARM7: 1/(1+0.3*3*(1-0.3))= 0.613 instructions per cycle
LONGPIPE: 1/(1+ 0.3*6*(1-0.9))=0.847 instructions per cycle
Note common mistake is P->1-P in equation 1/(1+PBS)
b)
[10 marks]

ARM7: max time = 3.5 ns => 286 MHz => 175MIPS
LONGPIPE: max time = 1.1ns => 909MHz => 770MIPS

The ARM execute stage includes sequentially: (1) read from registers, (2) shift,(3) ALU, (4) write back to registers. These could make 2 stages by merging (1) & (2), (3) & (4). Other plausible answers will be accepted.

Note that pipelining means that fetch & decode stages execute simultaneously with execute stage: however this part of the question relates only to what happens in execute stage.

c)
[10 marks]

```
        CMP R0, #1
        BPL THENPART
        MOV R2, #2
        B ENDCODE
THENPART
        MOV R3, #2
ENDCODE
```

This has one pipeline stall, due to branch that is executed, in all cases. Transforms to:

```
        CMP R0, #1
        MOVPL R3, #1
        MOVMI R2, #1
```

Same number of instructions executed (or one less) and no pipeline stall.

Solution to Question 4

This question tests understanding of direct-mapped caches. Part a is bookwork, part b is an application of learnt theory, c tests analysis of a simple assembly language program and deeper understanding of caches.

27 minutes for the question

a) **[10 marks]**
Bottom 2 bits are *byte select* and do not count because all access is word-based, but they mean that the cache control bits come from A31:2 of the ARM address.
4 words per line => A3:2 are select
8 lines => A6:4 are index
Tag is A31:7

b) **[10 marks]**

0x0, 0x4, 0x8, 0xC, 0x10, 0x14, 0x18, 0x1C
There is a miss on the first read 0x0, and on the first read of the second line, 0x10. In each case one line (4 words) are read from main memory to the cache.

c) **[10 marks]**

This program implements block copy of 6 words from &1000 to &2024.

(all addresses hex)

R1004
W2024
R1008
W2028
R100C
W202C
R1010
W2030
R1014
W2034

In case (i) the index bits are 4:3, in case (ii) they are 5:4. Therefore in case (i) each read and write occupies the same line, but with different tags, and every access is a miss. In case (ii) the reads & writes occupy different lines, and there are two read misses in the 5 reads, one at the start of each read line, and total read hit rate is (5-2)/5=60%. Write hit rate may affect memory performance in either case since although for a write-through cache all cache *writes* incur a memory write, a write miss will also result in the remaining 3 words of the line being updated with new data *read* from memory. Therefore a write-through miss may still take longer than a write-though hit, because of teh necessary memory read.

E1.9 – section B: Operating Systems
Sample model answers to exam questions 2006

Question 1

(a) [bookwork] The MQS scheduling algorithm divides ready processes into separate queues (for example, separate queues for foreground (interactive) and background (batch) processes); each queue has a priority associated with it, and queues with higher priority are given more time-slices for each of their processes. Each queue can have a separate algorithm for scheduling within the queue. Advantages: Flexible – allows fine control of scheduling. Disadvantages: Does not allow for the possibility of processes that change requirements throughout their execution (e.g. a process that started with a long CPU burst, but requires interaction later)

(b) [new computed example] Producer-consumer problem

```
var item, space, mutex: Semaphore;
init(item,0);  init(space,n);   init(mutex,1);
```

Producer process:

```
Procedure producer()
Begin
  While(TRUE) do
    Begin
      Produce_item;
      Wait(space);
      Wait(mutex);
      Write_item;
       Signal(mutex);
       Signal(item);
     End;
End;
```

Consumer process:

```
Procedure producer()
Begin
  While(TRUE) do
    Begin
      Wait(item);
      Wait(mutex);
      get_item;
      Signal(mutex);
      Signal(space);
      Consume_item;
    End;
End;
```

(c) [new computed example]
Optimal page replacement algorithm (5 page faults)

	2	**5**	**1**	**2**	**7**	**5**	**7**	**3**	**5**	**1**
Frame1	**2**	2	2		7			3		
Frame2	**-**	5	5		5			5		
Frame3	**-**	-	1		1			1		

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

	2	5	1	2	7	5	7	3	5	1
Frame1	2	2	2		2	2		3		3
Frame2	-	5	5		7	7		7		1
Frame3	-	-	1		1	5		5		5

(d) [Bookwork] Thrashing: the processor spends more time swapping memory pieces than executing instructions.

QUESTION 2:

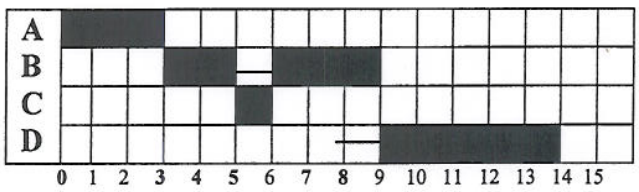
(a) [bookwork] In a microkernel design, Move most OS functionality in user mode processes, leaving only a minimal microkernel in the core of the system. The microkernel passes messages (requests) from a client-process (user program) to a server-process, effectively separating *policy decisions* from *mechanisms*. *Advantages:*

- Ease of extending OS: new services added to user space; microkernel is not modified.
- More security and reliability – if a service fails, the rest of the OS remains untouched.
- Easier to port to different architectures
- Can be easily adapted to distributed environments

Disadvantage: Little consensus on what should be included in the microkernel, but *typically: minimal process and memory management, and a communication facility.*

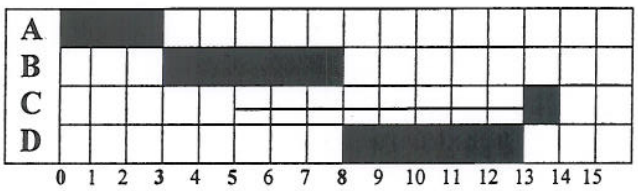
(b)

SRJF



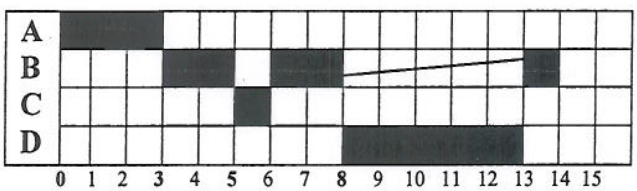
Average waiting time: (0+1+0+1) / 4 = 0.5 ms
Average turnaround time: (3+6+1+6) / 4 = 16 / 4 = 4 ms

Priority Scheduling (without pre-emption)



Average waiting time: (0+0+8+0) / 4 = 2 ms
Average turnaround time: (3+5+9+5) / 4 = 22 / 4 = 5.5 ms

Priority Scheduling (with pre-emption)



Average waiting time: (0+6+0+0) / 4 = 1.5 ms
Average turnaround time: (2+11+1+5) / 4 = 19 / 4 = 4.75 ms

(c) Banker's algorithm works as follows: when a process requests a resource, the algorithm first determines whether granting the request will lead to an unsafe state – if doesn't it grants the request, otherwise the decision is postponed until a process releases some of its resources. To check whether the state is safe, the algorithm:

- checks whether it has some resources to satisfy some process
- The resources of that process are presumed released and added to the available resources
- steps 1 and 2 are repeated until we find that all current processes can be satisfied.

The algorithm's main weakness is that it needs to know the resource requirements for each process in advance.

(d) *Paging* is a memory management scheme that permits the physical address space of a process to be non-contiguous. Physical memory is divided into fixed-sized blocks called *frames*. Logical memory is broken into blocks (of the same size as frames) called pages. Addresses generated by the CPU are now divided into two parts: a page number (p) and a page offset (d): the page number is used as an index into a page table, containing the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address

Advantages: No external fragmentation – any free frame can be allocated to a process that needs it.
Disadvantages: internal fragmentation still possible, since frames are allocated as units.

Demand paging is used to implement virtual memory. Pages are loaded into physical memory only when needed – the corresponding routine of the OS is called a *pager*. When a process is to be swapped in, the pager estimates which pages will be used before the process is swapped out again, and swaps in only those pages

(e) 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.