

Master - June 06

Paper Number(s): **E2.19**

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2006

EEE Part II: MEng, BEng and ACGI

**PRINCIPLES OF COMPUTING AND SOFTWARE ENGINEERING:
INTRODUCTION TO COMPUTER ARCHITECTURE**

Friday 9th June 2006 2:00pm

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.

~~Corrected Copy~~

~~Q/e~~

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:

$$(2.1) \quad a = a_0 + 2^{16}a_1$$

$$(2.2) \quad b = b_0 + 2^{16}b_1$$

The 64 bit product of a and b can be expressed in terms of four $16 \times 16 \rightarrow 32$ bit products as follows:

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

Identity (2.3) may be used to compute an unsigned $32 \times 32 \rightarrow 64$ bit multiply in ARM assembly code using four applications of the ARM $32 \times 32 \rightarrow 32$ 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

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_model1>	Refer to Table Addressing Mode 1
<a_model2>	Refer to Table Addressing Mode 2
<a_model3>	Refer to Table Addressing Mode 3
<a_model4>	Refer to Table Addressing Mode 4
<a_model5>	Refer to Table Addressing Mode 5
<a_model6>	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 = 0xFFFFFFF EOR <Oprnd2>	Architecture 3, 3M and 4 only
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	MUL{cond} {S} Rd, Rm, Rs	N Z	Rd = Rm * Rs	Not in Architecture 1
Multiply	MLA{cond} {S} Rd, Rm, Rs, Rn	N Z	Rd = (Rm * Rs) + Rn	Not in Architecture 1
accumulate	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 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		N Z C		See Table Oprnd2

Operation	Assembler	Action	Notes
Branch with link and exchange instruction set	B{cond} label BX{cond} Rn	R15:= address R14:=R15, R15:= address R15:=Rn, T bit= Rn[0]	Architecture 4 with Thumb only Thumb state; Rn[0] = 0 ARM state; Rn[0] = 1
Load	<p>Word with user-mode privilege</p> <p>Byte with user-mode privilege signed</p> <p>Halfword signed</p> <p>Multiple Block data operations Increment Before Increment After Decrement Before Decrement After Stack operations and restore CPSR User registers</p>	<p>Rd:= [address]</p> <p>Rd= [byte value from address] Loads bits 0 to 7 and sets bits 8-31 to 0</p> <p>Rd= [signed byte value from address] Loads bits 0 to 7 and sets bits 8-31 to bit 7</p> <p>Rd= [halfword value from address] Loads bits 0 to 15 and sets bits 16-31 to 0</p> <p>Rd= [signed halfword value from address] Loads bits 0 to 15 and sets bits 16-31 to bit 15</p> <p>Stack manipulation (pop)</p>	<p>Architecture 4 only</p> <p>Architecture 4 only</p> <p>Architecture 4 only</p> <p>I sets the W bit (updates the base register after the transfer) ^ sets the S bit</p> <p>I sets the W bit (updates the base register after the transfer)</p>
Store	<p>Word with user-mode privilege</p> <p>Byte with user-mode privilege</p> <p>Halfword</p> <p>Multiple Block data operations Increment Before Increment After Decrement Before Decrement After Stack operations User registers</p>	<p>[address]= Rd</p> <p>[address]= byte value from Rd</p> <p>[address]= halfword value from Rd</p> <p>Stack manipulation (push)</p>	<p>Architecture 4 only</p> <p>I sets the W bit (updates the base register after the transfer) ^ sets the S bit</p>
Swap	SWP{cond} Rd, Rn, [Rn] SWP{cond}B Rd, Rn, [Rn]		Not in Architecture 1 or 2 Not in Architecture 1 or 2
Coprocessors	<p>Data operations</p> <p>Move to ARM reg from coproc</p> <p>Move to coproc from ARM reg</p> <p>Load</p> <p>Store</p>	<p>CDP{cond} p<cpnum>, <op1>, CRd, CRn, CRm, <op2></p> <p>MRC{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2></p> <p>MCR{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2></p> <p>LDC{cond} p<cpnum>, CRd, <a_mode6></p> <p>STC{cond} p<cpnum>, CRd, <a_mode6></p>	Not in Architecture 1
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, RRR
Pre-indexed offset	[Rn], # +/- 12_Bit_Offset
Immediate	[Rn], +/- Rm
Register	[Rn], +/- Rm, LSL #shift_imm
Scaled register	[Rn], +/- Rm, LSR #shift_imm [Rn], +/- Rm, ASR #shift_imm [Rn], +/- Rm, ROR #shift_imm [Rn], +/- Rm, RRR
Post-indexed offset	[Rn], # +/- 12_Bit_Offset
Immediate	[Rn], +/- Rm
Register	[Rn], +/- Rm, LSL #shift_imm
Scaled register	[Rn], +/- Rm, LSR #shift_imm [Rn], +/- Rm, ASR #shift_imm [Rn], +/- Rm, ROR #shift_imm [Rn], +/- Rm, RRR

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

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 RRR

Field	Sets
Control field mask bit	(bit 3)
Flags field mask bit	(bit 0)
Status field mask bit	(bit 1)
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

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}
STMEQB 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

ARM REFERENCE NOTES

2005/2006

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	IST	Test	Sec on Rn AND Op2
1001	ITEQ	Test equivalence	Sec on Rn - Op2
1010	CMPE	Compare	Sec on Rn + Op2
1011	CMN	Compare negated	Sec 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

R2.4

Op-codes
AND
ANDEQ
ANDS
ANDEQS
S=> set flags

Data Processing Operand 2

Examples

ADD r0, r1, op2
MOV r0, op2

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,lsr,asr,asr,ror	rrx always sets carry ror sets carry if S=1 shifts do not set carry
Rm, shift Rs	shift => lsr,lsr,asr,asr,ror	shift by register value (takes 2 cycles)

R2.5

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
; assembles to instructions that set r0 to immediate
LDR r0, =numb ; value numb – numb may be too large for a MOV operand
;

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

R2.7

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
 - 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 r1, rh, rm, rs unsigned multiply (Rh:Rl) := Rm*Rs
UMLAL r1, rh, rm, rs unsigned multiply-acc (Rh:Rl) := (Rh:Rl)+Rm*Rs
SMULL r1, rh, rm, rs signed multiply (Rh:Rl) := Rm*Rs
SMLAL r1, rh, rm, rs signed multiply-acc (Rh:Rl) := (Rh:Rl)+Rm*Rs

ARM7DM core and above

ISE:1/EE2 Introduction to Computer Architecture

tpvc - 4-Jan-06

2.6

Exceptions & Interrupts

Exception	Return
SWI or undefined instruction	MOV5 pc, R14
IRQ, FIQ, prefetch abort	SUBS pc, r14, #4
Data abort (needs to rerun failed instruction)	SUBS pc, R14, #8

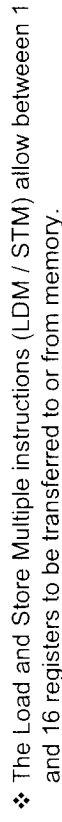
Exception Mode	Shadow registers
SVC,UND,IRQ,Abort	R13, R14, SPSR
FIQ	as above + R8-R12

(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

R2.8

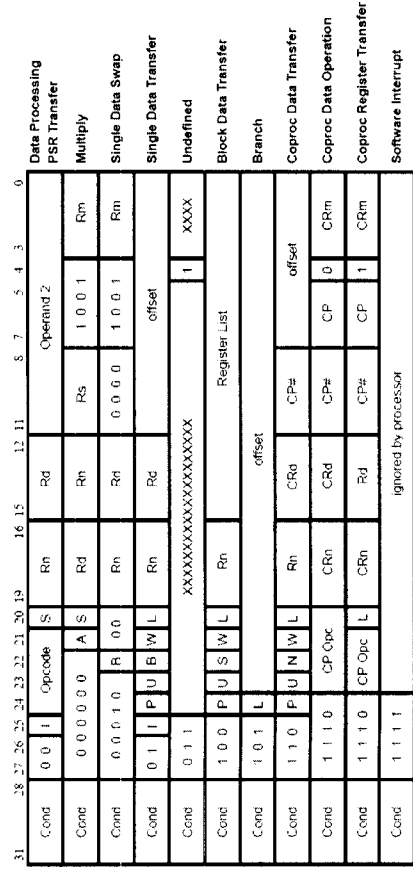
Multiple Register Transfer Binary Encoding



Instruction Set Overview

$$B\{\langle \text{cond} \rangle\} \text{ label}$$

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.

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)