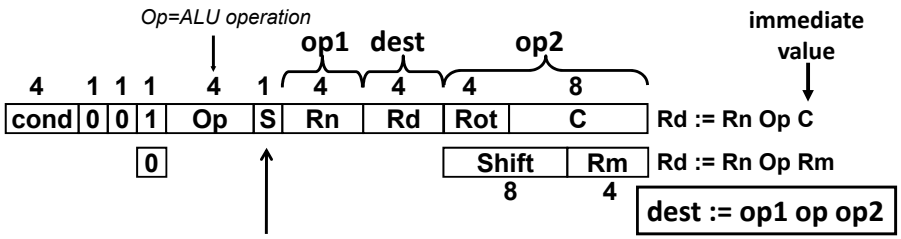
Data processing (ADD,SUB,AND,CMP,MOV, etc)



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

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

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

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

Op-codes

AND	
ANDEQ	EQ,NE,… => Condition
ANDS	S => set status on result
ANDSEQ	note position of S

Op	Assembly	Operation	Pseudocode
0000	AND Rd,Rn,op2	Bitwise logical AND	Rd := Rn AND op2
0001	EOR Rd,Rn,op2	Bitwise logical XOR	Rd := Rn XOR op2
0010	SUB Rd, Rn, op2	Subtract	Rd := Rn - op2
0011	RSB Rd, Rn, op2	Reverse subtract	Rd := op2 - Rn
0100	ADD Rd,Rn,op2	Add	Rd := Rn + op2
0101	ADC Rd,Rn,op2	Add with carry	Rd := Rn + op2 + C
0110	SBC Rd, Rn, op2	Subtract with carry	Rd := Rn - op2 + C - 1
0111	RSC Rd, Rn, op2	Reverse sub with C	Rd := op2 - Rn + C - 1
1000	TST Rn, op2	set NZ on AND	Rn AND op2
1001	TEQ Rn, op2	set NZ on EOR	Rn EOR op2
1010	CMP Rn, op2	set NZCV on subtract	Rn - op2
1011	CMN Rn, op2	set NZCV on add	Rn + op2
1100	ORR Rd,Rn,op2	Bitwise logical OR	Rd := Rn OR op2
1101	MOV Rd, op2	Move	Rd := op2
1110	BIC Rd,Rn,op2	Bitwise clear	Rd := Rn AND NOT op2
1111	MVN Rd,op2	Bitwise move invert	Rd := NOT op2

Data Processing Op2

ADD r0, r1, op2
MOV r0, op2

Examples

ADD r0, r1, r2
MOV r0, #0x100
CMP r0, #-120
EOR r0, r1, r2, LSR #10
RSB r0, r1, r2, ASR r3
MOVS r0, r0, RRX

Op2	Conditions	Notes
Rm		r15=pc, r14=lr, r13=sp
#imm	*imm* = s * 2^n (0 ≤ s ≤ 255, if n is even) (0 ≤ s ≤ 127, if n is odd)	Assembler will translate negative *imm* values into positive changing op-code as necessary Assembler will work out s,n from *imm* if they exist, or give error.
Rm, shift #s Rm, rrx	(1 ≤ s ≤ 31) shift => lsr,lsl,asr,asl,ror	Shifts/rotates write carry if S = 1 and there is no arithmetic/compare. Carry is last bit shifted/rotated off end of word.
Rm, shift Rs	shift => lsr,lsl,asr,asl,ror	shift Rm by no of bits equal to register Rs value (takes 2 cycles)

Multiply in detail

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

Register operands only
No constants, no shifts

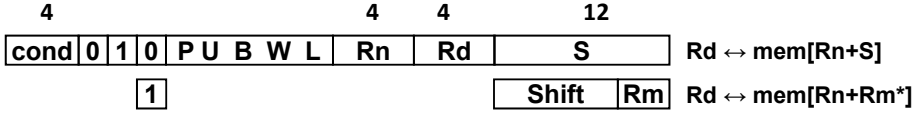
NB d & m must be different for MUL, MLA

ARM3 and above

MUL	rd, rm, rs	multiply (32 bit)	Rd := (Rm*Rs)[31:0]
MLA	rd,rm,rs,rn	multiply-acc (32 bit)	Rd:= (Rm*Rs)[31:0] + Rn
UMULL	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 (64 bit multiply result)

Data transfer (to or from memory LDR,STR)



Rm* = Rm shifted left by constant (scaled)

Bit in word	0	1
U	subtract offset [Rn-S]	add offset [Rn+S]
B	Word	Byte
L	Store	Load

P bit	W bit	address	Rn
0	0	[Rn]	Rn := Rn+S
0	1	not allowed	NB S = 0 gives Rn no change
1	0	[Rn+Rm] or [Rn+S]	no change
1	1	[Rn+Rm] or [Rn+S]	Rn := Rn+Rm or Rn := Rn+S

Data Transfer Instructions

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

```
; all opcodes below can have B suffix to indicate byte transfer or address

LDR     r0, [r1]            ; register-indirect addressing (r1)
LDR     r0, [r1, #offset]   ; pre-indexed addressing (r1 + offset)
LDR     r0, [r1, #offset]!  ; pre-indexed, auto-indexing (r1 + offset , r1 := r1+offset)
LDR     r0, [r1], #offset   ; post-indexed, auto-indexing (r1, r1 := r1+offset)
LDR     r0, [r1, r2]        ; register-indexed addressing (r1 + r2)
LDR     r0, [r1, r2, lsl #shift]  ;scaled register-indexed addressing (r1 + r2 * 2^shift)
LDR     r0, address_label ; PC relative addressing (pc+8 is read, offset calculated)
ADR     r0, address_label ; load PC relative address (pc+8 is read, offset calculated)
```

```
LDMED   r13!, {r0-r4,r6,r7} ;   ! => write-back to address register
STMFA   r13,  {r2}           ;   no write-back
STMEQIB r2!,  {r5-r12}       ;   note position of EQ or other condition
        higher register nos transfer to/from higher mem addresses
    [E|F][A|D]  Empty|Full,        Ascending|Descending
    [I|D][A|B]  Increment|Decrement,   After|Before
```

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

Instruction Timing

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

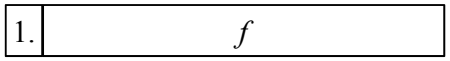
Instruction	Typical execution time (cycles)
Any instruction, with condition false	1
data processing (except register-valued shifts)	1 (+3 if Rd = R15)
data processing (register-valued shifts): MOV R1, R2, lsl R3	2 (+3 if Rd = R15)
LDR,LDRB, STR, STRB	4 (+3 more if Rd = R15)
LDM (n registers)	n+3 (+3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

IEEE 754

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

s, e, f are all **unsigned** binary fields

f defines number in range 1-2:
e defines binary multiplier or divisor (e-127)



exp	frac	meaning	notes
0	0	+/- 0	The smallest non-zero numbers are used to mean 0, otherwise there is no exact 0!
255	0	+/- ∞	The largest exponent is used for special cases
255	≠ 0	NaN	not a number, used for 0/0 etc

Shadow registers

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

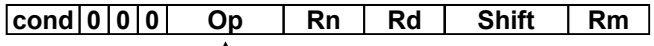
BL, SWI	R14 = Return address
IRQ,FIQ,UND	R14 = Return address + 4
Abort	R14 = Return address + 8

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

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

Machine Instruction Overview (1)

Data processing (ADD,SUB,CMP,MOV)



Rd := Rn Op Rm*

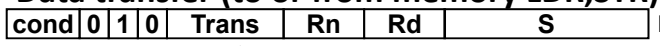
Rd := Rn Op S

ALU operation

Rm* = Rm with optional shift

multiply instructions are special case

Data transfer (to or from memory LDR,STR)

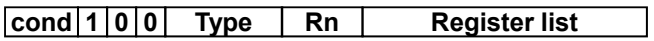


Rd ↔ mem[Rn+/-S]

Rd ↔ mem[Rn+/-Rm*]

Byte/word, load/store, etc

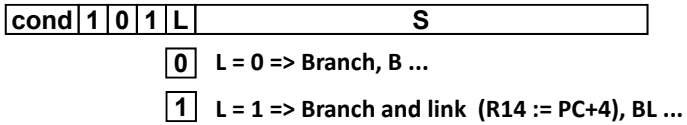
Multiple register transfer



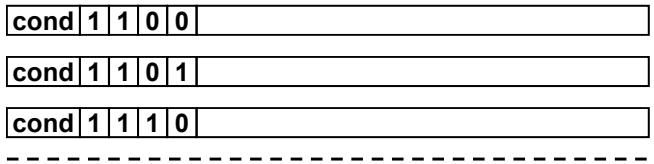
Transfer registers to/from stack

Overview (2)

Branch B, BL, BNE, BMI…

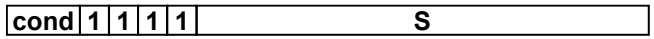


PC := PC+8+4*S
S is sign extended
NB +8 because of pipelining.



coprocessor interface

Software Interrupt (SWI)



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

ARM UAL Assembler Reference

```
<label>       <directive>   <operands> <; comment>  all fields are optional
<label>       <op-code>     <operands> <; comment>  all fields are optional
```

```
        DCD     c1, c2, c3, …     ; load memory with  consecutive 32 bit words
        DCB     b1, b2, 's'       ; load memory with consecutive bytes
SYM     EQU     TABLE + 4         ; set label SYM equal to constant expression
        END                       ; end of code section
```

constant	notes
1234	Decimal
0x10fc	Hex
0b101110	binary
's'	Single character
"cat"	Sequence of characters: LSB first (not VisUAL)
LABEL	Symbols are numeric constants, usually addresses

Symbols are case sensitive: Loop and LOOP are different

Mnemonics, **directives**, and shift, register names are case insensitive but must be all upper or lower case

ASCII code

Bits b_4 b_3 b_2 b_1	Row ↓ / Column → 0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	10	LF	SUB	*	:	J	Z	j	z
1 0 1 1	11	VT	ESC	+	;	K	[	k	{
1 1 0 0	12	FF	FC	,	<	L	\	l	|
1 1 0 1	13	CR	GS	-	=	M	]	m	}
1 1 1 0	14	SO	RS	.	>	N	^	n	~
1 1 1 1	15	SI	US	/	?	O	_	o	DEL