Data Processing (5A)

Young Won Lim
2/8/20

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Young Won Lim
2/8/20

Based on

ARM System-on-Chip Architecture, 2nd ed, Steve Furber

Introduction to ARM Cortex-M Microcontrollers
– Embedded Systems, Jonathan W. Valvano

Data Processing

1101	MOV	MOV	Rd := Op2
1111	MVN	Move Negated	Rd := NOT Op2
1000	TST	Test	*set condition codes on* Rn AND Op2
1001	TEQ	Test Equivalence	*set condition codes on* Rn EOR Op2
1010	CMP	Compare	*set condition codes on* Rn – Op2
1011	CMN	Compare Negated	*set condition codes on* Rn + Op2
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	RSUB	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
1100	ORR	Logical Bit-wise OR	Rd := Rn OR Op2

https://community.arm.com/processors/b/blog/posts/condition-codes-1-condition-flags-and-codes

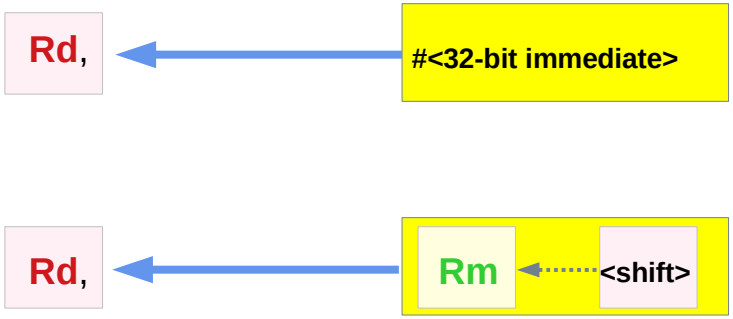
Data Processing Type 1 – no 1st operand **Rn**

No 1st operand Rn

<op_type1> {<cond>} {S} **Rd**, **#<32-bit immediate>** *… 12-bit encoded*

Rm, {<shift>}

MOV	Move
MVN	Move Negated



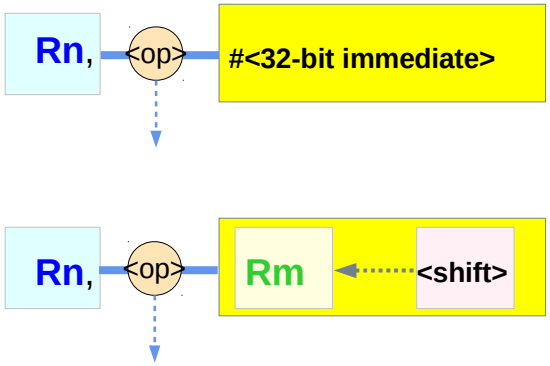
Data Processing Type 2 – no destination **Rd**

No destination Rd

<op_type2> {<cond>} **Rn**, **#<32-bit immediate>** *… 12-bit encoded*

Rm, {<shift>}

CMP	Compare
CMN	Compare Negated
TST	Test
TEQ	Test Equivalence



S is implicitly implied

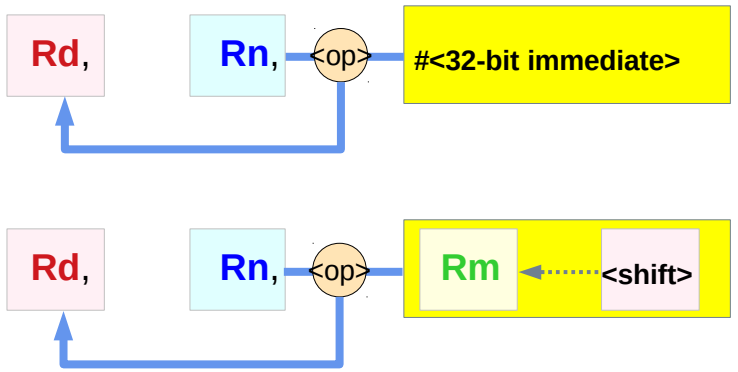
Data Processing Type 3 – Arithmetic & Logical Instructions

Both Rd and Rn

<op> {<cond>} {S} **Rd**, **Rn**, **#<32-bit immediate>** *… 12-bit encoded*

Rm, {<shift>}

AND	logical bit-wise AND
EOR	logical bit-wise Exclusive OR
SUB	Subtract
RSB	Reverse Subtract
ADD	Add
ADC	Add with Carry
SBC	Subtract with Carry
RSC	Reverse Subtract with Carry
ORR	logical bit-wise OR
BIC	Bit Clear



Data Processing Format

No 1st operand Rn

<op_type1> {<cond>} {S} **Rd**, **#<32-bit immediate>** *… 12-bit encoded*
Rm, {<shift>}

No destination Rd

<op_type2> {<cond>} **Rn**, **#<32-bit immediate>** *… 12-bit encoded*
Rm, {<shift>}

Both Rd and Rn

<op> {<cond>} {S} **Rd**, **Rn**, **#<32-bit immediate>** *… 12-bit encoded*
Rm, {<shift>}

<op2>

Operand2 **<op2>**

Operand2 **<op2>** is the flexible second operand to most instructions.

An immediate value

4-bit rotate + 8-bit immediate … *12-bit encoded*

An 8-bit number rotated right by an even number of places.

A register shifted by value

immediate shift amount : a 5-bit unsigned integer

A register shifted by register

register shift amount : the lower 8 bits of a register

http://www.davespace.co.uk/arm/introduction-to-arm/operand2.html

Operand2 **<op2>** Examples

Immediate values

```
MOV r0, #42                ; Move the value 42 into R0

ORR r1, r1, #0xFF00        ; OR the value 0xFF00 with R1
```

Registers shifted by values

```
MOV r2, r2, LSR #1         ; Shift R2 right by one bit

RSB r10, r5, r14, ASR #14  ; Shift R14 right by 14 bits while sign extending,
                             then subtract R5 from that. Put the result in R10.
                           (RSB = Reverse Subtract)
```

Registers shifted by registers

```
BIC r11, r11, r1, LSL r0   ; Take R1 and shift it left by R0, then use that as a
                             mask to clear bits in R11. Put the result in R11

CMP r9, r8, ROR r0         ; Take R8 and rotate it right by R0, then compare
                             that with R9. The result is the processor flags
```

http://www.davespace.co.uk/arm/introduction-to-arm/operand2.html

Data Processing Instrutioncs with a Shift Operand

Rm, <shift>

<op>	{<cond>} {S}	**Rd**,	**Rn**,	**Rm**, {**<shift>**}	AND,EOR,SUB,RSB,ADD,ADC,SBC,RSC,ORR,BIC
<op_type1>	{<cond>} {S}	**Rd**,		**Rm**, {**<shift>**}	MOV, MVN … special case
<op_type2>	{<cond>}		**Rn**,	**Rm**, {**<shift>**}	CMP, CMN, TST,TEQ … special case

<shift>

<shift type>	**# <#shift>**	…. instruction-specified shift amount
<shift type>	**Rs**	…. register-specified shift amount

LSL, **ASL**, **LSR**, **ASR**, **ROR**

<shift type>		…. no shift amount

RRX

12-bit immediate value encoding

the 12-bit immediate value
not as a 12-bit number.
but an 4-bit rotation with a 8-bit number

[24-bit padding zeros + 8-bit number] : a full 32-bit word

the 4-bit rotation value has 2^4=16 possible settings
16 possible rotations of 8-bit number in the 32-bit word
(0, 1, 2, …, 15) * 2
(0, 2, 4, …, 30) : even number of rotations

first, the 8-bit number is zero-padded to form a 32-bit number
then rotate right the 32-bit number by 4-bit rotation * 2

https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/

Immediate value encoding example

........................76543210	0	0000
0........................7654321	1	
10.......................765432	2	0001
210......................76543	3	
3210.....................7654	4	0010
43210....................765	5	
543210...................76	6	0011
6543210..................7	7	
76543210.................	8	0100
.76543210................	9	
..76543210...............	10	0101
...76543210..............	11	
....76543210.............	12	0110
.....76543210............	13	
......76543210...........	14	0111
.......76543210..........	15	
........76543210.........	16	1000
.........76543210........	17	
..........76543210.......	18	1001
...........76543210......	19	
............76543210.....	20	1010
.............76543210....	21	
..............76543210...	22	1011
...............76543210..	23	
................76543210.	24	1100
.................76543210	25	
..................76543210	26	1101
...................76543210	27	
....................76543210	28	1110
.....................76543210	29	
......................76543210	30	1111
.......................76543210	31	

Set, Clear, Toggle bits of 32-bit word

The rotated byte encoding allows the 12-bit value
to represent a much more useful set of numbers than just 0–4095.

ARM immediate values can represent any power of 2 from 0 to 31.
So you can **set**, **clear**, or **toggle** any bit with one instruction:

```
ORR r5, r5, #&8000      ; Set bit 15 of r5
BIC r0, r0, #&20        ; ASCII lower-case to upper-case
EOR r9, r9, #&80000000 ; Toggle bit 31 of r9
```

can specify a byte value at any of the four locations in the word:

```
AND r0, r0, #&ff000000 ; Only keep the top byte of r0
```

https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/

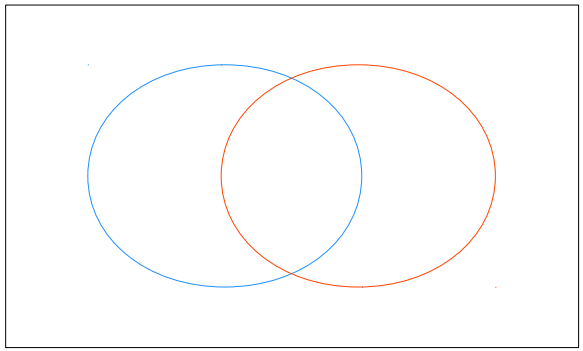
Usefulness of a 12-bit immediate number

In practice, this encoding gives a lot of values
that would not be available otherwise.

Large loop termination values,
bit selections and masks, and
lots of other weird constants are all available.

reuse the idle barrel shifter
to allow a wide range of useful numbers.

simple 12-bit



8-bit number +
4-bit rotation

https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/

Loading immediate value using MOV, MVN

```
MOV  r1, &00003C00
MOV r2, &00000DC0
MOV r3, &00000004
```

```
0000_0000_0000_0000_0011_1100_0000_0000    r1
0000_0000_0000_0000_0000_1101_1100_0000    r2
0000_0000_0000_0000_0000_0000_0000_0100    r3
```

```
&00003C00 = 3C >> 24 … 12-bit encoding is ok (12, 3C)
&00000DC0 = DC >> 28 … 12-bit encoding is ok (14, DC)
&00000004 = 04 >> 0 … 12-bit encoding is ok (0, 04)
```

If you write an instruction with an immediate value that is not available,
the assembler reports the error:
Immediate n out of range for this operation.

In such a case, try this

```
LDR r1, =number
LDR rd, [r1]
```

r1 contains the address of constant
rd contains the number in the literal pool

Logical and Shift Operation Examples

```
MOV  r1, &00003C00
MOV r2, &00000DC0
MOV r3, &00000004

AND  r5, r1, r2
ORR  r5, r1, r2
MVN  r5, r1
MOV  r6, r5
ADD  r5, r1, r2, LSL #2
MOV  r6, r5, LSR #4
MOV  r6, r5, LSR r3
```

```
0000_0000_0000_0000_0011_1100_0000_0000    r1
0000_0000_0000_0000_0000_1101_1100_0000    r2
0000_0000_0000_0000_0000_0000_0000_0100    r3

0000_0000_0000_0000_0000_1100_0000_0000    r5 = r1 & r2
0000_0000_0000_0000_0011_1101_1100_0000    r5 = r1 | r2

1111_1111_1111_1111_1100_0011_1111_1111    r5 = ~r1
1111_1111_1111_1111_1100_0011_1111_1111    r6 = r5


0000_0000_0000_0000_0011_1100_0000_0000    r1
0000_0000_0000_0000_0011_0111_0000_0000    r2 << 2
0000_0000_0000_0000_0111_0011_0000_0000    r5 = r1 + (r2 << 2)

0000_0000_0000_0000_0000_0111_0011_0000    r6 = (r5 >> 4)
0000_0000_0000_0000_0000_0111_0011_0000    r6 = (r5 >> r3)
```

Logical Operations

```
AND{S}{cond} {Rd,} Rn, <op2>      ; Rd = Rn & op2
ORR{S}{cond} {Rd,} Rn, <op2>      ; Rd = Rn | op2
EOR{S}{cond} {Rd,} Rn, <op2>      ; Rd = Rn ^ op2
BIC{S}{cond} {Rd,} Rn, <op2>      ; Rd = Rn &(~op2)
ORN{S}{cond} {Rd,} Rn, <op2>      ; Rd = Rn | (~op2)
```

<op2> :

#n	32-bit immediate … *12-bit encoded*
Rm	no shift
Rm, sh_type, #n	shift operand, instruction specified (#n)
Rm, sh_type, Rs	shift operand, register specified (Rs)

Logical Operation without Rd

If **Rd** is omitted, the result will be stored into **Rn**

```
AND{S}{cond}     Rn, <op2>     ; Rn = Rn & op2
ORR{S}{cond}     Rn, <op2>     ; Rn = Rn | op2
EOR{S}{cond}     Rn, <op2>     ; Rn = Rn ^ op2
BIC{S}{cond}     Rn, <op2>     ; Rn = Rn &(~op2)
ORN{S}{cond}     Rn, <op2>     ; Rn = Rn | (~op2)
```

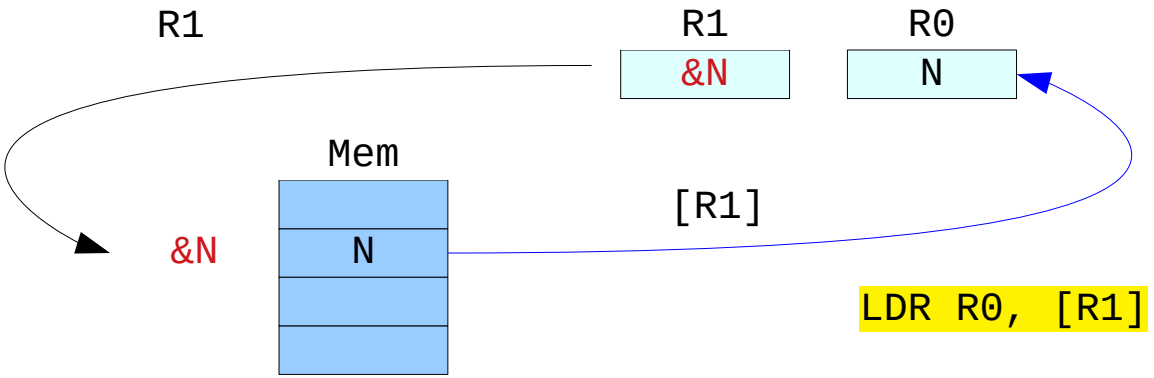
Syntactic Sugar

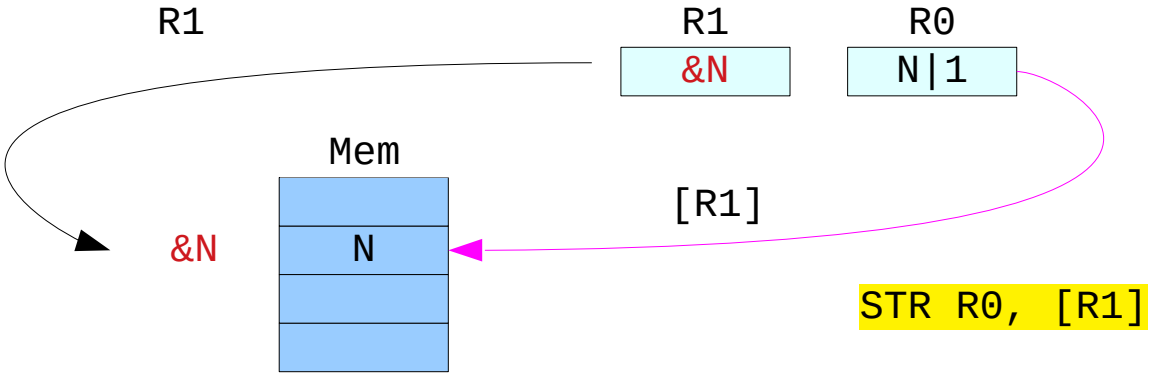
```
AND{S}{cond} Rn, Rn, <op2>     ; Rn = Rn & op2
ORR{S}{cond} Rn, Rn, <op2>     ; Rn = Rn | op2
EOR{S}{cond} Rn, Rn, <op2>     ; Rn = Rn ^ op2
BIC{S}{cond} Rn, Rn, <op2>     ; Rn = Rn &(~op2)
ORN{S}{cond} Rn, Rn, <op2>     ; Rn = Rn | (~op2)
```

ARM Cortex-M Microcontroller

Logical operation example

&N the address of the location
where the value N is stored





Shift Operations

```
LSR{S}{cond} Rd, Rm, Rs        ; Rd ← Rm >> Rs (unsigned)
LSR{S}{cond} Rd, Rm, #n        ; Rd ← Rm >> #n (unsigned)
ASR{S}{cond} Rd, Rm, Rs        ; Rd ← Rm >> Rs (signed)
ASR{S}{cond} Rd, Rm, #n        ; Rd ← Rm >> Rs (signed)
LSL{S}{cond} Rd, Rm, Rs        ; Rd ← Rm << Rs (both)
LSL{S}{cond} Rd, Rm, #n        ; Rd ← Rm << Rs (both)
ROR{S}{cond} Rd, Rm, Rs
ROR{S}{cond} Rd, Rm, #n
RXX{S}{cond} Rd, Rm
```

Shift Operations with **MOV**

LSR{S}{cond} Rd, Rm, Rs	=	MOV{S}{cond} Rd, Rm, LSR Rs
LSR{S}{cond} Rd, Rm, #n	=	MOV{S}{cond} Rd, Rm, LSR Rs
ASR{S}{cond} Rd, Rm, Rs	=	MOV{S}{cond} Rd, Rm, ASR Rs
ASR{S}{cond} Rd, Rm, #n	=	MOV{S}{cond} Rd, Rm, ASR Rs
LSL{S}{cond} Rd, Rm, Rs	=	MOV{S}{cond} Rd, Rm, LSL Rs
LSL{S}{cond} Rd, Rm, #n	=	MOV{S}{cond} Rd, Rm, LSL Rs
ROR{S}{cond} Rd, Rm, Rs	=	MOV{S}{cond} Rd, Rm, ROR Rs
ROR{S}{cond} Rd, Rm, #n	=	MOV{S}{cond} Rd, Rm, ROR Rs
RXX{S}{cond} Rd, Rm	=	MOV{S}{cond} Rd, Rm, RXX

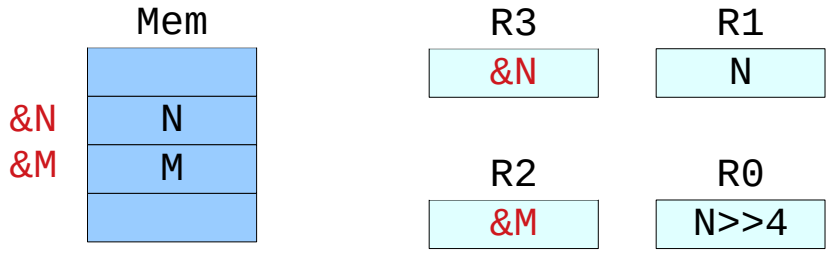
Syntactic Sugar

In ARM, shift operations are not separate instructions
ARM allows the second operand to be shifted
as a part of data processing instruction

ARM Cortex-M Microcontroller

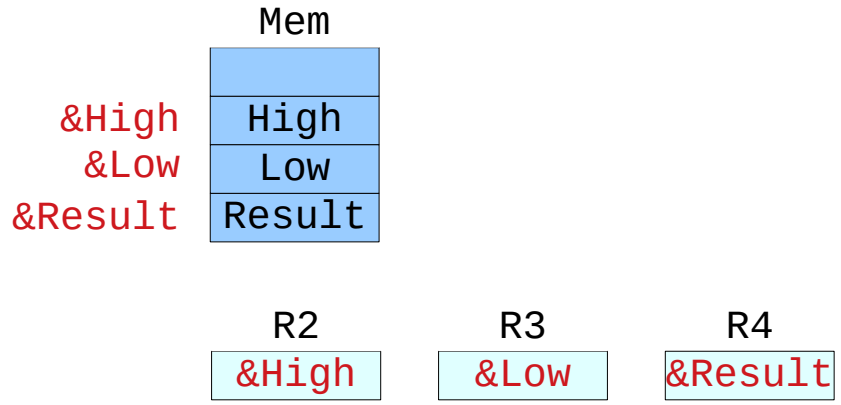
Shift Operation Example (1)

```
LDR R3, =N
LDR R1, [R3]
LSR R0, R1, #2
LDR R2, =M
STR R0, [R2]
```



Shifting Operation Example (2)

```
LDR  R2, =High
LDR  R3, =Low
LDR  R4, =Result
```




```
LDRB R1, [R2]
LSL  R0, R1, #4          ; R0 ← LSB(R1) << 4
LDRB R1, [R3]
ORR  R0, R0, R1          ; R0 ← R0 | R1
STRB R0, [R4]
```

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf
[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf