Logical Operations

- Logical operations perform operations on the bits themselves, rather than the values they represent
  - e.g. and, or, exclusive-or, not (invert)
- Truth tables

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

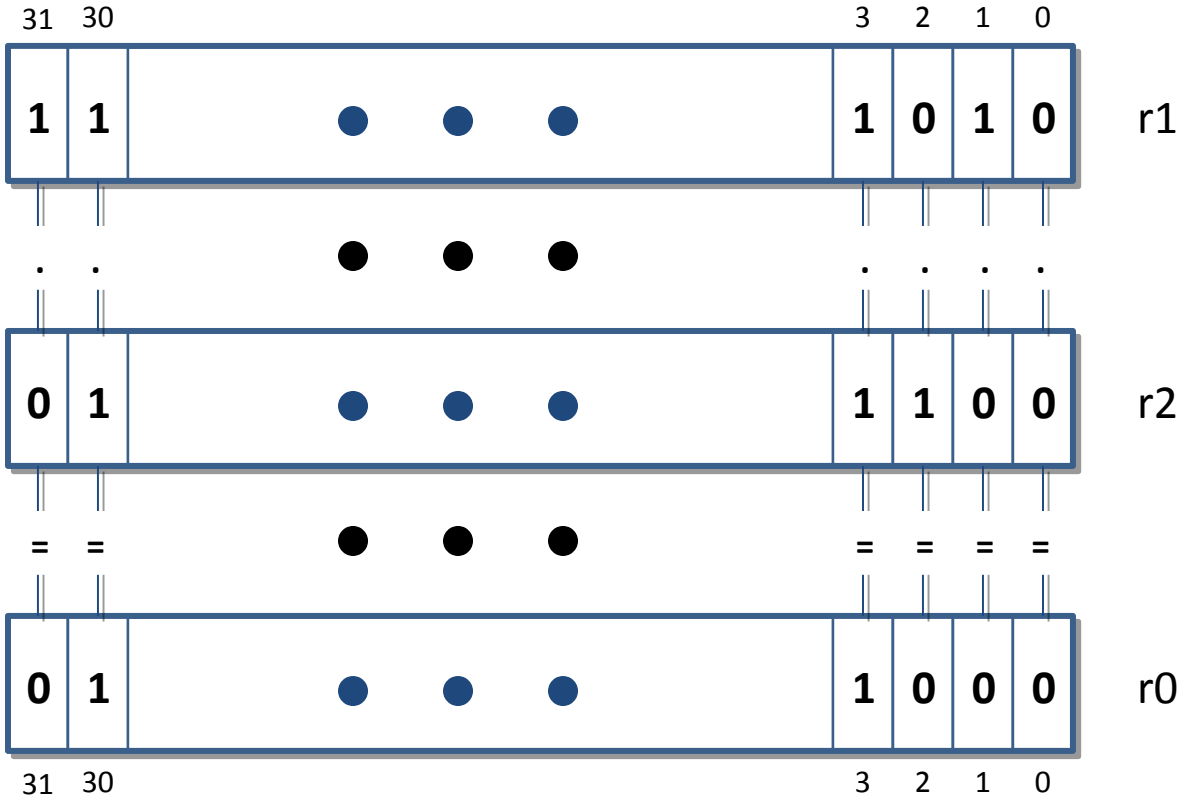
A	B	A EOR B
0	0	0
0	1	1
1	0	1
1	1	0

A	NOT A
0	1
1	0

Instructions for Logical Operations

- Bitwise logical AND – AND

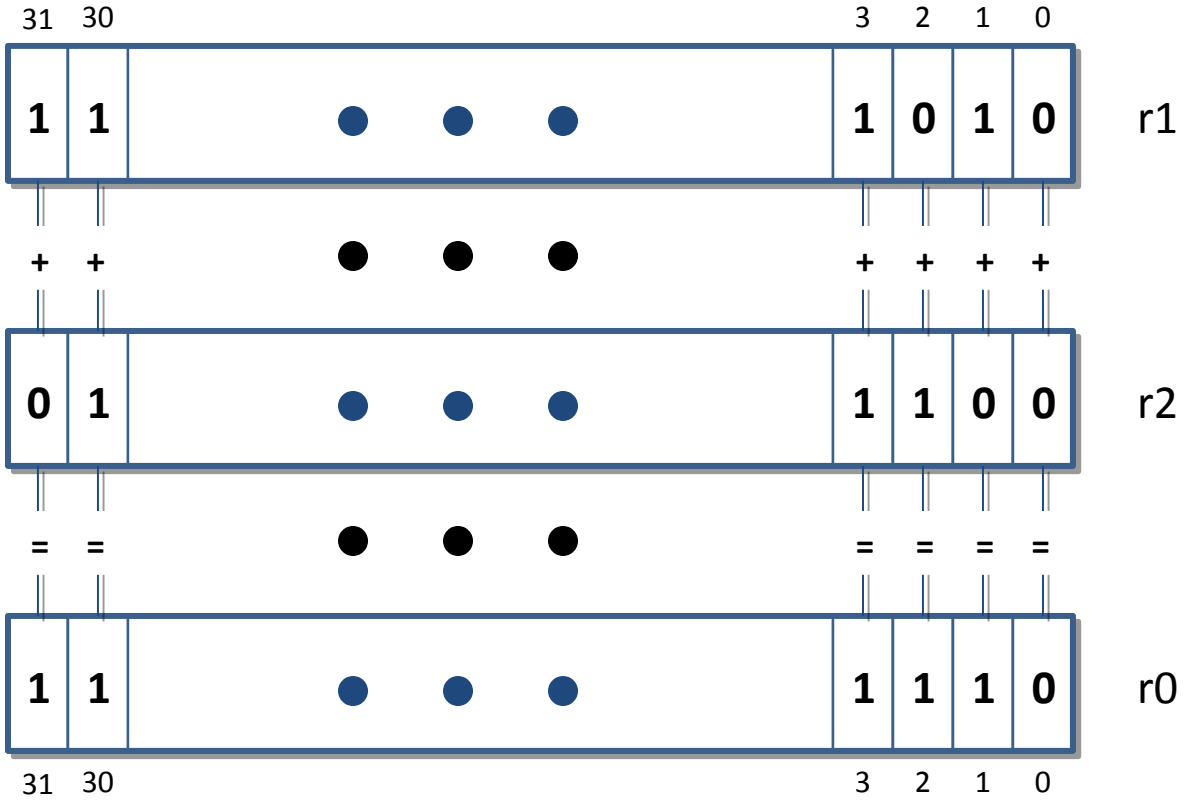
```
AND     r0, r1, r2      @ r0 = r1 . r2 (r1 AND r2)
```



Instructions for Logical Operations

- Bitwise logical OR – ORR

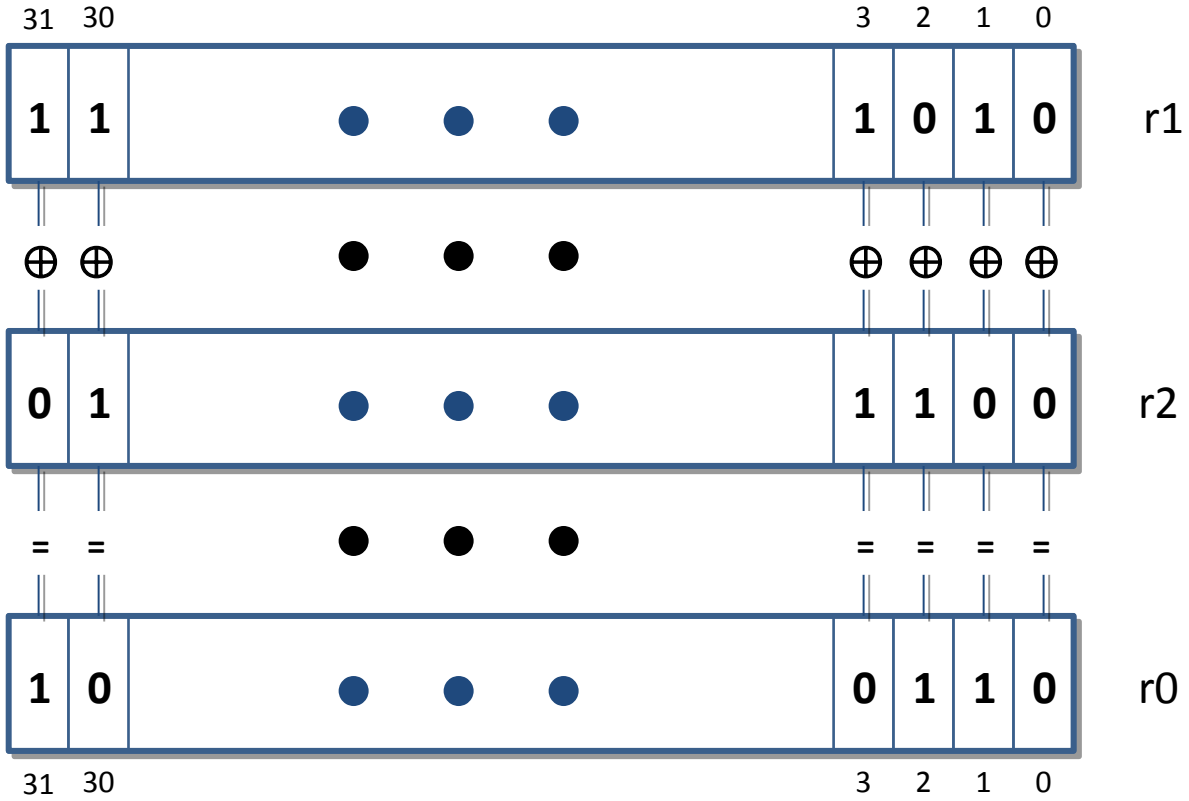
```
ORR       r0, r1, r2       @ r0 = r1 + r2 (r1 OR r2)
```



Instructions for Logical Operations

- Bitwise logical Exclusive OR – EOR

```
EOR       r0, r1, r2       @ r0 = r1 ⊕ r2 (r1 EOR r2)
```

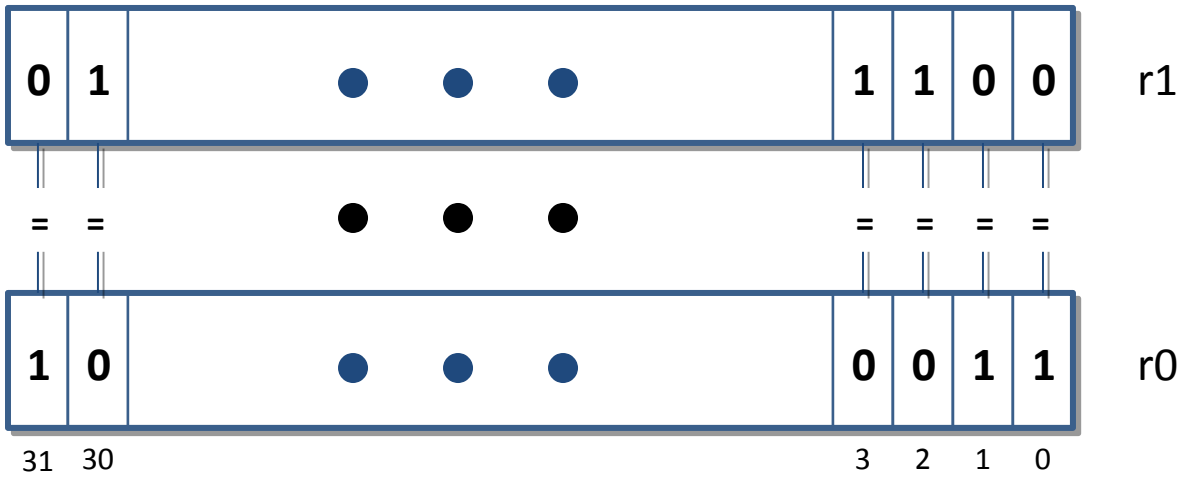


Instructions for Logical Operations

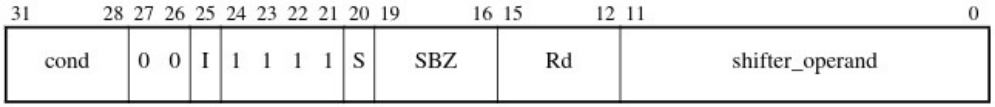
- Bitwise logical inversion
- MVN **M**o**V**e **N**egative – like MOV but moves the one's complement of a value (bitwise inversion) to a register

```
MVN     r0, r0          @ r0 = r0' (NOT r0)
```

```
MVN     r0, r1          @ r0 = r1' (NOT r1)
```



A4.1.41 MVN



MVN (Move Not) generates the logical ones complement of a value. The value can be either an immediate value or a value from a register, and can be shifted before the MVN operation.

MVN can optionally update the condition code flags, based on the result.

Syntax

```
MVN{<cond>}{S}  <Rd>, <shifter_operand>
```

where:

<cond> Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If <cond> is omitted, the AL (always) condition is used.

S Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If S is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when S is specified:

- If <Rd> is not R15, the N and Z flags are set according to the result of the operation, and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.
- If <Rd> is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

<Rd> Specifies the destination register.

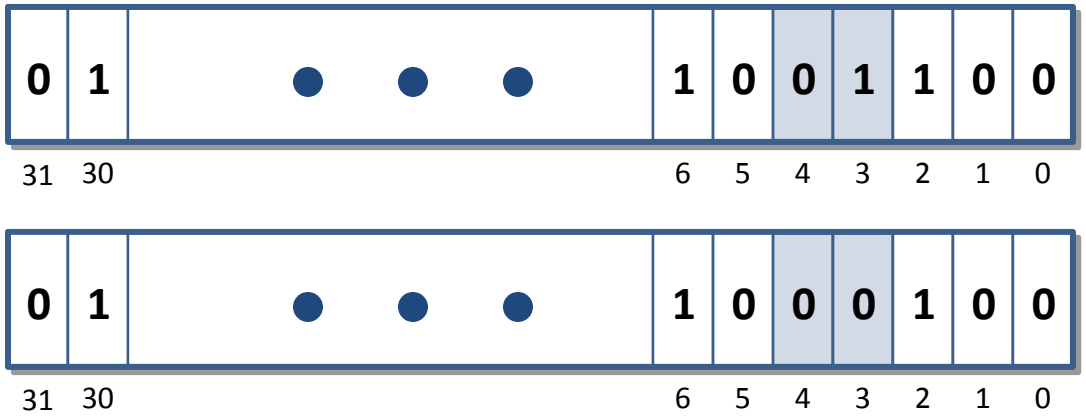
<shifter_operand>

Specifies the operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

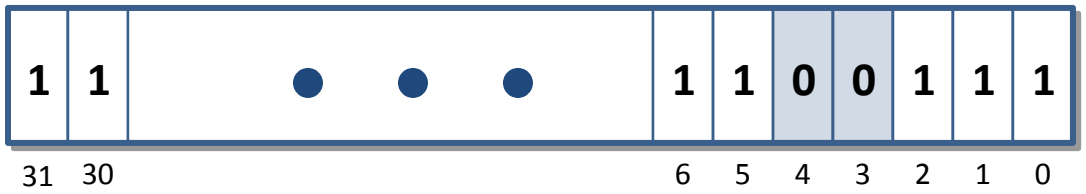
If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not MVN. Instead, see *Extending the instruction set* on page A3-32 to determine which instruction it is.

Bit Manipulation

- e.g. Clear bits 3 and 4 of the value in r1



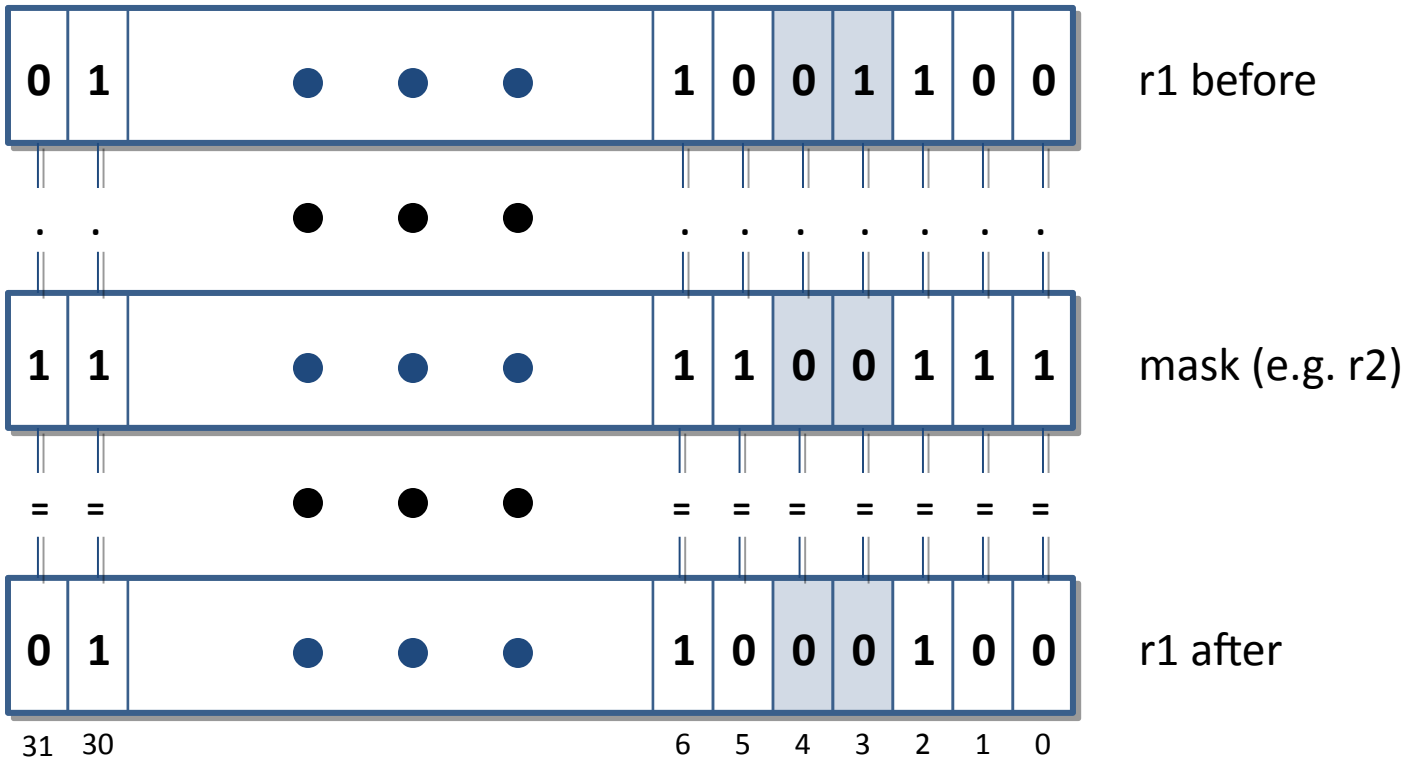
  - Observe $0 \cdot x = 0$ and $1 \cdot x = x$
  - Construct a mask with 0 in the bit positions we want to clear and 1 in the bit positions we want to leave unchanged



  - Perform a bitwise logical AND of the value with the mask

Bit Manipulation

- e.g. Clear bits 3 and 4 of the value in r1 (continued)



Program 5.1 – Clear Bits

- Write an assembly language program to clear bits 3 and 4 of the value in r1

```
start:
            LDR     r1, =0x61E87F4C             @ load test value
            LDR     r2, =0xFFFFFFE7             @ mask to clear bits 3 and 4
            AND     r1, r1, r2                  @ clear bits 3 and 4
                                    @ result should be 0x61E87F44

stop:       B       stop
```

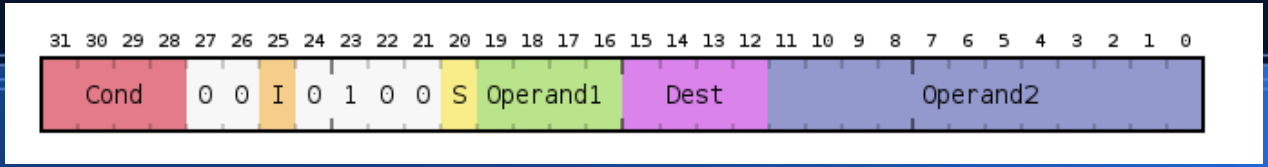
  - Alternatively, the BIC (**BI**t **C**lear) instruction allows us to define a mask with 1's in the positions we want to clear

```
            LDR     r2, =0x00000018             @ mask to clear bits 3 and 4
            BIC     r1, r1, r2                  @ r1 = r1 . NOT(r2)
```

  - Or use an immediate value, saving one instruction

```
            BIC     r1, r1, #0x00000018         @ r1 = r1 . NOT(0x00000018)
```

Here's the bit layout of an ARM data processing instruction



Any instruction with bits 27 and 26 as 00 is “data processing”. The four-bit opcode field in bits 24–21 defines exactly which instruction this is: add, subtract, move, compare, and so on. 0100 is ADD.

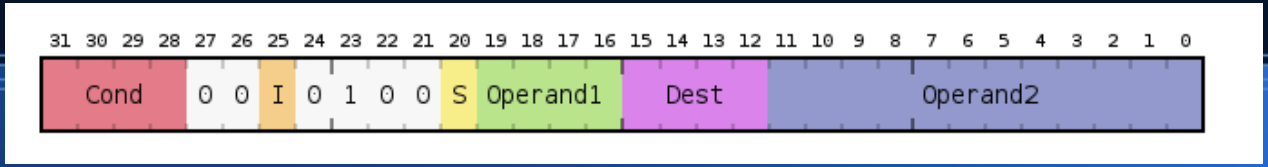
Bit 25 is the "immediate" bit. If it's 0, then operand 2 is a register. If it's set to 1, then operand 2 is an immediate value.
Note that operand 2 is only 12 bits. That doesn't give a huge range of numbers: 0–4095, or a byte and a half. Not great when you're mostly working with 32-bit numbers and addresses.

0000 - AND
0001 - EOR
0010 - SUB
0011 - RSB
0100 - ADD
0101 - ADC
0110 - SBC
0111 - RSC
1000 - TST
1001 - TEQ
1010 - CMP
1011 - CMN
1100 - ORR
1101 - MOV
1110 - BIC
1111 - MVN

The Condition Field

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Here's the bit layout of an ARM data processing instruction




```
ldr     r2, =0xFFFFFFE7
```

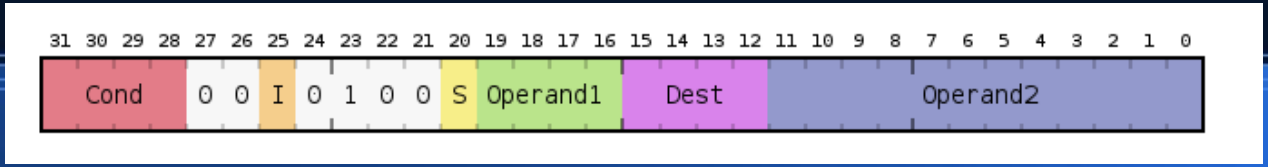
E3 E0 20 18

1110 0011 1110 0000

mvn

- 0000 - AND
- 0001 - EOR
- 0010 - SUB
- 0011 - RSB
- 0100 - ADD
- 0101 - ADC
- 0110 - SBC
- 0111 - RSC
- 1000 - TST
- 1001 - TEQ
- 1010 - CMP
- 1011 - CMN
- 1100 - ORR
- 1101 - MOV
- 1110 - BIC
- 1111 - MVN

Here's the bit layout of an ARM data processing instruction




```
add     r0,r1,#0xFFFFFFE7
```

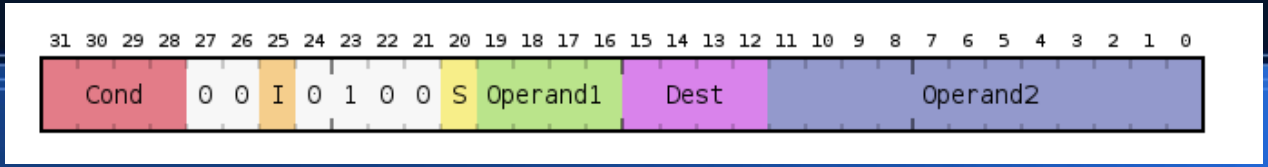
E2 41 00 19

0010 0100

SUB

- 0000 - AND
- 0001 - EOR
- 0010 - SUB
- 0011 - RSB
- 0100 - ADD
- 0101 - ADC
- 0110 - SBC
- 0111 - RSC
- 1000 - TST
- 1001 - TEQ
- 1010 - CMP
- 1011 - CMN
- 1100 - ORR
- 1101 - MOV
- 1110 - BIC
- 1111 - MVN

Here's the bit layout of an ARM data processing instruction




```
and     r1, r1,#0xFFFFFFE7
```

E3 C1 10 18

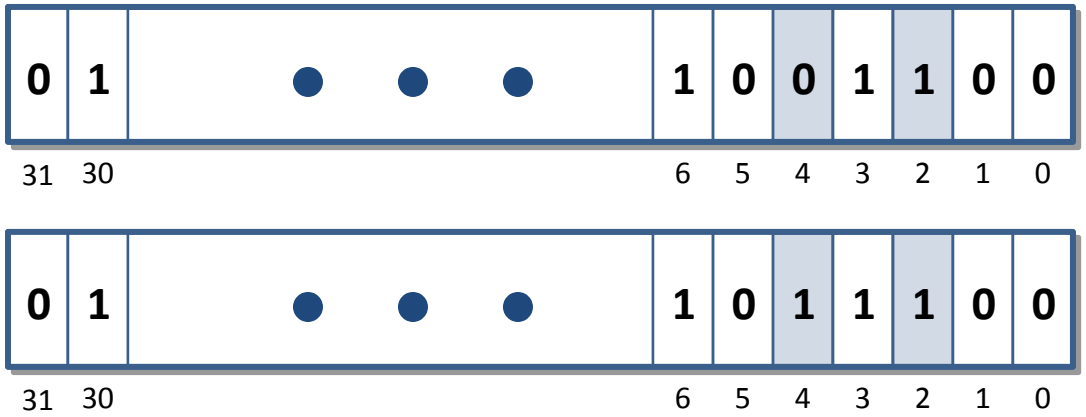
0011 1100

BIC

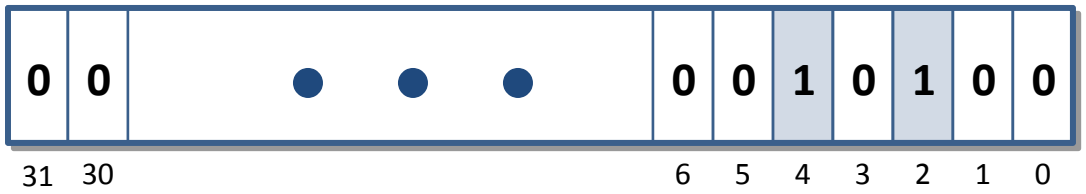
- 0000 - AND
- 0001 - EOR
- 0010 - SUB
- 0011 - RSB
- 0100 - ADD
- 0101 - ADC
- 0110 - SBC
- 0111 - RSC
- 1000 - TST
- 1001 - TEQ
- 1010 - CMP
- 1011 - CMN
- 1100 - ORR
- 1101 - MOV
- 1110 - BIC
- 1111 - MVN

Bit Manipulation

- e.g. Set bits 2 and 4 of the value in r1



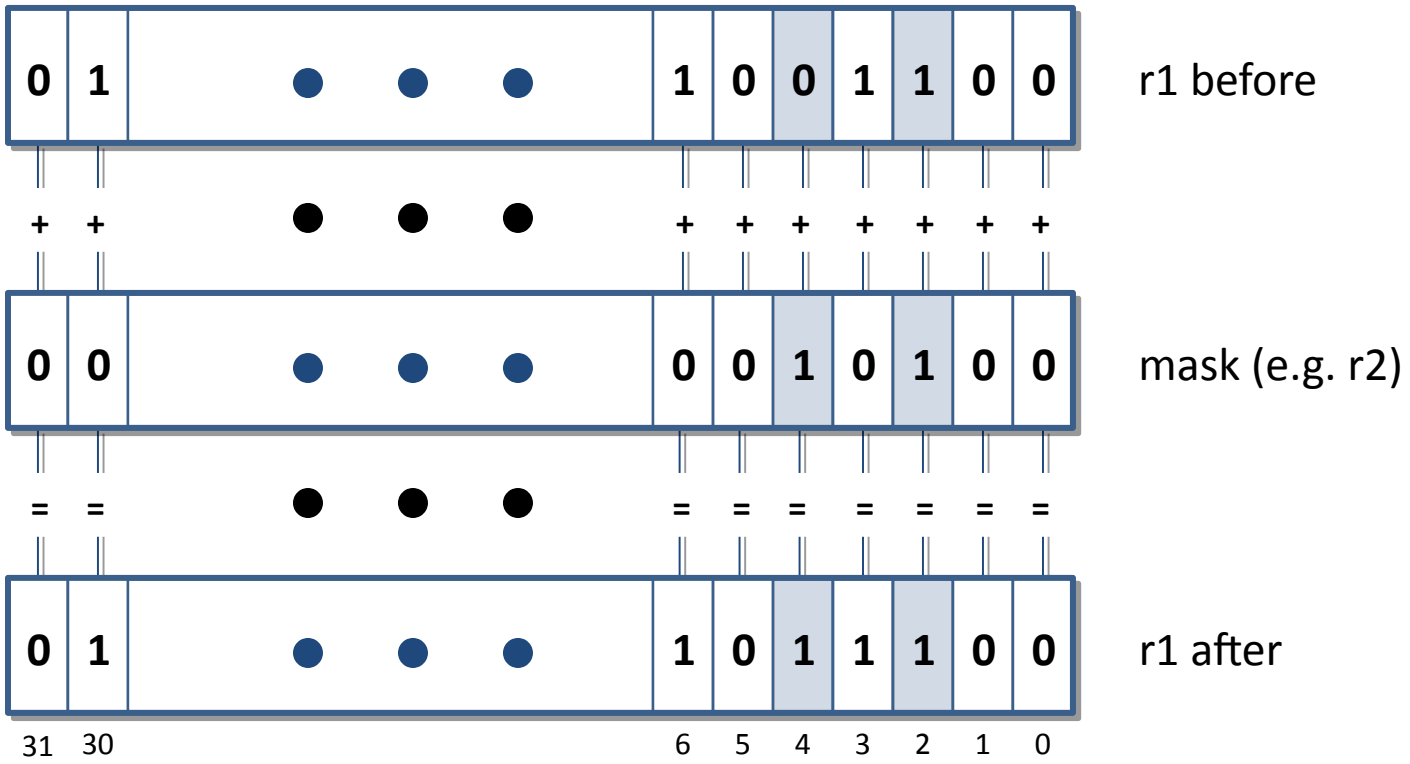
  - Observe $1 + x = 1$ and $0 + x = x$
  - Construct a mask with 1 in the bit positions we want to set and 0 in the bit positions we want to leave unchanged



  - Perform a bitwise logical OR of the value with the mask

Bit Manipulation

- e.g. Set bits 2 and 4 of the value in r1 (continued)



Program 5.2 – Set Bits

- Write an assembly language program to set bits 2 and 4 of the value in r1

```
start:
            LDR     r1, =0x61E87F4C             @ load test value
            LDR     r2, =0x00000014             @ mask to set bits 2 and 4
            ORR     r1, r1, r2                  @ set bits 2 and 4
                                    @ result should be 0x61E87F5C

stop:       B       stop
```

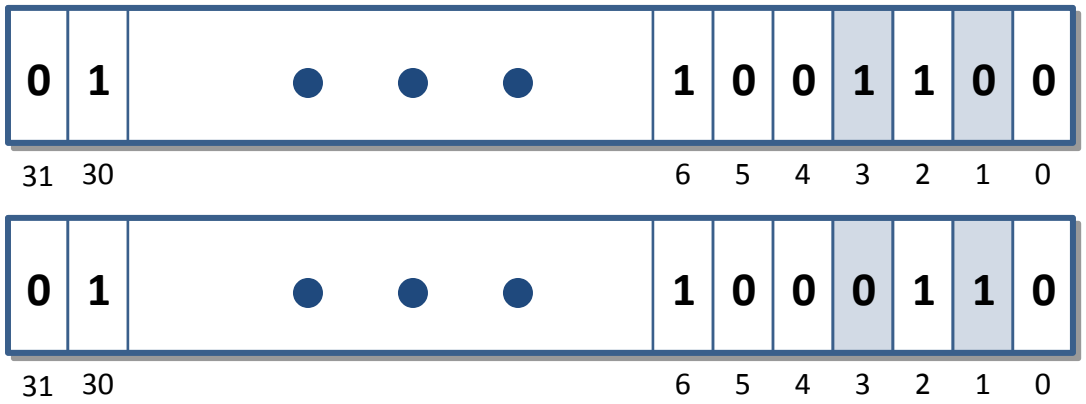
  - Can save an instruction by specifying the mask as an immediate operand in the ORR instruction

```
            ORR     r1, r1, #0x00000014         @ set bits 2 and 4
```

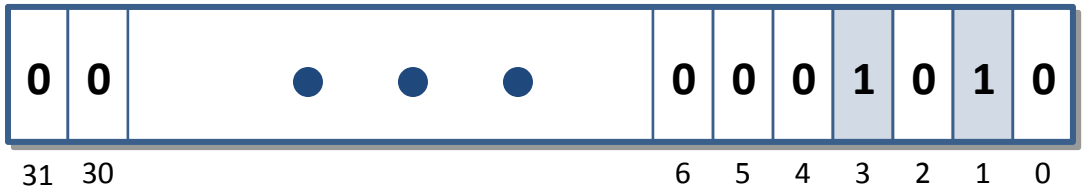
  - REMEMBER: since the ORR instruction must fit in 32 bits, only some 32-bit immediate operands can be encoded. Assembler will warn you if the immediate operand you specify is invalid.

Bit Manipulation

- e.g. Invert bits 1 and 3 of the value in r1



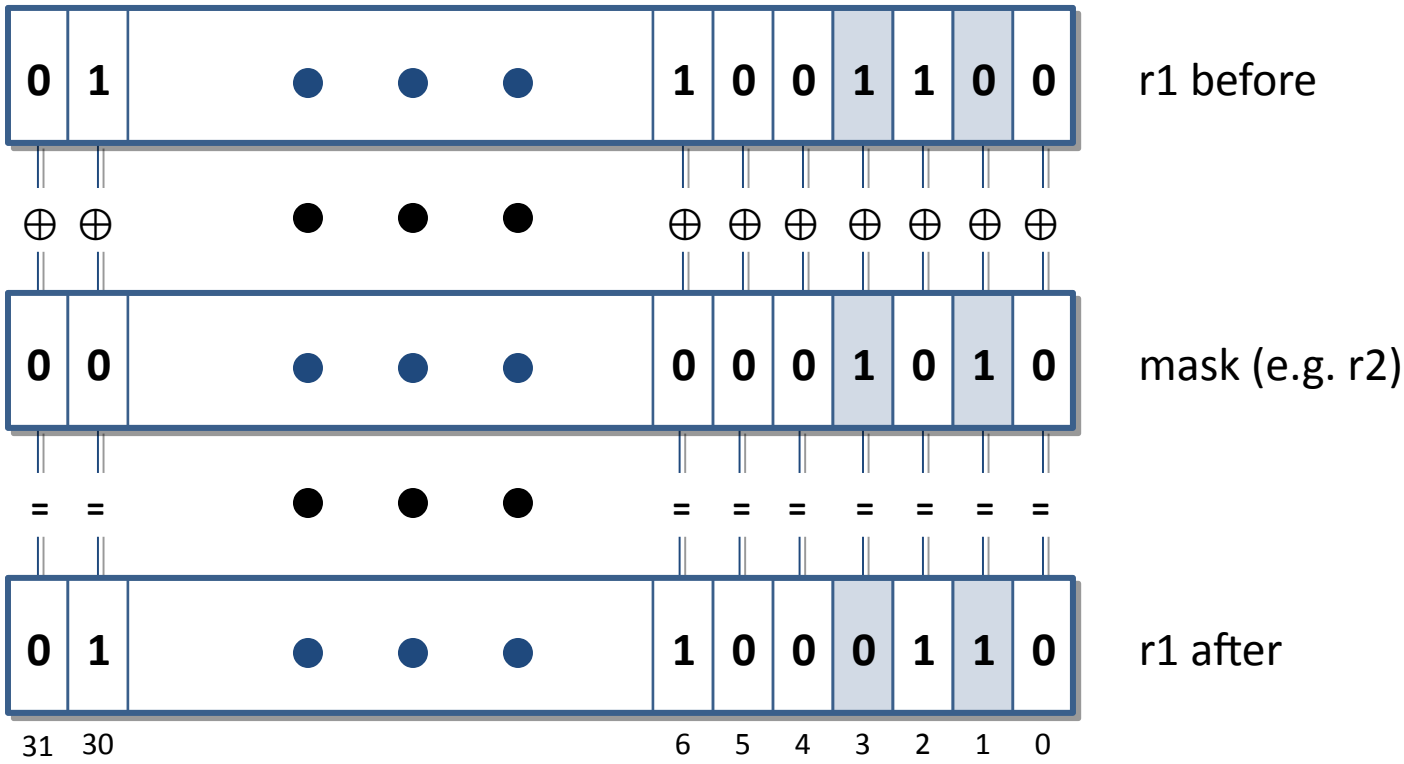
- Observe 1 $\oplus$ *x* = x' and 0 $\oplus$ *x* = *x*
- Construct a mask with 1 in the bit positions we want to invert and 0 in the bit positions we want to leave unchanged



- Perform a bitwise logical exclusive-OR of the value with the mask

Bit Manipulation

- e.g. Invert bits 1 and 3 of the value in r1 (continued)



Program 5.3 – Invert Bits

- Write an assembly language program to invert bits 1 and 3 of the value in r1

```
start:
            LDR     r1, =0x61E87F4C             @ load test value
            LDR     r2, =0x0000000A   @ mask to invert bits 1 and 3
            EOR     r1, r1, r2                  @ invert bits 1 and 3
                                    @ result should be 0x61E87F46

stop:       B       stop
```

- Again, can save an instruction by specifying the mask as an

```
            EOR     r1, r1, #0x0000000A         @ invert bits 1 and 3
```

- Again, only some 32-bit immediate operands can be encoded.

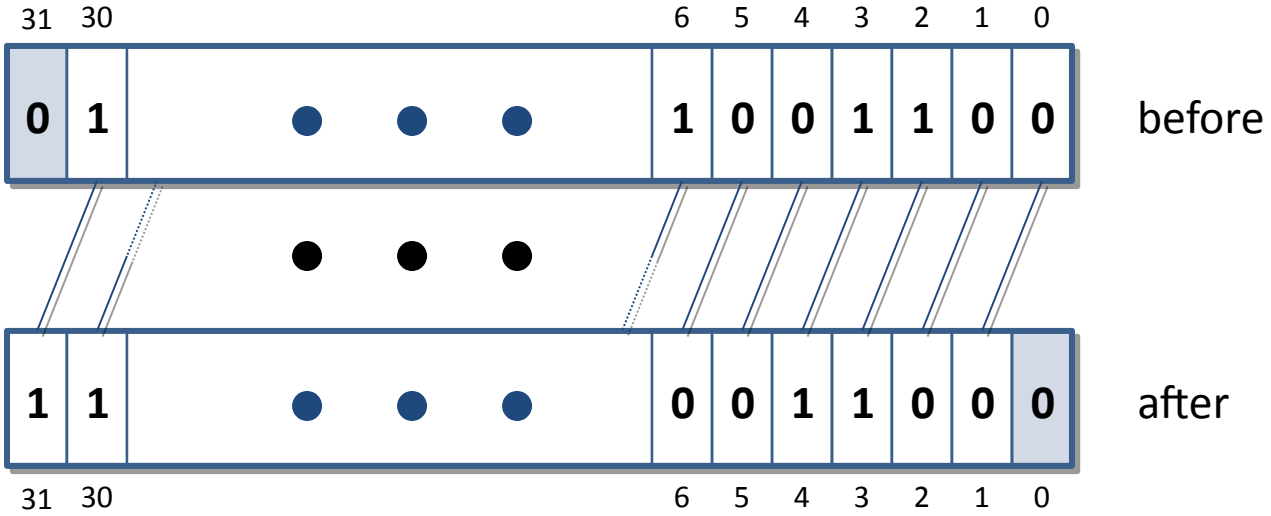
Program 5.4 – Upper Case

- Design and write an assembly language program that will make the ASCII character stored in r0 upper case.

	0	1	2	3	4	5	6	7
0	NUL	DLE	SPACE	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	“	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	‘	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	@	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Logical Shift Left

- Logical Shift Left by 1 bit position



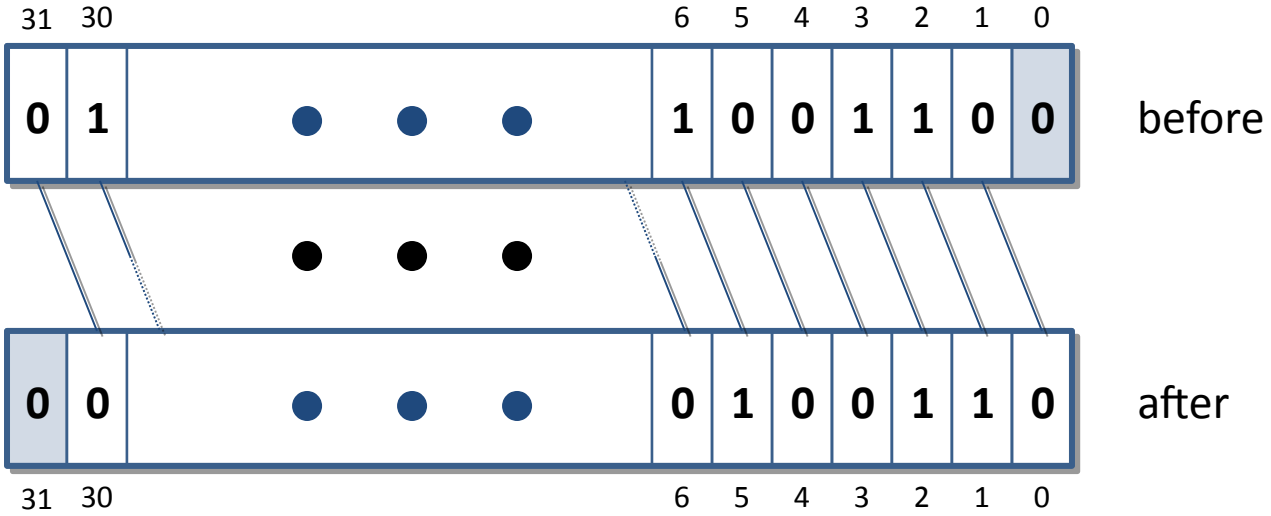
- ARM MOV instruction allows a source operand, R*m*, to be shifted left by *n* = 0 ... 31 bit positions before being stored in the destination operand, R*d*

`MOV Rd, Rm, LSL #n`

  - LSB of R*d* is set to zero, MSB of R*m* is discarded

Logical Shift Right

- Logical Shift Right by 1 bit position



- ARM MOV instruction allows a source operand, R*m*, to be shifted right by *n* = 0 ... 31 bit positions before being stored in the destination operand, R*d*

`MOV Rd, Rm, LSR #n`

  - MSB of R*d* is set to zero, LSB of R*m* is discarded

Logical Shift Left/Right – Examples

- Logical shift left r1 by 2 bit positions

```
MOV     r1, r1, LSL #2   @ r1 = r1 << 2
```

- Logical shift left r1 by 5 bit positions, store result in r0

```
MOV     r0, r1, LSL #5   @ r0 = r1 << 5
```

- Logical shift right r2 by 1 bit position

```
MOV     r2, r2, LSR #1   @ r2 = r2 >> 1
```

- Logical shift right r3 by 4 bit positions, store result in r1

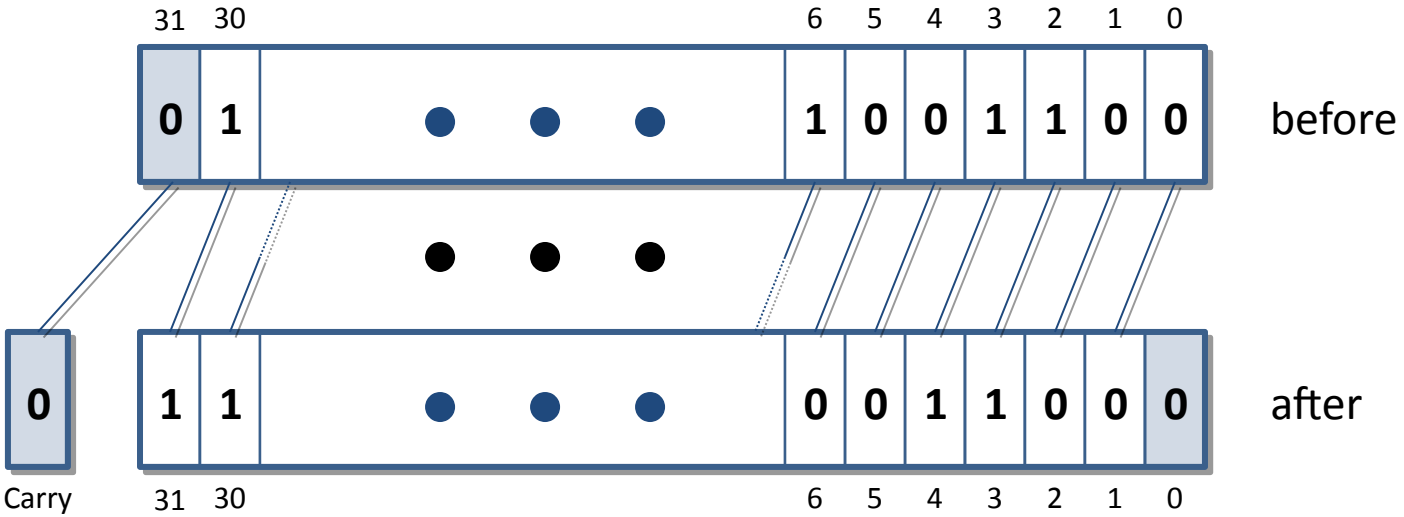
```
MOV     r1, r3, LSR #4   @ r1 = r3 >> 4
```

- Logical shift left r4 by the number of positions in r0

```
MOV     r4, r4, LSR r0   @ r4 = r4 >> r0
```

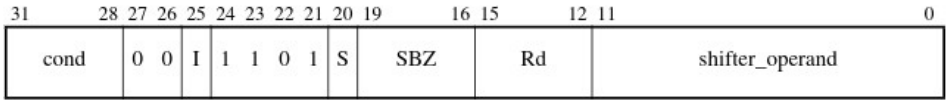
MOVS

- Instead of discarding the MSB when shifting left (or LSB when shifting right), we can cause the last bit shifted out to be stored in the Carry Condition Code Flag
  - By setting the S-bit in the MOV machine code instruction
  - By using MOVS instead of MOV




```
MOVS Rd, Rm, LSL #n
MOVS Rd, Rm, LSR #n
```

A4.1.35 MOV



MOV (Move) writes a value to the destination register. The value can be either an immediate value or a value from a register, and can be shifted before the write.

MOV can optionally update the condition code flags, based on the result.

Syntax

```
MOV{<cond>}{S}  <Rd>, <shifter_operand>
```

where:

<cond> Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If <cond> is omitted, the AL (always) condition is used.

S Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If S is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when S is specified:

- If <Rd> is not R15, the N and Z flags are set according to the value moved (post-shift if a shift is specified), and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.
- If <Rd> is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

<Rd> Specifies the destination register.

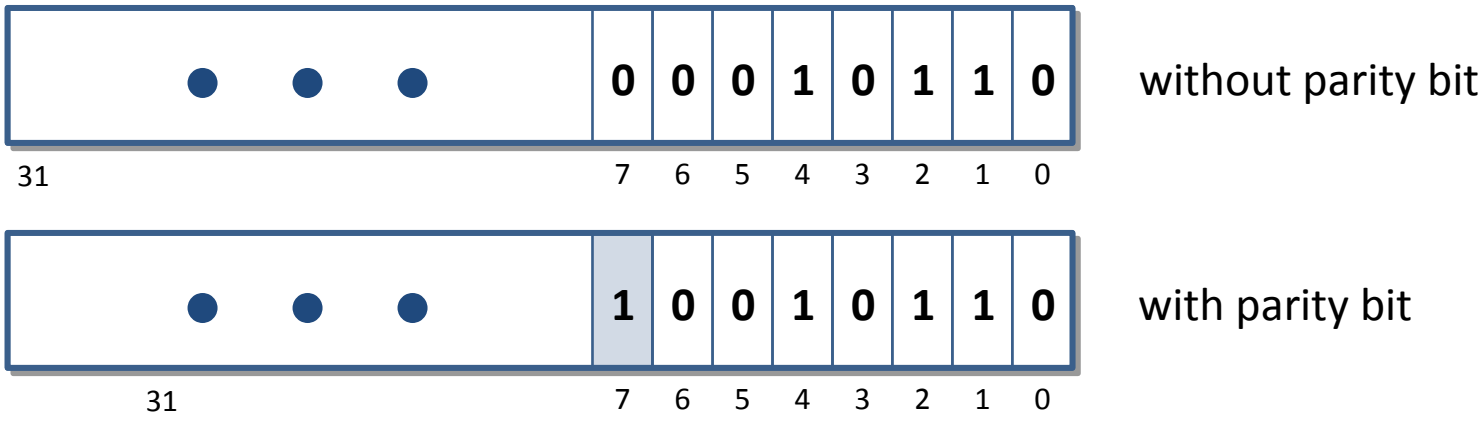
<shifter_operand>

Specifies the operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not MOV. Instead, see *Extending the instruction set* on page A3-32 to determine which instruction it is.

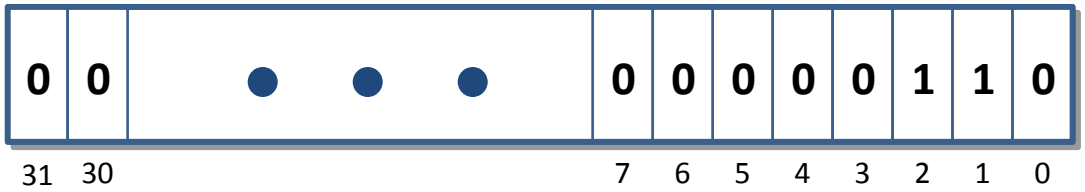
Program 5.6 – Parity

- Design and write an assembly language program that will calculate the parity bit for a 7-bit value stored in r1. The program should then store the computed parity bit in bit 7 of r1. Assume even parity.
- Parity bits are used to detect data transmission errors.
  - Using even parity, the parity bit of a value is set such that the number of set bits (1's) in a value is always even.
- Parity example



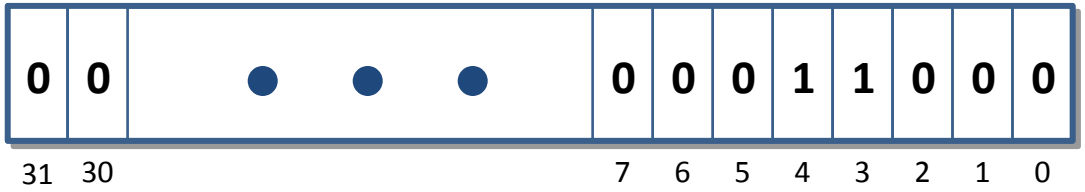
Shift-And-Add Multiplication

- Shifting a binary value left (right) by *n* bit positions is an efficient way of multiplying (~dividing) the value by 2^n
- Example



r1 = 6

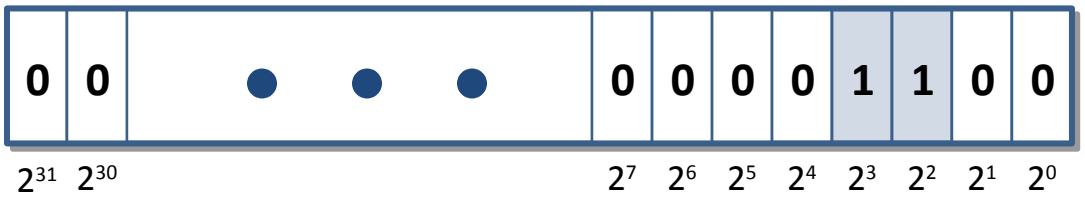
```
MOV r1, r1, LSL #2
```



r1 = 6 × 2^2 = 24

Shift-And-Add Multiplication

- We can express multiplication by any value as the **sum of the results of multiplying the value by different powers of 2**
- Example
  - $a \times 12 = a \times (8 + 4) = a \times (2^3 + 2^2) = (a \times 2^3) + (a \times 2^2)$
  - $a \times 12 = (a << 3) + (a << 2)$
- Is there a simple way to determine which powers of 2 we need to use for our partial products?



$12 = 2^3 + 2^2$

Program 5.7 – Shift And Add Multiplication

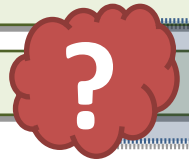
- Design and write an assembly language program that will multiply the value in r1 by 12 and store the result in r0

```
start:
            MOV     r0, r1, LSL #3          @ r0 = r1 * 2 ^ 3
            ADD     r0, r0, r1, LSL #2      @ r0 = r0 + r1 * 2^2

stop:       B       stop
```

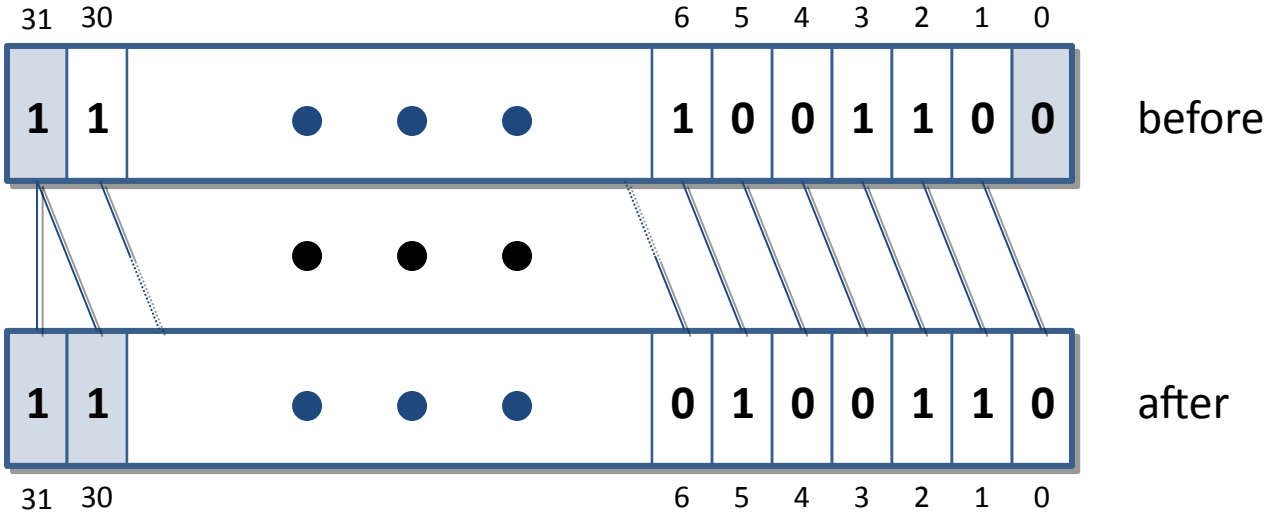
- [ASIDE] We can also formulate instructions to efficiently compute R*m* × (2^n-1) or R*m* × (2^n+1), saving one instruction

```
            ADD     r0, r1, r1, LSL #3      @ r0 = r1 * 9
            RSB     r0, r1, r1, LSL #3      @ r0 = r1 * 7
```



Arithmetic Shift Right

- Arithmetic Shift Right by 1 bit position



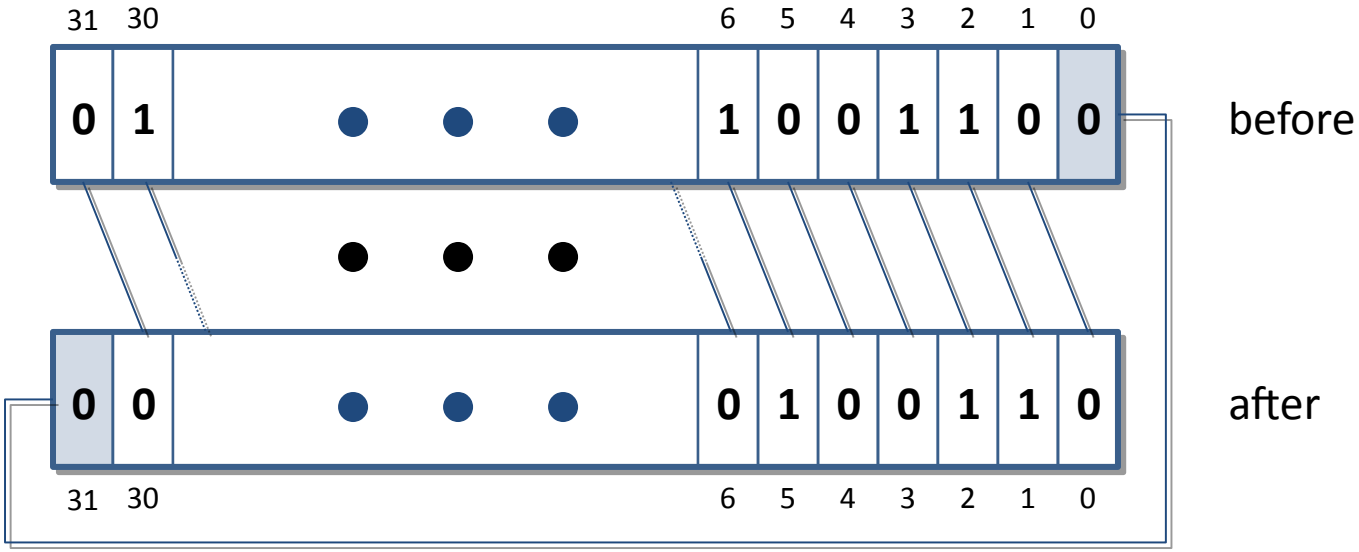
- ARM MOV instruction allows a source operand, R*m*, to be shifted right by *n* = 0 ... 31 bit positions before being stored in the destination operand, R*d*

`MOV Rd, Rm, ASR #n`

  - MSB of R*d* is set to MSB of R*m*, LSB of R*m* is discarded

Rotate Right

- Rotate Right by 1 bit position

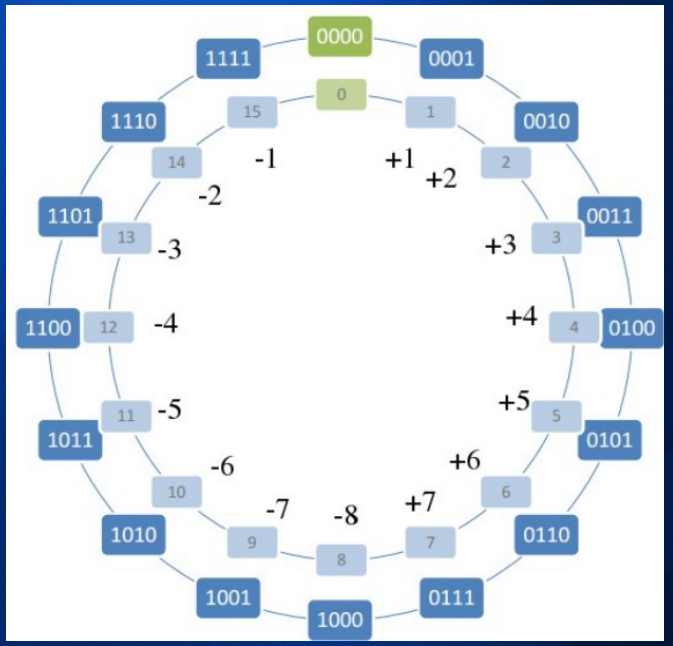


- ARM MOV instruction allows a source operand, R*m*, to be rotated right by *n* = 0 ... 31 bit positions before being stored in the destination operand, R*d*

`MOV Rd, Rm, ROR #n`

Sticky right-shift and zero-fill right-shift (arithmetic and logical shift)

1011	-5	0101	5
1101	-3	0010	2



Shifting right by n bits on a two's complement signed binary number has the effect of dividing it by 2n, but it always rounds down (towards negative infinity).

This is different from the way rounding is usually done in signed integer division (which rounds towards 0)

Shifting left by n bits on a signed or unsigned binary number has the effect of multiplying it by 2n

For an 8-bit signed integer -1 is 1111 1111. An arithmetic right-shift by 1 (or 2, 3, ..., 7)

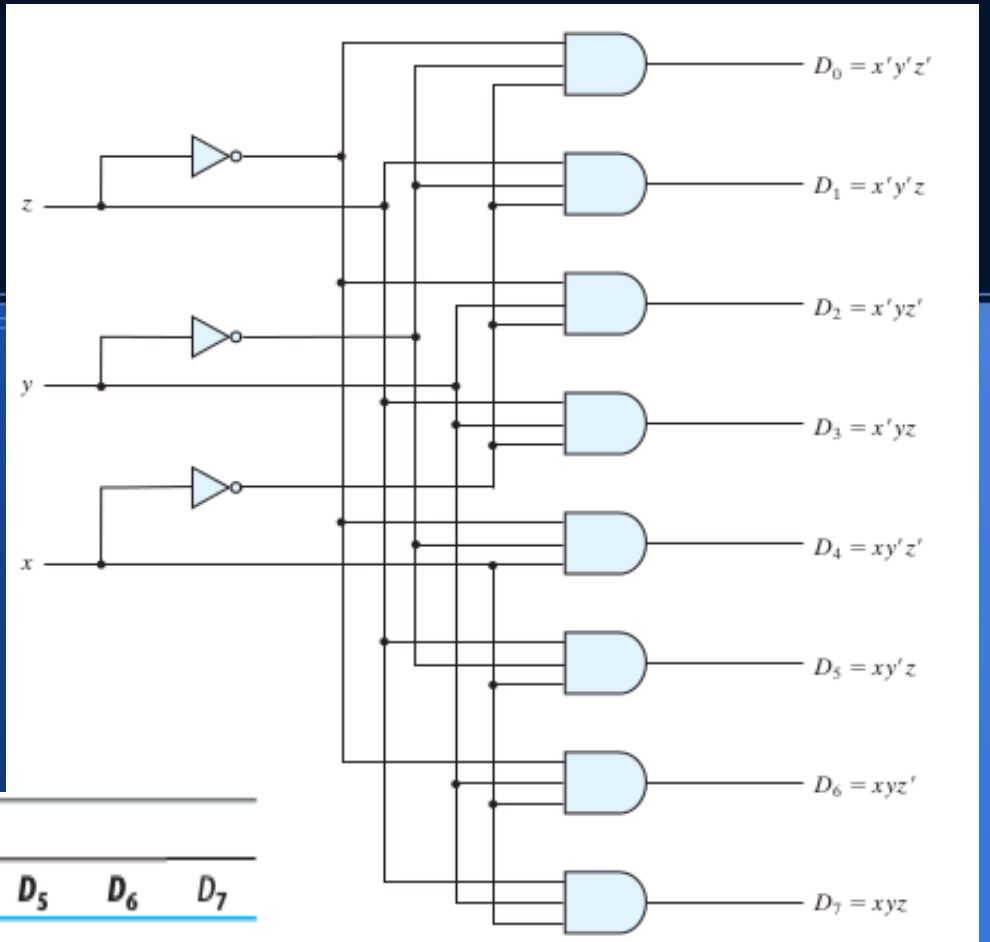
yields 1111 1111 again, which is still −1.

This corresponds to rounding down (towards negative infinity),

but is not the usual convention for division

Three-to-eight-line decoder

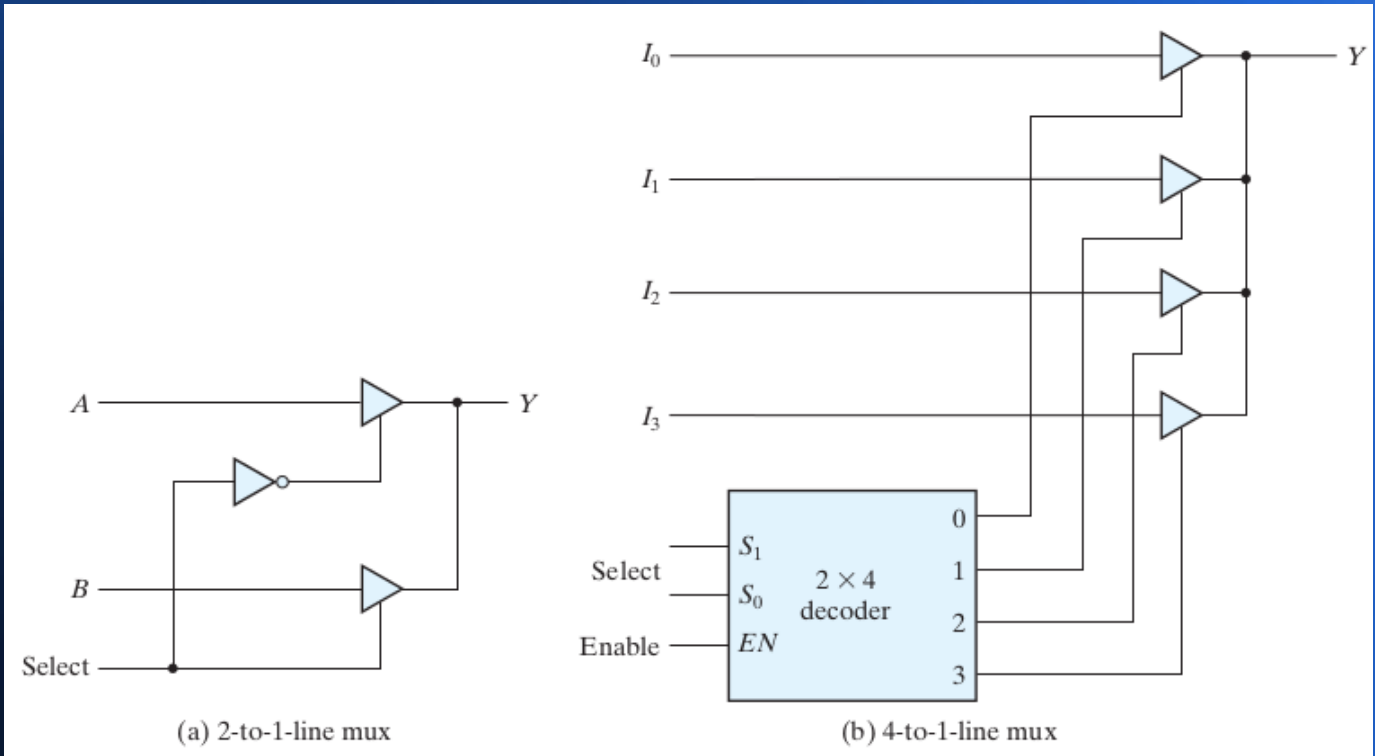
The decoder is a commonly used type of integrated circuit.
This decoder generates all of the minterms of the three input variables.



Inputs			Outputs							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

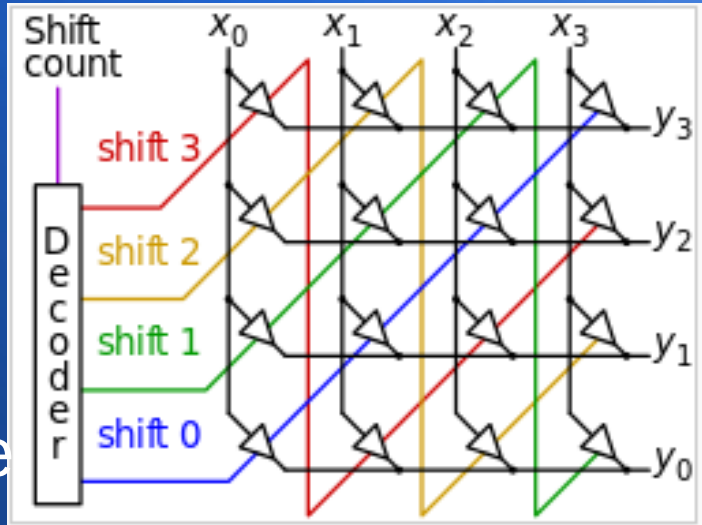
Exactly one of the output lines will be 1 for each combination of the values of the input variables.

Multiplexers with three-state gates

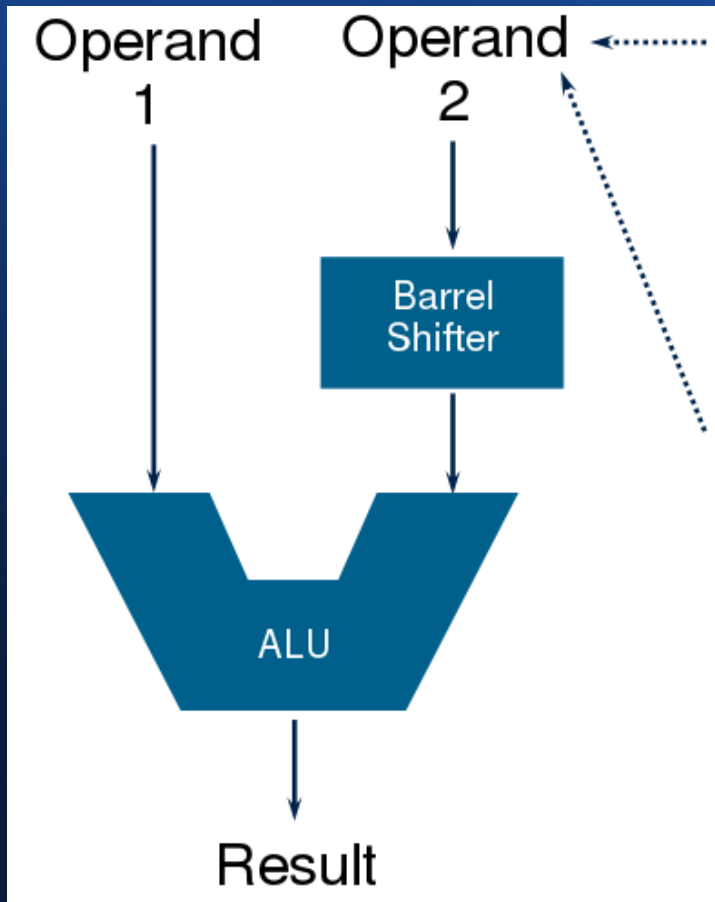


A barrel shifter is a digital circuit that can shift a data word by a specified number of bits in one clock cycle.

It can be implemented as a sequence of multiplexers (mux.), and in such an implementation the output of one mux is connected to the input of the next mux in a way that depends on the shift distance.



Schematic of a 4-bit crossbar barrel shifter. *x* denotes input bits and *y* denotes output bits.



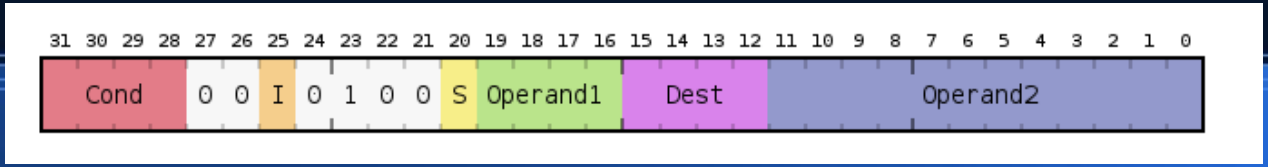
Register, optionally with shift operation

- Shift value can be either be:
  - 5 bit unsigned integer
  - Specified in bottom byte of another register.
- Used for multiplication by constant

Immediate value

- 8 bit number, with a range of 0-255.
  - Rotated right through even number of positions
- Allows increased range of 32-bit constants to be loaded directly into registers

Here's the bit layout of an ARM data processing instruction



Any instruction with bits 27 and 26 as 00 is “data processing”. The four-bit opcode field in bits 24–21 defines exactly which instruction this is: add, subtract, move, compare, and so on. 0100 is ADD.

Bit 25 is the "immediate" bit. If it's 0, then operand 2 is a register. If it's set to 1, then operand 2 is an immediate value.
Note that operand 2 is only 12 bits. That doesn't give a huge range of numbers: 0–4095, or a byte and a half. Not great when you're mostly working with 32-bit numbers and addresses.

0000 - AND
0001 - EOR
0010 - SUB
0011 - RSB
0100 - ADD
0101 - ADC
0110 - SBC
0111 - RSC
1000 - TST
1001 - TEQ
1010 - CMP
1011 - CMN
1100 - ORR
1101 - MOV
1110 - BIC
1111 - MVN

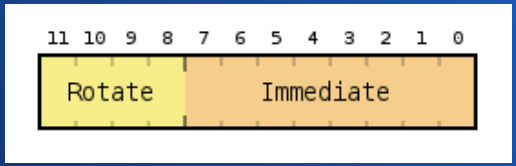
Each ARM instruction is encoded into a 32-bit word.

Access to memory is provided only by Load and Store instructions.

ARM data-processing instructions operate on data and produce
A new value.

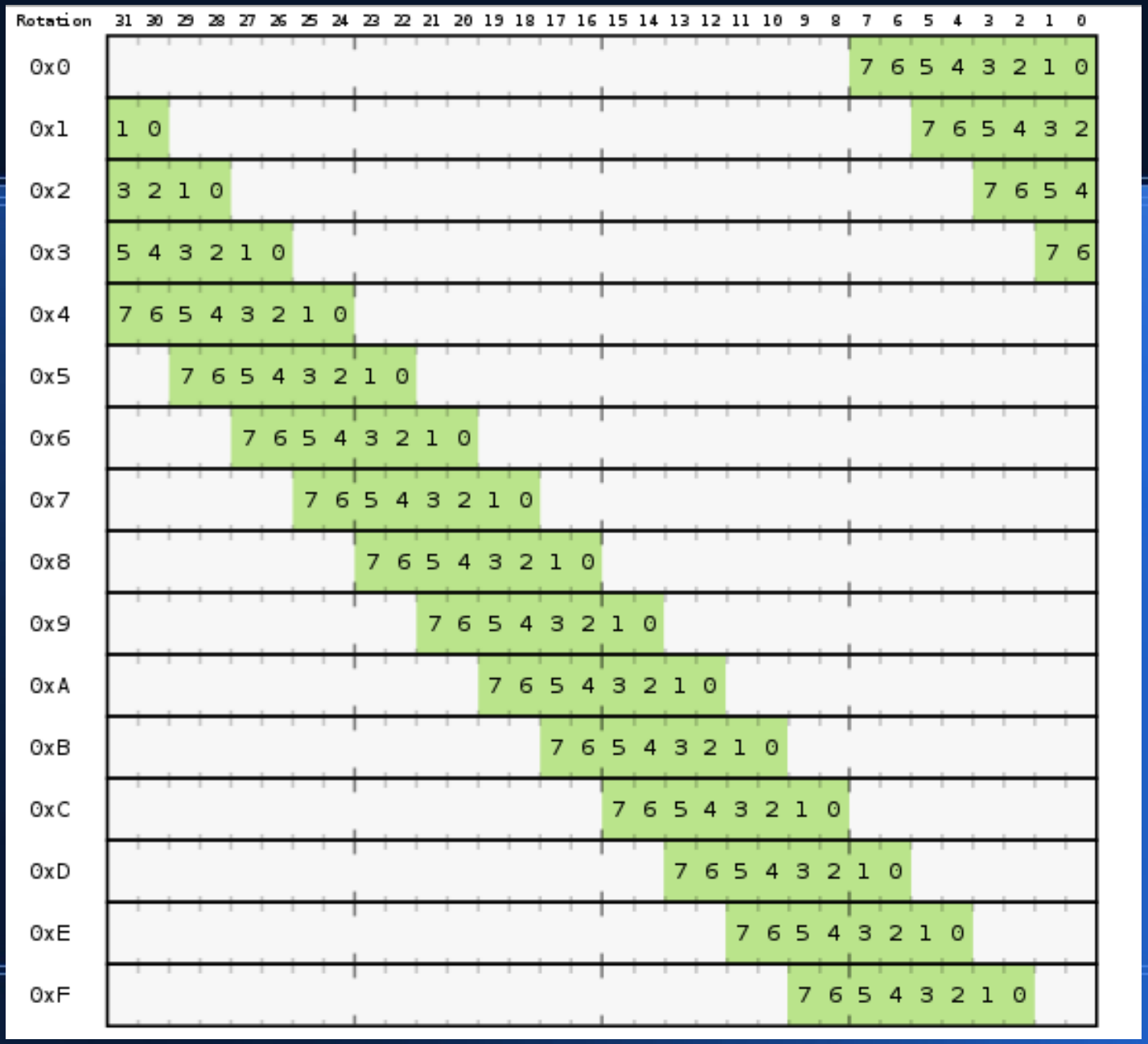
They are not like the branch instructions that control the operation
of the processor and sequencing of instructions.

But ARM doesn't use the 12-bit immediate value as a 12-bit number. Instead, it's an 8-bit number with a 4-bit rotation, like this:



The 4-bit rotation value has 16 possible settings, so it's not possible to rotate the 8-bit value to any position in the 32-bit word. The most useful way to use this rotation value is to multiply it by two. It can then represent all even numbers from zero to 30.

To form the constant for the data processing instruction, the 8-bit immediate value is extended with zeroes to 32 bits, then rotated the specified number of places to the right. For some values of rotation, this can allow splitting the 8-bit value between bytes.


Rotation
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0x0 7 6 5 4 3 2 1 0
0x1 1 0 7 6 5 4 3 2
0x2 3 2 1 0 7 6 5 4
0x3 5 4 3 2 1 0 7 6
0x4 7 6 5 4 3 2 1 0
0x5 7 6 5 4 3 2 1 0
0x6 7 6 5 4 3 2 1 0
0x7 7 6 5 4 3 2 1 0
0x8 7 6 5 4 3 2 1 0
0x9 7 6 5 4 3 2 1 0
0xA 7 6 5 4 3 2 1 0
0xB 7 6 5 4 3 2 1 0
0xC 7 6 5 4 3 2 1 0
0xD 7 6 5 4 3 2 1 0
0xE 7 6 5 4 3 2 1 0
0xF 7 6 5 4 3 2 1 0

The rotated byte encoding allows the 12-bit value to represent a much more useful set of numbers than just 0–4095. It's occasionally even more useful than the MIPS or PowerPC 16-bit immediate value.

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, #0x8000         @ Set bit 15 of r5
BIC  r0, r0, #0x20              @ ASCII lower-case to upper-case
EOR     r9, r9, #0x80000000     @ Toggle bit 31 of r9
```

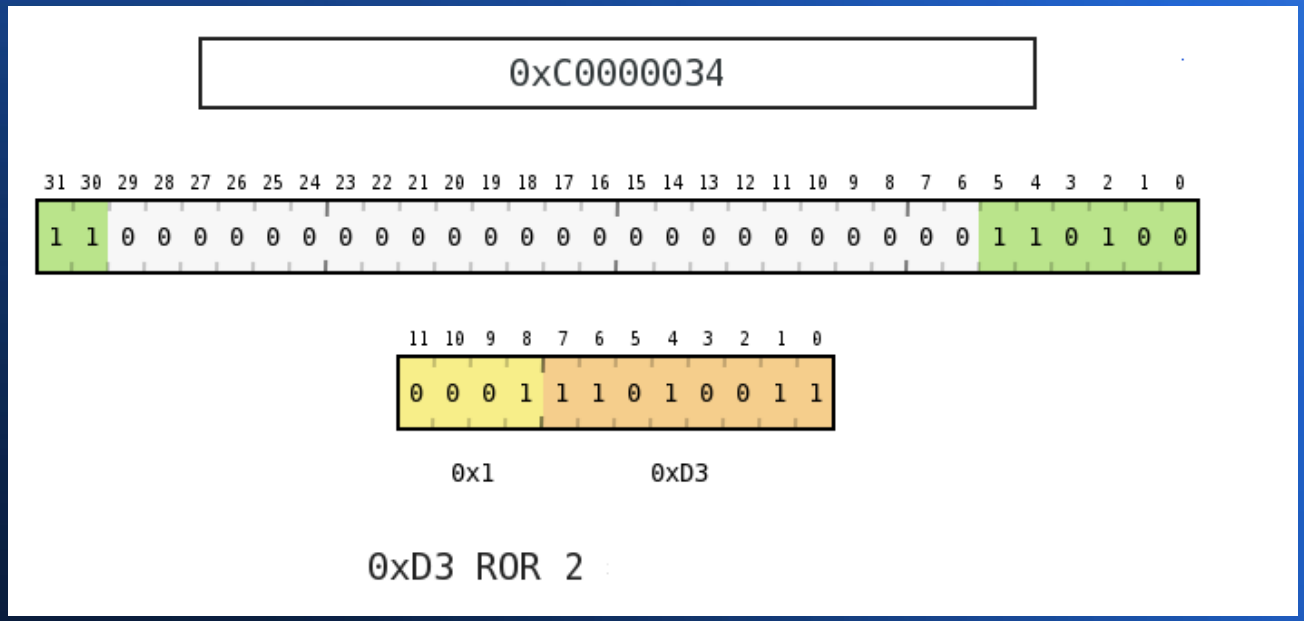
More generally, you can specify a byte value at any of the four locations in the word:

```
AND     r0, r0, #0xff000000     @ Only keep the top byte of r0
```

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.

Faced with the constraint of only having 12 bits to use, the ARM designers had the insight to reuse the idle barrel shifter to allow a wide range of useful numbers.

No other architecture has this feature. It's unique.


0xC0000034
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0
11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 1 1 1 0 1 0 0 1 1
0x1
0xD3
0xD3 ROR 2

However, for a large immediate value “mov” does NOT work,
it is not ok to use mov r0,#0x55555555
The reason is because there is no way to fit a 32-bit data into a 32-instruction (an instruction must have the instruction-code-part and the data-part, if the data-part is 32-bit long, there is no room to store the instruction-code-part).

Instead use ldr r0,=0x55555555

Then,the assembler will generate some code to place the constant 0x55555555 in a nearby table in the code area. Then it uses an instruction to load a data from that table pointed by the program counter and an offset to fill up r0.

```
LDR     rn, [pc, #offset to literal pool]
                @ load register n with one word
                @ from the address [pc + offset]
```

MOVS Rd, Operand2

Updates the N and Z flags according to the result.

Can update the C flag during the calculation of Operand2.

Does not affect the V flag.

When an Operand2 constant is used with the instructions MOVS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, the carry flag is updated to bit[31] of the constant, if the constant is greater than 255 and can be produced by shifting an 8-bit value. These instructions do not affect the carry flag if Operand2 is any other constant.