2.7 Logical operations

"Contrariwise," continued Tweedledee, "if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic."
Lewis Carroll, Alice's Adventures in Wonderland, 1865

Although the first computers operated on full words, it soon became clear that it was useful to operate on fields of bits within a word or even on individual bits. Examining characters within a word, each of which is stored as 8 bits, is one example of such an operation (see COD Section 2.9 (Communicating with people)). It follows that operations were added to programming languages and instruction set architectures to simplify, among other things, the packing and unpacking of bits into words. These instructions are called logical operations. The figure below shows logical operations in C, Java, and LEGv8.

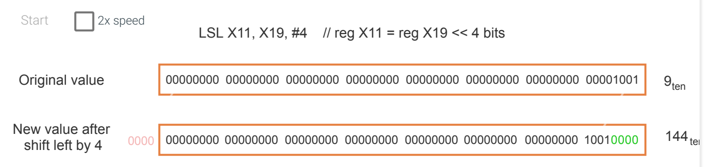
Figure 2.7.1: C and Java logical operators and their corresponding LEGv8 instructions (COD Figure 2.8).

One way to implement NOT is to use EOR with one operand being all ones (FFFF FFFF FFFF FFFF_{hex}).

Logical operations	C operators	Java operators	LEGv8 instructions
Shift left	<<	<<	LSL
Shift right	>>	>>>	LSR
Bit-by-bit AND	&	&	AND, ANDI
Bit-by-bit OR	|	|	ORR, ORRI
Bit-by-bit NOT	~	~	EOR, EORI

The first class of such operations is called *shifts*. A ***shift*** moves all the bits in a doubleword to the left or right, filling the emptied bits with 0s.

PARTICIPATION ACTIVITY 2.7.1: Example of shifting left by 4.



LSL X11, X19, #4 // reg X11 = reg X19 << 4 bits

Original value: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001001 9_{ten}

New value after shift left by 4: 0000 | 00000000 00000000 00000000 00000000 00000000 00000000 00000000 10010000 144_{te}

The dual of a shift left is a shift right. The actual names of the two LEGv8 shift instructions are *logical shift left* (`LSL`) and *logical shift right* (`LSR`). The following instruction performs the operation above, if original value was in register `X19` and the result should go in register `X11`:

```
LSL X11, X19, #4   // reg X11 = reg X19 << 4 bits
```

We delayed explaining the *shamt* field in the R-format. Used in shift instructions, it stands for *shift amount*. Hence, the machine language version of the instruction above is

opcode	Rm	shamt	Rn	Rd
1691	0	4	19	11

The encoding of `LSL` is 1691 in the opcode field, Rd contains 11, Rn contains 19, and shamt contains 4. The Rm field is unused and thus is set to 0.

Shift left logical provides a bonus benefit. Shifting left by *i* bits gives the same result as multiplying by 2^i, just as shifting a decimal number by *i* digits is equivalent to multiplying by 10^i. For example, the above `LSL` shifts by 4, which gives the same result as multiplying by 2^4 or 16. The first bit pattern above represents 9, and 9 × 16 = 144, the value of the second bit pattern.

PARTICIPATION ACTIVITY 2.7.2: Shifting.

Given the following initial value, determine the resulting value for the given operation.
00000011 00000000 00000000 00000000 00000000 00000000 00001111 00000001

Shift left by 8 | Shift right by 8 | Multiply by 2 | Shift left by 2

00000000 00000000 00000000
00000000 00000000 00001111
00000001 00000000

00000000 00000011 00000000
00000000 00000000 00000000
00000000 00001111

00001100 00000000 00000000
00000000 00000000 00000000
00111100 00000100

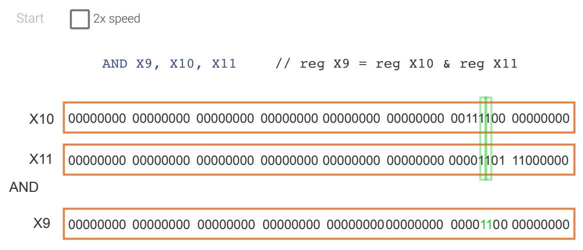
00000110 00000000 00000000
00000000 00000000 00000000
00011110 00000010

Reset

Another useful operation that isolates fields is *AND*. (We capitalize the word to avoid confusion between the operation and the English conjunction.) AND is a bit-by-bit operation that leaves a 1 in the result only if both bits of the operands are 1.

AND: A logical bit- by-bit operation with two operands that calculates a 1 only if there is a 1 in both operands.

PARTICIPATION ACTIVITY 2.7.3: Example of an AND operation.



```
AND X9, X10, X11     // reg X9 = reg X10 & reg X11
```

```
X10   00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
X11   00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
AND
X9    00000000 00000000 00000000 00000000 0000000000000000 00001100 00000000
```

As you can see, AND can apply a bit pattern to a set of bits to force 0s where there is a 0 in the bit pattern. Such a bit pattern in conjunction with AND is traditionally called a ***mask***, since the mask "conceals" some bits.

To place a value into one of these seas of 0s, there is the dual to AND, called *OR*. It is a bit-by-bit operation that places a 1 in the result if either operand bit is a 1. To elaborate, if the registers `X10` and `X11` are unchanged from the preceding example, the result of the LEGv8 instruction

```
ORR X9, X10, X11   // reg X9 = reg X10 | reg X11
```

is this value in register `X9`:

`00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000`$_{\text{two}}$

OR: A logical bit-by-bit operation with two operands that calculates a 1 if there is a 1 in either operand.

The final logical operation is a contrarian. *NOT* takes one operand and places a 1 in the result if one operand bit is a 0, and vice versa. Using our prior notation, it calculates $\bar{x}$.

NOT: A logical bit-by-bit operation with one operand that inverts the bits; that is, it replaces every 1 with a 0, and every 0 with a 1.

In keeping with the three-operand format, the designers of ARMv8 decided to include the instruction *EOR* (Exclusive OR) instead of NOT. Since exclusive OR creates a 0 when bits are the same and a 1 if they are different, the equivalent to NOT is an EOR 111...111.

EOR: A logical bit-by-bit operation with two operands that calculates the exclusive OR of the two operands. That is, it calculates a 1 only if the values are different in the two operands.

If the register `X10` is unchanged from the preceding example and register `X12` has the value 11...1111, the result of the LEGv8 instruction

```
EOR X9, X10, X12   // reg X9 = reg X10 | reg X12
```

is this value in register `X9`:

`11111111 11111111 11111111 11111111 11111111 11111111 11000011 11111111`$_{\text{two}}$

PARTICIPATION ACTIVITY 2.7.4: AND, OR, and NOT.

Given the following initial values, determine the resulting value for the given operation.
x: 00000011 00000000 00000000 00000000 00000000 00000000 11111111 00000001
y: 00000000 00000000 00000000 00000000 00000000 00000000 11111111 00001111

x AND y   NOT x   x EOR y   x OR y

00000000 00000000 00000000
00000000 00000000 00000000
11111111 00000001

00000011 00000000 00000000
00000000 00000000 00000000
11111111 00001111

11111100 11111111 11111111
11111111 11111111 11111111
00000000 11111110

00000011 00000000 00000000

00000000 00000000 00000000
00000000 00001110

Reset

The figure above (COD Figure 2.8 (C and Java logical operators and their corresponding LEGv8 instructions)) shows the relationship between the C and Java operators and the LEGv8 instructions. Constants are useful in logical operations as well as in arithmetic operations, so LEGv8 also provides the instructions *and immediate* (`ANDI`), *or immediate* (`ORRI`), and *exclusive or immediate* (`EORI`).

Elaboration

C allows bit fields or fields to be defined within doublewords, both allowing objects to be packed within a doubleword and to match an externally enforced interface such as an I/O device. All fields must fit within a single doubleword. Fields are unsigned integers that can be as short as 1 bit. C compilers insert and extract fields using logical instructions in LEGv8: `AND`, `ORR`, `LSL`, and `LSR`.

Elaboration

The immediate fields for `ANDI`, `ORRI`, and `EORI` of the full ARMv8 instruction set are not simple 12-bit immediates. Once again, like ARMv7, it has the unusual feature of using a complex algorithm for encoding immediate values; ARMv8 does it with repeating patterns. This means that some small constants (e.g., 1, 2, 3, 4, and 6) are valid, while others (e.g., -1, 0, 5) are not. LEGv8 simply uses normal 12-bit immediates as found in ADDI. This difference means `EORI X1,X1,#5` is legal for LEGv8 but not ARMv8. Once again, in addition to its rarity among other instruction sets, immediate encoding is omitted because it would complicate the datapaths in COD Chapter 4 (The Processor) significantly.

Elaboration

Unlike almost all other computer architectures, ARMv8 (and ARMv7) allows a register to be shifted as part of an arithmetic or logical instruction: add an optionally shifted register, subtract an optionally shifted register, AND an optionally shifted register, and so on. Since this combination is unusual in computer architectures and not frequently generated by compilers--and since supporting it would make the data path in COD Chapter 4 (The Processor) much more complicated and unlike of other computer datapaths--we decided to treat shifts as separate instructions, as it is in virtually every other computer architecture. While you can synthesize a shift using either `ADD` with `XZR` or `OR` with `XZR`, it would be confusing to use the same opcode for shifts as `ADD` or `OR`. Thus, we follow the ARMv8 recommendation of using `UBFM` (unsigned bitfield move) instruction and its opcode. We simplify the values put into the Rm and shamt fields to be 0 and the actual immediate shift amount, which is what it looks like in ARMv8 assembly language. The actual values in the Rm and shamt fields of UBFM should be (-shift amount MOD 64) and (63 - shift amount) for `LSL` and shift amount and 63 for `LSR`. The opcode field includes part of immediate field in ARMv8, so we make the two opcodes 1691 and 1690, respectively, to distinguish them.

PARTICIPATION ACTIVITY 2.7.5: Logical operations.

1) AND can be used to mask out particular bits (forcing those bits to 0's).
   - ○ True
   - ○ False

2) OR can be used to set particular bits to 1.
   - ○ True
   - ○ False

3) Goal: Force the rightmost bit of X10 to be 1.
   `____ X10, X10, 1`
   - ○ ANDI
   - ○ ORI

4) To isolate bits 7..4 in the rightmost 4 bits of a 64-bit doubleword, one can first shift left 56 bits, then right ____ bits.
   - ○ 56
   - ○ 60

5) Goal: multiply X10 by 8.
   `LSL X10, X10, ____`
   - ○ 3
   - ○ 8

PARTICIPATION ACTIVITY 2.7.6: Check yourself: Masking operators.

1) An AND operation can isolate a field in a

doubleword.

○ True

○ False

2) A shift left followed by a shift right operation can isolate a field in a doubleword.

○ True

○ False

CHALLENGE ACTIVITY

2.7.1: Logical operations.

Start

Compute: X4 = (X5 >> 3) & 7

```
LSR  X4, X4, # 0
ANDI X4, X4, # 0
```

Registers

X4	0
X5	0..0111111111110100000
X6	0

1 2 3 4 5

Check Next

Provide feedback on this section