# Binary Logic (review)

Basic logical operators:  

- **NOT**
- **AND** – outputs 1 only if both inputs are 1
- **OR** – outputs 1 if at least one input is 1
- **XOR** – outputs 1 if exactly one input is 1

<table>
<thead>
<tr>
<th>a</th>
<th>not</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>and</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>or</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>xor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>bic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Binary Logic

Logic operators in C: && and  
|| or 
! not 

(zero word = false, nonzero word = true)

Bitwise operators in C: & and  
| or 
~ not 
^ xor 

(each bit in the word is independent)
Uses for Logical Operators

• ANDing a bit with 0 produces a 0 at the output while ANDing a bit with 1 produces the original bit.
• This can be used to create a mask.

– Example:

  1011 0110 1010 0100 0011 1101 1001 1010
  0000 0000 0000 0000 0000 1111 1111 1111

  mask: 0000 0000 0000 0000 0000 1111 1111 1111

  – The result of ANDing these:

  0000 0000 0000 0000 0000 1101 1001 1010

  mask last 12 bits
Uses for Logical Operators

- ORing a bit with 1 produces a 1 at the output while ORing a bit with 0 produces the original bit.
- This can be used to force certain bits of a string to 1s.
  - For example, 0x12345678 OR 0x0000FFFF results in 0x1234FFFF (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s).
Uses for Logical Operators

• XORing a bit with 1 flips the bit (0 -> 1, 1 -> 0) at the output while XORing a bit with 0 produces the original bit.
Uses for Logical Operators

• BICing a bit with 1 resets the bit (sets to 0) at the output while BICing a bit with 0 produces the original bit.

• This can be used to force certain bits of a string to 0s.
  – For example, 0x12345678 BIC 0x0000FFFF results in 0x12340000 (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 0s).
As we have already discussed, Arm provides the following Boolean (logical) instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>and{cond}{S} rd, rn, op2</code></td>
<td>Performs logical AND of <code>rn</code> with <code>op2</code>.</td>
</tr>
<tr>
<td><code>orr{cond}{S} rd, rn, op2</code></td>
<td>Performs logical OR of <code>rn</code> with <code>op2</code></td>
</tr>
<tr>
<td><code>eor{cond}{S} rd, rn, op2</code></td>
<td>Performs logical exclusive or operation of <code>rn</code> with <code>op2</code></td>
</tr>
<tr>
<td><code>mvn{cond}{S} rd, op2</code></td>
<td>Performs a bitwise logical NOT operation on the value of <code>op2</code></td>
</tr>
<tr>
<td><code>bic{cond}{S} rd, rn, op2</code></td>
<td>Performs an AND operation on the bits in <code>Rn</code> with the complements of the corresponding bits in the value of <code>Op2</code></td>
</tr>
</tbody>
</table>
## Bitwise Instructions

<table>
<thead>
<tr>
<th>ARM opcode</th>
<th>description</th>
<th>a = 0 0 1 1</th>
<th>b = 0 1 0 1</th>
<th>some common names</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td></td>
<td>0 0 0 0</td>
<td></td>
<td>false, zero, clear</td>
</tr>
<tr>
<td>and</td>
<td>a and b</td>
<td>0 0 0 1</td>
<td></td>
<td>and</td>
</tr>
<tr>
<td>orr</td>
<td>a or b</td>
<td>0 1 1 1</td>
<td></td>
<td>or, inclusive or</td>
</tr>
<tr>
<td>bic</td>
<td>a and (not b)</td>
<td>0 0 1 0</td>
<td></td>
<td>and-not, inhibit, a&gt;b</td>
</tr>
<tr>
<td>eor</td>
<td>a xor b</td>
<td>0 1 1 0</td>
<td></td>
<td>xor, exclusive or, a!=b</td>
</tr>
<tr>
<td>mvn</td>
<td>not b</td>
<td>1 0 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>true</td>
<td></td>
<td>1 1 1 1</td>
<td></td>
<td>true, one, set</td>
</tr>
</tbody>
</table>
Barrel Shifter

- The *barrel shifter* is a functional unit which can be used in a number of different circumstances.
- It provides five types of shifts and rotates which can be applied to Operand2:
  - lsl – logical shift left
  - lsr – logical shift right
  - asr – arithmetic shift right
  - ror – rotate right

Note: These are not operations themselves in ARM mode.
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
**Shifts**

(PP. 127 – 129)

**lsl** – logical shift left by n bits - 0 enters from right, bits drop off left end – multiplication by $2^n$

Note: little-endian bit notation

![Diagram of bit shift](image)

```
01011110101000010001001000101100
```

```
5 e a 1 1 2 2 c
```

gcc generates lsl for `a = a << 1;`
**Shifts**

**lsr** - logical shift right by n bits - 0 enters from left; bits drop off right end – unsigned multiplication by \(2^n\)

```
0 1 2 3 4 5 6 7 8 9 10 11
```

```plaintext
a f 5 0 8 9 1
```

```plaintext
10101111010100001000100100010110
```

```plaintext
lsr by 1 becomes
0101011110101000010001001001001011
```

```plaintext
5 7 a 8 4 4 8 b
```

```plaintext
gcc generates srl for unsigned int a in
a = a >> 1;
```
**Shifts**

**asr** – arithmetic shift right by n bits - sign bit replicated on left, bits
drop off right end – signed division by $2^n$

```
  a  f  5  0  8  9  1  6
10101111010100001000100100010110
  asr by 1 becomes
11010111101010000100010010010010111
```
gcc generates asr for int variable a in
```
a = a >> 1;
```
Shifts

Isl and Lsr in ARM

```
MOV    r4, r6, LSL #4    /* r4 = r6 << 4 */
MOV    r4, r6, LSR #8    /* r4 = r6 >> 8 */
```

Also possible to shift by the value of a register

```
MOV    r4, r6, LSL r3
     /* r4 = r6 << value specified in r3 */
```
Shifts

• a shift right has a choice of what to put in the most significant bit (i.e., the sign bit) from the left: either zeros or replicating the sign bit; that is why there are two shift right opcodes

• a shift left has no choice as to what will go into the sign bit - it has to be one of the bits already in the register, which is determined by the shift amount; zeros always come into the least significant bit on the right
Bitwise Rotation

A bitwise rotation (or **circular shift**) is a shift operator that shifts all bits of its operand.

- Unlike an arithmetic shift, a rotate does not preserve a number's sign bit or distinguish a number's exponent from its mantissa.
- Unlike a logical shift, the vacant bit positions are not filled in with zeros but are filled in with the bits that are shifted out of the sequence.
Bitwise Rotation in ARM

**ror** – rotates right by #n bits; valid range for #n: 1 - 31
Bitwise Rotation in ARM

Example:

```assembly
mov r4, r6, ror #12 /* r4 = r6 rotated right 12 bits */
```

**Note:** r4 = r6 rotated left by 20 bits (32 -12)
Therefore there is no need for rotate left operation.
Field extraction with a bit mask - field already in rightmost bits

bit mask - a word where bits are used to zero out or select portions of another word

assume we are working with nibbles (4-bit fields)

remove (clear) all other bits and work with only the low four bits
(the mask will have 0s in all the bit positions we wish to clear and 1s in the bit positions we wish to select)

```
xxxx xxxx xxxx xxxx xxxx xxxx xxxx zzzz  r0
and 0000 0000 0000 0000 0000 0000 0000 1111    and r1, r0, #0xf
---------------------------------------------
0000 0000 0000 0000 0000 0000 0000 zzzz  r1
```
Application of Bit Operations: Field Extraction & Insertion

Field extraction with a bit mask

The previous example works; however, it uses a static mask, making the solution only useful for that special case.

We want a solution that can handle any value and mask.

You can use the following to create a mask with $m$ set bits (1), preceded by $k$ unset bits (0), and followed by $n$ unset bits (0):

$$((1 << m) - 1) << n;$$

Note: $k$ is not used.
Field extraction with a bit mask - field already in rightmost bits

Now, let’s use \(((1 \ll m) - 1) \ll n;\) to create the mask; In this example, \(m = 4, n = 0\)

\[
\begin{align*}
\text{mov } r3, \#4 & \quad // \ m = 4 \\
\text{mov } r4, \#0 & \quad // \ n = 0 \\
\text{mov } r2, \#1 \\
\text{lsl } r2, r2, r3 \\
\text{sub } r2, r2, \#1 \\
\text{lsl } r2, r2, r4 & \quad // \text{ omit this instruction.} \\
& \quad // \text{ not necessary, since } n = 0
\end{align*}
\]

\[
\begin{array}{cccccccccccc}
xxxx & xxxx & xxxx & xxxx & xxxx & xxxx & xxxx & xxxx & zzzz & r0 \\
\text{and } 0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 1111 & \text{and } r1, r0, r2
\end{array}
\]

\[
\begin{array}{cccccccccccc}
0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0000 & zzzz & r1
\end{array}
\]
Field Extraction

Field extraction with a bit mask - field not in rightmost bits

if you want to work with the next nibble to the left, remove (clear) all other bits and work with only next to last four bits

Let’s use \(((1 \ll m) - 1) \ll n;\) to create the mask;

In this example, \(m = 4, n = 4\)

```
  mov r3, #4                 // m = 4
  mov r4, #4                 // n = 4
  mov r2, #1
  lsl r2, r2, r3
  sub r2, r2, #1
  lsl r2, r2, r4
```
Field Extraction

Field extraction with a bit mask - field not in rightmost bits

if you want to work with the next nibble to the left, remove (clear) all other bits and work with only next to last four bits

```
  yyyyy yyyyy yyyyy yyyyy yyyyy zzzzz yyyy r0
  and 0000 0000 0000 0000 0000 0000 1111 0000  and r1, r0, r2
  0000 0000 0000 0000 0000 0000 zzzzz 0000 r1
```

it is usually easier to work with the field when it is shifted to the right

```
  0000 0000 0000 0000 0000 0000 0000 0000 zzzzz 0000 r1
  mov r2, r1, r4
  0000 0000 0000 0000 0000 0000 0000 0000 0000 zzzzz r2
```
Field Extraction

Field extraction with a bit mask - field not in rightmost bits

even better, you can shift and then mask

Exercise: Create the mask in r2

```
yyy yyy yyy yyy yyy yyy zzzz yyy r0
0000 yyy yyy yyy yyy yyy yyy yyy zzzz r1
and 0000 0000 0000 0000 0000 0000 0000 1111 0000 0000 0000 0000 0000 0000 zzzz r5
```
Field Extraction

Field extraction without a mask

a left shift and then a right shift can isolate the desired field

```assembly
mov r2, #8        // starting position of desired field
mov r3, #4        // width of field to extract
rsub r2, r2, #32
rsub r3, r3, #32
 yyyy yyyy yyyy yyyy yyyy yyyy zzzz yyyy r0
 zzzz yyyy 0000 0000 0000 0000 0000 0000 r1
0000 0000 0000 0000 0000 0000 0000 zzzz r1
```

```assembly
mov r1, r0, lsr r2
mov r1, r1, lsr r3
```
Field Extraction

Field extraction without a mask

A left shift and then a right shift can isolate the desired field:

```
mov r2, #8      // starting position of desired field
mov r3, #4      // width of field to extract
rsub r2, r2, #32
rsub r3, r3, #32
```

Use a logical right shift for an unsigned value or an arithmetic right shift for a signed value, which provides sign extension:

```
0001 0010 0011 0100 0101 0110 1101 1000  r0
1101 1000 0000 0000 0000 0000 0000 0000  r1
1111 1111 1111 1111 1111 1111 1111 1101  r1
```

```
mov r1, r0, lsl r2
mov r1, r1, asr r3
```
Field Insertion

Field insertion with a bit mask

• first, if necessary, clear out the field into which you wish to insert (the mask will have 0s in the bit positions we wish to clear and 1s in all other bit positions)

Exercise: Write code to create the mask 0xffffffff0f and place it in r1.

\[
\begin{align*}
\text{x} & \text{x} & \text{x} & \text{x} & \text{x} & \text{x} & \text{x} & \text{x} & \text{x} & \text{x} & \text{x} & \text{x} & \text{x} & \text{x} & \text{r0} \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & \text{r1} \\
\text{x} & \text{x} & \text{x} & \text{x} & \text{x} & \text{x} & \text{x} & \text{x} & \text{x} & \text{x} & \text{x} & 0 & 0 & 0 & \text{x} & \text{x} & \text{r2}
\end{align*}
\]
Field Insertion

Field insertion with a bit mask

- alternatively, you can use the bic instruction with a selection mask of 0xf0, that is, with a mask that uses 1s to define the field that we will insert

Exercise: Write code to create the mask 0xf0 and place it in r1

```
xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx r0
bic  r2, r0, r1

xxxx xxxx xxxx xxxx xxxx xxxx xxxx 0000 xxxx r2
```
Field Insertion

Field insertion with a bit mask
if necessary, shift the new value to the correct field position before inserting:

```
mov r2, #4
0000 0000 0000 0000 0000 0000 0000 zzzz r3
moval r4, r3, lsl r2
0000 0000 0000 0000 0000 0000 0000 zzzz 0000 r4
```

then insert the new value into the field

```
0000 0000 0000 0000 0000 0000 0000 zzzz 0000 r4
xxxx xxxx xxxx xxxx xxxx xxxx xxxx 0000 xxxx r1
```
```
or r5, r4, r1
```
```
xxxx xxxx xxxx xxxx xxxx xxxx xxxx zzzz xxxx r5
```
Example of Field Extraction and Field Insertion

(Taken from the Internet)

A perfect example for when you'd want to combine multiple values into a single variable is when you're doing graphics work with 32-bit color.

In a 32-bit color dword, there are four distinct values.

The low byte (bits 0 through 7) is the value for blue.

The next most significant byte is a value for green, then a byte for blue, and finally, the high byte is an alpha (transparency) value. So the color dword looks like this in memory:

AAAA AAAA RRRR RRRR GGGG GGGG B BBBB B BBBB
Example of Field Extraction and Field Insertion

Now, suppose you have a 32-bit integer called dwColor, and you want to extract the value for red. How should we do it? What we need is a way to eliminate the other three bytes, and leave the red byte untouched.

We will define a mask, which has 0s where we want to erase information, and 1s where we want to save information. Since we want to extract the red byte, our mask would look like this:

```
0000 0000 1111 1111 0000 0000 0000 0000
```
Example of Field Extraction and Field Insertion

We need to convert this to hex since we can't write binary numbers directly into C code

The hex equivalent is 0x00FF0000. If we use the bitwise AND on dwColor and our mask, we get the following result:

dwColor: AAAA AAAA RRRR RRRR GGGG GGGG BBBB BBBB
mask: & 0000 0000 1111 1111 0000 0000 0000 0000

result: 0000 0000 RRRR RRRR 0000 0000 0000 0000
Example of Field Extraction and Field Insertion

There is just one problem with this. To use the red byte by itself like we want, it would have to be the low byte. But it's not – it is 16 bits up in the dword. All we need now is a shift right by 16 places, and we're all set:

Previous: 0000 0000 RRRR RRRR 0000 0000 0000 0000
Shift: >> 16

Result: 0000 0000 0000 0000 0000 0000 RRRR RRRR
Example: Inserting and Combining Values

Suppose we want to reset a byte in a color dword. Maybe we have a color dword that represents the color (214, 53, 240), and we want to change it to (214, 166, 240).

The first step is to clear the green byte to 0.

To see how to rewrite that byte to contain the desired green byte, consider the truth table for the bitwise OR. Remember that any value ORed with 0 is that value.

So we must create a new mask to use. It will have zeroes wherever the color dword is already defined, and it will have an actual color value wherever the color dword has a 0.
Example of Field Extraction and Field Insertion

dwColor:     AAAA  AAAA  RRRR  RRRR  0000  0000  B BBB  B BBB
mask:       | 0000  0000  0000  0000  GGGG  GGGG  0000  0000
-------------
result:     AAAA  AAAA  RRRR  RRRR  GGGG  GGGG  B BBB  B BBB

In this case, the mask is the green byte, located at the appropriate position so that it merges correctly with the color dword. Use a bitwise shift to shift the green byte into the position we want it in. In the example above, the green byte is located eight bits above the low byte, so the shift operation you'd use would look like this:

Previous:  0000 0000 0000 0000 0000 0000  GGGG  GGGG
Shift:               << 8
---------------------
Result:  0000 0000 0000 0000  GGGG  GGGG  0000 0000