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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A AND B</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>0</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>A OR B</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>A EOR B</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>NOT A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
### Instructions for Logical Operations

- **Bitwise logical AND – AND**

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

```plaintext
31 30

```
1 1
```

```
0 1
```

```
0 1
```

\[ r0 = r1 \cdot r2 \]
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**

<table>
<thead>
<tr>
<th>EOR</th>
<th>r0, r1, r2</th>
<th>@ r0 = r1 ⊕ r2 (r1 EOR r2)</th>
</tr>
</thead>
</table>

![Bitwise logical Exclusive OR – EOR diagram](image)

**Example:**
- **r0:** 1 1
- **r1:** 1 0 1 0
- **r2:** 0 1 1 0
- **r0** AFTER operation: 0 1 1 0

**Operation:**
- **r0 ⊕ r2**
- **r0** ← **r0 ⊕ r2**

**Result:**
- **r0** AFTER operation: 0 1 1 0

**Equation:**
- **r0 = r1 ⊕ r2**
- **r0** ← **r1 ⊕ r2**

**Diagram:**
- Represents the bitwise logical Exclusive OR operation between registers r0, r1, and r2.
Instructions for Logical Operations

- Bitwise logical inversion
- **MVN MoVe Negative** – like MOV but moves the one’s complement of a value (bitwise inversion) to a register

### Example Instructions

- `MVN r0, r0` @ r0 = r0’ (NOT r0)
- `MVN r0, r1` @ r0 = r1’ (NOT r1)
**Bit Manipulation**

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

```
  0 1  .  .  .  .  1 0 0 0 1 1 0 0
  31 30  6  5  4  3  2  1  0
```

- 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

```
  0 1  .  .  .  .  1 0 0 0 0 1 0 0
  31 30  6  5  4  3  2  1  0
```

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

```
  1 1  .  .  .  .  1 1 0 0 0 1 1 1
  31 30  6  5  4  3  2  1  0
  mask
```
e.g. Clear bits 3 and 4 of the value in r1 (continued)
Write an assembly language program to clear bits 3 and 4 of the value in r1

```assembly
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 (Bit Clear) instruction allows us to define a mask with 1’s in the positions we want to clear

```assembly
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

```assembly
BIC    r1, r1, #0x00000018            @ r1 = r1 . NOT(0x00000018)
```
**Bit Manipulation**

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

  ![Example](image)

  - 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

  ![Mask](image)

  - Perform a bitwise logical OR of the value with the mask
e.g. Set bits 2 and 4 of the value in r1 (continued)

Bit Manipulation

- Bitwise Logic, Shifts and Rotates

Example:

- r1 before: 011001100
- Mask (e.g. r2): 001010100
- r1 after: 101111000
### Program 5.2 – Set Bits

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

```assembly
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.

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

\[
\begin{array}{llllllllll}
0 & 1 & \bullet & \bullet & \bullet & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
\oplus & \oplus & \bullet & \bullet & \bullet & \oplus & \oplus & \oplus & \oplus & \oplus & \oplus & \oplus \\
0 & 0 & \bullet & \bullet & \bullet & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
= & = & \bullet & \bullet & \bullet & = & = & = & = & = & = & = \\
0 & 1 & \bullet & \bullet & \bullet & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
\end{array}
\]
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.
Design and write an assembly language program that will make the ASCII character stored in r0 upper case.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
<td>DLE</td>
<td>SPACE</td>
<td>0</td>
<td>@</td>
<td>P</td>
<td>`</td>
<td>p</td>
</tr>
<tr>
<td>1</td>
<td>SOH</td>
<td>DC1</td>
<td>!</td>
<td>1</td>
<td>A</td>
<td>Q</td>
<td>a</td>
<td>q</td>
</tr>
<tr>
<td>2</td>
<td>STX</td>
<td>DC2</td>
<td>&quot;</td>
<td>2</td>
<td>B</td>
<td>R</td>
<td>b</td>
<td>r</td>
</tr>
<tr>
<td>3</td>
<td>ETX</td>
<td>DC3</td>
<td>#</td>
<td>3</td>
<td>C</td>
<td>S</td>
<td>c</td>
<td>s</td>
</tr>
<tr>
<td>4</td>
<td>EOT</td>
<td>DC4</td>
<td>$</td>
<td>4</td>
<td>D</td>
<td>T</td>
<td>d</td>
<td>t</td>
</tr>
<tr>
<td>5</td>
<td>ENQ</td>
<td>NAK</td>
<td>%</td>
<td>5</td>
<td>E</td>
<td>U</td>
<td>e</td>
<td>u</td>
</tr>
<tr>
<td>6</td>
<td>ACK</td>
<td>SYN</td>
<td>&amp;</td>
<td>6</td>
<td>F</td>
<td>V</td>
<td>f</td>
<td>v</td>
</tr>
<tr>
<td>7</td>
<td>BEL</td>
<td>ETB</td>
<td>'</td>
<td>7</td>
<td>G</td>
<td>W</td>
<td>g</td>
<td>w</td>
</tr>
<tr>
<td>8</td>
<td>BS</td>
<td>CAN</td>
<td>(</td>
<td>8</td>
<td>H</td>
<td>X</td>
<td>h</td>
<td>x</td>
</tr>
<tr>
<td>9</td>
<td>HT</td>
<td>EM</td>
<td>)</td>
<td>9</td>
<td>I</td>
<td>Y</td>
<td>i</td>
<td>y</td>
</tr>
<tr>
<td>A</td>
<td>LF</td>
<td>SUB</td>
<td>*</td>
<td>:</td>
<td>J</td>
<td>Z</td>
<td>j</td>
<td>z</td>
</tr>
<tr>
<td>B</td>
<td>VT</td>
<td>ESC</td>
<td>+</td>
<td>@</td>
<td>K</td>
<td>[</td>
<td>k</td>
<td>{</td>
</tr>
<tr>
<td>C</td>
<td>FF</td>
<td>FS</td>
<td>,</td>
<td>&lt;</td>
<td>L</td>
<td>\</td>
<td>l</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>CR</td>
<td>GS</td>
<td>-</td>
<td>=</td>
<td>M</td>
<td>]</td>
<td>m</td>
<td>}</td>
</tr>
<tr>
<td>E</td>
<td>SO</td>
<td>RS</td>
<td>.</td>
<td>&gt;</td>
<td>N</td>
<td>^</td>
<td>n</td>
<td>~</td>
</tr>
<tr>
<td>F</td>
<td>SI</td>
<td>US</td>
<td>/</td>
<td>?</td>
<td>O</td>
<td>_</td>
<td>o</td>
<td>DEL</td>
</tr>
</tbody>
</table>
Logical Shift Left by 1 bit position

ARM MOV instruction allows a source operand, Rm, to be shifted left by $n = 0 \ldots 31$ bit positions before being stored in the destination operand, Rd

$$\text{MOV Rd, Rm, LSL } \#n$$

- LSB of Rd is set to zero, MSB of Rm is discarded
Logical Shift Right by 1 bit position

ARM MOV instruction allows a source operand, Rm, to be shifted right by $n = 0 \ldots 31$ bit positions before being stored in the destination operand, Rd

$$\text{MOV} \ R_d, \ R_m, \ \text{LSR} \ #n$$

- MSB of Rd is set to zero, LSB of Rm 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
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 MOVVS instead of MOV

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

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

![Binary representation](image)

$r1 = 6 \times 2^2 = 24$
We can express multiplication by any value as the **sum** of the results of multiplying the value by different powers of 2.

**Example**
- \( a \times 12 = a \times (8 + 4) = a \times (2^3 + 2^2) = (a \times 2^3) + (a \times 2^2) \)
- \( a \times 12 = (a \ll 3) + (a \ll 2) \)

Is there a simple way to determine which powers of 2 we need to use for our partial products?
Design and write an assembly language program that will multiply the value in r1 by 12 and store the result in r0

\[
\text{start:}
\begin{align*}
\text{MOV} & \quad \text{r0, r1, LSL #3} @ \ r0 = r1 \times 2^3 \\
\text{ADD} & \quad \text{r0, r0, r1, LSL #2} @ \ r0 = r0 + r1 \times 2^2
\end{align*}
\text{stop:} \quad B \ \text{stop}
\]

[ASIDE] We can also formulate instructions to efficiently compute \( Rm \times (2^n-1) \) or \( Rm \times (2^n+1) \), saving one instruction

\[
\begin{align*}
\text{ADD} & \quad \text{r0, r1, r1, LSL #3} @ \ r0 = r1 \times 9 \\
\text{RSB} & \quad \text{r0, r1, r1, LSL #3} @ \ r0 = r1 \times 7
\end{align*}
\]
Arithmetic Shift Right

- Arithmetic Shift Right by 1 bit position

![Diagram showing an arithmetic shift right operation](image)

- ARM MOV instruction allows a source operand, \( R_m \), to be shifted right by \( n = 0 \ldots 31 \) bit positions before being stored in the destination operand, \( R_d \)

\[
\text{MOV } R_d, \ R_m, \ \text{ASR } \#n
\]

- 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, \( Rm \), to be rotated right by \( n = 0 \ldots 31 \) bit positions before being stored in the destination operand, \( Rd \)

\[
\text{MOV } Rd, \ Rm, \ ROR \ #n
\]
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 $2^n$, 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 $2^n$
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.
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.
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

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Cond</td>
<td>0 0</td>
</tr>
<tr>
<td>29-26</td>
<td>Immediate bit</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>25</td>
<td>S</td>
<td>0</td>
</tr>
<tr>
<td>24-21</td>
<td>Opcode field</td>
<td>0100</td>
</tr>
<tr>
<td>20-11</td>
<td>Operand 1</td>
<td></td>
</tr>
<tr>
<td>10-</td>
<td>Destination</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Operand 2</td>
<td></td>
</tr>
</tbody>
</table>

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.
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:

![Rotation Diagram]

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.
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

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]
MOVSD 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 MOVSD, 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.