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 XOR 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)
```

![Diagram showing bitwise AND operation](image)
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 MoVe Negative** – 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)
```

![Diagram](image)
- e.g. Clear bits 3 and 4 of the value in r1

```
before

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

```
after

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

- 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

```
mask

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

- Perform a bitwise logical AND of the value with the mask
- 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, =0xFFFFFFFFE7 ; 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

```
LDR r2, =0x000000018 ; 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, #0x000000018 ; r1 = r1 . NOT(0x000000018)
```
**Bit Manipulation**

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

  Before:
  
  | 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 |
  |----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
  | 0  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

  After:
  
  | 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 |
  |----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
  | 0  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

  • 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:
  
  | 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 |
  |----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
  | 0  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

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

  ![Diagram showing bit manipulation](image)

  - 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

  ![Mask](image)

  - 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)

```
0 1          1 0 0 1 1 0 0
⊕ ⊕          ⊕ ⊕ ⊕ ⊕ ⊕ ⊕ ⊕ ⊕ ⊕ ⊕
0 0          0 0 0 1 0 1 0
= =          = = = = = = = = = =
0 1          1 0 0 0 1 1 0
```

r1 before | mask (e.g. r2) | r1 after
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 immediate operand in the ORR instruction

```
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.

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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>
</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>
</tr>
<tr>
<td>2</td>
<td>STX</td>
<td>DC2</td>
<td>“</td>
<td>2</td>
<td>B</td>
<td>R</td>
<td>b</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>
</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>
</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>
</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>
</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>
</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>
</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>
</tr>
<tr>
<td>A</td>
<td>LF</td>
<td>SUB</td>
<td>*</td>
<td>:</td>
<td>J</td>
<td>Z</td>
<td>j</td>
</tr>
<tr>
<td>B</td>
<td>VT</td>
<td>ESC</td>
<td>+</td>
<td>;</td>
<td>K</td>
<td>[</td>
<td>k</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>
</tr>
<tr>
<td>D</td>
<td>CR</td>
<td>GS</td>
<td>-</td>
<td>=</td>
<td>M</td>
<td>]</td>
<td>m</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>
</tr>
<tr>
<td>F</td>
<td>SI</td>
<td>US</td>
<td>/</td>
<td>?</td>
<td>O</td>
<td>_</td>
<td>o</td>
</tr>
</tbody>
</table>
Logical Shift Left by 1 bit position

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

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

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

  \[
  \text{MOV Rd, Rm, LSR \#n}
  \]

  - 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 MOVS instead of MOV

MOVS Rd, Rm, LSL #n
MOVS 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

```
without parity bit

0 0 0 1 0 1 1 0

with parity bit

1 0 0 1 0 1 1 0
```
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

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

\[ r1 = 6 \times 2^2 = 24 \]
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
- \( a \times 12 = a \times (8 + 4) = a \times (2^3 + 2^2) = (a \times 2^3) + (a \times 2^2) \)
- \( a \times 12 = (a \ll 3) + (a \ll 2) \)

- Is there a simple way to determine which powers of 2 we need to use for our partial products?

\[
\begin{array}{cccccccccccc}
0 & 0 & \bullet & \bullet & \bullet & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
2^{31} & 2^{30} & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0
\end{array}
\]

12 = 2^3 + 2^2
Design and write an assembly language program that will multiply the value in r1 by 12 and store the result in r0

![Program 5.7 – Shift And Add Multiplication]

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

- Arithmetic 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, \( Rd \)

\[
\text{MOV} \ Rd, Rm, \text{ASR} \ #n
\]

- MSB of \( Rd \) is set to MSB of \( Rm \), LSB of \( Rm \) 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, $Rd$

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