Topic 6: Bitwise Instructions
Overview

- Logic Instructions
- Shifts and Rotates
- ARM Arithmetic Datapath
Logical Operators

- Basic logical operators:
  - **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

- In general, can define them to accept >2 inputs, but in the case of ARM assembly, both of these accept exactly 2 inputs and produce 1 output
  - Again, rigid syntax, simpler hardware
Logical Operators

- **Truth Table**: standard table listing all possible combinations of inputs and resultant output for each

- Truth Table for AND, OR and XOR

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>AND</th>
<th>OR</th>
<th>XOR</th>
<th>BIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Note that 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:

\[ \begin{array}{cccccccccccc}
1011 & 0110 & 1010 & 0100 & 0011 & 1101 & 1001 & 1010 \\
0000 & 0000 & 0000 & 0000 & 0000 & 1111 & 1111 & 1111 \\
\end{array} \]

mask:

\[ \begin{array}{cccccccccccc}
0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 1101 & 1001 & 1010 \\
1111 & 1111 & 1111 & 1111 \\
\end{array} \]

The result of ANDing these:

\[ \begin{array}{cccccccccccc}
0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 1101 & 1001 & 1010 \\
1101 & 1001 & 1010 \\
\end{array} \]
Uses for Logical Operators

- Similarly, note that 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 0x0000FFF 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

- Additionally, note that XORing a bit with 1 produces flips the bit (0 -> 1, 1 -> 0) at the output while XORing a bit with 0 produces the original bit.

- It tells whether two bits are unequal.

- It is an optional bit-flipper (the deciding input chooses whether to invert the data input).
Uses for Logical Operators

- Finally, note that 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).
Syntax of Instructions:
1  2, 3, 4
where:
1) instruction by name
2) operand getting result ("destination")
3) 1st operand for operation ("source1")
4) 2nd operand for operation ("source2")

Syntax is rigid (for the most part):
- 1 operator, 3 operands
- Why? Keep Hardware simple via regularity
Bitwise Logic Operations

- **Bitwise AND in Assembly**
  - Example: `AND r0, r1, r2` (in ARM)
  - Equivalent to: `a = b & c` (in C)
  - where ARM registers `r0, r1, r2` are associated with C variables `a, b, c`

- **Bitwise OR in Assembly**
  - Example: `ORR r3, r4, r5` (in ARM)
  - Equivalent to: `d = e | f` (in C)
  - where ARM registers `r3, r4, r5` are associated with C variables `d, e, f`
Bitwise Logic Operations

- **Bitwise XOR in Assembly**
  - Example: `EOR r0, r1, r2` (in ARM)
  - Equivalent to: `a = b ^ c` (in C)
  - where ARM registers `r0, r1, r2` are associated with C variables `a, b, c`

- **Bitwise Clear in Assembly**
  - Example: `BIC r3, r4, r5` (in ARM)
  - Equivalent to: `d = e & (!f)` (in C)
  - where ARM registers `r3, r4, r5` are associated with C variables `d, e, f`
Assignment Instructions

 Assignment in Assembly

Example:  MOV r0,r1  (in ARM)
Equivalent to:  a = b  (in C)
where ARM registers r0 are associated with C variables a

Example:  MOV r0,#10  (in ARM)
Equivalent to:  a = 10  (in C)
Assignment Instructions

- MOVN – Move Negative – moves one complement of the operand into the register.

Assignment in Assembly

- Example: MOVN r0, #0 (in ARM)
- Equivalent to: \( a = -1 \) (in C)

where ARM registers \( r0 \) are associated with C variables \( a \)

Since \(~0x00000000 == 0xFFFFFFFF\)
Shifts and Rotates

- **LSL** – logical shift left by n bits – multiplication by \(2^n\)

- **LSR** – logical shift right by n bits – unsigned division by \(2^n\)

- **ASR** – arithmetic shift right by n bits – signed division by \(2^n\)

- **ROR** – rotate right by n bits – 32 bit rotate
Shifts and Rotates

- **Shifting in Assembly**
  
  Examples:
  
  ```
  MOV    r4, r6, LSL #4 ; r4 = r6 << 4
  MOV    r4, r6, LSR #8 ; r4 = r6 << 8
  ```

- **Rotating in Assembly**
  
  Examples:
  
  ```
  MOV    r4, r6, ROR #12
  ; r4 = r6 rotated right 12 bits
  ; r4 = r6 rotated left by 20 bits (32 - 12)
  
  Therefore no need for rotate left.
  ```
Variable Shifts and Rotates

- Also possible to shift by the value of a register
- Examples:

  \[
  \text{MOV } r4, r6, \text{ LSL } r3 \\
  \hfill ; \ r4 = r6 << \text{ value specified in } r3 \\
  \text{MOV } r4, r6, \text{ LSR } #8 \hfill ; \ r4 = r6 >> 8
  \]
Constant Multiplication

- Constant multiplication is often faster using shifts and additions
  
  MUL r0, r2, #8 ; r0 = r2 * 8

  Is the same as:

  MOV r0, r2, LSL #3 ; r0 = r2 * 8

- Constant division

  MOV r1, r3, ASR #7 ; r1 = r3/128

  Vs.

  MOV r1, r3, LSR #7 ; r1 = r3/128

  The first treats the registers like signed values (shifts in MSB).
  The latter treats data like unsigned values (shifts in 0).
  int vs unsigned int >>
Constant Multiplication

- Constant multiplication with subtractions

  \[ \text{MUL } r0, r2, \#7 \quad ; \quad r0 = r2 \times 7 \]

  Is the same as:

  \[ \text{RSB } r0, r2, r2, \text{LSL} \#3 \quad ; \quad r0 = r2 \times 7 \]
  \[ ; \quad r0 = -r2 + 8\times r2 = 7\times r2 \]

  \[ \text{RSB } r0, r1, r2 \text{ is the same as} \]
  \[ \text{SUB } r0, r2, r1 \quad ; \quad r0 = r2 - r1 \]

  Multiply by 35:

  \[ \text{ADD } r9, r8, r8, \text{LSL} \#2; \quad r9 = r8 \times 5 \]

  \[ \text{RSB } r10, r9, r9, \text{LSL} \#3 \quad ; \quad r10 = r9 \times 7 \]

  Why have RSB? B/C only the second source operand can be shifted.
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
- 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

Using a Barrel Shifter

Operand 1 → Operand 2 → Barrel Shifter → ALU → Result
Loading Constants

- Constants can only be 8 bit numbers
  - Allows assignment of numbers between 0-255 or [0x00 – 0xFF]
  - Why? Constants stored in code and there is limited space (only 32 bits).
- Assignment in Assembly
  - Example: MOV r0, #0xFF (in ARM)
  - Equivalent to: a = 255 (in C)
  - where ARM registers r0 C variable a
Immediate Constants

- The data processing instruction format has 12 bits available for operand2

- 4 bit rotate value (0-15) is multiplied by two to give range 0-30 in steps of 2

- Rule to remember is “8-bits rotated right by an even number of bit positions”
## Loading Constants

<table>
<thead>
<tr>
<th>Rotate Value</th>
<th>Binary</th>
<th>Decimal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000000000000000000000000000xxxxxxxxxxxx</td>
<td>0-255</td>
<td>0-0xFF</td>
</tr>
<tr>
<td>Right, 30 bits</td>
<td>00000000000000000000000000000000xxxxxxxxxxxx00</td>
<td>4-1020</td>
<td>0x4-0x3FC</td>
</tr>
<tr>
<td>Right, 28 bits</td>
<td>00000000000000000000000000000000xxxxxxxxxxxx0000</td>
<td>16-4080</td>
<td>0x10-0xFF0</td>
</tr>
<tr>
<td>Right, 26 bits</td>
<td>00000000000000000000000000000000xxxxxxxxxxxx000000</td>
<td>128-16320</td>
<td>0x40-0x3FC0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Right, 8 bits</td>
<td>xxxxxxxxx0000000000000000000000000000000000</td>
<td>16777216-</td>
<td>0x10000000-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>255x2^24</td>
<td>0xFF000000</td>
</tr>
<tr>
<td>Right, 6 bits</td>
<td>xxxxxxx0000000000000000000000000000000000xx</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Right, 4 bits</td>
<td>xxx00000000000000000000000000000000000000000xxxx</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Right, 2 bits</td>
<td>xx000000000000000000000000000000000000000000000000000000xxxxxxx</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- This scheme can generate a lot, but not all, constants.
  - ~50% of constants between [-15,15];
  - ~90% between [-512,512]

- Others must be done using literal pools (more on that later)
Conclusion

- **Instructions so far:**
  - Previously:
    - ADD, SUB, MUL, MULA, [U|S]MULL, [U|S]MLAL
  - New instructions:
    - RSB
    - AND, ORR, EOR, BIC
    - MOV, MVN
    - LSL, LSR, ASR, ROR

- Shifting can only be done on the second source operand
- Constant multiplications possible using shifts and addition/subtractions