Bitwise Instructions

CSE 30: Computer Organization and Systems Programming

Dept. of Computer Science and Engineering
University of California, San Diego
Overview

- Bitwise 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>A</th>
<th>B</th>
<th>A AND B</th>
<th>A OR B</th>
<th>A XOR B</th>
<th>A BIC B</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>
Uses for Logical Operators

- 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:
    
    | 1011 0110 1010 0100 0011 1101 1001 1010 |
    | 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

- 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.
- 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

- 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: \texttt{MOV r0, r1} \ (in ARM)
  Equivalent to: \( a = b \) \ (in C)
  where ARM registers \( r0 \) are associated with C variables \( a \)

- Example: \texttt{MOV r0, #10} \ (in ARM)
  Equivalent to: \( a = 10 \) \ (in C)
Assignment Instructions

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

**Assignment in Assembly**

- Example: \( \text{MVN } r0, \#0 \) (in ARM)
  
  Equivalent to: \( a = -1 \) (in C)

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

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

- **LSL** – logical shift by \( n \) bits – multiplication by \( 2^n \)
  
  \[ \begin{array}{cccccc}
  & & & & & \\
  & & & & &
  \end{array} \quad \begin{array}{ccc}
  C & & 0
  \end{array} \]

- **LSR** – logical shift by \( n \) bits – unsigned division by \( 2^n \)
  
  \[ \begin{array}{cccccc}
  & & & & & \\
  & & & & &
  \end{array} \quad \begin{array}{ccc}
  0 & & C
  \end{array} \]

- **ASR** – logical shift by \( n \) bits – signed division by \( 2^n \)
  
  \[ \begin{array}{cccccc}
  & & & & & \\
  & & & & &
  \end{array} \quad \begin{array}{ccc}
  \uparrow & & C
  \end{array} \]

- **ROR** – logical shift by \( n \) bits – 32 bit rotate
  
  \[ \begin{array}{cccccc}
  & & & & & \\
  & & & & &
  \end{array} \quad \begin{array}{ccc}
  \uparrow & & C
  \end{array} \]
Shifts and Rotates

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

- **Rotating in Assembly**
  
  Examples:
  
  ```assembly
  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:
  
  ```assembly
  MOV   r4, r6, LSL r3
  ; r4 = r6 << value specified in r3
  MOV   r4, r6, LSR #8 ; r4 = r6 >> 8
  ```

- Rotating in Assembly

  - Examples:
    
    ```assembly
    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.
    ```
Constant Multiplication

- Constant multiplication is often faster using shifts and additions
  
  \[ \text{MUL } r0, r2, \#8 \ ; \ r0 = r2 * 8 \]

  Is the same as:

  \[ \text{MOV } r0, r2, \text{LSL} \#3 \ ; \ r0 = r2 * 8 \]

- Constant division

  \[ \text{MOV } r1, r3, \text{ASR} \#7 \ ; \ r1 = r3/128 \]

  Vs.

  \[ \text{MOV } r1, r3, \text{LSR} \#7 \ ; \ r1 = r3/128 \]

  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

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

Is the same as:

\[
\text{RSB } r0, r2, r2, \text{LSL} \#3 \; ; \; r0 = r2 \ast 7
\]

; \; r0 = -r2 + 8*r2 = 7*r2

RSB \( r0, r1, r2 \) is the same as

SUB \( r0, r2, r1 \) ; \; r0 = r1 - r2

Multiply by 35:

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

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

Why have RSB? B/C only the second source operand can be shifted.
Using a Barrel Shifter

Operand 1  Operand 2

- 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

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` and 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>000000000000000000000000xxxxxxxxxx</td>
<td>0-255</td>
<td>0-0xFF</td>
</tr>
<tr>
<td>Right, 30 bits</td>
<td>000000000000000000000000xxxxxxxxxx00</td>
<td>4-1020</td>
<td>0x4-0x3FC</td>
</tr>
<tr>
<td>Right, 28 bits</td>
<td>0000000000000000000000000xxxxxxxxxx0000</td>
<td>16-4080</td>
<td>0x10-0xFF0</td>
</tr>
<tr>
<td>Right, 26 bits</td>
<td>00000000000000000000000000xxxxxxxxxx000000</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>xxxxxxxx0000000000000000000000000000000000</td>
<td>16777216-255x2²⁴</td>
<td>0x10000000-0xF000000</td>
</tr>
<tr>
<td>Right, 6 bits</td>
<td>xxxxxxxx00000000000000000000000000000000xx</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Right, 4 bits</td>
<td>xxx00000000000000000000000000000000xxxx</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Right, 2 bits</td>
<td>xx000000000000000000000000000000000xxxxx</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