Overview

- Bitwise Logical Operations
  - OR
  - AND
  - XOR

- Shift Operations
  - Shift Left
  - Shift Right
  - Rotate
  - Field Extraction

Review: Assembly Variables: Registers

- Assembly Language uses registers as operands for data processing instructions

<table>
<thead>
<tr>
<th>Register No.</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>a1</td>
</tr>
<tr>
<td>r1</td>
<td>a2</td>
</tr>
<tr>
<td>r2</td>
<td>a3</td>
</tr>
<tr>
<td>r3</td>
<td>a4</td>
</tr>
<tr>
<td>r4</td>
<td>v1</td>
</tr>
<tr>
<td>r5</td>
<td>v2</td>
</tr>
<tr>
<td>r6</td>
<td>v3</td>
</tr>
<tr>
<td>r7</td>
<td>v4</td>
</tr>
<tr>
<td>r8</td>
<td>v5</td>
</tr>
<tr>
<td>r9</td>
<td>v6</td>
</tr>
<tr>
<td>r10</td>
<td>v7</td>
</tr>
<tr>
<td>r11</td>
<td>v8</td>
</tr>
<tr>
<td>r12</td>
<td>ip</td>
</tr>
<tr>
<td>r13</td>
<td>sp</td>
</tr>
<tr>
<td>r14</td>
<td>lr</td>
</tr>
<tr>
<td>r15</td>
<td>pc</td>
</tr>
</tbody>
</table>
**Bitwise Operations (#1/2)**

- Up until now, we’ve done arithmetic (add, sub, rsb) and data movement mov.
- All of these instructions view contents of register as a single quantity (such as a signed or unsigned integer)
- **New Perspective:** View contents of register as 32 bits rather than as a single 32-bit number

---

**Logical Operations**

- Operations on less than full words
  - Fields of bits or individual bits
- Think of word as **32 bits** vs. 2’s comp. integers or unsigned integers
- Need to extract bits from a word, or insert bits into a word
- Extracting via Shift instructions
  - C operators: << (shift left), >> (shift right)
- Inserting and inverting via And/OR/EOR instructions
  - C operators: & (bitwise AND), | (bitwise OR), ^ (bitwise EOR)

---

**Bitwise Operations (#2/2)**

- Since registers are composed of 32 bits, we may want to access individual bits rather than the whole.
- Introduce two new classes of instructions/Operations:
  - Logical Instructions
  - Shift Operators

---

**Logical Operators**

- **Operator Names:**
  - and, bic, orr, eor:
- **Operands:**
  - Destination : Register
  - Operand #1: Register
  - Operand #2: Register, or Shifted registers, or an immediate
  - Example: and a1, v1, v2
    - and a1, v1, v2, lsl #5
    - and a1, v1, v2, lsl v3
    - and a1, v1, #0x40
- **ARM Logical Operators are all bitwise,** meaning that bit 0 of the output is produced by the respective bit 0’s of the inputs, bit 1 by the bit 1’s, etc.
**Logical AND Operator (#1/3)**

° **AND:** 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:
  
  
  
  
  
  
  
  

**Logical AND Operator (#2/3)**

° The second bitstring in the example is called a **mask**. It is used to isolate the rightmost 8 bits of the first bitstring by masking out the rest of the string (e.g. setting it to all 0s).

° Thus, the and operator can be used to set certain portions of a bitstring to 0s, while leaving the rest alone.

  • In particular, if the first bitstring in the above example were in a1, then the following instruction would mask the rightmost 8 bits a and clears leftmost 24 bits:

  \[
  \text{and } a1, a1, #0xFF
  \]

**Logical AND Operator (#3/3)**

° **AND:** bit-by-bit operation leaves a 1 in the result only if both bits of the operands are 1. For example, if registers \(v_1\) and \(v_2\) are

\[
\begin{align*}
  v_1 &= 0000 0000 0000 0000 0000 1101 0000 0000_{\text{two}} \\
  v_2 &= 0000 0000 0000 0000 0011 1100 0000 0000_{\text{two}}
\end{align*}
\]

° After executing ARM instruction

\[
\text{and } a1, v1, v2 ; a1 \leftarrow v1 & v2
\]

° Value of register \(a1\)

\[
0000 0000 0000 0000 0000 1100 0000 0000_{\text{two}}
\]

**Logical BIC (AND NOT) Operator (#1/3)**

° **BIC (AND NOT):** Note that bicing a bit with 1 produces a 0 at the output while bicing a bit with 0 produces the original bit.

° This can be used to create a mask.

  • Example:

  \[
  1011 0110 1010 0100 0011 1101 1001 1010
  \]

  \[
  0000 0000 0000 0000 0000 0000 1111 1111
  \]

  • The result of bicing these two is:

  \[
  1011 0110 1010 0100 0011 1101 0000 0000
  \]
Logical BIC (AND NOT) Operator (#2/3)

° The second bitstring in the example is called a mask. It is used to isolate the leftmost 24 bits of the first bitstring by masking out the rest of the string (e.g. setting it to all 0s).

° Thus, the bic operator can be used to set certain portions of a bitstring to 0s, while leaving the rest alone.
  
  • In particular, if the first bitstring in the above example were in a1, then the following instruction would mask the leftmost 24 bits a and clears rightmost 8 bits:

\[
bic \ a1,a1,#0xFF
\]

Logical BIC (AND NOT) Operator (#3/3)

° BIC: bit-by-bit operation leaves a 1 in the result only if bit of the first operand is 1 and the second operands 0. For example, if registers v1 and v2 are

\[
\begin{array}{cccccccc}
0000 & 0000 & 0000 & 1100 & 0000 & 0000 \\
0000 & 0000 & 0000 & 1001 & 0011 & 1100 & 0000
\end{array}
\]

° After executing ARM instruction

\[
bic \ a1,v1,v2 \ ; \ a1 \leftarrow v1 \ \& \ \text{(not} \ v2)\]

° Value of register a1

\[
\begin{array}{cccccccc}
0000 & 0000 & 0110 & 0100 & 0000 & 0000 \\
0000 & 0000 & 0000 & 0000 & 0000 & 0000
\end{array}
\]

Quiz # 2 Result

<table>
<thead>
<tr>
<th>Title</th>
<th>N</th>
<th>% Correct Of:</th>
<th>Discrimination</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM add instruction</td>
<td>136</td>
<td>97 100 92</td>
<td>0.45</td>
<td>97.1% 17.0</td>
</tr>
<tr>
<td>ARM register set</td>
<td>136</td>
<td>95 100 88</td>
<td>0.62</td>
<td>95.6% 20.6</td>
</tr>
<tr>
<td>ARM store instruction</td>
<td>136</td>
<td>73 100 30</td>
<td>0.66</td>
<td>73.5% 44.3</td>
</tr>
<tr>
<td>Data type</td>
<td>136</td>
<td>94 100 86</td>
<td>0.28</td>
<td>94.9% 22.2</td>
</tr>
<tr>
<td>Fetch-decode-execute cycle</td>
<td>136</td>
<td>91 100 78</td>
<td>0.54</td>
<td>91.9% 27.4</td>
</tr>
<tr>
<td>Meaning of register</td>
<td>136</td>
<td>93 100 82</td>
<td>0.56</td>
<td>93.4% 25.0</td>
</tr>
</tbody>
</table>

Overall Mean: 91.1%

Computers in the News

Seagate NL35 500GB SATA hard drive

Model Number: ST3400832AS; Capacity:500 GB; Speed: 7200 rpm;
Seek time: 8.5 ms avg; Latency: 4.16 ms; Interface : SATA 3Gb/s

° Capacity is the amount of data that the drive can store, after formatting. Most disc drive companies, including Seagate, calculate disc capacity based on the assumption that 1 megabyte = 1000 kilobytes and 1 gigabyte=1000 megabytes.

° RPM is a measurement of how fast a hard disc’s platters are spinning (in revolutions per minute). The faster the spin rate, the less time it takes for the drive to read or write a given amount of data.

° Seek time is an average of how long a drive takes to move the read/write heads to a particular track on the disc. It includes controller overhead but does not include drive latency.

° The drive interface is the "language" or protocol a drive uses to communicate with a host computer or network. The three main types of drive interfaces are ATA (IDE), SCSI, and Fibre Channel). The ATA and SCSI interfaces have evolved to include many sub-types, which may or may not be backwardly compatible.
Logical OR Operator (#1/2)

- OR: Similarly, note that or'ing a bit with 1 produces a 1 at the output while or'ing a bit with 0 produces the original bit.
- This can be used to force certain bits of a string to 1s.
  - For example, if a1 contains 0x12345678, then after this instruction:
    ```
    orr a1, a1, #0xFF
    ```
  - … a1 contains 0x123456FF (e.g. the high-order 24 bits are untouched, while the low-order 8 bits are forced to 1s).

Logical OR Operator (#2/2)

- OR: bit-by-bit operation leaves a 1 in the result if either bit of the operands is 1. For example, if registers v1 and v2
  ```
  0000 0000 0000 0000 1101 0000 0000
  0000 0000 0000 0000 0011 1100 0000
  ```
- After executing ARM instruction
  ```
  ORR a1,v1,v2 ; a1 ← v1 | v2
  ```
- Value of register a1
  ```
  0000 0000 0000 0000 0011 1101 0000 0000
  ```

Logical XOR Operator (#1/2)

- EOR: xor'ing a bit with 1 produces its complement at the output while xor'ing a bit with 0 produces the original bit.
- This can be used to force certain bits of a string to invert.
  - For example, if a1 contains 0x12345678, then after this instruction:
    ```
    eor a1, a1, #0xFF
    ```
  - … a1 contains 0x12345687 (e.g. the high-order 24 bits are untouched, while the low-order 8 bits are forced to invert).

Logical XOR Operator (#2/2)

- EOR: bit-by-bit operation leaves a 1 in the result if bits of the operands are different. For example, if registers v1 and v2
  ```
  0000 0000 0000 0000 1101 0000 0000
  0000 0000 0000 0000 0011 1100 0000
  ```
- After executing ARM instruction
  ```
  eor a1,v1,v2 ; a1 ← v1 ^ v2
  ```
- Value of register a1
  ```
  0000 0000 0000 0000 0011 0001 0000 0000
  ```
Shift Operations (#1/3)
° Move (shift) all the bits in a word to the left or right by a number of bits, filling the emptied bits with 0s.
° Example: shift right by 8 bits

Example: shift left by 8 bits

Shift Operations (#2/3)
° Move (shift) all the bits in a word to the right by a number of bits, filling the emptied bits with the sign bits.
° Example: Arithmetic shift right by 8 bits

Shift Operations (#3/3)
° Move (shift) all the bits in a word to the right by a number of bits, filling the emptied bits with the bits falling of the right.
° Example: rotate right by 8 bits

ARM Shift Instructions
° ARM does not have separate shift instruction.
° Shifting operations is embedded in almost all other data processing instructions.
° Pure Shift operation can be obtained via the shifted version of mov Instruction.
° Data Processing Instruction with Shift Feature Syntax:

Example:
```assembly
add a1, v1,v3, lsl #8 ; a1 ← v1 + (v3 << 8 bits)
add a1, v1,v3, lsl v4 ; a1 ← v1 + (v3 << v4 bits)
```
ARM Shift Variants (#1/4)

1. **lsl** (logical shift left): shifts left and fills emptied bits with 0s

\[
\text{mov a1, v1, lsl #8 ; a1} \leftarrow \text{v1} \ll 8 \text{ bits}
\]

\[
\text{mov a1, v1, lsl v2 ; a1} \leftarrow \text{v1} \ll \text{v2} \text{ bits}
\]

2. **lsr** (logical shift right): shifts right and fills emptied bits with 0s

\[
\text{mov a1, v1, lsr #8 ; a1} \leftarrow \text{v1} \gg 8 \text{ bits}
\]

\[
\text{mov a1, v1, lsr v2 ; a1} \leftarrow \text{v1} \gg \text{v2} \text{ bits}
\]

Destination

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

ARM Shift Variants (#2/4)

3. **asr** (arithmetic shift right): shifts right and fills emptied bits by sign extending

\[
\text{mov a1, v1, asr #8 ; a1} \leftarrow \text{v1} \gg 8 \text{ bits}
\]

\[
\text{mov a1, v1, asr v2 ; a1} \leftarrow \text{v1} \gg \text{v2} \text{ bits}
\]

Sign bit shifted in

Destination

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

ARM Shift Variants (#3/4)

4. **ror** (rotate right): shifts right and fills emptied bits by bits falling of the right. (bits wrap around)

\[
\text{mov a1, v1, ror #8 ; a1} \leftarrow \text{v1} \gg 8 \text{ bits}
\]

\[
\text{mov a1, v1, ror v2 ; a1} \leftarrow \text{v1} \gg \text{v2} \text{ bits}
\]

\[
\text{mov a1, v1, ror} \leftarrow \text{v1}[31](31-v2) \leftarrow \text{v1}[v2:0]
\]

Rotate amount between 1 to 31

ARM Shift Variants (#4/4)

4. **rrx** (rotate right through carry): This operation uses the CPSR C flag as a 33rd bit for rotation. (wrap around through carry)

\[
\text{mov a1, v1, rrx ; a1} \leftarrow \text{v1} \gg 1 \text{ bit}
\]

\[
\text{mov a1, v1, rrx ; a1} \leftarrow \text{v1} \gg 1 \text{ bit}
\]

\[
\text{mov a1, v1, rrx ; a1} \leftarrow \text{v1} \gg 1 \text{ bit}
\]

Rotate Right through Carry

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Sign bit shifted in

Destination

<table>
<thead>
<tr>
<th>C</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

• Only Rotation by one bit is allowed
• Encoded as **ror #0**
Isolation with Shift Instructions (#1/2)

° Suppose we want to isolate byte 0 (rightmost 8 bits) of a word in \( a_0 \). Simply use:

\[
\text{and } a_0, a_0, \#0xFF
\]

° Suppose we want to isolate byte 1 (bit 15 to bit 8) of a word in \( a_0 \). We can use:

\[
\text{and } a_0, a_0, \#0xFF00
\]

but then we still need to shift to the right by 8 bits...

\[
\text{mov } a_0, a_0, \text{lsr} \#8
\]

Isolation with Shift Instructions (#2/2)

° Instead, we can also use:

\[
\begin{align*}
\text{mov } a_0, a_0, \text{lsl} \#16 \\
\text{mov } a_0, a_0, \text{lsr} \#24
\end{align*}
\]

ELEC2041 Reading Materials (Week #3)


° ARM Architecture Reference Manual –On CD ROM

“And in Conclusion…”

° New Instructions:

\[
\begin{align*}
\text{and} \\
\text{bic} \\
\text{orr} \\
\text{Eor}
\end{align*}
\]

° Data Processing Instructions with shift and rotate: