Lecture 08:
Logical Operations
Bitwise Operations

• Up until now, we’ve done
  • arithmetic (add, sub, addi),
  • memory access (lw and sw), and
  • branches and jumps.

• All of these instructions view contents of register as a single value (such as a signed or unsigned integer)

• New Perspective: View register as 32 raw bits rather than as a single 32-bit number

• Since registers are composed of 32 bits, we may want to access individual bits (or groups of bits) rather than the total value.

• Introduce two new classes of instructions:
  • Logical & Shift Ops
Logical Operators (1/3)

• Two basic logical operators:
  • AND: outputs 1 only if both inputs are 1
  • OR: outputs 1 if at least one input is 1

• Truth Table:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>A AND B</th>
<th>A OR B</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Logical Operators (2/3)

• Logical Instruction Syntax:
  1  2,3,4

  • where
    1) operation name
    2) register that will receive value
    3) first operand (register)
    4) second operand (register) or immediate (numerical constant)

• In general, can define them to accept more than 2 inputs, but in the case of MIPS assembly, these accept exactly 2 inputs and produce 1 output
  • Again, rigid syntax, simpler hardware
Logical Operators (3/3)

• Instruction Names:
  • and, or: Both of these expect the third argument to be a register
  • andi, ori: Both of these expect the third argument to be an immediate

• MIPS 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.

• In C: Bitwise AND is & (e.g., \( z = x \ & \ y \);)
• In C: Bitwise OR is | (e.g., \( z = x \ | \ y \);)
Uses for Logical Operators (1/3)

• Notes
  • *and* a bit with 0 produces a 0 at the output
  • *and* 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 |

    mask: 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
The second bitstring in the example is called a **mask**. It is used to isolate the rightmost 12 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, without changing the rest.

If the first bitstring in the above example were in \( \$t0 \), then the following instruction would mask it:

\[ \text{andi } \$t0,\$t0,0xFFF \]
Uses for Logical Operators (3/3)

• Note
  • \texttt{or} a bit with 1 produces a 1 at the output
  • \texttt{or} a bit with 0 produces the original bit.

• This can be used to change certain bits of a string to 1s.
  • For example, if $\text{t0}$ contains $0x12345678$, then after this instruction:
    \begin{center}
    \texttt{ori $\text{t0}, \text{t0}, 0xFFFF$}
    \end{center}
  • ... $\text{t0}$ contains $0x1234FFFF$ (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s).
Shift Instructions (1/3)

• Move (shift) all the bits in a word to the left or right by a number of bits.
  • Example: shift right by 8 bits
    0001 0010 0011 0100 0101 0110 0111 1000
    0000 0000 0001 0010 0011 0100 0101 0110
  • Example: shift left by 8 bits
    0001 0010 0011 0100 0101 0110 0111 1000
    0011 0100 0101 0110 0111 1000 0000 0000
Shift Instructions (2/3)

• Shift Instruction Syntax:
  1  2,3,4
  • where
    1) operation name
    2) register that will receive value
    3) first operand (register)
    4) shift amount (constant < 32)

• MIPS shift instructions:
  1. sll (shift left logical): shifts left and fills emptied bits with 0s
  2. srl (shift right logical): shifts right and fills emptied bits with 0s
  3. sra (shift right arithmetic): shifts right and fills emptied bits by sign extending
Shift Instructions (3/3)

• Example: shift right arith by 8 bits
  0001 0010 0011 0100 0101 0110 0111 1000
  \[\rightarrow\]
  0000 0000 0001 0010 0011 0100 0101 0110

• Example: shift right arith by 8 bits
  1001 0010 0011 0100 0101 0110 0111 1000
  \[\rightarrow\]
  1111 1111 1001 0010 0011 0100 0101 0110
Example: Uses for Shift Instructions (1/4)

- Extracting byte 0 (rightmost 8 bits) of a word in $t0. Simply use:

```assembly
andi $t0, $t0, 0xFF
```

- Isolating byte 1 (bit 15 to bit 8) of a word in $t0. We can use:

```assembly
andi $t0, $t0, 0xFF00
```

but then we still need to shift to the right by 8 bits...
Example: Uses for Shift Instructions (2/4)

- Could use instead:
  
  \[
  \text{\texttt{sll } } \texttt{\$t0,\$t0,16} \\
  \text{\texttt{srl } } \texttt{\$t0,\$t0,24}
  \]

\[
\begin{array}{cccccccccccccccc}
0001 & 0010 & 0011 & 0100 & 0101 & 0110 & 0111 & 1000 \\
0101 & 0110 & 0111 & 1000 & 0000 & 0000 & 0000 & 0000 \\
0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0101 & 0110
\end{array}
\]
Example: Uses for Shift Instructions (3/4)

- In decimal:
  - Multiplying by 10 is same as shifting left by 1:
    - $714_{10} \times 10_{10} = 7140_{10}$
    - $56_{10} \times 10_{10} = 560_{10}$
  - Multiplying by 100 is same as shifting left by 2:
    - $714_{10} \times 100_{10} = 71400_{10}$
    - $56_{10} \times 100_{10} = 5600_{10}$
  - Multiplying by $10^n$ is same as shifting left by $n$
Example: Uses for Shift Instructions (4/4)

- In binary:
  - Multiplying by 2 is same as shifting left by 1:
    - $11_2 \times 10_2 = 110_2$
    - $1010_2 \times 10_2 = 10100_2$
  - Multiplying by 4 is same as shifting left by 2:
    - $11_2 \times 100_2 = 1100_2$
    - $1010_2 \times 100_2 = 101000_2$
  - Multiplying by $2^n$ is same as shifting left by $n$
“And in Conclusion…”

• **Logical and Shift Instructions**
  - Operate on bits individually, unlike arithmetic, which operate on entire word.
  - Use to isolate fields, either by masking or by shifting back and forth.
  - Use **shift left logical**, `sll`, for multiplication by powers of 2
  - Use **shift right arithmetic**, `sra`, for division by powers of 2.

• **New Instructions:**
  `and, andi, or, ori, sll, srl, sra`