CSE220 - Logical Operations

- **New Perspective:** View contents of 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 whole.
- Two new classes of instructions: Logical Operators, Shift Instructions
  - Essential operations for manipulating bits within a word.
  - **Ex:** characters within a word, each which is 8 bits
  - Specific instructions in MIPS to simplify the pack/unpacking of bits into words.
  - AKA bitwise operations

<table>
<thead>
<tr>
<th>Logical ops</th>
<th>C operators</th>
<th>Java operators</th>
<th>MIPS instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift Left</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td>sll</td>
</tr>
<tr>
<td>Shift Right</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;&gt;</td>
<td>srl</td>
</tr>
<tr>
<td>Bit-by-bit AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and, andi</td>
</tr>
<tr>
<td>Bit-by-bit OR</td>
<td></td>
<td></td>
<td>or, ori</td>
</tr>
<tr>
<td>Bit-by-bit NOT</td>
<td>~</td>
<td>~</td>
<td>nor</td>
</tr>
</tbody>
</table>

- Also includes, NOR, XOR, NAND, etc
- Two basic logical operators:
  - **AND:** outputs 1 only if both inputs are 1
  - **OR:** outputs 1 if at least one input is 1
  - In general, can define them to accept >2 inputs, but in the case of MIPS assembly, both of these accept exactly 2 inputs and produce 1 output
    - Again, rigid syntax, simpler hardware
- Bitwise operations, therefore operate on each bit individually
  - **Ex:** $t3 = $t1 AND $t2
    - Each bit of $t1 and $t2 are operated with logical AND and the resulting bits are stored in proper position of $t3
  - OR works similarly. Not inverts bits of a register. It is a pseudo instruction.
- 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 |
    |------------------------|----------------|
    | Mask: 0000 0000 0000    | 1111 1111 1111 |
    | 0000 0000 0000 0000    | 1101 1001 1010 |
  - 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, while leaving the rest alone.
  - In particular, if the first bitstring in the above example were in $t0$, then the following instruction would mask it:
    - andi $t0,$t0,0Xfff
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, if $t0$ contains 0x12345678, then after this instruction:
  - ori $t0$, $t0$, 0xFFFF
- Therefore, $t0$ contains 0x1234FFFF (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s).

Examples of Logical instructions and formats in MIPS
- AND – NOR are logical instructions, remaining are shift instructions
  - Shifts, move bits in register left or right
  - Sign extension is a right arithmetic shift (retains sign)
  - Logical shift is with a zero

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>and $t1,$t2,$t3 # $t1 = $t2 &amp; $t3</td>
</tr>
<tr>
<td>and immediate</td>
<td>and $t1,$t2,$t3 # $t1 = $t2 &amp; $t3</td>
</tr>
<tr>
<td>or</td>
<td>or $t1,$t2,$t3 # $t1 = $t1</td>
</tr>
<tr>
<td>or immediate</td>
<td>or $t1,$t2,$t3 # $t1 = $t2</td>
</tr>
<tr>
<td>xor</td>
<td>xor $t1,$t2,$t3 # $t1 = $t2 ^ $t3</td>
</tr>
<tr>
<td>xor immediate</td>
<td>xor $t1,$t2,$t3 # $t1 = $t2 ^ $t3</td>
</tr>
<tr>
<td>nor</td>
<td>nor $t1,$t2,$t3 # $t1 = ~($t2 &amp; $t3)</td>
</tr>
<tr>
<td>shift left logical</td>
<td>slt $t1,$t2,$t3 # $t1 = $t2 &lt;&lt; 8</td>
</tr>
<tr>
<td>shift left logical by variable</td>
<td>slt $t1,$t2,$t3 # $t1 = $t2 &lt;&lt; $t3</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srl $t1,$t2,$t3 # $t1 = $t2 &gt;&gt; 10</td>
</tr>
<tr>
<td>shift right logical by variable</td>
<td>srl $t1,$t2,$t3 # $t1 = $t2 &gt;&gt; $t3</td>
</tr>
<tr>
<td>shift right arithmetic</td>
<td>sra $t1,$t2,$t3 # $t1 = $t2 &gt;&gt; 6 # sign extend</td>
</tr>
<tr>
<td>shift right arithmetic by variable</td>
<td>sra $t1,$t2,$t3 # $t1 = $t2 &gt;&gt; $t3 # sign extend</td>
</tr>
</tbody>
</table>

**Shift operations**
Move (shift) all the bits in a word to the left or right by a number of bits.
- Example: shift right by 8 bits  
  srl $t1$, $t2$, 8  
  # $t1 <- $t2 >> 8

```
0001 0010 0011 0100 0101 0110 0111 1000
```

```
0000 0000 0001 0010 0011 0100 0101 0110
```

- Shift left, logical, the leftmost bit is lost. The rightmost bit is ‘0’ (called zero fill).
- MIPS shift instructions:
  - sll (shift left logical): shifts left and fills emptied bits with 0s
  - srl (shift right logical): shifts right and fills emptied bits with 0s
- sra (shift right arithmetic): shifts right and fills emptied bits by sign extending

**Example: shift right arith by 8 bits**

```
0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110
```

**Example: shift right arith by 8 bits**

```
1001 0010 0011 0100 0101 0110 0111 1000

1111 1111 1001 0010 0011 0100 0101 0110
```

- Uses for Shift Instructions
  - Suppose we want to isolate byte 0 (rightmost 8 bits) of a word in $t0. Simply use:
    ```
    andi $t0,$t0,0xFF
    ```
  - Suppose we want to isolate byte 1 (bit 15 to bit 8) of a word in $t0. We can use:
    ```
    andi $t0,$t0,0xFF00
    ```
    but then we still need to shift to the right by 8 bits...

**Could use instead:**

```
    sll $t0,$t0,16
    srl $t0,$t0,24
```

- In decimal:
  - Multiplying by 10 is same as shifting left by 1:
    - 714 x 10 = 7140
    - 56 x 10 = 560
  - Multiplying by 100 is same as shifting left by 2:
    - 714 x 100 = 71400
    - 56 x 100 = 5600
  - Multiplying by 10^n is same as shifting left by n

- In binary:
  - Multiplying by 2 is same as shifting left by 1:
    - 112 x 102 = 1102
    - 10102 x 102 = 101002
  - Multiplying by 4 is same as shifting left by 2:
    - 112 x 1002 = 11002
    - 10102 x 1002 = 1010002
  - Multiplying by 2^n is same as shifting left by n

- Since shifting may be faster than multiplication, a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:
  ```
  a *= 8; (in C)
  ```
  would compile to:
• Likewise, shift right to divide by powers of 2
  o remember to use sra

• In addition to word data transfers lw, sw, MIPS has byte data transfers:
  o load byte: lb
  o store byte: sb
    ▪ same format as lw, sw
  o What do with other 24 bits in the 32 bit register?
    ▪ lb: sign extends to fill upper 24 bits

• Normally with characters don't want to sign extend
• MIPS instruction that doesn't sign extend when loading bytes:
  load byte unsigned: lbu

**Rotate or Circular Shift**
• Bits are not lost when we shift
• They wrap around and enter the register from the other end
• These are pseudo-instructions
  o rol: rotate left
  o ror: rotate right
  o Ex: rol $t2, $t2, 4
    ▪ Rotate left bits of $t2 by 4 positions
      1101 0010 0011 0100 0101 0110 0111 1000
      0010 0011 0100 0101 0110 0111 1000 1101