Summary of last lecture

```assembly
addi $s0, $0, 4
addi $s1, $0, 12
add $s2, $s0, $s1
lw $t1, 4($s0)
lw $t2, -4($s1)
sub $t3, $t1, $t2
sw $t3, 4($0)
```
Logical operations

- Instructions for bitwise manipulation

<table>
<thead>
<tr>
<th>Operation</th>
<th>C</th>
<th>Java</th>
<th>MIPS</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>Bitwise AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and, andi</td>
</tr>
<tr>
<td>Bitwise OR</td>
<td></td>
<td></td>
<td>or, ori</td>
</tr>
<tr>
<td>Bitwise NOT</td>
<td>~</td>
<td>~</td>
<td>nor</td>
</tr>
</tbody>
</table>

- Useful for extracting and inserting groups of bits in a word
Shift operations

- **shamt**: how many positions to shift
- **Shift left logical**
  - Shift left and fill with 0 bits
  - \texttt{sll} by \( i \) bits multiplies by \( 2^i \)
- **Shift right logical**
  - Shift right and fill with 0 bits
  - \texttt{srl} by \( i \) bits divides by \( 2^i \) (unsigned only)

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>
AND operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

and $t_0$, $t_1$, $t_2$

<table>
<thead>
<tr>
<th>$t_2$</th>
<th>0000 0000 0000 0000 0000 0000 1101 1100 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>0000 0000 0000 0000 0000 0011 1100 0000 0000</td>
</tr>
<tr>
<td>$t_0$</td>
<td>0000 0000 0000 0000 0000 0000 1100 0000 0000</td>
</tr>
</tbody>
</table>
### OR operations

- Useful to include bits in a word
- Set some bits to 1, leave others unchanged

or $$t0, t1, t2$$

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| **$t2** | 0000 0000 0000 0000 0000 0000 | 1101 | 1100 | 0000 |
| **$t1** | 0000 0000 0000 0000 0000 0000 | 0011 | 1100 | 0000 | 0000 |
| **$t0** | 0000 0000 0000 0000 0000 0000 | 0011 | 1101 | 1100 | 0000 |
NOT operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - \( a \text{ NOR } b = \text{ NOT } ( a \text{ OR } b ) \)

```asm
nor $t0, $t1, $zero
```

<table>
<thead>
<tr>
<th>$t1</th>
<th>0000 0000 0000 0000 0011 1100 0000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t0</td>
<td>1111 1111 1111 1111 1111 1100 0011 1111</td>
</tr>
</tbody>
</table>
Conditional operators

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- beq rs, rt, L1
  - if (rs == rt) branch to instruction labeled L1;
- bne rs, rt, L1
  - if (rs != rt) branch to instruction labeled L1;
- j L1
  - unconditional jump to instruction labeled L1
Compiling if statement

- C code:
  
  ```c
  if (i==j) f = g+h;
  else f = g-h;
  ```
  
  - $f, g, \ldots$ in $s0, s1, \ldots$

- Compiled MIPS code:

  ```
  bne $s3, $s4, Else
  add $s0, $s1, $s2
  j Exit
  Exit:
  ```

  ```
  Else: sub $s0, $s1, $s2
  Exit: ...
  ```
Compiling loop statements

- **C code:**
  
  ```c
  while (save[i] == k) i += 1;
  ```
  
  - i in $s3, k in $s5, address of save in $s6

- **Compiled MIPS code:**
  
  ```
  Loop: sll  $t1, $s3, 2
         add  $t1, $t1, $s6
         lw   $t0, 0($t1)
         bne  $t0, $s5, Exit
         addi $s3, $s3, 1
         j    Loop
  
  Exit: ...
  ```
More conditional operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- \texttt{slt rd, rs, rt}
  - if (rs < rt) rd = 1; else rd = 0;
- \texttt{slti rt, rs, constant}
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with \texttt{beq, bne}
  - \texttt{slt $t0, $s1, $s2}  # if ($s1 < $s2)
  - \texttt{bne $t0, $zero, L}  # branch to L
Branch instruction design

- Why not `blt`, `bge`, etc?
- Hardware for `<`, `≥`, … slower than `=`, `≠`
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- `beq` and `bne` are the common case
- This is a good design compromise
## Summary of instruction formats

### R-Type

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>rs</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>rt</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>rd</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>shamt</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>funct</td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

### I-Type

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>rs</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>rt</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>imm</td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

### J-Type

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>addr</td>
<td></td>
<td>26</td>
</tr>
</tbody>
</table>