Topic 4: More MIPS
Loops, Arrays, and Bit Shifting

- Set Less Than
- More complex branches
- Loops
- Arrays
- Bit shifting
- Multiplication & Division using bit shifting
- Special: The \textit{real} mult/div instructions
Set Less Than

- We've seen `beq/bne`, but how do we branch based on less-than comparisons?

- `slt` ("set less than") compares two registers, stores 0 or 1 in a third
  
  \[
  \text{slt } \$s0, \$s1, \$s2 \quad \# \ s0 = (s1 < s2)
  \]

- `slt` does NOT perform any branch by itself!
Branch if Less Than

- To compare and then branch, use two instructions:

  ```assembly
  slt $s0,$s1,$s2      # s0 = (s1 < s2)
  bne $s0,$zero,THERE  # if (s1 < s2) branch
  ```
Group Exercise:

Compose 2-instruction branches for:

\[ \text{\$s1 > \$s2} \]
\[ \text{\$s1 <= \$s2} \]
\[ \text{\$s1 >= \$s2} \]

Tricks to try:
- Swapping the operands
- Swapping \text{\texttt{beq/bne}}
slt $s0,$s1,$s2      # s0 = (s1 < s2)
bne $s0,$zero,THERE  # if (s1 < s2) branch

slt $s0,$s2,$s1      # s0 = (s1 > s2)
bne $s0,$zero,THERE  # if (s1 > s2) branch

slt $s0,$s2,$s1      # s0 = (s1 > s2)
beq $s0,$zero,THERE  # if (s1 <= s2) branch

slt $s0,$s1,$s2      # s0 = (s1 < s2)
beq $s0,$zero,THERE  # if (s1 >= s2) branch
More Complex Branches

- How to implement arbitrary conditional branches?

```plaintext
if (x == y && a < b)
{
    ...
}
```
More Complex Branches

- Step 1: Think in goto's!
  - Requires you to negate the condition

```c
if (!(x == y && a < b))
    goto AFTER;

... 

AFTER:
```
More Complex Branches

• Step 2: DeMorgan's Laws

  (And you thought that DeMorgan was only for theory)

  if (x != y || a >= b)
    goto AFTER;
  ...
AFTER:
More Complex Branches

• Step 3: Write Multiple Branches
  Details of AND/OR vary

if (x != y)
goto AFTER;

if (a >= b)
goto AFTER;

...

AFTER:
if (x != y)
    goto AFTER;
if (a >= b)
    goto AFTER;
...
AFTER:

Group Exercise:

Convert this C code to assembly. The variables are in the following registers:

- x $s0
- y $s1
- a $s2
- b $s3

Use $tX registers for all temporaries. Do NOT modify any $sX register.
bne $s0,$s1,\textbf{AFTER}  \quad \# \text{ if } (x!=y) \text{ goto }

slt $t0,$s2,$s3 \quad \# \ t0 = (s2 < s3)
beq $t0,$zero,\textbf{AFTER} \quad \# \text{ if } (s2 >= s3) \text{ goto }

\ldots

\textbf{AFTER:}

\begin{quote}
\textbf{The original code:}
\end{quote}

\begin{verbatim}
if (x == y && a < b) {
    \ldots
}
\end{verbatim}
More Complex Branches

- We've seen how to implement AND (two reasons we might skip the next block)

- How to implement OR (two reasons to go **INTO** the block)?
  - Two ways to jump **into** the block
    - Jump over the block if both fail
  - Need a third (unconditional) branch
if (foo >= bar || bar >= baz)
{
    ...
}

Question:

How to write this code using goto's?
if (foo >= bar || bar >= baz)
{
    ...
}

GO_IN:
    ...

AFTER:
if (foo >= bar || bar >= baz)
    goto GO_IN;

go to AFTER;

GO_IN:
...

AFTER:

Group Exercise:

Convert this C code to assembly. The variables are in the following registers:

 foo $s5
bar $s6
baz $s7

Use tX registers for all temporaries.
The original code:
if (foo >= bar || | bar >= baz)
{
    ...

AFTER:

GO_IN:

SLT $t0,$s5,$s6     # s0 = (foo < bar)
BEQ $t0,$zero,GO_IN # if (foo >= bar) goto
SLT $t0,$s6,$s7     # t0 = (bar < baz)
BEQ $t0,$zero,GO_IN # if (baz >= baz) goto
J AFTER
Loops

Loops in assembly:

- Test the condition (often at top)
  - Jump **out** of loop if fail
- Loop back to top when loop done
- Need two labels (top / done)
Normal C Code:

```c
for (int i=0; i<100; i++)
    ...
```

More Like Assembly:

```assembly
int i=0;
    Init

LOOP:
if (i >= 100)
    Test condition;
    Leave if false.
goto AFTER;

    Body
...
i++;
    Increment;
    Back to top

goto LOOP;

AFTER:
    End of Loop
```
for (int i=0; i<100; i++)
  ...

**Group Exercise:**

Convert this C code to assembly.

Assign a $tX$ register of your choice to hold $i$.

Use $tX$ registers for all temporaries.
addi $t0, $zero,0    # i = 0

LOOP:

Or:
slt $t2, $t0, $t1    # t2 = (i < 100)
beq $t2, $zero, AFTER # if (i >= 100) break

addi $t1, $zero, 100    # move this above???
slt $t2, $t0, $t1    # t2 = (i < 100)
beq $t2, $zero, AFTER # if (i >= 100) break

... 

addi $t0, $t0, 1
j LOOP

The original code:
for (int i=0; i<100; i++)
...

AFTER:
while (x < y)
{
    ...

    if (foo == bar)
        break;

    ...
}
LOOP:
slt $t0, $s2, $s3     # t2 = (x < y)
beq $t0, $zero, AFTER # if (x >= y) break
...
beq $s4,$s5, AFTER
...
j LOOP

AFTER:
The original code:
while (x < y)
{
    ...
    ...
    if (foo == bar)
        break;
    ...
}

Loops: Tips and Tricks

- Use same code order as C code
- **Comment heavily**
- Block comments typically **required** for each loop
  - What is the condition?
  - What variables are used?
  - What registers are used, and for what?
- Use blank lines to group related code together
Arrays

- An **array** is several variables of the same type, arranged sequentially in memory.
  
| 0  | 1  | 2  | 3  |
| 4  | 5  | 6  | 7  |
| 8  | 9  | a  | b  |
| C  | d  | e  | f  |

- The address of the array is the address of its first element.
- Element \([i]\) is at address \(\text{base} + i \times \text{size}\).
Declaring Arrays in MIPS

- Declare an array with one label followed by many variables

```assembly
myArray: .word    0    # [0]
          .word    0    # [1]
          .word    0    # [2]
          .word    0    # [3]
          .word    0    # [4]
          .word    0    # [5]
          .word    0    # [6]
```
Array Sizes

- Arrays work like C, not Java
  - Have we heard this before?

- No built-in length operator
- No out-of-bounds checking

- Often we'll either add a `len` variable or use null terminators
Strings as Arrays

- `.asciiz also declares an array – of bytes`
  - Adds automatic null terminator
- `.ascii (no Z) declares a string, doesn't add the null terminator`
int myArray[4];

...  
int i = ... ;

int x = myArray[0];
int y = myArray[1];
int z = myArray[i];

Group Exercise:

Convert this C code to assembly.

Assume that \(i\) is stored in \(s7\).

Assign \(sX\) registers for \(x, y, z\).

Use \(tX\) registers for all temporaries.
The original code:

```c
int myArray[4];
...
int i = ...;  // $s7
int x = myArray[0];
int y = myArray[1];
int z = myArray[i];
```
Bit Shifting

- What instructions can we use for **bit shifting** in assembly?

- Three operations:
  - Shift left, add zeroes on right
  - Shift right, add zeroes on left
  - Shift right, sign extend
Bit Shifting

\[
\text{\texttt{sll}} \quad \text{\texttt{$s0, \$s1, 2$}} \quad \# \text{ shift left logical} \\
\quad \# \quad s0 = (s1 << 2)
\]

\[
\text{\texttt{srl}} \quad \text{\texttt{$s0, \$s1, 2$}} \quad \# \text{ shift right logical}
\]

\[
\text{\texttt{sra}} \quad \text{\texttt{$s0, \$s1, 2$}} \quad \# \text{ shift right arithmetic} \\
\quad \# \quad (\text{sign-extend})
\]
## Questions:

Perform the following bit shift operations on 8-bit numbers:

<table>
<thead>
<tr>
<th>Number</th>
<th>Shift Operation</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100 1101</td>
<td>sll 4 bits</td>
<td></td>
</tr>
<tr>
<td>0100 1000</td>
<td>srl 4 bits</td>
<td></td>
</tr>
<tr>
<td>0100 1010</td>
<td>sra 2 bits</td>
<td></td>
</tr>
<tr>
<td>1100 0000</td>
<td>sra 3 bits</td>
<td></td>
</tr>
</tbody>
</table>
Bit Shifting

Group Exercise:

Convert each decimal number to an 8-bit, 2's complement number:

3  13  27  99
-5  -2  -128  -32  -31
## Bit Shifting

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td>0000 0011</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>0000 1101</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td>0001 1011</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td></td>
<td>0110 0011</td>
<td></td>
</tr>
<tr>
<td>−5</td>
<td></td>
<td>1111 1011</td>
<td></td>
</tr>
<tr>
<td>−2</td>
<td></td>
<td>1111 1110</td>
<td></td>
</tr>
<tr>
<td>−128</td>
<td></td>
<td>1000 0000</td>
<td></td>
</tr>
<tr>
<td>−32</td>
<td></td>
<td>1110 0000</td>
<td></td>
</tr>
<tr>
<td>−31</td>
<td></td>
<td>1110 0001</td>
<td></td>
</tr>
</tbody>
</table>
## Bit Shifting

<table>
<thead>
<tr>
<th>Number</th>
<th>Binary</th>
<th>Operation</th>
<th>Shift Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0000 0011</td>
<td>sll by 3</td>
<td>?</td>
</tr>
<tr>
<td>13</td>
<td>0000 1101</td>
<td>sra by 2</td>
<td>?</td>
</tr>
<tr>
<td>27</td>
<td>0001 1011</td>
<td>sra by 3</td>
<td>?</td>
</tr>
<tr>
<td>99</td>
<td>0110 0011</td>
<td>sra by 5</td>
<td>?</td>
</tr>
<tr>
<td>-5</td>
<td>1111 1011</td>
<td>sra by 2</td>
<td>?</td>
</tr>
<tr>
<td>-2</td>
<td>1111 1110</td>
<td>sll by 6</td>
<td>?</td>
</tr>
<tr>
<td>-128</td>
<td>1000 0000</td>
<td>sra by 2</td>
<td>?</td>
</tr>
<tr>
<td>-32</td>
<td>1110 0000</td>
<td>sra by 5</td>
<td>?</td>
</tr>
<tr>
<td>-31</td>
<td>1110 0001</td>
<td>sra by 5</td>
<td>?</td>
</tr>
</tbody>
</table>
Arithmetic Using Bit Shifting

- You can use bit shifting to perform multiplication and division by powers of 2
  
  \[\text{sll by } n = \text{ multiply by } 2^n\]
  
  \[\text{sra by } n = \text{ divide by } 2^n\]

- \text{sll can overflow}

- \text{sra will round down}
  - Towards \textit{negative infinity}, not zero!
Non-Powers of 2

- To multiply by non-powers of 2, shift, then add

\[
\begin{align*}
&\text{sll } \$t0, \$s0, 4 \quad \# \ t0 = s0 \times 16 \\
&\text{sll } \$t1, \$s0, 1 \quad \# \ t1 = s0 \times 2 \\
&\text{add } \$s1, \$t0,\$t1 \quad \# \ s1 = s0 \times 18 \\
&\text{add } \$s1, \$s1,\$s0 \quad \# \ s1 = s0 \times 19
\end{align*}
\]

- This is (basically) how multiplication works in hardware!
Group Exercise:

Write assembly to perform the following multiplications using shifts, addition, and subtraction:

\[ t_0 = s_0 \times 1024 \]  # best: 1 instruction
\[ t_1 = s_7 \times 7 \]  # best: 2 instructions
\[ t_2 = t_8 \times 36 \]  # best: 3 instructions
\[ t_3 = s_1 \times -8 \]  # best: 2 instructions

Question:
How can we use shift to make indexing into arrays of words more efficient?
mult / div

- MIPS has built-in multiplication and division
  - Much faster than shift/add for non-constant multipliers

- **mult**: 64 bit result
- **div**: two 32-bit results (quotient, remainder)

- Where are the results stored?


**mult / div**

- **mult / div only have two operands**
  - Both ordinary registers

- **Answers always go to hi and lo registers**
  - Special registers (**NOT** part of the normal 32)
  - **Use mfhi / mflo to read them**
mult / div

```assembly
mult $s0, $s1   # hi/lo = s0 * s1
mfhi $s2        # s2 = hi 32 bits
mflo $s3        # s3 = lo 32 bits

div  $s0, $s1
mfhi $s4        # s4 = s0 \% s1
mflo $s5        # s5 = s0 / s1
```
mult / div

• Do not use `mult / div` in your programs for now

• Maybe, we'll add it later – for now, this is just background information.