Load Address (la)

For programmer convenience, the MIPS assembler provides pseudo-instructions. These are instructions that aren’t really implemented by the machine, that the assembler translates into sequences of real instructions. One example is load address. The la pseudo-instruction writes into a register the address associated with a label.

`la $2, array`  # $2 = address associated with the label "array"

la works with labels from both the `.text` and `.data` segments, and in the above example will translate into something like:

`lui $1, upper_address`  # where upper_address are the highest 16 bits in address
`ori $2, $1, lower_address`  # where lower_address are the lowest 16 bits in address

Notice that the assembler introduces $1 as a temporary register to hold intermediate values. We will discuss more about this in future labs.

What values will be in the registers after this code executes?

```
.data  # data segment begins at address 0x10010000
array: .word 1 255 1024

.text
main:  addi $10, $0, 255
      la $2, array  # aka: lui $2, 0x1001
      lw $3, 0($2)
      lw $4, 4($2)
      beq $4, $10, equal
      addi $11, $0, 1
      j end

equal:  addi $12, $0, 2
end:    addi $13, $0, 3
```
What values will be in the registers after this code executes?

```assembly
.data
array: .word 1 255 1024 0xcafebabe

.text
main:  la $2, array       # aka: lui $2, 4097
       lw $3, 12($2)
       slt $8, $3, $0
       slt $9, $2, $0
       slt $10, $0, $2
       slt $11, $2, $3
       lbu $4, 12($2)
       lbu $5, 13($2)
       lbu $6, 14($2)
       lbu $7, 15($2)
       sw $0, 0($2)
       lw $12, 0($2)
       sb $4, 2($2)
       lw $13, 0($2)
```

<table>
<thead>
<tr>
<th>reg</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td></td>
</tr>
<tr>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
</tr>
<tr>
<td>r5</td>
<td></td>
</tr>
<tr>
<td>r6</td>
<td></td>
</tr>
<tr>
<td>r7</td>
<td></td>
</tr>
<tr>
<td>r8</td>
<td></td>
</tr>
<tr>
<td>r9</td>
<td></td>
</tr>
<tr>
<td>r10</td>
<td></td>
</tr>
<tr>
<td>r11</td>
<td></td>
</tr>
<tr>
<td>r12</td>
<td></td>
</tr>
<tr>
<td>r13</td>
<td></td>
</tr>
</tbody>
</table>
Extend the attached datapath figure to support this instruction:

```
jal target  # jump-and-link uses a J-type encoding
```

```
R[31] = PC + 4
PC = (PC & 0xf0000000) | (target << 2)
```

# the 26-bit target specifies bits [27:2] of new PC
Extend the attached datapath figure to support this instruction:

```
addm rd, rs, rt  # load word using contents of register rs
                # add the loaded value to the contents of register rt
                # store the result of the sum in register rd
                # uses R-type encoding
```

\[ R[rd] = \text{Memory}[R[rs] + 0] + R[rt] \]

Keep in mind a few things:

- This operation must complete in a single cycle
- You are allowed to use another ALU to aid in your design
- Notice that the offset for this particular instruction will always be 0
- You may need to declare additional wires for control signals and new muxes for the datapath to be configured correctly for this instruction