Multiplication

- special registers to store result $hi$, $lo$

`mult $8, $9`

- multiply registers 8 and 9
- place result in $hi:lo$: internal registers – not accessible directly

`div $8, $9`

- divide register 8 by 9
- place result (quotient) in $lo$, remainder in $hi`
Multiplication Results

\texttt{mfhi \$d}

- copy contents of \textit{hi} to \$d

\texttt{mflo \$d}

- copy contents of \textit{lo} to \$d
Example

- compute average of three numbers
  - values in $3, $4, $5
  - result in $2
Example: Average

add $2, $3, $4
add $2, $2, $5
addi $1, $0, 3
div $2, $1
mflo $2
jr $31
Logical Instructions

- Logical instructions perform bitwise operations instead of word-level operations.

<table>
<thead>
<tr>
<th>Logical operations</th>
<th>C operators</th>
<th>Java operators</th>
<th>MIPS instructions</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>
Shift Left – sll, Shift Right - srl

- Shift moves all the bits in a word to the left or right, filling the emptied bits with 0s.

Eg. \$16 contains the following value:

\[
\begin{array}{c}
0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 1001_{two} = 9_{ten}
\end{array}
\]

\( \text{sll } \$10, \ $16, \ 4 \quad \# \quad \$10 = \ $16 << 4 \)

\[
\begin{array}{c}
0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 1001 \ 0000_{two} = 144_{ten}
\end{array}
\]

<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>0</td>
<td>0</td>
<td>16</td>
<td>10</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

- Shift left multiplies by 2 for every left-shifted bit, Shift right divides by 2.
**AND, OR**

$2: \quad 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 1101 \ 1100 \ 0000_{\text{two}}$

$3: \quad 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0011 \ 1100 \ 0000 \ 0000_{\text{two}}$

and $4, \ 2, \ 3 \quad \# \quad 4 = 2 \ & \ 3$

$4: \quad 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 1100 \ 0000 \ 0000_{\text{two}}$

- “Masking bits”: Using AND with a bit pattern with specific bits as zeros to make those specific bits in the input bit pattern.

- or $4, \ 2, \ 3 \quad \# \quad 4 = 2 \ | \ 3$

- not?
In keeping with the three-operand format, the designers of MIPS decided to include the instruction NOR (NOT OR) instead of NOT.

If one operand is zero, then it is equivalent to NOT -> A NOR 0 = NOT (A OR 0) = NOT (A).

\[
\text{nor } \$2,\$3,\$4 \quad \# \quad \$2 = \sim(\$3 \mid \$4)
\]

xor is also included in MIPS
andi, ori

- andi $2, $3, 45
- 16 bits of constant are padded by zeros to the left for calculating AND, OR
Memory and Registers

Make sure you recognize what is meant by:

- Register NUMBER (0-31)
- Register CONTENTS ($20) (bits contained in the register)
- Memory ADDRESS
- Memory contents (value)

<table>
<thead>
<tr>
<th>registers</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x00000000</td>
</tr>
<tr>
<td>1</td>
<td>0x00000010</td>
</tr>
<tr>
<td>2</td>
<td>0x00000015</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>31</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Q: Example: Read values stored in memory addresses 64 and 72, add them together and store the result in address 80

```assembly
addi $1, $0, 64
lw $2, 0($1) #Copy value stored at address 64 into $2
lw $3, 8($1) #Copy value stored at address 72 into $3
add $4, $2, $3
sw $4, 16($1) #Store the value in $4 to address 80
jr $31
```

---

<table>
<thead>
<tr>
<th>registers</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x00000000</td>
</tr>
<tr>
<td>1</td>
<td>0x00000010</td>
</tr>
<tr>
<td>2</td>
<td>0x00000015</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>31</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Programs in memory

- When a program is running, it is stored in memory.

- Each word in memory is 32-bits. A MIPS instruction is also 32-bits. Therefore, a MIPS instruction is stored within exactly 1 word of the memory.

- The memory region has $2^{32}$ bytes:
  - Byte Addressing: 0, 1, 2, … $2^{32} - 1$
  - Only, 0, 4, 8, ….. $2^{32} - 4$ are valid
Program Counter

- PC – program counter register - stores the address of the current instruction being executed (byte address)
- Since each instruction is 4 bytes, PC is incremented by 4 for each instruction
- PC is an internal register not accessible by MIPS code
Program Counter

PC points to the first byte of the instruction `la $8, array`. 

```
main:
  la $8, array
  lb $9, ($8)
  add $11, $9, $10
  sb $11, ($8)
  addiu $8, $8, 4
  lh $9, ($8)
  lhu $10, 2($8)
  add $11, $9, $10
  sh $11, ($8)
  addiu $8, $8, 4
  lw $9, ($8)
  lw $10, 4($8)
  sub $11, $9, $10
  sw $11, ($8)
```
Conditional Execution

- essential feature of von Neumann computers!
- What distinguishes a computer from a simple calculator is its ability to make decisions.
- Based on the input data and the values created during computation, different instructions execute.
Conditional Branches

beq $5, $6, Label1
- compare registers $5 and $6
- if equal, go to the instruction labelled label1

bne $5, $6, Label1
- compare registers 's' and 't'
- if not equal, go to the instruction labelled label1
Example – Branching

C code –  if (i==j) f = g+h; else f = g-h;
(f, g, h, i, j correspond to $8-$12 resp. )

MIPS:

beq $11, $12, L1
sub $8, $9, $10
j L2
L1:  add $8, $9, $10
L2:  ...
Example – Loop

Calculate 10!

x=1;
j=10;
while (j !=0){
   x = x*j;
   j= j-1;
}

addi $8, $0, 1
addi $1, $0, 10
L1: mult $8, $1
mflo $8
addi $1, $1, -1
bne $1, $0, L1