COMP 303
Computer Architecture
Lecture 4
Load Upper Immediate

- Example: `lui R8, 255`

  - Transfers the immediate field into the register’s top 16 bits and fills the register’s lower 16 bits with zeros

  \[
  \begin{array}{c|c|c|c|c|c|c}
  31 & 26 & 25 & 21 & 20 & 16 & 15 \\
  \hline
  001111 & 00000 & 01000 & \text{ immediate} \\
  \end{array}
  \]

  \[
  \begin{array}{cccccccc}
  31 & 26 & 25 & 21 & 20 & 16 & 15 & 0 \\
  \hline
  0000 0000 1111 1111 & 0000 0000 0000 0000 \\
  \end{array}
  \]

  \[
  \begin{array}{cccccccc}
  R8 & 31 & 26 & 25 & 21 & 20 & 16 & 15 & 0 \\
  \hline
  0000 0000 1111 1111 & 0000 0000 0000 0000 \\
  \end{array}
  \]
Large constants

- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

\[
\text{lui } \$t0, \ 1010101010101010 \\
\text{ori } \$t0, \$t0, \ 1111000011001010
\]

Then must get the lower order bits right, i.e.,

\[
\text{ori } \$t0, \$t0, \ 1111000011001010
\]

Filled with zero

\[
\begin{array}{c}
1010 & 1010 & 1010 & 1010 \\
\hline
0000 & 0000 & 0000 & 0000
\end{array}
\]

\[
\begin{array}{c}
0000 & 0000 & 0000 & 0000 \\
\hline
1111 & 0000 & 1100 & 1010
\end{array}
\]

\[
\begin{array}{c}
1010 & 1010 & 1010 & 1010 \\
\hline
1111 & 0000 & 1100 & 1010
\end{array}
\]
## MIPS data transfer instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw $3, 500($4)</td>
<td>Store word</td>
</tr>
<tr>
<td>sh $3, 502($2)</td>
<td>Store half</td>
</tr>
<tr>
<td>sb $2, 41($3)</td>
<td>Store byte</td>
</tr>
<tr>
<td>lw $1, 30($2)</td>
<td>Load word</td>
</tr>
<tr>
<td>lh $1, 40($3)</td>
<td>Load halfword</td>
</tr>
<tr>
<td>lhu $1, 40($3)</td>
<td>Load halfword unsigned</td>
</tr>
<tr>
<td>lb $1, 40($3)</td>
<td>Load byte</td>
</tr>
<tr>
<td>lbu $1, 40($3)</td>
<td>Load byte unsigned</td>
</tr>
<tr>
<td>lui $1, 40</td>
<td>Load Upper Immediate</td>
</tr>
<tr>
<td></td>
<td>(16 bits shifted left by 16)</td>
</tr>
</tbody>
</table>
Store byte (sb) instruction

- Example: `sb $3, 12($1)`
Load byte (lb) instruction

- Example: lb $3, 12($1)
Translating and starting a program

C Program

Compiler

Assembly language program

Assembler

Obj: Machine lang. module

Obj: Library Routine (Mach. lang.)

Linker

Exec: Machine lang. prog

Loader

Memory
Assembly language

- **Assembly language** is the symbolic representation of a computer’s binary encoding, which is called **machine language**.
- Assembly language is more readable than machine language because it uses symbols instead of bits.
- Assembly language permits programmers to use **labels** to identify and name particular memory words that hold instructions or data.
- A tool called **assembler** translates assembly language into binary instructions.
- An assembler reads a single assembly language **source file** and produces **object file** containing machine instructions and bookkeeping information that helps combine several object files into a program.
Advantages & disadvantages

- Assembly programming is useful when the speed or size of a program is important.
- But assembly languages are machine specific and they must be rewritten to run on another machine.
- Another disadvantage is that assembly language programs are longer than the equivalent programs written in a high-level languages.
- It is also true that programs written in assembly are more difficult read and understand and they may contain more bugs.
MIPS memory allocation for program & data

- **Stack**
  - Dynamic data
  - Static data
  - Text
  - Reserved

- Stack pointer (sp): $7FF$ EFFC<sub>hex</sub>
- Global pointer (gp): $1000$ 8000<sub>hex</sub>
- Program counter (pc): $0040$ 0000<sub>hex</sub>
A translation hierarchy for Java

- Java Program
- Compiler
  - Class files (Java bytecodes)
  - Just In Time (JIT) compiler
  - Compiled Java methods (mach. lang.)
- Java Library routines (Mach. lang.)
- Java Virtual Machine (JVM)
Array vs pointer

void clear1(int array[], int size)
{
    int i;
    for (i = 0; i < size; i++);
        array[i] = 0;
}

void clear2(int *array, int size)
{
    int *p;
    for (p = &array[0]; p < &array[size]; p++);
        *p = 0;
}
Array version of “clear”

- Assume that the two parameters array and size are found in the registers $a0 and $a1, and that i is allocated to register $t0.

```assembly
move $t0,$zero  # i = 0
loop1: add $t1,$t0,$t0  # $t1 = i * 2
add $t1,$t1,$t1  # $t1 = i * 4
add $t2,$a0,$t1  # $t2 = address of array[i]
sw $zero,0($t2)  # array[i] = 0
addi $t0,$t0,1  # i = i + 1
slt $t3,$t0,$a1  # $t3 = (i < size)
bne $t3,$zero,loop1  # if (i < size) go to loop1
```
Pointer version of “clear”

- Assume that the two parameters array and size are found in the registers $a0 and $a1, and that p is allocated to register $t0.

```
move $t0, $a0  # p = address of array[0]
add $t1, $a1, $a1  # $t1 = size * 2
add $t1, $t1, $t1  # $t1 = size * 4
add $t2, $a0, $t1  # $t2 = address of array[size]

loop2:
sw $zero, 0($t0)  # Memory[p] = 0
addi $t0, $t0, 4  # p = p + 4
slt $t3, $t0, $t2  # $t3 = (p < &array[size])
bne $t3, $zero, loop2  # if (p < &array[size]) go to loop2
```
Comparing two versions of “clear”

<table>
<thead>
<tr>
<th>move $t0,$zero # i = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop1: add $t1,$t0,$t0 # $t1 = i * 2</td>
</tr>
<tr>
<td>add $t1,$t1,$t1 # $t1 = i * 4</td>
</tr>
<tr>
<td>add $t2,$a0, $t1 # $t2 = address of array[i]</td>
</tr>
<tr>
<td>sw $zero,0($t2) # array[i] = 0</td>
</tr>
<tr>
<td>addi $t0,$t0,1 # i = i + 1</td>
</tr>
<tr>
<td>slt $t3,$t0,$a1 # $t3 = (i &lt; size)</td>
</tr>
<tr>
<td>bne $t3,$zero,$loopp1 # if (i &lt; size) go to loop1</td>
</tr>
</tbody>
</table>

| move $t0,$a0 # p = address of array[0] |
| add $t1,$a1,$a1 # $t1 = size * 2 |
| add $t1,$t1,$t1 # $t1 = size * 4 |
| add $t2,$a0, $t1 # $t2 = address of array[size] |
| loop2: sw $zero,0($t0) # Memory[p] = 0 |
| addi $t0,$t0,4 # p = p + 4 |
| slt $t3,$t0,$t2 # $t3 = (p < &array[size]) |
| bne $t3,$zero,$loopp2 # if (p < &array[size]) go # to loop2 |

- The pointer version reduces the instructions executed per iteration from 7 to 4.
Reading assignment

- Read 2.6, 2.8, 2.10 (Linker), 2.13