Thought question before class

(answer in clickers)

Why are we picking register transfer language

\[
\begin{align*}
  r_2 &\leftarrow r_1 + r_0 \\
  r_0 &\leftarrow M[\text{address}] 
\end{align*}
\]

as the basis of our processor’s programming language?
CS 2630
Computer Organization

Meeting 5: MIPS
Brandon Myers
University of Iowa
Where we are going

Instruction set architecture (e.g., MIPS)

Compiler
  translating source code (C or Java)
  Programs to assembly language
  And linking your code to Library code

Instruction set architecture (e.g., MIPS)

Memory system

Processor

I/O system

Datapath & Control

Digital logic

How the software talks To the hardware

How a processor runs MIPS Programs!

How switches (1 or 0) can be used to build Interesting functions:
  from integer arithmetic to programmable computers
Register transfer language

- 4 categories of **instructions**
  - perform an operation on two registers and store result in a register
  - perform an operation on one register and a constant and store the result in a register
  - move a value between a register and memory
  - determine which instruction to execute next

(later!)
Introducing MIPS

```
lw $t0, 4($0)  # r0 <- M[0x4]
lw $t1, 8($0)  # r1 <- M[0x8]
add $t2, $t0, $t1  # r2 <- r0 + r1
lw $t0, 12($0)  # r0 <- M[0xC]
sub $t2, $t2, $t0  # r2 <- r2 - r0
sw $t2, 0($0)  # M[0x0] <- r2
```
MIPS registers (32 of them)

<table>
<thead>
<tr>
<th>Register Number</th>
<th>Conventional Name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>$zero</td>
<td>hard-wired to 0</td>
</tr>
<tr>
<td>$1</td>
<td>$at</td>
<td>reserved for assembler</td>
</tr>
<tr>
<td>$2 - $3</td>
<td>$v0, $v1</td>
<td>return values from functions</td>
</tr>
<tr>
<td>$4 - $7</td>
<td>$a0 - $a3</td>
<td>arguments</td>
</tr>
<tr>
<td>$8 - $15</td>
<td>$t0 - $t7</td>
<td>temporary registers</td>
</tr>
<tr>
<td>$16 - $23</td>
<td>$s0 - $s7</td>
<td>saved registers</td>
</tr>
<tr>
<td>$24 - $25</td>
<td>$t8 - $t9</td>
<td>temporary registers</td>
</tr>
<tr>
<td>$26 - $27</td>
<td>$k0 - $k1</td>
<td>reserved for OS</td>
</tr>
<tr>
<td>$28</td>
<td>$gp</td>
<td>global pointer</td>
</tr>
<tr>
<td>$29</td>
<td>$sp</td>
<td>stack Pointer</td>
</tr>
<tr>
<td>$30</td>
<td>$fp</td>
<td>frame Pointer</td>
</tr>
<tr>
<td>$31</td>
<td>$ra</td>
<td>return Address</td>
</tr>
</tbody>
</table>
### MIPS instruction format: R-type

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Behavior</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add $rd, $rs, $rt</code></td>
<td><code>R[$rd] ← R[$rs] + R[$rt]</code></td>
<td><code>add $t2, $t0, $t1</code></td>
</tr>
<tr>
<td><code>sub $rd, $rs, $rt</code></td>
<td><code>R[$rd] ← R[$rs] - R[$rt]</code></td>
<td><code>sub $t6, $t7, $t2</code></td>
</tr>
<tr>
<td><code>or $rd, $rs, $rt</code></td>
<td>`R[$rd] ← R[$rs]</td>
<td>R[$rt]`</td>
</tr>
</tbody>
</table>
MIPS instruction format: **I-type**

<table>
<thead>
<tr>
<th>syntax</th>
<th>behavior</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lw $rt, imm($rs)</code></td>
<td>$R[$rt] \leftarrow \text{Mem}<em>{4B}(R[$rs] + \text{SignExt}</em>{16b}(imm))$</td>
<td><code>lw \$t0, 4(\$0)</code></td>
</tr>
<tr>
<td>“load word”</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>addi $rt, $rs, imm</code></td>
<td>$R[$rt] \leftarrow R[$rs] + \text{SignExt}_{16b}(imm)$</td>
<td><code>addi \$t1, \$t5, 100</code></td>
</tr>
<tr>
<td>“add immediate”</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sw $rt, imm($rs)</code></td>
<td>$	ext{Mem}<em>{4B}(R[$rs] + \text{SignExt}</em>{16b}(imm)) \leftarrow R[$rt]$</td>
<td><code>sw \$t2, 0(\$0)</code></td>
</tr>
<tr>
<td>“store word”</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

I is for *immediate*, which in MIPS means a constant
load and store

\textbf{lw} \ $t0, \ 4($0) \\
\textbf{sw} \ $t2, \ 0($0)

\begin{align*}
\text{address} & = \text{base} + \text{offset} \\
 & = \text{src} + \text{Imm} \\
 & = 0 + 4 = 4 \\
\text{address} & = \text{base} + \text{offset} \\
 & = \text{src} + \text{Imm} \\
 & = 0 + 0 = 0
\end{align*}

\textcolor{red}{\text{register} \$0 \ (\text{or} \ \$\text{zero}) \ \text{is \ hard-wired \ to} \ 0}
Logical operations

• R-type

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>sll $rd, $rt, shamt</td>
<td>$rd \leftarrow R[$rt] \ll \text{shamt}</td>
</tr>
<tr>
<td>srl $rd, $rt, shamt</td>
<td>$rd \leftarrow R[$rt] \gg \text{shamt}</td>
</tr>
<tr>
<td>sra $rd, $rt, shamt</td>
<td>$rd \leftarrow R[$rt] \gg \text{shamt}</td>
</tr>
</tbody>
</table>

• I-type

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>andi $rt, $rs, imm</td>
<td>$rt \leftarrow R[$rs] &amp; {0 \times 16, \text{imm}}</td>
</tr>
<tr>
<td>ori $rt, $rs, imm</td>
<td>$rt \leftarrow R[$rs] \mid {0 \times 16, \text{imm}}</td>
</tr>
</tbody>
</table>

This notation just means extend the 16-bit immediate with 0’s in front so that it is 32 bits.

Yes, sll and srl are R-type; we’ll soon see why (hint: what is the largest shift value needed?)
Translate the following code into MIPS.

// k is an int[]. Assume the first byte of k is already
// stored at address 0x10
k[0] = k[1] + 1;
What instructions are we missing?
Making decisions

if (z == 0) then a = x + y
else a = z

bne "branch (on) not equal"
j "(unconditional) jump"

bne $t0,$zero,there_plz
add $t1,$t2,$t3
j finish_plz

there_plz: or $t1,$t0,$zero
finish_plz: sw $t1,0($t7)
A Case against the GO TO Statement.

by Edsger W. Dijkstra

Technological University
Eindhoven, The Netherlands

Since a number of years I am familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. Later I discovered why the use of the go to statement has such disastrous effects and did I become convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except -perhaps- plain machine code),

article AKA “Go-to statement considered harmful”
Branch/jump instructions

\texttt{bne \$t0,\$zero, there\_plz}

\begin{itemize}
\item \texttt{src}
\item \texttt{src}
\item \texttt{label}
\end{itemize}

\texttt{beq \$t1,\$t3, launch}

\begin{itemize}
\item \texttt{src}
\item \texttt{src}
\item \texttt{label}
\end{itemize}

\texttt{j over\_the\_moon}

\begin{itemize}
\item \texttt{label}
\end{itemize}

\texttt{jr \$t5}

\begin{itemize}
\item \texttt{src}
\end{itemize}

\begin{itemize}
\item “branch on not equal”
\item “branch on equal”
\item “jump”
\item “jump register”
\end{itemize}
Peer instruction

What value is stored in $t0 when the program finishes?

```
ori   $t1,$zero,3
addi  $t2,$zero,1
yum: addiu $t1,$t1,-1
sll   $t2,$t2,1
bne   $t1,$zero,yum
nop    # no-op, this instruction does nothing
addiu $t2,$t2,1
end: or    $t0,$zero,$t2
```

(numeric response)