Thought question before class

(answer in clickers)

Why are we picking register transfer language

\[ r2 \leftarrow r1 + r0 \]
e.g.,
\[ r0 \leftarrow M[\text{address}] \]

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

How the software talks to the hardware

Memory system

Processor

Datapath & Control

I/O system

How a processor runs MIPS Programs!

Digital logic

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: l-type**

<table>
<thead>
<tr>
<th>syntax</th>
<th>behavior</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $rt, imm($rs)</td>
<td>$R[Rrt] \leftarrow \text{Mem}<em>{4B}(R[rs] + \text{SignExt}</em>{16B}(imm))</td>
<td>lw $t0, 4($0)</td>
</tr>
</tbody>
</table>

“load word”

| addi $rt, $rs, imm | $R[Rrt] \leftarrow R[rs] + \text{SignExt}_{16B}(imm) | addi $t1, $t5, 100 |

“add immediate”

| sw $rt, imm($rs) | \text{Mem}_{4B}(R[rs] + \text{SignExt}_{16B}(imm)) \leftrightarrow R[Rrt] | sw $t2, 0($0)      |

“store word”

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

\[
\text{lw } \$t0, \ 4(\$0)\\
dst \hspace{1cm} \text{Imm} \hspace{1cm} \text{src}
\]

\[
\text{sw } \$t2, \ 0(\$0)\\
dst \hspace{1cm} \text{Imm} \hspace{1cm} \text{src}
\]

\[
\text{address} = \text{base} + \text{offset}\\
= \text{src} + \text{Imm}\\
= 0 + 4 = 4
\]

\[
\text{address} = \text{base} + \text{offset}\\
= \text{src} + \text{Imm}\\
= 0 + 0 = 0
\]

register $0$ (or $\$zero$) is hard-wired to 0
Logical operations

• R-type

<table>
<thead>
<tr>
<th>Format</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>slt $rd, $rt, shamt</td>
<td>$R[rd] \leftarrow R[rt] \ll shamt \quad \text{Signed left shift}</td>
<td></td>
</tr>
<tr>
<td>srl $rd, $rt, shamt</td>
<td>$R[rd] \leftarrow R[rt] \ll\ll shamt \quad \text{Unsigned right shift}</td>
<td></td>
</tr>
<tr>
<td>sra $rd, $rt, shamt</td>
<td>$R[rd] \leftarrow R[rt] \ll\ll shamt \quad \text{Signed right shift}</td>
<td></td>
</tr>
</tbody>
</table>

Rauważ, że ten notację można traktować jako rozszerzenie 16-bitowej wartości o 0's na przód, w celu uzyskania 32-bitowej wartości.

• I-type

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

Takie, sll i srl to typ R; wkrótce to zobaczymy (wskazówka: co to jest największą wartość skifatu potrzebną?)
Peer instruction

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;

use the MIPS reference sheet handed out in class or
linked from the resources tab of the website
What instructions are we missing?
Making decisions

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

\textit{bne} “branch (on) not equal”
\textit{j} “(unconditional) jump”

\texttt{bne \$t0,\$zero,there\_plz}
\texttt{add \$t1,\$t2,\$t3}
\texttt{j finish\_plz}
\texttt{there\_plz: or \$t1,\$t0,\$zero}
\texttt{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

`bne $t0,$zero,there_plz`
src src label

“branch on not equal”

`beq $t1,$t3,launch`
src src label

“branch on equal”

`j over_the_moon`
label

“jump”

`jr $t5`
src

“jump register”
Peer instruction

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

```
ori   $t1,$zero,3
addi  $t2,$zero,1
yalum: 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)
• https://www.youtube.com/watch?v=P47VBYF9Woo&feature=youtu.be&t=34m12s
Stored program concept

Instruction memory  Execution engine  Data memory

program represented as bits and stored here
What to do now

• Quiz 2 and HW2

• do Part 0 immediately if you haven’t yet
  • install MARS (our IDE for MIPS programming)
  • watch two 5-minute videos on how to use MARS

• trying to get into the class and being stubborn about picking 2630 over 3350?
  • bring me a Change of Registration form but I have to hold onto it until enrollment drops below 60