CS 2630
Computer Organization
Meeting 3: bits, and MIPS intro
Brandon Myers
University of Iowa
Where we are going

- Compiler: translating source code (C or Java) Programs to assembly language And linking your code to Library code
- Instruction set architecture (e.g., MIPS)
  - How the software talks To the hardware
  - Processor: How a processor runs MIPS Programs!
  - Memory system
  - Datapath & Control
  - I/O system
  - Digital logic: How switches (1 or 0) can be used to build Interesting functions: from integer arithmetic to programmable computers
Where we are going

Instruction memory  Execution engine  Data memory

Learn what gets stored in here
memory

0000
0004
0008
000C
0010
0014
0018
001C
0020
0024
0028

registers

...
The main idea

- 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
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!)
Example assembly program

**MEMORY**

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0</td>
</tr>
<tr>
<td>0x4</td>
<td>-14</td>
</tr>
<tr>
<td>0x8</td>
<td>50</td>
</tr>
<tr>
<td>0xC</td>
<td>1</td>
</tr>
</tbody>
</table>

**REGISTERS**

- r0
- r1
- r2

**high level**

\[ a = x + y - z \]

Peer instruction:
Write an assembly program using a sequence of instructions (use register transfer language)
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**

\[
\text{add} \quad \text{dst} \quad \text{src} \quad \text{src} \\
\text{sub} \quad \text{dst} \quad \text{src} \quad \text{src} \\
\text{or} \quad \text{dst} \quad \text{src} \quad \text{src}
\]
MIPS instruction format: I-type

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

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

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

address = base + offset
= src + Imm
= 0 + 4 = 4

remember, register $0$ (or $\$zero$) is hard-wired to 0

address = base + offset
= src + Imm
= 0 + 0 = 0
Logical operations

• R-type
  • sll $t0, $t1, 2     # t0 <- t1 << 12  (shift left logical)
  • srl $t0, $t1, 3    # t0 <- t1 >> 4   (shift right logical)
  • sra $t0, $t1, 3    # t0 <- t1 >>> 4  (shift right arithmetic)
  • and $t0, $t1, $t2  # t00 <- t1 & t2  (AND)
  • or $t0, $t1, 2     # t00 <- t1 | t2   (OR)

• I-type
  • andi $t0, $t1, 22  # t0 <- t1 & 22
  • ori  $t0, $t1, 0xFF # t0 <- t1 | 0xFF

yes, sll and srl are R-type; we’ll soon see why (hint: what is the largest shift value needed?)
Peer instruction

Translate the following code into MIPS.

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

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

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

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

finish_plz:
or $t1, $t0, $zero
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}  \hspace{1cm} \text{“branch on not equal”}
\hspace{1cm} \text{src} \hspace{0.5cm} \text{src} \hspace{0.5cm} \text{label}

\texttt{beq \ $t1,\$t3,\ launch}  \hspace{1cm} \text{“branch on equal”}
\hspace{1cm} \text{src} \hspace{0.5cm} \text{src} \hspace{0.5cm} \text{label}

\texttt{j \ over\_the\_moon}  \hspace{1cm} \text{“jump”}
\hspace{1cm} \text{label}

\texttt{jr \$t5}  \hspace{1cm} \text{“jump register”}
\hspace{1cm} \text{src}
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)
Stored program concept

program represented as bits and stored here
What to do now

• HW1 due tonight 11:59pm