C335
Computer Structures

Introduction to MIPS Assembly Languages

Dr. David R. Surma
Department of Computer and Information Sciences
Adapted from Morgan Kaufmann, Dr. Zhang, and others
Review: Instruction Set Architecture

- Basic job of a CPU: execute lots of instructions.
- Instructions are the primitive operations that the CPU may execute.
- Different CPUs implement different sets of instructions. The set of instructions a particular CPU implements is an Instruction Set Architecture (ISA).
  - Examples: Intel x86, Apple/IBM/Motorola PowerPC (Macintosh), ARM, MIPS, Intel IA64, x86-64...
MIPS Architecture

- MIPS (Microprocessor without Interlocked Pipeline Stages) – semiconductor company that built one of the first commercial RISC architectures (RISC = reduced instruction set computer)

- We will study the MIPS architecture in some detail in this class

- Why MIPS instead of Intel 80x86?
  - MIPS is simple, elegant. Don’t want to get bogged down in gritty details.
  - MIPS widely used in embedded apps, x86 little used in embedded, and more embedded computers than PCs
MIPS Architecture

- Historically, many video game consoles used the MIPS architecture.
  - Sony Playstation (PS2)
  - Playstation portable
  - Nintendo 64

- As of April 2017 (according to Wikipedia), MIPS processors are used in embedded systems such as residential gateways and routers.
Assembly Variables: Registers (1/4)

- Unlike High-level languages like C or Java, assembly language cannot use variables.
  - Why not?
    - Keep Hardware Simple

- Assembly Operands are registers
  - limited number of special locations built directly into the CPU hardware
  - Arithmetic/logical operations can only be performed on these!

- Benefit: Since registers are directly in CPU hardware, they are very fast (faster than 1 billionth of a second)
Assembly Variables: Registers (2/4)

- Drawback: Since registers are in CPU, there are a predetermined number of them.
  - Solution: MIPS code must be very carefully put together to efficiently use registers

- 32 registers in MIPS, Why only 32?
  - Smaller is faster

- Each MIPS register is 32 bits wide
  - Groups of 32 bits called a word in MIPS
Assembly Variables: Registers (3/4)

- Registers are numbered from 0 to 31
- Each register can be referred to by number or name
- Number references: $0, $1, $2, ... $30, $31

MIPS General-Purpose Registers

- 32 General Purpose Registers (GPRs)
  - All registers are 32-bit wide in the MIPS 32-bit architecture
  - Software defines names for registers to standardize their use
  - Assembler can refer to registers by name or by number ($ notation)

<table>
<thead>
<tr>
<th>Name</th>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>$0</td>
<td>Always 0</td>
</tr>
<tr>
<td>$at</td>
<td>$1</td>
<td>Reserved for assembler use</td>
</tr>
<tr>
<td>$v0 – $v1</td>
<td>$2 – $3</td>
<td>Result values of a function</td>
</tr>
<tr>
<td>$a0 – $a3</td>
<td>$4 – $7</td>
<td>Arguments of a function</td>
</tr>
<tr>
<td>$t0 – $t7</td>
<td>$8 – $15</td>
<td>Temporary Values</td>
</tr>
<tr>
<td>$s0 – $s7</td>
<td>$16 – $23</td>
<td>Saved registers (preserved across call)</td>
</tr>
<tr>
<td>$t8 – $t9</td>
<td>$24 – $25</td>
<td>More temporaries</td>
</tr>
<tr>
<td>$k0 – $k1</td>
<td>$26 – $27</td>
<td>Reserved for OS kernel</td>
</tr>
<tr>
<td>$gp</td>
<td>$28</td>
<td>Global pointer (points to global data)</td>
</tr>
<tr>
<td>$sp</td>
<td>$29</td>
<td>Stack pointer (points to top of stack)</td>
</tr>
<tr>
<td>$fp</td>
<td>$30</td>
<td>Frame pointer (points to stack frame)</td>
</tr>
<tr>
<td>$ra</td>
<td>$31</td>
<td>Return address (used for function call)</td>
</tr>
</tbody>
</table>
By convention, each register also has a name to make it easier to code

For now:

$16 - $23 \rightarrow $s0 - $s7

(correspond to C variables)

$8 - $15 \rightarrow $t0 - $t7

(correspond to temporary variables)

Later will explain other 16 register names

In general, use names to make your code *more readable!*
C variables vs. registers

- In C (and most High Level Languages) variables are declared first and given a type
  - Example:
    ```c
    int fahr, celsius;
    char a, b, c, d, e;
    ```
- Each variable can ONLY represent a value of the type it was declared as.
- In Assembly Language, the registers have no type; the operation determines how register contents are treated
Comments in Assembly

- Another way to make your code more readable: **comments**!
- Hash (＃) is used for MIPS comments
  - anything from hash mark to end of line is a comment and will be ignored
- Note: Different from C.
  - C comments have format
    ```
    /* comment */
    ```
    so they can span many lines

**Examples:**

```assembly
la $a0, str2
    #load system register $a0, with the address of the string to be output
sub $12, $4, $5  #subtract contents of R5 from R4 and store in R12
```
Assembly Instructions

- In assembly language, each statement (called an Instruction), executes exactly one of a short list of simple commands.
- Unlike in C (and most other High Level Languages), each line of assembly code contains at most 1 instruction.
- Instructions are related to operations (=, +, -, *, /) in C or Java.
- Ok, enough already… let’s learn some MIPS!
MIPS Addition and Subtraction (1/4)

- Syntax of Instructions:
  1 2,3,4
  where:
  1) operation - by name
  2) operand getting result ("destination")
  3) 1st operand for operation ("source1")
  4) 2nd operand for operation ("source2")

- Syntax is rigid (Why?)
  - 1 operator, 3 operands
  - *Keep Hardware simple via regularity*
Addition and Subtraction of Integers (2/4)

- **Addition in Assembly**
  - Example: `add $s0,$s1,$s2` (in MIPS)
  - Equivalent to: `a = b + c` (in C)
  - where MIPS registers `$s0, $s1, $s2` are associated with C variables `a, b, c`

- **Subtraction in Assembly**
  - Example: `sub $s3,$s4,$s5` (in MIPS)
  - Equivalent to: `d = e - f` (in C)
  - where MIPS registers `$s3, $s4, $s5` are associated with C variables `d, e, f`
How do we do for the following C statement?

\[ a = b + c + d - e; \]

Break into multiple instructions

- `add $t0, $s1, $s2`  # temp = b + c
- `add $t0, $t0, $s3`  # temp = temp + d
- `sub $s0, $t0, $s4`  # a = temp - e

Notice: A single line of C may “break up” into several lines of MIPS.

Notice: Everything after the hash mark on each line is ignored (comments)
How do we do this?

\[ f = (g + h) - (i + j); \]

Use intermediate *temporary register*!

\[
\begin{align*}
\text{add} & \quad \$t0,\$s1,\$s2 \quad \# \text{temp} = g + h \\
\text{add} & \quad \$t1,\$s3,\$s4 \quad \# \text{temp} = i + j \\
\text{sub} & \quad \$s0,\$t0,\$t1 \quad \# f=(g+h)-(i+j)
\end{align*}
\]
Register Zero

- One particular immediate, the number zero (0), appears very often in code.

- So we define register zero ($0 or $zero) to always have the value 0; eg
  
  add $s0,$s1,$zero (in MIPS)
  
  \[ f = g \] (in C)

  where MIPS registers $s0, $s1 are associated with C variables f, g

- defined in hardware, so an instruction

  add $zero,$zero,$s0

  will not “do” anything! {Certainly something is actually done!}
Immediates

- “Immediates” are numerical constants.
- They appear often in code, so there are special instructions for them.
- Add Immediate:

  \[
  \text{addi } \$s0,\$s1,10 \quad \text{(in MIPS)} \\
  f = g + 10 \quad \text{(in C)}
  \]

  where MIPS registers \( \$s0, \$s1 \) are associated with C variables \( f, g \).

- Syntax similar to add instruction, except that last argument is a number instead of a register.
There is no Subtract Immediate in MIPS: Why?

Desire to *limit types of operations* that can be done to as few as possible (absolute minimum)

- if an operation can be decomposed into a simpler operation, don’t include it in the set of instructions
- \texttt{addi \ldots, -X = subi \ldots, X} $\Rightarrow$ so no \texttt{subi}

\texttt{addi} $\$s0,\$s1,-10$ (in MIPS)

\[ f = g - 10 \] (in C)

where MIPS registers $\$s0,\$s1$ are associated with C variables $f, g$
Question

A. Types are associated with declaration in C (normally), but are associated with instruction (operator) in MIPS.

B. Since there are only 8 local ($s$) and 10 temp ($t$) variables, we can’t write MIPS for C expressions that contain > 18 variables.

C. If $p$ (stored in $s0$) were a pointer to an array of ints, then $p++$ would be `addi $s0, $s0, 1`
“And in Conclusion…”

- In MIPS Assembly Language:
  - Registers replace C variables
  - One Instruction (simple operation) per line
  - Simpler is Better
  - Smaller is Faster
- New Instructions:
  - `add`, `addi`, `sub`
- New Registers:
  - C Variables: `$s0 - $s7`
  - Temporary Variables: `$t0 - $t9`
  - Zero: `$zero`