New-School Machine Structures
(It’s a bit more complicated!)

- Parallel Requests
  - Assigned to computer
  - e.g., Search “Katz"
- Parallel Threads
  - Assigned to core
  - e.g., Lookup, Ads
- Parallel Instructions
  - >1 instruction @ one time
  - e.g., 5 pipelined instructions
- Parallel Data
  - >1 data item @ one time
  - e.g., Add of 4 pairs of words
- Hardware descriptions
  - All gates @ one time
- Programming Languages

Agenda
- Machine Language
- Administrivia
- Operands
- Technology Break
- Decisions
- Summary

The Language a Computer Understands
- Word a computer understands: instruction
- Vocabulary of all words a computer understands: instruction set (aka instruction set architecture or ISA)
- Different computers may have different vocabularies (i.e., different ISAs)
  - iPhone not same as Macbook
- Or the same vocabulary (i.e., same ISA)
  - iPhone and iPad computers have same instruction set

Agenda
- Machine Language
- Administrivia
- Operands
- Technology Break
- Decisions
- Summary

Levels of Representation/Interpretation
- High-Level Language (e.g., C)
- Compiler
- Assembler (e.g., MIPS)
- Machine Language (e.g., MIPS)
- Architecture
- Hardware Architecture Description (e.g., block diagrams)
- Logic Circuit Description (Circuit Schematic Diagrams)
The Language a Computer Understands

- Why not all the same? Why not all different?
  - Why not all the same? Why not all different?
  - What might be pros and cons?

- Single ISA (to rule them all):
  - Leverage common compilers, operating systems, etc.
  - But fairly easy to retarget these for different ISAs (e.g., Linux, gcc)
- Multiple ISAs:
  - Specialized instructions for specialized applications
  - Different tradeoffs in resources used (e.g., functionality, memory demands, complexity, power consumption, etc.)
  - Competition and innovation is good, especially in emerging environments (e.g., mobile devices)

Instruction Set in CS 61C

- MIPS
  - Invented by John Hennessy @ Stanford
  - Why not Berkeley/Sun RISC invented by Dave Patterson? Ask him!
  - MIPS is a real world ISA
  - Standard instruction set for networking equipment
  - Elegant example of Reduced Instruction Set Computer (RISC) instruction set

RISC Design Principles

- Basic RISC principle: “A simpler CPU (the hardware that interprets machine language) is a faster CPU” (CPU ➔ Core)
- Focus of the RISC design is reduction of the number and complexity of instructions in the ISA
- A number of the more common strategies include:
  - Fixed instruction length, generally a single word;
  - Simplifies process of fetching instructions from memory
  - Simplified addressing modes;
  - Simplifies process of fetching operands from memory
  - Fewer and simpler instructions in the instruction set;
  - Simplifies process of executing instructions
  - Only load and store instructions access memory;
  - E.g., no add memory to register, add memory to memory, etc.
  - Let the compiler do it. Use a good compiler to break complex high-level language statements into a number of simple assembly language statements

Mainstream ISAs

- ARM (Advanced RISC Machine) is most popular RISC
  - In every smart phone-like device (e.g., iPhone, iPad, iPod, …)
- Intel 80x86 is another popular ISA and is used in Macbook and PCs in Core i3, Core i5, Core i7, …
  - 20x ARM as 80x86 (i.e., 5 billion vs. 0.3 billion)
Inspired by the IBM 360 “Green Card”

MIPS Instructions

- Every computer does arithmetic
- \textit{Instruct} a computer to do addition:
  \begin{align*}
  \text{add } a, b, c \quad & \text{— Add } b \text{ to } c \text{ and put sum into } a \\
  \text{3 operands: 2 sources + 1 destination for sum} \\
  \text{One operation per MIPS instruction} \\
  \text{How do you write the same operation in C?}
  \end{align*}

Guess More MIPS instructions

- Subtract \( c \) from \( b \) and put difference in \( a \)?
  \( \text{sub } a, b, c \)
- Multiply \( b \) by \( c \) and put product in \( a \)?
  \( \text{mul } a, b, c \)
- Divide \( b \) by \( c \) and put quotient in \( a \)?
  \( \text{div } a, b, c \)

Example Instructions

- MIPS instructions are inflexible, rigid:
  \begin{itemize}
  \item Just one arithmetic operation per instruction
  \item Always with three operands
  \end{itemize}
- How write this C expression in MIPS?
  \begin{align*}
  a &= b + c + d + e \\
  \text{add } t1, d, e \\
  \text{add } t2, c, t1 \\
  \text{add } a, b, t2
  \end{align*}

Comments in MIPS

- Can add comments to MIPS instruction by putting \# that continues to end of line of text
  \begin{align*}
  \text{add a, b, c } \# b + c \text{ is placed in } a \\
  \text{add a, d } \# b + c + d \text{ is now in } a \\
  \text{add a, e } \# b + c + d + e \text{ is in } a
  \end{align*}
- \textit{Are extremely useful!}
C to MIPS

• What is MIPS code that performs same as?
  
  \[ a = b + c; \quad \text{add } a, b, c \]
  \[ d = a - e; \quad \text{sub } d, a, e \]

• What is MIPS code that performs same as?
  
  \[ f = (g + h) - (i + j); \]
  \[ \text{add } t1, i, j \]
  \[ \text{add } t2, g, h \]
  \[ \text{sub } f, t2, t1 \]

For a given function, which programming language likely takes the most lines of code? (most to least)

- Scheme, MIPS, C
- C, Scheme, MIPS
- MIPS, Scheme, C
- **MIPS, C, Scheme**

Agenda

• Machine Language
• Administrivia
• Operands
• Technology Break
• Decisions
• Summary

Administrivia

• This week in lab and homework:
  
  – Project #1 posted
    – Note: intermediate checkpoint due Sunday
    – Find a partner (optional), tell TA who in section
  
  – HW #3 Posted (due a week from Sunday)
  – Lab #3 EC2 to be posted soon
    – TAs say it is more doable than last week

Agenda

• Machine Language
• Administrivia
• Operands
• Technology Break
• Decisions
• Summary

Computer Hardware Operands

• High-Level Programming languages: could have millions of variables
• Instruction sets have fixed, smaller number
• Called registers
  
  – “Bricks” of computer hardware
  
  – Fastest way to store data in computer hardware
  
  – Visible to (the “assembly language”) programmer
• MIPS Instruction Set has 32 registers
Why Just 32 Registers?

• RISC Design Principle: Smaller is faster
  – But you can be too small …
• Hardware would likely be slower with 64, 128, or 256 registers
• 32 is enough for compiler to translate typical C programs, and not run out of registers very often
  – ARM instruction set has only 16 registers
  – May be faster, but compiler may run out of registers too often (aka “spilling registers to memory”)

Names of MIPS Registers

• For registers that hold programmer variables: $s0, s1, s2, …
• For registers that hold temporary variables: $t0, $t1, $t2, …

Names of MIPS Registers

• Suppose variables $f, g, h, i, and j are assigned to the registers $s0, $s1, $s2, $s3, and $s4, respectively. What is MIPS for $f = (g + h) - (i + j);
\[
\text{add } t0, s3, s4 \\
\text{add } t2, s1, s2 \\
\text{sub } s0, t2, t1
\]

Size of Registers

• Bit is the atom of Computer Hardware:
  – True “alphabet” of computer hardware is 0, 1
  – Will eventually express MIPS instructions as combinations of 0s and 1s (in Machine Language)
• MIPS registers are 32 bits wide
• MIPS calls this quantity a word
  – Some computers use 16-bit or 64-bit wide words
  – E.g., Intel 80x86, MIPS64

Data Structures vs. Simple Variables

• In addition to registers, a computer also has memory that holds millions / billions of words
• Memory is a single dimension array, starting at 0
• To access memory, need an address (like an array index)
• But MIPS instructions only operate on registers!
• Solution: instructions specialized to transfer words (data) between memory and registers
  • Called data transfer instructions

Transfer from Memory to Register

• MIPS instruction: Load Word, abbreviated lw
• Assume A is an array of 100 words, variables $g$ and $h$ map to registers $s1$ and $s2$, the starting address/base address of the array A is in $s3$
\[
\text{int A[100];} \\
g = h + A[3];
\]
• Becomes:
\[
lw \ \$t0, 3($s3) \quad \# \text{Temp reg } \$t0 \text{ gets } A[3] \\
\text{add } s1, s2, \$t0 \quad \# g = h + A[3]
\]
Memory Addresses are in Bytes

- Lots of data is smaller than 32 bits, but rarely smaller than 8 bits – works fine if everything is a multiple of 8 bits
- 8 bit item is called a byte (1 word = 4 bytes)
- Memory addresses are really in bytes, not words
- Word addresses are 4 bytes apart
  - Word address is same as leftmost byte

Add of lowest byte in word is addr of word

<table>
<thead>
<tr>
<th>Addr of lowest byte in word</th>
<th>Addr of word</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 12 14 15</td>
<td>0 5 10 15</td>
</tr>
</tbody>
</table>

Transfer from Memory to Register

- MIPS instruction: Load Word, abbreviated lw
- Assume A is an array of 100 words, variables g and h map to registers $s1 and $s2, the starting address/base address of the array A is in $s3
  - $g = h + A[3];
  - Becomes:
    - lw $t0, 12($s3) # Temp reg $t0 gets A[3]
    - add $s1,$s2,$t0 # $g = h + A[3]

Transfer from Register to Memory

- MIPS instruction: Store Word, abbreviated sw
- Assume A is an array of 100 words, variables g and h map to registers $s1 and $s2, the starting address, or base address, of the array A is in $s3
  - Turns into
    - lw $t0,12($s3) # Temp reg $t0 gets A[3]
    - add $t0,$s2,$t0 # $t0 = h + A[3]
    - sw $t0,40($s3) # $A[10] = h + A[3] 

Speed of Registers vs. Memory

- Given that
  - Registers: 32 words (128 Bytes)
  - Memory: Billions of bytes (2 GB to 8 GB on laptop)
- and the RISC principle is...
  - Smaller is faster
- How much faster are registers than memory??
- About 100-500 times faster!

Which of the following is TRUE?

- add $t0,$t1,4($t2) is valid MIPS
- Can byte address 8GB with a MIPS word
- imm must be a multiple of 4 for
  - lw $t0,imm($s0) to be valid
- If MIPS halved the number of registers available, it would be twice as fast
Agenda

- Machine Language
- Administrivia
- Operands
- Technology Break
- Decisions
- Summary

---

**Computer Decision Making**

- Based on computation, do something different
- In programming languages: if-statement
  - Sometimes combined with gotos and labels
- MIPS: if-statement instruction is
  \[ \text{beq register1,register2,L1} \]
  means go to statement labeled L1
  if value in register1 = value in register2
  (otherwise, go to next statement)
- \text{beq} stands for \textit{branch if equal}
- Other instruction: \text{bne for branch if not equal}

---

**Example If Statement**

- Assuming translations below, compile if block
  \[ f \rightarrow s0 \quad g \rightarrow s1 \quad h \rightarrow s2 \]
  \[ i \rightarrow s3 \quad j \rightarrow s4 \]

\[
\begin{align*}
\text{if (i == j)} & \quad \text{bne } s3,s4,\text{Exit} \\
\text{f = g + h;} & \quad \text{add } s0,s1,s2 \\
\text{Exit:} & \\
& \text{May need to negate branch condition}
\end{align*}
\]

---

**Types of Branches**

- **Branch** – change of control flow

- **Conditional Branch** – change control flow depending on outcome of comparison
  - \text{branch if equal (beq)} or \text{branch if not equal (bne)}

- **Unconditional Branch** – always branch
  - a MIPS instruction for this: \textit{jump (:)}

---

**Making Decisions in C or Java**

\[
\begin{align*}
\text{if (i == j)} & \\
\text{f = g + h;} \\
\text{else} & \\
\text{f = g - h;} \\
& \text{If false, skip over “then” part to “else” part} \\
& \Rightarrow \text{use conditional branch bne} \\
& \text{Otherwise, (its true) do “then” part and skip} \\
& \text{over “else” part} \\
& \Rightarrow \text{use unconditional branch j}
\end{align*}
\]
Making Decisions in MIPS

- Assuming translations below, compile

\[
\begin{align*}
  f &\rightarrow s_0 \\
  g &\rightarrow s_1 \\
  h &\rightarrow s_2 \\
  i &\rightarrow s_3 \\
  j &\rightarrow s_4 \\
  \text{if } (i == j) &\text{ bne } s_3, s_4, \text{Else} \\
  f &\rightarrow g + h; \quad \text{add } s_0, s_1, s_2 \\
  \text{else} &\quad j \text{ Exit} \\
  f &\rightarrow g - h; \quad \text{Else: } \text{sub } s_0, s_1, s_2 \quad \text{Exit:}
\end{align*}
\]

Which of the following is FALSE?

- Can make an unconditional branch from a conditional branch instruction
- Can make a loop with \textit{j}
- Can make a loop with \textit{beq}
- Can always return from a function call with \textit{j}

Strings: C vs. Java

- Recall: a string is just a long sequence of characters (i.e., array of chars)
- C: 8-bit ASCII, define strings with end of string character NUL (0 in ASCII)
- Java: 16-bit Unicode, first entry gives length of string

Strings

- “Cal” in ASCII in C; How many bytes?
- Using 1 integer per byte, what does it look like?

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Over</th>
<th>Over</th>
<th>Over</th>
<th>Over</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x41</td>
<td>0x61</td>
<td>0x6c</td>
<td>0x6f</td>
</tr>
<tr>
<td></td>
<td>0x4c</td>
<td>0x6f</td>
<td>0x6c</td>
<td>0x6f</td>
</tr>
<tr>
<td></td>
<td>0x6f</td>
<td>0x6f</td>
<td>0x6f</td>
<td>0x6f</td>
</tr>
</tbody>
</table>

Support for Characters and Strings

- Load a word, use andi to isolate byte
  \[
  \text{lw } s_0, 0(s_1) \\
  \text{andi } s_0, s_0, 255 \quad \# \text{ Zero everything but last 8 bits}
  \]
- RISC Design Principle: “Make the Common Case Fast”—Many programs use text: MIPS has \textit{load byte} instruction (\textit{lb})
  \[
  \text{lb } s_0, 0(s_1)
  \]
- Also \textit{store byte} instruction (\textit{sb})
Support for Characters and Strings

- Load a word, use `andi` to isolate half of word
  
  ```
  lw $s0,0($s1)
  andi $s0,$s0,65535  # Zero everything but last 16 bits
  ```

- RISC Design Principle #3: “Make the Common Case Fast”—Many programs use text, MIPS has `load halfword` instruction (`lh`)
  
  ```
  lh $s0,0($s1)
  ```

- Also `store halfword` instruction (`sh`)

And In Conclusion ...

- Computer words and vocabulary are called instructions and instruction set respectively
- MIPS is example RISC instruction set in this class
- Rigid format: 1 operation, 2 source operands, 1 destination
  - add, sub, mul, div, and, or, sl, srl
  - lw, sw to move data to/from registers from/to memory
- Simple mappings from arithmetic expressions, array access, if-then-else in C to MIPS instructions