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**
  Today's Lecture
Levels of Representation/Interpretation

<table>
<thead>
<tr>
<th>High Level Language Program (e.g., C)</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly Language Program (e.g., MIPS)</td>
<td>Assembler</td>
</tr>
<tr>
<td>Machine Language Program (MIPS)</td>
<td>Machine Interpretation</td>
</tr>
<tr>
<td>Hardware Architecture Description (e.g., block diagrams)</td>
<td>Logic Circuit Description (Circuit Schematic Diagrams)</td>
</tr>
</tbody>
</table>

Compiler:
- `lw $t0, 0($2)`
- `lw $t1, 4($2)`
- `sw $t1, 0($2)`
- `sw $t0, 4($2)`

Assembler:
- `temp = v[k];`  
- `v[k] = v[k+1];`  
- `v[k+1] = temp;`

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Pointers and Strings in C

- `char *p;` # `p` is a pointer to a character
- `char x[] = “Randy Katz”;` # `x` points to a literal string
- `p = x;` # `p` points to the same place as `x`
- `p = &x[0];` # same as `p = x`
- `Cannot write x = “Randy Katz”;`
- `Strings are not a primitive type in C (but character arrays are)`
- `Element/character at a time processing possible, but whole string processing requires special routines`
Agenda

• Machine Language
• Administrivia
• Operands
• Technology Break
• Decisions
• And in Conclusion ...
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

The Language a Computer Understands

• Why not all the same? Why not all different? What might be pros and cons?
The Language a Computer Understands

• 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)

MIPS:
Instruction Set for CS 61C

• MIPS is a real-world ISA (see www.mips.com)
  – Standard instruction set for networking equipment
  – Was also used in original Nintendo-64!
• Elegant example of a Reduced Instruction Set Computer (RISC) instruction set
• Invented by John Hennessy @ Stanford
  – Why not Berkeley/Sun RISC invented by Dave Patterson? Ask him!
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 (Core i3, Core i5, Core i7, ...)
  - x86 is a Complex Instruction Set Computer (CISC)
  - 20x ARM sold vs. 80x86 (i.e., 5 billion vs. 0.3 billion)
MIPS Instructions

- Every computer does arithmetic
- *Instruct* a computer to do addition:
  
  ```
  add a, b, c
  ```
  - Add b to c and put sum into a
- 3 operands: 2 sources + 1 destination for sum
- One operation per MIPS instruction
- How do you write the same operation in C?
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 \( a \)?

\[ \text{div } a, b, c \]
Guess More MIPS instructions

- C operator &: c & b with result in a?
- C operator |: c | b with result in a?
- C operator <: b << c with result in a?
- C operator >>: b >> c with result in a?
Example Instructions

• MIPS instructions are inflexible, rigid:
  – Just one arithmetic operation per instruction
  – Always with three operands
• How write this C expression in MIPS?
  \[ a = b + c + d + e \]
Comments in MIPS

• Can add comments to MIPS instruction by putting # that continues to end of line of text

  add a, b, c # b + c is placed in a
  add a, a, d # b + c + d is now in a
  add a, a, e # b + c + d + e is in a

• Are extremely useful!

C to MIPS

• What is MIPS code that performs same as?

  a = b + c;
  d = a - e;

• What is MIPS code that performs same as?

  f = (g + h) - (i + j);
C to MIPS

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

- 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)

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

- Python, MIPS, C
- C, Python, MIPS
- MIPS, Python, C

- **MIPS, C, Python**

---

**Agenda**

- Machine Language
- **Administrivia**
- Operands
- Technology Break
- Decisions
- And in Conclusion ...

---
Administrivia

• This week in lab and homework:
  – HW #2 due Sunday
  – Lab #3 EC2 to be posted soon

CS61c in the News
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
  
  \[ f = (g + h) - (i + j); \]

  
  add $t1, s3, s4$
  
  add $t2, s1, s2$
  
  sub $s0, t2, t1$
Size of Registers

• *Bit* is the atom of Computer Hardware: contains either 0 or 1
  – 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`
- `int A[100];`
  
  `g = h + A[3];`
- **Becomes:**
  
  `lw $t0, 3($s3)  # Temp reg $t0 gets A[3]`
  `add $s1, $s2, $t0  # 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
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:
  
  \[
  \begin{align*}
  &\text{lw } $t0, 12($s3) \quad \# \text{ Temp reg } $t0 \text{ gets } A[3] \\
  &\text{add } $s1,$s2,$t0 \quad \# g = h + A[3]
  \end{align*}
  \]

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
  
  \[
  \begin{align*}
  &\text{lw } $t0,12($s3) \quad \# \text{ Temp reg } $t0 \text{ gets } A[3] \\
  &\text{add } $t0,$s2,$t0 \quad \# t0 = h + A[3] \\
  \end{align*}
  \]
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
  \[
  \text{lw} \quad $t0,12($s3) \quad \# \text{Temp reg $t0$ gets } A[3] \\
  \text{add} \quad $t0,$s2,$t0 \quad \# \text{$t0 = h + A[3]$} \\
  \text{sw} \quad $t0,40($s3) \quad \# A[10] = h + A[3]
  \]

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

Which of the following is TRUE?

NONE!

☐ 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
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: one operation, two source operands, one destination
  - add, sub, mul, div, and, or, sll, 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