CS 61C: Great Ideas in Computer Architecture (Machine Structures)
Introduction to Machine Language

Instructors:
Randy H. Katz
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/sp11

2/3/11
Spring 2011 - Lecture #5

Agenda

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

2/3/11
Spring 2011 - Lecture #5

Levels of Representation/Interpretation

High-Level Language Program (e.g., C)

Machine Language Program (e.g., MIPS)

Assembler

Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

2/3/11
Spring 2011 - Lecture #5

2/3/11
Spring 2011 - Lecture #5

2/3/11
Spring 2011 - Lecture #5

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

2/3/11
Spring 2011 - Lecture #5

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

2/3/11
Spring 2011 - Lecture #5

Agenda

• Machine Language
• Administrivia
• Operands
• Technology Break
• Strings
• Summary
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)

Instruction Set in CS61c
• 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
  – Simplified process of fetching instructions from memory
  – Simplified addressing modes
  – Simplified 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

CS61c’s Instruction Set
• Most popular RISC is ARM (Advanced RISC Machine)
  – In every smart phone-like device (e.g., iPhone, iPad, iPod, …)
• Another popular Instruction Set is used in Macbook and PCs: Intel 80x86 in Core i3, Core i5, Core i7, …
  – 20x ARM versus 80x86 (i.e., 05 billion vs. .3 billion)

MIPS Green Card
MIPS Green Card

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?

Inspired by the IBM 360 “Green Card”

Guess More MIPS instructions

- Subtract c from b and put difference in a?
  sub a, b, c
- Multiply b by c and put product in a?
  mul a, b, c
- Divide b by c and put quotient in a?
  div a, b, c

Guess More MIPS instructions

- Subtract c from b and put difference in a?
- Multiply b by c and put product in a?
- Divide b by c and put quotient in a?
- C operator &: c & b with result in a?
- C operator |: c | b with result in a?
- C operator <<: c << b with result in a?
- C operator >>: c >> b with result in a?
Guess More MIPS instructions

- C operator &: c & b with result in a?
  and a, b, c
- C operator |: c | b with result in a?
  or a, b, c
- C operator <<: c << b with result in a?
  sll a, b, c
- C operator >>: c >> b with result in a?
  srl a, b, c

Example Instructions

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

Example Instructions

- How write this C expression in MIPS?
  \[ a = b + c + d + e \]
  add t1, d, e
  add t2, c, t1
  add a, b, t2

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

C to MIPS

- Operators almost identical
- 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?
  \[ a = b + c; \]
  \[ add a, b, c \]
  \[ d = a - e; \]
  \[ sub d, a, e \]
- What is MIPS code that performs same as?
  \[ f = (g + h) - (i + j); \]
  \[ add t1, i, j \]
  \[ add t2, g, h \]
  \[ sub f, t2, t1 \]
Peer Instruction

• For a given function, which programming language likely takes the most lines of code?
  
I. Scheme
II. C
III. MIPS instructions

Put the three representations in order:
Green. II, III, I  Yellow. I, II, III
Pink. III, II, I  Blue. III, I, II

Agenda

• Machine Language
• Administrivia
• Operands
• Technology Break
• Strings
• 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
  — Used to construct computer hardware
  — Visible to (the “assembly language”) programmer
• MIPS Instruction Set has 32 registers

Administrivia

• This week in lab and homework:
  — Lab #3 EC2 posted (TAs say it is more doable than last week)
    • Note: labs graded on a scale of 0 to 2
  — HW #3 Posted
    • Note: HWs graded on a scale from 0 to 3
  — Project #1 posted
    • Note: intermediate checkpoint due Sunday!
• HKN Announcements/Tutoring
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, ...
- 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
  - Subtract $s0, $t2, $t1

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 $t2, s1, s2$
  - Subtract $s0, t2, t1$

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
- MIPS registers are 32 bits wide
- MIPS calls this quantity a *word*
  - Some computers use 16-bit wide words
  - E.g., Intel 80x86

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:
  
  `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!

Agenda

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

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

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?

(For Latin alphabet, 1st byte is 0, 2nd byte is ASCII)
Strings

• “Cal” in Unicode in Java; How many bytes?
• Using 1 integer per byte, what does it look like?
  (For Latin alphabet, 1st byte is 0, 2nd byte is ASCII)

Support for Characters and Strings

• Load a word, use andi to isolate byte
  \text{lw} \ $s0,0($s1)
  \text{andi} \ $s0,$s0,255 \ # Zero everything but last 8 bits
• RISC Design Principle: “Make the Common Case Fast”—Many programs use text: MIPS has \textit{load byte} instruction (\text{lb})
  \text{lb} \ $s0,0($s1)
• Also \textit{store byte} instruction (\text{sb})

Support for Characters and Strings

• Load a word, use andi to isolate byte
  \text{lw} \ $s0,0($s1)
  \text{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 \textit{load halfword} instruction (\text{lh})
  \text{lh} \ $s0,0($s1)
• Also \textit{store halfword} instruction (\text{sh})

Making Decisions in C or Java

\text{if (i == j)} \ f = g + h; \text{else} \ f = g - h;
• If false, skip over “then” part to “else” part
  \text{=>} \ use conditional branch \text{bne}
• Otherwise, (its true) do “then” part and skip over “else” part \text{=>} \ need an always branch instruction (“\textbf{unconditional branch}”)
• MIPS name for this instruction: \textit{jump} (\text{j})

Making Decisions in MIPS

\text{if (i == j)} \ f = g + h; \text{else} \ f = g - h;
• \text{f} \Rightarrow $s0, \text{g} \Rightarrow $s1, \text{h} \Rightarrow $s2, \text{i} \Rightarrow $s3, \text{j} \Rightarrow $s4
• If false, skip “then” part to “else” part
• Otherwise, (its true) do “then” part and skip over “else” part
Making Decisions in MIPS

if (i == j) f = g + h; else f = g - h;
• f => $s0, g => $s1, h => $s2, i => $s3, j => $s4
• If false, skip "then" part to "else" part
• Otherwise, (its true) do "then" part and skip over "else" part

```
  bne $s3,$s4,Else  # go to Else part if i # j
  add $s0,$s1,$s2  # f = g + h (Then part)
  j Exit
Else: sub $s0,$s1,$s  # f = g - h (Else part)
Exit:
```

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, 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