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

Instructors:
Randy H. Katz
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/fa10

Agenda

• Review
• Overflow and Real Numbers
• Administrivia
• Technology Break
• Instructions as Numbers
• Assembly Language to Machine Language
• Summary
Review from Last Lecture

- Registers selectively saved/restored on call
  - Saved registers $s0-$s7; temporary regs $t0-$t9 not saved
- C splits memory into text, static, heap, stack, with registers dedicated to support: $gp, $sp, $fp
- Program can interpret binary number as unsigned integer, two’s complement signed integer, floating point number, ASCII characters, Unicode characters, ...
- Integers have largest positive and largest negative numbers, but represent all in between
  - Two’s comp. weirdness is one extra negative num

What if result of operation doesn’t fit in 32 bits?

- Called *overflow*: calculate too big a number to represent within a word
- Unsigned numbers: $1 + 4,294,967,295 (2^{32}-1)$
- Signed numbers: $1 + 2,147,483,647 (2^{31}-1)$
Answer Depends on Language

- C unsigned number arithmetic ignores overflow (arithmetic modulo $2^{32}$)
  
  \[ 1 + 4,294,967,295 = \]

- C signed number arithmetic also ignores overflow
  
  \[ 1 + 2,147,483,647 (2^{31}-1) = \]

- Other languages want overflow signal on signed numbers (e.g., Fortran)

- What’s a computer architect to do?

MIPS Solution: offer both

- Instructions that overflow:
  - add, sub, mult, div, addi, multi, divi

- Instructions that don’t overflow called “unsigned” (but really means no overflow):
  - addu, subu, multu, divu, addiu, multiu, diviu

- Given semantics of C, always use unsigned versions

- Note: slt and slti do signed comparisons, while sltu and sliu do unsigned comparisons
  - Nothing to do with overflow
  - When would get different answer for slt vs. sltu?
What about Real Numbers?

- Normalized scientific notation (aka standard form or exponential notation):
  - \( r \times E^i \), \( E \) is where exponent (usually 10), \( i \) is a positive or negative integer, \( r \) is a real number \( \geq 1.0, < 10 \)
  - Normalized => No leading 0s
  - 61 is \( 6.10 \times 10^2 \), 0.000061 is \( 6.10 \times 10^{-5} \)
- Computers version of normalized scientific notation called \textit{Floating Point} notation
- \( r \times E^i \), \( E \) where \( i \) is exponent (2), \( i \) is a positive or negative integer, \( r \) is a real number \( \geq 1.0, < 2 \)

Floating Point Numbers

- 32-bit word has \( 2^{32} \) patterns, so must be approximation of real numbers \( \geq 1.0, < 2 \)
- IEEE 754 Floating Point Standard:
  - 1 bit for \textit{sign} (\( s \)) of floating point number
  - 8 bits for \textit{exponent} (\( E \))
  - 23 bits for \textit{fraction} (\( F \))
    (get 1 extra bit of precision if leading 1 is implicit)
    \((-1)^s \times (1 + F) \times 2^E\)
- Can represent from \( 2.0 \times 10^{-38} \) to \( 2.0 \times 10^{38} \)
Floating Point Numbers

• What about bigger or smaller numbers?
• IEEE 754 Floating Point Standard:
  *Double Precision* (64 bits)
    – 1 bit for *sign* \((s)\) of floating point number
    – 11 bits for *exponent* \((E)\)
    – 52 bits for *fraction* \((F)\)
    (get 1 extra bit of precision if leading 1 is implicit)
  \((-1)^{s} \times (1 + F) \times 2^{E}\)
• Can represent from \(2.0 \times 10^{-308}\) to \(2.0 \times 10^{308}\)
• 32 bit format called *Single Precision*

More Floating Point

• What about 0?
  – Bit pattern all 0s means 0, so no implicit leading 1
• What if divide 1 by 0?
  – Can get infinity symbols \(+\infty, -\infty\)
  – Sign bit 0 or 1, largest exponent, 0 in fraction
• What if do something stupid? \((\infty - \infty, 0 \div 0)\)
  – Can get special symbols NaN for Not-a-Number
  – Sign bit 0 or 1, largest exponent, not zero in fraction
• What if result is too big? \((2 \times 10^{308} \times 2 \times 10^{2})\)
  – Get overflow in exponent, alert programmer!
• What if result is too small? \((2 \times 10^{-308} \div 2 \times 10^{2})\)
  – Get underflow in exponent, alert programmer!
MIPS Floating Point Instructions

• C, Java has single precision (float) and double precision (double) types
• MIPS instructions: .s for single, .d for double
  – Fl. Pt. Addition single precision:
    Fl. Pt. Addition double precision:
  – Fl. Pt. Subtraction single precision:
    Fl. Pt. Subtraction double precision:
  – Fl. Pt. Multiplication single precision:
    Fl. Pt. Multiplication double precision:
  – Fl. Pt. Divide single precision:
    Fl. Pt. Divide double precision:

• Since rarely mix integers and Fl. Pt., MIPS has separate
  registers for floating-point operations: $f0, f1, ..., $f31
  – Double precision uses adjacent even-odd pairs of registers:
    – $f0 and $f1, $f2 and $f3, $f4 and $f5, ..., $f30 and $f31
• Need data transfer instructions for these new registers
  – lw1 (load word), swc1 (store word)
  – Double precision uses two lw1 instructions, two swc1 instructions
Peer Instruction

Suppose Big, Tiny, and BigNegative are floats in C, with Big initialized to a big number (e.g., age of universe in seconds or $4.32 \times 10^{17}$), Tiny to a small number (e.g., seconds/femtosecond or $1.0 \times 10^{-15}$), BigNegative = - Big. Here are two conditionals about associativity:

I. $(\text{Big} \times \text{Tiny}) \times \text{BigNegative} == (\text{Big} \times \text{BigNegative}) \times \text{Tiny}$

II. $(\text{Big} + \text{Tiny}) + \text{BigNegative} == (\text{Big} + \text{BigNegative}) + \text{Tiny}$

Which statement below is correct?

A. I. is false and II. is false
B. I. is false and II. is true
C. I. is true and II. is false
D. I. is true and II. is true

Pitfall

• Floating point addition is NOT associative
• Some optimizations can change order of floating point computations, which can change results
• Need to ensure that floating point algorithm is correct even with optimizations
Peer Instruction

• Increase real-time learning in lecture, test understanding of concepts vs. details
  mazur-www.harvard.edu/education/pi.phtml

• As complete a “segment”
  ask multiple choice question
  – 1-2 minutes: decide yourself, vote
  – 2-3 minutes: discuss in pairs, then team vote; flash cards
    • Try to convince partner; learn by teaching

Instructions as Numbers

• Instructions are kept as binary numbers in memory too
  – Stored program concept
    – As easy to change programs as it is to change data
• Saw mapping of register names to numbers
• Need to map instruction operation to a part of number
Instructions as Numbers

• `addu $t0,$s1,$s2`
  - Destination register $t0$ is register 8
  - Source register $s1$ is register 17
  - Source register $s2$ is register 18
  - Add unsigned instruction encoded as number 33

<table>
<thead>
<tr>
<th>0</th>
<th>17</th>
<th>18</th>
<th>8</th>
<th>0</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>10001</td>
<td>10010</td>
<td>01000</td>
<td>00000</td>
<td>100001</td>
</tr>
</tbody>
</table>

- 6 bits 5 bits 5 bits 6 bits

• Groups of bits call *fields* (unused field default is 0)
• Layout called *instruction format*
• Binary version called *machine instruction*

Instructions as Numbers

• `sll $zero,$zero,0`
  - $zero$ is register 0
  - Shift amount 0 is 0
  - Shift left logical instruction encoded as number 0

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td></td>
</tr>
</tbody>
</table>

- 6 bits 5 bits 5 bits 6 bits

• Can also represent machine code as base 16 or base 8 number: 0000 0000_{hex}, 0000000000_{oct}
Everything in a computer is just a Binary Number

• Up to program to decide what data means
• Example 32-bit data shown as binary number:
  0000 0000 0000 0000 0000 0000 0000 _two
What does it mean if its treated as
1. Signed integer
2. Unsigned integer
3. Floating point
4. ASCII characters
5. Unicode characters
6. MIPS instruction

Implications of everything is a number?

• Stored program concept
  – Invented about 1947 (many claim invention)
• As easy to change programs as to change data!
• Implications?
Names of MIPS fields

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- **op**: Basic operation of instruction, or *opcode*
- **rs**: 1<sup>st</sup> register source operand
- **rt**: 2<sup>nd</sup> register source operand.
- **rd**: register destination operand (result of operation)
- **shamt**: Shift amount.
- **funct**: Function. This field, often called *function code*, selects the specific variant of the operation in the op field

What about load, store, immediate, branches, jumps?

- Fields for constants only 5 bits (-16 to +15)
  - Too small for many common cases
- #1 Simplicity favors regularity (all instructions use one format) vs. #3 Make common case fast (multiple instruction formats)?
- 4<sup>th</sup> Design Principle: *Good design demands good compromises*
- Better to have multiple instruction formats and keep all MIPS instructions same *size*
  - All MIPS instructions are 32 bits or 4 bytes
**Names of MIPS fields in I-type**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address or constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- **op**: Basic operation of instruction, or *opcode*
- **rs**: 1\(^{st}\) register source operand
- **rt**: 2\(^{nd}\) register source operand for branches but register destination operand for lw, sw, and immediate operations
- **Address/constant**: 16-bit two’s complement number
  - Note: equal in size of rd, shamt, funct fields

---

**Register (R), Immediate (I), Jump (J) Instruction Formats**

<table>
<thead>
<tr>
<th>R-type</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I-type</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address or constant</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- Now loads, stores, branches, and immediates can have 16-bit two’s complement address or constant: -32,768 \((-2^{15})\) to +32,767 \((2^{15}-1)\)
- What about jump, jump and link?

<table>
<thead>
<tr>
<th>J-type</th>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>
Encoding of MIPS Instructions: Must Be Unique!

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu</td>
<td>R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>33_{ten}</td>
<td>n.a.</td>
</tr>
<tr>
<td>subu</td>
<td>R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>35_{ten}</td>
<td>n.a.</td>
</tr>
<tr>
<td>situ</td>
<td>R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>43_{ten}</td>
<td>n.a.</td>
</tr>
<tr>
<td>sll</td>
<td>R</td>
<td>0</td>
<td>reg</td>
<td>n.a</td>
<td>reg</td>
<td>constant</td>
<td>0_{ten}</td>
<td>n.a.</td>
</tr>
<tr>
<td>addi unsigned</td>
<td>I</td>
<td>9_{ten}</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>constant</td>
</tr>
<tr>
<td>lw (load word)</td>
<td>I</td>
<td>35_{ten}</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>address</td>
</tr>
<tr>
<td>sw (store word)</td>
<td>I</td>
<td>43_{ten}</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>address</td>
</tr>
<tr>
<td>beq</td>
<td>I</td>
<td>4_{ten}</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>address</td>
</tr>
<tr>
<td>bne</td>
<td>I</td>
<td>5_{ten}</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>address</td>
</tr>
<tr>
<td>j (jump)</td>
<td>J</td>
<td>2_{ten}</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>address</td>
</tr>
<tr>
<td>jal</td>
<td>J</td>
<td>3_{ten}</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>address</td>
</tr>
<tr>
<td>jr (jump reg)</td>
<td>R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>8_{ten}</td>
<td>n.a.</td>
</tr>
</tbody>
</table>

Summary

- Integer and floating point operations can lead to results too big to store within their representations: overflow/underflow
- Floating point is an approximation of reals
- Everything is a (binary) number in a computer
  - Instructions and data; stored program concept
- MIPS ISA guided by 4 design principles:
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
  - MIPS has 3 instruction formats, but all instructions 32 bits