Big Idea #1: 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>
</tbody>
</table>

temp = v[k];
v[k+1] = temp;

Anything can be represented as a number; i.e., data or instructions

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

Instructor:
Krsti Asanovic, Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa12

Agenda

- Review
- Instructions as Numbers
- Administration
- Floating Point Numbers
- And in Conclusion, ...

Optimized Function Convention

To reduce expensive loads and stores from spilling and restoring registers, MIPS divides registers into two categories:

1. Preserved across function call
   - Caller can rely on values being unchanged
   - $ra, $sp, $gp, $fp, “saved registers” $s0-$s7
2. Not preserved across function call
   - Caller cannot rely on values being unchanged
   - Return value registers $v0,$v1, Argument registers $a0-$a3, “temporary registers” $t0-$t9

Where is the Stack in Memory?

- MIPS convention
- Stack starts in high memory and grows down
  - Hexadecimal (base 16): 7fff fffehex
- MIPS programs (text segment) in low end
  - 0040 0000hex
- static data segment (constants and other static variables) above text for static variables
  - MIPS convention global pointer ($gp) points to static
  - (30 of 32, 2 left!) – will see when talk about OS
- Heap above static for data structures that grow and shrink; grows up to high addresses

MIPS Memory Allocation

- Stack
- Dynamic data
- Static data
- Text
- Reserved
Signed Integers and Two’s Complement Representation

- Signed integers in C; want \( \frac{1}{2} \) numbers <0, want \( \frac{1}{2} \) numbers >0, and want one 0
- Two’s complement treats 0 as positive, so 32-bit word represents \( 2^{32} \) integers from \(-2^{31} \) to \( 2^{31} -1 \) (2,147,483,648)
  - Note: one negative number with no positive version
  - Book lists some other options, all of which are worse
- Every computer uses two’s complement today

- Most significant bit (leftmost) is the sign bit, since 0 means positive (including 0), 1 means negative
  - Bit 31 is most significant, bit 0 is least significant

Twos Complement Examples

- Assume for simplicity 4 bit width, -8 to +7 represented

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>+2</td>
<td>0010</td>
<td>+(-2)</td>
<td>1110</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>1</td>
<td>10000</td>
</tr>
</tbody>
</table>

- Overflow: Carry into MSB = Carry Out MSB
- Underflow: Carry into MSB = Carry Out MSB

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 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?

Instructions as Numbers

- Instructions are also kept as binary numbers in memory
  - Stored program concept
  - As easy to change programs as it is to change data
- Register names mapped to numbers
- Need to map instruction operation to a part of number
Names of MIPS fields

- **op**: Basic operation of instruction, or *opcode*
- **rs**: 1st register source operand
- **rt**: 2nd 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.

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>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

  - Can also represent machine code as base 16 or base 8 number: 0000000000000000

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)?
- 4th 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

- **op**: Basic operation of instruction, or *opcode*
- **rs**: 1st register source operand
- **rt**: 2nd 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

- **R-type**
  - 6 bits 5 bits 5 bits 5 bits 5 bits 6 bits
- **I-type**
  - 6 bits 5 bits 5 bits 5 bits 5 bits 6 bits
  - Now loads, stores, branches, and immediates can have 16-bit two's complement address or constant: -32,768 (-2<sup>15</sup>) to +32,767 (2<sup>15</sup>-1)
- **What about jump, jump and link?**
Encoding of MIPS Instructions: Must Be Unique!

<table>
<thead>
<tr>
<th>Instruction</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</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</td>
</tr>
<tr>
<td>sll</td>
<td>R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>43</td>
</tr>
<tr>
<td>addi signed</td>
<td>I</td>
<td>32</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>I</td>
<td>35</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>33</td>
</tr>
<tr>
<td>sw</td>
<td>I</td>
<td>35</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>33</td>
</tr>
</tbody>
</table>

Converting C to MIPS Machine code

\$t0 (reg 8), &A in \$t1 (reg 9), h=\$s2 (reg 18)

A[300] = h + A[300];

\[
\begin{align*}
  \text{lw } \&t0,1200(\&t1) &\quad 35 \quad 9 \quad 8 \quad 1200 \\
  \text{addu } \&t0,\&s2,\&t0 &\quad 0 \quad 18 \quad 8 \quad 0 \quad 33 \\
  \text{sw } \&t0,1200(\&t1) &\quad 43 \quad 9 \quad 8 \quad 1200 \\
\end{align*}
\]

Agenda

- Review
- Floating Point Numbers
- Administrivia
- Instructions as Numbers
- And in Conclusion, ...

CS61c in the News

- iPhone 5, 9/12/12
  - ARM Cortex A-15 core
  - Claimed 2x Performance of iPhone 4's A6 Chip (fabricated by Samsung)!
  - Still dual core
Agenda

• Review
• Instructions as Numbers
• Administrivia
• Floating Point Numbers
• And in Conclusion, ...

Goals for Floating Point

• Standard arithmetic for reals for all computers
  – Like two’s complement
• Keep as much precision as possible in formats
• Help programmer with errors in real arithmetic
  – \( \pm \infty, -\infty \), Not-A-Number (NaN), exponent overflow, exponent underflow
• Keep encoding that is somewhat compatible with two’s complement
  – E.g., 0 in Fl. Pt. is 0 in two’s complement
  – Make it possible to sort without needing to do floating point comparison

Scientific Notation (e.g., Base 10)

• Normalized scientific notation (aka standard form or exponential notation):
  – \( r \times E \); \( E \) is exponent (usually 10), \( i \) is a positive or negative integer, \( r \) is a real number \( \geq 1.0, < 10 \)
  – Normalized \( \Rightarrow \) No leading 0s
  – 61 is \( 6.10 \times 10^1 \); 0.000061 is \( 6.10 \times 10^{-5} \)

Scientific Notation (e.g., Base 10)

• \( (r \times e^i) \times (s \times e^j) = (r \times s) \times e^{i+j} \)
  – \( (1.999 \times 10^2) \times (5.5 \times 10^3) = 10.9945 \times 10^5 = 1.09945 \times 10^6 \)
  – \( (r \times e^i) / (s \times e^j) = (r / s) \times e^{i-j} \)
  – \( (1.999 \times 10^2) / (5.5 \times 10^3) = 0.3634545... \times 10^{-1} = 3.634545... \times 10^{-2} \)
  – For addition/subtraction, you first must align:
    – \( (1.999 \times 10^2) + (5.5 \times 10^3) = (1.1999 \times 10^2) + (5.5 \times 10^3) = 5.6999 \times 10^3 \)

Which is Less?
(i.e., closer to \( -\infty \))

• 0 vs. \( 1 \times 10^{-127} ? \)
• \( 1 \times 10^{-125} \) vs. \( 1 \times 10^{-127} ? \)
• \(-1 \times 10^{-127} \) vs. 0?
• \(-1 \times 10^{-126} \) vs. \(-1 \times 10^{-127} ? \)
Floating Point: Representing Very Small Numbers

- **Zero:** Bit pattern of all 0s is encoding for 0.000
  - But 0 in exponent should mean most negative exponent (want 0 to be next to smallest real)
  - Can’t use two’s complement (1000 0000\text{two})
- **Bias notation:** subtract bias from exponent
  - Single precision uses bias of 127; DP uses 1023
  - 0 uses 0000 0000\text{two} \Rightarrow 0-127 = -127;
  - \infty, NaN uses 1111 1111\text{two} \Rightarrow 255-127 = +128
  - Smallest SP real can represent: 1.00…00 x 2^{-126}
  - Largest SP real can represent: 1.11…11 x 2^{127}

Bias Notation (+127)

<table>
<thead>
<tr>
<th>Decimal Value of</th>
<th>How it is encoded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponent</td>
<td>Normalized</td>
</tr>
<tr>
<td></td>
<td>Exponent</td>
</tr>
<tr>
<td></td>
<td>1111 1111</td>
</tr>
<tr>
<td></td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td>1111 1111</td>
</tr>
<tr>
<td></td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td>0000 0000</td>
</tr>
</tbody>
</table>

Getting closer to zero

- Zero
- For Infinities
- For Normals

What If Operation Result Doesn’t Fit in 32 Bits?

- **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)

Depends on the Programming Language

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

Depends on the Programming Language

- **C signed number arithmetic also ignores overflow**
  - 1 + 2,147,483,647 (2^{31} - 1) =
Depends on the Programming Language

• C signed number arithmetic also ignores overflow
  \[ 1 + 2,147,483,647 \times 2^{31-1} = 1 + \text{FFFF}_{\text{hex}} = \text{FFFF}_{\text{hex}} = -1 \]

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

• What’s a computer architect to do?

MIPS Solution: Offer Both

• Instructions that can trigger overflow:
  – add, sub, mult, div, addi, multi, divi

• Instructions that don’t overflow are called “unsigned” (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 sltiu do unsigned comparisons
  – Nothing to do with overflow
  – When would get different answer for slt vs. sltu?

What About Real Numbers in Base 2?

• \[ r \times 2^i, \text{ where exponent is } (2), i \text{ is a positive or negative integer}, r \text{ is a real number } \geq 1.0, < 2 \]

• Computers version of normalized scientific notation called Floating Point notation

Floating Point Numbers

• 32-bit word has \( 2^{23} \) patterns, so must be approximation of real numbers \( \geq 1.0, < 2 \)

• IEEE 754 Floating Point Standard:
  – 1 bit for sign \((s)\) of floating point number
  – 8 bits for exponent \((E)\)
  – 23 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 \(2.0 \times 10^{-38}\) 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

Floating Point Add Associativity?

- \(A = (1000000.0 + 0.000001) - 1000000.0\)
- \(B = (1000000.0 - 1000000.0) + 0.000001\)
- In single precision floating point arithmetic, A does not equal B
  - \(A = 0.000000, B = 0.0000001\)
- Floating Point Addition is not Associative!
  - Integer addition is associative
- When does this matter?

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? (\(= -\infty, 0 \div 0\))
  - 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? (\(2 \times 10^{308} \times 2 \times 10^3\))
  - Get overflow in exponent, alert programmer!
- What if result is too small? (\(2 \times 10^{-308} + 2 \times 10^3\))
  - 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: add.s
    Fl. Pt. Add double precision: add.d
  - Fl. Pt. Subtraction single precision: sub.s
    Fl. Pt. Subtraction double precision: sub.d
  - Fl. Pt. Multiplication single precision: mul.s
    Fl. Pt. Multiplication double precision: mul.d
  - Fl. Pt. Divide single precision: div.s
    Fl. Pt. Divide double precision: div.d

MIPS Floating Point Instructions

- C, Java have single precision (float) and double precision (double) types
- MIPS instructions: .s for single, .d for double
  - Fl. Pt. Comparison single precision: flt.s
    Fl. Pt. Comparison double precision: flt.d
  - Fl. Pt. branch:
    - Since rarely mix integers and Floating Point, MIPS has separate registers for floating-point operations: $f0, f1, \ldots, f31$
      - Double precision uses adjacent even-odd pairs of registers:
        - $f0 \text{ and } f1, f2 \text{ and } f3, f4 \text{ and } f5, \ldots, f30 \text{ and } f31$
    - Need data transfer instructions for these new registers
      - lwcl (load word), swcl (store word)
      - Double precision uses two lwcl instructions, two swcl instructions
Peer Instruction Question

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:

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 about these is correct?

Orange. I. is false and II. is false
Green. I. is false and II. is true
Pink. I. is true and II. is false
Yellow. I. is true and II. is true

Peer Instruction Answer

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:

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 about these is correct?

Yellow. I. is true and II. is false (if we don’t consider overflow)—but there are cases where one side overflows while the other does not!

I. Works ok if no overflow, but because exponents add, if Big \text{ * } BigNeg overflows, then result is overflow, not -1

II. Left hand side is 0, right hand side is tiny

Pitfalls

• 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

“And in Conclusion, …”

• Program can interpret binary number as unsigned integer, two’s complement signed integer, floating point number, ASCII characters, Unicode characters, ... even instructions!
• Integers have largest positive and largest negative numbers, but represent all in between
  – Two’s comp. weirdness is one extra negative numinteger and floating point operations can lead to results too big to store within their representations: overflow/underflow
• Floating point is an approximation of reals