CS 61C:
Great Ideas in Computer Architecture
*Instructions as Numbers*

Instructor:
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/sp12

Spring 2012 - Lecture #7

**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

**Agenda**

- Everything is a Number
- Administrivia
- Overflow and Real Numbers
- Instructions as Numbers
- Assembly Language to Machine Language
- Summary

**Key Concepts**

- Inside computers, everything is a number
- But everything is of a fixed size
  - 8-bit bytes, 16-bit half words, 32-bit words, 64-bit double words, ...
- Integer and floating point operations can lead to results too big to store within their representations: *overflow/underflow*
Signed and Unsigned Integers

- C, C++, and Java have signed integers, e.g., 7, -255:
  ```
  int x, y, z;
  ```
- C, C++ also have unsigned integers, which are used for addresses
- 32-bit word can represent 2^{32} binary numbers
- Unsigned integers in 32 bit word represent 0 to 2^{32} - 1 (4,294,967,295)

Signed Integers and Two’s Complement Representation

- Signed integers in C want ½ numbers <0, want ½ 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 (2,147,483,647)
  - Note: one negative number with no positive version
  - Book lists some other options, all of which are worse
  - Every computers 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

Suppose we had a 5 bit word. What integers can be represented in two’s complement?

- -32 to +31
- 0 to +31
- -16 to +15
- -15 to +16

Unsigned Integers

- 0000 0000 0000 0000 0000 0000 0000 0000
- 0000 0000 0000 0000 0000 0000 0000 0100 = 8
- 0000 0000 0000 0000 0000 0000 0000 0101 = 16
- 0000 0000 0000 0000 0000 0000 0000 0110 = 32
- 0000 0000 0000 0000 0000 0000 0000 0111 = 40
- 0000 0000 0000 0000 0000 0000 0000 1000 = 48
- 0000 0000 0000 0000 0000 0000 0000 1001 = 56
- 0000 0000 0000 0000 0000 0000 0000 1010 = 64
- 0000 0000 0000 0000 0000 0000 0000 1011 = 72
- 0000 0000 0000 0000 0000 0000 0000 1100 = 80
- 0000 0000 0000 0000 0000 0000 0000 1101 = 88
- 0000 0000 0000 0000 0000 0000 0000 1110 = 96
- 0000 0000 0000 0000 0000 0000 0000 1111 = 104
- 0000 0000 0000 0000 0000 0000 0001 0000 = 16
- 0000 0000 0000 0000 0000 0000 0001 0001 = 17
- 0000 0000 0000 0000 0000 0000 0001 0010 = 18
- 0000 0000 0000 0000 0000 0000 0001 0011 = 19
- 0000 0000 0000 0000 0000 0000 0001 0100 = 20
- 0000 0000 0000 0000 0000 0000 0001 0101 = 21
- 0000 0000 0000 0000 0000 0000 0001 0110 = 22
- 0000 0000 0000 0000 0000 0000 0001 0111 = 23
- 0000 0000 0000 0000 0000 0000 0001 1000 = 24
- 0000 0000 0000 0000 0000 0000 0001 1001 = 25
- 0000 0000 0000 0000 0000 0000 0001 1010 = 26
- 0000 0000 0000 0000 0000 0000 0001 1011 = 27
- 0000 0000 0000 0000 0000 0000 0001 1100 = 28
- 0000 0000 0000 0000 0000 0000 0001 1101 = 29
- 0000 0000 0000 0000 0000 0000 0001 1110 = 30
- 0000 0000 0000 0000 0000 0000 0001 1111 = 31

MIPS Logical Instructions

- Useful to operate on fields of bits within a word — e.g., characters within a word (8 bits)
- Operations to pack / unpack bits into words
- Called logical operators

<table>
<thead>
<tr>
<th>Logical</th>
<th>C operators</th>
<th>Java operators</th>
<th>MIPS instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit-by-bit</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and</td>
</tr>
<tr>
<td>OR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOT (~)</td>
<td>~</td>
<td>~</td>
<td>nor</td>
</tr>
<tr>
<td>Shift left</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td>sll</td>
</tr>
<tr>
<td>Shift right</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;</td>
<td>srl</td>
</tr>
</tbody>
</table>
Bit-by-bit Definition

<table>
<thead>
<tr>
<th>Operation</th>
<th>Input 1</th>
<th>Input 2</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AND</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>AND</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AND</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>OR</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OR</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>OR</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>OR</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>NOR</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>NOR</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>NOR</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NOR</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Examples

- If register $t2$ contains and $0000000000000000110111000000_{t\text{wo}}$
- Register $t1$ contains $0000000000000000001111000000_{t\text{wo}}$
- What is value of $t0$ after:
  or $t0,t1,t2$ # reg $t0 = \text{reg } t1 \lor \text{reg } t2$

Examples

- If register $t2$ contains and $0000000000000000110111000000_{t\text{wo}}$
- Register $t1$ contains $0000000000000000001111000000_{t\text{wo}}$
- What is value of $t0$ after:
  or $t0,t1,t2$ # reg $t0 = \text{reg } t1 \lor \text{reg } t2$

Examples

- If register $t2$ contains and $0000000000000000110111000000_{t\text{wo}}$
- Register $t1$ contains $0000000000000000001111000000_{t\text{wo}}$
- What is value of $t0$ after:
  or $t0,t1,t2$ # reg $t0 = \text{reg } t1 \lor \text{reg } t2$

Shifting

- Shift left logical moves n bits to the left (insert 0s into empty bits)
  - Same as multiplying by $2^n$ for two's complement number
- For example, if register $s0$ contained $00000000000000001001_{t\text{wo}} = 9_{t\text{en}}$
- If executed sll $s0$, $s0$, 4, result is:
  $00000000000000000000000000000000_{t\text{wo}} = 144_{t\text{en}}$
- And $9_{t\text{en}} \times 2^n = 9_{t\text{en}} \times 16_{t\text{en}} = 144_{t\text{en}}$
- Shift right logical moves n bits to the right (insert 0s into empty bits)
  - NOT same as dividing by $2^n$ (negative numbers fail)
Shifting

- Shift right arithmetic moves n bits to the right (insert high order sign bit into empty bits)
- For example, if register $s0$ contained 1111 1111 1111 1111 1111 1111 1110 0111two = -25ten
- If executed sra $s0$, $s0$, 4, result is:

Impact of Signed and Unsigned Integers on Instruction Sets

- What (if any) instructions affected?
  - Load word, store word?
  - Branch equal, branch not equal?
  - And, or, srl, srl?
  - Add, sub, mult, div?
  - slti (set less than immediate)?

C provides two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||) while MIPS doesn't. Why?

- Logical operations AND and OR do & and |
- While conditional branches do && and ||
- The previous statement has it backwards: && and || logical ops, & and | are branches
- They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C

Administrivia

- Lab #4 posted
- Project #1 Due Sunday @ 11:59:59
- This week should be easier
- Midterm is now on the horizon:
  - No discussion during exam week
  - TA Review: Su, Mar 4, starting 2 PM, 2050 VLSB
  - Exam: Tu, Mar 6, 6:40-9:40 PM, 2050 VLSB (room change)
  - Small number of special consideration cases, due to class conflicts, etc. - contact me

Getting to Know Profs: My Old Friends

- 2 to 3 times/year spend weekend with old friends I went to high school (South Torrance) with to play poker, watch Superbowl, go bodysurfing, talk about life

Old friends even more valuable as you age

Agenda

- Everything is a Number
- Administrivia
- Overflow and Real Numbers
- Instructions as Numbers
- Technology Break
- Assembly Language to Machine Language
- Summary
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
  - NaN, Not-A-Number (NaN), exponent overflow, exponent underflow
- Keep encoding that is somewhat compatible with two’s complement
  - E.g., 0 in FP. 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
  - \(-61 \times 10^{2} \) is 6.10 \times 10^{-5}

- For addition/subtraction, you first must align:
  - \( (1.999 \times 10^{5}) + (5.5 \times 10^{3}) \)
  - \( (1.999 \times 10^{5}) + (5.5 \times 10^{3}) = 5.6999 \times 10^{3} \)

- For very small numbers (e.g., Base 10)
  - \( 0.000061 \) is \( 6.10 \times 10^{-5} \)

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 00000)

- Bias notation: subtract bias from exponent
  - Single precision uses bias of 127; DP uses 1023

- 0 uses 000 00000 \( \Rightarrow 0 - 127 = -127 \); \( \infty \) uses 1111 11111 \( \Rightarrow 255 - 127 = 128 \)

  - Smallest SP real can represent: 1.00...00 \times 2^{-126}
  - Largest SP real can represent: 1.11...11 \times 2^{127}

Bias Notation (+127)

<table>
<thead>
<tr>
<th>How it is interpreted</th>
<th>How it is encoded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal Exponent</td>
<td>sign</td>
</tr>
<tr>
<td>( \Rightarrow ) NaN</td>
<td>1</td>
</tr>
<tr>
<td>Getting closer to zero</td>
<td></td>
</tr>
<tr>
<td>( 2 )</td>
<td>0</td>
</tr>
<tr>
<td>( 1 )</td>
<td>0</td>
</tr>
<tr>
<td>( 0 )</td>
<td>0</td>
</tr>
<tr>
<td>( -1 )</td>
<td>1</td>
</tr>
<tr>
<td>( -2 )</td>
<td>1</td>
</tr>
<tr>
<td>( \ldots )</td>
<td></td>
</tr>
<tr>
<td>Zero</td>
<td></td>
</tr>
<tr>
<td>For Denorms</td>
<td>10000000</td>
</tr>
</tbody>
</table>

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

- 0 vs. \( 1 \times 10^{-127} \)?
- \( 1 \times 10^{-126} \) vs. \( 1 \times 10^{-127} \)?
- \(-1 \times 10^{-127} \) vs. 0?
- \(-1 \times 10^{-126} \) vs. \(-1 \times 10^{-127} \)?
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 $2^{32}$)
  
  $1 + 4,294,967,295 = \text{Overflow}$

Depends on the Programming Language

- C signed number arithmetic also ignores overflow
  
  $1 + 2,147,483,647 = \text{Overflow}$

MIPS Solution: Offer Both

- Instructions that can trigger overflow:
  - `add`, `sub`, `mul`, `div`, `addi`, `mult`, `divi`
- Instructions that don’t overflow are called “unsigned” (really means “no overflow”):
  - `addu`, `subu`, `mulu`, `divu`, `addiu`, `multi`, `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?

- $r \times E^i$, $E$ where is exponent (2), $i$ is a positive or negative integer, $r$ 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 $2^{32}$ 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 $2.0 \times 10^{38}$

More Floating Point

- 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 $B$
  - $A = 0.000000, B = 0.000001$
- Floating Point Addition is not Associative!
  - Integer addition is associative
- When does this matter?

MIPS Floating Point Instructions

- C, Java has single precision ($\text{float}$) and double precision ($\text{double}$) types
- MIPS instructions: .s for single, .d for double
  - Fl. Pt. Addition single precision: add.s
  - Fl. Pt. Addition 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: \texttt{s} for single, \texttt{d} for double
  - Fp. Pt. Comparison single precision:
- Since rarely mix integers and Floating Point, 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
  - \texttt{lwc1} (load word), \texttt{swc1} (store word)
  - Double precision uses two \texttt{lwc1} instructions, two \texttt{swc1} 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} * \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

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

Key Concepts

- MIPS ISA guided by four RISC design principles:
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises

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
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

Groups of bits call fields (unused field default is 0)

- Layout called instruction format
- Binary version called machine instruction

<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>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
</tr>
</tbody>
</table>

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>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<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`: 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

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

<table>
<thead>
<tr>
<th>Field</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</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>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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

Encoding of MIPS Instructions: Must Be Unique!

<table>
<thead>
<tr>
<th>Instruction</th>
<th>For</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>32 prosecute n.a.</td>
<td></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>32 prosecute n.a.</td>
<td></td>
</tr>
<tr>
<td>sli</td>
<td>R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>43 prosecute n.a.</td>
<td></td>
</tr>
<tr>
<td>addi unsigned</td>
<td>I</td>
<td>rs</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>constant</td>
<td>n.a.</td>
<td></td>
</tr>
<tr>
<td>lw (load word)</td>
<td>I</td>
<td>35 prosecute</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>address</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>lw (store word)</td>
<td>I</td>
<td>43 prosecute</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>address</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>I</td>
<td>4</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td>I</td>
<td>5</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>j (jump)</td>
<td>I</td>
<td>2</td>
<td>rt</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>jal</td>
<td>I</td>
<td>3</td>
<td>rt</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>jr (jump reg)</td>
<td>I</td>
<td>rs</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>8 prosecute n.a.</td>
<td></td>
</tr>
</tbody>
</table>

Converting C to MIPS Machine code

\[ A[300] = h + A[300]; \]
\[ \text{lw } s0, 1200(s1); \]
\[ \text{addu } s1, s2, s0; \]
\[ \text{sw } s0, 1200(s1); \]

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

<table>
<thead>
<tr>
<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>R-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Agenda

- Everything is a Number
- Administration
- Overflow and Real Numbers
- Instructions as Numbers
- Assembly Language to Machine Language
- Summary

Converting C to MIPS Machine code

\[ A[300] = h + A[300]; \]
\[ \text{lw } s0, 1200(s1); \]
\[ \text{addu } s1, s2, s0; \]
\[ \text{sw } s0, 1200(s1); \]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>For</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>32 prosecute n.a.</td>
<td></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>32 prosecute n.a.</td>
<td></td>
</tr>
<tr>
<td>sli</td>
<td>R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>43 prosecute n.a.</td>
<td></td>
</tr>
<tr>
<td>addi unsigned</td>
<td>I</td>
<td>rs</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>constant</td>
<td>n.a.</td>
<td></td>
</tr>
<tr>
<td>lw (load word)</td>
<td>I</td>
<td>35 prosecute</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>address</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>lw (store word)</td>
<td>I</td>
<td>43 prosecute</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>address</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>I</td>
<td>4</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td>I</td>
<td>5</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>j (jump)</td>
<td>I</td>
<td>2</td>
<td>rt</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>jal</td>
<td>I</td>
<td>3</td>
<td>rt</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>jr (jump reg)</td>
<td>I</td>
<td>rs</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>8 prosecute n.a.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>For</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>32 prosecute n.a.</td>
<td></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>32 prosecute n.a.</td>
<td></td>
</tr>
<tr>
<td>sli</td>
<td>R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>43 prosecute n.a.</td>
<td></td>
</tr>
<tr>
<td>addi unsigned</td>
<td>I</td>
<td>rs</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>constant</td>
<td>n.a.</td>
<td></td>
</tr>
<tr>
<td>lw (load word)</td>
<td>I</td>
<td>35 prosecute</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>address</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>lw (store word)</td>
<td>I</td>
<td>43 prosecute</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>address</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>I</td>
<td>4</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td>I</td>
<td>5</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>j (jump)</td>
<td>I</td>
<td>2</td>
<td>rt</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>jal</td>
<td>I</td>
<td>3</td>
<td>rt</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>jr (jump reg)</td>
<td>I</td>
<td>rs</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>8 prosecute n.a.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>For</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>32 prosecute n.a.</td>
<td></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>32 prosecute n.a.</td>
<td></td>
</tr>
<tr>
<td>sli</td>
<td>R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>43 prosecute n.a.</td>
<td></td>
</tr>
<tr>
<td>addi unsigned</td>
<td>I</td>
<td>rs</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>constant</td>
<td>n.a.</td>
<td></td>
</tr>
<tr>
<td>lw (load word)</td>
<td>I</td>
<td>35 prosecute</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>address</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>lw (store word)</td>
<td>I</td>
<td>43 prosecute</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>address</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>I</td>
<td>4</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td>I</td>
<td>5</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>j (jump)</td>
<td>I</td>
<td>2</td>
<td>rt</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>jal</td>
<td>I</td>
<td>3</td>
<td>rt</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>jr (jump reg)</td>
<td>I</td>
<td>rs</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>8 prosecute n.a.</td>
<td></td>
</tr>
</tbody>
</table>
Addressing in Braches

Programs much bigger than $2^{16}$ bytes, but branch address must fit in 16-bit field
- Must specify a register for branch addresses for big programs: PC + Branch address
- Which register?

Conditional branching for IF-statement, loops
- Tend to be near branches; $\frac{1}{2}$ within 16 instructions
- Idea: PC-relative branching

Addressing in Jumps

Same trick for Jumps, Jump and Link
PC = Jump address * 4
Since PC = 32 bits, and Jump address * 4 = 28 bits, what about other 4 bits?
Jump and Jump and Link only changes bottom 28 bits of PC

32 bit Constants in MIPS

- Can create a 32-bit constant from two 32-bit MIPS instructions
- Load Upper Immediate ($lui$ or “Louie”) puts 16 bits into upper 16 bits of destination register
- MIPS to load 32-bit constant into register $s0$:
  \[
  \text{0000 0000 0011 1101 0001 0000 0000}_{\text{two}}
  \]
  $lui$ $s0$, 61 # 61 = 0000 0000 0011 1101$_{\text{two}}$
- ori $s0$, $s0$, 2304 # 2304 = 0000 1001 0000 0000$_{\text{two}}$

Converting to MIPS Machine code

And in Conclusion, …

- 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 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
- Everything is a (binary) number in a computer
  - Instructions and data; stored program concept