Agenda

• Everything is a Number
• Administrivia
• Overflow and Real Numbers
• Instructions as Numbers
• Technology Break
• 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

Levels of Representation/Interpretation

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

Compiler

Assembly Language Program (e.g., MIPS)

Machine Language Program (MIPS)

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

Logic Circuit Description (Circuit Schematic Diagrams)

temp = v[k];
v[k+1] = v[k] + temp;
Anything can be represented as a number,
1010 1111 0101 1000
1010 1111 0101 1000
1010 1111 0101 1000
1010 1111 0101 1000

Agenda

• Everything is a Number
• Administrivia
• Overflow and Real Numbers
• Instructions as Numbers
• Technology Break
• 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

Agenda

• Everything is a Number
• Administrivia
• Overflow and Real Numbers
• Instructions as Numbers
• Technology Break
• 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
Number Representation

- Value of i-th digit is \( d \times \text{Base}^i \) where \( i \) starts at 0 and increases from right to left:
  \[
  123_{10} = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 = 1 \times 100 + 2 \times 10 + 3 \\
  = 100_{10} + 20_{10} + 3_{10} = 123_{10}
  \]
- Binary (Base 2), Hexadecimal (Base 16), Decimal (Base 10) different ways to represent an integer
  - We use \( 1_{10}, 5_{16}, 10_{10} \), to be clearer
  (vs. \( 1_2, 4_5, 5_{10} \))

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 \( \frac{1}{2} \) numbers <0, want \( \frac{3}{2} \) 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\(^{31} - 1\) (2,147,483,647)
  - 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

Hexadecimal digits:

- \( 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F \)
- \( \text{FFFF}_{16} = 15_{10} \times 16^{15} + 15_{10} \times 16^{14} + \ldots + 15_{10} \times 16^0 \\
  = 3840_{10} + 240_{10} + 15_{10} = 4095_{10} \)
- 1111 1111 1111 two = \text{FFFF}_{16} = 4095_{10}
- May put blanks every group of binary, octal, or hexadecimal digits to make it easier to parse, like commas in decimal

Unsigned Integers

- \( 0000 \ldots 0000 \times 10^0 \) = 0
- \( 0000 \ldots 0001 \times 10^0 \) = 1
- \( 1000 \ldots 1000 \times 10^0 \) = \( -2^{31} \)
- \( 1111 \ldots 1110 \times 10^0 \), \( \ldots \)
Peer Instruction Question
• Suppose we had a 5 bit word. What integers can be represented in two’s complement?
  Red: -32 to +31
  Orange: -31 to +32
  Green: 0 to +31
  Yellow: -16 to +15
  Pink: -15 to +15
  Blue: -15 to +16

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 operations

<table>
<thead>
<tr>
<th>Logical operations</th>
<th>C operators</th>
<th>Java operators</th>
<th>MIPS instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit-by-bit AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and</td>
</tr>
<tr>
<td>Bit-by-bit OR</td>
<td></td>
<td></td>
<td>or</td>
</tr>
<tr>
<td>Bit-by-bit 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>

Examples
• If register $t2$ contains and
  0000 0000 0000 0000 0000 1101 1100 0000<sub>two</sub>
• Register $t1$ contains
  0000 0000 0000 0000 0011 1100 0000 0000<sub>two</sub>
• What is value of $t0$ after:
  and $t0,$t1,$t2 # reg $t0 = reg $t1 & reg $t2

Peer Instruction Answer
• Suppose we had a 5 bit word. What integers can be represented in two’s complement?
  Red: -32 to +31
  Orange: -31 to +32
  Green: 0 to +31
  Yellow: -16 to +15
  Pink: -15 to +15
  Blue: -15 to +16

Bit-by-bit Definition

<table>
<thead>
<tr>
<th>Operation</th>
<th>Input</th>
<th>Input</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
  0000 0000 0000 0000 0000 1101 1100 0000<sub>two</sub>
• Register $t1$ contains
  0000 0000 0000 0000 0011 1100 0000 0000<sub>two</sub>
• What is value of $t0$ after:
  and $t0,$t1,$t2 # reg $t0 = reg $t1 & reg $t2
  0000 0000 0000 0000 0000 1100 0000 0000<sub>two</sub>
Examples

• If register $t2$ contains and
  0000 0000 0000 0000 0000 1101 1100 0000 ${\text{two}}$
• Register $t1$ contains
  0000 0000 0000 0000 0011 1100 0000 0000 ${\text{two}}$
• What is value of $t0$ after:
or $t0,t1,t2$ # reg $t0$ = reg $t1$ | reg $t2$

Examples

• If register $t2$ contains and
  0000 0000 0000 0000 0000 1101 1100 0000 ${\text{two}}$
• Register $t1$ contains
  0000 0000 0000 0000 0011 1100 0000 0000 ${\text{two}}$
• What is value of $t0$ after:
or $t0,t1,t2$ # reg $t0$ = ~ (reg $t1$ | 0)

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
  0000 0000 0000 0000 0000 1001 1001 0000
  If executed sll $s0$, $s0$, 4, result is:
  0000 0000 0000 0000 0000 1001 1001 1001
  And $9_0 = 2^4 = 9_{10} \times 16_{10} = 144_{10}$
• 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 $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 25_{\text{ten}}$
- If executed sra $s0, s0, 4$, result is: $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 3_{\text{ten}}$

- Shift right arithmetic moves n bits to the right (insert high order sign bit into empty bits)
- For example, if register $s0$ contained $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} = -25_{\text{ten}}$
- 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, slt, srl?
- add, sub, mult, div?
- slt (set less than immediate)?

Peer Instruction Question

- C provides two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||) while MIPS doesn’t. Why?
- Red: Logical operations AND and OR implement & and | while conditional branches implement && and ||
- Orange: The previous statement has it backwards: && and || correspond to logical operations while & and | map to conditional branches
- Green: They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C

Peer Instruction Answer

- C provides two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||) while MIPS doesn’t. Why?
- Red: Logical operations AND and OR implement & and | while conditional branches implement && and ||
- Reason: e.g., && is logical and: true && true is true and everything else is false
  & is bitwise and: e.g., $(1010_{\text{two}} \& 1000_{\text{two}}) = 1000_{\text{two}}$
Agenda

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

Administrivia

- Lab #4 posted (?)
- Project #1 Due Sunday @ 11:59:59
- Good news! No HW this week!
- Want in from the Wait List?
  - Sign up for Lab 019 (Friday, 7-9 PM – there is space available!)
- Midterm is now on the horizon:
  - No discussion during exam week
  - TA Review: Su, Mar 6, 2-5 PM, 2050 VLSB
  - Exam: Tu, Mar 8, 6-9 PM, 145/155 Dwinelle
- Small number of special consideration cases, due to class conflicts, etc.—contact Dave or Randy

CS61c in the News

Ken Olsen, Who Built DEC Into a Power, Dies at 84

Ken Olsen, who helped launch the computer industry as a founder of the Digital Equipment Corporation, at one time the world’s second-largest computer company, died on Sunday. He was 84.

Mr. Olsen, who was proclaimed “America’s most successful entrepreneur” by Fortune magazine in 1968, built Digital Equipment to large

Don’t Go Driving with Randy!

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, 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 \( 6.10 \times 10^{-5} \)

- \( r \times E \), \( E \) where is exponent, \( i \) is a positive or negative integer, \( r \) is a real number \( \geq 1.0, < 10 \)

---

Which is Smaller?
(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. \(-1 \times 10^{-127} \)?

---

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_steps)

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

- 0 uses 0000 0000_steps \( \Rightarrow \) 0-127 = -127
  - \( \infty \), NaN uses 1111 1111_steps \( \Rightarrow \) 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</td>
<td>signed 32's complement</td>
</tr>
<tr>
<td>111111</td>
<td>0000000</td>
</tr>
<tr>
<td>111111</td>
<td>0000000</td>
</tr>
<tr>
<td>111110</td>
<td>0000000</td>
</tr>
<tr>
<td>111101</td>
<td>0000000</td>
</tr>
<tr>
<td>111011</td>
<td>0000000</td>
</tr>
<tr>
<td>110101</td>
<td>0000000</td>
</tr>
<tr>
<td>100000</td>
<td>0000000</td>
</tr>
<tr>
<td>000000</td>
<td>0000000</td>
</tr>
</tbody>
</table>

Getting closer to zero

- For infinities
- 111111
- 111110
- 111101
- 111011
- 110101
- 110100
- 110011
- 110010
- 110001
- 110000
- 101111
- 101110
- 101101
- 101100
- 101011
- 101010
- 101001
- 101000
- 100111
- 100110
- 100101
- 100100
- 100011
- 100010
- 100001
- 100000
- 011111
- 011110
- 011101
- 011100
- 011011
- 011010
- 011001
- 011000
- 010111
- 010110
- 010101
- 010100
- 010011
- 010010
- 010001
- 010000
- 001111
- 001110
- 001101
- 001100
- 001011
- 001010
- 001001
- 001000
- 000111
- 000110
- 000101
- 000100
- 000011
- 000010
- 000001
- 000000
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
  $1 + 2,147,483,647 = 2^{31} - 1$

• 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”):
  – adds, 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?

• $r \times E^i$, $E$ 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 $2^{23}$ 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 ($F$)
  
  \[-1 \times (1 + F) \times 2^E\]
  
- Can represent from $2.0 \times 10^{-38}$ to $2.0 \times 10^{38}$

More Floating Point

- What about 0?
  - Bit pattern all 0s means 0, so no implicit leading 1
- What if divide 1 by 0?
  - Can get infinity symbols $\pm \infty$,
  - Sign bit 0 or 1, largest exponent, 0 in fraction
- What if do something stupid? ($\infty - \infty$, $0 \times 0$)
  - 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? ($2 \times 10^{38} \times 2 \times 10^6$)
  - Get overflow in exponent, alert programmer!
- What if result is too small? ($2 \times 10^{-38} \times 2 \times 10^{-6}$)
  - 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. 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

- 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 ($F$)
  
  \[-1 \times (1 + F) \times 2^E\]
  
- Can represent from $2.0 \times 10^{-308}$ to $2.0 \times 10^{308}$
- 32 bit format called Single Precision

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:
  - Fl. Pt. Comparison double precision:
  - 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:
  - $f0$ and $f1, f2$ and $f3, f4$ and $f5, \ldots, f30$ and $f31$
- Need data transfer instructions for these new registers
  - lwc1 (load word), swc1 (store word)
  - Double precision uses two lwc1 instructions, two swc1 instructions
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?

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

---

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

---

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 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

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

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

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?

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

<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>

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

- **R-type**: op | rs | rt | rd | shamt | funct
- **I-type**: op | rs | rt | address or constant
- **J-type**: op | address

- 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?

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_naa</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_naa</td>
</tr>
<tr>
<td>slt</td>
<td>R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>43_naa</td>
</tr>
<tr>
<td>sll</td>
<td>R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>reg</td>
<td>const</td>
</tr>
<tr>
<td>addi unsigned</td>
<td>I</td>
<td>32</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>lw (load word)</td>
<td>I</td>
<td>35</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>sw (store word)</td>
<td>I</td>
<td>43</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>beq</td>
<td>I</td>
<td>4</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>bne</td>
<td>I</td>
<td>5</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>j (jump)</td>
<td>I</td>
<td>2</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>jal</td>
<td>I</td>
<td>3</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</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>33_naa</td>
</tr>
</tbody>
</table>

Agenda

- Everything is a Number
- Administrivia
- Overflow and Real Numbers
- Instructions as Numbers
- Technology Break
- Assembly Language to Machine Language
- Summary
Converting C to MIPS Machine code
\&A = $t0 (reg 8), $t1 (reg 9), h = $s2 (reg 18)
A[300] = h + A[300];

Instruction | op | rs | rt | rd | shamt | funct | address
--- | --- | --- | --- | --- | --- | --- | ---
lw | 1 | 35 | n.a | n.a | n.a | n.a | address
addu | 0 | 0 | n.a | n.a | n.a | n.a | address
sw | 1 | 43 | n.a | n.a | n.a | n.a | address

Format?
lw $t0,1200($t1)
addu $t1,$s2,$t0
sw $t0,1200($t1)

Addressing in Branches
\textbf{i-type} | \textbf{op} | \textbf{rs} | \textbf{rt} | \textbf{address or constant}
--- | --- | --- | --- | ---

6 bits | 5 bits | 5 bits | 16 bits

- 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 = Register + Branch address
  - Which register?
- Conditional branching for IF-statement, loops
  - Tend to be near branches; ½ within 16 instructions
- Idea: \textit{PC-relative branching}

Addressing in Jumps
\textbf{j-type} | \textbf{op} | \textbf{address}
--- | --- | ---

6 bits | 26 bits

- 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

Converting to MIPS Machine code
Add Loop: Format?
800 sll $t1,$s3,2
804 addu $t1,$s1,$s6
808 lw $t0,0($t1)
812 bne $t0,$s5, Exit
816 addiu $s3,$s3,1
820 j Loop
Exit:
R-type | \textbf{op} | \textbf{rs} | \textbf{rt} | \textbf{rd} | \textbf{shamt} | \textbf{funct} | \textbf{address}
--- | --- | --- | --- | --- | --- | --- | ---
I-type | \textbf{op} | \textbf{rs} | \textbf{rt} | \textbf{address or constant}
--- | --- | --- | ---
J-type | \textbf{op} | \textbf{address}
---
Converting to MIPS Machine code

### Loop:
```
sll $t1,$s3,2
addu $t1,$t1,$s6
lw $t0,0($t1)
bne $t0,$s5, Exit
addiu $s3,$s3,1
j Loop
```

### Exit:
```
```

---

### 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:
  
  \[
  \begin{array}{c}
  \text{lui} \ $s0, \ 61 \\
  \text{ori} \ $s0, \ $s0, \ 2304
  \end{array}
  \]
  
  \[
  \begin{array}{c}
  \text{two's complement}
  \end{array}
  \]

```

---

### Assembly and Pseudo-instructions
- Turning textual MIPS instructions into machine code called *assembly*, program called *assembler*
  - Calculates addresses, maps register names to numbers, produces binary machine language
  - Textual language called *assembly language*
- Can also accept instructions convenient for programmer but not in hardware
  - *Load immediate* (li) allows 32-bit constants, assembler turns into lui + ori (if needed)
  - *Load double* (ld) uses two lwc1 instructions to load a pair of 32-bit floating point registers
  - *Called Pseudo-Instructions*

---

### “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
- Assemblers can enhance machine instruction set to help assembly-language programmer