Big Idea #1: Levels of Representation/Interpretation

High Level Language Program (e.g., C)  temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;

Assembly Language Program (e.g., MIPS)  lw  $t0, 0($2)
lw  $t1, 4($2)
sw  $t1, 0($2)
sw  $t0, 4($2)

Machine Language Program (MIPS) 0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111

Compiler

Assembler

Machine Interpretation

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

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

Register File

ALU

Logic Circuit

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

• Signed vs. Unsigned Numbers
• Instructions as Numbers
• Administrivia
• Instructions as Numbers
• Technology Break
• Floating Point Numbers
• And in Conclusion, ...

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 $2^{32}$)
  \[ 1 + 4,294,967,295 = \]

\[ \text{FFFFFFFF}_{\text{hex}} + 1 = 0 \]
Depends on the Programming Language

• C signed number arithmetic also ignores overflow

\[ 1 + 2,147,483,647 \times (2^{31}-1) = \]

\[ 1 + 7FFFFFFF_{\text{hex}} = 80000000_{\text{hex}} = -2,147,483,648 \]
Depends on the Programming Language

- 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?
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?
    - -1 < 0 signed, but FFFF_{hex} > 0 unsigned!

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, sll, 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

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
Agenda

• Signed vs. Unsigned Numbers
• Instructions as Numbers
• Administrivia
• Instructions as Numbers Continued
• Technology Break
• Floating Point Numbers
• And in Conclusion, ...

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
  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?

RISC ISA Design Principles

#1: Simplicity Favors Regularity
  Small number of formats, fixed fields
#2: Smaller is Faster
  Few instructions, basic primitives
#3: Good Design Demands Good Compromises
  Balance usefulness and simplicity
#4: *Make the Common Case Fast*
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

<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

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>000000</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 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

• \texttt{sll \$zero,\$zero,0}
  
  – \$zero is register 0
  
  – Shift amount 0 is 0
  
  – Shift left logical instruction encoded as number 0

\begin{center}
\begin{tabular}{cccccccc}
 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
00000 & 00000 & 00000 & 00000 & 00000 & 00000 \\
6 bits & 5 bits & 5 bits & 5 bits & 5 bits & 6 bits \\
\end{tabular}
\end{center}

• Can also represent machine code as base 16 or base 8 number: 0000 0000_\text{hex}, 00000000000_\text{oct}

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 (justify multiple instruction formats)?

• 4\textsuperscript{th} Design Principle: \textit{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**: 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**

<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>

**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>

- 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>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&lt;sub&gt;ten&lt;/sub&gt;</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&lt;sub&gt;ten&lt;/sub&gt;</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&lt;sub&gt;ten&lt;/sub&gt;</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&lt;sub&gt;ten&lt;/sub&gt;</td>
<td>n.a.</td>
</tr>
<tr>
<td>addi unsigned</td>
<td>I</td>
<td>9&lt;sub&gt;ten&lt;/sub&gt;</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&lt;sub&gt;ten&lt;/sub&gt;</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&lt;sub&gt;ten&lt;/sub&gt;</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&lt;sub&gt;ten&lt;/sub&gt;</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>n.a.</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td>I</td>
<td>5&lt;sub&gt;ten&lt;/sub&gt;</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&lt;sub&gt;ten&lt;/sub&gt;</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&lt;sub&gt;ten&lt;/sub&gt;</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&lt;sub&gt;ten&lt;/sub&gt;</td>
<td>n.a.</td>
</tr>
</tbody>
</table>

Converting C to MIPS Machine code
$t0$ (reg 8), &A in $t1$ (reg 9), h=$s2$ (reg 18)


lw $t0,1200($t1)
addu $t0,$s2,$t0
sw $t0,1200($t1)
Agenda

• Signed vs. Unsigned Numbers
• Instructions as Numbers
• Administrivia
• Instructions as Numbers (continued)
• Technology Break
• Floating Point Numbers
• And in Conclusion, ...

Administrivia

• Tuesday discussion sections henceforth cancelled:
  – Sagar: Tu 9-10 → Friday, 8-9, 405 Soda
  – Sung Roa: Tu 10-11 → Friday, 1-2, 320 Soda
• Piazza Etiquette
  – Please don’t post your code
  – Please don’t post copyrighted material
  – Remember do look before you post
Agenda

• Signed vs. Unsigned Numbers
• Instructions as Numbers
• Administrivia
• Instructions as Numbers (cont.)
• Technology Break
• Floating Point Numbers
• And in Conclusion, ...

BlackBerry Share of Sales

Steep Decline

Worldwide smartphone share of sales

Source: Gartner

9/24/13
Addressing in Branches

### l-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>

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

---

Addressing in Branches

### l-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>

- Hardware increments PC early, so relative address is \(PC = (PC + 4) + \text{Branch address}\)
- Another optimization since all MIPS instructions 4 bytes long?
- Multiply value in branch address field by 4!
- MIPS PC-relative branching
  \(PC = (PC + 4) + (\text{Branch address} \times 4)\)
Addressing in Jumps

<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>

- Same trick for Jumps, Jump and Link
  \[ \text{PC} = \text{Jump address} \times 4 \]
- Since PC = 32 bits, and Jump address $\times 4 = 28$ bits, what about other 4 bits?
- Jump and Jump and Link only changes bottom 28 bits of PC

Converting to MIPS Machine code

**Loop:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Format?</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>sll $t1, s3, 2</td>
</tr>
<tr>
<td>804</td>
<td>addu $t1, $t1, $s6</td>
</tr>
<tr>
<td>808</td>
<td>lw $t0, 0($t1)</td>
</tr>
<tr>
<td>812</td>
<td>bne $t0, $s5, Exit</td>
</tr>
<tr>
<td>816</td>
<td>addiu $s3, $s3, 1</td>
</tr>
<tr>
<td>820</td>
<td>j Loop</td>
</tr>
</tbody>
</table>

**Exit:**

<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>op</td>
<td>rs</td>
<td>rt</td>
<td>address or constant</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>op</td>
<td></td>
<td></td>
<td>address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

9/22/13  Fall 2013 – Lecture #8  36

9/24/13  Fall 2013 – Lecture #8  37
32 bit Constants in MIPS

- Can create a 32-bit constant from two 32-bit MIPS instructions
- \textit{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{align*}
  \text{lui } & s0, 61 \# 61 = 0000 \ 0000 \ 0011 \ 1101 \text{two} \\
  \text{ori } & s0, s0, 2304 \# 2304 = 0000 \ 1001 \ 0000 \ 0000 \text{two}
  \end{align*}

Agenda

- Signed vs. Unsigned Numbers
- Instructions as Numbers
- Administrivia
- Instructions as Numbers (continued)
- Technology Break
- 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
  – $+\infty$, $-\infty$, 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):
  – $r \times E^i$, $E$ is 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 $6.10 \times 10^{-5}$
Scientific Notation (e.g., Base 10)

- \((r \times e^i) \times (s \times e^j) = (r \times s) \times e^{i+j}\)
  
  \((1.999 \times 10^2) \times (5.5 \times 10^3) = (1.999 \times 5.5) \times 10^5\)
  
  = \(10.9945 \times 10^5\)
  
  = \(1.09945 \times 10^6\)

- \((r \times e^i) / (s \times e^j) = (r / s) \times e^{i-j}\)
  
  \((1.999 \times 10^2) / (5.5 \times 10^3) = 0.3634545... \times 10^{-1}\)
  
  = \(3.634545... \times 10^{-2}\)

- For addition/subtraction, you first must align:
  
  \((1.999 \times 10^2) + (5.5 \times 10^3)\)
  
  = \((.1999 \times 10^3) + (5.5 \times 10^3)\) = \(5.6999 \times 10^3\)

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

- 0 vs. \(1 \times 10^{-127}\)?

- \(1 \times 10^{-126}\) 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_{two})
- **Bias notation**: subtract bias from exponent
  - Single precision uses bias of 127; DP uses 1023
- 0 uses 0000 0000_{two} ⇒ 0-127 = -127;
  ∞, NaN uses 1111 1111_{two} ⇒ 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 encoded</th>
<th>How it is encoded</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Decimal</strong></td>
<td><strong>Biased Notation</strong></td>
</tr>
<tr>
<td>Exponent</td>
<td>signed 2's</td>
</tr>
<tr>
<td></td>
<td>complement</td>
</tr>
<tr>
<td><strong>∞, NaN</strong></td>
<td>11111111</td>
</tr>
<tr>
<td>Getting closer to zero</td>
<td>11111110</td>
</tr>
<tr>
<td><strong>Zero</strong></td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>01111111</td>
</tr>
<tr>
<td></td>
<td>00000001</td>
</tr>
<tr>
<td></td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>01111111</td>
</tr>
<tr>
<td></td>
<td>01111110</td>
</tr>
<tr>
<td></td>
<td>00000001</td>
</tr>
<tr>
<td></td>
<td>00000000</td>
</tr>
</tbody>
</table>

9/22/13
Fall 2013 -- Lecture 8
46
What About *Real* Numbers in Base 2?

- $r \times E^i$, $E$ where exponent is (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 $2.0 \times 10^{38}$
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? \((\infty - \infty, 0 ÷ 0)\)
  - 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? \((2 \times 10^{-308} ÷ 2 \times 10^2)\)
  - Get *underflow* in exponent, alert programmer!
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:
    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:
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: $f0$, $f1$, ..., $f31$
  – 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
  – lwcl (load word), swcl (store word)
  – Double precision uses two lwcl instructions, two swcl instructions

9/22/13
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
“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!
- Floating point is an approximation of reals