MIPS R-format Instructions

Instruction fields
- op: operation code (opcode)
- rs: first source register register number
- rt: second source register register number
- rd: destination register register number
- shamt: shift amount (000000 for now)
- funct: function code (extends opcode)
R-format Example

<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>

add $t0$, $s1$, $s2$

<table>
<thead>
<tr>
<th>special</th>
<th>$s1$</th>
<th>$s2$</th>
<th>$t0$</th>
<th>0</th>
<th>add</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>18</td>
<td>8</td>
<td>0</td>
<td>32</td>
</tr>
</tbody>
</table>

000000 10001 10010 01000 00000 100000

$00000010001100100100000000100000_2 = 02324020_{16}$
MIPS I-format Instructions

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

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: $-2^{15}$ to $+2^{15} - 1$
  - Address: offset added to base address in rs

- **Design Principle 4**: Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible
Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

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

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time
Jump Addressing

- Jump (j and jal) targets could be anywhere in text segment
  - Encode full address in instruction

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

- (Pseudo)Direct jump addressing
  - Target address = PC_{31...28} : (address \times 4)
Addressing Mode Summary

1. Immediate addressing
   \[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{Immediate} \]

2. Register addressing
   \[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{rd} \quad \ldots \quad \text{funct} \quad \text{Registers} \]
   \[ \text{Register} \quad \rightarrow \quad \text{Memory} \quad \begin{array}{c} \text{Byte} \\ \text{Halfword} \\ \text{Word} \end{array} \]

3. Base addressing
   \[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{Address} \quad \text{Register} \quad \rightarrow \quad \text{Memory} \quad \text{Word} \]

4. PC-relative addressing
   \[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{Address} \quad \text{PC} \quad \rightarrow \quad \text{Memory} \quad \text{Word} \]

5. Pseudodirect addressing
   \[ \text{op} \quad \text{Address} \quad \text{PC} \quad \rightarrow \quad \text{Memory} \quad \text{Word} \]
Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don’t synchronize
    - Result depends on order of accesses

- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write

- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions
Synchronization in MIPS

- Load linked: `ll rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
  - Returns 1 in `rt`
  - Fails if location is changed
  - Returns 0 in `rt`

- Store conditional: `sc rt, offset(rs)`

Example: atomic swap (to test/set lock variable)
```assembly
try: add $t0,$zero,$s4 ;copy exchange value
    ll $t1,0($s1)    ;load linked
    sc $t0,0($s1)    ;store conditional
    beq $t0,$zero,try ;branch store fails
    add $s4,$zero,$t1 ;put load value in $s4
```
## ARM & MIPS Similarities

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

<table>
<thead>
<tr>
<th></th>
<th>ARM</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date announced</td>
<td>1985</td>
<td>1985</td>
</tr>
<tr>
<td>Instruction size</td>
<td>32 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>Address space</td>
<td>32-bit flat</td>
<td>32-bit flat</td>
</tr>
<tr>
<td>Data alignment</td>
<td>Aligned</td>
<td>Aligned</td>
</tr>
<tr>
<td>Data addressing modes</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Registers</td>
<td>15 × 32-bit</td>
<td>31 × 32-bit</td>
</tr>
<tr>
<td>Input/output</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
</tr>
</tbody>
</table>
Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions
Instruction Encoding

- **Register-register**
  - ARM:
    - 31 28 27 20 19 16 15 12 11 4 3 0
    - \( O_{px}^4 \) \( O_{p}^8 \) \( R_{s1}^4 \) \( R_{d}^4 \) \( O_{px}^6 \) \( R_{s2}^4 \)
  - MIPS:
    - 31 28 27 20 19 16 15 11 10 6 5 0
    - \( O_{p}^6 \) \( R_{s1}^8 \) \( R_{s2}^8 \) \( R_{d}^8 \) \( C_{onst}^8 \) \( O_{px}^4 \)

- **Data transfer**
  - ARM:
    - 31 28 27 20 19 16 15 12 11 0
    - \( O_{px}^4 \) \( O_{p}^8 \) \( R_{s1}^4 \) \( R_{d}^4 \) \( C_{onst12} \)
  - MIPS:
    - 31 28 27 20 19 16 15 0
    - \( O_{p}^6 \) \( R_{s1}^8 \) \( R_{d}^8 \) \( C_{onst16} \)

- **Branch**
  - ARM:
    - 31 28 27 24 28 0
    - \( O_{px}^4 \) \( O_{p}^4 \) \( C_{onst24} \)
  - MIPS:
    - 31 28 27 21 20 16 15 0
    - \( O_{p}^3 \) \( R_{s1}^5 \) \( O_{px}/R_{s2}^3 \) \( C_{onst16} \)

- **Jump/Call**
  - ARM:
    - 31 28 27 24 23 0
    - \( O_{px}^4 \) \( O_{p}^4 \) \( C_{onst24} \)
  - MIPS:
    - 31 26 25 0
    - \( O_{p}^3 \) \( C_{onst26} \)
The Intel x86 ISA

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments
The Intel x86 ISA

Further evolution…

- i486 (1989): pipelined, on-chip caches and FPU
  - Compatible competitors: AMD, Cyrix, …
- Pentium (1993): superscalar, 64-bit datapath
  - Later versions added MMX (Multi-Media eXtension) instructions
  - The infamous FDIV bug
  - New microarchitecture (see Colwell, *The Pentium Chronicles*)
- Pentium III (1999)
  - Added SSE (Streaming SIMD Extensions) and associated registers
- Pentium 4 (2001)
  - New microarchitecture
  - Added SSE2 instructions
The Intel x86 ISA

- And further...
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn’t extend with compatibility, its competitors would!
  - Technical elegance ≠ market success
### x86 Instruction Encoding

- **Variable length encoding**
- **Postfix bytes specify addressing mode**
- **Prefix bytes modify operation**
- Operand length, repetition, locking, …

#### a. JE EIP + displacement

![Instruction Format](image)

- **JE**
- Condition: 4
- Displacement: 8

#### b. CALL

![Instruction Format](image)

- **CALL**
- Offset: 32

#### c. MOV EBX, [EDI + 45]

![Instruction Format](image)

- **MOV**
- d: 6
- w: 1
- r/m: 8
- Displacement: 8

#### d. PUSH ESI

![Instruction Format](image)

- **PUSH**
- Reg: 5
- Reg: 3

#### e. ADD EAX, #6765

![Instruction Format](image)

- **ADD**
- Reg: 4
- w: 3
- Immediate: 32

#### f. TEST EDX, #42

![Instruction Format](image)

- **TEST**
- w: 7
- Postbyte: 1
- Immediate: 32
Fallacies

- Powerful instruction $\Rightarrow$ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions

- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code $\Rightarrow$ more errors and less productivity
Fallacies

- Backward compatibility ⇒ instruction set doesn’t change
  - But they do accrete more instructions

![Graph showing the increasing number of instructions in the x86 instruction set from 1978 to 2008.](image)
Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
- Consider making the common case fast
- Consider compromises

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>MIPS examples</th>
<th>SPEC2006 Int</th>
<th>SPEC2006 FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add, sub, addi</td>
<td>16%</td>
<td>48%</td>
</tr>
<tr>
<td>Data transfer</td>
<td>lw, sw, lb, lbu, lh, lhu, sb, lui</td>
<td>35%</td>
<td>36%</td>
</tr>
<tr>
<td>Logical</td>
<td>and, or, nor, andi, ori, sll, srl</td>
<td>12%</td>
<td>4%</td>
</tr>
<tr>
<td>Cond. Branch</td>
<td>beq, bne, slt, slti, sltiu</td>
<td>34%</td>
<td>8%</td>
</tr>
<tr>
<td>Jump</td>
<td>j, jr, jal</td>
<td>2%</td>
<td>0%</td>
</tr>
</tbody>
</table>
Chapter 3

Arithmetic for Computers
Chapter 3 — Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers
  - Representation and operations
Integer Addition

Example: 7 + 6

<table>
<thead>
<tr>
<th>(0)</th>
<th>(0)</th>
<th>(1)</th>
<th>(1)</th>
<th>(0)</th>
<th>(Carries)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>1</td>
<td>(1)</td>
</tr>
<tr>
<td>...</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>0</td>
<td>(0)</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Overflow if result out of range

- Adding +ve and –ve operands, no overflow
- Adding two +ve operands
  - Overflow if result sign is 1
- Adding two –ve operands
  - Overflow if result sign is 0
Integer Subtraction

- Add negation of second operand
- Example: 7 – 6 = 7 + (–6)
  
  +7: 0000 0000 ... 0000 0111
  
  –6: 1111 1111 ... 1111 1010
  
  +1: 0000 0000 ... 0000 0001

- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign is 1
Dealing with Overflow

- Some languages (e.g., C) ignore overflow
  - Use MIPS `addu`, `addui`, `subu` instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
  - Use MIPS `add`, `addi`, `sub` instructions
  - On overflow, invoke exception handler
    - Save PC in exception program counter (EPC) register
    - Jump to predefined handler address
    - `mfc0` (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action
Other Adders

- BASICS of ADDING LOGIC
  - Carry-out
  - Sum Generation

- Ripple Add
- Carry Bypass
- Carry Select
- Carry Lookahead
The Ripple-Carry Adder

Worst case delay linear with the number of bits

$$t_d = O(N)$$

$$t_{adder} = (N-1)t_{carry} + t_{sum}$$

Goal: Make the fastest possible carry path circuit
Carry-Bypass Adder

Also called Carry-Skip

Idea: If (P0 and P1 and P2 and P3 = 1) then \( C_{o3} = C_0 \), else “kill” or “generate”.

BP = \( P_0 P_1 P_2 P_3 \)
Linear Carry Select

\[ t_{add} = t_{setup} + \left( \frac{N}{M} \right) t_{carry} + M t_{mux} + t_{sum} \]
LookAhead - Basic Idea

\[ C_{0,k} = f(A_k, B_k, C_{0,k-1}) = G_k + P_k C_{0,k-1} \]
Look-Ahead: Topology

Expanding Lookahead equations:

\[ C_{o,k} = G_k + P_k (G_{k-1} + P_{k-1} C_{o,k-2}) \]

All the way:

\[ C_{o,k} = G_k + P_k (G_{k-1} + P_{k-1} (\cdots + P_1 (G_0 + P_0 C_{i,0}))) \]
Carry Lookahead Trees

\[ C_{o,0} = G_0 + P_0 C_{i,0} \]
\[ C_{o,1} = G_1 + P_1 G_0 + P_1 P_0 C_{i,0} \]
\[ C_{o,2} = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{i,0} \]
\[ = (G_2 + P_2 G_1) + (P_2 P_1)(G_0 + P_0 C_{i,0}) = G_{2:1} + P_{2:1} C_{o,0} \]

Can continue building the tree hierarchically.
Multiplication

Start with long-multiplication approach

\[
\begin{array}{c}
\text{multiplicand} \\
1000 \\
\times \\
1001 \\
\hline
1000 \\
0000 \\
0000 \\
1000 \\
\hline
1001000
\end{array}
\]

Length of product is the sum of operand lengths

§3.3 Multiplication
Multiplication Hardware

Multiplicand

Shift left

64 bits

Multiplier

Shift right

32 bits

Control test

Initially 0

Chapter 3 — Arithmetic for Computers — 32
Optimized Multiplier (ignore)

- Perform steps in parallel: add/shift
- One cycle per partial-product addition
  - That’s ok, if frequency of multiplications is low
Faster Multiplier

- Uses multiple adders
- Cost/performance tradeoff

- Can be pipelined
- Several multiplication performed in parallel
MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32 bits

- Instructions
  - `mult rs, rt` / `multu rs, rt`
    - 64-bit product in HI/LO
  - `mfhi rd` / `mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - Least-significant 32 bits of product → rd
Division (IGNORE THIS)

- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

$n$-bit operands yield $n$-bit quotient and remainder
Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - $-2.34 \times 10^{56}$ (normalized)
  - $+0.002 \times 10^{-4}$ (not normalized)
  - $+987.02 \times 10^9$
- In binary
  - $\pm 1.xxxxxxxxx_{2} \times 2^{yyyy}$
- Types float and double in C
Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)
IEEE Floating-Point Format

S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
Normalize significand: $1.0 \leq |\text{significand}| < 2.0$
   - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
   - Significand is Fraction with the “1.” restored
Exponent: excess representation: actual exponent + Bias
   - Ensures exponent is unsigned
   - Single: Bias = 127; Double: Bias = 1203

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$
Single-Precision Range

- Exponents 00000000 and 11111111 reserved

- Smallest value
  - Exponent: 00000001
    ⇒ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 ⇒ significand = 1.0
  - ±1.0 × 2^{−126} ≈ ±1.2 × 10^{−38}

- Largest value
  - exponent: 11111110
    ⇒ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - ±2.0 × 2^{+127} ≈ ±3.4 × 10^{+38}
Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    ⇒ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 ⇒ significand = 1.0
    ±1.0 × 2^{–1022} ≈ ±2.2 × 10^{–308}
- Largest value
  - Exponent: 11111111110
    ⇒ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 ⇒ significand ≈ 2.0
    ±2.0 × 2^{+1023} ≈ ±1.8 × 10^{+308}
Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx $2^{-23}$
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx $2^{-52}$
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision
Floating-Point Example

- Represent \(-0.75\)
  - \(-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}\)
  - \(S = 1\)
  - Fraction = \(1000\ldots00_2\)
  - Exponent = \(-1 + \text{Bias}\)
    - Single: \(-1 + 127 = 126 = 01111110_2\)
    - Double: \(-1 + 1023 = 1022 = 01111111110_2\)
- Single: \(1011111101000\ldots00\)
- Double: \(10111111111101000\ldots00\)
Floating-Point Example

What number is represented by the single-precision float

11000000101000...00

\[ x = (-1)^1 \times (1 + 01) \times 2^{129 - 127} = -5.0 \]
Denormal Numbers

- Exponent = 000...0 ⇒ hidden bit is 0
  
  \[ x = (-1)^S \times (0 + \text{Fraction}) \times 2^{-\text{Bias}} \]

- Smaller than normal numbers
  - allow for gradual underflow, with diminishing precision

- Denormal with fraction = 000...0
  
  \[ x = (-1)^S \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0 \]

Two representations of 0.0!
Infinities and NaNs

- **Exponent = 111...1, Fraction = 000...0**
  - $\pm\text{Infinity}$
  - Can be used in subsequent calculations, avoiding need for overflow check

- **Exponent = 111...1, Fraction \neq 000...0**
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., $0.0 / 0.0$
  - Can be used in subsequent calculations
Floating-Point Addition (IGNORE)

Consider a 4-digit decimal example
- $9.999 \times 10^1 + 1.610 \times 10^{-1}$

1. Align decimal points
- Shift number with smaller exponent
  - $9.999 \times 10^1 + 0.016 \times 10^1$

2. Add significands
  - $9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$

3. Normalize result & check for over/underflow
  - $1.0015 \times 10^2$

4. Round and renormalize if necessary
  - $1.002 \times 10^2$
FP Adder Hardware (ignore)
Consider a 4-digit decimal example

\[ 1.110 \times 10^{10} \times 9.200 \times 10^{-5} \]

1. Add exponents
   - For biased exponents, subtract bias from sum
   - New exponent = 10 + –5 = 5

2. Multiply significands
   - \[ 1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5} \]

3. Normalize result & check for over/underflow
   - \[ 1.0212 \times 10^{6} \]

4. Round and renormalize if necessary
   - \[ 1.021 \times 10^{6} \]

5. Determine sign of result from signs of operands
   - \[ +1.021 \times 10^{6} \]
FP Arithmetic Hardware (ignore)

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder

- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP ↔ integer conversion

- Operations usually takes several cycles
  - Can be pipelined
FP Instructions in MIPS

- **Single-precision arithmetic**
  - `add.s`, `sub.s`, `mul.s`, `div.s`
  - e.g., `add.s $f0, $f1, $f6`

- **Double-precision arithmetic**
  - `add.d`, `sub.d`, `mul.d`, `div.d`
  - e.g., `mul.d $f4, $f4, $f6`

- **Single- and double-precision comparison**
  - `c.xx.s`, `c.xx.d` (xx is eq, lt, le, ...)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`

- **Branch on FP condition code true or false**
  - `bc1t`, `bc1f`
    - e.g., `bc1t TargetLabel`
FP Example: °F to °C

- **C code:**
  ```c
  float f2c (float fahr) {
    return ((5.0/9.0)*(fahr - 32.0));
  }
  ```
  - `fahr` in $f12$, result in $f0$, literals in global memory space

- **Compiled MIPS code:**
  ```assembly
  f2c: lwc1  $f16, const5($gp)  
lwc2  $f18, const9($gp)  
div.s $f16, $f16, $f18  
lwc1  $f18, const32($gp)  
sub.s $f18, $f12, $f18  
mul.s $f0, $f16, $f18  
jr    $ra
  ```
Accurate Arithmetic

IEEE Std 754 specifies additional rounding control
- Extra bits of precision (guard, round, sticky)
- Choice of rounding modes
- Allows programmer to fine-tune numerical behavior of a computation

Not all FP units implement all options
- Most programming languages and FP libraries just use defaults

Trade-off between hardware complexity, performance, and market requirements
x86 FP Architecture (ignore)

- Originally based on 8087 FP coprocessor
  - 8 × 80-bit extended-precision registers
  - Used as a push-down stack
  - Registers indexed from TOS: ST(0), ST(1), …
- FP values are 32-bit or 64 in memory
  - Converted on load/store of memory operand
  - Integer operands can also be converted on load/store
- Very difficult to generate and optimize code
  - Result: poor FP performance
## x86 FP Instructions (ignore)

<table>
<thead>
<tr>
<th>Data transfer</th>
<th>Arithmetic</th>
<th>Compare</th>
<th>Transcendental</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILDMem/ST(i)</td>
<td>FIADDPmem/ST(i)</td>
<td>FICOMP</td>
<td>FPATAN</td>
</tr>
<tr>
<td>FISTPMem/ST(i)</td>
<td>FISUBLPmem/ST(i)</td>
<td>FIUCOMP</td>
<td>F2XMI</td>
</tr>
<tr>
<td>FLDP</td>
<td>FIMULPmem/ST(i)</td>
<td>FSTSWAX/mem</td>
<td>FCOS</td>
</tr>
<tr>
<td>FLDP</td>
<td>FIDIVRPmem/ST(i)</td>
<td></td>
<td>FPTAN</td>
</tr>
<tr>
<td>FLDP</td>
<td>FSQRT</td>
<td></td>
<td>FPREM</td>
</tr>
<tr>
<td>FLDP</td>
<td>FABS</td>
<td></td>
<td>FPSIN</td>
</tr>
<tr>
<td>FLDP</td>
<td>FRNDINT</td>
<td></td>
<td>FYL2X</td>
</tr>
</tbody>
</table>

- Optional variations
  - **I**: integer operand
  - **P**: pop operand from stack
  - **R**: reverse operand order
  - But not all combinations allowed
Who Cares About FP Accuracy?

- Important for scientific code
  - But for everyday consumer use?
    - “My bank balance is out by 0.0002¢!” 😞

- The Intel Pentium FDIV bug
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*
Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow
- MIPS ISA
  - Core instructions: 54 most frequently used
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent