Chapter 3

Arithmetic for Computers
Arithmetic

Where we've been:
Arithmetic

Where we've been:
- Abstractions:
  - Instruction Set Architecture
  - Assembly Language and Machine Language
- Performance (seconds, cycles, instructions)

What's up ahead:
- Implementing the Architecture
Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow

- Floating-point real numbers
  - Representation and operations
Interpretation of Data

- Bits have no inherent meaning
  - Interpretation depends on the instructions applied
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs
Numbers

- Of course it gets more complicated:
  - numbers are finite (overflow)
  - fractions and real numbers
  - negative numbers
  - e.g., no MIPS subi instruction; addi can add a negative number)
- How do we represent negative numbers? i.e., which bit patterns will represent which numbers?
Possible Representations

- Sign Magnitude:
  - 000 = +0
  - 001 = +1
  - 010 = +2
  - 011 = +3
  - 100 = -0
  - 101 = -1
  - 110 = -2
  - 111 = -3

- One's Complement:
  - 000 = +0
  - 001 = +1
  - 010 = +2
  - 011 = +3
  - 100 = -3
  - 101 = -2
  - 110 = -1
  - 111 = -0

- Two's Complement:
  - 000 = +0
  - 001 = +1
  - 010 = +2
  - 011 = +3
  - 100 = -4
  - 101 = -3
  - 110 = -2
  - 111 = -1

- Issues: balance, number of zeros, ease of operations
- Which one is best? Why?
32 bit signed numbers:

- 0000 0000 0000 0000 0000 0000 0000 0000$_{two}$ = 0$_{ten}$
- 0000 0000 0000 0000 0000 0000 0000 0001$_{two}$ = + 1$_{ten}$
- 0000 0000 0000 0000 0000 0000 0000 0010$_{two}$ = + 2$_{ten}$
- ... 
- 0111 1111 1111 1111 1111 1111 1111 1110$_{two}$ = + 2,147,483,646$_{ten}$
- 0111 1111 1111 1111 1111 1111 1111 1111$_{two}$ = + 2,147,483,647$_{ten}$
- 1000 0000 0000 0000 0000 0000 0000 0000$_{two}$ = -2,147,483,648$_{ten}$
- 1000 0000 0000 0000 0000 0000 0000 0001$_{two}$ = -2,147,483,647$_{ten}$
- 1000 0000 0000 0000 0000 0000 0000 0010$_{two}$ = -2,147,483,646$_{ten}$
- ... 
- 1111 1111 1111 1111 1111 1111 1111 1101$_{two}$ = -3$_{ten}$
- 1111 1111 1111 1111 1111 1111 1111 1110$_{two}$ = -2$_{ten}$
- 1111 1111 1111 1111 1111 1111 1111 1111$_{two}$ = -1$_{ten}$

$maxint$

$minint$
Two's Complement Operations

- Negating a two's complement number:
  invert all bits and add 1
  - remember: “negate” and “invert” are quite different!
Two's Complement Operations

- Converting n bit numbers into numbers with more than n bits:
  - MIPS 16 bit immediate gets converted to 32 bits for arithmetic
  - copy the most significant bit (the sign bit) into the other bits
    
    \[
    \begin{align*}
    0010 & \rightarrow 0000 \ 0010 \\
    1010 & \rightarrow 1111 \ 1010 \\
    \end{align*}
    \]
  - "sign extension" (lbu vs. lb)
Integer Addition

Example: $7 + 6$

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: $7 - 6 = 7 + (-6)$
  
  $$
  \begin{array}{ll}
  +7: & 0000\,0000\,\ldots\,0000\,0111 \\
  -6: & 1111\,1111\,\ldots\,1111\,1010 \\
  +1: & 0000\,0000\,\ldots\,0000\,0001 \\
  \end{array}
  $$

- 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
Detecting Overflow

- Consider the operations $A + B$, and $A - B$
  - Can overflow occur if $B$ is 0?
  - Can overflow occur if $A$ is 0?
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
  - `mfco` (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action
Effects of Overflow

Don't always want to detect overflow — new MIPS instructions: addu, addiu, subu

note: addiu still sign-extends!

note: sltu, sltiu for unsigned comparisons
Arithmetic for Multimedia

- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use 64-bit adder, with partitioned carry chain
    - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
  - SIMD (single-instruction, multiple-data)
- Saturating operations
  - On overflow, result is largest value that can be represented
    - c.f. 2s-complement modulo arithmetic
  - E.g., clipping in audio, saturation in video

Electrical & Computer Engineering
School of Engineering
THE COLLEGE OF NEW JERSEY
Problem: Consider a logic function with three inputs: A, B, and C.

- Output D is true if at least one input is true.
- Output E is true if exactly two inputs are true.
- Output F is true only if all three inputs are true.
Review: Boolean Algebra & Gates

- Show the truth table for these three functions.

- Show the Boolean equations for these three functions.

- Show an implementation consisting of inverters, AND, and OR gates.
An ALU (arithmetic logic unit)

- Let's build an ALU to support the \textit{andi} and \textit{ori} instructions
  - we'll just build a 1 bit ALU, and use 32 of them

Possible Implementation (sum-of-products):
Review: The Multiplexer

- Selects one of the inputs to be the output, based on a control input

  S

  A \rightarrow 0 \rightarrow C
  B \rightarrow 1

  note: we call this a 2-input mux even though it has 3 inputs!

- Lets build our ALU using a MUX:
Different Implementations

- Not easy to decide the “best” way to build something
  - Don’t want too many inputs to a single gate
  - Don’t want to have to go through too many gates
  - For our purposes, ease of comprehension is important
- Let’s look at a 1-bit ALU for addition:

- \[ c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in} \]
- \[ \text{sum} = a \oplus b \oplus c_{in} \]

- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?
Building a 32 bit ALU
What about subtraction \((a - b)\) ?

- Two's complement approach: just negate \(b\) and add.
- How do we negate?
- A very clever solution:
Tailoring the ALU to the MIPS

- Need to support the set-on-less-than instruction (slt)
  - remember: slt is an arithmetic instruction
  - produces a 1 if rs < rt and 0 otherwise
  - use subtraction: \((a-b) < 0\) implies \(a < b\)

- Need to support test for equality (beq $t5, $t6, $t7)
  - use subtraction: \((a-b) = 0\) implies \(a = b\)
Can we figure out the idea?
Test for equality

Notice control lines:

- 000 = and
- 001 = or
- 010 = add
- 110 = subtract
- 111 = slt

*Note: zero is a 1 when the result is zero!
Conclusion

- We can build an ALU to support the MIPS instruction set
  - key idea: use multiplexer to select the output we want
  - we can efficiently perform subtraction using two’s complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU
Conclusion

- Important points about hardware
  - all of the gates are always working
  - the speed of a gate is affected by the number of inputs to the gate
  - the speed of a circuit is affected by the number of gates in series (on the “critical path” or the “deepest level of logic”)
Conclusion

- Our primary focus: comprehension, however,
  - Clever changes to organization can improve performance (similar to using better algorithms in software)
  - we’ll look at two examples for addition and multiplication
Problem: ripple carry adder is slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition?
  - two extremes: ripple carry and sum-of-products

Can you see the ripple? How could you get rid of it?

\[
\begin{align*}
c_1 &= b_0c_0 + a_0c_0 + a_0b_0 \\
c_2 &= b_1c_1 + a_1c_1 + a_1b_1 \\
c_3 &= b_2c_2 + a_2c_2 + a_2b_2 \\
c_4 &= b_3c_3 + a_3c_3 + a_3b_3
\end{align*}
\]

Not feasible! Why?
Carry-look-ahead adder

- An approach in-between our two extremes
- Motivation:
  - If we didn't know the value of carry-in, what could we do?
  - When would we always generate a carry? \( g_i = a_i b_i \)
  - When would we propagate the carry? \( p_i = a_i \oplus b_i \)
  - Did we get rid of the ripple?

\[
\begin{align*}
    c_1 &= g_0 + p_0 c_0 \\
    c_2 &= g_1 + p_1 c_1 \\
    c_3 &= g_2 + p_2 c_2 \\
    c_4 &= g_3 + p_3 c_3 \\
\end{align*}
\]

Feasible! Why?

Electrical & Computer Engineering
School of Engineering
THE COLLEGE OF NEW JERSEY
Use principle to build bigger adders

- Can’t build a 16 bit adder this way... (too big)
- Could use ripple carry of 4-bit CLA adders
- Better: use the CLA principle again!
Multiplication

Start with long-multiplication approach

- multiplicand
- multiplier
- product

Length of product is the sum of operand lengths

Multiplicand × multiplier = product

Length of product is the sum of operand lengths

Electrical & Computer Engineering
School of Engineering
THE COLLEGE OF NEW JERSEY
Multiplication Hardware

1. Test Multiplier0
   - Multiplier0 = 1
   - Multiplier0 = 0

   1a. Add multiplicand to product and place the result in Product register

   2. Shift the Multiplicand register left 1 bit

   3. Shift the Multiplier register right 1 bit

   32nd repetition?

   No: < 32 repetitions

   Yes: 32 repetitions

   Done

   Multiplicand

   64-bit ALU

   Product

   Write

   Control test

   Initially 0

   Multiplier Shift right

   32 bits

   64 bits

   Shift left

   64 bits

Electrical & Computer Engineering
School of Engineering
THE COLLEGE OF NEW JERSEY
Optimized Multiplier

- 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` / `mulu 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

- Check for 0 divisor
- Long division approach
  - If divisor $\leq$ 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
Division Hardware

Initially divisor in left half

1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register

2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0

3. Shift the Divisor register right 1 bit

No: < 33 repetitions
Yes: 33 repetitions

Done

64-bit ALU

Divisor

Remainder

Write

Control test

Initially dividend

Electrical & Computer Engineering
School of Engineering
THE COLLEGE OF NEW JERSEY
Optimized Divider

- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both
Faster Division

- Can’t use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder
- Faster dividers (e.g. SRT division) generate multiple quotient bits per step
  - Still require multiple steps
MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient

- Instructions
  - div rs, rt / divu rs, rt
  - No overflow or divide-by-0 checking
    - Software must perform checks if required
  - Use mfhi, mflo to access result
Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - $-2.34 \times 10^{56}$
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$
- In binary
  - $\pm1.xxx\ldots x_{2} \times 2^{yyyy}$
- 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)
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
IEEE Floating-Point Format

<table>
<thead>
<tr>
<th></th>
<th>Single: 8 bits</th>
<th>Single: 23 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Double: 11 bits</td>
<td>Double: 52 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
</table>

\[ x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})} \]

- **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 = 1023
**Single-Precision Range**

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    ⇒ actual exponent = 1 – 127 = −126
  - Fraction: 000...00 ⇒ significand = 1.0
  - $±1.0 \times 2^{-126} \approx ±1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    ⇒ actual exponent = 254 – 127 = +127
  - Fraction: 111...11 ⇒ significand ≈ 2.0
  - $±2.0 \times 2^{+127} \approx ±3.4 \times 10^{+38}$
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: $-1 + 1023 = 1022 = 0111111110_2$
- Single: $1011111101000\ldots00$
- Double: $1011111111101000\ldots00$
Floating-Point Example

- What number is represented by the single-precision float
  \[11000000101000\ldots00\]
  - \(S = 1\)
  - Fraction = \[01000\ldots00\]₂
  - Exponent = \[10000001\]₂ = 129
  - \(x = (-1)^1 \times (1 + 01) \times 2^{(129 - 127)}\)
  - \[= (-1) \times 1.25 \times 2^2\]
  - \[= -5.0\]
Floating Point Complexities

- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round
  - four rounding modes
  - positive divided by zero yields “infinity”
  - zero divide by zero yields “not a number”
  - other complexities
Floating Point Complexities

- Implementing the standard can be tricky
- Not using the standard can be even worse
  - see text for description of 80x86 and Pentium bug!
Floating Point Complexities

- Operations are somewhat more complicated (see text)
- In addition to overflow we can have “underflow”
Non Normal Numbers

- Exponent = 000...0 $\Rightarrow$ 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

- Non Normal 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
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, Fraction ≠ 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

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

   - $1.002 \times 10^2$
Floating-Point Addition

Now consider a 4-digit binary example

1.000₂ × 2⁻¹ + –1.110₂ × 2⁻² (0.5 + –0.4375)

1. Align binary points

   Shift number with smaller exponent

   1.000₂ × 2⁻¹ + –0.111₂ × 2⁻¹

2. Add significands

   1.000₂ × 2⁻¹ + –0.111₂ × 2⁻¹ = 0.001₂ × 2⁻¹

3. Normalize result & check for over/underflow

   1.000₂ × 2⁻⁴, with no over/underflow

4. Round and renormalize if necessary

   1.000₂ × 2⁻⁴ (no change) = 0.0625
FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined
FP Adder Hardware

Step 1
- Compare exponents
- Shift smaller number right

Step 2
- Add
- Normalize
- Round

Step 3

Step 4
Floating-Point Multiplication

- 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
   - $+1.021 \times 10^6$
Floating-Point Multiplication

Now consider a 4-digit binary example

1.000₂ × 2⁻¹ × −1.110₂ × 2⁻² (0.5 × −0.4375)

1. Add exponents
   - Unbiased: −1 + −2 = −3
   - Biased: (−1 + 127) + (−2 + 127) = −3 + 254 − 127 = −3 + 127

2. Multiply significands
   - 1.000₂ × 1.110₂ = 1.110₂ ⇒ 1.110₂ × 2⁻³

3. Normalize result & check for over/underflow
   - 1.110₂ × 2⁻³ (no change) with no over/underflow

4. Round and renormalize if necessary
   - 1.110₂ × 2⁻³ (no change)

5. Determine sign: +ve × −ve ⇒ −ve
   - −1.110₂ × 2⁻³ = −0.21875
FP Arithmetic Hardware

- 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

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: \$f0, \$f1, ... \$f31
  - Paired for double-precision: \$f0/\$f1, \$f2/\$f3, ...
    - Release 2 of MIPs ISA supports 32 × 64-bit FP reg’s
- FP instructions operate only on FP registers
  - Programs generally don’t do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - \texttt{lwc1, ldc1, swc1, sdc1}
  - e.g., \texttt{ldc1 \$f8, 32(\$sp)}
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 Target Label
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:

```mips
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
```

Electrical & Computer Engineering
School of Engineering
THE COLLEGE OF NEW JERSEY
FP Example: Array Multiplication

- $X = X + Y \times Z$
  - All $32 \times 32$ matrices, 64-bit double-precision elements
- C code:

```c
void mm(double x[][[]], double y[][[]], double z[][[]]) {
    int i, j, k;
    for (i = 0; i! = 32; i = i + 1)
        for (j = 0; j! = 32; j = j + 1)
            for (k = 0; k! = 32; k = k + 1)
                x[i][j] = x[i][j]
                        + y[i][k] * z[k][j];
}
```
- Addresses of $x, y, z$ in $a0, a1, a2$, and $i, j, k$ in $s0, s1, s2$
### FP Example: Array Multiplication

**MIPS code:**

```mips
li   $t1, 32       # $t1 = 32 (row size/loop end)
li   $s0, 0        # i = 0; initialize 1st for loop
L1:  li   $s1, 0        # j = 0; restart 2nd for loop
L2:  li   $s2, 0        # k = 0; restart 3rd for loop

sll  $t2, $s0, 5   # $t2 = i * 32 (size of row of x)
addu $t2, $t2, $s1 # $t2 = i * size(row) + j
sll  $t2, $t2, 3   # $t2 = byte offset of [i][j]
addu $t2, $a0, $t2 # $t2 = byte address of x[i][j]
l.d $f4, 0($t2)    # $f4 = 8 bytes of x[i][j]

L3: sll  $t0, $s2, 5   # $t0 = k * 32 (size of row of z)
addu $t0, $t0, $s1 # $t0 = k * size(row) + j
sll  $t0, $t0, 3   # $t0 = byte offset of [k][j]
addu $t0, $a2, $t0 # $t0 = byte address of z[k][j]
l.d $f16, 0($t0)   # $f16 = 8 bytes of z[k][j]
```

...
## FP Example: Array Multiplication

```
sll $t0, $s0, 5       # $t0 = i*32 (size of row of y)
addu $t0, $t0, $s2    # $t0 = i*size(row) + k
sll $t0, $t0, 3       # $t0 = byte offset of [i][k]
addu $t0, $a1, $t0    # $t0 = byte address of y[i][k]
l.d $f18, 0($t0)      # $f18 = 8 bytes of y[i][k]
```

```
mul.d $f16, $f18, $f16 # $f16 = y[i][k] * z[k][j]
add.d $f4, $f4, $f16   # $f4=x[i][j] + y[i][k]*z[k][j]
addiu $s2, $s2, 1     # $k k + 1
bne $s2, $t1, L3      # if (k != 32) go to L3
s.d $f4, 0($t2)       # x[i][j] = $f4
```

```
addiu $s1, $s1, 1     # $j = j + 1
bne $s1, $t1, L2      # if (j != 32) go to L2
addiu $s0, $s0, 1     # $i = i + 1
bne $s0, $t1, L1      # if (i != 32) go to L1
```
Associativity

Parallel programs may interleave operations in unexpected orders

Assumptions of associativity may fail

Need to validate parallel programs under varying degrees of parallelism

<table>
<thead>
<tr>
<th></th>
<th>(x+y)+z</th>
<th>x+(y+z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-1.50E+38</td>
<td>-1.50E+38</td>
</tr>
<tr>
<td>y</td>
<td>1.50E+38</td>
<td>0.00E+00</td>
</tr>
<tr>
<td>z</td>
<td>1.0</td>
<td>1.50E+38</td>
</tr>
<tr>
<td></td>
<td>1.00E+00</td>
<td>0.00E+00</td>
</tr>
</tbody>
</table>
**x86 FP Architecture**

- 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

<table>
<thead>
<tr>
<th>Data transfer</th>
<th>Arithmetic</th>
<th>Compare</th>
<th>Transcendental</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILD mem/ST(i)</td>
<td>F1 ADDP mem/ST(i)</td>
<td>F1 COMP</td>
<td>FPATAN</td>
</tr>
<tr>
<td>FISTP mem/ST(i)</td>
<td>F1 SUBRP mem/ST(i)</td>
<td>F1 UCOMP</td>
<td>F2XM</td>
</tr>
<tr>
<td>FLDPI</td>
<td>F1 MULP mem/ST(i)</td>
<td></td>
<td>FCOS</td>
</tr>
<tr>
<td>FLDI</td>
<td>F1 DIVRP mem/ST(i)</td>
<td></td>
<td>FPTAN</td>
</tr>
<tr>
<td>FLDZ</td>
<td>FSQRT</td>
<td>FSTSW AX/ mem</td>
<td>FPREM</td>
</tr>
<tr>
<td></td>
<td>FABS</td>
<td></td>
<td>FPS1N</td>
</tr>
<tr>
<td></td>
<td>FRNDNT</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

---

Electrical & Computer Engineering
School of Engineering
THE COLLEGE OF NEW JERSEY
Streaming SIMD Extension 2 (SSE2)

- Adds 4 × 128-bit registers
  - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
  - 2 × 64-bit double precision
  - 4 × 32-bit double precision
  - Instructions operate on them simultaneously
    - Single-Instruction Multiple-Data
Right Shift and Division

- Left shift by $i$ places multiplies an integer by $2^i$
- Right shift divides by $2^i$?
  - Only for unsigned integers
- For signed integers
  - Arithmetic right shift: replicate the sign bit
  - e.g., $-5 / 4$
    - $1111011_2 >> 2 = 11111110_2 = -2$
    - Rounds toward $-\infty$
  - c.f. $1111011_2 >>> 2 = 001111110_2 = +62$
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
Computer arithmetic is constrained by limited precision

Bit patterns have no inherent meaning but standards do exist
  - two’s complement
  - IEEE 754 floating point

Computer instructions determine “meaning” of the bit patterns
Chapter Three Summary

- Performance and accuracy are important so there are many complexities in real machines (i.e., algorithms and implementation).
- Algorithm choice is important and may lead to hardware optimizations for both space and time (e.g., multiplication).
Chapter Three Summary

- We are ready to move on

You may want to look back (Section 3.10 is great reading!)