




Arithmetic for Computers



Arithmetic for Computers

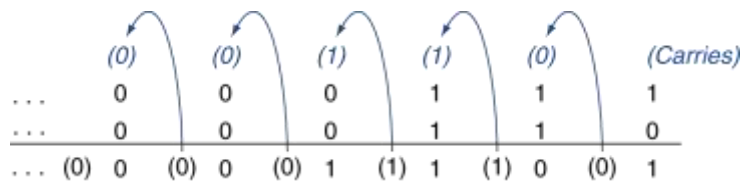
- Operations on integers
 - Addition and subtraction
 - Multiplication and division
 - Dealing with overflow
- Floating-point real numbers
 - Representation and operations

2



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)

+7:	0000 0000 ... 0000 0111
-6:	<u>1111 1111 ... 1111 1010</u>
+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

Arithmetic for Computers




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

5

Arithmetic for Computers




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 representable value
 - c.f. 2s-complement modulo arithmetic
 - E.g., clipping in audio, saturation in video

6

Arithmetic for Computers



Multiplication

- Start with long-multiplication approach

multiplicand

multiplier

product

1000

1001

1000000

x

1000

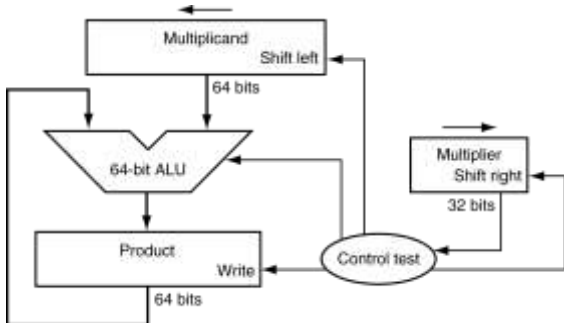
0000

0000

1000


1001000

Length of product is the sum of operand lengths




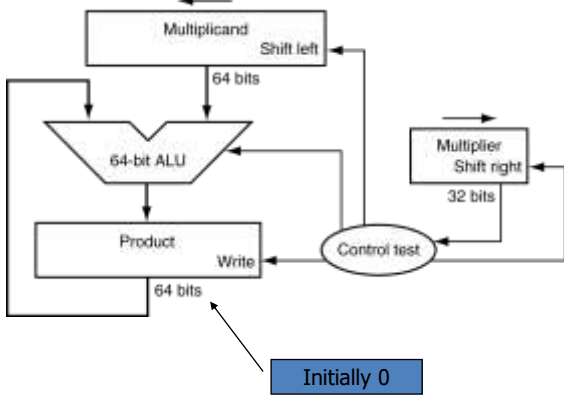
7

Arithmetic for Computers




Multiplication Hardware





8

Arithmetic for Computers




Example

- Using 4-bit numbers, multiply $2_{10} \times 3_{10}$

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 \Rightarrow Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 \Rightarrow Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 \Rightarrow No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 \Rightarrow No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

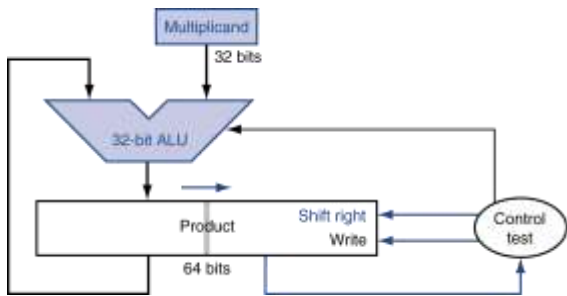
9

Arithmetic for Computers



Optimized Multiplier


- Perform steps in parallel: add/shift



- One cycle per partial-product addition
 - That's ok, if frequency of multiplications is low

10

Arithmetic for Computers



Division

quotient

dividend

divisor

remainder

1001

1000

10

101

1010

10

1001010

-1000

-1000


10

n-bit operands yield *n*-bit quotient and remainder

- 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

13

Arithmetic for Computers



Division Hardware

Start

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

Test Remainder

Remainder > 0

2a. Shift the Quotient register to the left, setting the new rightmost bit to 1

Remainder < 0

2b. Restore the original value by adding the Divisor register to the Remainder register and placing the result in the Remainder register. Also shift the Quotient register to the left, setting the new most significant bit to 0

3. Shift the Divisor register right 1 bit

Shift repetition?

No: < 32 repetitions

Yes: 32 repetitions

Done

Initially divisor in left half

Divisor

Shift right

64 bits

64-bit ALU

Remainder

Write

64 bits

Control test

Initially dividend


Quotient

Shift left

32 bits

14

Arithmetic for Computers




Example

- Using 4-bit numbers, let's try dividing 7_{10} by 2_{10}

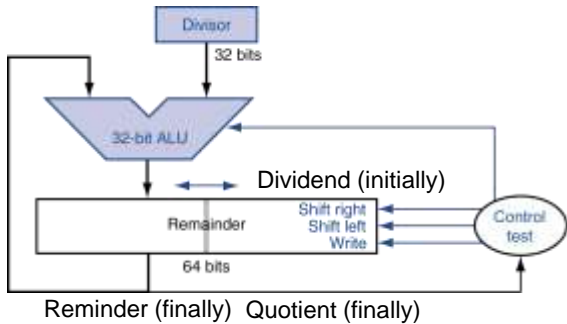
Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem - Div	0000	0010 0000	0110 0111
	2b: Rem < 0 \Rightarrow +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem - Div	0000	0001 0000	0111 0111
	2b: Rem < 0 \Rightarrow +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem - Div	0000	0000 1000	0111 1111
	2b: Rem < 0 \Rightarrow +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem - Div	0000	0000 0100	0000 0011
	2a: Rem \geq 0 \Rightarrow sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem - Div	0001	0000 0010	0000 0001
	2a: Rem \geq 0 \Rightarrow sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Arithmetic for Computers




Optimized Divider

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



16

Arithmetic for Computers




Faster Division

- Can't use parallel hardware as in multiplier
 - Subtraction is conditional on sign of remainder
- Faster dividers (e.g. SRT division, use a backup table instead of restoring) generate multiple quotient bits per step
 - Still require multiple steps

17

Arithmetic for Computers




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

18

Arithmetic for Computers



Floating Point


- Representation for non-integral numbers
 - Including very small and very large numbers
- Like scientific notation
 - -2.34×10^{56}
 - $+0.002 \times 10^{-4}$
 - $+987.02 \times 10^9$

normalized

not normalized
- In binary
 - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

19

Arithmetic for Computers




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)

20

Arithmetic for Computers



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 = 1023

single: 8 bits
double: 11 bits


single: 23 bits
double: 52 bits

S	Exponent	Fraction
---	----------	----------

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

21

Arithmetic for Computers




Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
 - Exponent: 00000001
⇒ actual exponent = $1 - 127 = -126$
 - Fraction: 000...00 ⇒ significand = 1.0
 - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
 - exponent: 11111110
⇒ actual exponent = $254 - 127 = +127$
 - Fraction: 111...11 ⇒ significand ≈ 2.0
 - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

22

Arithmetic for Computers




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
 - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
 - Exponent: 11111111110
 - ⇒ actual exponent = 2046 – 1023 = +1023
 - Fraction: 111...11 ⇒ significand ≈ 2.0
 - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

23

Arithmetic for Computers




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

24

Arithmetic for Computers




Real to IEEE 754 Conversion

- Step 1: Decide S
- Step 2: Decide Fraction
 - Convert the integer part to Binary
 - Convert the fractional part to Binary
 - Adjust the integer and fractional parts according the Significand format (1.xxx)
- Step 3: Decide exponent

25

Arithmetic for Computers




Floating-Point Example

- Represent -0.75
 - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
 - $S = 1$
 - Fraction = $1000...00_2$
 - Exponent = $-1 + \text{Bias}$
 - Single: $-1 + 127 = 126 = 01111110_2$
 - Double: $-1 + 1023 = 1022 = 01111111110_2$
- Single: $1011111101000...00$
- Double: $1011111111101000...00$

26

Arithmetic for Computers




Floating-Point Example

- What number is represented by the single-precision float
 $11000000101000...00$
 - $S = 1$
 - Fraction = $01000...00_2$
 - Exponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$
 $= (-1) \times 1.25 \times 2^2$
 $= -5.0$

27

Arithmetic for Computers




Infinites and NaNs

- Exponent = $111...1$, Fraction = $000...0$
 - \pm 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

29

Arithmetic for Computers




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
 - 1.0015×10^2
- 4. Round and renormalize if necessary
 - 1.002×10^2

30

Arithmetic for Computers




Floating-Point Addition

- Now consider a 4-digit binary example
 - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ ($0.5 + -0.4375$)
- 1. Align binary points
 - Shift number with smaller exponent
 - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
 - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
 - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
 - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

31

Arithmetic for Computers




FP Adder Hardware

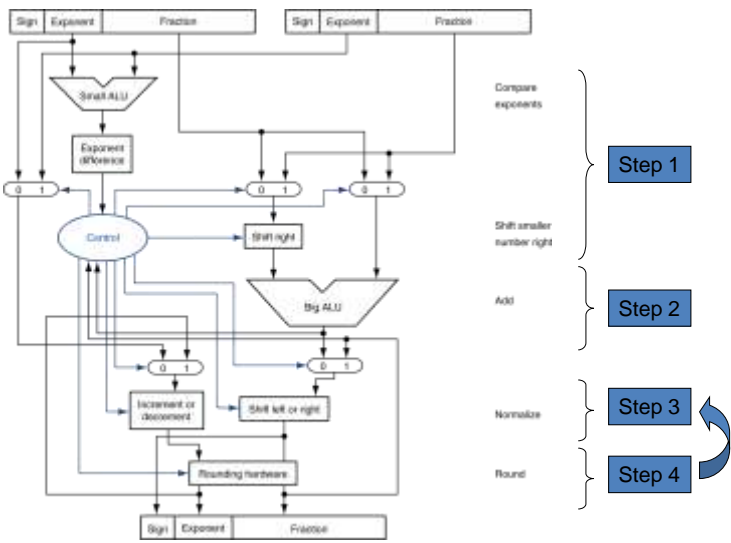
- Much more complex than integer adder
- Doing it in one clock cycle would take too long
 - Much longer than integer operations
 - Slower clock would penalize all instructions
- FP adder usually takes several cycles
 - Can be pipelined

32

Arithmetic for Computers




FP Adder Hardware



33

Arithmetic for Computers




Example

- $12.75 + 7.5 \Leftrightarrow 0x414C0000 + 0x40F00000$
- Calculate the following operator:
 $0x41A20000 + 0xC14C0000$

34

Arithmetic for Computers




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
 - 1.0212×10^6
- 4. Round and renormalize if necessary
 - 1.021×10^6
- 5. Determine sign of result from signs of operands
 - $+1.021 \times 10^6$

35

Arithmetic for Computers




Floating-Point Multiplication

- Now consider a 4-digit binary example
 - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)
- 1. Add exponents
 - Unbiased: $-1 + -2 = -3$
 - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
- 2. Multiply significands
 - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
 - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
 - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: $+ve \times -ve \Rightarrow -ve$
 - $-1.110_2 \times 2^{-3} = -0.21875$

36

Arithmetic for Computers




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
 - $FP \leftrightarrow integer$ conversion
- Operations usually takes several cycles
 - Can be pipelined

37

Arithmetic for Computers




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, ...
 - Odd-number registers: right half of 64-bit floating-point numbers
 - 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
 - lwc1, ldc1, swc1, sdc1
 - e.g., ldc1 \$f8, 32(\$sp)

38

Arithmetic for Computers




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

39

Arithmetic for Computers



FP Example: °F to °C

- C code:


```
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:

```
f2c: lwc1    $f16, const5($gp)  
     lwc1    $f18, const9($gp)  
     div.s   $f16, $f16, $f18  
     lwc1    $f18, const32($gp)  
     sub.s   $f18, $f12, $f18  
     mul.s   $f0,  $f16, $f18  
     jr      $ra
```

40

Arithmetic for Computers




FP Instruction Fields

Name	Format	Example						Comments
add.s	R	17	16	6	4	2	0	add.s \$f2,\$f4,\$f6
sub.s	R	17	16	6	4	2	1	sub.s \$f2,\$f4,\$f6
mul.s	R	17	16	6	4	2	2	mul.s \$f2,\$f4,\$f6
div.s	R	17	16	6	4	2	3	div.s \$f2,\$f4,\$f6
add.d	R	17	17	6	4	2	0	add.d \$f2,\$f4,\$f6
sub.d	R	17	17	6	4	2	1	sub.d \$f2,\$f4,\$f6
mul.d	R	17	17	6	4	2	2	mul.d \$f2,\$f4,\$f6
div.d	R	17	17	6	4	2	3	div.d \$f2,\$f4,\$f6
lwc1	I	49	20	2		100		lwc1 \$f2,100(\$s4)
swc1	I	57	20	2		100		swc1 \$f2,100(\$s4)
bclt	I	17	8	1		25		bclt 25
bcif	I	17	8	0		25		bcif 25
c.lt.s	R	17	16	4	2	0	60	c.lt.s \$f2,\$f4
c.lt.d	R	17	17	4	2	0	60	c.lt.d \$f2,\$f4
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits

41

Arithmetic for Computers



FP Example: Array Multiplication


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

```
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

42

Arithmetic for Computers




FP Example: Array Multiplication

- MIPS code:

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]
ldc1	\$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]
ldc1	\$f16, 0(\$t0)	# \$f16 = 8 bytes of z[k][j]
...		

43

Arithmetic for Computers




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]
ldc1	\$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
sdc1	\$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

44

Arithmetic for Computers




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

45

Arithmetic for Computers




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*

46

Arithmetic for Computers




Concluding Remarks

- 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

47

Arithmetic for Computers



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

48