Chapter 3

Arithmetic for Computers

Hyuk-Jun Lee, PhD

Dept. of Computer Science and Engineering
Sogang University
Seoul, Korea

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

- 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

Bad naming

flag를 무시

- 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 Setting flag bits
  - 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

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

Multiplication

- Start with long-multiplication approach

multiplicand

multiplier

product

```
      1000
  ×   1001
  --------
      1000
     0000
    0000
   1000
  --------
   1001000
```

Length of product is the sum of operand lengths

Multiplication Hardware

Multiplication Example

multiplier 비트 수만큼 iterate

multiplicand 가 양수면 앞에 0채우고
음수면 1채운다

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⟹ 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 ⟹ 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 ⟹ 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 ⟹ 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

한번 더하고 나서 다음거 곱하기 전에 leftshift

lsb 1일 때 product = mutiplicand + product

FIGURE 3.6 Multiply example using algorithm in Figure 3.4. The bit examined to determine the next step is circled in color.

Optimized Multiplier

- Perform steps in parallel: add/shift

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

Optimized Multiplier Ex2

2 *3 = (multiplicand)*(multiplier)

0010 * 0011

product

처음이라 0 multiplier

Cycle 0: 0000 | 0011

Cycle 1: 0010 | 0011

오른쪽 쉬프트

00010|001

+ 00100

Cycle 2: 00110| 001

000110| 00

Cycle 3: 000110| 00

0000110| 0

Cycle 4: 0000110| 0

00000110

lsb에 따라 multiplicand를 더하거나 아무거도 안하거나
여기선 1이니깐 더함(0010 + 0000)

Optimized Multiplier

양수여야함

-6 * 5 = (multiplicand)*(multiplier)

1010 * 0101

multiplier는 항상 양수가 되도록 multiplicand랑 값을 바꿔야 하고,
음수 * 음수 일 때는 답은 양수니 multiplicand랑 multiplier를 양수로 바꾼다

Cycle 0: 0000 | 0101

Cycle 1: 1010 | 0101

11010 | 010

Cycle 2: 11010 | 010

111010| 01

오른쪽 쉬프트 할 때 sign extension
logical shift

Cycle 3: 100010| 01

1100010| 0

Cycle 4: 1100010| 0

11100010 => (pos)11110 = 16+8+4+2 = 30

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

unsigned multiplication
addition에서는 u가 overflow를 체크할건지 뜻하는 거지 진짜 unsigned 계산은 아니었는데
multi에서는 진짜 unsigned 계산임

  - `mult rs, rt / multu rs, rt`
    - 64-bit product in HI/LO

HI/LO special reg

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

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 to get original number
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

Division Hardware

dividant - divisor을 한다음 sign 값이 1이면
remainder에 더하지 않고 옛날 값이 남아있어서 restore 안해도된다

32bit div인데 33 iteratiion임

Division Example

0010 0111

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⟹ +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	1111 0111
	2b: Rem < 0 ⟹ +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	1111 1111
	2b: Rem < 0 ⟹ +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 ≥ 0 ⟹ 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 ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

FIGURE 3.10 Division example using the algorithm in Figure 3.9. The bit examined to determine the next step is circled in color.

Optimized Divider

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

dont shift divisor but shift remainder

Optimized Divisor

multiplicand는 앞에 sign bit를 붙여서 remainder랑 같은 길이를 만들었는데
divisor는 뒤에 0을 붙임

remainder가 음수면 앞에 1로 채움

0	steps	Divisor 0010 0000 (2)	Remainder 00000 0111 (7) ^signbit을 확인하기 위해 1비트 더 있음
1	Rem = rem – div Rem < 0 restore Shift remainder left, q=0		11110 0111 00000 0111 00000 1110 left shift quotion bit 0
2	Rem = rem – div Rem < 0 Shift remainder left, q=0		11110 1110 00000 1110 00001 1100
3	Rem = rem – div Rem < 0 Shift remainder left, q=0		11111 1100 00001 1100 00011 1000
4	Rem = rem – div Rem >= 0 sign bit이 0이라 양수 Shift remainder left, q=1	sign bit이 0 이제 original remainder를 복구 안해도됨 (e.g.) top 5b 11110+00011 =00001	00001 1000 00001 1000 00011 0001 나눠질 수 있으면 quotion bit 1로 shift
5	Rem = rem – div Rem >= 0 Shift remainder left, q=1	blue - remainder red - quotiant	00001 0001 00001 0001 00010 0011

Optimized Divisor

unsigned division임

0	steps	Divisor 0001 0000 (1)	Remainder 00000 1111 (15)
1	Rem = rem – div Rem < 0 Shift remainder left, q=0		11111 1111 00000 1111 00001 1110
2	Rem = rem – div Rem >= 0 Shift remainder left, q=1		00000 1110 00000 1110 00001 1101
3	Rem = rem – div Rem >= 0 Shift remainder left, q=1		00000 1101 00000 1101 00001 1011
4	Rem = rem – div Rem >= 0 Shift remainder left, q=1		00000 1011 00000 1011 00001 0111
5	Rem = rem – div Rem >= 0 Shift remainder left, q=1		00000 0111 00000 0111 00000 1111

divisor number of bit + 1 만큼 shfit
즉 여기에서는 5단계

To represent 4 bit unsigned number you need 5 bits.

Optimized Divisor

0	steps	Divisor 1111 0000 (15)	Remainder 0000 1110 (14)
1	Rem = rem – div Rem < 0 Shift remainder left, q=0 한 클럭 사이클마다 shift 해야한다	00000 -01111 = 10001	10001 1110 00000 1110 00001 1100
2	Rem = rem – div Rem < 0 Shift remainder left, q=0	e.g.) top 5b 10001+00001= 10010	10010 1100 00001 1100 00011 1000
3	Rem = rem – div Rem < 0 Shift remainder left, q=0	e.g.) top 5b 10001+00011= 10100	10100 1000 00011 1000 00111 0000
4	Rem = rem – div Rem <0 Shift remainder left, q=0	e.g.) top 5b 10001+00111= 11000	11000 0000 00111 0000 01110 0000
5	Rem = rem – div Rem < 0 Shift remainder left, q=0	e.g.) top 5b 10001+01110= 11111	11111 0000 01110 0000 11100 0000

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

계산기의 하드웨어가 int랑 다름

Floating Point

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

한 자릿수 * 지수로 표현

normalized

not normalized

- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- 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

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

S	Exponent	Fraction

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

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: 1.0 ≤ |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

  이 값은 안됨
- Smallest value
  - Exponent: 00000001

    가장 작은 표현지수

    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110

    가장 큰 표현지수

    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$ 이 범위 벗어나면 over,underflow

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 M = 0.1 e-bias = -1
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: $-1 + 127 = 126 = 01111110_2$
    - Double: $-1 + 1023 = 1022 = 01111111110_2$
- Single: 10111111101000…00
- Double: 1011111111101000…00

bias 쓰는 이유
exp는 양수, 음수 다 되는데
exp가 0000000 일 때 0값을 나타내고 싶어서
만약 signed 자리로 쓴다면 exp가 1000000일 때 exp 00000000을 표현하는 값보다 작은데
exp값이 크기 때문에 보기에 햇갈린다.
그래서 그냥 exp를 unsigned로 써서 절댓값을 표현하고 bias로 실제 표현 값을 나타낸다

Floating-Point Example

- What number is represented by the single-precision float

  11000000101000…00

  – S = 1

  – Fraction = $01000\ldots00_2$

  – Fxponent = 10000001_2 = 129

- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

Denormal Numbers

- Exponent = 000…0 ⇒ hidden bit is 0

$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{-\text{Bias}+1}$$

^이게 exp가 000001 일 때보다 작다는 걸 보증

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

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
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$ 작은걸 큰 거에 맞춤

11.1110×2^{-1}

- 2. Add significands

result of big alu

  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-}1 = 0.001_2 \times 2^{-1}$
- 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

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

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} 0이 아닌 한자리가 되도록
- 4. Round and renormalize if necessary
  - 1.021×10^{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
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (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 × –ve ⇒ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875$

FP Arithmetic Hardware

int add, int multi, int div
float add, float multi, float div alu가 cpu에 있음

- 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 float, double용 register가 있음
  - 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
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `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 TargetLabel`

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
```

FP Example: Array Multiplication

- X = X + Y × 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

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]
    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
```

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

Subword Parallellism

- Graphics and audio applications can take advantage of performing simultaneous operations on short vectors
  - Example: 128-bit adder:
    - Sixteen 8-bit adds
    - Eight 16-bit adds
    - Four 32-bit adds

32bit alu를
8bit 등 작은 계산을 하는 걸로 나눠서 병렬 계산

- Also called data-level parallelism, vector parallelism, or Single Instruction, Multiple Data (SIMD)

- In 1966, Michael Flynn
- Flynn Taxonomy – computer classification
  - SISD : single instruction single data
  - SIMD : single instruction multiple data
  - MISD : not popular
  - MIMD : multi-core processor

single core

vector computer
하나의 명령어를
병렬 계산

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

Data transfer	Arithmetic	Compare	Transcendental
FILD mem/ST(i) FISTP mem/ST(i) FLDPI FLD1 FLDZ	FIADDP mem/ST(i) FISUBRP mem/ST(i) FIMULP mem/ST(i) FIDIVRP mem/ST(i) FSQRT FABS FRNDINT	FICOMP FIUCOMP FSTSW AX/mem	FPATAN F2XMI FCOS FPTAN FPREM FPSIN FYL2X

- Optional variations
  - I: integer operand
  - P: pop operand from stack
  - R: reverse operand order
  - But not all combinations allowed

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

dont use shift for division

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

sign of remainder should be same as sign of divident

  - Arithmetic right shift: replicate the sign bit
  - e.g., –5 / 4

The sign of a remainder should be same as that of a dividend.

    - $11111011_2 >> 2 = 11111110_2 = -2$
    - Rounds toward $-\infty$
  - c.f. $11111011_2 >> 2 = 00111110_2 = +62$

Associativity

- Parallel programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail

		(x+y)+z	x+(y+z)
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

- Need to validate parallel programs under varying degrees of parallelism

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

- 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

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