EIE/ENE 334 Microprocessors

Lecture 4:

Arithmetic for Computers

Week #04/05 : Dejwoot KHAWPARISUTH

Adapted from *Computer Organization and Design, 4th Edition,*

Patterson & Hennessy, © 2009, Elsevier (MK)

http://webstaff.kmutt.ac.th/~dejwoot.kha/

Introduction:

- How are negative numbers represented?
- What is the largest number that can be represented by a computer word?
- What happens if an operation creates a number bigger than that can be represented?
- What about fractions and real number?
- How by hardware to add, subtract, multiply, or divide numbers?

Chapter 3

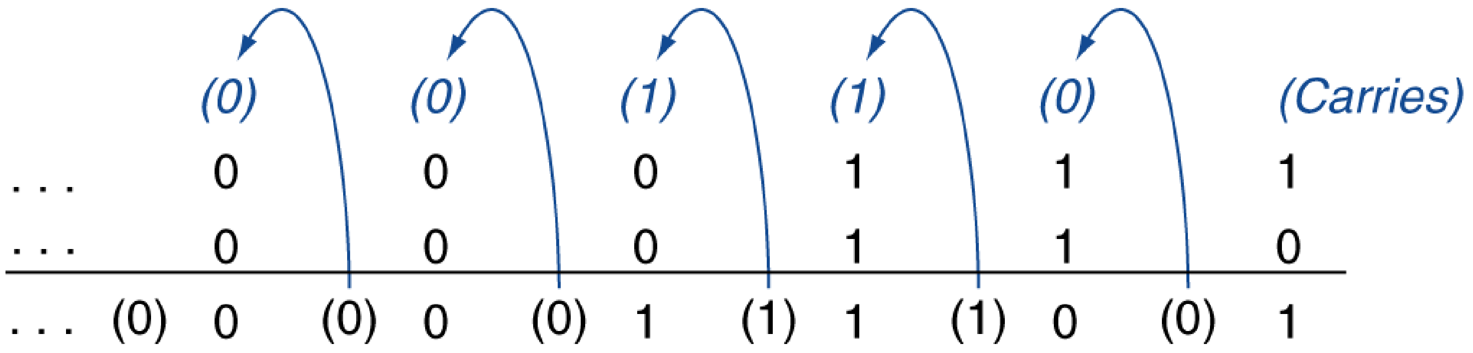
Arithmetic for Computers

Arithmetic for Computers

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

Integer Addition

- Example: 7 + 6



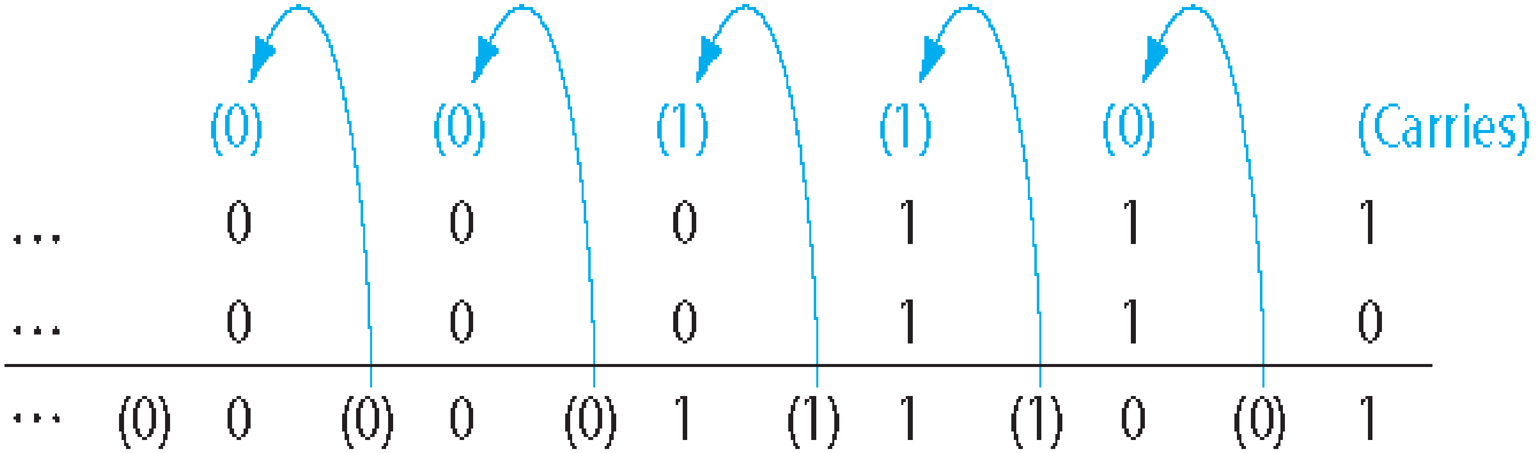
- Overflow if result out of range
  - Adding +ve and –ve operands, no overflow
  - Adding two +ve operands
    - Overflow if result sign is 1
  - Adding two –ve operands
    - Overflow if result sign is 0

Addition and Subtraction:

Example:

```
23ten  =   0001 0111
           1110 1000   # one's complement
                  +1   # two's complement

-23ten =   1110 1001   # two's complement
```



Addition and Subtraction:

Example: Addition

```
 58ten   =      0011 1010
-23ten   =      1110 1001
---------------------------
 35ten   =    1 0010 0011

 26ten   =      0001 1010
-34ten   =      1101 1110
---------------------------
 -8ten   =      1111 1000
```

OK

Addition and Subtraction:

Example: Addition

```
 127ten   =     0111 1111
   2ten   =     0000 0010
---------------------------
 129ten   =     1000 0001 = -127ten

-128ten   =     1000 0000
  -3ten   =     1111 1101
---------------------------
-131ten   =   1 0111 1101 = 125ten
```

Not OK: Overflow! ...How to detect

Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
  - overflow when adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive and get a negative
  - or, subtract a positive from a negative and get a positive
- Consider the operations A + B, and A – B
  - Can overflow occur if B is 0 ?
  - Can overflow occur if A is 0 ?

Overflow conditions:

operation	Operand A	Operand B	Result
A+B	≥ 0	≥ 0	< 0
A+B	< 0	< 0	≥ 0
A-B	≥ 0	< 0	< 0
A-B	< 0	≥ 0	≥ 0

Effects of Overflow

- An exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
- Details based on software system / language
  - example: flight control vs. homework assignment
- Don't always want to detect overflow
  — new MIPS instructions: `addu, addiu, subu`

note: `addiu` *still sign-extends!*
note: `sltu, sltiu` *for unsigned comparisons*

MIPS: exception

MIPS detect overflow with exception. The address of the instruction that overflowed is saved in a register and the computer jumps to a predefined address to invoke the appropriate routine for that exception

EPC: Exception Program Counter contains the address of the instruction that cause the exception

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

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

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

Questions?:

[3.10] Page 230

Find the shortest sequence of MIPS instructions to determine if there is a carry out from the addition of two registers, say, register $t3 and $t4. Place a 0 or 1 in register $t2 if the carry out is 0 or 1, respectively. (Hint: It can be done in two instructions

Questions?:

[3.10]

Sol:

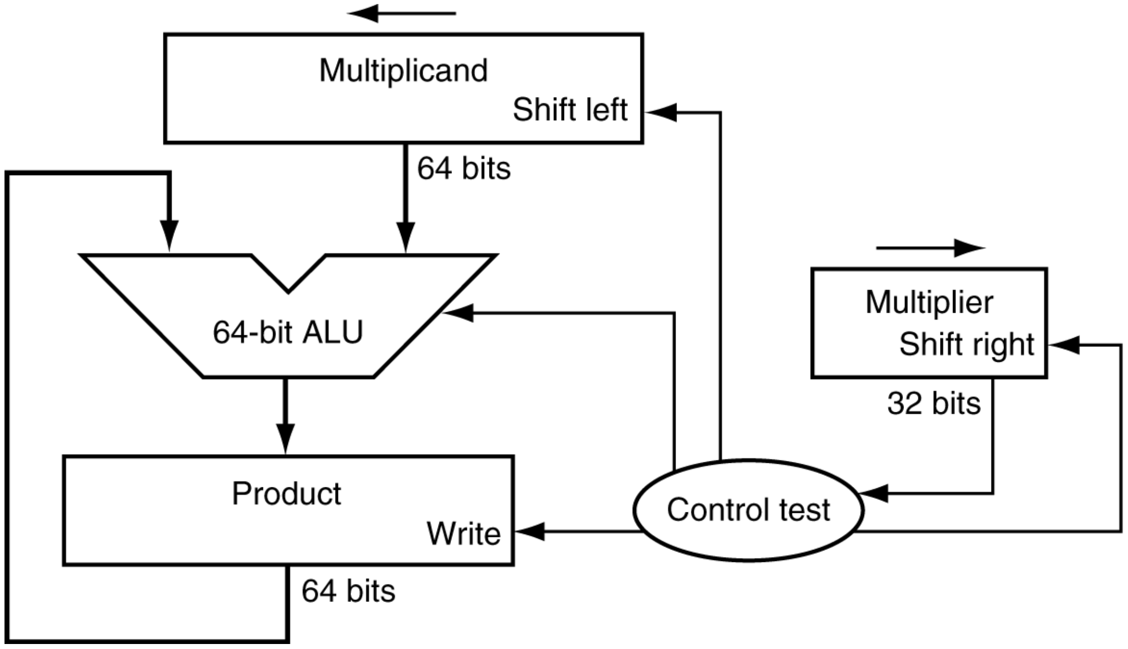
```
  addu $t2,$t3,$t4
  sltu $t2,$t2,$t4
or
  addu $t2,$t3,$t4
  sltu $t2,$t2,$t3
```

Multiplication

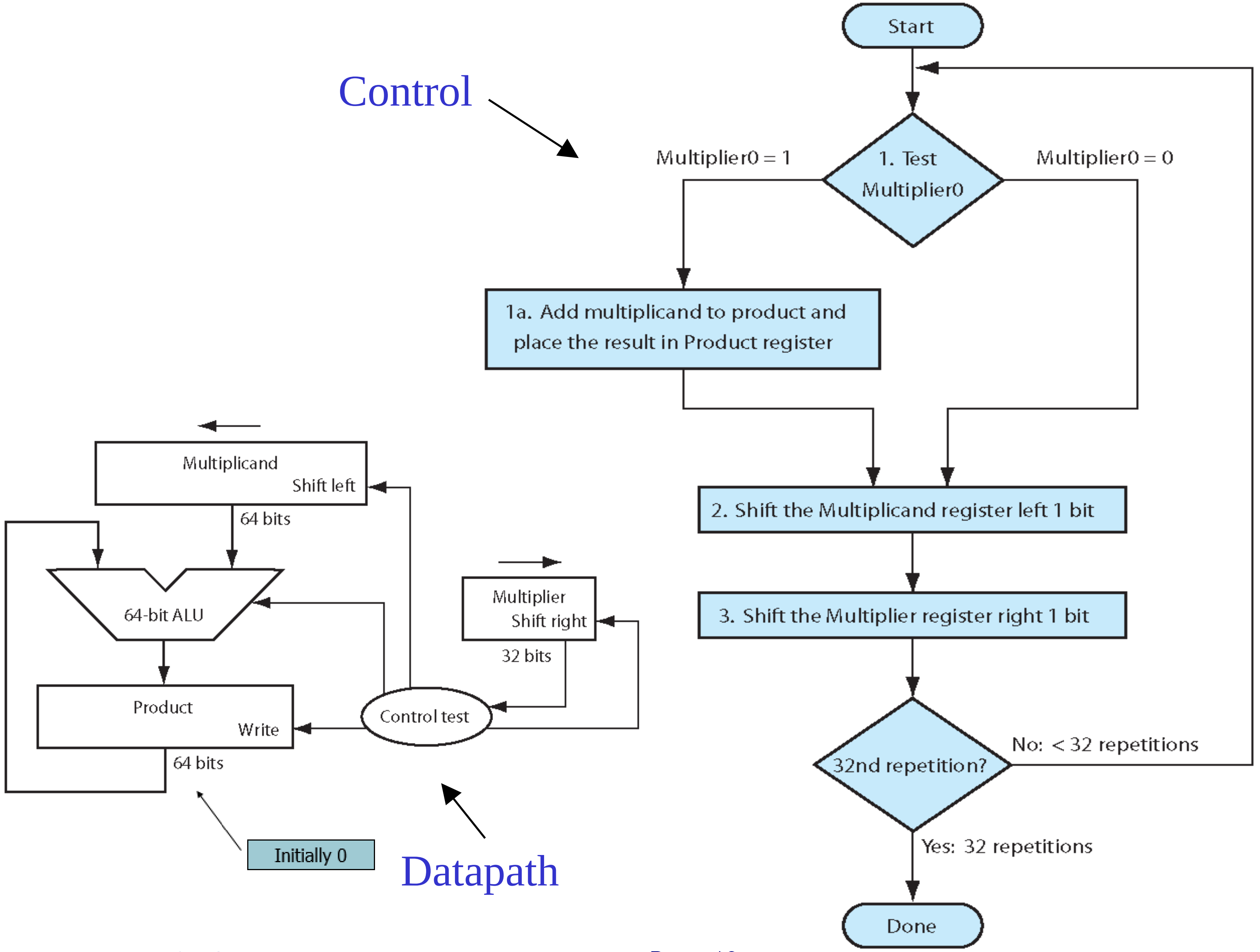
- Start with long-multiplication approach

```
multiplicand →    1000
multiplier   →  × 1001
                  ----
                  1000
                 0000
                0000
               1000
               -------
product      → 1001000
```

Length of product is the sum of operand lengths



Multiplication: hardware I

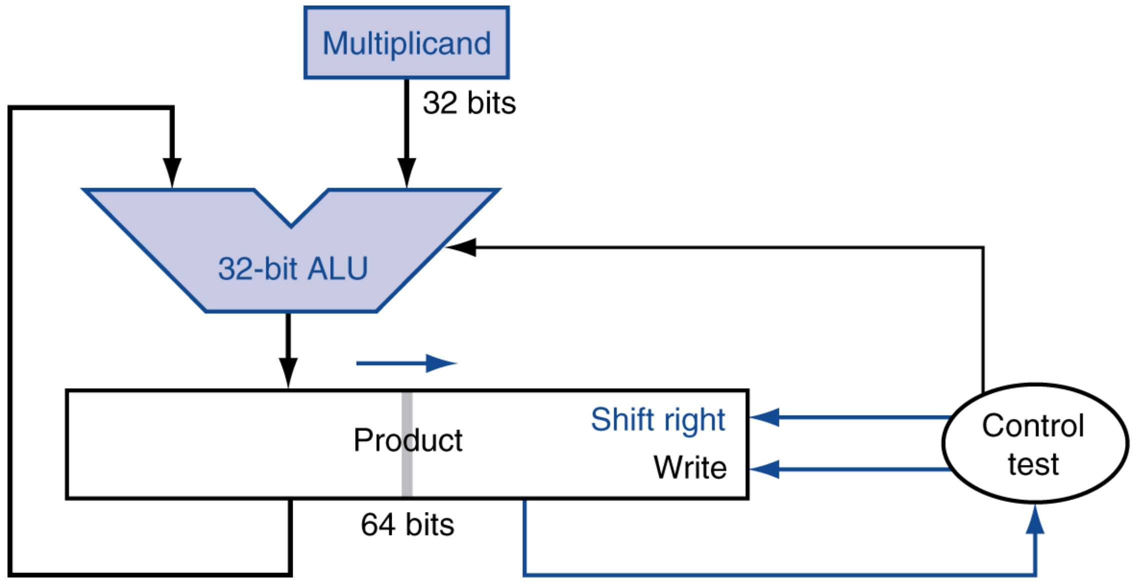


Multiplication: Example $2_{ten} \times 3_{ten}$

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

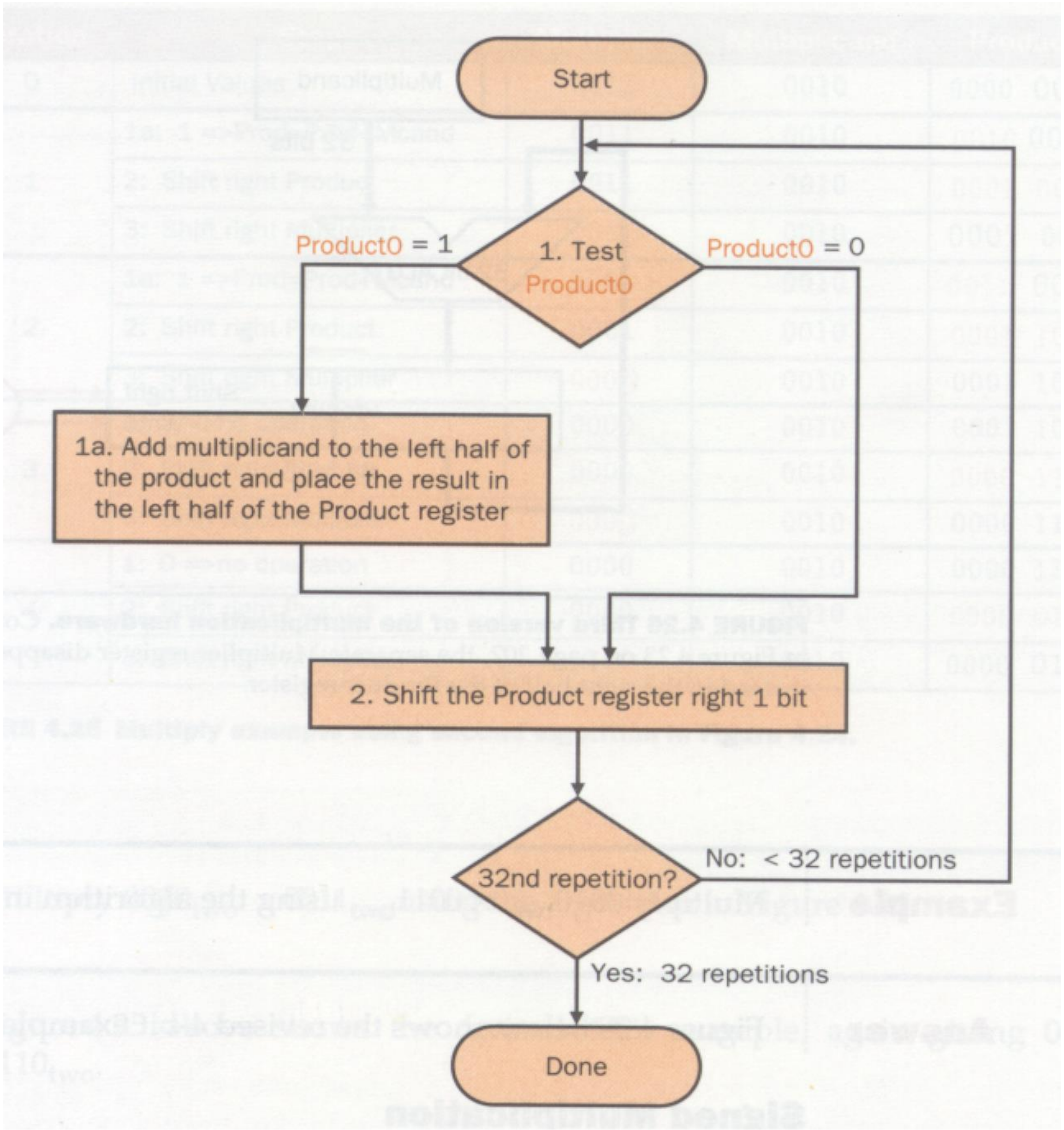
Optimized Multiplier

- Perform steps in parallel: add/shift



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

Multiplication: algorithm II


Start
1. Test Product0
Product0 = 1
Product0 = 0
1a. Add multiplicand to the left half of the product and place the result in the left half of the Product register
2. Shift the Product register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

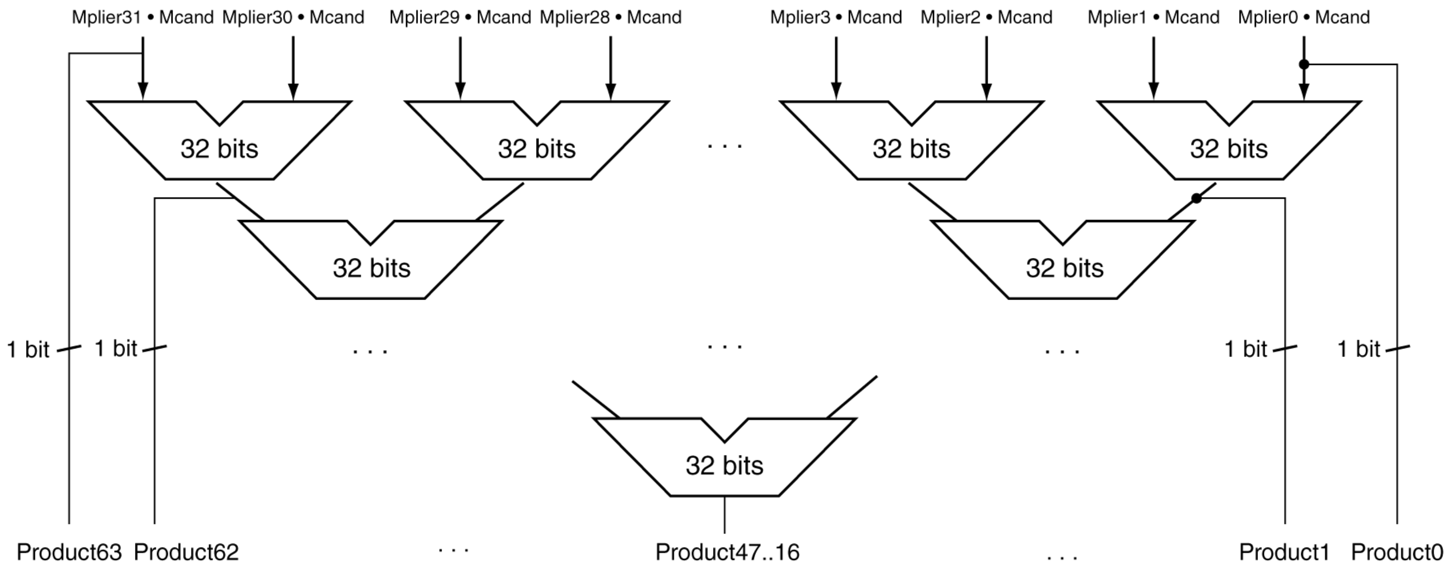
Multiplication: example II

$2_{ten} \times 3_{ten}$

Iteration	Step	Multiplicand (Md)	Product (P)
0	Initial values	0010	0000 0011
1	1a: 1 > P=P+Md	0010	0010 0011
	2: Shift right P.	0010	0001 0001
2	1a: 1 > P=P+Md	0010	0011 0001
	2: Shift right P.	0010	0001 1000
3	1: 0 > no operation	0010	0001 1000
	2: Shift right P.	0010	0000 1100
4	1: 0 > no operation	0010	0000 1100
	2: Shift right P.	0010	0000 0110

Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff



- Can be pipelined
  - Several multiplication performed in parallel

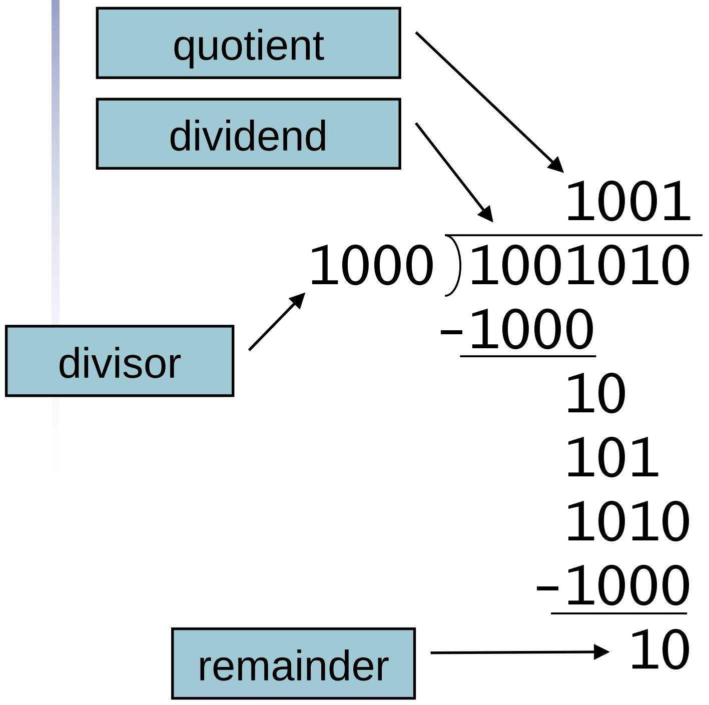
MIPS Multiplication

- 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

MIPS: Multiplication

Instruction	Example	Meaning
multiply	mult \$s2,\$s3	Hi,Lo = \$s2 x \$s3
multiply unsigned	mult \$s2,\$s3	Hi,Lo = \$s2 x \$s3
mov from Hi	mfhi \$s1	\$s1 = Hi
mov from Lo	mflo \$s1	\$s1 = Lo

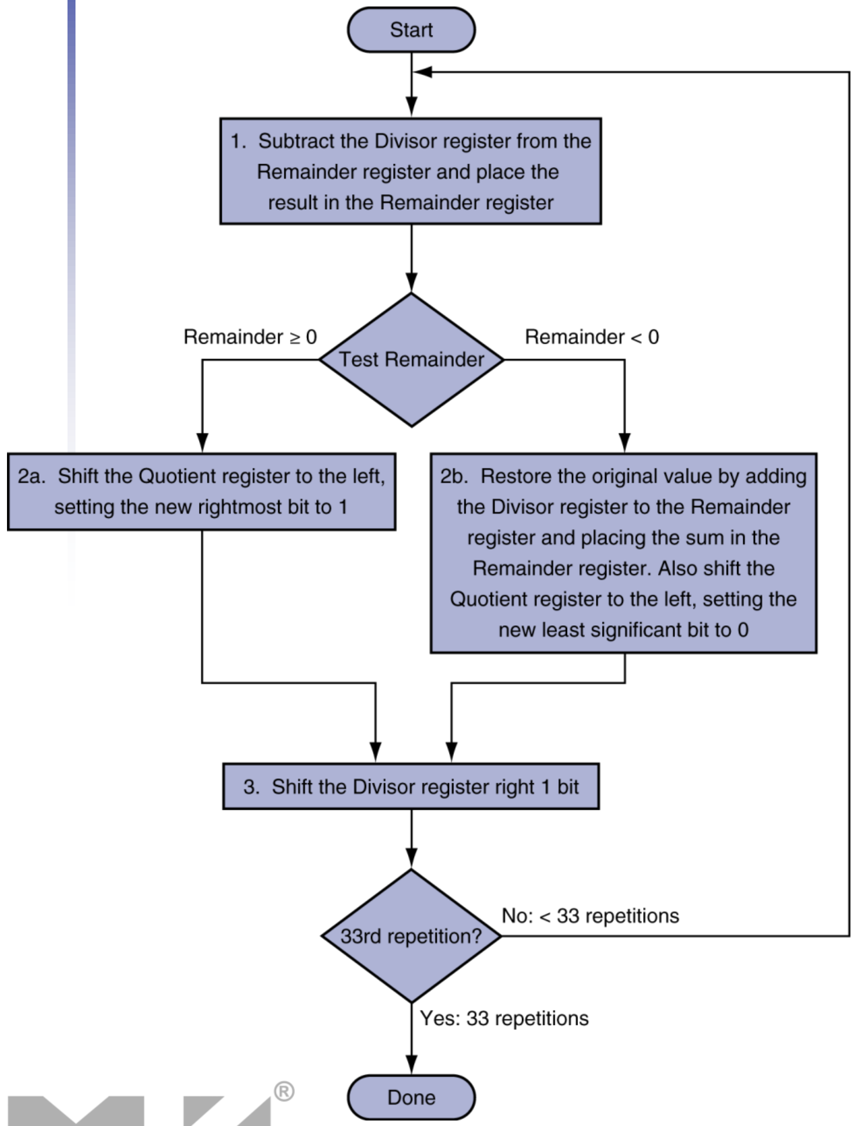
Division



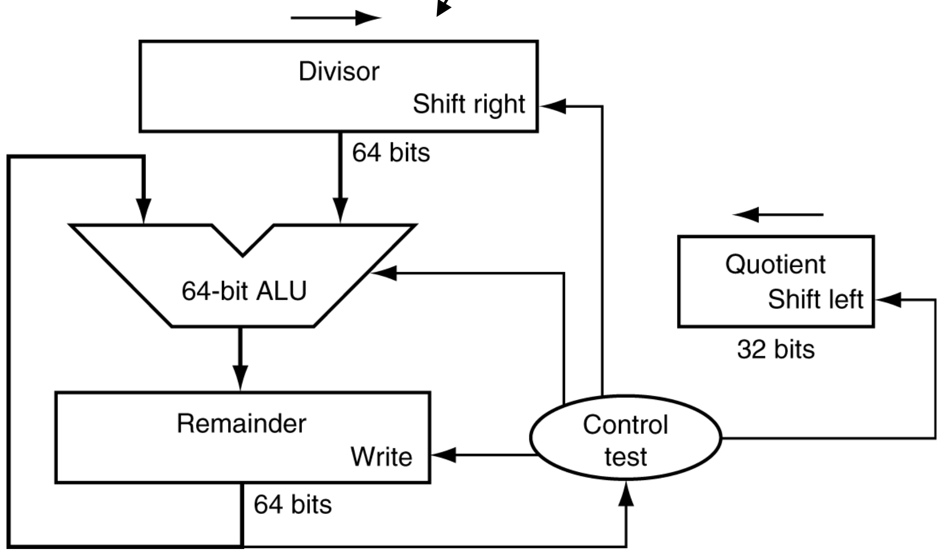
n-bit operands yield *n*-bit quotient and remainder

- 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
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

Division Hardware



Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
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
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done



Initially divisor in left half

Initially dividend

Division: Example

7_{ten} by 2_{ten}

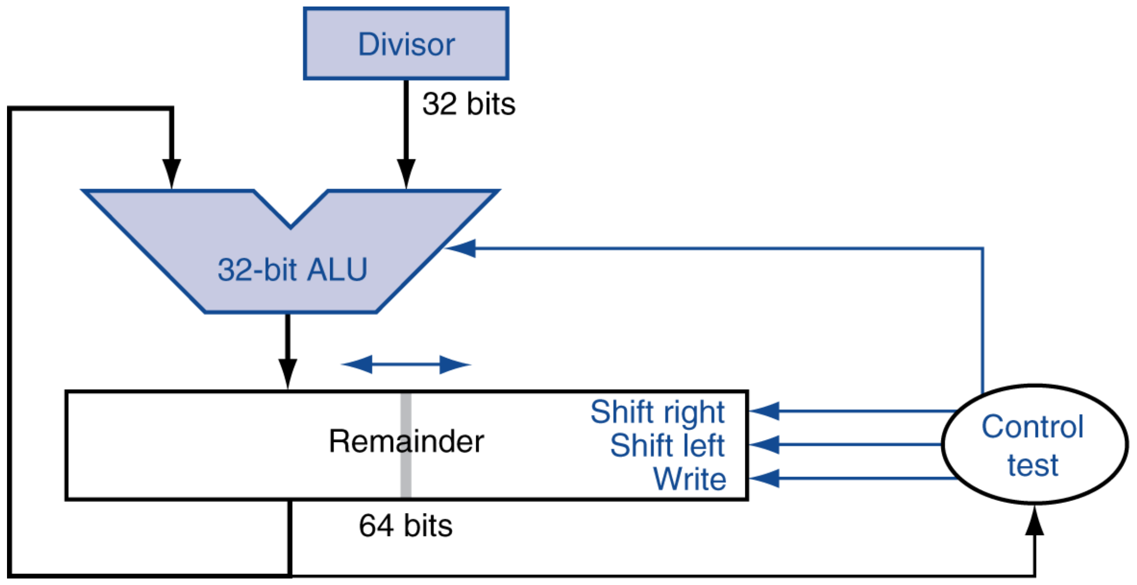
Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem=Rem-Div	0000	0010 0000	1110 0111
	2b: Rem<0,+Div,sll Q,Q0=0	0000	0010 0000	0000 0111
	3:shift Div Right	0000	0001 0000	0000 0111
2	1: Rem=Rem-Div	0000	0001 0000	1111 0111
	2b: Rem<0,+Div,sll Q,Q0=0	0000	0001 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

Division: Example

7_{ten} by 2_{ten}

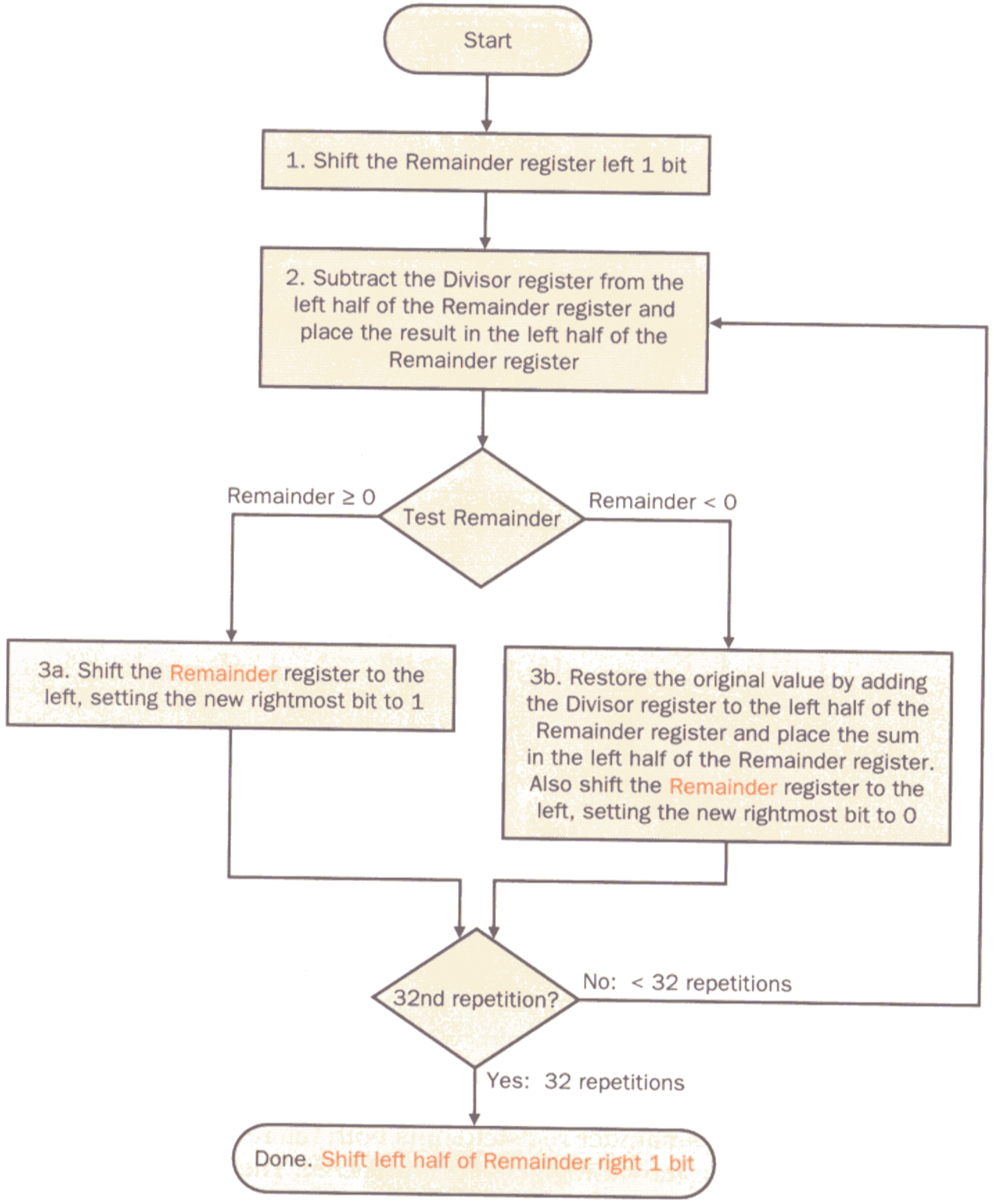
Iteration	Step	Quotient	Divisor	Remainder
3	1: Rem=Rem-Div	0000	0000 1000	1111 1111
	2b: Rem<0,+Div,sll Q,Q0=0	0000	0000 1000	0000 0111
	3:shift Div Right	0000	0000 0100	0000 0111
4	1: Rem=Rem-Div	0000	0000 0100	0000 0011
	2a: Rem>=0,sll Q,Q0=1	0001	0000 0100	0000 0011
	3:shift Div Right	0001	0000 0010	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

Optimized Divider



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

Division: algorithm II


Start
1. Shift the Remainder register left 1 bit
2. Subtract the Divisor register from the left half of the Remainder register and place the result in the left half of the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
3a. Shift the Remainder register to the left, setting the new rightmost bit to 1
3b. Restore the original value by adding the Divisor register to the left half of the Remainder register and place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new rightmost bit to 0
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done. Shift left half of Remainder right 1 bit

Division: Example

7_{ten} by 2_{ten}

Iteration	Step	Divisor	Remainder
0	Initial values	0010	0000 0111
	Shift Rem left 1	0010	0000 1110
1	2: Rem=Rem-Div	0010	1110 1110
	3b: Rem<0,+Div,sll R,R0=0	0010	0001 1100
2	2: Rem=Rem-Div	0010	1111 1100
	3b: Rem<0,+Div,sll R,R0=0	0010	0011 1000
3	1: Rem=Rem-Div	0010	0001 1000
	3a: Rem>=0,sll R,R0=1	0010	0011 0001
4	1: Rem=Rem-Div	0010	0001 0001
	3a: Rem>=0,sll R,R0=1	0010	0010 0011
	Shift left half Rem Right 1	0010	0001 0011

Faster Division

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

Division: Signed Division

Dividend = Quotient x Divisor + Remainder

Ex: +7 ÷ +2: Quotient = +3, Remainder = +1

Checking: 7 = 3 x 2 +1

Ex: -7 ÷ +2: Quotient = -3, Remainder = ...

Remainder = Dividend - (Quotient x Divisor) = -7-(-3 x +2)=-1

Checking: -7 = -3 x 2 +(-1)

or Quotient = -4, Remainder = +1 ???

Division: Signed Division

Dividend = Quotient x Divisor + Remainder

Ex: -7 ÷ +2: Quotient = -4, Remainder = +1

Checking: -7 = -4 x 2 +(+1)

but ...

the dividend and remainder must have the same signs.

Division: Signed Division

Dividend = Quotient x Divisor + Remainder

Ex: +7 ÷ -2: Quotient = ...,

Remainder = ...

Ex: -7 ÷ -2: Quotient = ...,

Remainder = ...

Division: Signed Division

Dividend = Quotient x Divisor + Remainder

Ex: +7 ÷ -2: Quotient = -3,

Remainder = +1

Checking: +7 = -3 x -2 +(+1)

Ex: -7 ÷ -2: Quotient = +3,

Remainder = -1

Checking: -7 = +3 x -2 +(-1)

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

MIPS: Division

Instruction	Example	Meaning

divide	div $s2,$s3	Lo = $s2 / $s3,
		Hi = $s2 mod $s3

divide unsigned	div $s2,$s3	Lo = $s2 / $s3,
		Hi = $s2 mod $s3

mov from Hi	mfhi $s1	$s1 = Hi
mov from Lo	mflo $s1	$s1 = Lo

Floating Point (a brief look)

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g., 3.15576×10^9
- Representation:
  - sign, exponent, significand: $(-1)^{\text{sign}} \times \text{significand} \times 2^{\text{exponent}}$
  - more bits for significand gives more accuracy
  - more bits for exponent increases range
- IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit significand
  - double precision: 11 bit exponent, 52 bit significand

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56} normalized
  - $+0.002 \times 10^{-4}$ not normalized
  - $+987.02 \times 10^{9}$
- 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 = 1203

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ 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
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 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
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

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

FP: Floating-Point Representation

three advantages: (normalized form)

- simplifies exchange of data
- simplifies the floating point arithmetic algorithms
- increases the accuracy of the numbers that can be stored in a word

Binary numbers in scientific notation:

$1.XXX_{two} \times 2^{YYYY}$

Ex: $1.0_{two} \times 2^{-1} = 0.5_{ten}$

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 + Bias
    - Single: –1 + 127 = 126 = 01111110_2
    - Double: –1 + 1023 = 1022 = 01111111110_2
- Single: 1011111101000…00
- Double: 1011111111101000…00

Floating-Point Example

- What number is represented by the single-precision float

  11000000101000…00

  - S = 1
  - Fraction = $01000\ldots00_2$
  - Fxponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$

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

  $= -5.0$

FP: Floating-Point Representation

IEEE 754: (sign and magnitude representation)

$$(-1)^S \times (1+F) \times 2^E$$

$$\rightarrow (-1)^S \times (1+(s1 \times 2^{-1})+(s2 \times 2^{-2})+(s3 \times 2^{-3})+\ldots) \times 2^E$$

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	+- denormalized number
1-254	Anything	1-2046	Anything	+- floating-point number
255	0	2047	0	+- infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

FP: Exponent

IEEE 754: uses a bias of 127 for single precision.

Ex: -1 represented by $-1+127_{ten}$ or $126_{ten} = 0111\ 1110_{two}$

+1 represented by $+1+127_{ten}$ or $128_{ten} = 1000\ 0000_{two}$

and a bias 1023 for double precision.

IEEE 754: (sign and magnitude representation)

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

Why bias notation?, overflow, underflow

FP: Floating-Point Representation

Why bias notation?

Negative exponents pose a challenge to simplified sorting. The desirable notation must therefore represent the most negative exponent as $00...00_{two}$ and the most positive as $11...11_{two}$. This is called biased notation.

FP: Floating-Point Representation

IEEE 754: (sign and magnitude representation)

$$(-1)^S \times (1+F) \times 2^E$$

A single precision floating point: $2.0_{ten} \times 10^{-38} \leftrightarrow 2.0_{ten} \times 10^{38}$

31	30 29 28 27 26 25 24 23	22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
s	Exponent (E)	Fraction (F)
	8 bits	23 bits

A double precision floating point: $2.0_{ten} \times 10^{-308} \leftrightarrow 2.0_{ten} \times 10^{308}$

31	30 29 28 27 26 25 24 23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
s	exponent	fraction
	11 bits	20 bits

fraction (continued)

FP: Example

Ex: Show the IEEE 754 binary representation of -0.75_{ten} in single and double precision.

Sol: $-0.75_{ten} = -0.11_{two} = -1.1_{two} \times 2^{-1}$

a single precision number: $(-1)^S \times (1+F) \times 2^{(\text{Exponent}-127)}$

$-1.1_{two} \times 2^{-1}$ (Normalized scientific notation)

$= (-1)^1 \times (1+.1000\ 0000\ 0000\ 0000\ 0000\ 000) \times 2^{(126-127)}$

31	30 29 28 27 26 25 24 23	22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
s	exponent	fraction
	126	0.1
1	0 1 1 1 1 1 1 0	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

FP: Example

a double precision number: $(-1)^S \times (1+F) \times 2^{(\text{Exponent}-1023)}$

$-1.1_{\text{two}} \times 2^{-1}$

$= (-1)^1 \times (1+.1000\ 0000\ 0000\ 0000\ 0000\ \ldots) \times 2^{(1022-1023)}$

31	30 29 28 27 26 25 24 23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
s	exponent	fraction

1022 0.1

1	0 1 1 1 1 1 1 1 1 1 0	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
fraction (continued)		
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		

FP: Example

Ex: What decimal number is represented by this number?

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Sol:

a single precision number: $(-1)^S \times (1+F) \times 2^{(\text{Exponent}-127)}$

$= (-1)^1 \times (1+.25) \times 2^{(129-127)}$

$= -1.25 \times 2^2$

$= -5.0$

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

FP: Addition 4-binary digit

Ex: add 0.5_{ten} to -0.4375_{ten}

$0.5_{ten} = 1.000_{two} \times 2^{-1}$

$-0.4375_{ten} = -1.110_{two} \times 2^{-2}$

Step 1: align the binary point of the number with the smaller exponent.

$-1.110_{two} \times 2^{-2} = -0.111_{two} \times 2^{-1}$

Step 2: the addition of the significands:

$1.000_{two} \times 2^{-1} + (-0.111_{two} \times 2^{-1}) = 0.001_{two} \times 2^{-1}$

FP: Addition

Step 3: put it into normalized form and check for overflow or underflow

$0.001_{two} \times 2^{-1} = 1.000_{two} \times 2^{-4}$

Since $-126 \leq -4 \leq 127$, there is no overflow or underflow.

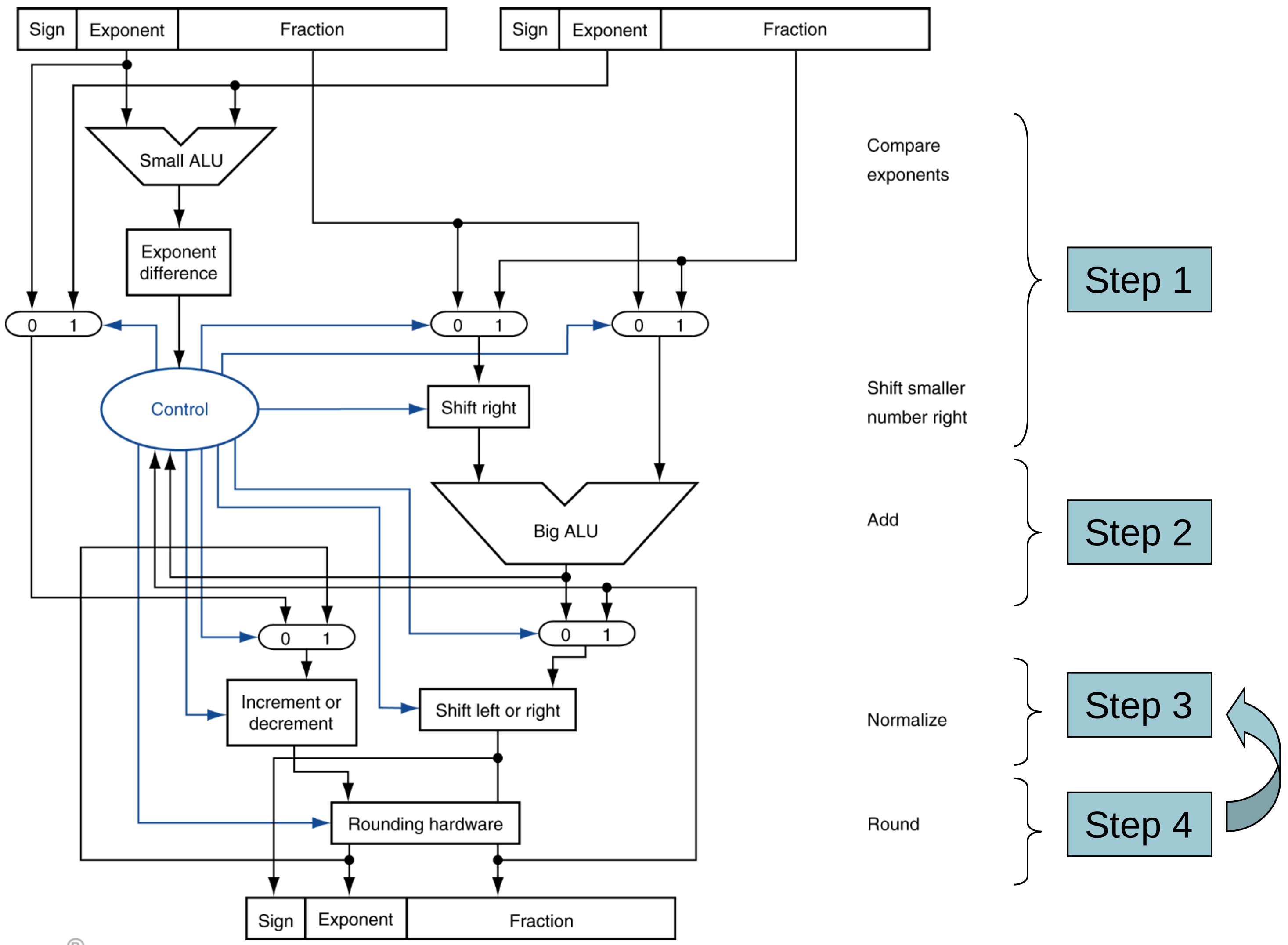
Step 4: round the number

$1.000_{two} \times 2^{-4}$

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



FP: Multiplication

Ex: $(1.000_{two} \times 2^{-1})\times(1.110_{two} \times 2^{-2})$

Step 1: adding the exponents.

$-1+(-2) = -3$

with the bias notation: $-1+127 = 126$ and $-2+127 = 125$

adding the exponents: $126+125-127 = 124 \rightarrow -3$

FP: Multiplication

Step 2: the multiplication of the significants:

```
     1000
   x 1110
     0000
    1000
   1000
  1000
  1110000
```

$1110000_{two} \rightarrow 1.110000_{two} = 1.110_{two} \times 2^{-3}$

FP: Multiplication

Step 3: normalize, check for overflow or underflow

(127 ≥ -3 ≥ -126) or (bias notation: 254 ≥ 124 ≥ 1)

$1.110_{two} \times 2^{-3}$

Step 4: assume that the significant is only 4 digits, so round the number

$1.110_{two} \times 2^{-3}$

Step 5: sign the product:

$-1.110_{two} \times 2^{-3}$

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

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
  - `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)
     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
```

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
```

FP: Accurate Arithmetic

Rounding with Guard Digits:

Ex: $2.56_{ten} \times 10^0 + 2.34_{ten} \times 10^2$

with guard and round bit:

$$\begin{array}{rr} & 2.3400 \\ + & \underline{0.0256} \\ & 2.3656 \end{array}$$

□□ $\rightarrow$ 2.37

without:

$$\begin{array}{rr} & 2.34 \\ + & \underline{0.02} \\ & 2.36 \end{array}$$

MIPS: FP operands

MIPS floating point Operands

Name	Example	Comments
32 registers	\$f0, \$f1, \$f2,..., \$f31	MIPS floating-point registers are used in pairs for double precision numbers
2^{30} memory words	Memory[0], Memory[4],..., Memory [4293967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls

MIPS: FP assembly language

Arithmetic

Instruction	Example	Meaning	Comments
FP add single	add.s $f2,$f4,$f6	$f2=$f4+$f6	FP add(single precision)
FP subtract single	sub.s $f2,$f4,$f6	$f2=$f4-$f6	FP sub(single precision)
FP multiply single	mul.s $f2,$f4,$f6	$f2=$f4x$f6	FP mul(single precision)
FP divide single	div.s $f2,$f4,$f6	$f2=$f4/$f6	FP div(single precision)
FP add double	add.d $f2,$f4,$f6	$f2=$f4+$f6	FP add(double precision)
FP subtract double	sub.d $f2,$f4,$f6	$f2=$f4-$f6	FP sub(double precision)
FP multiply double	mul.d $f2,$f4,$f6	$f2=$f4x$f6	FP mul(double precision)
FP divide double	div.d $f2,$f4,$f6	$f2=$f4/$f6	FP div(double precision)

MIPS: FP assembly language

Data transfer & Conditional branch

Instruction	Example	Meaning	Comments
load word copr.1	lwc1 $f1,100($s2)	$f1=Memory[$s2+100]	32-bit data to FP register
store word copr.1	swc1 $f1,100($s2)	Memory[$s2+100]=$f1	32-bit data to memory
branch on FP true	bclt 25	If (cond==1) goto PC+4+100	PC-relative branch if FP cond.
branch on FP false	bclf 25	If (cond==0) goto PC+4+100	PC-relative branch if not cond.
FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s $f2,$f4	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than single precision
FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d $f2,$f4	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than single precision

Interpretation of Data

The BIG Picture

- 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

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

Floating Point Complexities

- Operations are somewhat more complicated (see text)
- In addition to overflow we can have “underflow”
- 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
- Implementing the standard can be tricky
- Not using the standard can be even worse
  - see text for description of 80x86 and Pentium bug!

Chapter Three Summary

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

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)`	`FIADDP  mem/ST(i)`	`FICOMP`	`FPATAN`
`FISTP mem/ST(i)`	`FISUBRP mem/ST(i)`	`FIUCOMP`	`F2XMI`
`FLDPI`	`FIMULP  mem/ST(i)`	`FSTSW AX/mem`	`FCOS`
`FLD1`	`FIDIVRP mem/ST(i)`		`FPTAN`
`FLDZ`	`FSQRT`		`FPREM`
	`FABS`		`FPSIN`
	`FRNDINT`		`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

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
    - 11111011_2 >> 2 = 11111110_2 = –2
    - Rounds toward –∞
  - c.f. 11111011_2 >>> 2 = 00111110_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