Chapter 3

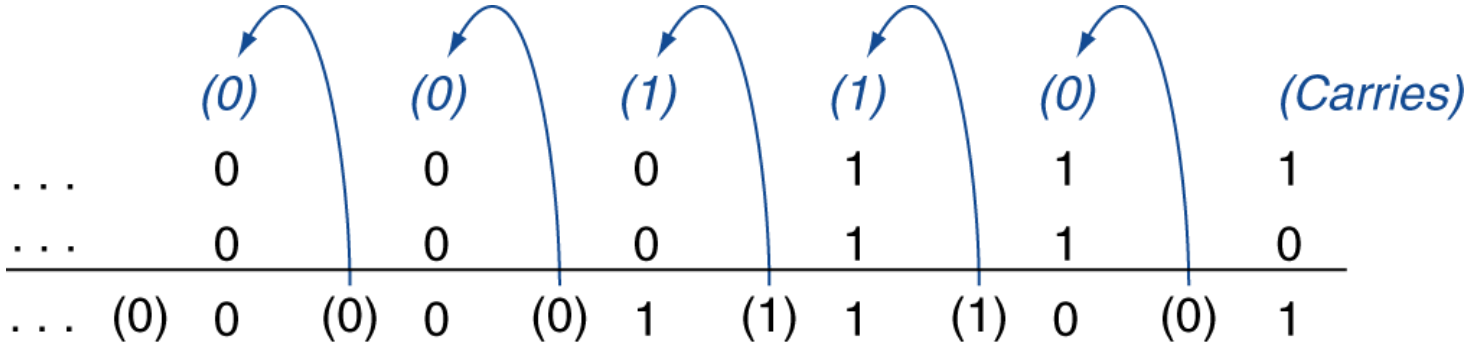
Arithmetic for Computers

Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
- 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

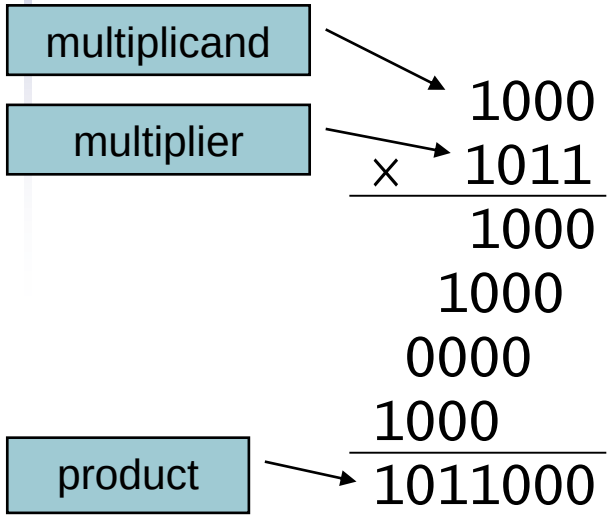
- Overflow detection

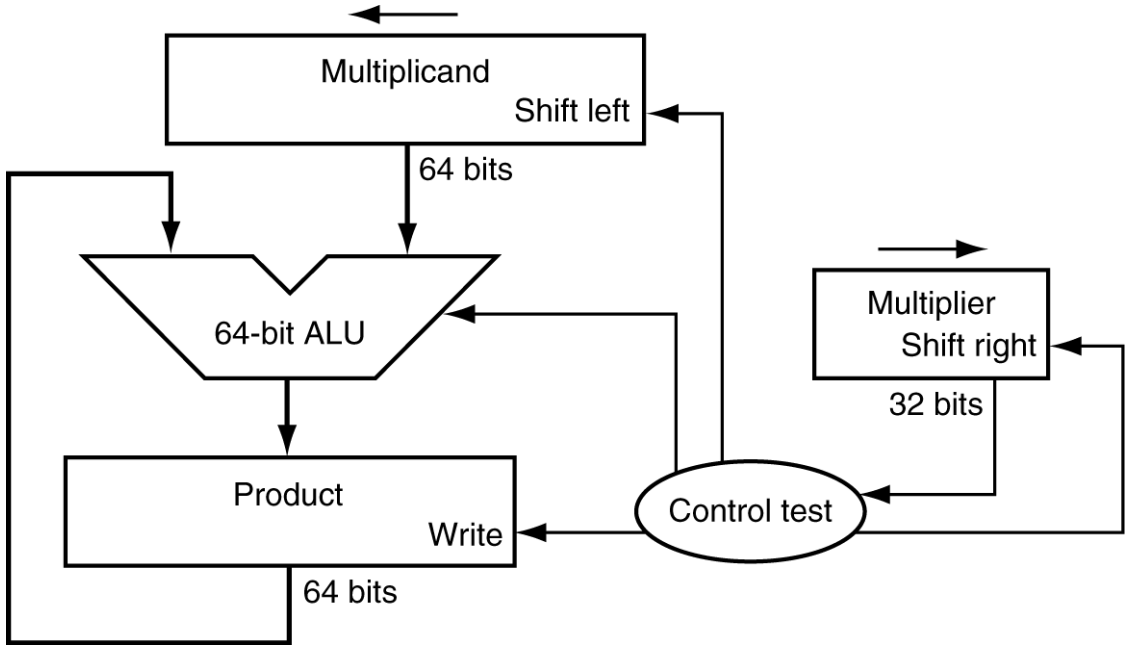
Operation	Operand A	Operand B	Result indicating overflow
A+B	≥ 0	≥ 0	< 0
A+B	< 0	< 0	≥ 0
A-B	≥ 0	< 0	< 0
A-B	< 0	≥ 0	≥ 0

- Some languages (e.g., C) ignore overflow
  - Use MIPS `addu`, `addui`, `subu` instructions
- Other languages (e.g., Ada) raise 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
    - retrieve EPC value, to return after corrective action

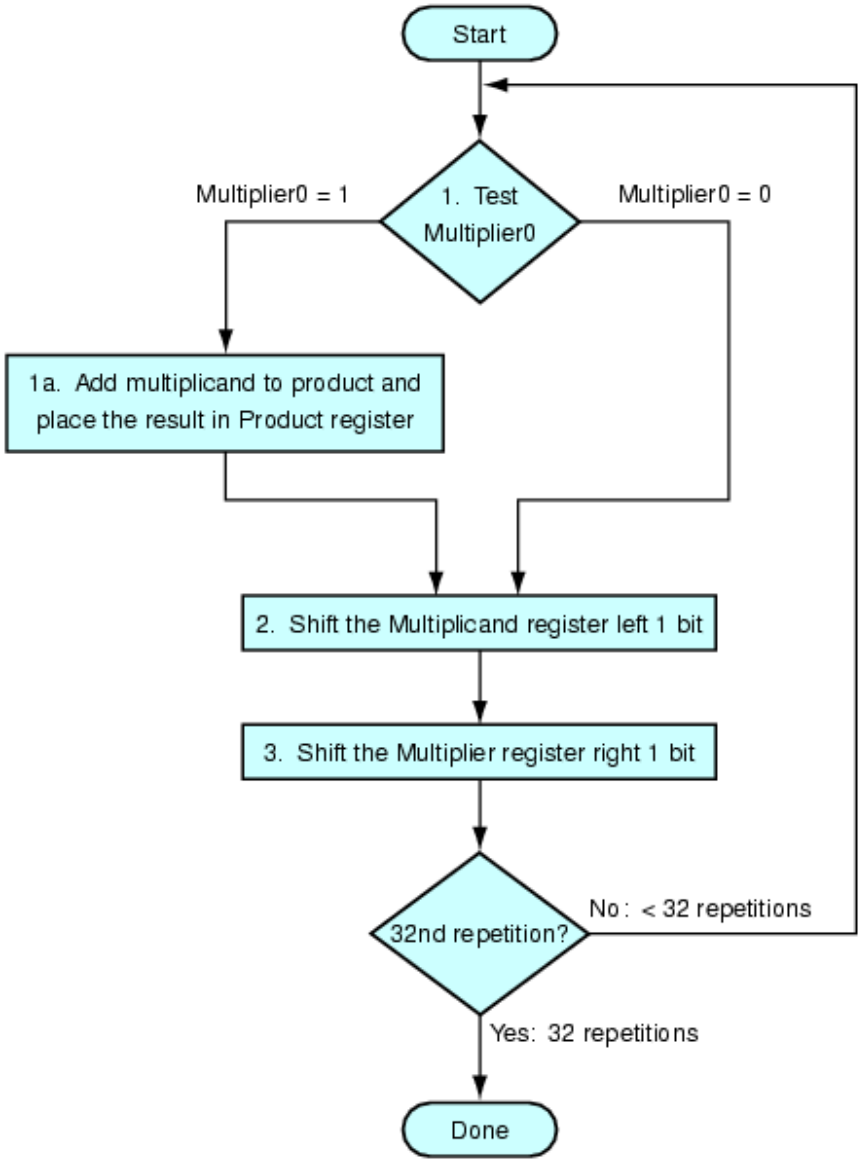
Multiplication

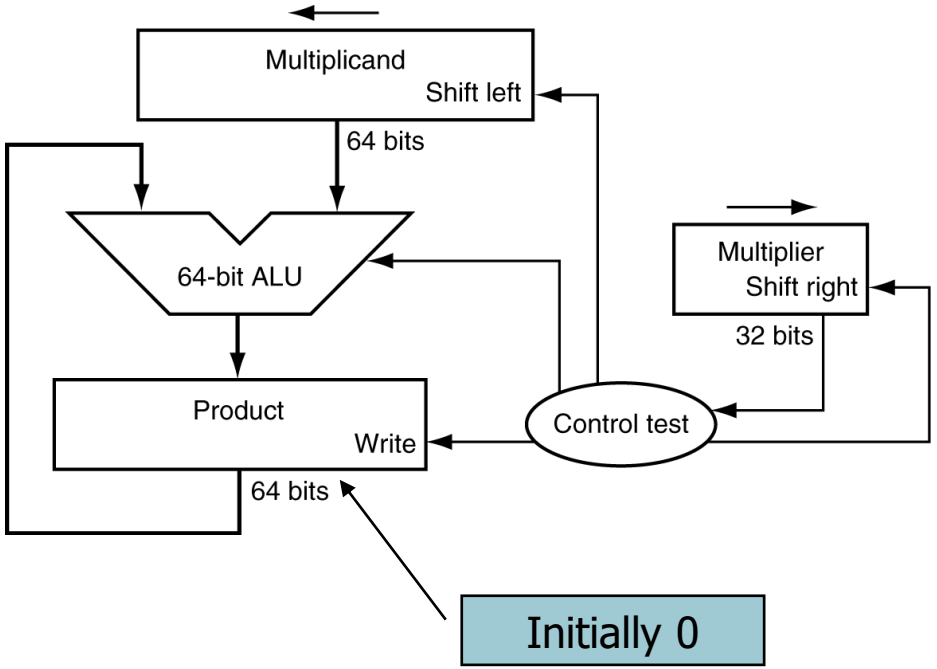
- Start with long-multiplication approach





Multiplication Hardware





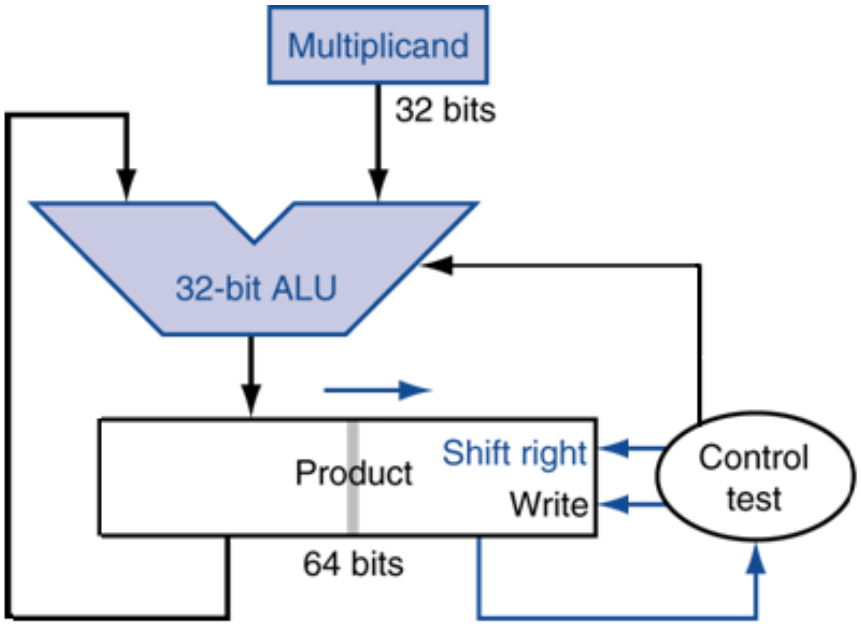
Example

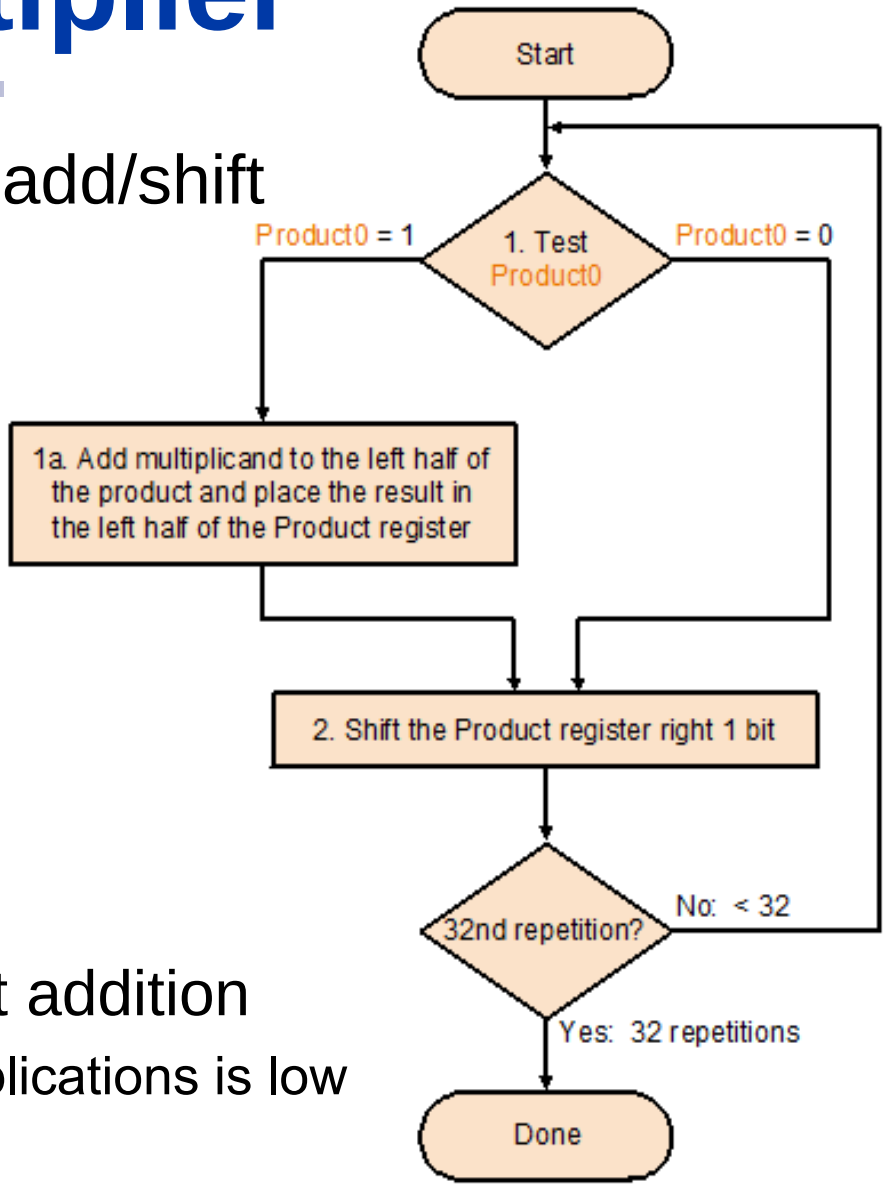
- Using 4-bit numbers to save space, multiply $2_{ten} \times 3_{ten}$; or $0010_{two} \times 0011_{two}$

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

Example

- Multiply $0010_{two} \times 0011_{two}$ using optimized multiplier hardware

Iteration	Step	Multiplicand	Product
0	Initial values	0010	0000 0011
1	1a: 1 => Prod = Prod + Mcand	0010	0010 0011
	2: Shift right Product	0010	0001 0001
2	1a: 1 => Prod = Prod + Mcand	0010	0011 0001
	2: Shift right Product	0010	0001 1000
3	1: 0 => no operation	0010	0001 1000
	2: Shift right Product	0010	0000 1100
4	1: 0 => no operation	0010	0000 1100
	2: Shift right Product	0010	0000 0110

Signed Multiplication

- The simplest approach:

  Negate all negative operands at the beginning, perform unsigned multiplication on the resulting numbers, and then negate the product if necessary.

  - Disadv:
    - Extra clock cycles may be needed to negate multiplicand, multiplier, and the double length product.

Booth's Algorithm

- E.g.: $2_{10} \times 6_{10} = 0010_2 \times 0110_2$

$6 = 0110_2$

0010_{two}
x 0110_{two}

+	0000	shift (0 in multiplier)
+	0010	add (1 in multiplier)
+	0010	add (1 in multiplier)
+	0000	shift (0 in multiplier)
	00001100_{two}	

$6 = -2 + 8 = -0010_2 + 1000_2$

0010_{two}
x 0110_{two}

+	0000	shift (0 in multiplier)
-	0010	sub (first 1 in multiplier)
+	0000	shift (middle of string of 1s)
+	0010	add (prior step had last 1)
	00001100_{two}	

- Consider $01110_2 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1$ (three additions)
- Faster calculation
  - $01110_2 = 1 \times 2^4 - 1 \times 2^1$ (one addition and one subtraction)

$14 = 16 - 2$

9 8 7 6 5 4 3 2 1 0

00**1111111**000 ➔ ?

0011111111111 ➔ $2^9 - 1$

– 111 ➔ $2^3 - 1$

00**1111111**000 ➔ $(2^9-1) - (2^3-1) = 2^9 - 2^3$

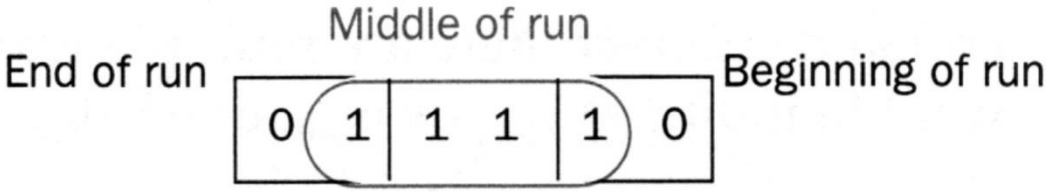
m n

000**1**111111111**1**000000

➔ $2^{m+1} - 2^n$

Booth's Algorithm

- The key to Booth's insight:
  - classify groups of bits into the beginning, the middle, or the end of a run of 1s



Current bit	Bit to the right	Explanation	Example
1	0	Beginning of a run of 1s	0000111100_{two}
1	1	Middle of a run of 1s	0000111000_{two}
0	1	End of a run of 1s	0000111000_{two}
0	0	Middle of a run of 0s	0000111000_{two}

Booth's Algorithm

- Booth's algorithm
  1. Depending on the current and previous bits, do one of the following:

     00: Middle of a string of 0s ⇒ **no** arithmetic op

     01: End of a string of 1s ⇒ **add** the multiplicand to the left half of the product

     10: Beginning of a string of 1s ⇒ **sub** the multiplicand from the left half of the product

     11: Middle of a string of 1s ⇒ **no** arithmetic op

  2. Shift the Product register right 1 bit

Booth's Algorithm

- Requirements:
  - Start with a 0 for the bit to the right of the rightmost bit
  - Booth's ops is identified according to the values in 2 bits.
  - Extend the sigh when the product is shifted to the right.

E.g., $2_{10} \times 6_{10} = 0010_2 \times 0110_2$

Sign extension

Iteration	Multiplicand	Original algorithm		Booth's algorithm	
		Step	Product	Step	Product
0	0010	Initial values	0000 0110	Initial values	0000 0110 0
1	0010	1: 0 ⇒ no operation	0000 0110	1a: 00 ⇒ no operation	0000 0110 0
	0010	2: Shift right Product	0000 0011	2: Shift right Product	0000 0011 0
2	0010	1a: 1 ⇒ Prod = Prod + Mcand	0010 0011	1c: 10 ⇒ Prod = Prod – Mcand	1110 0011 0
	0010	2: Shift right Product	0001 0001	2: Shift right Product	1111 0001 1
3	0010	1a: 1 ⇒ Prod = Prod + Mcand	0011 0001	1d: 11 ⇒ no operation	1111 0001 1
	0010	2: Shift right Product	0001 1000	2: Shift right Product	1111 1000 1
4	0010	1: 0 ⇒ no operation	0001 1000	1b: 01 ⇒ Prod = Prod + Mcand	0001 1000 1
	0010	2: Shift right Product	0000 1100	2: Shift right Product	0000 1100 0

Example

- Let's try Booth's algorithm with negative numbers:
- $2_{ten} \times -3_{ten} = -6_{ten}$ or $0010_{two} \times 1101_{two} = 1111\ 1010_{two}$

Sign extension

Iteration	Step	Multiplicand	Product
0	Initial values	0010	0000 1101 0
1	1c: 10 ⇒ Prod = Prod – Mcand	0010	1110 1101 0
	2: Shift right Product	0010	1111 0110 1
2	1b: 01 ⇒ Prod = Prod + Mcand	0010	0001 0110 1
	2: Shift right Product	0010	0000 1011 0
3	1c: 10 ⇒ Prod = Prod – Mcand	0010	1110 1011 0
	2: Shift right Product	0010	1111 0101 1
4	1d: 11 ⇒ no operation	0010	1111 0101 1
	2: Shift right Product	0010	1111 1010 1

2-Bit Booth Encoding

- Using more bits for faster multiplies

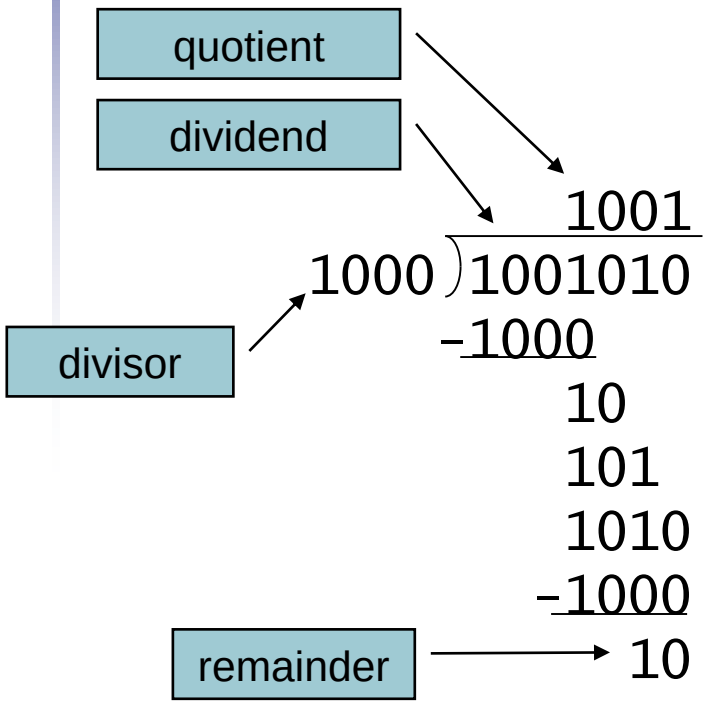
  b: multiplicand

Current bits		Previous bit	Operation	Reason
ai+1	*ai*	*ai–1*		
0	0	0	**NOP**	
0	0	1	**+b**	
0	1	0	**+b**	
0	1	1	**+2b**	
1	0	0	**-2b**	
1	0	1	**-b**	
1	1	0	**-b**	
1	1	1	**NOP**	

MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - `mult rs, rt  /  multu rs, rt`
    - 64-bit product in HI/LO
  - `mfhi rd  /  mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - Least-significant 32 bits of product –> rd

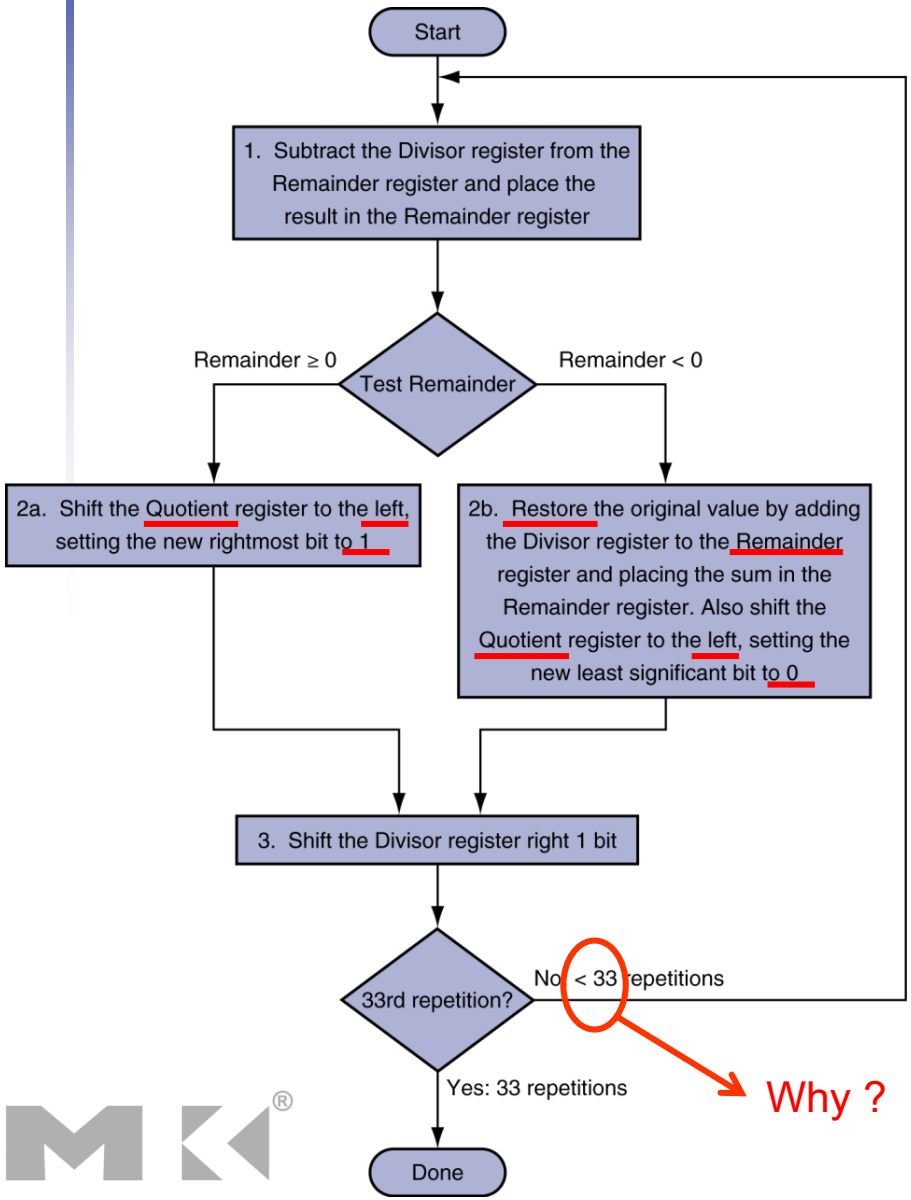
Division

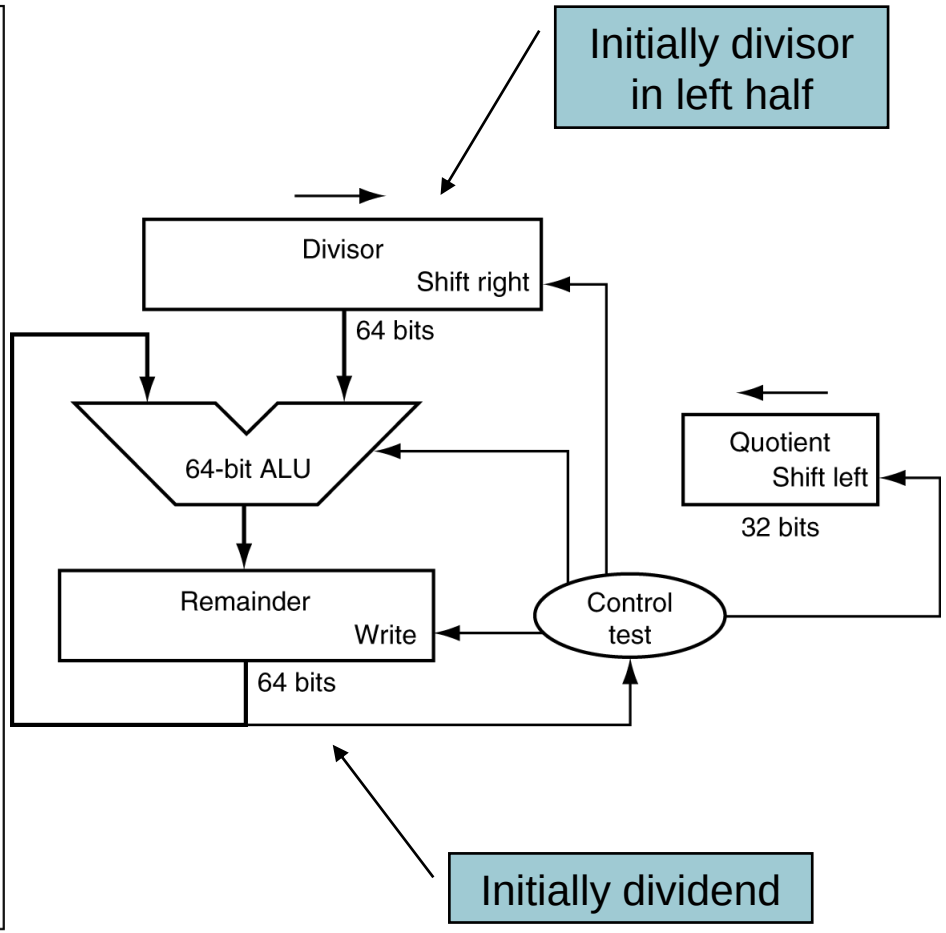


n-bit operands yield *n*-bit quotient and remainder

- 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





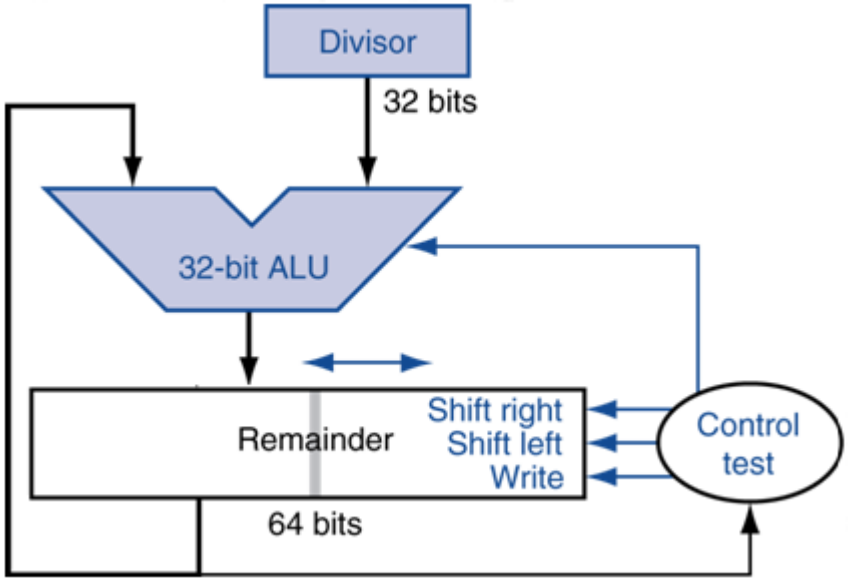
Example

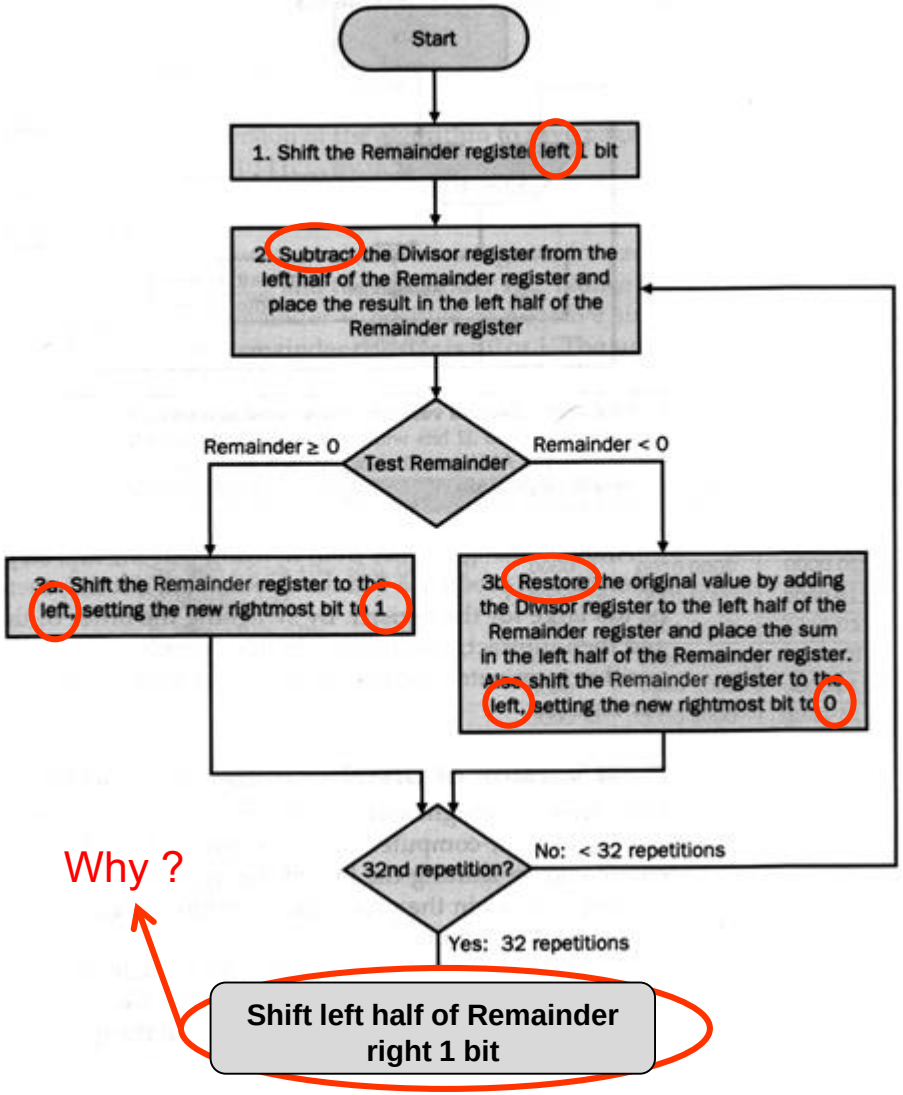
- Using a 4-bit version to save pages, try dividing 7_{ten} by 2_{ten} or $0000\ 0111_{two}$ by 0010_{two}

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

- Note: it takes N+1 steps to obtain the correct result.

Optimized Divider





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

Why ?

Example

- Using optimized divider hardware to divide 7_{ten} by 2_{ten} or $0000\ 0111_{two}$ by 0010_{two}

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	2: Rem = Rem – Div	0010	0001 1000
	3a: Rem ≥ 0 ⇒ sll R, R0 = 1	0010	0011 0001
4	2: Rem = Rem – Div	0010	0001 0001
	3a: Rem ≥ 0 ⇒ sll R, R0 = 1	0010	0010 0011
	Shift left half of Rem right 1	0010	0001 0011

Signed Division

- Simplest solution:
  - remember the signs of the divisor and dividend and then negate the quotient if the signs disagree
  - Note: the dividend and the remainder must have the same signs!
  - Example
    - **+7 ÷ +2 → Quotient = +3, Remainder = +1**
    - **–7 ÷ +2 → Quotient = –3, Remainder = –1**
    - **+7 ÷ –2 → Quotient = –3, Remainder = +1**
    - **–7 ÷ –2 → Quotient = +3, Remainder = –1**

MIPS Division

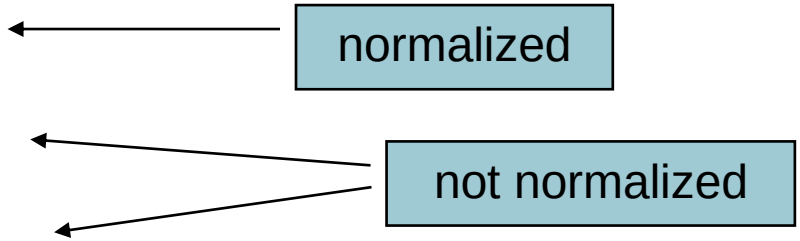
- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Instructions
  - `div rs, rt / divu rs, rt`
  - No overflow or divide-by-0 checking
    - Software must perform checks if required
  - Use `mfhi`, `mflo` to access result

3.5

Floating Point

Floating Point

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

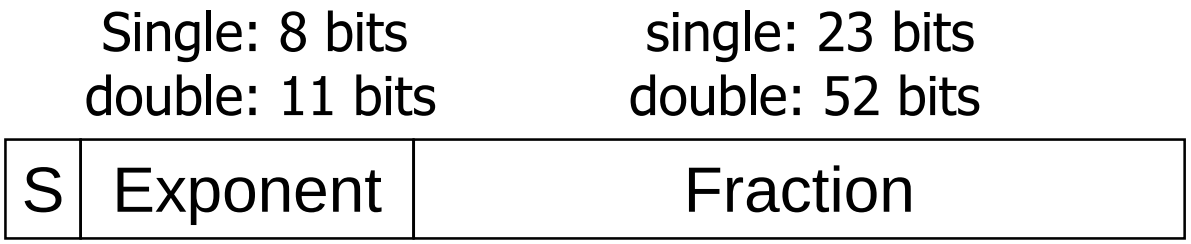


- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

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



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

- 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
- 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 $\approx$ 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
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

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 + Bias
    - Single: –1 + 127 = 126 = 01111110_2
    - Double: –1 + 1023 = 1022 = 01111111110_2
- Single: 10111111101000…00
- Double: 10111111111101000…00

Floating-Point Example

- What number is represented by the single-precision float

  1100000101000…00

  - S = 1
  - Fraction = $01000\ldots00_2$
  - Exponent = $10000001_2 = 129$

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

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

  $= -5.0$

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

Denormal Numbers

- Exponent = 000...0 $\Rightarrow$ hidden bit is 0
  - Smaller than normal numbers
    - for gradual underflow, with diminishing precision
    - The smallest single precision de-normalized number is: $0.0000\ 0000\ 0000\ 0000\ 0000\ 001_{two} \times 2^{-126}$
  - De-normal with fraction = 000...0

$$X = (-1)^S \times (0 + 0) \times 2^{-126} = \pm 0.0$$

Two representations of 0.0

Floating-Point Summary

Single Precision		Double Precision		Meaning
Exponent	Significant	Exponent	Significant	
0	0	0	0	0
0	Non-zero	0	Non-zero	+/- de-normalized number
1-254	Anything	1-2046	Anything	+/- floating-point number
255	0	2047	0	+/- infinity
255	Non-zero	2047	Non-zero	NaN (Not a number)

The smallest positive single precision normalized number is:

$$1.0000\ 0000\ 0000\ 0000\ 0000\ 000_{\text{two}} \times 2^{-126}$$

The smallest single precision de-normalized number is:

$$0.0000\ 0000\ 0000\ 0000\ 0000\ 001_{\text{two}} \times 2^{-126} \quad \text{or} \quad 1.0 \times 2^{-149}$$

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} (Already fits in 4 bits, so no change)

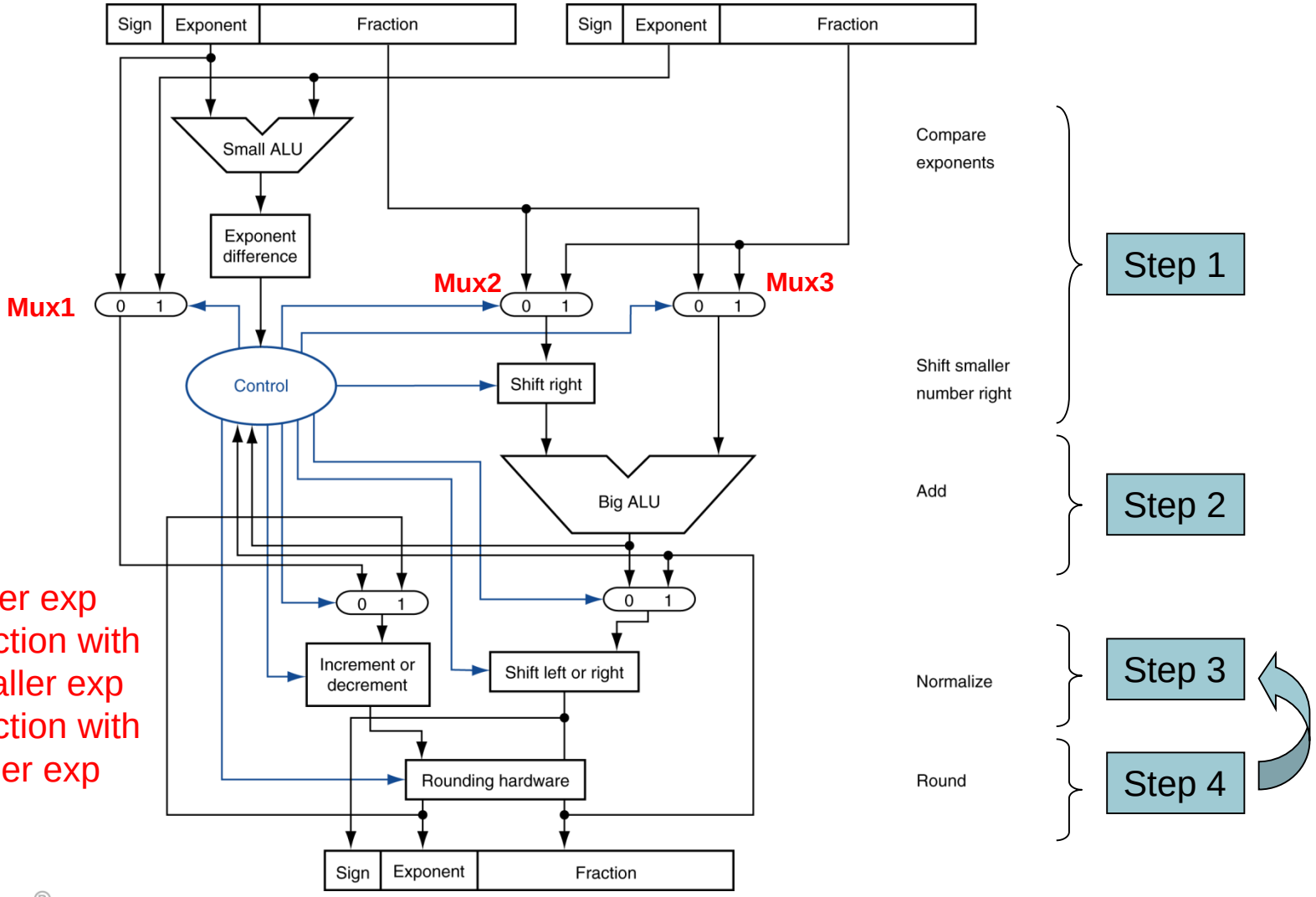
Floating-Point Addition

- 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 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



Mux1: larger exp
Mux2: Fraction with smaller exp
Mux3: Fraction with larger exp

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
  - $+1.021 \times 10^{6}$

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: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$ (**– 127**)
- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$
- 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
  - $-1.110_2 \times 2^{-3} = -0.21875$

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: 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
```

Example: Rounding with Guard Digits

Add $2.56_{ten} \times 10^0$ to $2.34_{ten} \times 10^2$, assuming that we have three significant decimal digits. Round to the nearest decimal number with three significant decimal digits, first with guard and round digits, and then without them.

First we must shift the smaller number to the right to align the exponents, so $2.56_{ten} \times 10^0$ becomes $0.0256_{ten} \times 10^2$. The guard digit holds 5 and the round digit holds 6. The sum is

$$\begin{array}{r} 2.3400_{ten} \\ + \quad 0.0256_{ten} \\ \hline 2.3656_{ten} \end{array}$$

Thus the sum is $2.3656_{ten} \times 10^2$. Rounding sum up with three significant digits yields $2.37_{ten} \times 10^2$.

```
[              ] [ ][ ][           ]
 result in the   g  r  combined to
  given                produce
  precision            a sticky bit
```

Doing this *without* guard and round digits drops two digits from the calculation. The new sum is

$$\begin{array}{r} 2.34_{ten} \\ + \quad 0.02_{ten} \\ \hline 2.36_{ten} \end{array}$$

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

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
  - Other instructions: less frequent