Arithmetic for Computers

Computer Organization
502044

Acknowledgement

This slide show is intended for use in class, and is not a complete document. Students need to refer to the book to read more lessons and exercises. Students have the right to download and store lecture slides for reference purposes; Do not redistribute or use for purposes outside of the course.

[2]. David A. Patterson, John L. Hennessy, [2014], **Computer Organization and Design: The Hardware/Software Interface**, 5th edition, Elsevier, Amsterdam.

[3]. John L. Hennessy, David A. Patterson, [2012], **Computer Architecture: A Quantitative Approach**, 5th edition, Elsevier, Amsterdam.

📧 **trantrungtin.tdtu.edu.vn**

Syllabus

- 8.1 Introduction
- 8.2 Addition and Subtraction
- 8.3 Multiplication
- 8.4 Division
- 8.5 Floating Point

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

		(0)		(0)		(1)		(1)		(0)		(Carries)
. . .		0		0		0		1		1		1
. . .		0		0		0		1		1		0
. . .	(0)	0	(0)	0	(0)	1	(1)	1	(1)	0	(0)	1

- 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

- Some languages (e.g., C) ignore overflow
  - Use MIPS `addu`, `addui`, `subu` instructions
- Other languages (e.g., Ada, Fortran) require raising an exception/interrupt
  - Use MIPS `add`, `addi`, `sub` instructions
  - On overflow, invoke exception/interrupt 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

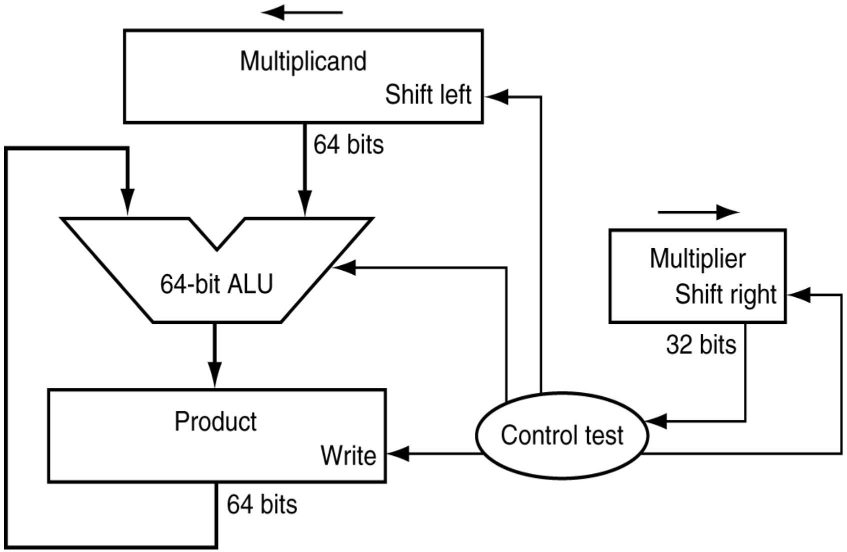
Multiplication

- Start with long-multiplication approach

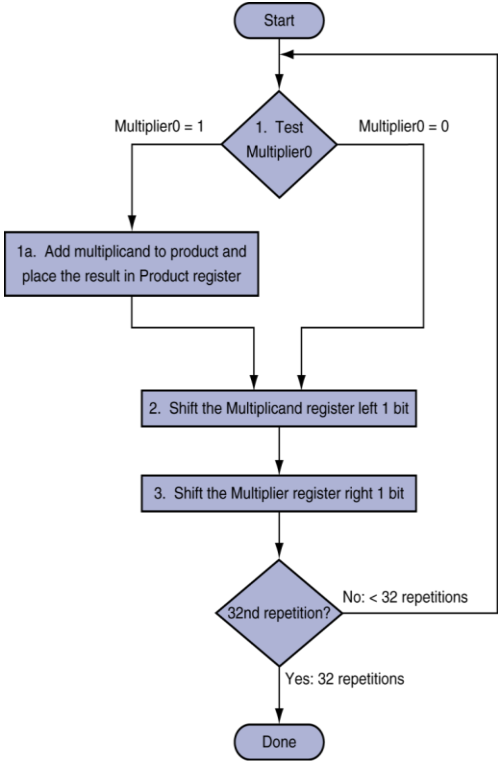
multiplicand
multiplier
product

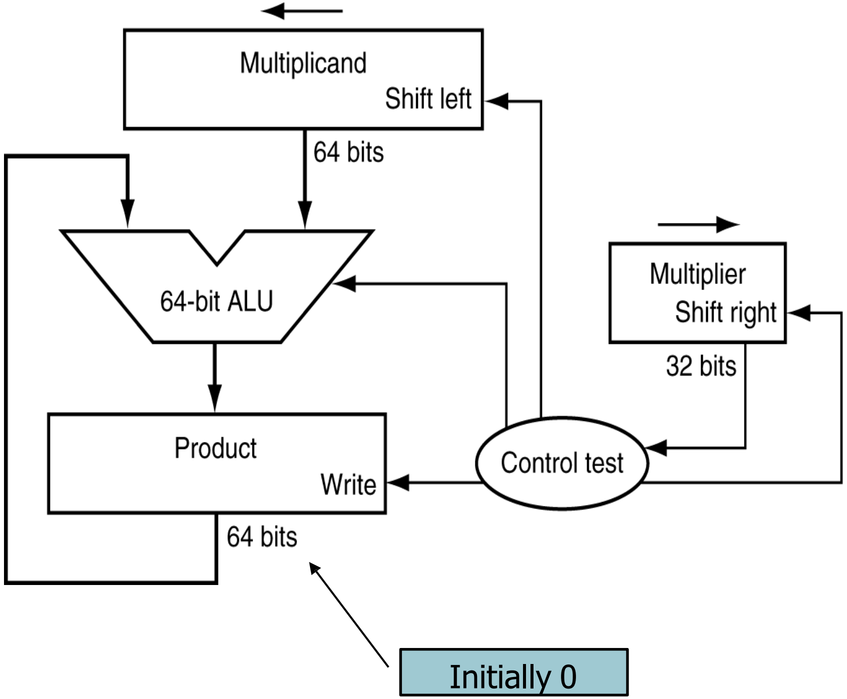
```
        1000
  ×     1001
  ----------
        1000
       0000
      0000
     1000
  ----------
     1001000
```

Length of product is the sum of operand lengths



Multiplication Hardware





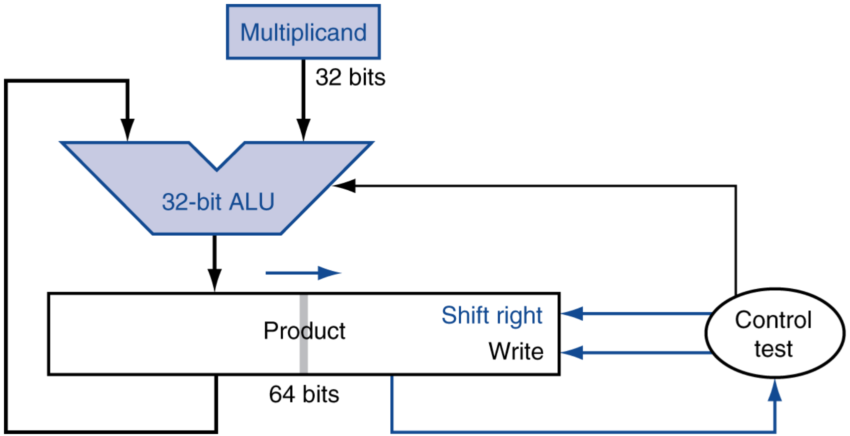
Multiplication Hardware (2)

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

- Multiply example using flow chart algorithm
- The bit examined to determine the next step is circled in color

Optimized Multiplier

- Perform steps in parallel: add/shift

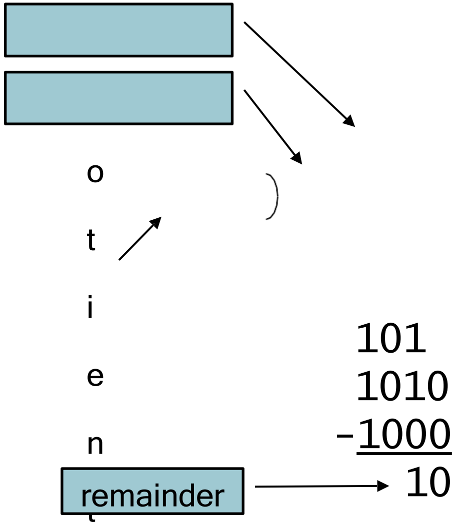


- One cycle per partial-product addition
  - That's ok, if frequency of multiplications is low

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

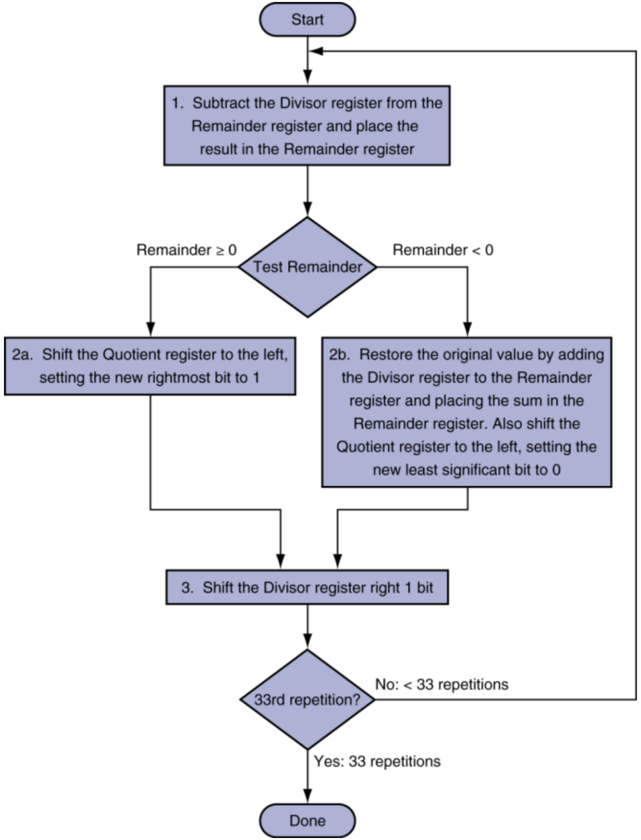
Division

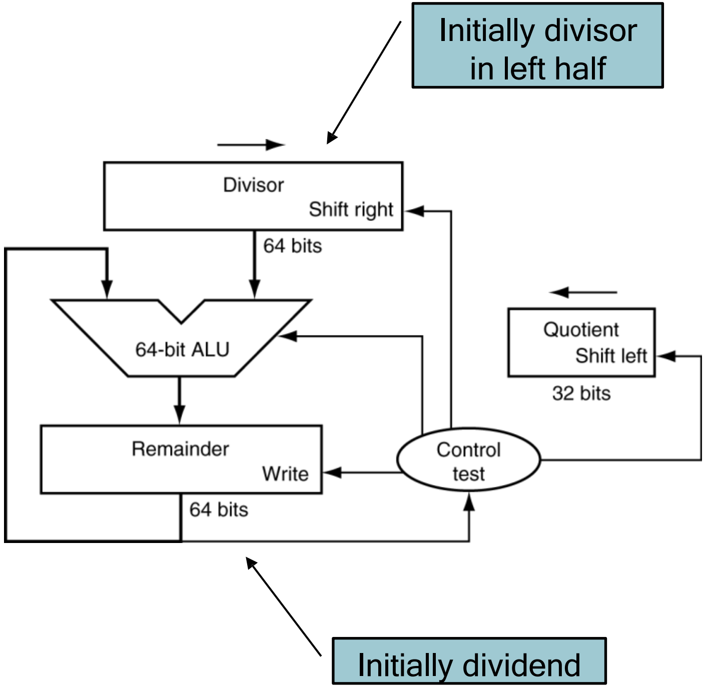


n-bit operands yield *n*-bit quotient and remainder

- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits ct
    - 1 bit in quotient, subtra dow next
  - Otherwise n
    - 0 bit in quotient, bring dividend bit
  - 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



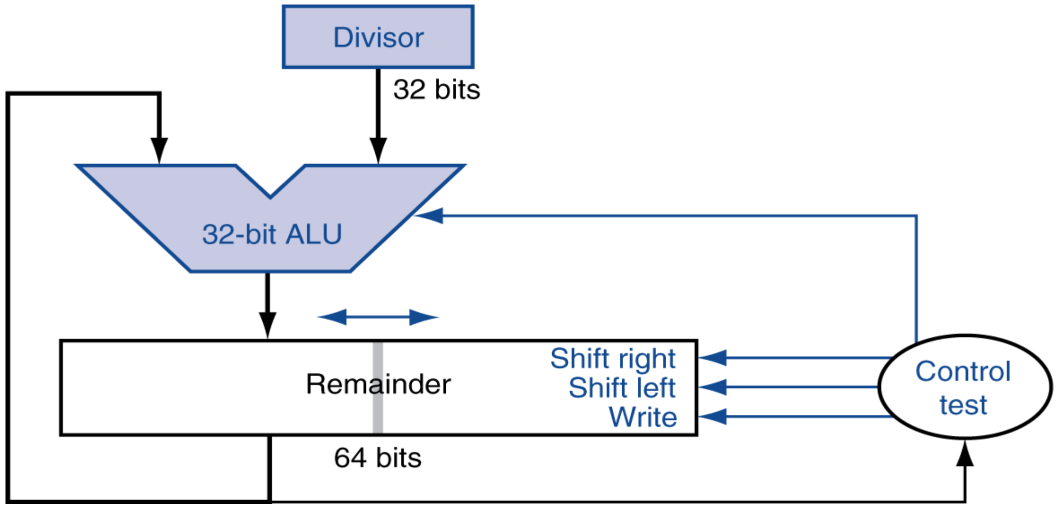


Division Example

Using a 4-bit version of the algorithm divide 7_{10} by 2_{10}, or $0000\ 0111_2$ by 0010_2.

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 $\Longrightarrow$ +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 $\Longrightarrow$ +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 $\Longrightarrow$ +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 $\Longrightarrow$ 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 $\Longrightarrow$ 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

MIPS Division

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

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

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56} ← normalized
  - $+0.002 \times 10^{-4}$ ← not normalized
  - $+987.02 \times 10^{9}$ ← 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 $\Rightarrow$ non-negative, 1 $\Rightarrow$ 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
    □ 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
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
- actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 significand = 1.0
  - ±1.0 × 2–1022 ≈ ±2.2 × 10–308
- Largest value
  - Exponent: 11111111110
- actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 significand ≈ 2.0
  - ±2.0 × 2+1023 ≈ ±1.8 × 10+308

Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx 2^{-23}
    - Equivalent to $23 \times \log_{10} 2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx 2^{-52}
    - Equivalent to $52 \times \log_{10} 2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Floating-Point Example

- Represent -0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = −1 + 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 = $01000\ldots00_2$
  - Exponent = 10000001_2 = 129
- $x = (-1)^1 \times (1 + .01_2) \times 2^{(129 - 127)}$

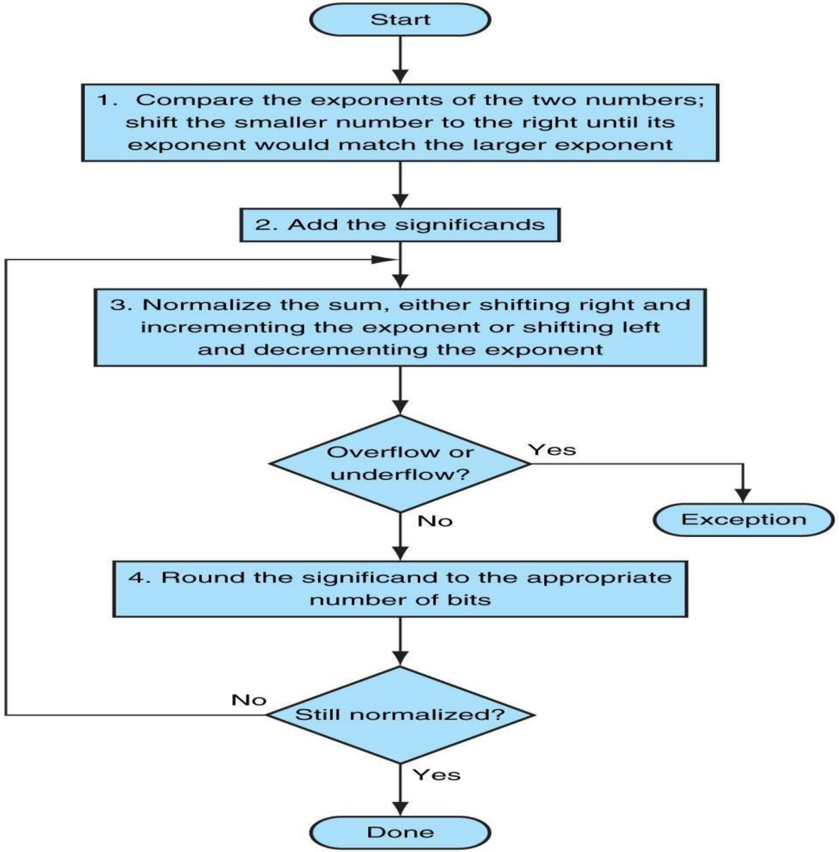
  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

Floating-Point Addition

- Consider a 4-digit decimal example
  - 9.999 × 101 + 1.610 × 10–1
- 1. Align decimal points
  - Shift number with smaller exponent
  - 9.999 × 101 + 0.016 × 101
- 2. Add significands
  - 9.999 × 101 + 0.016 × 101 = 10.015 × 101
- 3. Normalize result & check for over/underflow
  - 1.0015 × 102
- 4. Round and renormalize if necessary
  - 1.002 × 102

Floating-Point Addition



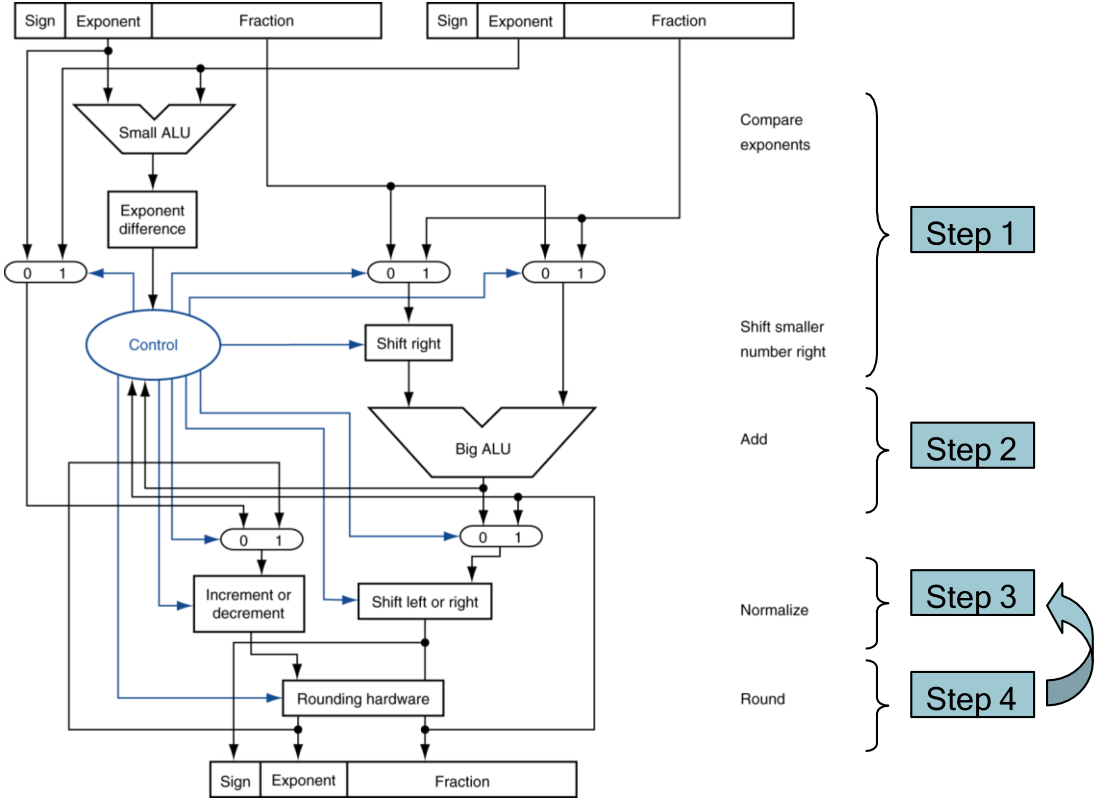
Floating-Point Addition

- Now consider a 4-digit binary example
    - 1.0002 × 2–1 + –1.1102 × 2–2 (0.5 + –0.4375)
- 1. Align binary points
    - Shift number with smaller exponent
    - 1.0002 × 2–1 + –0.1112 × 2–1
- 2. Add significands
    - 1.0002 × 2–1 + –0.1112 × 2–1 = 0.0012 × 2–1
- 3. Normalize result & check for over/underflow
    - 1.0002 × 2–4, with no over/underflow
- 4. Round and renormalize if necessary
    - 1.0002 × 2–4 (no change) = 0.0625

FP Adder Hardware

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

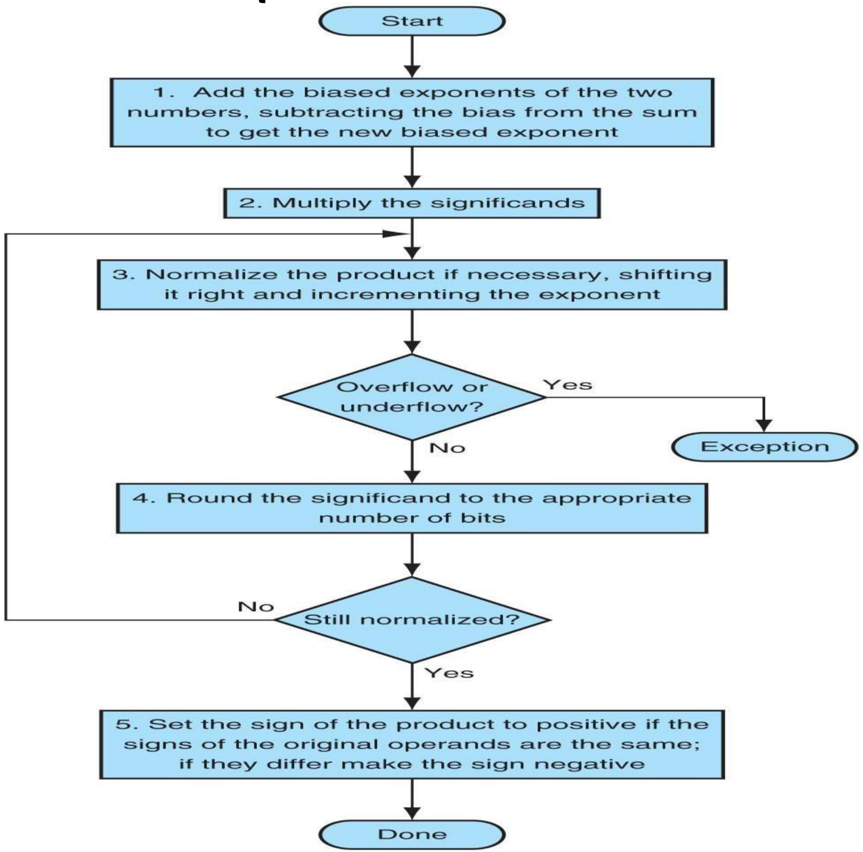
FP Adder Hardware



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
  - $1.110 \times 9.200 = 10.212$ □ 10.212×10^{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(2)



Floating-Point Multiplication(3)

- Now consider a 4-digit binary example
  - 1.0002 × 2–1 × –1.1102 × 2–2 (0.5 × –0.4375)
- 1. Add exponents
  - Unbiased: –1 + –2 = –3
  - Biased: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 127
- 2. Multiply significands
  - 1.0002 × 1.1102 = 1.1102 1.1102 × 2–3
- 3. Normalize result & check for over/underflow
  - 1.1102 × 2–3 (no change) with no over/underflow
- 4. Round and renormalize if necessary
  - 1.1102 × 2–3 (no change)
- 5. Determine sign: +ve × –ve –ve
  - –1.1102 × 2–3 = –0.21875

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP □ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

FP Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: $f0, $f1, … $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
    - Release 2 of MIPs ISA supports 32 × 64-bit FP reg’s
- FP instructions operate only on FP registers
  - Programs generally don’t do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - `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
```

Right Shift and Division

- Left shift by i places multiplies an integer by 2i
- Right shift divides by 2i?
  - Only for unsigned integers
- For signed integers
  - Arithmetic right shift: replicate the sign bit
  - e.g., –5 / 4
    - 111110112 >> 2 = 111111102 = –2
    - Rounds toward –∞
  - c.f. 111110112 >>> 2 = 001111102 = +62

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