Assembly Language and Computer Organization

Arithmetic for Computers

Tuesday, October 25th, 2022

Dr. Robin Pottathuparambil, CSE Department, UNT

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

Arithmetic for Multimedia

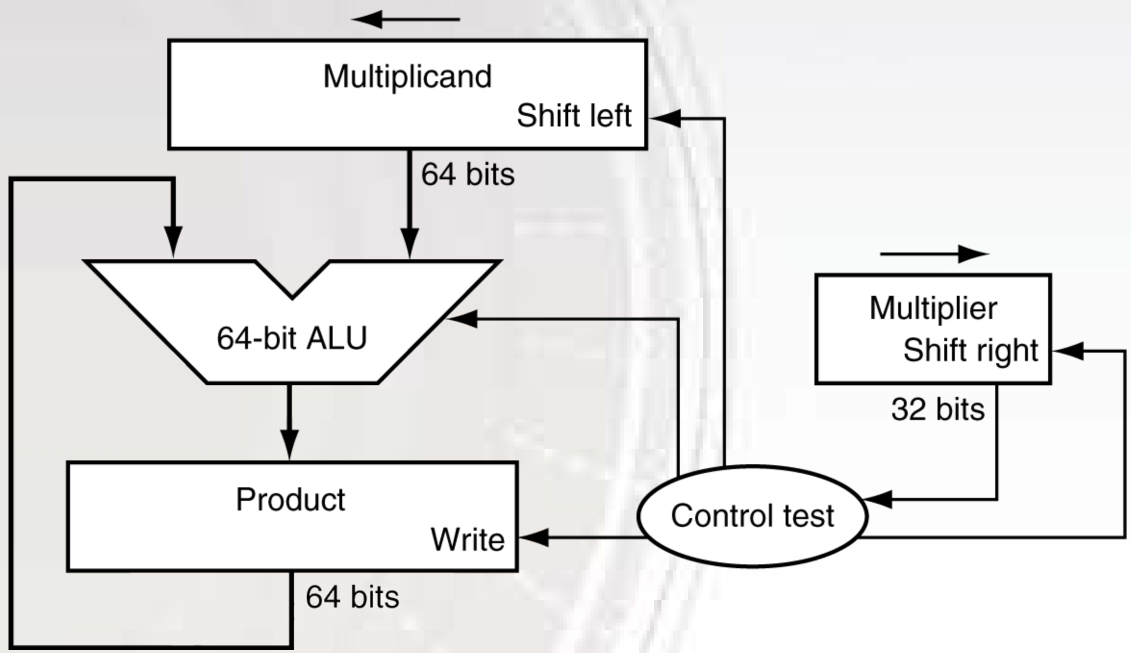
- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use 64-bit adder, with partitioned carry chain
    - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
  - SIMD (single-instruction, multiple-data)
- Saturating operations
  - On overflow, result is largest representable 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
```



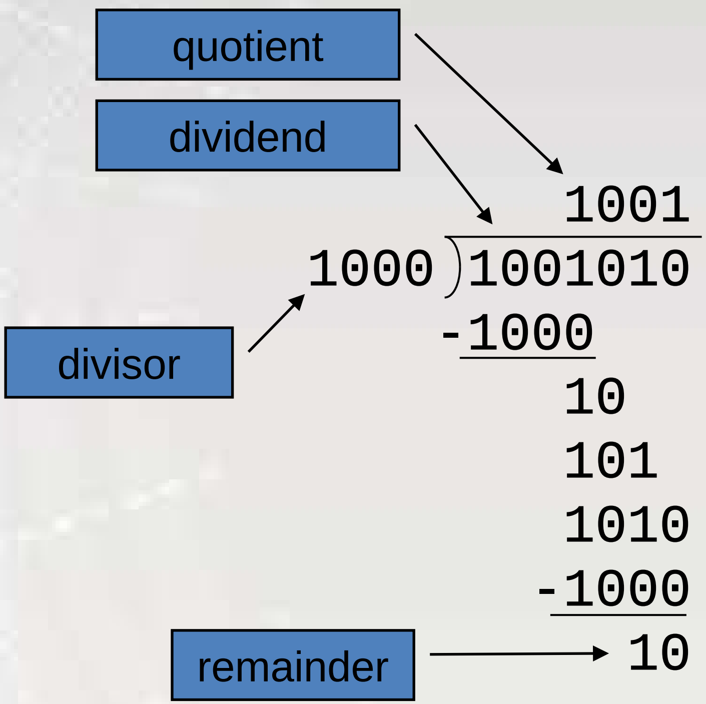
Number of bits of product is the sum of operand bit lengths

LEGv8 Multiplication

Three multiply instructions:

- MUL: multiply
  - Gives the lower 64 bits of the product
- SMULH: signed multiply high
  - Gives the upper 64 bits of the product, assuming the operands are signed
- UMULH: unsigned multiply high
  - Gives the upper 64 bits of the product, assuming the operands are unsigned

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

LEGv8 Division

- Two instructions:
  - SDIV (signed)
  - UDIV (unsigned)

- Both instructions ignore overflow and division-by-zero

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

- 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: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

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

Floating-Point Example 1

- Represent –0.75 in single and double precision

S	Exponent	Fraction

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

Floating-Point Example 1

- Represent –0.75 in single and double precision
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: $-1 + 127 = 126 = 01111110_2$
    - Double: $-1 + 1023 = 1022 = 01111111110_2$
- Single: 1011111101000…00 = 0xBF400000
- Double: 1011111111101000…00 = 0xBFE8000000000000

S	Exponent	Fraction

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

Floating-Point Example 2

- Represent 2.375 in single and double precision

S	Exponent	Fraction

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

Floating-Point Example 3

- What number is represented by the single-precision float 11000000101000…00 (0xC0A00000)

S	Exponent	Fraction

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

Floating-Point Example 3

- What number is represented by the single-precision float 11000000101000…00 (0xC0A00000)
  - S = 1
  - Fraction = $01000\ldots00_2$
  - Exponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + .01_2) \times 2^{(129-127)}$
  - $= (-1) \times 1.25 \times 2^2$
  - $= -5.0$

S	Exponent	Fraction

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

Floating-Point Example 4

- What number is represented by the double-precision float 0x3FF6000000000000?

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

Denormal Numbers

- Exponent = 000...0 $\Rightarrow$ hidden bit is 0

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

- 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 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 Addition (Decimal)

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 (Binary)

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

IEEE Floating-Point Addition

Now consider the computation (0.5 + 0.375)

- 0 01111110 00000…000 + 0 01111101 10000…000

1. Align binary points (The bit in red is the 1.0 in fraction)
   - Shift the fraction of the number with smaller exponent
     - $(-1)^0 \times (1 + 0) \times 2^{(126-127)} = 1 \times 1 \times 2^{-1}$
       - 0 01111110 1|00000…000 (The bit in red is the 1.0 in the fraction)
     - $(-1)^0 \times (1 + 0.5) \times 2^{(125-127)} = 1 \times 1.5 \times 2^{-2} = 1 \times 0.75 \times 2^{-1}$
       - 0 01111101 1|10000…000 ⇒ 0 01111110 0|11000…000 (shifting right)
2. Add significands
   - 0 01111110 1|11000…000
3. Normalize result & check for over/underflow
   - 0 01111110 1|11000…000 (no over/underflow)
4. Round and renormalize if necessary
   - 0 01111110 1|11000…000 (no change) = 0.875

IEEE Floating-Point Subtraction

Now consider the computation (0.5 - 0.375)

- 0 01111110 00000…000 - 0 01111101 10000…000

1. Align binary points
   - Shift the fraction of the number with smaller exponent
     - $(-1)^0 \times (1 + 0) \times 2^{(126-127)} = 1 \times 1 \times 2^{-1}$
       - 0 01111110 1|00000…000 (The bit in red is the 1.0 in the fraction)
     - $(-1)^1 \times (1 + 0.5) \times 2^{(125-127)} = 1 \times 1.5 \times 2^{-2} = 1 \times 0.75 \times 2^{-1}$
       - 1 01111101 1|10000…000 ⇒ 1 01111110 0|11000…000 (shifting right)
2. Subtract significands
   - 0 01111110 0|01000…000
3. Normalize result & check for over/underflow
   - 0 01111100 1|00000…000 (no over/underflow)
4. Round and renormalize if necessary
   - 0 01111100 1|00000…000 (no change) = 0.125

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