Integer Arithmetic

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Introduction

- **Bits are just bits**
  - No inherent meaning: conventions define relationship between bits and numbers
  - $n$-bit binary numbers: $0 \sim 2^n-1$ decimal numbers

- **Of course it gets more complicated**
  - Numbers are finite (overflow)
  - Negative numbers
  - Fractions and real numbers
  - Precision and accuracy
  - Error propagation, ...
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 0 1 1 1 1
... 0 0 0 1 1 0
... 0 0 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
Simple Adder

- \( n \)-bit ripple-carry adder
Overflow

Dealing with overflow

• Some languages (e.g., C) ignore overflow
  – Use MIPS addu, addiu, 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
Arithmetic for Multimedia

- **Graphics and media processing**
  - Operate 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
    - cf. 2s-complement modulo arithmetic
  - E.g., clipping in audio, saturation in video
Multiplication (1)

- Long-multiplication approach

```
  multiplicand
   1000
   × 1001

  multiplier

  product
   1000
   0000
   0000
  1001000

Length of product is the sum of operand lengths
```
Multiplication (2)

- Multiplication hardware
Multiplication (3)

- Optimized multiplier
  - Perform steps in parallel: add/shift

- One cycle per partial-product addition
  - That’s ok, if frequency of multiplications is low
Multiplication (4)

- **Fast multiplier**
  - Many transistors are available
    - 31 adders for multiplication of 32-bit numbers
    - Upper 31 bits + carry to the next level
    - 1 bit LSB to the product of final result
    - Cost/performance tradeoff
  - Delay can be reduced further by using a parallel tree
  - Can be pipelined
    - Several multiplication operations performed in parallel
Multiplication (5)

- **MIPS multiplication**
  - Two 32-bit registers for product
    - HI: most-significant 32 bits
    - LO: least-significant 32 bits
  - Instructions
    - 64-bit product in HI/LO
      - `mult rs, rt` and `mulu rs, rt`
    - Move from HI/LO to rd
      - `mfhi rd` and `mflo rd`
    - Can test HI value to see if product overflows 32 bits
    - Least-significant 32 bits of product → rd
      - `mul rd, rs, rt`
Division (1)

- Long-division approach
  - Check for 0 divisor
    - If divisor \( \leq \) 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

\[
\begin{array}{c}
\text{quotient} \\
\text{dividend} \\
\text{divisor} \\
\text{remainder}
\end{array}
\]

\[
\begin{array}{c}
1001 \\
1001010 \\
-1000 \\
10 \\
101 \\
1010 \\
-1000 \\
10
\end{array}
\]

\( n \)-bit operands yield \( n \)-bit quotient and remainder
Division (2)

- Division hardware

Initially divisor in left half

Initially dividend

1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register.

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.

If repetitions < 33 repetitions, then:

- No: Shift right 64 bits.
- Yes: 33 repetitions
  - 64-bit ALU
  - Remainder
  - Quotient Shift Left
  - Control test
  - Write
  - 64 bits

Start

Remainder ≥ 0

Remainder < 0

Test Remainder

Divisor

Shift right

64 bits

32 bits
Division (3)

- Optimized divider

- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both
Division (4)

- 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
Division (5)

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