### ALU Design

Lecturer: Hui Annie Guo
huig@cse.unsw.edu.au
K17-501F (ext. 57136)

Lecture notes adapted from MK (textbook publisher)

### Integer Addition

- Example: 7 + 6

```
          (0)    (0)    (0)    (1)    (1)    (1)
          0      0      0      1      1      1
            0      0      0      1      1      0
            (0)   (0)    (0)    (1)    (1)    (0)
            0      0      1      1      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

### Typical Designs for Addition

- Ripple-Carry Adder
- Carry Look Ahead Adder
- Carry Select Adder

### Overview

- How basic ALU operations are implemented?
  - Operations on integers
    - Addition and subtraction
    - Multiplication and division
    - Dealing with overflow
  - Operations on floating-point real numbers

Typical Designs for Addition
Ripple-Carry Adder

- The delay of critical path of n-bit ripple-carry adder is $n \cdot T_d$
  - $T_d$: the delay of a full adder

4-bit RCA

Carry Look Ahead Adder

4-bit CLA

Cascaded Carry Look-ahead (16-bit)

Carry-Select Adder

- $\text{delay}(n) = 2 \cdot \text{delay}(n/2)$
- $\text{delay}(n) = \text{delay}(n/2) + \text{delay}(\text{mux})$
Integer Subtraction

- Add the negation of second operand
  - How to implement it in hardware?
- Example: $7 - 6 = 7 + (-6)$
  
  \[
  \begin{align*}
  +7: & \quad 0000 \ 0000 \ldots \ 0000 \ 0111 \\
  -6: & \quad 1111 \ 1111 \ldots \ 1111 \ 1010 \\
  +1: & \quad 0000 \ 0000 \ldots \ 0000 \ 0001
  \end{align*}
  \]

  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) or designs ignore overflow
  - E.g. MIPS addu, addu1, subu instructions
- Some languages (e.g., Ada, Fortran) or designs require raising an exception
  - E.g. MIPS add, add1, 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

Overflow Detection Logic

- Carry into MSB $\oplus$ Carry out of MSB
  - For a N-bit ALU: Overflow = $\text{CarryIn}[\text{N-1}] \oplus \text{CarryOut}[\text{N-1}]

\[
\begin{array}{c|c|c|c|c}
\text{CarryIn} & \text{ALU} & \text{CarryOut} & \text{ALU} & \text{CarryOut} \\
\hline
0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

Multiplication

- Start with long-multiplication approach

\[
\begin{array}{c|c|c|c|c}
\text{multiplicand} & 1000 & \times & 1001 & \text{multiplier} \\
\hline
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 \\
\end{array}
\]

Length of product is the sum of operand lengths
Multiplication Hardware

Initially 0

Optimized Multiplier

• Perform steps in parallel: add/shift

Faster Multiplier

• Uses multiple adders
  – Cost/performance tradeoff

Division

• 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 < 0, add divisor back
• Signed division
  – Divide using absolute values
  – Adjust sign of quotient and remainder as required

n-bit operands yield n-bit quotient and remainder
### Division Hardware

1. Subtract the quotient from the remainder register and place the result in the remainder register.
2. Shift the quotient register right 1 bit.
3. Shift the dividend register right 1 bit.

### Optimized Divider

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

### Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - $-2.34 \times 10^{96}$
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$
- In binary
  - $\pm1.xxxxxx \times 2^{yyyy}$
- 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

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

<table>
<thead>
<tr>
<th>S</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000001</td>
<td>01000...00</td>
</tr>
</tbody>
</table>

Single-Precision Range

- Exponents 0000000 and 1111111 reserved
- Smallest value
  - Exponent: 0000001
    - actual exponent = 1 – 127 = –126
  - Fraction: 000...00 ⇒ significand = 1.0
  - ±1.0 \times 2^{-126} ≈ ±1.2 \times 10^{-38}
- Largest value
  - Exponent: 1111110
    - actual exponent = 254 – 127 = +127
  - Fraction: 111...11 ⇒ significand ≈ 2.0
  - ±2.0 \times 2^{+127} ≈ ±3.4 \times 10^{+38}

Floating-Point Example

- Represent –0.75
  - –0.75 = (–1)^1 \times 1.1_2 \times 2^{-1}
  - S = 1
  - Fraction = 1000...00_2
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110_2
    - Double: –1 + 1023 = 1022 = 01111111110_2
- Single: 1011111101000...00
- Double: 011111111101000...00

Floating-Point Example

- What number is represented by the single-precision float
  1\text{000000101000...00}
  - S = 1
  - Fraction = 01000...00_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
Denormal Numbers

- Exponent = 000...0 ⇒ 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
- Exponent = 111...1, Fraction ≠ 000...0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0

Floating-Point Addition

- Consider a 4-digit decimal example
  \[ 9.999 \times 10^1 + 1.610 \times 10^{-1} \]
- Calculation steps:
  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 \times 10^2
  4. Round and renormalize if necessary
     - 1.002 \times 10^2

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 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