Arithmetic for Computers

陳伯寧 教授
電信工程學系
國立交通大學

Arithmetic

- Where we've been:
  - Abstractions:
    - Instruction Set Architecture
    - Assembly Language and Machine Language
- What's up ahead:
  - Implementing the Architecture
**Numbers**

- **Bits are just bits (no inherent meaning)**
  — conventions define relationship between bits and numbers

- **Binary numbers (base 2)**
  
  \[
  \begin{array}{c}
  0000 \ 0001 \ 0010 \ 0011 \ 0100 \ 0101 \ 0110 \ 0111 \ 1000 \ 1001 \ldots \\
  \text{decimal: } 0...2^n-1
  \end{array}
  \]

**Numbers**

- Of course it gets more complicated in real computers:
  - numbers are finite (can possibly overflow)
  - fractions and real numbers
  - negative numbers

  e.g., no `subi` instruction in MIPS;
  it can be done by using `addi` since `addi` can add a negative number.

- **How do we represent negative numbers in computer system?**
Possible representations

<table>
<thead>
<tr>
<th>Sign Magnitude:</th>
<th>1's Complement</th>
<th>2's Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 = +0</td>
<td>000 = +0</td>
<td>000 = +0</td>
</tr>
<tr>
<td>001 = +1</td>
<td>001 = +1</td>
<td>001 = +1</td>
</tr>
<tr>
<td>010 = +2</td>
<td>010 = +2</td>
<td>010 = +2</td>
</tr>
<tr>
<td>011 = +3</td>
<td>011 = +3</td>
<td>011 = +3</td>
</tr>
<tr>
<td>100 = −0</td>
<td>100 = −3</td>
<td>100 = −4</td>
</tr>
<tr>
<td>101 = −1</td>
<td>101 = −2</td>
<td>101 = −3</td>
</tr>
<tr>
<td>110 = −2</td>
<td>110 = −1</td>
<td>110 = −2</td>
</tr>
<tr>
<td>111 = −3</td>
<td>111 = −0</td>
<td>111 = −1</td>
</tr>
</tbody>
</table>

• How 2's complement gets its name?

(+)2's complement + (−x)2's complement = 2^n

Possible representations

- Issues:
  - Balance: 2's complement only provides −4 but no −4.
  - Number of zeros:
  - Sign decision:
  - Ease of hardware implementation:
Two's complement operation shortcuts

- **Negation shortcut**
  - Negating a two's complement number: invert all bits and add 1
  - Remember: “negate” and “invert” are quite different!

- **Sign extension shortcut**
  - Sign extension of 2's complement numbers
    - Converting n-bit numbers into numbers with more than n bits:
      - copy the most significant bit (the sign bit) into the other bits
  
  
<table>
<thead>
<tr>
<th>Original</th>
<th>Extended</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>00000000 1010</td>
<td>+2</td>
</tr>
<tr>
<td>1010</td>
<td>11111010</td>
<td>-6</td>
</tr>
</tbody>
</table>
### lb versus lbu

lb = *load byte as a signed number*
- treats the byte as a signed number, and hence sign-extend to fill the 24 left-most bits of the register

lbu = *load byte as an unsigned number*
- hence, sign extension is not performed

### Bounds check shortcut

- **Bounds check shortcut**
  - Use to check whether index-out-of-bound occurs or not

  ```
  sltu $t0, $a1, $t2; $t0=1 if 0<=$a1<$t2
  beq $t0, $zero, Error; if bad, goto Error.
  ```
Addition & subtraction of 2’s complements

- Two’s complement operations make it easy
  - Subtraction using addition of negative numbers

\[
\begin{array}{c c c}
0111 & + & 1010 \\
\hline
0001 & + & -6 \\
\end{array}
\]

Addition & subtraction of 2’s complements

- Overflow (result too large for finite computer word):
  - e.g., adding two n-bit numbers does not yield an n-bit number

\[
\begin{array}{c c c}
0111 & + & 0001 \\
\hline
1000 & + & -8 \\
\end{array}
\]
How to detect overflow?

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
  - Overflow when adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive and get a negative
  - or, subtract a positive from a negative and get a positive

How to detect overflow?

(+) + (+) -> (-)
(+)+(-) -> (no overflow)
(-)+ (+) -> (no overflow)
(-)+(-) -> (+)

(+)-(+)->(no overflow)
(+)-(−)->(−)
(−)-(+)->(+)
(−)-(−)->(no overflow)
Effects of overflow

- MIPS detects overflow with an exception (sometimes, called interrupt in other instruction set design).
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
    - MIPS includes a register called the exception program counter (EPC) to contain the address of the instruction that caused the exception.
    - The instruction mfc0 (move from system control) is used to copy EPC into a general-purpose register so that MIPS software has the option of returning to the offending instruction via a jump register instruction.
      
      ```
      mfc0 $k0, $epc; $k0=$epc
      ```

Signed and unsigned operations

- Other sign and unsigned operations are:

  ```
  slt, slti, add, addi
  sltu, sltiu, addu, addiu (no overflow detection for the last two instructions)
  ```

  Notably, addiu also sign-extends its 16-bit immediate to 32-bit, when performing addition with a 32-bit register. This is because in its design, the immediate can be negative! So addiu is only an addi counterpart that ignores overflow detection.
Multiplication

- More complicated than addition
  - Accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on grade school algorithm

\[
\begin{array}{c}
0010 \\
\times 1011 \\
\hline
0010 \\
0010 \\
0010 \\
0010110
\end{array}
\]

- Negative numbers: Convert and multiply
  - There are better techniques. We however won’t look at them.

Sequential version of multiplication hardware
Sequential implementation

Multiplier0 = Bit 0 of Multiplier Register

1. Test Multiplier0
   - Multiplier0 = 1
     - 1a. Add multiplicand to product and Place the result in Product register
   - Multiplier0 = 0

2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit

32nd repetition
- No: < 32 repetitions
- Yes: 32 repetitions

Done

Question: Is implementation of 64-bit adder (ALU) necessary?

Refined implementation

Multiplier
32 bits

32-bit ALU
32 bits

Product
64 bits

Shift right
Write

Multiplier Shift right
32 bits

Control test
Question: Since the right 32-bit of the Product register is unused, can we place the multiplier in the space to save some implementation cost?

Further improvement

[Diagram showing the refined implementation process and the further improvement control flowchart]
**Further improvement**

1. **Test**
   - Product0 = 1
   - Product0 = 0

1a. Add multiplicand to the left half of the product and place the result in the left half of the Product register

2. Shift the Product register right 1 bit

32nd repetition?

- No: < 32 repetitions
- Yes: 32 repetitions

Start

**Fast multiplication hardware**

- Can we unroll the 32 additions in the implementation of multiplication hardware?
  - Notably, it required 32 clocks to finish a multiplication for sequential multiplier.
Signed multiplication

- The previous multiplication hardware only deals with multiplication of positive numbers.
- To run the previous hardware 31 iterations, leaving the signs out of the calculation.
- Then, negate the product only if the original signs disagree.

MIPS instructions for multiplication

- **mult** (multiply) and **multu** (multiply unsigned)
- MIPS provides a separate pair of 32-bit registers to contain the 64-bit product, called **Hi** and **Lo**.
- The programmer can use **mflo** (move from lo) and **mfhi** (move from hi) to place the product into normal 32-bit registers.
Division

1. Subtract the Divisor register from the dividend register and place the result in the remainder register.
2a. If the remainder is greater than or equal to 0, shift the quotient register to the left by a fixed amount, setting the new least significant bit to 0.
2b. Otherwise, set the new least significant bit to 1.
3. Shift the divisor register right by 1 bit.

---

Division

- Divisor: 1001010
- Dividend: 1001
- Quotient: 101
- Remainder: 1010

Divisor
- Shift right
- 64 bits
- 64-bit ALU
- Quotient: Shift left, 32 bits
- Remainder: Write, 64 bits
- Control test
Division

This step is redundant, because Rem < 0 always.

### Division: Revised implementation

Again, we can combine the Quotient and Remainder into one register.
Division: Further enhanced implementation

Start

1. Shift the Remainder register left 1 bit

2. Subtract the Divisor register from the left half of the Remainder register and place the result in the left half of the Remainder register

3a. Shift the Remainder register to the left, setting the new rightmost bit to 1

3b. Restore the original value by adding the Divisor register to the left half of the Remainder register and shift the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new rightmost bit to 0

End repetition

No. < 32 repetitions

Done. Shift left half of Remainder right 1 bit

See the next slide.
Signed division

- **Rule:** The dividend and the remainder must coincide in their signs.

\[
\begin{align*}
(+7) \div (+2) &= (+3)\ldots(+1) \Rightarrow (+7) = (+2)(+3) + (+1) \\
(-7) \div (+2) &= (-3)\ldots(-1) \Rightarrow (-7) = (+2)(-3) + (-1) \\
(+7) \div (-2) &= (-3)\ldots(+1) \Rightarrow (+7) = (-2)(-3) + (+1) \\
(-7) \div (-2) &= (+3)\ldots(-1) \Rightarrow (-7) = (-2)(+3) + (-1)
\end{align*}
\]

- So we only need to implement the first and third divisions, and use "negation" to obtain the results of the second and fourth divisions.

Final note on multiplication and division

- We can use "the same" hardware structure to implement multiplication and subtraction, as long as the 32-ALU can do both addition and subtraction.

- The case of "Divide by zero" shall also be considered.

- **div** (divide) and **divu** (divide unsigned)

\[
\begin{array}{c|c|c|c|c}
\text{div} & \text{rs} & \text{rt} & 0 & \text{0x1a} \\
0 & 5 & 5 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
\text{divu} & \text{rs} & \text{rt} & 0 & \text{0x1b} \\
0 & 5 & 5 & 0 & 0 \\
\end{array}
\]

(rs ÷ rt = lo ... hi)
Floating point

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g., 3.15576 × 10⁹
- Representation:
  - sign, exponent, significand:
    \((-1)^{\text{sign}} \times \text{significand} \times 2^{\text{exponent}}\)
  - more bits for significand gives more accuracy
  - more bits for exponent increases range

IEEE 754 floating point standard: sign-and-magnitude representation

- single precision: 8 bit exponent, 23 bit significand
- double precision: 11 bit exponent, 52 bit significand

Leading “1” bit of significand is implicit

Exponent is “biased” to make sorting easier

- all 0s is smallest exponent all 1s is largest
- bias of 127 for single precision and 1023 for double precision

summary: \((-1)^{\text{sign}} \times (1+\text{significand}) \times 2^{\text{exponent} - \text{bias}}\)
Single Precision Real Number

- Floating-point number (real number)

  - short (4-byte) form: single precision

  

  ![Diagram showing 32-bit format of single precision floating-point number]

  - 31 ~ 24
  - 23 ~ 16
  - 15 ~ 8
  - 7 ~ 0

  - 8-bit (biased) exponent
  - 23-bit mantissa/significand

  - Hidden one bit

  e.g., \((12)_{10} = (1100)_{2} = (1.10000000000000000000000...)_2 \times 2^3\)

  - S(ign) = 0B
  - biased exponent = \((127)_{10} (\text{bias}) + (3)_{10} = (130)_{10} = 82H\)
  - mantissa = \(1000000,00000000,00000000B\)

Double Precision Real Number

- Floating-point number (real number)

  - long (8-byte) form: double precision

  

  ![Diagram showing 64-bit format of double precision floating-point number]

  - 63 ~ 56
  - 55 ~ 48
  - 47 ~ 40
  - 39 ~ 32
  - 31 ~ 24
  - 23 ~ 16
  - 15 ~ 8
  - 7 ~ 0

  - 11-bit exponent
  - 52-bit mantissa/significand

  e.g., \((12)_{10} = (1100)_{2} = (1.10000000000000000000000...)_2 \times 2^3\)

  - S(ign) = 0B
  - biased exponent = \((1023)_{10} (\text{bias}) + (3)_{10} = (1026)_{10} = 402H\)
  - mantissa = \(1000000,00000000,00000000, ..., 00000000, 00000000B\)
Exceptions to Real Number

- **Note**
  - Having a separate sign bit facilitates the test of less than, greater than, or equal to zero.
  - Placing the exponent before the significand, as well as biased exponent format, simplifies sorting of floating-point numbers.

- **Zero**
  - all zeros format
  - sign bit can be either 1 or 0, which indicating $-0$ or $+0$.

- **Infinity**
  - exponent = all ones
  - mantissa = all zeros
  - sign bit can be either 1 or 0, which indicating $-\infty$ or $+\infty$.

---

### Exceptions to Real Number

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
<th>Exponent</th>
<th>Significand</th>
<th>Object represented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>± 0</td>
</tr>
<tr>
<td>0</td>
<td>Nonzero</td>
<td>0</td>
<td>Nonzero</td>
<td>± denormalized number</td>
</tr>
<tr>
<td>1-254</td>
<td>Anything</td>
<td>1-2046</td>
<td>Anything</td>
<td>± floating-point number</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>2047</td>
<td>0</td>
<td>± infinity</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>2047</td>
<td>nonzero</td>
<td>NaN (Not a number)</td>
</tr>
</tbody>
</table>

* “denormalized” = the leading bit in Significand is zero.
Floating-point operations

- Operations, such as additions, are somewhat more complicated
- In addition to overflow we can have “underflow”

Floating-point complexity

- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round
  - four rounding modes
    - always round up (toward +\infty)
    - always round down (toward -\infty)
    - truncate
    - round to nearest even
      - If the least significant bit retained in a halfway case would be odd, add one; if it’s even, truncate.
  - positive divided by zero yields “infinity”
  - zero divide by zero yields “not a number”
Example of Rounding with Guard Digits

- Assume three significant decimal digits.
- Add $2.34 \times 10^0$ and $2.51 \times 10^{-2}$

```
2.3400
+ 0.0251
2.3651
```

Guard round:
- $(gr)=(0\sim49)$ round down,
- $(gr)=(51\sim99)$ round up,
- $(gr)=50$ tie.

Floating-point addition

1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent.
2. Add the significands.
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent.
4. Round the significand to the appropriate number of bits.
5. Add normalization (if necessary).
Floating-point addition

The two Signs should be sent to the Big ALU in case one of the operands is negative. (See Example on page 198)

Floating-point multiplication

1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the true biased exponent.
2. Multiply the significands.
3. Normalize the product of exponents, shifting right and decrementing the exponent.
4. Round the significand to the appropriate number of bits.
5. Set the sign of the product to preserve if the signs of the original operands are the same; otherwise, reverse the sign if necessary.

Done.
More about IEEE 754

- Implementing the standard can be tricky (complicated, e.g., both overflow and underflow need to be considered)
- Not using the standard can be even worse
  - see text for description of 80x86 and Pentium bug!

Floating-point instructions in MIPS

- Floating-point addition
  - `add.s $f2, $f4, $f6`; single precision
  - `add.d $f2, $f4, $f6`; double precision
  - Note that `$f0, $f1, $f2, ...` are separate floating-point registers provided by MIPS, which can be load-and-stored by `lwc1` and `swc1`.
- Floating-point subtraction, `sub.s` and `sub.d`
- Floating-point multiplication, `mul.s` and `mul.d`
- Floating-point division, `div.s` and `div.d`
- Floating-point comparison, `c.x.s` and `c.x.d`, where `x = eq, neq, lt, le, gt and ge`.
- Floating-point branch, `bclt` (branch coprocessor true) and `bclf` (branch coprocessor false)
The IA-32 floating-point architecture is different from all other computers in the world.

- Intel provided a stack architecture with its floating-point instructions
  - Loads push numbers onto the stack
  - Operations find operands in the two top elements of the stacks
  - Stores can pop elements off the stack.
  - As anticipated, floating-point performance of IA-32 family lagged far behind other computers.

In 2001, Intel adds 144 instructions with 8 SSE2 (Streaming SIMD Extension 2) registers that can be used for floating-point operands.

- AMD later expanded the register number to 16, as part of AMD64.
- Intel then re-labeled AMD64 as EM64T for its own use.
- In addition, EM64T can load or store 128-bit aligned data to the 128-bit SSE2 register (4 single-precisions or 2 double-precisions) in one instruction.
- This new architecture can then more than double performance over Intel’s traditional stack architecture.
Concluding remarks

- Computer arithmetic is constrained by limited precision
  - So it is better to avoid adding a very big number to a very small number.
  
  \((-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0 = 1\) but \(-1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0) = 0;\)

- Bit patterns have no inherent meaning but standards do exist
  - two’s complement
  - IEEE 754 floating point

- Computer instructions determine “meaning” of the bit patterns

- Performance and accuracy are important so there are many complexities in real machines (i.e., algorithms and implementation).

Something to think about....

- Could the below statement be stopped by a divide-by-zero error?

  If \((x == 0.0)\) \(y = 17.0\) else \(y = z/x\).

  \(\text{Answer: Yes, because the examine of } x == 0.0 \text{ and division of } z/x \text{ are usually done separately from hardware aspect.}\)

- Could overflow happen for \(\gamma = z/x\), if \(z\) is approximately equal to \(x\)?

  \(\text{Answer: Yes, because some computers do } \gamma = z/x \text{ by } \gamma = z * (1/x); \text{ so if } x \text{ is too small, } 1/x \text{ can overflow.}\)
Suggestive exercises

- 3.1 ~ 3.12, 3.14, 3.27, 3.30 ~ 3.32, 3.34 ~ 3.40, 3.42 ~ 3.46

- Should you have questions on exercises, you can ask the teaching assistants (助教).