Computer Architecture. Week 8

Arithmetic for Computers and Associated Issues

Alexander Tormasov

Innopolis University

a.tormasov@innopolis.ru

October 14, 2021

Content of the class

- Recap: Numbers
- Detecting and Effects of Overflow and Underflow
- Bitwise and Shift Instructions
- Load and Storing bytes
- Multiplication and Division
- Fast Computation: MIMD and SIMD
- Floating Point Numbers and Problems
- Conversion Issues

Recap: Numbers

- Computer words are composed of bits (Bits are just bits)
  - No inherent meaning
- It gets more complicated when:
  - Numbers are finite (overflow)
  - Fractions and real numbers
  - Negative numbers
- Today our discussion contains the concept of **"exceptions"**
  - An exception can change the expected control flow in a program
  - This is important in understanding MIPS arithmetic instructions

Premise on Exceptions, Interrupts and Trap

- **Exception (applies to software):** a change in execution caused by a condition that occurs during the execution of CPU instructions:
  - Division by 0 or overflow
  - Segmentation fault (access outside program boundaries)
  - Bus error
  - Page fault (virtual memory...)

- **Interrupt (hardware exception):** a change in execution caused by an external event
  - Devices: disk, network, keyboard, etc.
  - Clock for time-sharing (multitasking)
  - These are useful events, must do something when they occur.

- **Trap:** a user-requested exception
  - Operating system call (syscall)
  - Breakpoints (debugging mode)

Premise on Exceptions, Interrupts and Trap

- Term "exception" might generalize software exception, hardware interrupt, and trap;
- Exceptions may be caused by an instruction, by external interrupts, or by hardware malfunctions;
- **Exceptions** trigger the execution of an exception handler (e.g. interrupt service routine (ISR)), which might take a significant time to execute;
- Exceptions are part of the ISA specification and must be supported by any hardware implementation;

- More details are discussed in Operating System course

Overflow: An Example of Exception

- Overflow (result too large for finite computer word):
  - Assume signed values, with the left bit reserved for sign;
  - Below, the addition of two positive numbers yields a negative number

```
  0111
+ 0001
------
  1000
```

- Note: Overflow term is somewhat misleading, it does not mean a carry "overflowed" – (Signed Overflow)
- Note 2: Overflow handling depends on a specific Instruction Set (can be handled or not)

Detecting Overflow

We are discussing signed numbers.

- Sequence of bits can be treated as signed and unsigned numbers.
- **No overflow**
  - When adding a positive and a negative number
  - When signs are the same for subtraction
- **Overflow occurs when the value affects the sign:**
  - 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

Effects of Overflow

Assume that a given Instruction Set handles an overflow.

Overflow handling phases:

- An exception is raised;
  - The control jumps to a predefined address for exception handling;
  - The interrupted address is saved for possible resumption

- Some instructions ignore overflow. For example, for MIPS:
  - `addu`, `addiu`, `subu`

Dealing with Overflow

- Some languages (e.g., C) ignore overflow
  - MIPS uses `addu`, `addiu`, `subu` instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
  - MIPS uses `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

x86 Architecture and Overflow (1/2)

- An exception might be triggered by the overflow (or underflow) caused by floating point operations
- x86 platforms provide status flags that record the occurrence of floating point exceptions

x86 Architecture and Overflow (2/2)

- The overflow flag is set when the signed result of an operation is invalid or out of range.

```
; Example 1
mov al, +127
add al, 1

; Example 2
mov al,7Fh        ; OF = 1, AL = 80h
add al,1
```

- The two examples are identical at the binary level because $7Fh$ equals +127.

Note: To determine the value of the destination operand, it is often easier to calculate in hexadecimal.

Underflow

- Underflow is a condition which occurs in a computer or similar device when a mathematical operation results in a number which is smaller than what the device is capable of storing.
- It is the opposite of overflow, which relates to a mathematical operation resulting in a number which is bigger than what the machine can store.
- Similar to overflow, underflow can cause significant errors.

Remarks

- Overflow occurs when there is a mistake in arithmetic due to the limited precision in computers.
- Example (4-bit unsigned numbers):

```
+ 15              1111
+  3              0011
--------------------
+ 18             10010
```

- But we don't have room for 5^{th}-bit, so the solution would be 0010, which is +2, and wrong.

Underflow – Example

```
#include <stdio.h>
int main(void) {
    printASCII(65);
    printASCII(-200);
    return 0;
}
void printASCII(char number){
    printf("%d\t", number);
    printf("%c\n",number);
}
```

- **Output**

  65 A

  56 8

- **Explanation**

  If the limit of char type is -128 to 127 then assigning -200 would result underflow and the output would be printed as 56 [127 - (-200 - -128)+1] and equivalent character to 56 is '8'.

Interpreting Bit Patterns

- One of the most important things to remember is that bit patterns have different meanings under different representations!
  - As a six-bit unsigned number, 101101 denotes 45 in decimal.
  - But as a two's complement number, 101101 denotes –19 in decimal.

Interpreting Bit Patterns – Example

```
main()
{
    int x = 0xFFFFFFFF; // 32-bit integers
    printf("x=%d\n", x); // Signed; prints -1
    printf("x=%u\n", x); // Unsigned; prints 4294967295
}
```

- The above program prints the same data (`0xFFFFFFFF`) twice, but under different interpretations.

Unsigned Inequalities

- Just as unsigned arithmetic instructions:

  `addu`, `subu`, `addiu`

  (really "don't overflow" instructions)

  . . . there are unsigned inequality instructions:

  `sltu`, `sltiu`

  which mean unsigned comparison:

  0x80000000 $<$ 0x7FFFFFFF signed (`slt`, `slti`)
  0x80000000 $>$ 0x7FFFFFFF unsigned (`sltu`,`sltiu`)

Bitwise Operations (1/2)

- Up until now, we have considered instructions viewing contents of registers as a single quantity
  - arithmetic (`add, sub, addi`),
  - memory access (`lw` and `sw`), and
  - branches and jumps (`jal`, `j`).

Note: We already discussed in the previous lecture

Bitwise Operations (2/2)

- **New Perspective**: view contents of register as 32 bits rather than as a single 32-bit number

- There are instructions:
  - viewing the register as two portions of 16 bits, e.g., `lui`
    - `lui`: "load upper immediate",
    - "upper" meaning the upper 16 bits
    - "immediate" meaning that you are giving it a literal value.

  - **For Example:** `lui $r, 1000`
  - **Explanation:** Moves the number 1000 into the top 16 bits of register and zeros all the other bits `r`

Bitmask

- A mask is data that is used for bitwise operations, particularly in a bit field;
- Used for an efficient implementation of bitwise instructions (see the next slide);
- Using a mask, multiple bits in a byte, nibble, word etc. can be set either on, off or inverted from on to off (or vice versa) in a single bitwise operation.
- **For Example**
  - Masks are used with IP addresses

Uses of Logical Operators (1/4)

- Note that ANDing a bit with 0 produces a 0 while ANDing a bit with 1 produces the original bit.
- This can be used to create a mask.

- **For Example:**

```
1011 0110 1010 0100 0011 1101 1001 1010
0000 0000 0000 0000 0000 1111 1111 1111
```

- **The result of ANDing:**

```
0000 0000 0000 0000 0000 1101 1001 1010
```

Uses of Logical Operators (2/4)

```
1011 0110 1010 0100 0011 1101 1001 1010
0000 0000 0000 0000 0000 1111 1111 1111
```

ANDing result:

```
0000 0000 0000 0000 0000 1101 1001 1010
```

- **Example Explanation:** The second bit-string in the example is a mask. It is used to isolate the rightmost 12 bits of the first bit-string (setting it to all 0s).

- In particular, if the first bit-string in the above example were in \$t0, then the following instruction would mask it:

```
andi  $t0,  $t0,  0xFFF
```

Uses of Logical Operators (3/4)

- Similarly, note that ORing a bit with 1 produces a 1 at the output, while ORing a bit with 0 produces the original bit.
- **This can be used to force certain bits of a string to 1s.**
- For example, if $t0 contains 0x12345678, then after this instruction:

```
ori  $t0,  $t0,  0xFFFF
```

- ... $t0 contains 0x1234FFFF (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s).

Uses of Logical Operators (4/4)

- An IP address has two components, the network address and the host address.
  - A subnet mask separates the IP address into the network and host addresses
  - The network bits are represented by the 1's in the mask, and the host bits are represented by 0's.
  - Performing a bitwise logical AND operation on the IP address with the subnet mask produces the network address.

- **For Example**

```
IP:   1101 1000 . 0000 0011 . 1000 0000 . 0000 1100(216.003.128.012)
Mask:1111 1111 . 1111 1111 . 1111 1111 . 0000 0000(255.255.255.000)
       ---------------------------------------------
      1101 1000 . 0000 0011 . 1000 0000 . 0000 0000(216.003.128.000)
```

Shift Instructions (1/2)

- MIPS shift instructions:
  - `sll` (shift left logical):
    - It shifts left and fills with 0s
    - Equivalent to an unsigned multiplication by 2^p, where p is the number of shift positions
  - `srl` (shift right logical):
    - It shifts right and fills with 0s
    - Equivalent to an unsigned division by 2^p, where p is the number of shift positions
  - `sra` (shift right arithmetic):
    - It shifts right and fills sign extending
    - It can be used to divide signed numbers by the power of 2

Note: We already talked about **`shift`** instructions in previous lectures.

Shift Instructions (2/2)

- Move (shift) all the bits in a word to the left or right by a number of bits.
- **Example: Shift left logical by 8 bits**

```
0001 0010 0011 0100 0101 0110 0111 1000
0011 0100 0101 0110 0111 1000 0000 0000
```

- **Example: Shift right logical by 8 bits**

```
0001 0010 0011 0100 0101 0110 0111 1000
0000 0000 0001 0010 0011 0100 0101 0110
```

- **Example: Shift right arithmetic by 8 bits (+ number)**

```
0001 0010 0011 0100 0101 0110 0111 1000
0000 0000 0001 0010 0011 0100 0101 0110
```

- **Example: Shift right arithmetic by 8 bits (− number)**

```
1001 0010 0011 0100 0101 0110 0111 1000
1111 1111 1001 0010 0011 0100 0101 0110
```

Uses of Shift Instructions (Bitmasking)

- Let us assume that in $t0 we have the following content – in brown we have what we want to isolate:

  0001 0010 0011 0100 **0101 0110** 0111 1000

  We can use:

```
sll $t0, $t0, 16
srl $t0, $t0, 24
```

  0001 0010 0011 0100 **0101 0110** 0111 1000
  0101 0110 0111 1000 0000 0000 0000 0000
  0000 0000 0000 0000 0000 0000 **0101 0110**

Loading and Storing bytes (1/2)

- The MIPS instruction set includes dedicated load and store instructions for accessing memory.
- MIPS uses indexed addressing to refer memory.
  - The **address operand** specifies a signed constant and a register.
  - These values are added to generate the effective address.
- The MIPS **load byte** instruction `lb` transfers one byte of data from main memory to a register.

```
lb $t0, 20($a0)     # $t0 = Memory[$a0 + 20]
```

- The **store byte** instruction `sb` transfers the lowest byte of data from a register into main memory.

```
sb $t0, 20($a0)     # Memory[$a0 + 20] = $t0
```

Loading and Storing bytes (2/2)

- load byte: `lb`
- load byte unsigned: `lbu`

- **For Example:**
  - Read `0xFE` and to extend it from 8 to 32 bits
    - `lb` sign-extends to `0xFFFFFFFE (-2 decimal)`
    - `lbu` 0-extends to `0x000000FE (254 decimal)`

Loading and Storing Words

- A "word" means the number of bits that can be transferred at one time on the data bus, and stored in a register
- 1 MIPS word = 32 bits
- Instructions for loading and storing words:

```
lw $t0, 20($a0)   # $t0 = Memory[$a0 + 20]
sw $t0, 20($a0)   # Memory[$a0 + 20] = $t0
```

- Most programming languages support several 32-bit data types.
  - Integers
  - Single-precision floating-point numbers
  - Memory addresses, or pointers
- We'll assume words are the basic unit of data.

Multiplication

- In MIPS, we multiply registers, so:
  - 32-bit value × 32-bit value = 64-bit value
- Syntax of multiplication (signed):
  - `mult r1, r2`
  - where
    - `r1` and `r2` are the two 32 bits registers holding the two operands
  - The 64 bits output is stored in two special registers:
    - `hi` for the upper half
    - `lo` for the lower half
  - Next Step (legacy instructions): Move the result of a multiplication into a general purpose register:
    - `mfhi` to move the upper half in another register
    - `mflo` to move the lower half in another register

Multiplication – Example

- Example: in Java: `a = b * c;`
- In MIPS:
  - Let `b` be `$s2`; let `c` be `$s3`; and let `a` be `$s0` and `$s1` (since it may be up to 64 bits)

```
mult $s2, $s3          # b * c
mfhi $s0               # upper half of product into $s0
mflo $s1               # lower half of product into $s1
```

- Remarks
  - The modern MIPS assembly mostly use `mul` instead of `mult`.
  - The `mul` pseudo instruction makes it look as if MIPS has a 32-bit multiply instruction that places its 32-bit result.
  - The bits of `hi` are not examined nor saved.
  - There is no overflow checking.

Division

- MIPS syntax of division (signed):
  - `div register1, register2`
  - Divides 32-bit values in register 1 by 32-bit value in register 2:
    - puts remainder of division in `hi`
    - puts quotient of division in `lo`
- Notice that this can be used to implement both the division operator (/) and the modulo operator (%)

Division – Example

- Example: in Java

```
a = c / d;
b = c % d;
```

- In MIPS:

```
div $s2, $s3          # lo = c/d, hi = c%d
mflo $s0              # get quotient
mfhi $s1              # get remainder
```

- Remarks
  - Logical instructions are faster as compared to arithmetic operations.
  - For Example: When we do multiplication, compiler automatically converts into shift instructions.
  - **Reason:** Shift takes less cycles as compared to `mult`.

Unsigned Instructions and Overflow

- MIPS also has versions of these two arithmetic instructions for unsigned operands: `multu` and `divu`
- Determines whether or not the product and quotient are changed if the operands are signed or unsigned.
- MIPS does not check overflow on ANY signed/unsigned multiply, divide instructions
- Up to the software to check `hi`

32/64-bit Instructions for Multiply and Divide

Mnemonic	Instruction	Defined in MIPS ISA
MUL	Multiply word, Low part, signed	MIPS32 Release 6
MUH	Multiply word, High part, signed	MIPS32 Release 6
MULU	Multiply word, How part, Unsigned	MIPS32 Release 6
MUHU	Multiply word, High part, Unsigned	MIPS32 Release 6
DMUL	Multiply doubleword, Low part, signed	MIPS64 Release 6
DMUH	Multiply doubleword, High part, signed	MIPS64 Release 6
DMULU	Multiply doubleword, How part, Unsigned	MIPS64 Release 6
DMUHU	Multiply double word, High part, Unsigned	MIPS64 Release 6
DIV	Divide words, signed	MIPS32 Release 6
MOD	Modulus remainder word division, signed	MIPS32 Release 6
DIVU	Divide words, Unsigned	MIPS32 Release 6
MODU	Modulus remainder word division, Unsigned	MIPS32 Release 6
DDIV	Divide doublewords, signed	MIPS64 Release 6
DMOD	Modulus remainder doubleword division, signed	MIPS64 Release 6
DDIVU	Divide doublewords, Unsigned	MIPS64 Release 6
DMODU	Modulus remainder doubleword division, Unsigned	MIPS64 Release 6

Fast Computation with Parallelism

- **Goal:** We want to do fast computation.
- We can go fast with multiple processors.

Multiple Instruction Multiple Data (MIMD)

- MIMD contains multiple fully-featured, independent processing cores, whether they are on the same chip (multi-core) or different ones (multi-processor).
- For Example: each unit have separate set of instructions. One may be adding, another multiplying, yet another evaluating a branch condition, and so on.
- This makes more difficult to architect the solution

MIMD – Example

- Computers with multiple CPUs or single CPUs with dual cores are examples of MIMD architecture.
- Hyper-threading also results in a certain degree of MIMD performance as well.

Single Instruction Multiple Data (SIMD)

- The MIPS SIMD Architecture (MSA) technology incorporates a software-programmable solution into the CPU.
- All parallel units share the same instruction on different data elements.
- This programmable solution increased the system flexibility.

SIMD – Example

- Current generation mobile and home entertainment devices must deliver extremely high-quality audio, video, image, and graphics performance in order to be competitive.
- These advanced processing requirements are optimized and accelerated with SIMD
- It is an important technology for modern CPU designs that improves performance.

Tensor Processing Unit (TPU)

- The tensor processing unit (TPU) was announced in May 2016 by Google
- TPU is an AI accelerator application-specific integrated circuit (ASIC) for neural network machine learning
- TPU is optimized for high volume low precision computations (typically 8 bits precision)
- The chip has been specifically designed for Google's TensorFlow framework, a library which is used for machine learning applications such as neural networks
- However, Google still uses CPUs and GPUs for other types of machine learning.
- Other AI accelerator designs are appearing from other vendors also and are aimed at embedded and robotics markets.

Floating Point Numbers (1/3)

- Used to represent real numbers
- Very similar to scientific notation

$$3.5 \times 10^6, 0.82 \times 10^{-5}, 75 \times 10^6, \ldots$$

- Both decimal numbers in scientific notation and floating point numbers can be normalized:

$$3.5 \times 10^6, 8.2 \times 10^{-6}, 7.5 \times 10^7, \ldots$$

Floating Point Numbers (2/3)

- Standard definition (i.e., architecture independent)
- Single precision (uses 32 bits to represent sign, exponent and Significand)

Sign (1 bit)	Exponent (8 bits)	Significand (23 bits)

- Double precision (uses 64 bits)

Sign (1 bit)	Exponent (11 bits)	Significand (52 bits)

$$(-1)^S * (1 + Significand) * 2^{(Exponent-127)}$$

  - Double precision identical, except with exponent bias of 1023

Floating Point Numbers (3/3)

- **Sign bit**
  - 0 indicates a positive number
  - 1 indicates a negative number
- **Exponent**
  - 8 bits for single precision (32-bits)
  - 11 bits for double precision (64-bits)
  - With 8 bits, we can represent exponents between -126 and + 127
    - All-zeroes value is reserved for the zeroes and denormalized numbers
    - All-ones value are reserved for the infinities and NaNs (Not a Number)
- **Significand**
  - Also known as fraction, coefficient, or mantissa
  - The use of term ”mantissa” to refer to significant digits in a floating point number or scientific notation is discouraged by some in favor of ”significand”

Encoding a Signed Floating Point Number – Example

We want to represent 7:

- Conversion to binary: 111
- Normalization: 1.11×2^2
- Sign bit: 0 (positive)
- Biased exponent: $127 + 2 = 1111111_2 + 10_2 = 10000001_2$
- We eliminate the 1 in front
- Significand: $1100\ldots0$

0 10000001 11000000000000000000000

Encoding a Floating Point Number

We want to represent 1/2:

- Conversion to binary: 0.1
- Normalization: 1.0×2^{-1}
- Sign bit: 0
- Biased exponent: $127 - 1 = 01111110_2$
- We eliminate the 1 in front
- Significand: 0000...0

0 01111110 00000000000000000000000

FP Addition – Decimal – Example

$$5.25 \times 10^3 + 1.22 \times 10^2 = ?$$

- Denormalize number with smaller exponent:

$$5.25 \times 10^3 + 0.122 \times 10^3$$

- Add the numbers:

$$5.25 \times 10^3 + 0.122 \times 10^3 = 5.372 \times 10^3$$

- The result is normalized.
- **NOTE:** If result is not normalized we need to normalize it.

FP Addition – Binary – Example

$$1.001 \times 2^3 + 1.0 \times 2^1 = ?$$

- Denormalize number with smaller exponent:

$$1.001 \times 2^3 + 0.01 \times 2^3$$

- Add the numbers:

$$1.001 \times 2^3 + 0.01 \times 2^3 = 1.011 \times 2^3$$

- The result is already normalized

- Binary Representation:

0 10000010 01100000000000000000000

FP Subtraction – Example

$$1.01 \times 2^2 - 1.1 \times 2^1 = ?$$

- Denormalize number with smaller exponent:

$$1.01 \times 2^2 - 0.11 \times 2^2$$

- Subtract the numbers:

$$1.01 \times 2^2 - 0.11 \times 2^2 = 0.10 \times 2^2$$

- Normalize the results:

$$0.10 \times 2^2 = 1.0 \times 2^1$$

- Binary Representation:

0 10000000 10000000000000000000000

FP Problems (1/2)

- The chosen sizes of exponent and significand give MIPS computer arithmetic an extraordinary range.
- But, it is still possible for numbers to be too large.
- Thus, overflow interrupts can occur in floating-point arithmetic.
- Overflow means that the exponent is too large to be represented in the exponent field.

FP Problems (2/2)

- Smallest positive single precision normalized number:

$$1.0000\ 0000\ 0000\ 0000\ 0000\ 000_2 \times 2^{-126}$$

- Any number that is smaller then above number can raise exception.
- This situation occurs when the negative exponent is too large to fit in the exponent field.
- This situation is called **underflow**.

FP Solution

- One way to reduce the chances of underflow or overflow is to have another format with a larger exponent: **double precision**

Floating Point Instructions

- Separate floating point instructions:
  - Single Precision:
    `add.s`, `sub.s`, `mul.s`, `div.s`
  - Double Precision:
    `add.d`, `sub.d`, `mul.d`, `div.d`
- These instructions are far more complex than their integer counterparts, so they can take much longer to execute.

Architectural Issues

- It is inefficient to have different instructions taking vastly differing amounts of time.
- **Generally**, a particular piece of data will not change from FP to int, or vice versa, within a program. So only one type of instruction may be used on it.
- Performing fast floating point operations requires lots of hardware as compared to integers.

Solution – Actual Architectures

- In the end of 1980, computers actually contained multiple separate chips:
  - **Processor:** handles all the normal stuff
  - **Coprocessor 1:** handles floating point operations
- Today, most CPUs contain the floating point co-processor inside it

Coprocessor 1

- Contains 32, 32-bit registers: `$f0`, `$f1`, ...`$f31`
- Registers are specified as

  `.s` (single precision)
  `.d` (double precision)

- Separate load and store instructions:

  `lwc1` ("load word coprocessor 1")
  `swc1` ("store word coprocessor 1")

- Double Precision (DP): By convention, even/odd pair contain one DP FP number: `$f0`/`$f1`, `$f2`/`$f3`,..., `$f30`/`$f31`

- Instructions to move data between main processor and coprocessors:
  - `mfc0`, `mtc0`, `mfc1`, `mtc1`, etc.

Example of FP Computations

Addition

```
add.s $f1, $f2, $f3  # Single-precision $f1 = $f2 + $f3
add.d $f2, $f4, $f6  # Double-precision $f2 = $f4 + $f6
```

Register transfers

```
mtc1 $t0, $f0  # $f0 = $t0
mfc1 $t0, $f0  # $t0 = $f0
```

Transferring data between the registers and memory

```
lwc1 $f2, 0($a0)  # $f2 = M[$a0]
swc1 $f4, 4($sp)  # M[$sp+4] = $f4
```

Note: More examples and practice in tutorial session.

Special Numbers

- What do I get if I compute

  `sqrt(-4.0)` or `0/0`?

  - If infinity is not an error, these should not be either.
    - Called Not a Number (`NaN`)
    - Representation: 1 11111111 11111111111111111111

- Why is this useful?
  - `NaN` is used to present signal error conditions such as overflows, underflows, division by 0, and so on
  - When one of these error conditions occur in an operation, the hardware generates a `NaN` as its result rather than signaling an exception

Summary of Floating Point Numbers

- IEEE 754 Encoding of FP Numbers

Single Precision		Double Precision		Object Represented
Exponent	Significand	Exponent	Significand	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1-254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Rounding

- When we perform math on real numbers, we have to worry about rounding to fit the result in the significand field.
- The FP hardware carries two extra bits of precision, and then round to get the proper value
- Rounding also occurs when converting a double to a single precision value, or converting a floating point number to an integer

IEEE Rounding Modes

1. Round towards $+\infty$
   - ALWAYS round "up":
     $2.001 \rightarrow 3$
     $-2.001 \rightarrow -2$
2. Round towards $-\infty$
   - ALWAYS round "down":
     $1.999 \rightarrow 1$,
     $-1.999 \rightarrow -2$
3. Truncate
   - Just drop the last bits (round towards 0)
4. Round to (nearest) even
   - Normal rounding, almost

Round to Even

- Round like you learned in high school
- Except if the value is right on the borderline, in which case we round to the nearest EVEN number
  $2.5 \rightarrow 2$
  $3.5 \rightarrow 4$
- Insures fairness on calculation
  - This way, half the time we round up on tie, the other half time we round down
  - Ask statisticians...
- This is the default rounding mode

Conversion Issues # 1

- Does converting a **float** → **int** → **float** result into the same **float** number?

Converting float → int → float

```
if (i == (float)((int) i)
{
     printf (‘‘true’’);
}
```

- Will **not** always print “true”
- Small **float**ing point numbers (<1) don’t have integer representations
- For other numbers, rounding errors

Conversion Issues # 2

- Does converting a **int** → **float** → **int** result into the same **int** number?

Converting int → float → int

```
if (f == ((int) (float) f)
{
     printf (‘‘true’’);
}
```

- Will **not** always print “true”
- Large values of integers don’t have exact floating point representations
- What about double?

Proposed Issue

- Are FP additions and subtraction associative ?

$$(x+y)+z = x+(y+z)$$

FP Associative

- Are FP additions and subtraction associative ? NO!

$$x = -1.5 \times 10^{38}, y = 1.5 \times 10^{38}, z = 1.0$$

$$x + (y + z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0)$$

$$= -1.5 \times 10^{38} + (1.5 \times 10^{38}) = 0.0$$

$$(x + y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0$$

$$= (0.0) + 1.0 = 1.0$$

Why?

- FP **approximates** the real result!
- In the above example: 1.5×10^{38} is so much larger than 1.0 that $1.5 \times 10^{38} + 1.0$ in floating point representation is still 1.5×10^{38}

Summary

Today we discussed:

- the underflow and overflow issues with respect to different architectures;
- the loading and storing instructions;
- bitmasking;
- fast computation via parallelism;
- floating-point numbers and associated issues

Acknowledgements

- This lecture was created and maintained by Muhammad Fahim, Giancarlo Succi, Alexander Tormasov, and Artem Burmyakov