University of Nottingham
UK | CHINA | MALAYSIA

Computer Arithmetic

Dr. Heng Yu

AY2022-23, Spring Semester
COMP1047: Systems and Architecture
Week 2

Recall from your CSF

What is the decimal value of the binary number 011_2 ?

What is the binary value of the decimal number 8_{10} ?

What is the binary value of the decimal number -8_{10} ?

Outline

- **Number Formats**
- **Representing Negative Numbers**
  - **Overflow**
- **Shift and multiplication**
- **Floating Point Numbers**

Learning Objective

After this lecture, you should be able to

- Understand and implement the format conversion between decimal and binary formats
- Representing Negative Numbers
  - Know sign/magnitude and 2's complement formats to represent a negative binary number, and sign extension
  - Understand their pros and cons.
  - Implement the related computation and conversions.
  - Identify the overflow conditions.

- Shift and multiplication
  - Implement arithmetic and logical shift. Implement binary multiplications.
- Floating Point Numbers
  - Know scientific notation, IEEE754 standard
  - Implement and convert single-precision binary values.
  - Know the special cases.

Outline

- **Number Formats**

- **Representing Negative Numbers**
  - **Overflow**

- **Shift and multiplication**

- **Floating Point Numbers**

Binary Counting

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101

Decimal	Binary
6	110
7	111
8	1000
9	1001
10	1010
11	1011

Decimal	Binary
12	1100
13	1101
14	1110
15	1111
16	10000
17	10001

- In binary, the result of 1 + 1 is 0 and carries a 1 to the next digit

Binary to Decimal (Integers)

- Each binary digit corresponds to a power of 2

Place	7th	6th	5th	4th	3rd	2nd	1st	0th
Weight	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	128	64	32	16	8	4	2	1

- Where the digit is 1, we add the corresponding weight
- Example: Convert $1100\ 1010_2$ into decimal

$$\begin{aligned} 1100\ 1010_2 &= 1 \times 128 + 1 \times 64 + 0 \times 32 + 0 \times 16 \\ &\quad + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 \\ &= 128 + 64 + 8 + 2 = 202_{10} \end{aligned}$$

Binary to Decimal (Fractional)

- Each binary digit corresponds to a power of -2

Place	-1st	-2nd	-3rd	-4th	-5th	-6th	-7th	-8th
Weight	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
	0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125	0.00390625

- Where the digit is 1, we add the corresponding weight
- Example: Convert 0.101_2 into decimal

- $0.101_2 = 1 \times 2^{-1} + 1 \times 2^{-3} = 0.625_{10}$

Binary to Decimal (Integer + Fractional)

- Convert the integer and fractional numbers separately.

Example: Convert 1011.101_2 to Decimal

$$= 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 + 1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3}$$

$$= 8 + 0 + 2 + 1 + 0.5 + 0 + 0.125$$

$$= 11.625_{10}$$

Decimal to Binary (Integers)

- Repeatedly divide by 2, until we reach 0
- The first remainder calculated is placed at rightmost for the binary number.
- Example: Convert 101_{10} into binary

101	Remainder
50	1
25	0
12	1
6	0
3	0
1	1
0	1

- $101_{10} = 1100101_2$

Decimal to Binary (Fractional)

- Repeatedly multiply by 2, until we reach x.0, or infinity
- The first integer calculated is placed at leftmost after the decimal point.
- Example: Convert 0.75_{10} into binary

- $0.75_{10} = 0.11_2$

Decimal to Binary (Fractional)

- Example: Convert 0.3_{10} into binary

	product	integer part
0.3×2	0.6	0
0.6×2	1.2	1
0.2×2	0.4	0
0.4×2	0.8	0
0.8×2	1.6	1
0.6×2	1.2	1
0.2×2	0.4	0
0.4×2	0.8	0
0.8×2	1.6	1
…	…	…

Hence $0.3_{10} = 0.0\ 1001\ 1001\ 1001\ 1001\ldots_2$

- Digits in fraction part repeats forever, 0.3 cannot be expressed exactly by finite digits

Decimal to Binary (Integer + Fractional)

- Convert the integer and fractional numbers separately.

Convert the decimal number 95.125_{10} to binary

$95_{10} = ?_2$

$0.125_{10} = ?_2$

$95.125_{10} = ?.?_2$

Decimal to Binary (Integer + Fractional)

- Convert the integer and fractional numbers separately.

Convert the decimal number 95.125_{10} to binary

$95_{10} = 1011111_2$

$0.125_{10} = 0.001_2$

$95.125_{10} = 1011111.001_2$

Outline

- Number Formats
- Representing Negative Numbers
  - Overflow
- Shift and multiplication
- Floating Point Numbers

Sign/Magnitude Representation

- So far, we've dealt with unsigned numbers
  - How are negative numbers represented on a computer?
- One way to represent negative number is called sign/magnitude.
- For an N-bit binary number:

1 sign bit, N-1 magnitude bits

Sign bit is the most significant bit (leftmost)

  - Negative number: sign bit = 1
  - Positive number: sign bit = 0

- Rest of the bits are numerical value of the number

Sign/Magnitude Representation

1 sign bit, N-1 magnitude bits

Sign bit is the most significant bit (leftmost)

- Rest of the bits are numerical value of the number
- Hence, in a binary number with eight bits, the magnitude can range from 0000000 (0) to 1111111 (127)
- Thus numbers ranging from -127_{10} to $+127_{10}$ can be represented, once the sign bit (the eighth bit) is added

Problems

- Two representations of zero [0000, 1000]
- Difficulties in arithmetic operations:
  - 1101 (-5) + 0011 (3) = 0000

2's Complement Representation

The most significant bit still indicates the sign

Same as unsigned binary, but the value of the most significant bit is -2^{N-1}

Example: What is the value of 10110

- Unsigned: 10110 = 16 + 4 + 2 = 22
- Twos complement 10110 = -16 + 4 + 2 = -10

Single representation of zero (0000)

Arithmetic works fine

- 1011 (-5) + 0011 (3) = 1110 (-8 + 4 + 2) = -2

2's Complement Representation

How to convert a 2's complement binary to its decimal value?

- Assume in an 8-bit two's-complement numeral system

8-Bit Two's Complement ($-128 \le x < 127$)

	MSB							LSB
Bit	7^{th}	6^{th}	5^{th}	4^{th}	3^{rd}	2^{nd}	1^{st}	0^{th}
Weight	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

- What does 0000 0001 represent? 1
- What does 1111 1111 represent? -1
- What does 0101 1011 represent? 91

2’s Complement Representation

How to convert a positive 2’s complement binary to its negative value?

E.g., what is the 2’s complement of the binary number 011010

First method	**Complement each digit to get the first complement**	100101
	Add one to the first complement.	100110
Second Method	Start from the least significant bit, locate the first digit with value 1	011010
	Complement each digit after the first digit with value 1.	100110

2's Complement Representation

- Exercise: Represent -27_{10} into 2's complement binary format

2’s Complement Representation

- The two's-complement numeral system is the most common method of representing signed integers on computers
  - Advantages: The fundamental arithmetic operations of addition, subtraction, and multiplication are identical to those for unsigned binary numbers
  - This property makes the system both simpler to implement and capable of easily handling higher precision arithmetic
  - Also, zero has only a single representation 0, eliminating the subtleties associated with negative zero
- All modern processors primarily use two’s complement
  - Includes all the MIPS’s integer arithmetic instructions
  - Different instruction variants for signed and unsigned

Sign Extension

- Sign extension is the operation of increasing the number of bits of a binary number while preserving the number's sign (positive/negative) and value
  - This is done by appending digits to the most significant side of the number, following a procedure dependent on the particular signed number representation used
- In MIPS, all arithmetic immediate values are sign-extended
- e.g., If six bits are used to represent the number "00 1010" (decimal +10) and the sign extension operation increases the word length to 16 bits, then the new representation is simply "0000 0000 0000 1010“ – padding the left side with 0s

Sign Extension

- Question: If ten bits are used to represent the value "11 1111 0001" (decimal -15) in two's complement, and the sign extension operation increases the word length to 16 bits, what is the new representation?

Sign Extension

- Solution: If ten bits are used to represent the value "11 1111 0001" (decimal -15) in two's complement, and the sign extension operation increases the word length to 16 bits, the new representation is "1111 1111 1111 0001" – padding the left side with 1s

Long Addition

Long Addition in Decimal

		6	2	9	5	1	4	1	3	
+		2	8	1	8	2	8	1	7	
	0	1	1	1	0	1	0	1		Carry
		9	1	1	3	4	2	3	0	

0111 0110 + 1101 0101

Long Addition in Binary

		0	1	1	1	0	1	1	0	$= 118$
+		1	1	0	1	0	1	0	1	$= 213$
	1	1	1	1	0	1	0	0		Carry
	1	0	1	0	0	1	0	1	1	$= 331$

Overflow

- One issue in computer arithmetic is dealing with finite amounts of storage, such as 32-bit MIPS registers
- Overflow occurs when the result of an operation is too large to be stored
- For an unsigned number, overflow happens when the last carry (1) cannot be accommodated. E.g. 1111+0001

Overflow

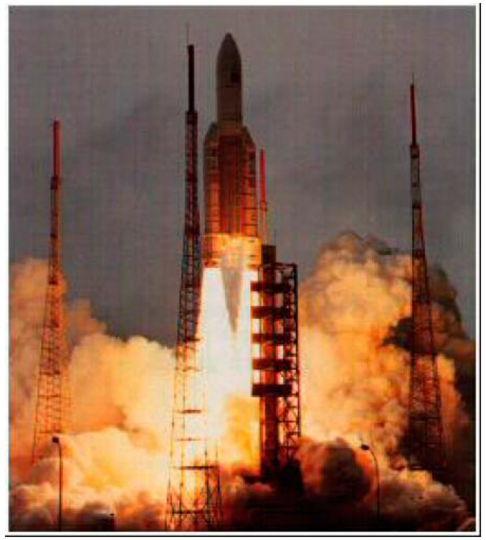
- For a signed number, overflow occurs when
  - Adding two positives yields a negative. E.g. (0111 + 0001)
  - Or, adding two negatives gives a positive. E.g. (1000 + 1000)
  - Or, subtract a negative from a positive gives a negative
  - Or, subtract a positive from a negative gives a positive

- One way to detect overflow is to check whether the sign bit is consistent with the sign of the inputs when the two inputs are of the same sign – if you added two positive numbers and got a negative number, something is wrong, and vice versa

Overflow

Ariana 5

- In 1996, the European Space Agency's Ariane5 rocket was launched for the first time… and it exploded 40 seconds after liftoff
- It turns out that the Ariane5 used software designed for the older Ariane4
  - The Ariane4 stored its horizontal velocity as a 16-bit signed integer
  - But the Ariane5 reaches a much higher velocity, which caused an overflow in the 16-bit quantity
- The overflow error was never caught, so incorrect instructions were sent to the rocket boosters and main engine
- For a modern CPU like MIPS, it detects overflow with an exception (an internal interrupt signal to the CPU)
  - Control jumps to predefined address for exception
  - Interrupted address is saved for resumption



Outline

- Number Formats

- Representing Negative Numbers
    - Overflow

- Shift and multiplication

- Floating Point Numbers

Shift Operations

- Shift operations shift a word a number of places to the left or right
- Bits which are shifted out, just disappear
- E.g. on 4 bits: 1011 shifted left 1 bit, results in 0110.

- Logical shift: in right shift, 0s are filled in empty positions.
  - E.g., 1011 shift right 1 bit => 0101
- Arithmetic shift: in right shift, sign bits are filled in empty positions.
  - E.g., 1011 shift right 1 bit => 1101

Shift Operations

- For unsigned and 2's complement numbers, logical/arithmetic shift left by 1 is multiplication by 2 if there is no overflow
  - e.g. 0011 (3) shifted left 1 bit results in 0110 (6)
  - e.g. 1100 (-4) shifted left 1 bit results in 1000 (-8)

- For unsigned numbers, logical shift right by 1 is division by 2, ignoring remainder.
  - e.g. 0101 (5) shifted right 1 bit results in 0010 (2)
  - For 2's complement numbers, logical shift right by 1 does not correspond to dividing 2.
  - e.g. 1100 (-4) shifted right results in 0110 (6)

- For 2's complement numbers, arithmetic shift right by 1 performs division by 2.
  - e.g. 1100 (-4) shifted right arithmetical results in 1110 (-2)

Multiplication

- Binary multiplication similar to decimal multiplication – multiplying with each bit and add the (shifted) results together.
- If we multiply an m-bit with an n-bit number, we need $m+n$ bits to store the result
  - Multiplying 4-digit numbers needs 8 digits
  - e.g. 1101_2 x 1011_2

Long Multiplication in Binary

$$
\begin{array}{ccccccccccl}
 & & & & & 1 & 1 & 0 & 1 & & a = 13_{10} \\
\times & & & & & 1 & 0 & 1 & 1 & & b = 11_{10} \\
\hline
 & & & & & 1 & 1 & 0 & 1 & & \\
 & & & & 1 & 1 & 0 & 1 & & & \\
 & & & 0 & 0 & 0 & 0 & & & & \\
+ & & 1 & 1 & 0 & 1 & & & & & \text{Partial Sums} \\
\hline
= & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & & c = 143_{10}
\end{array}
$$

Outline

- Number Formats
- Representing Negative Numbers
  - Overflow
- Shift and multiplication
- Floating Point Numbers






Scientific Notation

- Scientific notation
  - A real number notation that renders with a single digit to the left of the decimal point
- General form: $\pm m \times b^e$
  - m is called the significand
  - b is the base number, can be any positive integer
  - Exponent e is an integer
- E.g.
  - 2.73×10^7
  - 0.273×10^{-6}

Normalized Scientific Notation

- Normalized scientific notation
  - The exponent *e* is chosen so that the absolute value of *m* remains at least one but less than ten ($1 \leq |m| < 10$)
  - A number in proper scientific notation that has no leading zeros
- E.g.
  - 2.73×10^{7}
  - But 0.273×10^{-6} is not normalized
- Normalized scientific notation for binary
  - A number in floating-point notation that has no leading 0s
  - E.g. $11.101_2 = 11.101_2 \times 2^0$
    $= 1.1101_2 \times 2^1$ (normalized form)
  - Fraction: the value placed in the fraction field (1101 in this case)

Normalized Scientific Notation

General form: $\pm 1.xxxxxxxx \times 2^{yyyy}$

Binary point Fraction (mantissa) Base exponent

- When we normalize a non-zero binary number, we'll always have a 1 to the left of the binary point
- To represent a real (floating point) number in a fixed number of digits, we need to decide how to allocate some fixed number of bits to both the fraction *xxxx* and the exponent *yyyy*
  - This is a tradeoff between precision and range

Floating Point Standard

- The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation established in 1985
- It gives exact layout of bits, and defines basic arithmetic
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)
- Now almost universally adopted

IEEE 754 Floating Point Standard

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

- A type of sign and magnitude representation
- S: sign bit of the fraction (0 for +, 1 for -)

S	Biased Exponent	Fraction

- Single-precision (32-bit)

	8 bits	23 bits
s	e	fraction

- Double-precision (64-bit)

	11 bits	52 bits
s	e	fraction

IEEE 754 Floating Point Standard

S	Biased Exponent	Fraction

- Biased Exponent
  - A fixed value, called bias, is subtracted from the field to get the true exponent value
  - Typically, bias = $2^{k-1} - 1$, where k is the number of bits in the exponent field
  - E.g. the biased representation $0000\ 0101_2$ with a bias 127, its actual exponent value is: 5-127=-122 ($00000101_2=5_{10}$)
- Biased Exponent = Actual Exponent + Bias
- IEEE 754 Single precision, bias=127, double precision, bias=1023

A Single Precision Example

- Represent the following binary floating point in the Single-Precision Floating Point Format
  - -10.101_2
- Step 1: normalize number $(-1) \times 1.0101 \times 2^1$
- Step 2:
  - Sign: 1, fraction F=0101
  - Biased Exponent E= Actual Exponent + Bias

$$=1+127 = 128_{10} = 1000\ \ 0000_2$$

	8 bits	23 bits
1	E=1000 0000	F=0101 0000 0000 0000 0000 000

Try By Yourself

- Represent the following binary floating point in the Single-Precision Floating Point Format
  - 1011.1101_2

- Step 1: normalize number $(1) \times 1.0111101 \times 2^3$
- Step 2:
  - Sign: 0, fraction F=0111101
  - Biased Exponent E= Actual Exponent + Bias

$$=3+127 = 130_{10} = 1000\ \ 0010_2$$

	8 bits	23 bits
0	E=1000 0010	F=0111 1010 0000 0000 0000 000

Try By Yourself

- Represent the following binary floating point in the Single-Precision Floating Point Format
  - 1011.1101_2

Special Values (Single Precision)

- Smallest positive normalized number: $1.0_2 \times 2^{-126}$
- Least negative normalized number: $-1.0_2 \times 2^{-126}$
- Zero: $\pm 1.0_2 \times 2^{-127}$
- Largest positive normalized number:

  $1.1111\ 1111\ 1111\ 1111\ 1111\ 111_2 \times 2^{127}$
- Most negative normalized number:

  $-1.1111\ 1111\ 1111\ 1111\ 1111\ 111_2 \times 2^{127}$
- $\pm\infty$: $\pm 1.0 \times 2^{128}$
- NaN: $1.f \times 2^{128}$ where $f \neq 0$

NaN (Not-a-Number)

- A numeric data type value representing an undefined or unrepresentable value, especially in floating-point calculations
- A few cases where we get NaN:
  - 0.0 / 0.0
  - $\pm\infty$ / $\pm$ ∞,
  - $0 \times \pm\infty$,
  - $-\infty + \infty$,
  - sqrt (-1.0),
  - log(-1.0)

```c
void nanFun()
{
    printf("0.0/0.0: %f\n", 0.0/0.0);
    printf("inf/inf: %f\n", (1.0/0.0)/(1.0/0.0));
    printf("0.0*inf: %f\n", 0.0*(1.0/0.0));
    printf("-inf + inf: %f\n", (-1.0/0.0) + (1.0/0.0));
    printf("sqrt(-1.0): %f\n", sqrt(-1.0));
    printf("log(-1.0): %f\n", log(-1.0));

    float n = log(-1.0);
    printBits(sizeof(n), &n);
}
```

Putting It All Together

- Convert the following decimal number to single-precision floating point representation in binary format

  $x = -13.825$

- Hint: first convert to the binary correspondence, then change it into single-precision floating point representation

Putting It All Together

$x =$ -13.825

Integer part: $13_{10} = 1101_2$, *fraction part:*

0.825	integer
$\cancel{1}.65$	1
$\cancel{1}.3$	1
0.6	0
$\cancel{1}.2$	1
0.4	0
0.8	0
$\cancel{1}.6$	1
$\cancel{1}.2$	1

repeats ...

therefore, $x = 1101.110\dot{1}00\dot{1}_2 = 1.101110\dot{1}00\dot{1} \times 2^3 {}_2.$

Exponent: $3 + 127 = 130 = 1000\ 0010.$

	8 bits	23 bits
1	E=1000 0010	F=101 1101 0011 0011 0011 0011

Equality Conditions

- In most computers, $0.3+0.2 \neq 0.5$
  - Neither 0.3 nor 0.2 has exact binary representation
- Never directly test floating point numbers for equality (i.e. 0.3+0.2 == 0.5)
  - Using the following statement instead

$$|0.3+0.2-0.5|<\varepsilon$$

  where ε is a very small number

- You don't have to worry about it as a user. MIPS instructions already implement the details.

Summary

- Conversion between binary and decimal numbers
- Representing negative numbers
  - Sign/magnitude
  - 2's complement
  - Sign extension
- Binary additions and overflow, multiplication
- Shift operations
  - Logical shift
  - Arithmetic shift
- Floating point number representation.
  - IEEE 754 standard for single precision.
  - Special values.

University of Nottingham
UK | CHINA | MALAYSIA

Stay Tuned.