Computer Mathematics

Week 4
Signed integer representations and arithmetic

KUAS 京都先端科学大学
KYOTO UNIVERSITY of ADVANCED SCIENCE
Department of Mechanical and Electrical System Engineering

last week

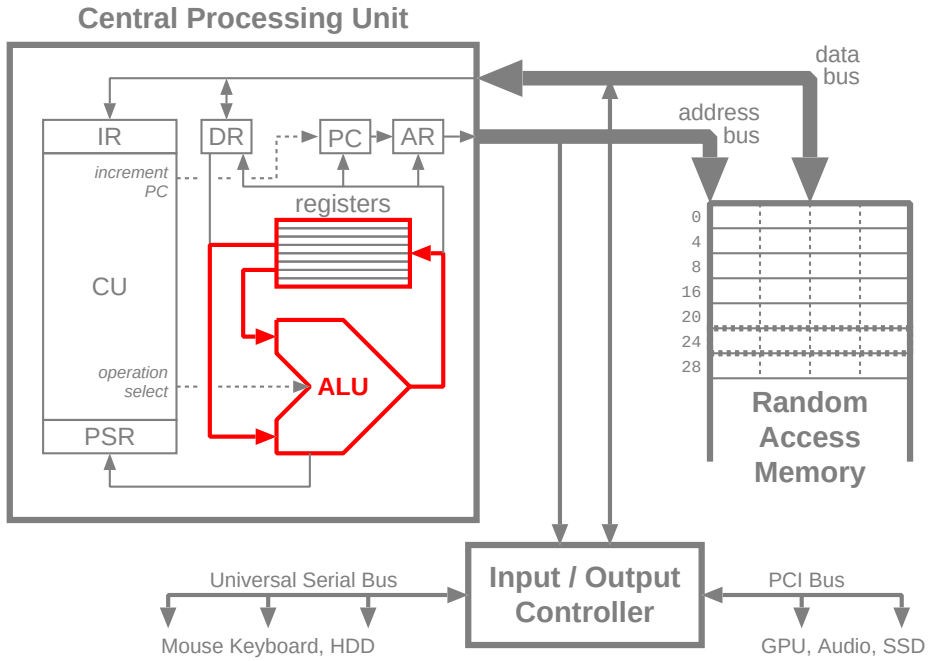
range of representable numeric values

unsigned arithmetic

- terminology
- basic mathematical operations
  - $+$ $-$ $\times$ $\div$
  - in decimal
  - in binary

integer overflow

- conditions and detection



this week

binary representations of signed numbers

- one’s complement, two’s complement
- sign-magnitude, biased

signed binary arithmetic

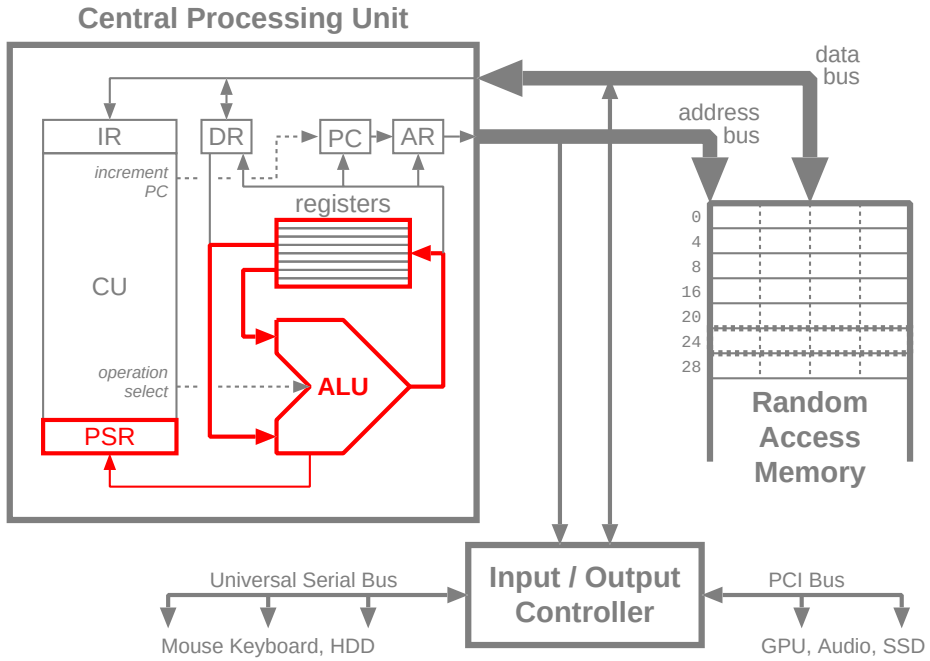
- negation
- addition, subtraction
- signed overflow detection
- multiplication, division

width conversion

- sign extension

bonus material, if there is time and interest...

- floating-point numbers



one's complement representation

the idea: to negate a n-bit number, *invert* (or '*flip*') each bit

this is equivalent to subtracting it from $2^n - 1$

$$-x = 2^n - 1 - x$$

- e.g., with 4-bit numbers, $-5 = 2^4 - 1 - 5 = 15 - 5$

$$\begin{array}{rcccc} & 1 & 1 & 1 & 1 \\ - & 0 & 1 & 0 & 1 \\ \hline & 1 & 0 & 1 & 0 \end{array} \quad \begin{array}{l} = 2^4 - 1 = 15_{10} \\ = 5_{10} \end{array}$$

note: borrows will never occur

using n bits, the representable range is

$$-(2^{n-1} - 1) \ \ldots \ 2^{n-1} - 1$$

KUAS
KYOTO UNIVERSITY of ADVANCED SCIENCE

one's complement representation

advantages

- negation is efficient, and can always be performed
- the most significant bit represents the sign

disadvantages

- there are two representations of 0, one of them negative ($-0 = 2^n - 1$)
- binary arithmetic operations do not work without some adjustment

one's complement is rarely used as a final representation, but. . .

using it as a temporary representation *significantly* simplifies the ALU

- which is why we are studying it today
  (we will use it later, when designing hardware to perform subtraction)

two's complement representation

the idea: to negate a n-bit number, subtract it from 0

$$-x = 0 - x \qquad (\text{duh!})$$

but note: because $2^n = 0$ (for n-bit numbers) we have

$$-x = 0 - x = 2^n - x$$

(which can be peformed using *unsigned* numbers, which are easy to deal with)

x and $-x = (2^n - x)$ are *additive inverses* of each other

$$x + (-x) = x + (2^n - x) = 2^n = 0 \quad \text{(for } n\text{-bit numbers)}$$

- unsigned addition/subtraction work on 2's complement numbers *without adjustment*
- this is easy to verify; e.g., with 4-bit numbers, $-5 = 2^4 - 5 = 16 - 5$

$$\begin{array}{rcccc|l} 1 & 0 & 0 & 0 & 0 & = 2^4 = 16_{10} = 0 \\ - & 0 & 1 & 0 & 1 & = \ \ 5_{10} \\ \hline & 1 & 0 & 1 & 1 & = -5_{10} \end{array} \qquad \text{verifying} \ldots \qquad \begin{array}{rcccc|l} & 0 & 1 & 0 & 1 & = \ \ 5_{10} \\ + & 1 & 0 & 1 & 1 & = -5_{10} \\ \hline & 0 & 0 & 0 & 0 & yay! \end{array}$$

two's complement representation

using n bits, the representable range is

$$-(2^{n-1}) \ \ldots \ 2^{n-1} - 1$$

advantages

- the most significant bit represents the sign
- there is only one representation for zero, and it is non-negative
- signed addition/subtraction are the same as for unsigned integers

disadvantages

- negation is more complex than with one's complement
- one representable value, $-(2^{n-1})$, cannot be negated

two's complement is by far the most widely-used signed binary integer representation

two's complement radix conversion

positive numbers can be converted just like unsigned numbers

negative numbers can be converted in either of two ways

- negate the number (make it positive), convert it, then prepend a '$-$'; or
- consider the sign bit to have value $-(2^{n-1})$ and convert as usual

for example, if $n = 4$ then $-5 = 1011_2$

sign bit has negative weight $\downarrow$

$$\begin{array}{rcccc} & -2^3 & 2^2 & 2^1 & 2^0 \\ & -8 & 4 & 2 & 1 \\ \times & 1 & 0 & 1 & 1 \\ \hline & -8+2+1=-5 \end{array}$$

to see why this works

- calculate $0 - 5$ using three bits
- consider what amount was borrowed from the 4th position by the 3rd position
- that value must be the positional weight of the sign bit

negation

let's write the two's complement of x as $-x$, and the one's complement as $\sim x$

two's complement negation can be performed easily by noticing that

$$\begin{aligned} -x &= 2^n - x \\ &= (2^n - 1 - x) + 1 \\ &= \sim x + 1 \end{aligned}$$

in other words, to find the two's complement of x

- invert each bit in x (which gives us the one's complement of x) and then
- add one

for example, to negate 5:

$$\begin{array}{rcccc} & 0 & 1 & 0 & 1 \\ & 1 & 0 & 1 & 0 \\ + & 0 & 0 & 0 & 1 \\ \hline & 1 & 0 & 1 & 1 \\ \hline \end{array} \quad \begin{array}{l} \\ = \sim 5 \\ \\ = -5 \end{array} \quad \begin{array}{l} \\ \text{(invert bits to get one's complement)} \\ \text{(add one)} \\ \text{(two's complement)} \end{array}$$

addition, subtraction, and overflow

two's complement addition and subtraction are the same as for unsigned

- because $-x$ is the additive inverse of x, modulo 2^n

unsigned overflow is not the same as signed overflow

- the carry out from the sign bits does not indicate signed overflow

the conditions for signed overflow during addition and subtraction are

overflow conditions

inputs		result	
x	y	$x+y$	$x-y$
$+ve$	$+ve$	$-ve$	
$+ve$	$-ve$		$-ve$
$-ve$	$+ve$		$+ve$
$-ve$	$-ve$	$+ve$	

(overflow is *impossible* in the four cases left blank)

one way to detect overflow is therefore to check the sign bits of inputs and result

multiplication, division, and sign extension

signed multiplication is the same as unsigned multiplication, however

- the product may require $2n$ bits, so
- both operands are *sign-extended* to $2n$ bits before being multiplied

to sign extend, the sign bit is copied to the left as many times as required

$$\begin{array}{rl} 0\,1\,0\,1 & 5 \text{ (as a 4-bit number)} \\ 0\,0\,0\,0\,0\,1\,0\,1 & 5 \text{ (as an 8-bit number)} \\ copy \leftarrow \uparrow & \\ sign & \end{array}$$

$$\begin{array}{rl} 1\,0\,1\,1 & -5 \text{ (4-bit number)} \\ 1\,1\,1\,1\,1\,0\,1\,1 & -5 \text{ (8-bit number)} \\ copy \leftarrow \uparrow & \\ sign & \end{array}$$

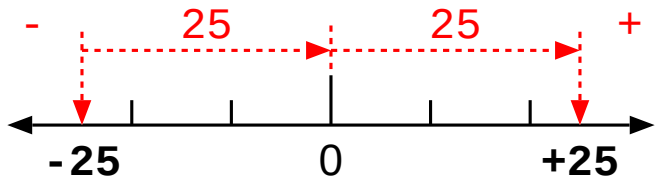
signed division can be performed in two ways

- convert operands to positive, then adjust the signs of quotient/remainder
  (the remainder conventionally has the same sign as the dividend)
- using a slightly more complex algorithm (described in the lecture notes)

other signed number represention

the *magnitude* of a number tells us how far from 0 it is on the number line

the *sign* of a number tells us on which side of 0 it is located



the magnitude of a number n is written $|n|$

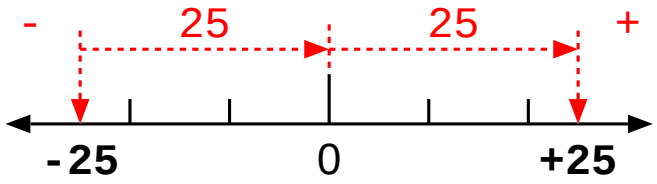
sign-magnitude representation

the idea: use a symbol/digit to represent the sign, just like in decimal

in decimal

- leftmost 'position' indicates the sign ('$+$' to the right, '$-$' to the left, of 0)
- the other positions indicate the magnitude (the distance from 0)

$$
\begin{array}{rl}
 & \textit{sign} \\
 & \downarrow \\
\textit{positive} & +\,2\,5 \\
\textit{negative} & -\,2\,5 \\
 & \uparrow\uparrow \\
 & \textit{magnitude}
\end{array}
$$



in binary

- most significant (leftmost) bit represents the sign ('0' positive, '1' negative)
- the other bits represent the magnitude

$$
\begin{array}{rll}
 & \textit{sign} & \\
 & \downarrow & \\
\textit{positive} & 0\,1\,0\,1 & = +5_{10} \\
\textit{negative} & 1\,1\,0\,1 & = -5_{10}
\end{array}
$$

sign-magnitude representation

using n bits (including sign), the range of representable values is symmetrical

$$-(2^{n-1}-1) \ \ldots \ 0 \ \ldots \ 2^{n-1}-1$$

advantages

- easy to convert to/from decimal
- negation is trivial, and can always be performed
- simplifies some operations *iff* arithmetic is always signed, e.g:
  - multiplication/division on the magnitudes
  - logic on the signs to compute the sign of the result

disadvantages

- two representations for zero ($+0$ and -0)
  - reduces representable range by 1
- algorithms (and hardware) are more complex
  - signed and unsigned arithmetic are different

almost never used (for integers)

biased representation

the idea: add a fixed offset k to every number (also called 'excess-k representation')

- store signed numbers as $n + k$ (no sign bit required)
- subtract k from the stored value to obtain n, the original signed value

typically, $k = 2^{n-1}$ is chosen so that 0 falls in the middle of the range

using n bits, the range of representable values is

$$-k \ \ldots \ 2^n - 1 - k$$

advantages

- there is only one representation for 0
- the representable range is contiguous
  - unsigned comparisons give correct results for signed numbers
  - the magnitude and sign of differences are very easy to compute

disadvantages

- arithmetic operators are difficult to implement
- $0 + 0 \neq 0$ (except when $k = 0$, which is *useless*)
- negation requires subtraction ($-x = 2k - x$), and cannot always be performed

floating-point numbers

floating-point numbers are stored as $x = s \times 2^e$ where

- s is a signed *significand*, in the range $1 \leq |s| < 2$, and
- e is a signed binary *exponent*, in the range $-126 \leq e \leq 127$

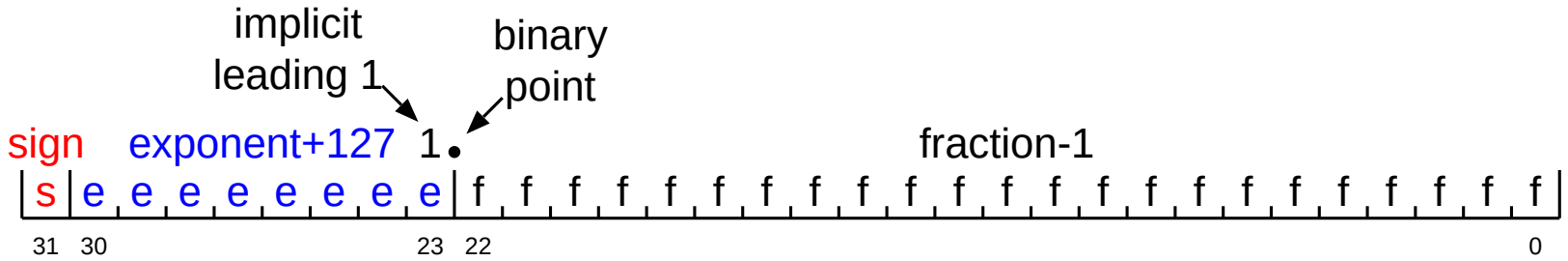
e is stored using biased representation

- the difference between exponents is important during addition/subtraction
  - it tells us how far to shift the binary points to align them

s is stored using sign-magnitude representation, as a sign s and fraction f

- aligning binary points during addition/subtraction is easier without a sign
- multiplication and division are simplified
  - multiply/divide significands directly
  - add/subtract the exponents, adjust by 1 if necessary

floating-point numbers



the significands are *normalised* in the range $1.0000 \ldots 1.1111$

- they always begin with $1.$, so the leading 1 is made implicit
- one extra bit of precision is gained 'for free'
- a special representation for zero ($e = 0, f = 0$) has to be used

the value of the stored number is therefore

$$(-1)^s \times 1.f \times 2^{e-127}$$

advantages

- floating point numbers can be compared as if they were sign-magnitude integers

disadvantages

- $e = 0$ and $e = 255$ are special (infinity, 'not a number', and *denormalised* numbers)
- zero can be positive or negative

summary

signed integers can be stored many ways

- each method has some advantage that makes it useful somewhere

biased and sign-magnitude representations mostly used within floating-point numbers

- good for representing the relative importance of two exponents
- allow integer comparisons to work for floating point numbers
- simplify arithmetic where binary points have to be shifted into alignment

one's complement is easy (flip bits) and useful (2's complement negation), but

- two representations of zero
- not often used as a final representation

summary

two's complement has many desirable properties

- most widely-used signed integer representation

negating a number gives you its additive inverse

- negation is easy: take one's complement then add one
- signed arithmetic 'just works', even using unsigned algorithms, but . . .

overflow detection is different from signed integers

- can be done by comparing operand and result signs

width conversion is done by sign extension

- copy the sign bit left to fill the number of bits required

radix conversion is just as easy as for unsigned numbers

- treat the sign bit as if it had positional value $-(2^{n-1})$

next week

information theory

- information content
- Hamming distance

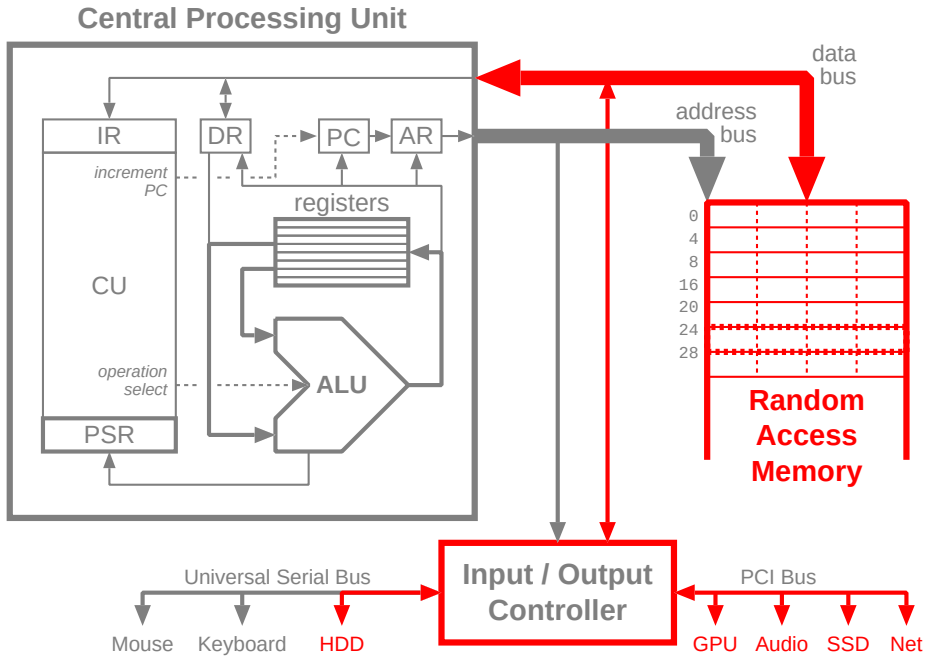
source coding vs. channel coding

error detection

- motivations
- parity
- cyclic redundancy checks

error correction

- motivations
- block parity
- Hamming codes



homework

practice two's complement radix conversions

practice two's complement negations

practice two's complement arithmetic

- including signed and unsigned overflow detection

reinforce your knowledge

- do you know about arrays (or lists) in Python?
- **write** some Python functions that perform positional arithmetic
  - digit by digit, with explicit carry and borrow
  - use any radix you like, but definitely try binary
    (all the computers in EPL1/CC207 have Python installed)

ask about anything you do not understand

- it will be too late for you to try to catch up later!
- I am always happy to explain things differently and practice examples with you

glossary

additive inverse — of a number x is that number y such that $x + y = 0$. In other words, $y = -x$. In modular arithmetic, two positive numbers can be additive inverses of each other. Taken modulo 2^n, x and $2^n - x$ are additive inverses, which is the basis for two's complement arithmetic.

denormalised — a number that is no longer normalised. In floating-point representation, a significand that does not have a single implicit leading 1 before the binary point is a denormalised number.

exponent — the 'power' to which another number is raised. In x^e, e is the exponent, describing how many times x should be multiplied by itself. In floating-point representation, the exponent indicates the power of 2 by which the significand must be multiplied to obtain the true magnitude of the represented number.

flipping — a more casual term for 'inverting'.

inverting — changing the state of something, often between either of two states. In binary arithmetic and logic, inverting a bit means changing $0 \to 1$ or $1 \to 0$.

iff — a concise way of writing 'if and only if'.

normalised — a number that has a standard form or which falls within a standard range. In floating-point representation, a normalised significand s is always in the range $1 \leq s < 2$; in other words, when written as a binary fraction, there is a single 1 to the left of the binary point.

sign-extend — widening a binary number to a larger number of bits by replicating the sign bit to the left until the desired width is obtained.

glossary

significand — the part of a floating point number which provides the digits without regard to the position of the binary point. Multiplying the significand by 2 to the power of the exponent yields the true magnitude of the represented number.

useless — pointless, purposeless, impractical, ineffectual, unproductive, inutile, worthless, inadequate, no good, or a dead loss. For example, in an excess-k biased, signed, integer representation, the fact that $0 + 0 = 0$ provided $k = 0$ is useless because it is impossible to represent any negative number in that representation.