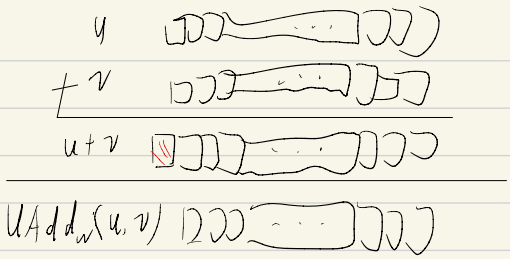
Operands: w bits

True Sum: w+1 bits

Discard Carry: w bits

进位



- standard Addition Function

  Ignores carry output

- Implements Modular Arithmetic

  $s = UAdd_w(u, v) = u + v \bmod 2^w$

Integer Addition

① 4-bit integers u, v

② compute true sum $Add_4(u, v)$

③ Values increase linearly with u and v

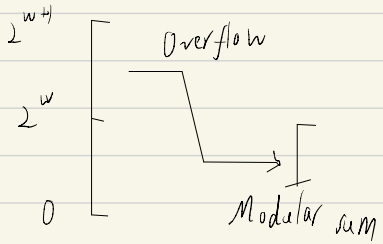
④ Forms planar surface.

Wraps Around:

If true sum $\geq 2^w$.

At most once.

True Sum:



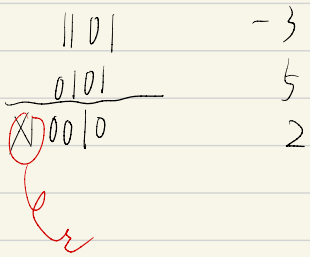
TAdd and UAdd have Identical Bit-Level Behavior.

- Signed v.s. unsigned addition in C:

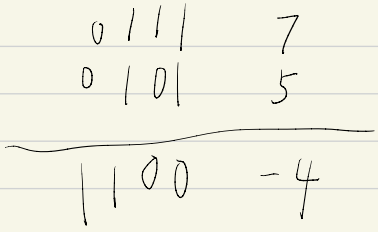
```
int s, t, u, v;
s = int( unsigned (u) + unsigned(v) );
t = u+v
```

will give s == t

e.g.1.



e.g.2.



TAdd Overflow

Functionality.

(i) True sum requires w+1 bits

(ii) Drop off MSB (most significant bit)

(iii) Treat remaining bits as 2's comp. integer.

Visualizing 2's complement Addition.

(I) Values

(i) 4-bit two's comp.

(ii) Range from -8 to +7.

(II) Wraps Around.

(i) If sum $\geq 2^{w-1}$
- Becomes negative
- At most once

(ii) If sum $< -2^{w-1}$
- Becomes positive
- At most once

Multiplication.

(I) Goal: Computing Product of w-bit numbers x, y.
Either signed or unsigned.

(II) But, exact results can be bigger than w bits:

(i) Unsigned: up to 2w bits
- Result range: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$

(ii) Two's complement min (negative): Up to 2w-1 bits
- Result range: $x * y \ge (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$

(iii) Two's complement max (positive): Up to 2w bits, but only for $(TMin_w)^2$.
- Result range: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$.

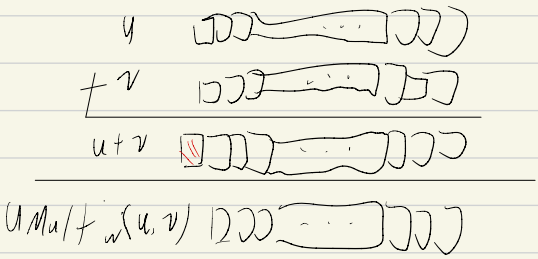
So, maintaining exact results...
(i) would need to keep expanding word size with each product computed

(iii) is done in software, if needed.
- e.g. by "arbitrary precision" arithmetic packages.

Operands: w bits

True product 2*w bits

Discard w bits: w bits



(I) Standard Multiplication Function:

(ii) Ignores high order w bits.

(II) Implements Modular Arithmetic

$UMult_w(u,v) = u \cdot v \bmod 2^w$.

e.g.

3 * 5

= 15 OK

discard

5 * 5

= 0(1010)

$2^4\ 2^3\ 2^1$

= 9 (rather than 25)

Signed Multiplication in C.

Operands: w bits.

- Standard Multiplication Function:
  - (i) Ignores high order w bits
  - (ii) Some of which are different for signed vs. unsigned multiplication
  - (iii) Lower bits are the same.

4bits

5 times 5

```
  101
  101
 ----
  101
 101
 ----
 1001
```

discard the leading 1

$= -(2^3) + 2^0 =$ minus 7.

Power-of-2 multiply with shift

1. Operations:
   (i) u << k gives $u * 2^k$
   Both signed and unsigned.
   Operands: w bits.

historically, shift operation is much faster than multiplication operation.

Unsigned power-of-2 divide with shift.

Quotient of Unsigned by Power of 2.

(i) u >> k gives $\lfloor u/2^k \rfloor$

(ii) Uses logical shift. (filled with 0) (NOT arith. shift filled with 1)

unsigned case:

0110 (6)$_{10}$

0011 6>>1 (3)$_{10}$

0001 6>>2 (1)$_{10}$

signed case:

1010 (-6)

1101 (-6>>1) (-3) (arith. shift)

1110 (-6>>2) (-2)

division still much slower than right-shift.

So, compiler does it with shifting and tweaking (某些调整) things around it....

$x \to -x$ step 1: complement step 2: increment

Flip $\begin{array}{r} 1010 \\ +\ 0101 \\ +\ 0001 \\ \hline 0110 \end{array}$ -6 $\quad$ 6

$\begin{array}{r} 0110 \\ \sim\ 1001 \\ +\ 0001 \\ \hline 1010 \end{array}$ $\quad$ 6

>>> Java logical shift

>> C arithmetic shift.

check { error_soln.c

error_soln1.c.

Byte-Oriented Memory Organization.

00...0 FF...F

(I) Programs refer to data by address.
  (i) Conceptually, envision it as a very large array of bytes
  (ii) In reality, it's not, but can think of it that way.

(II) An address is like an index into that array.
  and a pointer variable stores an address

(III) Note: system provides private address spaces to each "program"
  (i) Think of a process as a program being executed
  (ii) So, a program can clobber its own data but not that of others.

clobbering: To overwrite, usu. unintentionally; "I walked off the end of the array and clobbered the stack."

Machine Words.

(i) Any given number has a "Word Size": (these days, typically 32 bits or 64 bits)
   (i) Nominal size of integer-valued data and of address.

It refers to the number of bits processed by a computer's CPU in one go.

(ii) Until recently, most machines use 32 bits (4 bytes) as word size.
   - Limits addresses to 4GB (2^{32} bytes)

(iii) Increasingly, machines have 64-bit word size.
   - Potentially, could have 18PB (petabytes) of addressable memory
   - That's 18.4×10^{15}

(iv) Machines still support multiple data formats:
   - Fractions or multiples of word size.
   - Always integral number of bytes.

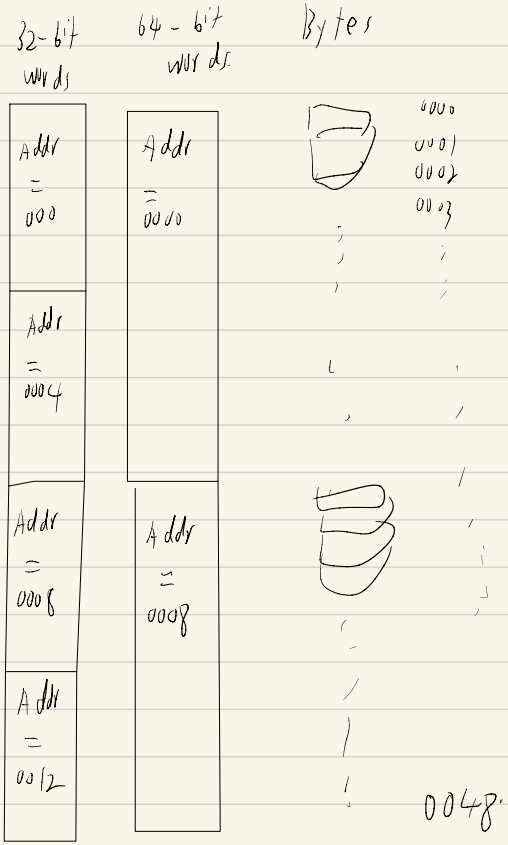
It's compiler and computer together who decide the word size in a certain program.

Word-Oriented Memory Organization.

(2) Addresses Specify Byte Locations.

(i) Address of first byte in word.

(ii) Addresses of successive words differ by 4 (32-bit) or 8 (64-bit).



Byte Ordering:

(I) So, how are the bytes within a multi-byte word ordered in memory?

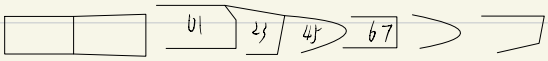
(I) Conventions:

(i) Big Endian: Sun, PPC Mac, Internet. Least signficant byte has highest addrs
Least Endian: x86, ARM processors running Android,
Least significant byte has lowest address

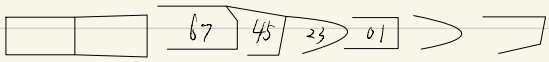
Byte ordering Example:

(I) Example
(i) Variable x has 4-byte value of 0x01234567
(iii) Address given by &x is 0x100

Big Endian: 0x100 0x101 0x102 0x103

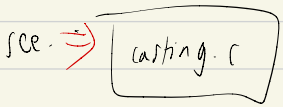


Little Endian: 0x100 0x101 0x102 0x103



Examing Data Representations:

Code to print Byte Representation of Data.



You can't directly copy a pointer from another.

Representing Strings.

(I) strings in C.

(i) Represented by array of characters

(ii) Each character encoded in ASCII format.

(a) Std. 7-bit encoding of character set

(b) Character "0" has code 0x30.

— Digit i has code 0x30 + i

(iii) String should be null-terminated.

(a) Final character = 0

(II) Compatibility

(i) Byte ordering not an issue.

Tmin

$x < 0 \Rightarrow ((x*2) < 0)$ FALSE

$ux > 0$ TRUE

$x \& 7 == 7 \Rightarrow (x << 30) < 0$. (TRUE)

(32 bytes)

$ux > -1$ (-1) Always false

↘ It will be cast into U_{max}

$x > y \Rightarrow -x < -y$ (F)

If $T_{Min} = y$ Then...

$x * x >= 0$ FALSE.

$x > 0 \&\& y > 0 \Rightarrow x + y > 0$ FALSE

$x >= 0 \Rightarrow -x <= 0$ TRUE.

$x <= 0 \Rightarrow -x >= 0$ FALSE

$|Tmin| - |Tmax| == 1$

$(x|-x) >> 31 == -1$

- $x == 0$ FALSE
- otherwise TRUE.