DNA
Data and Program Representation

Alexandre David
1.2.05
adavid@cs.aau.dk

Introduction

- Very important to understand how data is represented.
  - operations
  - limits
  - precision
- Digital logic built on 2-valued logic system
  - high/low 5V/0V true/false
  - we abstract from that from now on
    $\rightarrow$ bits 0/1

Basics

<u>Natural/Real Numbers</u>

- Base 10
- Infinite
- Exact

<u>Computer Numbers</u>

- Base 2
- Finite
- Rounding - overflow

<u>In this lecture</u>

- How to represent numbers & characters – range, encoding.
- A little arithmetic.
- How to use these numbers.

Questions

- How to code negative numbers?
- How to code real numbers?
- Which kind of precision do we get?
  - Small numbers vs. big numbers.
- What about characters?

Example

- Overflow:
```
main() {
  printf("%d\n",200*300*400*500);
}
```
outputs -88490188.
- Fix – sort of:
```
main() {
  printf("%lld\n",200LL*300LL*400LL*500LL);
}
```
outputs 120000000000.

What if I forget LL?

Example

- Loss of precision:
  (3.14+1e20)-1e20==0.0
  3.14+(1e20-1e20)==3.14
- Test x == 0.0 not very useful when solving equations.
- In this lecture you will know why.

Data Storage

- Basic unit is the byte (= 8 bits).

C-declaration	Typical 32-bit	Typical 64-bit
char	1	1
short int	2	2
int	4	4
long int	4	8
char *	4	8
float	4	4
double	8	8

“Features”

- Limits on addressable memory.
- Size linked to architecture – 32/64.
- Aligned memory allocation (32/64 bits).
- Careful on addressing:

```
main() {
  char a[]="Hello world!";
  int *p=&a[1];
  printf("%d\n",*p);
}
```

Bus error on some CPUs

Integer Encoding

- Unsigned integers: $$UB = \sum_{i=0}^{w-1} x_i 2^i$$
- Signed integers:
  Called 2 complement. $$SB = -x_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

w: size of a word (in bits)
x: bits (0 or 1)

- Highest bit codes the sign.

Range & Examples

- Examples: *1010* = 10, *0110* = 6, *0101* = 5

$2^3=8$	$2^2=4$	$2^1=2$	$2^0=1$

- Range:
  - unsigned 2^k numbers from 0 to 2^k-1
  - signed 2^k numbers from -2^{k-1} to $2^{k-1}-1$
    - one more negative number than positive ones
- How to convert between types?
  - int – char – long int…
  - sign extension

Basic Arithmetic

- Logical operations (bitwise): &,|,^,~,<<,>>.
  Example: a ^= b; b ^= a; a ^= b;
- Arithmetic operations: + - * /.
- Careful with shifts on signed integers!
  - arithmetic & logical shifts
- Do not mess up with boolean operators (&&, ||).

Properties

- Most operators are the same on signed/unsigned integers – from a binary point of view – beauty of the encoding.
  - One hardware implementation for
    + - / * …
    valid for signed and unsigned integers.
  - Example on 4 bits:
    1 + 1001 = 1010
    unsigned: 1+9 = 10
    signed: 1+(-7) = -6

Properties

- Operations based on the algebra <Z_n,$+_n$,$*_n$,$-_n$,0,1> (commutativity, associativity, distributivity,…)
  Operations modulo n and -a=0 or -a=n-a.
- Similar to boolean algebra <{0,1},|,&,~,0,1> with the addition of DeMorgan laws
  ~(a&b)=~a|~b, ~(a|b)=~a&~b

Practice: Shifts & Masks

- Read bit n: Use mask (1 << n).
- Set bit n on int bits[]:
```
ipos   = n / 32;
imask = 1 << (n % 32);
bits[ipos] |= imask;
```
- Division/multiplications by powers of 2 seen as shifts.
- 2-complement: -a = ~(a-1) = ~a+1

Arithmetic

- Machine code of + - * / same for int/uint.
- Integer convertion == type casting.
  - Padding for the sign (int).
  - Conversion is modulo the size of the new int.
  - Beware of implicit conversions in C!
- Optimizations for some operations:

```
2*a    == a+a == a<<1       a/2    == a>>1
a*2^i == x << i             a/2^i == a>>i
a%2^i == a & ((1<<i)-1)   2^i    == 1<<i
```

Notes

- Beware of precedence of operators:
  - if (x & mask == value) WRONG
  - if ((x & mask) == value) RIGHT
- Test odd numbers: if (x & 1)
- Careful:

```
unsigned int i;
for(i = 0; i < n-1; ++i) …
```

?

Overflow – "carry"

```
 1011
+1101
 111
-----
11000
```

Subtraction?

```
    1011   multiplicand
   *1101   multiplier
--------
    1011
   0000    partial
  1011     products
 1011
--------
10001111   product
```

Hexadecimal Notation

- Learn the first powers of 2.
- Hexadecimal more useful:
  - One digit codes 4 bits. 0…F=0…15=16 numbers.
  - Notation: 0x…
  - Why?
- Examples:
  - 0xa57e=1010 0101 0111 1110
    10=8+2,5=4+1,7=4+2+1,14=8+4+2
  - 0xf = 1111, 0x7=0111, 0x3=0011

Endianness: Beware!

- “0xa57e” is a notation for humans. Corresponds to “1010 0101 0111 1110” in base 2.
  - Little endian: stored as 0111111010100101.
  - Big endian: stored as 1010010101111110.
- Does not matter in C/Java/C#, except for
  - bitmap manipulation
  - device drivers
  - network transfers

Testing for Endianness

- Write a value on 32 bits.
  - Read 8/16 bits and check what was written.
  - Exercise for Sun/Intel.

```
main() {
    int a = 0xf0000000;
    char *c = &a;
    printf("%x\n", *c);
}
```

  - Intel? → 0
  - Sun? → 0xfffffff0
  - What if a = 0x70000000?

Representation of Reals

- How to code a real number with bits?
  - Finite precision $\rightarrow$ approximation.
  - Represent very small and very large numbers $\rightarrow$ density of encoding varies.
- Scientific notation used, e.g. (base 10), 3.141e2 – but in base 2.
- Starter: fractional numbers – bad for large or small numbers.
  - Decimal (d):
  - Binary(b):

$$d = \sum_{i=-n}^{m} 10^i d_i \quad b = \sum_{i=-n}^{m} 2^i b_i$$

IEEE Floats

- IEEE 754 floating point standard

S	E	M

  - $V=(-1)^S M * 2^E$
  - Number of bits (float/double):
    s[1], m[23/52], e[8/11].
  - Normalized and de-normalized values.
  - Bit fields: s, m, e to code respectively S, M, E.
- Normalized values ($e \neq 0$, $e \neq 111\ldots$)
  - $E=e-bias$ (-126…127/-1022…1023).
  - $M=1+m \qquad (1 \leq M < 2)$
  - Trick for more precision: implied leading 1.

IEEE Floats

- De-normalized values (e= 0 or 11…)
  - e=0: E=1-bias, bias=2^{k-1}-1
    - Coding compensates for M not having an implied leading 1.
    - M=m
    - For numbers very close to 0.
  - e=11…:
    - m=0, (signed infinite)
    - m!=0, NaN.

No more of this.
We abstract from
this complexity.

Ranges of Floats

- Single precision (float)
  $2^{-126}...2^{127} \sim 10^{-38}...10^{38}$.
- Double precision (double)
  $2^{-1022}...2^{1023} \sim 10^{-308}...10^{308}$.

“Features” of IEEE floats

- +0.0 == 0 (binary representation = 00…).
- If interpreted as unsigned int, floats can be sorted (+x ascending, -x descending).
- All int values representable by doubles.
- Not all int values representable by floats.
  - round to even (avoid stat. bias)
  - round towards 0
  - round up
  - round down
  - can’t choose in C…

Properties (floats)

- Operations NOT associative.
- Not always inverse (infinity).
- Loss of precision.
- Ex: x=a+b+c; y=b+c+d;
  Optimize or not?
- Monotonicity $a \geq b \Rightarrow a+x \geq b+x$
- Casts:
  - int2float rounded, double2float rounded/overflow
  - int2double, float2double OK
  - float2int, double2int truncated/rounded/overflow.

Important for compilers and programmers.

IA32: The Good And The Bad

- Good: Uses internally 80 bits extended registers for more precision.
- Bad:
  - Stack based.
  - Side effects like changing values when loading or saving numbers in memory whereas register transfers are exact.
- Extensions: MMX, SSE, (Altivec). SIMD instructions = operations working in parallel on multiple data.

Numerical Precision

- Evaluation of precision
  - absolute $x \pm \alpha$
  - relative $x*(1 \pm \alpha)$
- Be careful with division by very small values: Can amplify numerical errors.
  - Numerical justification for Gauss' method to solve linear equations.

Character Sets

- By convention (standard), we assign codes to characters.
  - Careful with programming languages
    C char = 1 byte, Java char = 2 bytes
  - ASCII (American Standard Code for Information Interchange) common (7 bits), extended ASCII (8 bits).
  - Unicode
  - Other IEEE8859-x codings.
  - Other Japanese codings…

More Complex Structures

- How to represent, e.g.,
  struct { int a, b; char c}?

- Use continuous bits for the successive fields.
  - Compilers will *align* data!
  - Try:

```
typedef struct { int a; char c; } foo_t;
int main()
{
  printf("%d\n", sizeof(foo_t));
  return 0;
}
```

What About Programs?

- Instructions coded into bytes.
  - “opcode”
  - Processors interpret them as their particular instruction sets (standard).