Bits & Bytes

Number systems

Decimal (base 10): *India*

why 10?

digits: 0 .. 9

$9031 = 9 \cdot 10^3 + 0 \cdot 10^2 + 3 \cdot 10^1 + 1 \cdot 10^0$

Binary (base 2): *India*

why 2?

digits: 0, 1

$1011_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11$

Byte = 8 Bits: 10110011_2 value $\in \{0 .. 255\}$

conversion decimal → binary: division with rest with 2

example:

$11 = 5 \cdot 2 + 1$

$5 = 2 \cdot 2 + 1$

$2 = 1 \cdot 2 + 0$

$1 = 0 \cdot 2 + 1$

(remainders, read upward: result)

Hexadecimal (base 16): *IBM 1963 in this form*

why 16?

digits: 0 .. 9 A .. F $B_{16} = 11_{10} = 1011_2$

1 Byte = 2 hex. digits 00 ... FF

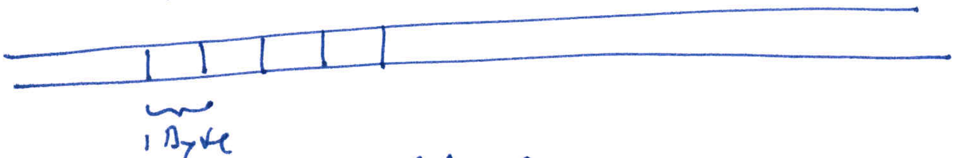
$0x2AB1 = 2AB1_{16} = 2 \cdot 16^3 + 10 \cdot 16^2 + 11 \cdot 16^1 + 1$

in C

conversion decimal → hex.: division with rest using 16

Words, Adresses, Data Representations

(Virtual) memory is divided into Bytes:



every Byte has an address

size of adress = size of pointer (e.g. char *) = word size

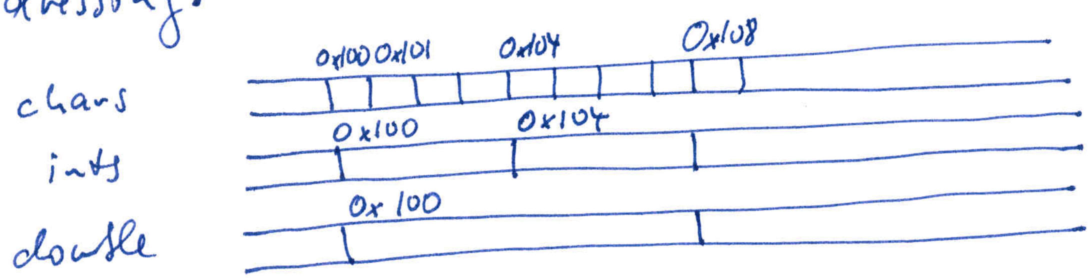
32-Bit architecture: 4 Bytes → 2^{32} B = 4 GB virtual address space

64-Bit architecture: 8 Bytes → 2^{64} B

Data types and sizes:

	ia 32	x86-64
char	1	1
short	2	2
int	4	4
long	4	8
long long	8	8
float	4	4
double	8	8
char * (or any other)	4	8

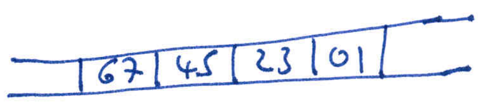
Addressing:



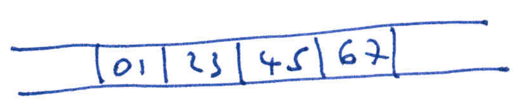
Byte ordering:

int x = 0x01234567

little endian (ia32, x86-64):



big endian (Sun):



&x: depends on machine / compiler

Boolean Algebra

George Boole, ~1850, foundation of logic: $0 = \text{false}$, $1 = \text{true}$
England

		commutative	associative
and	&	✓	✓
or	|	✓	✓
xor	^	✓	✓
not	~	-/.	-/.

and &:

&	0	1
0	0	0
1	0	1

or |:

|	0	1
0	0	1
1	1	1

xor ^:

^	0	1
0	0	1
1	1	0

not ~:

~	
0	1
1	0

relationship with integers:

		distributive		
rings	$(\mathbb{Z}, +, \cdot)$	✓	$a+0=a$	$a+(-a)=0$
	$(\{0,1\}, \mid, \&)$	✓	$a \mid 0 = a$	$a \mid ? = 0$
	$(\{0,1\}, \wedge, \&)$	✓	$a \wedge 0 = a$	$a \wedge (\sim a) = 0$

$= (\{0,1\}, + \bmod 2, \cdot \bmod 2)$

Bit operations in C

&, |, ^, ~
- available for integral data types char, int, ..
- bitwise operation

$\sim 0x97 = \sim 10010111_2 = 01101000_2 = 0x68$

$0x97 \;\&\; 0x3C = 10010111_2 \;\&\; 00111100 = 00010100$
masks out bits

Application 1: Set representation, $S = \{0, .., 7\}$

subset ⟷ 8 bits

01100001 ⟷ $\{1,2,7\}$

& ⟷ intersection
| ⟷ union
^ ⟷ symm. difference
~ ⟷ complement

Application 2:

p = malloc(...)

change p to the next address divisible by 16

(array is 16-Byte aligned)

a.) set last 4 bits to zero

$p_1 = p$ & 0xFFFFFFF0

b.) if $p_1 = p$ ✓

else $p = p_1 + $ 0x10

c.) make it independent of word size

0xFFFFFFF0 ⟷ ~0x0F

Logical operations in C (different from bit operations)

and, or, not ⟷ &&, ||, !

0 = false

nonzero = true

always returns 0 or 1

early termination

examples:

!0x4C = 0x00

!0x00 = 0x01

!!0x4C = 0x01

0x69 && 0x55 = 0x01

a && $\frac{1}{a}$ never division by 0

p && *p never deref. of zero pointer

Shift operations in C

left shift by k: x << k

$[x_{n-1}, x_{n-2}, \ldots, x_0]_2 \longrightarrow [x_{n-k-1}, \ldots, x_0, \underbrace{0, \ldots, 0}_{k \text{ bits}}]_2$

right shift by k: x >> k

logical $[x_{n-1}, x_{n-2}, \ldots, x_0]_2 \longrightarrow [\underbrace{0, \ldots, 0}_{k \text{ bits}}, x_{n-1}, \ldots, x_k]_2$

(for unsigned data)

arithmetic $\longrightarrow [\underbrace{x_{n-1} \ldots x_{n-1}}_{k}, x_{n-1}, \ldots, x_k]_2$

(usually for signed data)