Lecture 2: Data representation, addresses

Phillipp Koehn, David Hovemeyer

January 29, 2020

601.229 Computer Systems Fundamentals

Welcome!

- Today:
  - Data representation
  - Addresses
  - Bitwise operations

Data representation

Motto

There are only 10 kinds of people.
Those who understand binary
and those who don't.

Data representation

Let's consider ways of representing numbers...

Roman Numerals

- Basic units

I	V	X	L	C	D	M
1	5	10	50	100	500	1000

Roman Numerals

- Basic units

I	V	X	L	C	D	M
1	5	10	50	100	500	1000

- Additive combination of units

  II III VI XVI XXXIII MDCLXVI MMXVI

Roman Numerals

- Basic units

I	V	X	L	C	D	M
1	5	10	50	100	500	1000

- Additive combination of units

II	III	VI	XVI	XXXIII	MDCLXVI	MMXVI
2	3	6	16	33	1666	2016

Roman Numerals

- Basic units

I	V	X	L	C	D	M
1	5	10	50	100	500	1000

- Additive combination of units

II	III	VI	XVI	XXXIII	MDCLXVI	MMXVI
2	3	6	16	33	1666	2016

- Subtractive combination of units

IV IX XL XC CD CM MCMLXXI

Roman Numerals

- Basic units

I	V	X	L	C	D	M
1	5	10	50	100	500	1000

- Additive combination of units

II	III	VI	XVI	XXXIII	MDCLXVI	MMXVI
2	3	6	16	33	1666	2016

- Subtractive combination of units

IV	IX	XL	XC	CD	CM	MCMLXXI
4	9	40	90	400	900	1971

Arabic Numerals

- Developed in India and Arabic world during the European Dark Age
- Decisive step: invention of zero by Brahmagupta in AD 628
- Basic units

$$0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9$$

- Positional system

$$1 \quad 10 \quad 100 \quad 1000 \quad 10000 \quad 100000 \quad 1000000$$

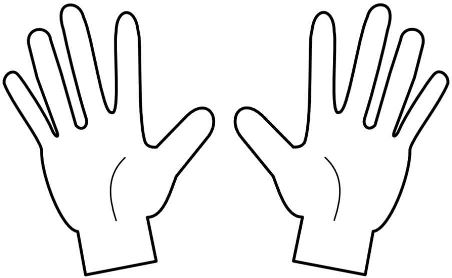
Why Base 10?

dig·it
/ˈdijit/

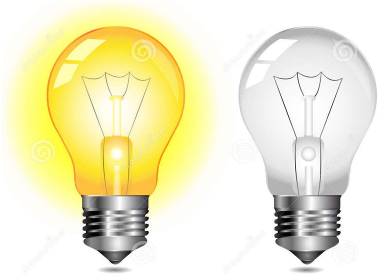
noun

1. any of the numerals from 0 to 9, especially when forming part of a number.
   synonyms: numeral, number, figure, integer
   "the door code has ten digits"

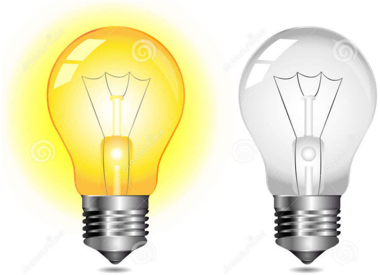
2. a finger (including the thumb) or toe.
   synonyms: finger, thumb, toe; extremity
   "we wanted to warm our frozen digits"



Base 2



Base 2



Computer hardware is based on *digital logic*

- where *digital voltages* (high and low) represent 1 and 0

Base 2

- Decoding binary numbers

Binary number	1	1	0	1	0	1	0	1

Base 2

- Decoding binary numbers

Binary number	1	1	0	1	0	1	0	1
Position	7	6	5	4	3	2	1	0

Base 2

- Decoding binary numbers

Binary number	1	1	0	1	0	1	0	1
Position	7	6	5	4	3	2	1	0
Value	2^7	2^6	0	2^4	0	2^2	0	2^0

Base 2

- Decoding binary numbers

Binary number	1	1	0	1	0	1	0	1	
Position	7	6	5	4	3	2	1	0	
Value	2^7	2^6	0	2^4	0	2^2	0	2^0	
	128	64	0	16	0	4	0	1	$= 213$

Clicker quiz 1

Clicker quiz omitted from public slides

Base 8

- Numbers like 11010101 are very hard to read

$\Rightarrow$ Octal numbers

Binary number	1	1	0	1	0	1	0	1
Octal number		3		2			5	

Base 8

- Numbers like 11010101 are very hard to read

$\Rightarrow$ Octal numbers

Binary number	1	1	0	1	0	1	0	1
Octal number		3		2			5	
Position		2		1			0	

Base 8

- Numbers like 11010101 are very hard to read

⇒ Octal numbers

Binary number	1	1	0	1	0	1	0	1
Octal number	3		2			5		
Position	2		1			0		
Value	3×8^2		2×8^1			5×8^0		

Base 8

- Numbers like 11010101 are very hard to read

$\Rightarrow$ Octal numbers

Binary number	1	1	0	1	0	1	0	1	
Octal number	3			2			5		
Position	2			1			0		
Value	3×8^2			2×8^1			5×8^0		
	192			16			5		$= 213$

- ... but grouping **three** binary digits is a bit odd

Base 16

- Grouping 4 binary digits $\rightarrow$ base $2^4 = 16$
- "Hexadecimal" (hex = Greek for six, decimus = Latin for tenth)

Base 16

- Grouping 4 binary digits $\rightarrow$ base $2^4 = 16$
- "Hexadecimal" (hex $=$ Greek for six, decimus $=$ Latin for tenth)
- Need characters for 10-15:

Base 16

- Grouping 4 binary digits $\rightarrow$ base $2^4 = 16$
- "Hexadecimal" (hex = Greek for six, decimus = Latin for tenth)
- Need characters for 10-15: use letters a-f

Binary number	1	1	0	1	0	1	0	1
Hexadecimal number	d				5			

Base 16

- Grouping 4 binary digits $\rightarrow$ base $2^4 = 16$
- "Hexadecimal" (hex = Greek for six, decimus = Latin for tenth)
- Need characters for 10-15: use letters a-f

Binary number	1	1	0	1	0	1	0	1
Hexadecimal number	d				5			
Position	1				0			

Base 16

- Grouping 4 binary digits $\rightarrow$ base $2^4 = 16$
- "Hexadecimal" (hex = Greek for six, decimus = Latin for tenth)
- Need characters for 10-15: use letters a-f

Binary number	1	1	0	1	0	1	0	1	
Hexadecimal number	d				5				
Position	1				0				
Value	13×16^1				5×16^0				
	208				5				$= 213$

Clicker quiz 2

Clicker quiz omitted from public slides

Examples

Decimal	Binary	Octal	Hexademical
0			
1			
2			
3			
8			
15			
16			
20			
23			
24			
30			
50			
100			
255			
256			

Examples

Decimal	Binary	Octal	Hexademical
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
8	1000	10	8
15	1111	17	f
16	10000	20	10
20	10100	24	14
23	10111	27	17
24	11000	30	18
30	11110	36	1e
50	110010	62	32
100	1100100	144	64
255	11111111	377	ff
256	100000000	400	100

Bytes and Words

- On all modern computers data is accessed in chunks of 8 bits: 1 *byte*
- Larger chunks of data (“words”) are formed from multiple bytes:
  - 2 bytes = 16 bits
  - 4 bytes = 32 bits
  - 8 bytes = 64 bits
- Modern CPUs have instructions for doing operations on word-sized data values

C data types

- The "primitive" C data types typically map onto machine word sizes
  - ... but unfortunately, not in a way that's completely consistent across different machines and compilers
- "Typical" representations of C data types:

	Bytes used on...	
Data type	32-bit systems	64-bit systems
`char`	1	1
`short`	2	2
`int`	4	4
`long`	4	8

(Note inconsistency in last row)

Portable integer types

- The `stdint.h` header file provides portable integer types providing an exact number of bits: `int32_t`, `uint32_t`, `int64_t`, `uint64_t`, etc.
- Note that constant values are still a problem!
  - For example, 0x100000000UL (2^{32}) is likely to be a valid on a 64-bit system but not on a 32-bit system
    - The "UL" suffix means "unsigned long"

Addresses

Memory and addresses

- Conceptually, memory (RAM) is a sequence of byte-sized storage locations
- Each byte storage location has an integer *address*
  - 0 is the lowest address
  - Highest address determined by number of *address bits* processor uses:
    - 32-bit processors $\Rightarrow$ addresses have 32 bits
    - 64-bit processors $\Rightarrow$ addresses have 64 bits

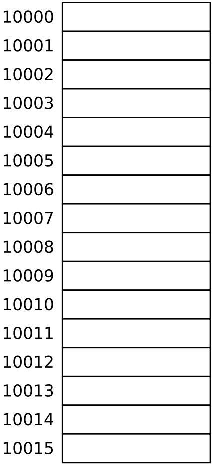
32 bit vs. 64 bit addresses

- 1 GB $= 2^{30}$, 1 TB $= 2^{40}$
- A 32-bit system can directly address 2^{32} bytes (4 GB)
  - Not that much memory by today's standards!
- A 64-bit system can (in theory) directly access 2^{64} = 17,179,869,184 GB = 16,777,216 TB
  - This is a *huge* address space
  - Note that actual systems don't support that much physical memory
  - However, tens or hundreds of GB of physical memory is not uncommon

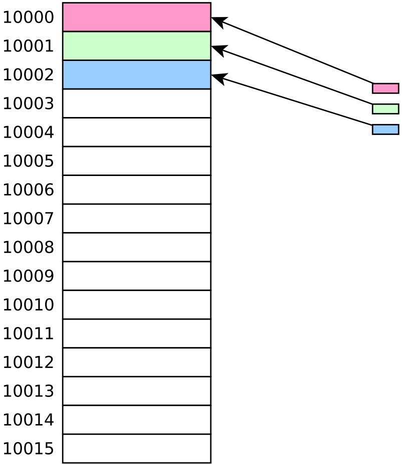
Alignment

- To store the value of an *n*-bit word in memory, *n* contiguous bytes are used
- The address of the first byte is the address of the overall word
- *Typically*, an *n*-byte word must have an address that is an exact multiple of *n* ("natural" alignment)
  - For example, the first byte allocated for an 8-byte word must have an address that is an exact multiple of 8
- Attempt to load or store an *n*-byte word at an address that is not a multiple of *n* is an *unaligned access*
  - Best case: access works, reduced performance
  - Worst case: runtime exception that kills the program

Visualizing alignment



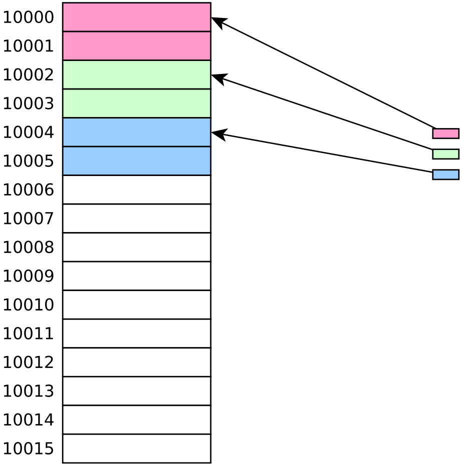
Visualizing alignment



char variables (1 byte) can be placed at any address

```
char a;
char b;
char c;
```

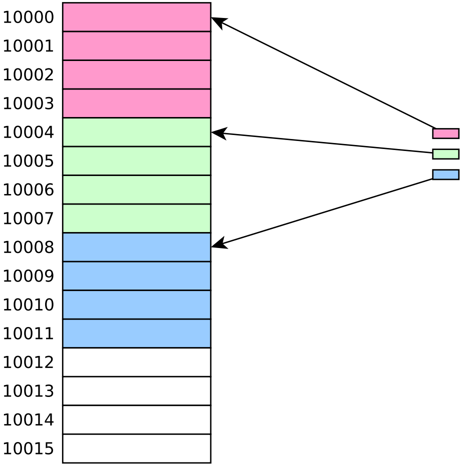
Visualizing alignment




```
short variables (2 bytes)
must be aligned on an
address that is a multiple
of 2

short a;
short b;
short c;
```

Visualizing alignment

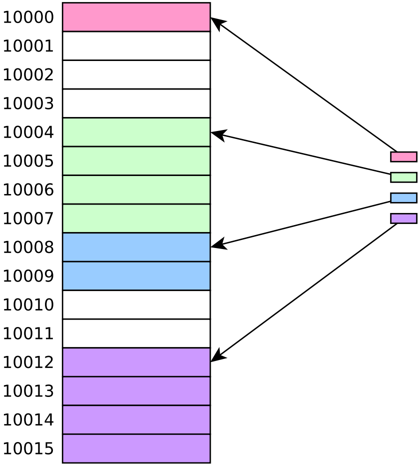



```
int variables (4 bytes)
must be aligned on an
address that is a multiple
of 4

int a;
int b;
int c;
```

Visualizing alignment

When variables with different alignment requirements are defined, the compiler may insert padding to ensure correct alignment



C pointers

- *Pointers* in C are just memory addresses!
- The address-of operator (`&`), when applied to a variable, yields a pointer to the variable (i.e., the address of the first memory byte that is part of the variable's storage)
- The dereference operator ($*$), when applied to a pointer value (address), refers to the variable whose storage location is indicated by the address

Example C program

```
#include <stdio.h>
#include <stdlib.h>

long g;

int main(void) {
  long* p = malloc(sizeof(long));
  long x;
  int a, b;
  short c, d, e, f;
  scanf("%ld %ld %ld %d %d %hd %hd %hd %hd",
        p, &g, &x, &a, &b, &c, &d, &e, &f);
  long sum = *p + g + x + a + b + c + d + e + f;
  printf("%ld\n", sum);
  printf("%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n",
         p, &g, &x, &a, &b, &c, &d, &e, &f);
  return 0;
}
```

Running example C program

```
$ gcc address.c
$ ./a.out
1 2 3 4 5 6 7 8 9
45
0x56142dfba260
0x56142c265018
0x7ffc7e6b2fd0
0x7ffc7e6b2fc8
0x7ffc7e6b2fcc
0x7ffc7e6b2fc0
0x7ffc7e6b2fc2
0x7ffc7e6b2fc4
0x7ffc7e6b2fc6
```

Running example C program

```
$ gcc address.c
$ ./a.out
1 2 3 4 5 6 7 8 9
45
0x56142dfba260      <-- address of malloc'ed buffer
0x56142c265018
0x7ffc7e6b2fd0
0x7ffc7e6b2fc8
0x7ffc7e6b2fcc
0x7ffc7e6b2fc0
0x7ffc7e6b2fc2
0x7ffc7e6b2fc4
0x7ffc7e6b2fc6
```

Running example C program

```
$ gcc address.c
$ ./a.out
1 2 3 4 5 6 7 8 9
45
0x56142dfba260
0x56142c265018     <-- address of global variable
0x7ffc7e6b2fd0
0x7ffc7e6b2fc8
0x7ffc7e6b2fcc
0x7ffc7e6b2fc0
0x7ffc7e6b2fc2
0x7ffc7e6b2fc4
0x7ffc7e6b2fc6
```

Running example C program

```
$ gcc address.c
$ ./a.out
1 2 3 4 5 6 7 8 9
45
0x56142dfba260
0x56142c265018
0x7ffc7e6b2fd0     <-- address of long variable on stack
0x7ffc7e6b2fc8
0x7ffc7e6b2fcc
0x7ffc7e6b2fc0
0x7ffc7e6b2fc2
0x7ffc7e6b2fc4
0x7ffc7e6b2fc6
```

Running example C program

```
$ gcc address.c
$ ./a.out
1 2 3 4 5 6 7 8 9
45
0x56142dfba260
0x56142c265018
0x7ffc7e6b2fd0
0x7ffc7e6b2fc8      <-- address of int variable on stack
0x7ffc7e6b2fcc      <-- address of int variable on stack
0x7ffc7e6b2fc0             (note addresses differ by 4)
0x7ffc7e6b2fc2
0x7ffc7e6b2fc4
0x7ffc7e6b2fc6
```

Running example C program

```
$ gcc address.c
$ ./a.out
1 2 3 4 5 6 7 8 9
45
0x56142dfba260
0x56142c265018
0x7ffc7e6b2fd0
0x7ffc7e6b2fc8
0x7ffc7e6b2fcc
0x7ffc7e6b2fc0     \
0x7ffc7e6b2fc2      | <-- addresses of short variables on stack
0x7ffc7e6b2fc4      |          (note addresses differ by 2)
0x7ffc7e6b2fc6     /
```

Bitwise operations

Bitwise operations

- *Bitwise* operations operate on the binary (bit-level) representation of an integer data value
- Logical operations: and, or, exclusive or, complement
- Shifts: left shift, right shift

Operations on boolean values

We can think of bit values (1 or 0) as being *Boolean* values (true or false)

Logical operations on bits **a** and **b**:

a	b	and a & b	or a | b	xor a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Logical negation (“complement”) on a single bit **a**:

a	~a
0	1
1	0

Bitwise operations in C

- The C *bitwise operators* perform logical operations (and, or, xor, negation) on the *bits* of the binary representation(s) of integer values
  - For example, `x | y` computes a result whose bits are formed by applying the bitwise or operator (|) to each pair of bits in `x` and `y`
- Example code (bitwise *or*):

```
int x = 11;
int y = 40;
int z = x | y;
printf("%d\n", z);
```

- What does this code do?

Explanation of bitwise *or* example

```
int x = 11;
int y = 40;
int z = x | y;
printf("%d\n", z);
```

	decimal	binary

Explanation of bitwise *or* example

```
int x = 11;
int y = 40;
int z = x | y;
printf("%d\n", z);
```

	decimal	binary
x	$11 = 8 + 2 + 1$	00001011

Explanation of bitwise *or* example

```
int x = 11;
int y = 40;
int z = x | y;
printf("%d\n", z);
```

	decimal	binary
x	$11 = 8 + 2 + 1$	00001011
y	$40 = 32 + 8$	00101000

Explanation of bitwise *or* example

```
int x = 11;
int y = 40;
int z = x | y;
printf("%d\n", z);
```

	decimal	binary
x	$11 = 8 + 2 + 1$	00001011
y	$40 = 32 + 8$	00101000
x | y	$43 = 32 + 8 + 2 + 1$	00101011

Explanation of bitwise *or* example

```
int x = 11;
int y = 40;
int z = x | y;
printf("%d\n", z);
```

	decimal	binary
x	$11 = 8 + 2 + 1$	00001011
y	$40 = 32 + 8$	00101000
x | y	$43 = 32 + 8 + 2 + 1$	00101011

Bit is 1 in result if corresponding bit is 1 in either operand value

Shifts

- Shifts move bits to the left or right in the binary representation of a data value
- Example code (left shift):

```
int x = 21;
int y = x << 3;
printf("%d\n", y);
```

- What does this code do?

Explanation of left shift example

```
int x = 21;
int y = x << 3;
printf("%d\n", y);
```

	decimal	binary

Explanation of left shift example

```
int x = 21;
int y = x << 3;
printf("%d\n", y);
```

	decimal	binary
x	$21 = 16 + 4 + 1$	00010101

Explanation of left shift example

```
int x = 21;
int y = x << 3;
printf("%d\n", y);
```

	decimal	binary
`x`	$21 = 16 + 4 + 1$	00010101
`x << 3`	$168 = 128 + 32 + 8$	10101000

Explanation of left shift example

```
int x = 21;
int y = x << 3;
printf("%d\n", y);
```

	decimal	binary
`x`	$21 = 16 + 4 + 1$	00010101
`x << 3`	$168 = 128 + 32 + 8$	10101000

Each bit in original value is shifted 3 places to the left; the lowest 3 bits of result become 0

Why bitwise operations are useful

- Bitwise operations (logical operations and shifts) are useful because they allow precise manipulations of data values at the level of individual bits:
  - Selecting arbitrary bits
  - Clearing or setting arbitrary bits

Bitwise idioms

Set bit n of variable x to 1

```
x |= (1 << n);
```

Set bit n of variable x to 0

```
x &= ~(1 << n);
```

Get just the lowest n bits of variable x

```
x & ~(~0U << n)
```