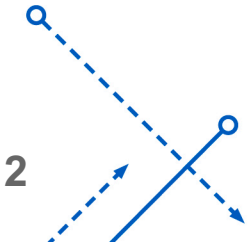
Bits, Bytes and Integers

Karthik Dantu
Ethan Blanton
Computer Science and Engineering
University at Buffalo
kdantu@buffalo.edu

Administrivia

- PA1 due this Friday – test early and often!
  - We cannot help everyone on Friday!
  - Don't expect answers on Piazza into the night and early morning
- Avoid using actual numbers (80, 24 etc.) – use macros!
- Lab this week is on testing
- Programming best practices
- Lab Exam – four students have already failed class!
  - Lab exams are EXAMS – no using the Internet, submitting solutions from dorm, home
  - Please don't give code/exam to friends – we will know!



Everything is Bits

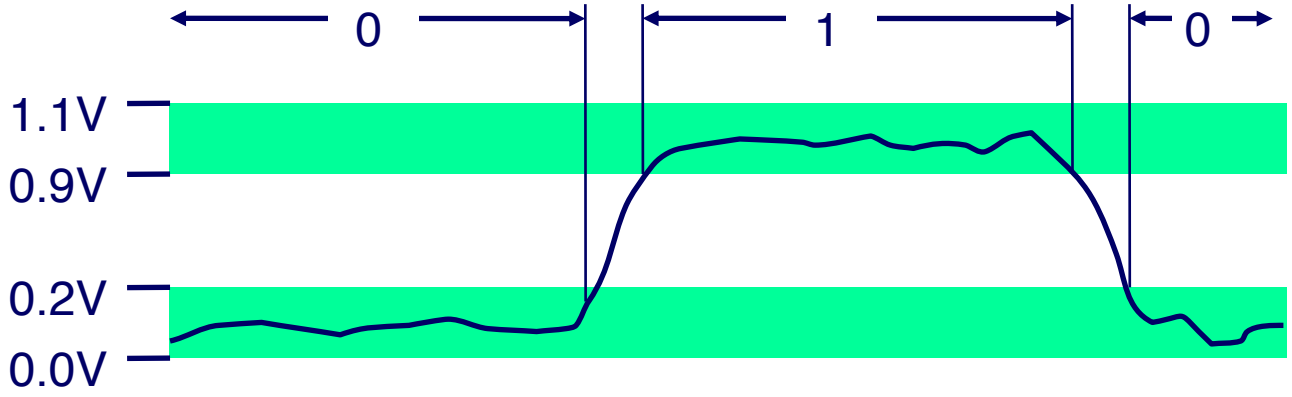
- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways

  Computers determine what to do (instructions)

  … and represent and manipulate numbers, sets, strings, etc…
- Why bits? Electronic Implementation

  Easy to store with bistable elements

  Reliably transmitted on noisy and inaccurate wires



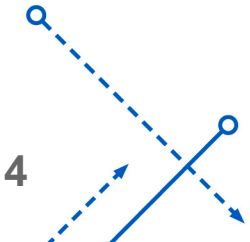
Memory as Bytes

- To the computer, memory is just bytes
- Computer doesn't know data types
- Modern processor can only manipulate:

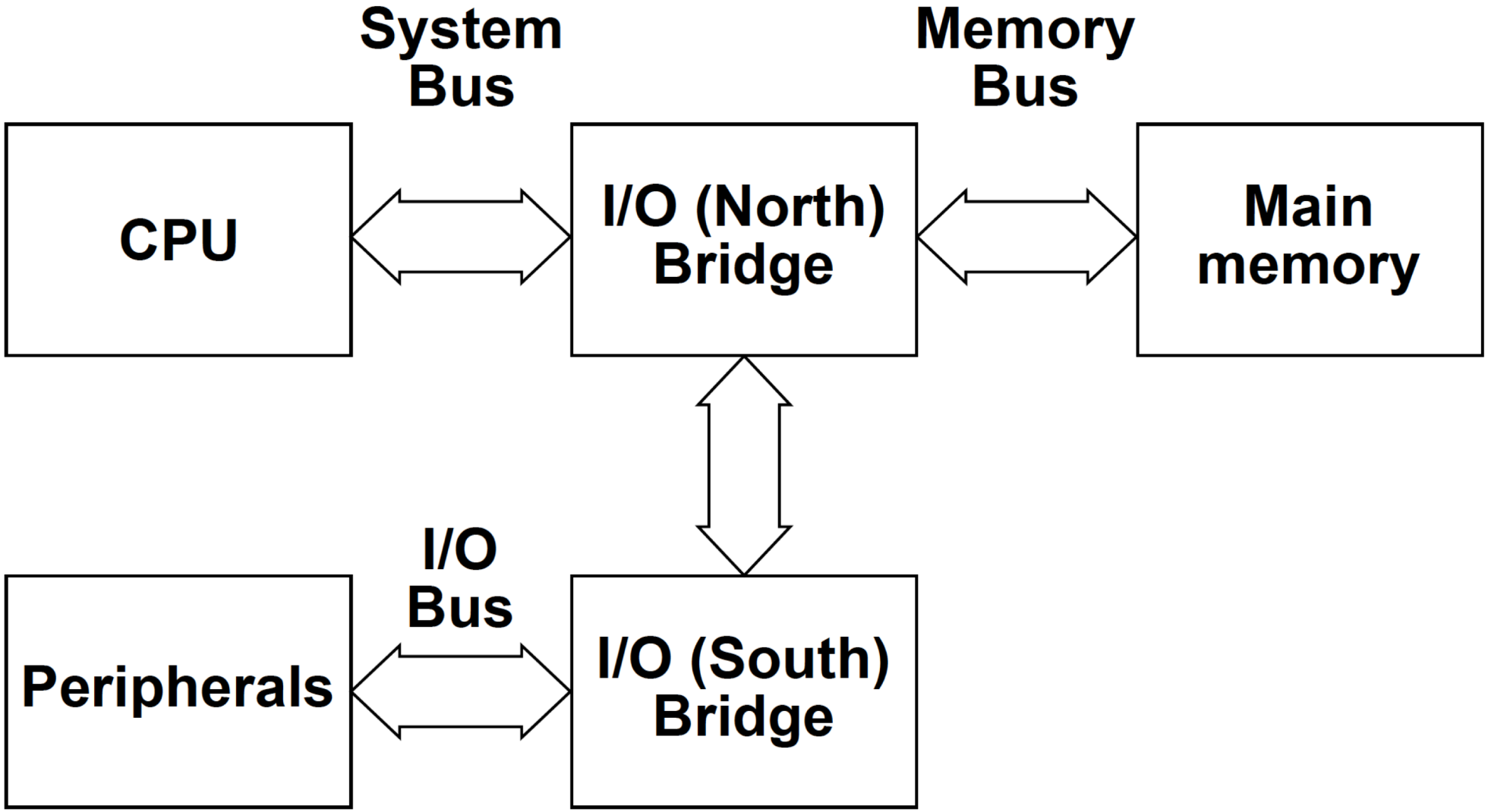
  Integers (Maybe only single bits)

  Maybe floating point numbers

  … repeat !
- Everything else is in software

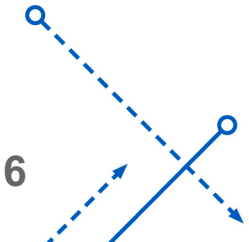


Reminder: Computer Architecture



Buses

- Each bus has a width, which is literally the number of wires it has
- Each wire transmits one bit per transfer
- Each bus transfer is of that width, though some bits might be ignored
- Therefore, memory has a word size from the viewpoint of the CPU: the number of wires on that bus



CPU to Memory Transfer

- CPU fetches data from memory in words the width of the memory bus
- It places that data in registers the width of the CPU word
- The register width is the native integer size
- These word widths may or may not be the same
  - It is in x86-64
- If they are not, a transfer may require:
  - Multiple registers
  - Multiple memory transfers

Imposing Structure on Memory

- Programming languages expose things such as:
  Booleans
  Strings
  Structures
  Classes
- How? -> We impose meaning on words in memory by convention

E.g., We discussed previously that a C string is a sequence of bytes that are adjacent in memory

Counting in Binary

- Base 2 Number Representation

  Represent 15213_{10} as 11101101101101_2

  Represent 1.20_{10} as $1.0011001100110011[0011]\ldots_2$

  Represent 1.5213×10^4 as $1.1101101101101_2 \times 2^{13}$

- Byte = 8 bits

  Binary 00000000_2 to 11111111_2

  Decimal: 0_{10} to 255_{10}

  Hexadecimal 00_{16} to FF_{16}

  Base 16 number representation

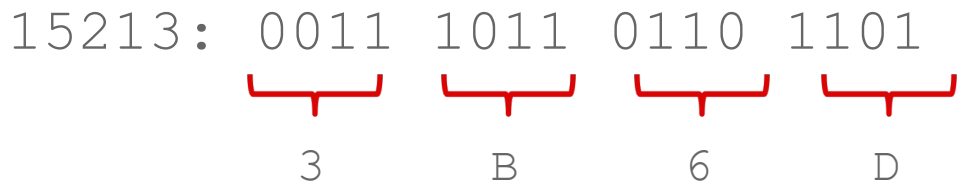
  Use characters '0' to '9' and 'A' to 'F'

  Write $FA1D37B_{16}$ in C as

  0xFA1D37B

  0xfa1d37b

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111



Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode "True" as 1 and "False" as 0

And

- A&B = 1 when both A=1 and B=1

&	0	1
0	0	0
1	0	1

Or

- A|B = 1 when either A=1 or B=1

|	0	1
0	0	1
1	1	1

Not

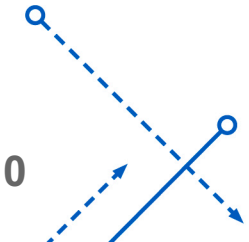
- ~A = 1 when A=0

~	
0	1
1	0

Exclusive-Or (Xor)

- A^B = 1 when either A=1 or B=1, but not both

^	0	1
0	0	1
1	1	0



Generalized Boolean Algebra

- Operate on Bit Vectors

  Operations applied bitwise

```
  01101001     01101001      01101001
& 01010101   | 01010101    ^ 01010101     ~ 01010101
```

- All of the Properties of Boolean Algebra Apply

Bit-Level Operations in C

- Operations &, |, ~, ^ available in C
  - Apply to any "integral" data type
    - `long, int, short, char, unsigned`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- Examples (`char` data type)
  - `~0x41` →
  - `~0x00` →
  - `0x69 & 0x55` →
  - `0x69 | 0x55` →

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Example Data Representations

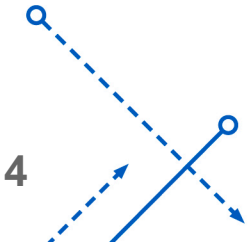
C Data Type	Typical 32-bit	Typical 64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`float`	4	4	4
`double`	8	8	8
pointer	4	8	8

Integer Modifiers

- Every integer type may have modifiers
- Those modifiers include `signed` and `unsigned`
- All unmodified integer types except `char` are signed
- `char` may be signed or unsigned
- Following are equivalent:

```
int x;
signed int x;
```

```
long long x;
signed long long int x;
```



Integers of Explicit Size

- Confusion in size has led to definition of explicitly sized integers
- Definitions are in `<stdint.h>`
- Exact width types are of the form `intN_t`
- They are exactly *N* bits wide – e.g., `int32_t`
- There are also unsigned equivalent types, which start with `u`:

e.g., `unit32_t`

- *N* can be 8, 16, 32, 64

sizeof()

- `sizeof()` looks like a function but it is not
- It is computed by the compiler
- `sizeof()` returns the size in bytes of its argument, which can be:
  - A variable
  - An expression that is "like" a variable
  - A type

```
char str[32];
int matrix[2][3];

sizeof(int);
// 4
sizeof(str);
// 32
sizeof(matrix);
//24
```

dump_mem()

- Function to examine memory
- Takes a memory address and number of bytes
- Prints the hex value of the bytes at that address

```
int x=98303; // 0x17fff
dump_mem(&x, sizeof(x));
```

Output:

```
ff 7f 01 00
```

??

Byte Ordering

- Why is `98303 (0x17fff)` represented by `ff 7f 01 00`?
- Answer is Endianness
- Words are organized into bytes in memory – but in what order?
- Big Endian – “big end” comes first. How we write.
- Little Endian – “little end” comes first. How x86 processors represent integers
- NOTE: Cannot assume anything about byte ordering in C

Sign Extension

```
char c = 0x80;
int i = c;
dump_mem(&i, sizeof(i));
```

OUTPUT:

```
80 ff ff ff
```

Required readings

- B&O 2.1, 2.2