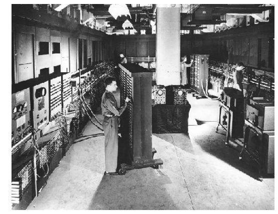
Lecture 16: Numbers

http://www.cs.virginia.edu/cs216

ENIAC

- Started 1943 – early electronic programmable computer
- Operational in 1946
- Computed ballistics tables
- 17,468 vacuum tubes
- 150 kW of power



Earlier Computers:
Z3 (Konrad Zuse) 1941
Colossus 1943

Directions for Getting **6**

1. Choose any regular accumulator (ie. Accumulator #9).
2. Direct the Initiating Pulse to terminal *5i*.
3. The initiating pulse is produced by the initiating unit's *Io* terminal each time the Eniac is started. This terminal is usually, by default, plugged into Program Line 1-1 (described later). Simply connect a program cable from Program Line 1-1 to terminal *5i* on this Accumulator.
4. Set the Repeat Switch for Program Control 5 to 6.
5. Set the Operation Switch for Program Control 5 to ADD.
6. Set the Clear-Correct switch to C.
7. Turn on and clear the Eniac.
8. Normally, when the Eniac is first started, a clearing process is begun. If the Eniac had been previously started, or if there are random neons illuminated in the accumulators, the ``Initial Clear'' button of the Initiating device can be pressed.
9. Press the ``Initiating Pulse Switch'' that is located on the Initiating device.
10. **Stand back.**

ENIAC number representation

- Decimal system
  - Ring of 36 vacuum tubes to store one digits (10 flip-flops to store 0-9)
  - Designed to emulate mechanical adding machine electronically
  - 20 accumulators (~registers), each stores 10-digits
- 5,000 cycles per second
  - Perform addition/subtraction between 2 accumulators each cycle

Binary Number Representations

- First presented by Gottfried Leibniz, 1705 ("Explication de l'Arithmétique Binaire") See http://www.cs.virginia.edu/evans/academic-roots.html for Leibniz's advising relationship to me (academic great^{14}-grandadvisor!)
- George Boole ("Boolean" logic), 1854
- Claude Shannon's 1937 Master's thesis: implemented Boolean algebra with switches and relays
- Used by Atanasoff-Berry Computer, Colossus and Z3

Binary Representation

$$b_{n-1}b_{n-2}b_{n-3}...b_2b_1b_0$$

$$\text{Value} = \sum_{i=0..n-1} b_i * 2^i$$

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 carry 1

What should n be?

What is n?

- Java:
  - byte, char = 8 bits
  - short = 16 bits
  - **int = 32 bits**
  - long = 64 bits
- C: implementation-defined
  - **int**: can hold between 0 and UINT_MAX
    - UINX_MAX must be at least 65535

      n >= 16, typical current machines n = 32
- Python? n is not fixed (numbers work)

The Great Debate

- "Big Endian": most significant **first** (lowest address)

  1000 0000 0000 0000 = 2^{15} = 32768
- "Little Endian": most significant **last** (highest address)

  1000 0000 0000 0000 = 2^{0} = 1

Which is better?

Endianness

- Its a "religious" argument: names taken from Big-Endians and Little-Endians in *Gulliver's Travels* who argued over which end of an egg to crack
- Different orderings problematic
  - Consider what << means in C
    - big endian ~ divide by 2
    - little endian ~ multiply by 2
- Some architectures support both ("bi-endian"): PowerPC, DEC Alpha, IA/64
- Most Internet standards: big-endian

Other Kinds of Numbers

- Positive and Negative Integers
  - Sign Bit, Ones Complement, Twos Complement
  - Section this week
- Real numbers

Real Numbers

1/3

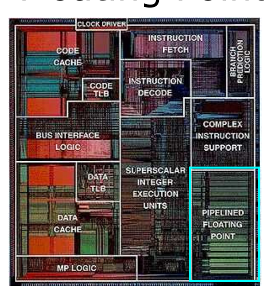
π

0.1

3.333333333333… * 10^{-1}

$\sqrt{2}$

Floating Point



Pentium II

IEEE Floating Point Single Precision (32 bits)

1	8 bits	23 bits
Sign	Exponent	Fraction
31	30 … 23	22 … 0

Exponent values:
- 0 — zeroes
- 1-254 — exp + 127
- 255 — infinities, NaN

Value = $(1 - 2*Sign)\ (1 + Fraction)^{Exponent\ -\ 127}$

Fraction

$b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9 b_{10} b_{11} b_{12} b_{13} b_{14} b_{15} b_{16} b_{17} b_{18} b_{19} b_{20} b_{21} b_{22} b_{23}$

$$Fraction = \sum_{i = 1..23} b_i / 2^i$$

IEEE Floating Point Single Precision (32 bits)

1	8 bits	23 bits
Sign	Exponent	Fraction
31	30 … 23	22 … 0

Value = $(1 - 2*Sign)\ (1 + Fraction)^{Exponent\ -\ 127}$

What is the largest float?

exponent = 11111111 = 255

fraction = $1 + \sum_{i = 1..23} 1/2^i$

Example

1/10 = 0.1 (Decimal)

What is this in binary?

$1/10 \approx \underbrace{1/16 + 1/32}_{3/32}$

$.2/32 = 2/320 \approx \underbrace{1/256 + 1/512}_{3/512\ =\ 1.875/320}$

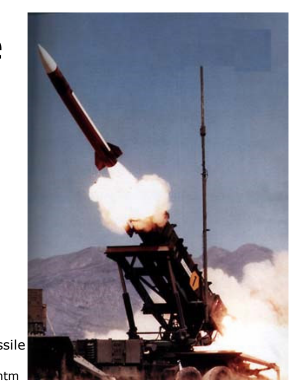
=0011001100110011…


```
          0.0011001100110011001100110011…
1010 | 10.000000000000000000000000000
        1 010
          1100
          1010
            10000
             1010
              1100
               1010
```

Even common decimals like 0.1 cannot be represented exactly!

Patriot Missile

- Gulf War I
- Failed to intercept incoming Iraqi scud missile (Feb 25, 1991)
- 28 American soldiers killed



GAO Report: GAO/IMTEC-92-26 Patriot Missile Software Problem
http://www.fas.org/spp/starwars/gao/im92026.htm

Patriot Design

- Intended to operate only for a few hours
  - Defend Europe from Soviet aircraft and missile
- Four 24-bit registers (1970s design!)
- Kept time with integer counter: incremented every 1/10 second
- Calculate speed of incoming missile to predict future positions:
  - velocity = $loc_1 - loc_0/(count_1 - count_0) * 0.1$
- But, cannot represent 0.1 exactly!

Floating Imprecision

- 24-bits:

$0.1 = 1/2^4 + 1/2^5 + 1/2^8 + 1/2^9 + 1/2^{12} + 1/2^{13} + 1/2^{16} + 1/2^{17} + 1/2^{20} + 1/2^{21}$

$= 209715 / 2097152$

Error is 0.2/2097152 = 1/10485760

One hour = 3600 seconds

3600 * 1/10485760 * 10 = 0.0034s

20 hours = 0.0687s Miss target! (137 meters)

Two weeks before the incident, Army officials received Israeli data indicating some loss in accuracy after the system had been running for 8 consecutive hours. Consequently, Army officials modified the software to improve the system's accuracy. However, the modified software did not reach Dhahran until February 26, 1991--the day after the Scud incident.

GAO Report

Better Floating Point: Use More Bits

- IEEE Double Precision (64 bits)

1	11 bits	52 bits
Sign	Exponent	Fraction

Single Precision:
0.1 = 209715/2097152
Error = $9.5*10^{-8}$ (20 hours to miss target)

Double Precision:
0.1 = 56294995342131/562949953421312
Error = $3.608 *10^{-16}$ (2,172,375,450 **years** to miss)

Better Floating Point (?)

- IBM Floating Point ("Hexadecimal")
  - Use more bits in fraction, fewer in exponent (7/24 and 7/56 instead of 8/23 and 11/52)
- Decimal Formats (IEEE 754d)
  - Naive: 1 decimal digit into 4 binary digits
  - Cowlishaw encoding:
    - Exact representation of decimals (e.g., 0.1)
    - 3 decimal digits (0-999) into 10 binary digits (0-1023) (24 wasted out of 1024)

Smaller Floating Point

- 16-bit floating point representations
  - Minifloat: 1 sign, 5-bit exponent (-15), 10-bit mantissa
  - Range from 2.98×10^{-8} to 65504

NVIDIA.

Your graphics card uses this (if you have a good one)

40B Floating Point Ops per second (3GHZ Pentium = 12B)

High Dynamic Range

(Example from Paul Debevec's HDRShop)

8-bit integer color

16-bit float color

Charge

- If you have to worry about how numbers are represented, you are doing low-level programming
- Are there any high-level programming languages yet?
  - Java: only if you never use floating point numbers or integers bigger than 2 147 483 647 (can keep track of National Debt for about 23 hours)
  - Python: almost a "high-level language" (but still need to worry about floating point numbers)
  - Scheme (PLT implementation): is a "high-level" language (code used to calculate error values)

Code

```
; smarter implementation would compute these...
(define seq (list 4 5 8 9 12 13 16 17 20 21))
(define seq64 (list 4 5 8 9 12 13 16 17 20 21 24 25 28 29 32 33
                    36 37 40 41 44 45 48 49))
(define (value seq)
  (if (null? seq) 0 (+ (/ 1 (expt 2 (car seq))) (value (cdr seq)))))
```


DrScheme Interactions

```
> (define onetenth (value seq))
> onetenth
209715/2097152
> (define onetenth64 (value seq64))
> onetenth64
56294995342131/562949953421312
> (- .1 onetenth)
9.536743164617612e-008
> (- .1 onetenth64)
3.608224830031759e-016
> (* 20 3600 (- .1 onetenth))
0.006866455078852468
> (/ (* 20 3600 (- .1 onetenth)) (- .1 onetenth64))
19030008943384.617
> (/ (/ (/ (* 20 3600 (- .1 onetenth)) (- .1 onetenth64)) 24) 365)
2172375450.1580615
```