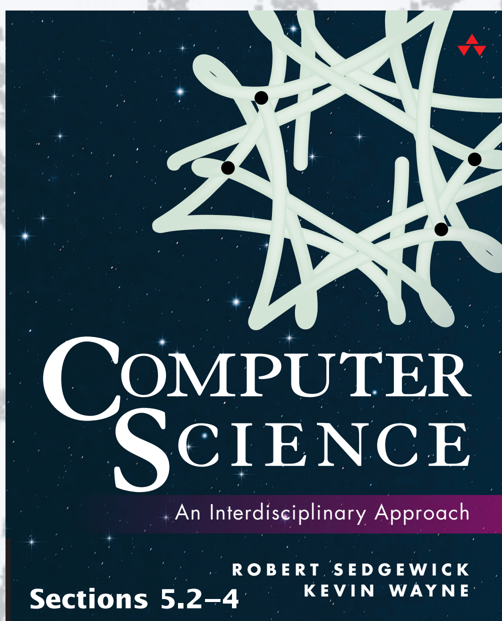
COMPUTER SCIENCE
SEDGEWICK / WAYNE

PART II: ALGORITHMS, MACHINES, and THEORY



http://introcs.cs.princeton.edu

18. von Neumann Machines

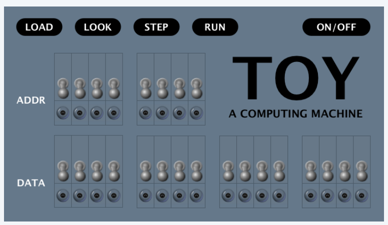
18. von Neumann machines

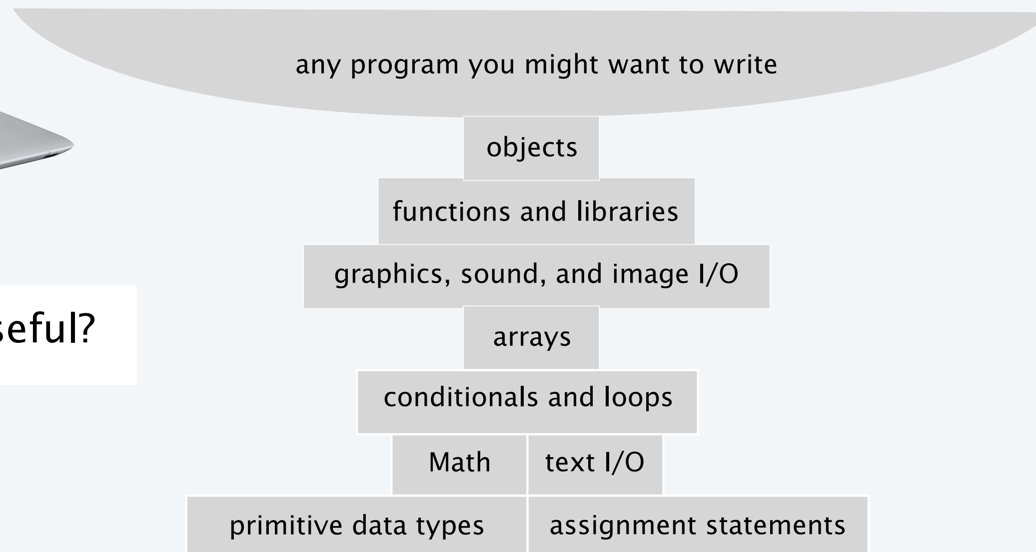
- **Perspective**
- A note of caution
- Practical implications
- Simulation

TOY vs. your laptop

Two different computing machines

- Both implement basic data types, conditionals, loops, and other low-level constructs.
- Both can have arrays, functions, libraries, and other high-level constructs.
- Both have infinite input and output streams.



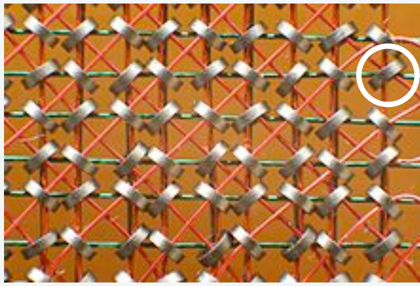


Q. Is 256 words enough to do anything useful?

A. Yes! (Stay tuned.)

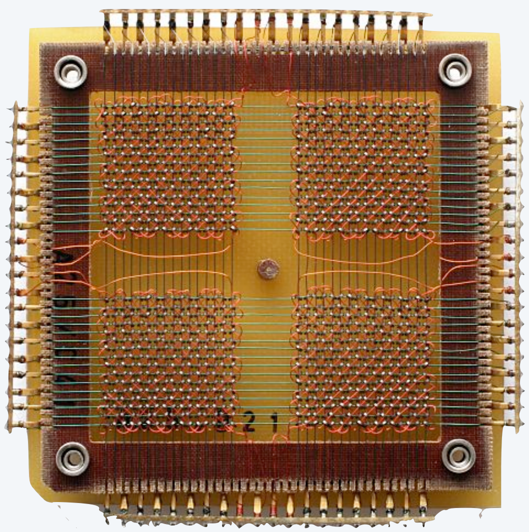
OK, we definitely want a faster version with more memory when we can afford it...

Is 4096 bits of memory enough to do anything useful?



1 bit

1024 bits



Core memory from the Apollo Guidance Computer, 1966–1975

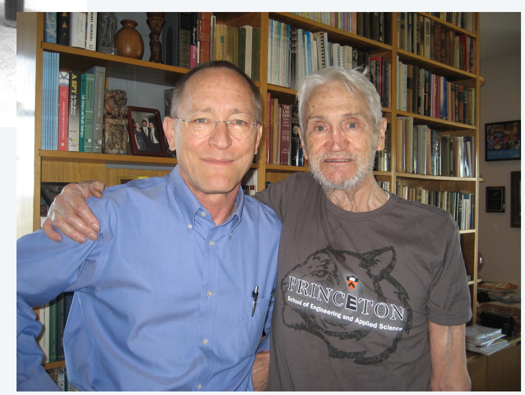
Is thousands of bits of memory enough to do anything useful?

LINC computer, MIT

12×2048 = 24576 bits of memory

Used for many biomedical and other experiments

Wes Clark, 1963



Doug Clark and his father Wes, 2013

Is 4096 bits of main memory enough to do anything useful?

Contents of memory, registers, and PC at a particular time
- Provide a record of what a program has done.
- Completely determines what the machine will do.

Total number of bits in the state of the machine
- 255×16 (memory)
- 15×16 (registers)
- 8 (PC)

Total number of different states: $2^{4328} > 10^{1302}$ (!!!)

Total number of different states that could be observed *if every electron in the universe had a supercomputer examining states for its entire lifetime*: $<< 10^{109}$.

Estimates	
Age of the universe:	10^{17} seconds
Size of the universe:	10^{79} electrons
instructions per second:	10^{13}

Bottom line: We will never know what a machine with 4096 bits of main memory can do.

An early computer

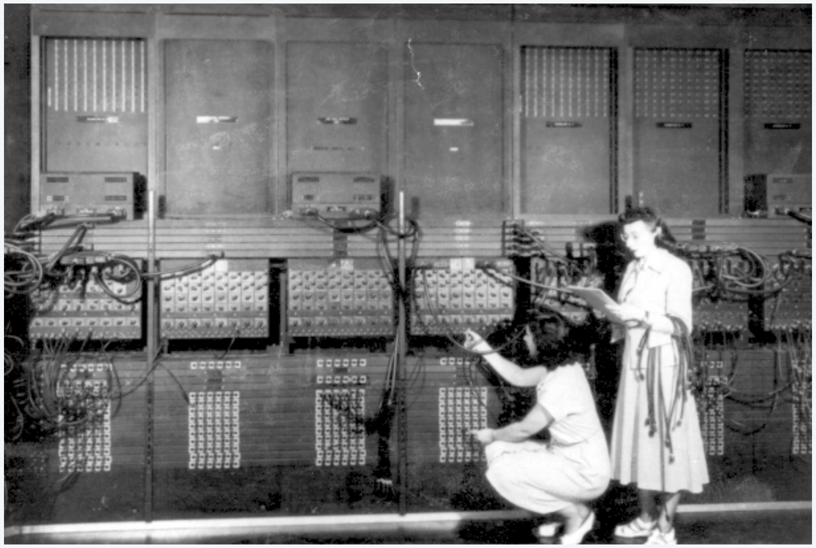
ENIAC. Electronic Numerical Integrator and Calculator

- First widely-known general-purpose electronic computer.
- Conditional jumps, programmable, but *no memory*.
- Programming: Change switches and cable connections.
- Data: Enter numbers using punch cards.

John W. Mauchly
1907–1980

J. Presper Eckert
1919–1995

ENIAC
1946



Facts and figures
30 tons
30 x 50 x 8.5 ft
17,468 vacuum tubes
300 multiply/sec

A bit

A famous memo

First Draft of a report on the EDVAC, 1945

- Written by John von Neumann, Princeton mathematician
- EDVAC: second computer proposed by Eckert and Mauchly.
- Memo written on a train trip to Los Alamos.
- A brilliant summation of the *stored-program* concept.
- Influenced by theories of Alan Turing.
- *Has influenced the design of every computer since.*

John von Neumann
1903–1957

Who invented the stored-program computer?

- Fascinating controversy.
- Eckert-Mauchly discussed the idea before von Neumann arrived on the scene.
- Goldstine circulated von Neumann's first draft because of intense interest in the idea.
- Public disclosure prevented EDVAC design from being patented.
- von Neumann never took credit for the idea, but never gave credit to others, either.

Another early computer

EDSAC. Electronic Delay Storage Automatic Calculator

- Another *stored-program* computer (just after EDVAC).
- Data and instructions encoded in binary.
- Could load programs, not just data, into memory.
- Could change program without rewiring.

Maurice Wilkes
1913–2010

EDSAC
1949

Facts and figures
512 17-bit words (8074 bits)
2 registers
16 instructions
input: paper tape
output: teleprinter

A bit

Implications

Stored-program (*von Neumann*) architecture is the basis of nearly all computers since the 1950s.

Practical implications

- Can load programs, not just data, into memory (download apps).
- Can write programs that produce programs as *output* (compilers).
- Can write programs that take programs as *input* (simulators).

Profound implications (see theory lectures)

- TOY can solve *any problem* that *any other* computer can solve (!)
- Some problems *cannot be solved* by *any computer at all* (!!)

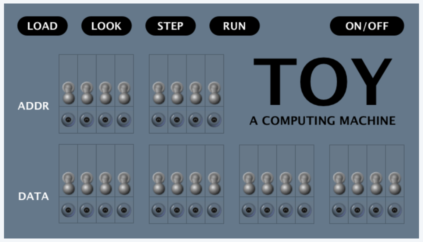


Image sources

http://en.wikipedia.org/wiki/Magnetic-core_memory#/media/File:KL_CoreMemory.jpg

http://en.wikipedia.org/wiki/Apollo_(spacecraft)#/media/File:Apollo_17_Command_Module_AS17-145-22261HR.jpg

http://www.computerhistory.org/timeline/?year=1962

http://www.computermuseum.li/Testpage/05HISTORYCD-ENIAC-Photos-I.htm

http://www.seas.upenn.edu/about-seas/eniac/mauchly-eckert.php

http://en.wikipedia.org/wiki/John_von_Neumann#/media/File:JohnvonNeumann-LosAlamos.gif

http://www.american-rails.com/humming-bird.html

http://en.wikipedia.org/wiki/Electronic_Delay_Storage_Automatic_Calculator

18. von Neumann machines

- Perspective
- A note of caution
- Practical implications
- Simulation

Arrays

To implement an array

- Keep items in an array contiguous starting at memory address a.
- Access a[i] at M[a+i].

To access an array element, use *indirection*

- Keep array address in a register.
- Add index
- Indirect load/store uses *contents* of a register.

opcode	*instruction*
7	load address
A	load indirect
B	store indirect

Array of length 11

80	0 0 0 0
81	0 0 0 1
82	0 0 0 1
83	0 0 0 2
84	0 0 0 3
85	0 0 0 5
86	0 0 0 8
87	0 0 0 D
88	0 0 1 5
89	0 0 2 2
8A	0 0 3 7

Example: Indirect store

12	7A80	Load the address 80 into R[A]	array starts at mem location 80
13	7900	Set R[9] to 0	i is the index
...			
16	1CA9	R[C] = R[A] + R[9]	compute address of a[i]
17	BD0C	M[R[C]] = R[D]	a[i] = d
18	1991	R[9] = R[9] + 1	increment i
...			

Arrays example: Read an array from standard input

To implement an array

- Keep items in an array contiguous starting at M[a].
- Access a[i] at M[a+i].

Note: this example is simplified for this lecture.

Array processing in the book includes the length, so arrays can be passed as arguments and return values to functions.

PC			
→	10	**7 1 0 1**	R[1] = 1
	11	**8 B F F**	R[B] = *stdin*
	12	**7 A 8 0**	R[A] = 80
	13	**7 9 0 0**	R[9] = 0
	14	**2 2 B 9**	R[2] = R[B] - R[9]
	15	**C 2 1 B**	if (R[2] == 0) PC = 1B
	16	**1 C A 9**	R[C] = R[A] + R[9]
	17	**8 D F F**	R[D] = *stdin*
	18	**B D 0 C**	M[R[C]] = R[D]
	19	**1 9 9 1**	R[9] = R[9] + 1
	1A	**C 0 1 4**	PC ← 14
	1B		[array processing code]
		...	

```
N = StdIn.read();
a = address of a[0];
i = 0;
while (i < N)
{


  a[i] = StdIn.read();
  i = i + 1;
}
```

Arrays example: Read an array from standard input

Register trace

1	1		9	0	1	2	3	4	5	6
B	6		C	80	81	82	83	84	85	
A	80		D	1	2	3	5	8	D	

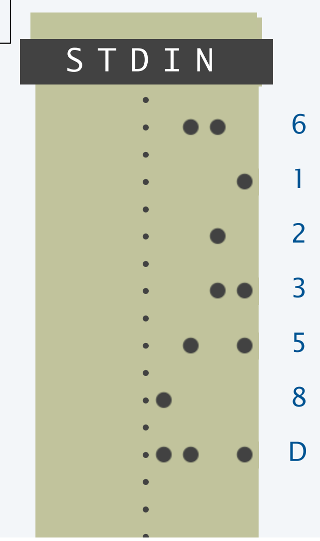
Memory

80	0 0 0 1
81	0 0 0 2
82	0 0 0 3
83	0 0 0 5
84	0 0 0 8
85	0 0 0 D
. . .	

PC → 10

10	7 1 0 1	R[1] = 1
11	8 B F F	R[B] = *stdin*
12	7 A 8 0	R[A] = 80
13	7 9 0 0	R[9] = 0
14	2 2 B 9	R[2] = R[B] - R[9]
15	C 2 1 B	if (R[2] == 0) PC = 1B
16	1 C A 9	R[C] = R[A] + R[9]
17	8 D F F	R[D] = *stdin*
18	B D 0 C	M[R[C]] = R[D]
19	1 9 9 1	R[9] = R[9] + 1
1A	C 0 1 4	PC ← 14
1B		[array processing code]
. . .		

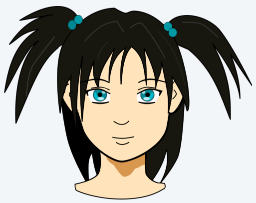
```
N = StdIn.read();
a = address of a[0];
i = 0;
while (i < N)
{
   a[i] = StdIn.read();
   i = i + 1;
}
```



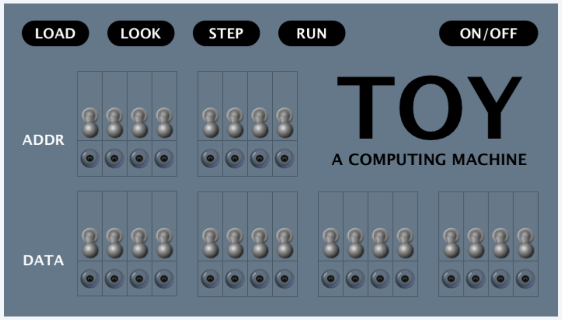
An instructive scenario

Alice, a scientist, develops a procedure for her experiments.

- Uses a scientific instrument connected to a paper tape punch.
- Takes the paper tape to a *computer* to process her data.
- Uses array code just described to load her data.
- Writes array-processing code that analyzes her data.
- Punches out the results on paper tape to save them.



Alice

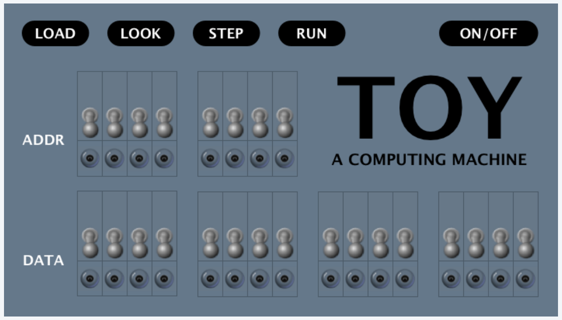


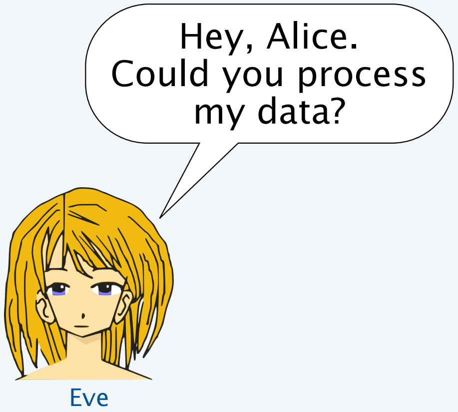
An instructive scenario (continued)

Alice, a scientist, develops a procedure for her experiments.

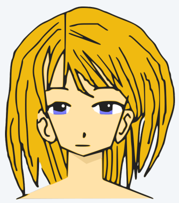
- Uses a scientific instrument connected to a paper tape punch.
- Takes the paper tape to a *computer* to process her data.
- Uses array code from last lecture to load her data.
- Writes array-processing code that analyzes her data.

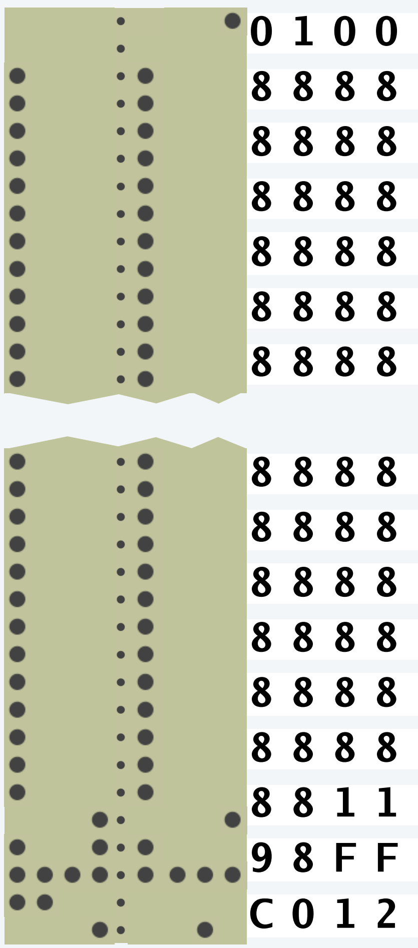
Eve, a fellow scientist, runs some experiments, too.





Eve's tape


Eve



256_{10} ? A first clue that something is fishy.

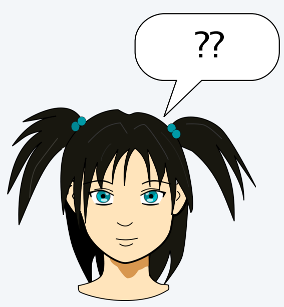
146 words, all **8888**.

Three additional suspicious words at the end.

What happens with Eve's tape

Not what Alice expects!

- Memory 80–FE fills with **8888**.
- **8888** appears on output.
- Address overflow from FF to 00.
- Memory 00–0F is overwritten.

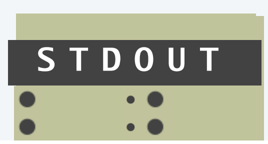


Memory

00	8 8 8 8	10	7 1 0 1		80	8 8 8 8		F0	8 8 8 8
01	8 8 8 8	11	8 B F F		81	8 8 8 8		F1	8 8 8 8
02	8 8 8 8	12	7 A 8 0		82	8 8 8 8		F2	8 8 8 8
03	8 8 8 8	13	7 9 0 0		83	8 8 8 8		F3	8 8 8 8
04	8 8 8 8	14	2 2 B 9		84	8 8 8 8		F4	8 8 8 8
05	8 8 8 8	15	C 2 1 B		85	8 8 8 8		F5	8 8 8 8
06	8 8 8 8	16	1 C A 9		86	8 8 8 8		F6	8 8 8 8
07	8 8 8 8	17	8 D F F	. . .	87	8 8 8 8	. . .	F7	8 8 8 8
08	8 8 8 8	18	B D 0 C		88	8 8 8 8		F8	8 8 8 8
09	8 8 8 8	19	1 9 9 1		89	8 8 8 8		F9	8 8 8 8
0A	8 8 8 8	1A	C 0 1 4		8A	8 8 8 8		FA	8 8 8 8
0B	8 8 8 8	1B	0 0 1 0		8B	8 8 8 8		FB	8 8 8 8
0C	8 8 8 8	1C	0 1 0 0		8C	8 8 8 8		FC	8 8 8 8
0D	8 8 8 8	1D	1 0 0 0		8D	8 8 8 8		FD	8 8 8 8
0E	8 8 8 8	1E	0 1 0 0		8E	8 8 8 8		FE	8 8 8 8
0F	8 8 8 8	1F	0 0 1 0		8F	8 8 8 8		FF	8 8 8 8

And then things get worse...

10	7 1 0 1	R[1] = 1
11	8 B F F	R[B] = *stdin*
12	7 A 8 0	R[A] = 80



What happens with Eve's tape when things get worse

	8 8 8 8	
	8 8 8 8	
	8 8 8 8	
	8 8 8 8	
10	8 8 8 8	
11	8 8 8 8	*stdin*
12	8 8 1 1	
13	9 8 F F	
14	C 0 1 2	
15	C 2 1 B	if (R[2] == 0) PC = 1B
16	1 C A 9	R[C] = R[A] + R[9]
17	8 D F F	R[D] = *stdin*
18	B D 0 C	M[R[C]] = R[D]
19	1 9 9 1	R[9] = R[9] + 1
1A	C 0 1 4	PC ← 14
1B		[array processing code]

...

PC → 18

Data is overwriting code!

Or is it *code* overwriting code?

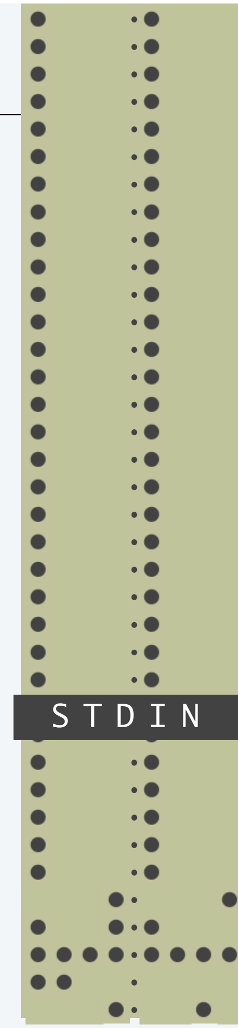
Register trace

D	8888	8888	8888	8811	98FF	C012
C	0F	10	11	12	13	14

```
int N = StdIn.read();
a = address of a[0];
int i = 0;
while (i < N)
{
  a[i] = StdIn.read();
  i = i + 1;
}
```

Memory

80	8 8 8 8
81	8 8 8 8
82	8 8 8 8
83	8 8 8 8
84	8 8 8 8
85	8 8 8 8
...	
FE	8 8 8 8
FF	



8888
8888
8888
8811
98FF
C012

What happens when things get worse: Eve Owns Alice's computer

	8 8 8 8
	8 8 8 8
	8 8 8 8
	8 8 8 8
10	8 8 8 8
11	**8 8 8 8**
12	**8 8 1 1**
13	**9 8 F F**
14	**C 0 1 2**
15	8 C F F
16	1 5 6 B
17	B C 0 5
18	1 B B 1
19	2 A A 1
1A	C 0 1 4
1B	

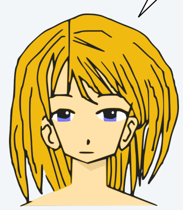
. . .

R[8] ← 8888

write R[8] to stdout

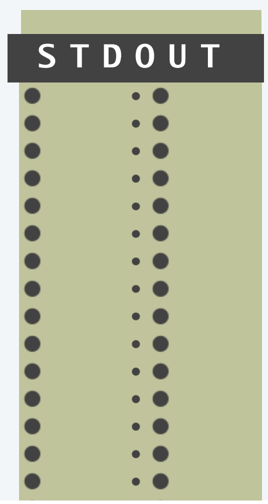
PC ← 12

Remember me?
[maniacal laugh]



Eve

She could have loaded *any program at all . . .*



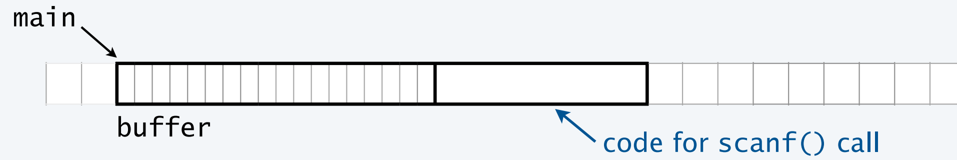
Buffer overflow in the real world

C/C++/Objective C string/array overflow
- Program does not check for long string.
- Hacker puts code at end of long string.
- Hacker *Owns* your computer.

```
#include <stdio.h>
int main(void)
{
   char buffer[100];
   scanf("%s", buffer);
   printf("%s\n", buffer);
   return 0;
}
```

← unsafe C code

Memory representation



1988
Morris Worm
infected research computers throughout US

2010-present
iPhone/iPad
Buffer overflow is "top 5 vulnerability"

2004
JPEG of death
Windows browsers buffer overflow on an image

2000s
Xbox/Zelda/Pokemon
Buffer overflow enables use of unlicensed games

Note: Java tries to help us write secure code
- Array bounds checking.
- Type safety.

COMPUTER SCIENCE
SEDGEWICK / WAYNE

CS.18.B.MachineII.Caution

18. von Neumann machines

- Perspective
- A note of caution
- **Practical implications**
- Simulation

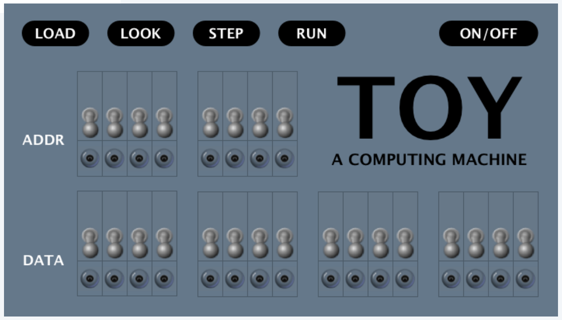
Programs that process programs on TOY

von Neumann architecture

- No difference between data and instructions.
- Same word can be data one moment, an instruction the next.

Early programmers immediately realized the advantages

- Can save programs on physical media (dump).
- Can load programs at another time (boot).
- Can develop higher-level languages (assembly language).



Dumping

Q. How to save a program for another day?

- Day's work represents patches and other code entered via switches.
- Must power off (vacuum tubes can't take the heat).

A. Write a short program to dump contents of memory to tape. ← Simplified version of book code (which can do partial dumps).

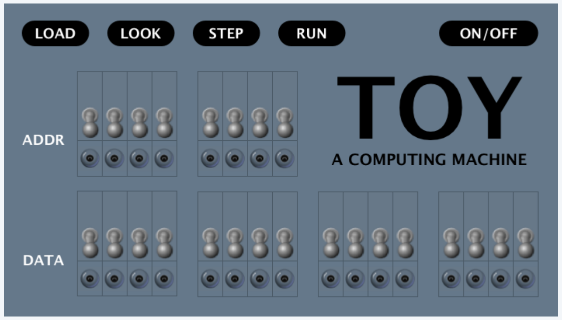
- Key in program via switches in memory locations 00-08.
- Run it to save data/instructions in memory 10-FE. ← Why not FF? It's StdIn/StdOut.

DUMP code

00	**7 1 0 1**	R[1] = 1
01	**7 2 1 0**	R[2] = 10
02	**7 3 F F**	R[3] = 00FF
03	**A A 0 2**	R[A] = M[R[A]]
04	**9 A F F**	write R[A] to stdout
05	**1 2 2 1**	R[2] = R[2] + 1
06	**2 4 3 2**	R[4] = 00FF - R[2]
07	**D 4 0 3**	if (R[4] > 0) PC = 03
08	**0 0 0 0**	halt

hex literal ↓

```
i = 0x10;
do {

   StdOut.print(M[i]);
   i++;

} while (i < 0xFF);
```



Booting

Q. How to load a program on another day?

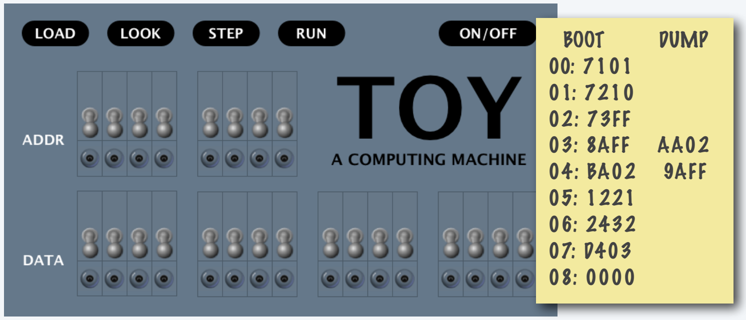
A. Reboot the computer.

- Turn it on.
- Key in *boot code* via switches in memory locations 00-08.
- Run it to load data/instructions in memory 10-FE. ← Why not 00-0F? Would overwrite boot program!

BOOT code

00	7 1 0 1	R[1] = 1
01	7 2 1 0	R[2] = 10
02	7 3 F F	R[3] = 00FF
03	8 A F F	R[A] = *stdin*
04	B A 0 2	M[R[2]] = R[A]
05	1 2 2 1	R[2] = R[2] + 1
06	2 4 3 2	R[4] = 00FF - R[2]
07	D 4 0 3	if (R4 > 0) PC = 03
08	0 0 0 0	halt

```
i = 0x10;
do {

      M[i] = StdIn.read();
      i++;

   } while (i < 0xFF);
}
```

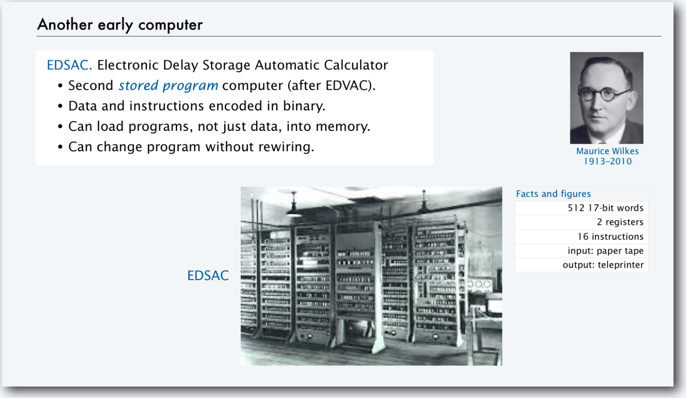


Early programmers would pride themselves on how fast they could enter such code

Assembly language

Assembly language

- Program in a higher-level language.
- Write a machine-language program to translate.
- Used widely from early days through the 1990s.
- Still used today.



First assembly language

	TOY machine code		TOY assembly code
00	7 0 0 1		LA R1,01
01	7 2 1 0		LA R2,10
02	7 3 F F		LA R3,FF
03	8 A F F	LOOP	RD RA
04	B A 0 2		SI RA,R2
05	1 2 2 1		A R2,R2,R1
06	2 4 3 2		S R4,R3,R2
07	D 4 0 3		BP R4, LOOP
08	0 0 0 0		H

Advantages

- Mnemonics, not numbers, for opcodes.
- Symbols, not numbers, for addresses.
- *Relocatable.*

Tip of the iceberg

Practical implications of von Neumann architecture

- Installers that download applications.
- Compilers that translate Java into machine language.
- Simulators that make one machine behave like another (stay tuned).
- Cross-compilers that translate code for one machine on another.
- Dumping and booting.
- Viruses.
- Virus detection.
- Virtual machines.
- Thousands of high-level languages.
- [an extremely long list]

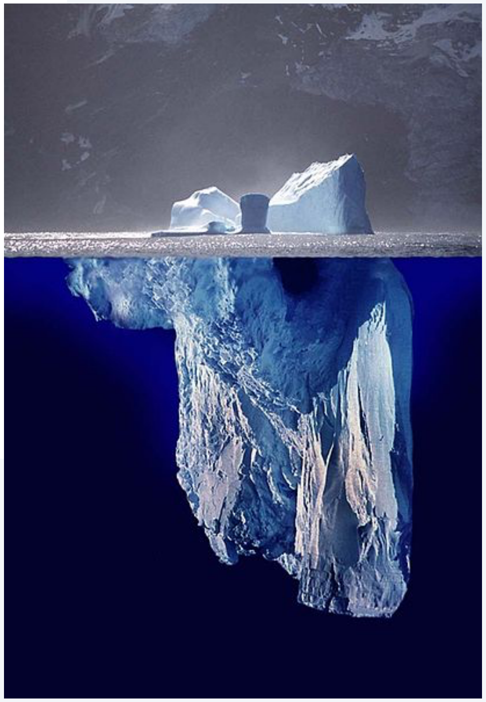


Image sources

http://commons.wikimedia.org/wiki/File:Iceberg.jpg

18. von Neumann machines

- Perspective
- A note of caution
- Practical implications
- **Simulation**

Is TOY real?

Q. How did we debug all our TOY programs?

A. We wrote a Java program to *simulate* TOY.

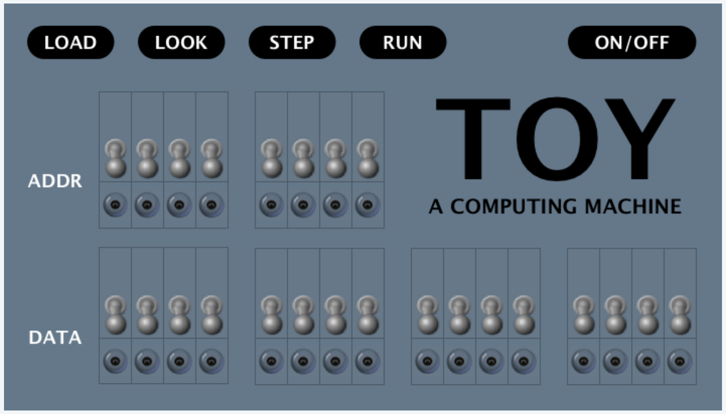
Comments

- YOU could write a TOY simulator (stay tuned).
- We designed TOY by refining this code.
- *All* computers are designed in this way.

Provocative questions

- Is Android real?
- Is Java real?
- Suppose we run our TOY simulator on Android. Is TOY real?

Estimated number of TOY devices: 0



Estimated number of Android devices: 1 billion+

Estimated number of TOY devices: 1 billion+

Toy simulator in Java

A Java program that simulates the TOY machine.

- Take program from a file named in the command line.
- Take TOY stdin/stdout from Java StdIn/StdOut.

```
public class TOYlecture
{
   public static void main(String[] args)
   {
      int pc  = 0x10;          // program counter
      int[] R = new int[16];   // registers
      int[] M = new int[256];  // main memory

      In in = new In(args[0]);
      for (int i = 0x10; i < 0xFF && !in.isEmpty(); i++)
            M[i] = Integer.parseInt(in.readString(), 16);

      while (true)
      {
         int ir = M[pc++];  // fetch and increment
         // decode  (next slide)
         // execute (second slide following)
      }
   }
}
```

base 16

```
% more add-stdin.toy
8C00
8AFF
CA15
1CCA
C011
9CFF
0000

% more data.txt
00AE
0046
0003
0000

% java TOY add-stdin.toy < data.txt
00F7
```

TOY code to add ints on StdIn

data

TOY simulator: decoding instructions

Bitwhacking is the same in Java as in TOY

- Extract fields for both instruction formats.
- Use shift and mask technique.

Example: Extract destination d from 1CAB

ir

1				C				A				B			
0	0	0	1	1	1	0	0	1	0	1	0	1	0	1	1

ir >> 8

0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0

0xF

0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

(ir >> 8) & 0xF

0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
												C			

Bitwise AND of data and "mask"
result is 0 where mask is 0
data bit where mask is 1

decode

```
int ir = M[pc++];              // fetch and increment
int op   = (ir >> 12) &  0xF;  // opcode   (bits 12-15)
int d    = (ir >>  8) &  0xF;  // dest d   (bits 08-11)
int s    = (ir >>  4) &  0xF;  // source s (bits 04-07)
int t    = (ir >>  0) &  0xF;  // source t (bits 00-03)
int addr = (ir >>  0) & 0xFF;  // addr     (bits 00-07)
```

TOY simulator: executing instructions

Use Java switch statement to implement the simple state changes for each instruction.

execute

```
if (op == 0) break;       // halt

switch (op)
{
   case  1: R[d] = R[s] +  R[t];      break;
   case  2: R[d] = R[s] -  R[t];      break;
   case  3: R[d] = R[s] &  R[t];      break;
   case  4: R[d] = R[s] ^  R[t];      break;
   case  5: R[d] = R[s] << R[t];      break;
   case  6: R[d] = R[s] >> R[t];      break;
   case  7: R[d] = addr;              break;
   case  8: R[d] = M[addr];           break;
   case  9: M[addr] = R[d];           break;
   case 10: R[d] = M[R[t]];           break;
   case 11: M[R[t]] = R[d];           break;
   case 12: if (R[d] == 0) pc = addr; break;
   case 13: if (R[d] >  0) pc = addr; break;
   case 14: pc = R[d];                break;
   case 15: R[d] = pc; pc = addr;     break;
}
```

Toy simulator in Java

load

fetch/inc

decode

execute

```
public class TOYlecture
{
   public static void main(String[] args)
   {
      int pc  = 0x10;          // program counter
      int[] R = new int[16];   // registers
      int[] M = new int[256];  // main memory

      In in = new In(args[0]);
      for (int i = 0x10; i < 0xFF && !in.isEmpty(); i++)
           M[i] = Integer.parseInt(in.readString(), 16);

      while (true)
      {
         int ir = M[pc++];  // fetch and increment

         int op   = (ir >> 12) &  0xF;  // opcode   (bits 12-15)
         int d    = (ir >>  8) &  0xF;  // dest d   (bits 08-11)
         int s    = (ir >>  4) &  0xF;  // source s (bits 04-07)
         int t    = (ir >>  0) &  0xF;  // source t (bits 00-03)
         int addr = (ir >>  0) & 0xFF;  // addr     (bits 00-07)
         if (op == 0) break;       // halt

         switch (op)
         {
            case  1: R[d] = R[s] +  R[t];      break;
            case  2: R[d] = R[s] -  R[t];      break;
            case  3: R[d] = R[s] &  R[t];      break;
            case  4: R[d] = R[s] ^  R[t];      break;
            case  5: R[d] = R[s] << R[t];      break;
            case  6: R[d] = R[s] >> R[t];      break;
            case  7: R[d] = addr;              break;
            case  8: R[d] = M[addr];           break;
            case  9: M[addr] = R[d];           break;
            case 10: R[d] = M[R[t]];           break;
            case 11: M[R[t]] = R[d];           break;
            case 12: if (R[d] == 0) pc = addr; break;
            case 13: if (R[d] >  0) pc = addr; break;
            case 14: pc = R[d];                break;
            case 15: R[d] = pc; pc = addr;     break;
         }
      }
   }
}
```

Important TOY design goal:

Simulator must fit on one slide for this lecture!

A few omitted details.

- R[0] is always 0 (put R[0] = 0 before execute).
- StdIn/StdOut (add code to do it if addr is FF).
- Need casts and bitwhacking in a few places because TOY is 16-bit and Java is 32-bit.
- Need more flexible input format to allow for loading programs elsewhere in memory.

See full implementation `TOY.java` on booksite

Toy simulator in Java

```
public class TOYlecture
{
   public static void main(String[] args)
   {
      int pc  = 0x10;          // program counter
      int[] R = new int[16];  // registers
      int[] M = new int[256]; // main memory

      In in = new In(args[0]);
      for (int i = 0x10; i < 0xFF && !in.isEmpty(); i++)
             M[i] = Integer.parseInt(in.readString(), 16);

      while (true)
      {
         int ir = M[pc++];  // fetch and increment

         int op   = (ir >> 12) &  0xF;  // opcode   (bits 12-15)
         int d    = (ir >>  8) &  0xF;  // dest d   (bits 08-11)
         int s    = (ir >>  4) &  0xF;  // source s (bits 04-07)
         int t    = (ir >>  0) &  0xF;  // source t (bits 00-03)
         int addr = (ir >>  0) & 0xFF;  // addr     (bits 00-07)
         if (op == 0) break;       // halt

         switch (op)
         {
            case  1: R[d] = R[s] +  R[t];      break;
            case  2: R[d] = R[s] -  R[t];      break;
            case  3: R[d] = R[s] &  R[t];      break;
            case  4: R[d] = R[s] ^  R[t];      break;
            case  5: R[d] = R[s] << R[t];      break;
            case  6: R[d] = R[s] >> R[t];      break;
            case  7: R[d] = addr;              break;
            case  8: R[d] = M[addr];           break;
            case  9: M[addr] = R[d];           break;
            case 10: R[d] = M[R[t]];           break;
            case 11: M[R[t]] = R[d];           break;
            case 12: if (R[d] == 0) pc = addr; break;
            case 13: if (R[d] >  0) pc = addr; break;
            case 14: pc = R[d];                break;
            case 15: R[d] = pc; pc = addr;     break;
         }
      }
   }
}
```

Comments.

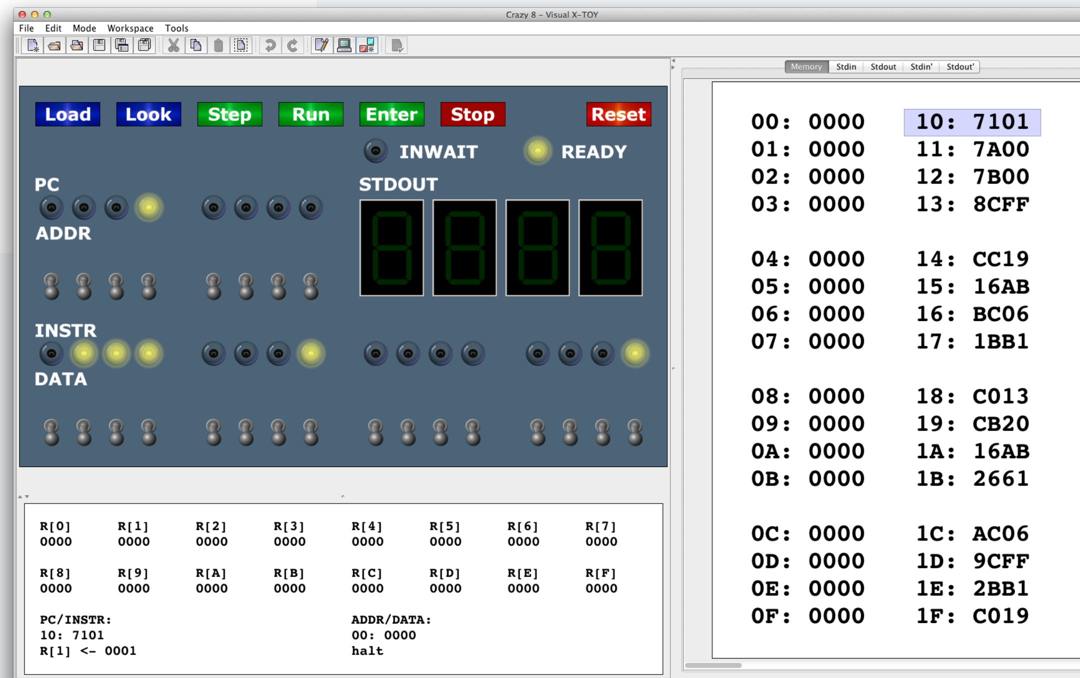
- Runs any TOY program!
- Easy to change design.
- Can develop TOY code on another machine.
- Could implement in TOY (!!).

```
% more read-array.toy
7100
8AFF
7680
...

% more eves-tape.txt
0100
8888
8888
....

% java TOYlecture read-array.toy < eves-tape.txt
8888
8888
8888
8888
8888
```

Toy development environment

Another Java program that simulates the TOY machine

- Includes *graphical* simulator.
- Includes single stepping, full display of state of machine, and many other features.
- Includes many simple programs.
- Written by a graduate of this course.
- Available on the booksite.
- YOU can develop TOY software.



Same approach used for *all* new systems nowadays

- Build simulator and development environment.
- Develop and test software.
- Build and sell hardware.

Backward compatibility

Q. Time to build a new computer. What to do about old software?

Approach 1: Rewrite it all

- Costly and time-consuming.
- Error-prone.
- Boring.

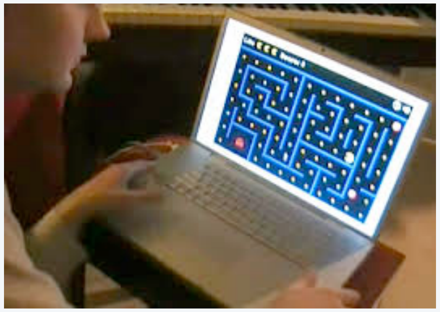
Approach 2: Simulate the old computer on the new one.

- Not very difficult.
- Still likely more efficient.
- Succeeds for *all* old software.

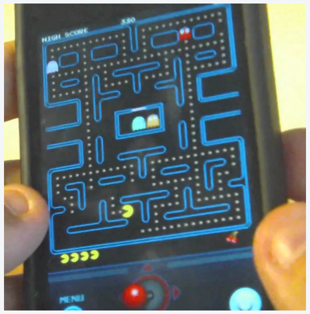
Result. Old software remains available.

Disturbing thought: Does anyone know how it works?

PacMac machine 1980s



PacMac on a laptop 2000s



PacMac on a phone 2010s

Another note of caution

An urban legend about backward compatability.

- Space shuttle solid rocket booster needed to be transported by rail.
- US railroads were built by English expats, so the standard rail gauge is 4 feet 8.5 inches.
- English rail gauge was designed to match ruts on old country roads.
- Ruts on old country roads were first made by Roman war chariots.
- Wheel spacing on Roman war chariots was determined by the width of a horse’s back end.

End result. Key space shuttle dimension determined by the width of a war horse’s back end.

Worthwhile takeaway. Backwards compatability is Not Necessarily Always a Good Thing.

Backward compatibility is pervasive in today's world

Documents need backward compatibility with .doc format

Broadcast TV needs backward compatibility with analog B&W

Business software is written in a dead language and run with many layers of emulation

Airline scheduling uses 1970s software

web pages need compatibility with new and old browsers

iPhone software is written in an unsafe language

Much of our infrastructure was built in the 1970s on machines not so different from TOY.

Time to design and build something suited for today's world? Go for it! ⟵ That means YOU !

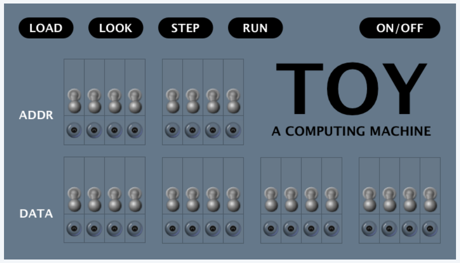
Virtual machines

Building a new rocket? Simulate it to test it.

- Issue 1: Simulation may not reflect reality.
- Issue 2: Simulation may be too expensive.

Building a new *computer*? Simulate it to test it.

- Advantage 1: Simulation *is* reality (it defines the new machine).
- Advantage 2: Can develop software without having machine.
- Advantage 3: Can simulate machines that may never be built.



A machine that may never be built

Examples in today's world.

- Virtual memory.
- Java virtual machine.
- Amazon cloud.

Virtual machines of many, many types
(old and new) are available for use on the web.

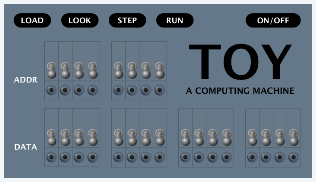
Internet commerce is moving to such machines.

Forming a startup? Use a virtual machine.
It is likely to perform *better* for you than
whatever real machine you might be able to afford.

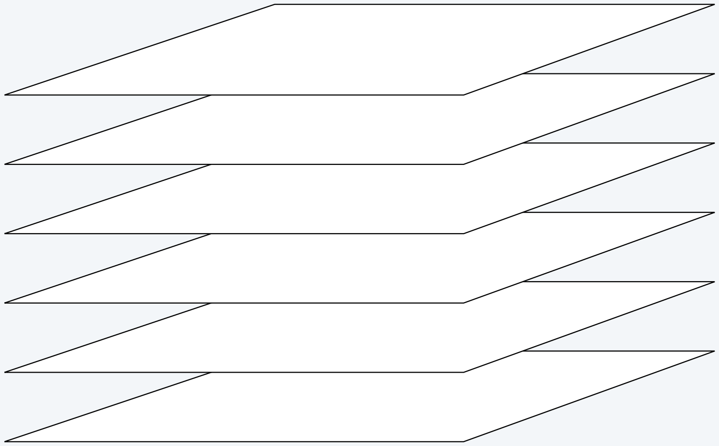
Layers of abstraction

Computer systems are built by accumulating layers of abstraction.

Is TOY real?



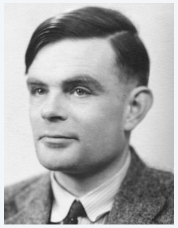
Is *your computer* real?



Approaching a new problem?

- Build an (abstract) language for expressing solutions.
- Design an (abstract) machine to run programs written in the language.
- Food for thought: Why build the machine? ← Just simulate it instead!

Turing and von Neumann

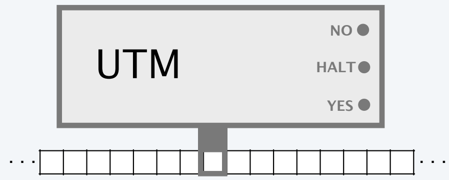


Alan Turing
1912–1954

Theorem (Turing, 1936). *It is possible to invent a single machine which can be used to do any computable task.*

Proof sketch. (See theory lectures.)

- Any task can be described as a Turing machine.
- A "universal" TM (UTM) can simulate any TM.
- Key concept: *Program as data.*

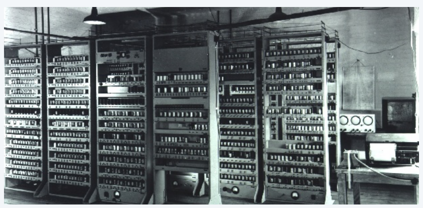


A virtual machine

John von Neumann
1903–1957

First Draft of a report on the EDVAC, (von Neumann,1945).

- A computer design with an ALU, memory, and I/O.
- Physical realization of *program as data* concept.



A UTM implementation

Bottom line: *Program as data* concept has always stood at the foundation of computer science.

Image sources

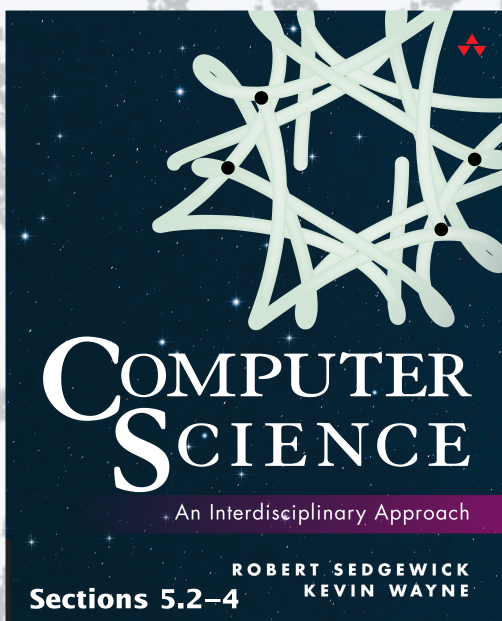
http://en.wikipedia.org/wiki/John_von_Neumann#/media/File:JohnvonNeumann-LosAlamos.gif

http://en.wikipedia.org/wiki/Electronic_Delay_Storage_Automatic_Calculator

http://en.wikipedia.org/wiki/Alan_Turing#/media/File:Alan_Turing_photo.jpg

COMPUTER SCIENCE
SEDGEWICK / WAYNE

PART II: ALGORITHMS, MACHINES, and THEORY



http://introcs.cs.princeton.edu

18. von Neumann Machines