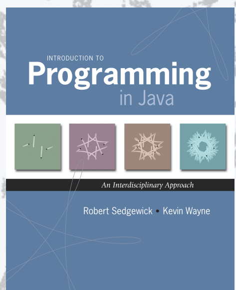
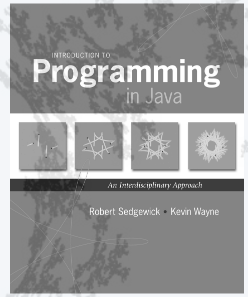
COMPUTER SCIENCE
S E D G E W I C K / W A Y N E



http://introcs.cs.princeton.edu

12. von Neumann Machines

COMPUTER SCIENCE
S E D G E W I C K / W A Y N E



http://introcs.cs.princeton.edu

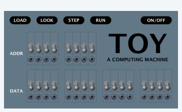
12. von Neumann machines

- **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.



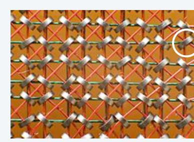
any program you might want to write
objects
functions and libraries
graphics, sound, and image I/O
arrays
conditionals and loops
Math | text I/O
primitive data types | assignment statements

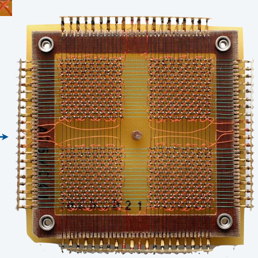
Q. Is 256 words enough to do anything useful?

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?





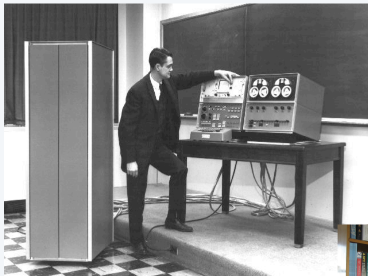
Core memory from the Apollo Guidance Computer, 1966–1975

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

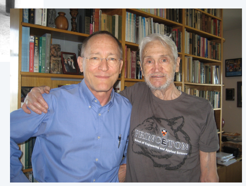
LINC computer, MIT

12×2048 = 24576 bits of memory

Used for many biomedical and other experiments



Prof. Clark's father, 1963



Prof. Clark and his father, 2013

Is 4096 bits 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
- 256 × 16 (memory)
- 16 × 16 (registers)
- 8 (PC)

Total number of different states: 2^{4360} (!!!)

Total number of different states that could be observed *if the universe were fully packed with laptops examining states for its entire lifetime*: << 2^{400}.

Estimates

Age of the universe:	2^{34} years
Size of the universe:	2^{267} cubic meters
Laptops per cubic meter:	2^{14}
States per year:	2^{60}

Bottom line: We will never know what a 256-word machine can do.

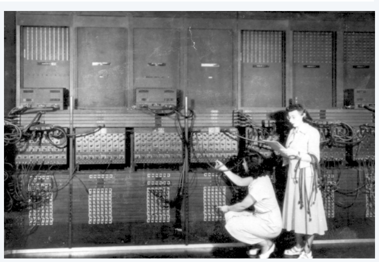
An early computer

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 to 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.
- Memo placed the idea in the public domain and prevented it 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

- Second *stored program* computer (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

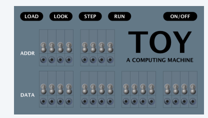
Stored-program (*von Neumann*) architecture is the basis of nearly all computers since the 1950s.

Practical implications

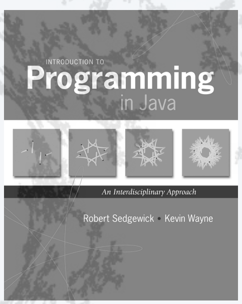
- Can load programs, not just data, into memory (download apps).
- Can write programs that produce programs as *output* (compilers).
- Can write programs that take programs as *input* (simulators).

Profound implications (stay tuned for theory lectures)

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



COMPUTER SCIENCE
SEDGEWICK / WAYNE

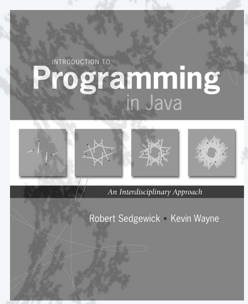


http://introcs.cs.princeton.edu

12. von Neumann machines

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

COMPUTER SCIENCE
SEDGEWICK / WAYNE



http://introcs.cs.princeton.edu

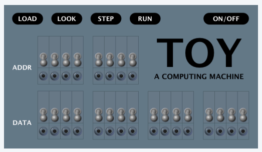
12. von Neumann machines

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

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 from last lecture to load her data.
- Writes array-processing code that analyzes her data.
- Punches out the results on paper tape to save them.

Alice



Arrays example: Read an array from standard input (continued from last lecture)

Register trace

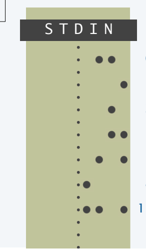
A	6	5	4	3	2	1	0
B	0	1	2	3	4	5	6
C		1	2	3	5	8	D

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
...	

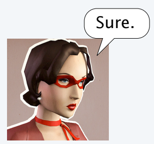
PC → 10

10	7 1 0 1	R1 ← 1	
11	8 A F F	RA ← N	`int a = StdIn.read();`
12	7 6 8 0	R6 ← 80	`arr = new int[];`
13	7 B 0 0	RB ← 0	`int b = 0;`
14	C A 1 B	if (RA == 0) PC ← 1B	`while (a != 0) {`
15	8 C F F	read RC from stdin	`int c = StdIn.read();`
16	1 5 6 B	R5 ← R6 + RB	
17	B C 0 5	mem[R5] ← RC	`arr[b] = c;`
18	1 B B 1	RB ← RB + 1	`b++;`
19	2 A A 1	RA ← RA - 1	`a--;`
1A	C 0 1 4	PC ← 14	`}`
1B		[begin array processing code]	
...			



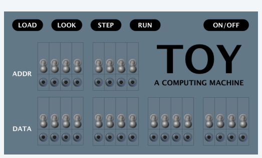
An instructive scenario (continued)

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.



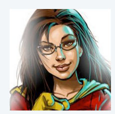
Alice

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

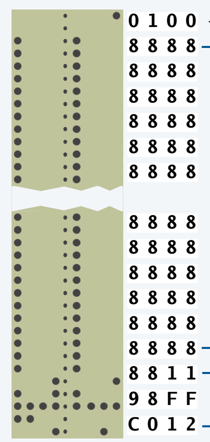


Eve

Eve's tape



Eve



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

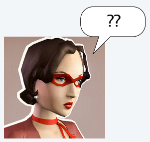
146 words, all **8 8 8 8**.

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.



Alice

Memory

00	8888	10	7101		80	8888		F0	8888
01	8888	11	8AFF		81	8888		F1	8888
02	8888	12	7680		82	8888		F2	8888
03	8888	13	7B00		83	8888		F3	8888
04	8888	14	CA1B		84	8888		F4	8888
05	8888	15	8CFF		85	8888		F5	8888
06	8888	16	156B		86	8888		F6	8888
07	8888	17	BC05	...	87	8888	...	F7	8888
08	8888	18	1BB1		88	8888		F8	8888
09	8888	19	2AA1		89	8888		F9	8888
0A	8888	1A	C014		8A	8888		FA	8888
0B	8888	1B	0010		8B	8888		FB	8888
0C	8888	1C	0100		8C	8888		FC	8888
0D	8888	1D	1000		8D	8888		FD	8888
0E	8888	1E	0100		8E	8888		FE	8888
0F	8888	1F	0010		8F	8888		FF	8888

And then things get worse...

STDOUT

What happens with Eve's tape when things get worse

	8888	
	8888	
	8888	
	8888	
10	8888	
11	8888	
12	8811	
13	98FF	
PC → 14	C012	
15	8CFF	read RC from stdin
16	156B	R5 ← R6 + RB
17	BC05	mem[R5] ← RC
18	1BB1	RB ← RB + 1
19	2AA1	RA ← RA − 1
1A	C014	PC ← 14
1B		[begin array processing code]
...		

Data is overwriting code!

Or is it *code* overwriting code?

Register trace

C	8888	8888	8888	8811	98FF	C012
5	0F	10	11	12	13	14

```
int a = StdIn.read();
arr = new int[];
int b = 0;
while (a != 0) {
  int c = StdIn.read();

  arr[b] = c;
  b++;
  a--;
}
```

Memory

80	8888
81	8888
82	8888
83	8888
84	8888
85	8888
...	
FE	8888
FF	

STDIN

8888
8888
8888
8811
98FF
C012

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

	8888	
	8888	
	8888	
	8888	
10	8888	
11	8888	
12	8811	R8 ← 8888
13	98FF	write R8 to stdout
14	C012	PC ← 12
15	8CFF	
16	156B	
17	BC05	
18	1BB1	
19	2AA1	
1A	C014	
1B		
...		

Eve

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

STDOUT

Buffer overflow in the real world

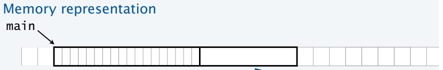
C/C++/Objective C string/array overflow

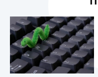
- Program does not check for long string.
- Hacker puts code at end of long string.
- Hacker *owns* your computer.

```
#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

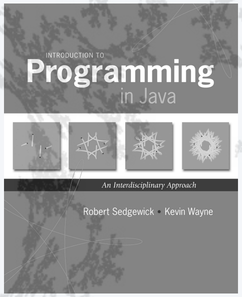
2010-present
iPhone/iPad
Buffer overflow is "top 5 vulnerability"

Note: Java tries to help us write secure code

- Array bounds checking.
- Type safety.

2004
.jpeg of death
Windows browsers buffer overflow on an image

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

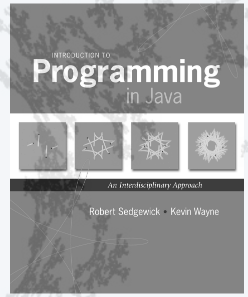


http://introcs.cs.princeton.edu

12. von Neumann machines

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

COMPUTER SCIENCE
SEDGEWICK / WAYNE



http://introcs.cs.princeton.edu

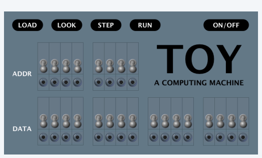
12. von Neumann machines

- 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).



TEQ 3 on TOY

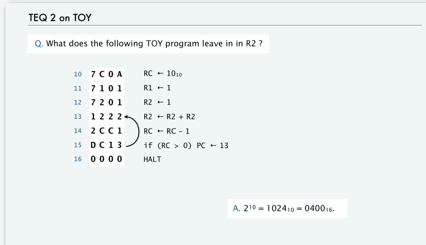
Q. What does the following program leave in R2?

10	7 C 0 A	RC ← 10_{10}
11	C 0 1 6	PC ← 12
12	1 2 2 2	R2 ← R2 + R2
13	2 C C 1	RC ← RC - 1
14	D C 1 2	if (RC > 0) PC ← 12
15	0 0 0 0	HALT
16	7 1 0 1	R1 ← 1
17	7 2 0 1	R2 ← 1
18	C 0 1 2	PC ← 12

TEQ 3 on TOY

Q. What does the following program leave in R2?

10	7 C 0 A	RC ← 10_{10}
11	C 0 1 6	PC ← 12
12	1 2 2 2	R2 ← R2 + R2
13	2 C C 1	RC ← RC - 1
14	D C 1 2	if (RC > 0) PC ← 12
15	0 0 0 0	HALT
16	7 1 0 1	R1 ← 1
17	7 2 0 1	R2 ← 1
18	C 0 1 2	PC ← 12


TEQ 2 on TOY

Q. What does the following TOY program leave in R2 ?

10	7 C 0 A	RC ← 10_{10}
11	7 1 0 1	R1 ← 1
12	7 2 0 1	R2 ← 1
13	1 2 2 2	R2 ← R2 + R2
14	2 C C 1	RC ← RC - 1
15	D C 1 3	if (RC > 0) PC ← 13
16	0 0 0 0	HALT

A. $2^{10} = 1024_{10} = 0400_{16}$.

A. $2^{10} = 1024_{10} = 0400_{16}$. Same as TEQ 2.

Example of a patch—very common in early programming.

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).

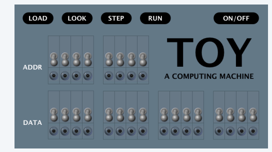
A. Write a short program to dump contents of memory to tape.
- Key in program via switches in memory locations 00-08. ← Why not FF? It's StdIn/StdOut.
- Run it to save data/instructions in memory 10-FE. ← Why not 00-0F? Stay tuned.

DUMP code

00	7 1 0 1	R1 ← 1
01	7 2 1 0	R2 ← 10
02	7 3 F F	R3 ← 00FF
03	A A 0 2	RA ← mem[R2]
04	9 A F F	write RA to stdout
05	1 2 2 1	R2 ← R2 + 1
06	2 4 3 2	R4 ← 00FF - R2
07	D 4 0 3	if (R4 > 0) PC ← 03
08	0 0 0 0	halt

hex literal

```
int i = 0x10;
do {
   a = mem[i];
   StdOut.print(a);
   i++;

   } while (i < 255)
}
```



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 program!

BOOT code

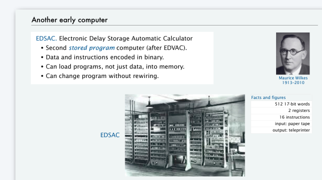
00	7 1 0 1	R1 ← 1
01	7 2 1 0	R2 ← 10
02	7 3 F F	R3 ← 00FF
03	8 A F F	read from stdin to RA
04	B A 0 2	mem[R2] ← RA
05	1 2 2 1	R2 ← R2 + 1
06	2 4 3 2	R4 ← 00FF - R2
07	D 4 0 3	if (R4 > 0) PC ← 03
08	0 0 0 0	halt

```
int i = 0x10;
do {
   StdIn.read(a);
   mem[i] = a;
   i++;

   } while (i < 255)
}
```

Early programmers would pride themselves in the speed 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

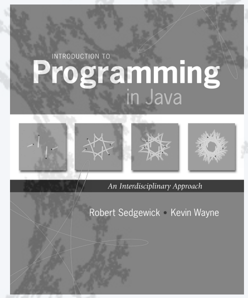
	TOY machine code	TOY assembly code
00	7 0 0 1	LA R1,01
01	7 2 1 0	LA R2,10
02	7 3 F F	LA R3,FF
03	8 A F F	LOOP RD RA
04	B A 0 2	SI RA,R2
05	1 2 2 1	A R2,R2,R1
06	2 4 3 2	S R4,R3,R2
07	D 4 0 3	BP R4, LOOP
08	0 0 0 0	H

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 make code for one machine on another.
- Dumping and booting.
- Viruses.
- Virus detection.
- Virtual machines.
- Thousands of high-level languages.
- [an extremely long list]

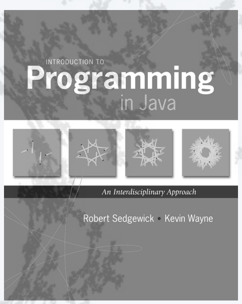


http://introcs.cs.princeton.edu

12. von Neumann machines

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

COMPUTER SCIENCE
SEDGEWICK / WAYNE



http://introcs.cs.princeton.edu

12. 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.

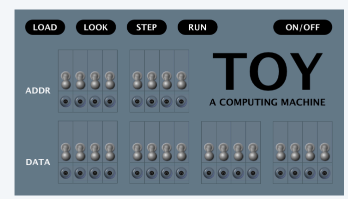
Comments

- YOU could write this program (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.

like StdIn but reads from a file (see text)

base 16

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

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

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

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

% more data
00AE
0046
0003
0000

% java TOY add-stdin.toy < data
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.

decode

```
int inst = mem[pc++];          // fetch and increment
int op   = (inst >> 12) &  15;  // opcode   (bits 12-15)
int d    = (inst >>  8) &  15;  // dest d   (bits 08-11)
int s    = (inst >>  4) &  15;  // source s (bits 04-07)
int t    = (inst >>  0) &  15;  // source t (bits 00-03)
int addr = (inst >>  0) & 255;  // addr     (bits 00-07)
```

Example: Extract destination d from 1CAB

inst

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

inst >> 8

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

15

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

(inst >> 8) & 15

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

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] = mem[addr];         break;
   case  9: mem[addr] = R[d];         break;
   case 10: R[d] = mem[R[t]];         break;
   case 11: mem[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[] mem = new int[256];  // main memory

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

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

         int op   = (inst >> 12) &  15;  // opcode   (bits 12-15)
         int d    = (inst >>  8) &  15;  // dest d   (bits 08-11)
         int s    = (inst >>  4) &  15;  // source s (bits 04-07)
         int t    = (inst >>  0) &  15;  // source t (bits 00-03)
         int addr = (inst >>  0) & 255;  // 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] = mem[addr];         break;
            case  9: mem[addr] = R[d];         break;
            case 10: R[d] = mem[R[t]];         break;
            case 11: mem[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.

- R0 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.

See full implementation `TOY.java` on booksite (also supports a more flexible input format)

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[] mem = new int[256]; // main memory

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

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

         int op   = (inst >> 12) &  15;  // opcode   (bits 12-15)
         int d    = (inst >>  8) &  15;  // dest d   (bits 08-11)
         int s    = (inst >>  4) &  15;  // source s (bits 04-07)
         int t    = (inst >>  0) &  15;  // source t (bits 00-03)
         int addr = (inst >>  0) & 255;  // 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] = mem[addr];         break;
            case  9: mem[addr] = R[d];         break;
            case 10: R[d] = mem[R[t]];         break;
            case 11: mem[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.

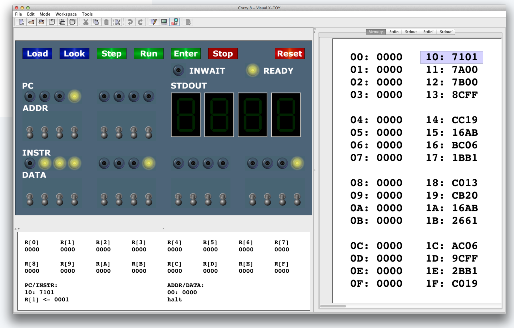
- Runs any TOY program!
- Easy to change design.
- Can develop TOY code on another machine.
- Could implement in TOY (!!).

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

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

% java TOYlecture read-array.toy < eves-tape
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 COS 126 graduate.
- 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.

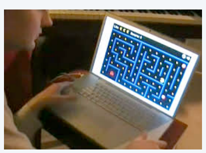
Approach 2: Simulate the old computer on the new one.

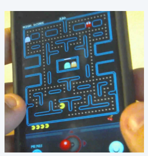
- Not very difficult.
- Still likely more efficient.
- Succeeds for *all* old software.

Result. Old software remains available.

Disturbing thought: Does anyone know how it works?

PacMac machine 1980s

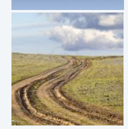
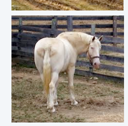

PacMac on a laptop 2000s


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 compatibility 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 !

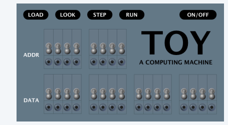
Virtual machines

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.

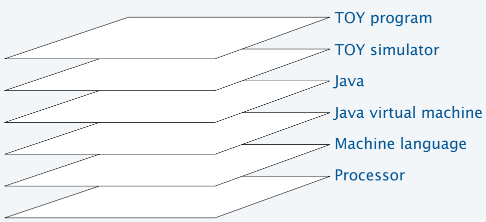
Internet commerce is moving to such machines.

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?

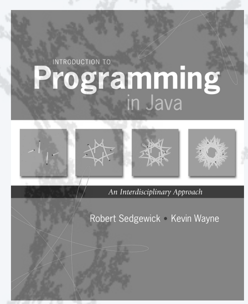


Is *your computer* real?

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!

COMPUTER SCIENCE
SEDGEWICK / WAYNE

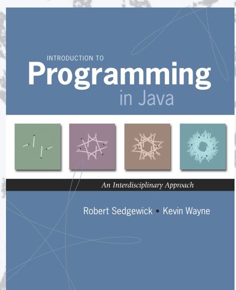


http://introcs.cs.princeton.edu

12. von Neumann machines

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

COMPUTER SCIENCE
SEDGEWICK / WAYNE



http://introcs.cs.princeton.edu

12. von Neumann Machines