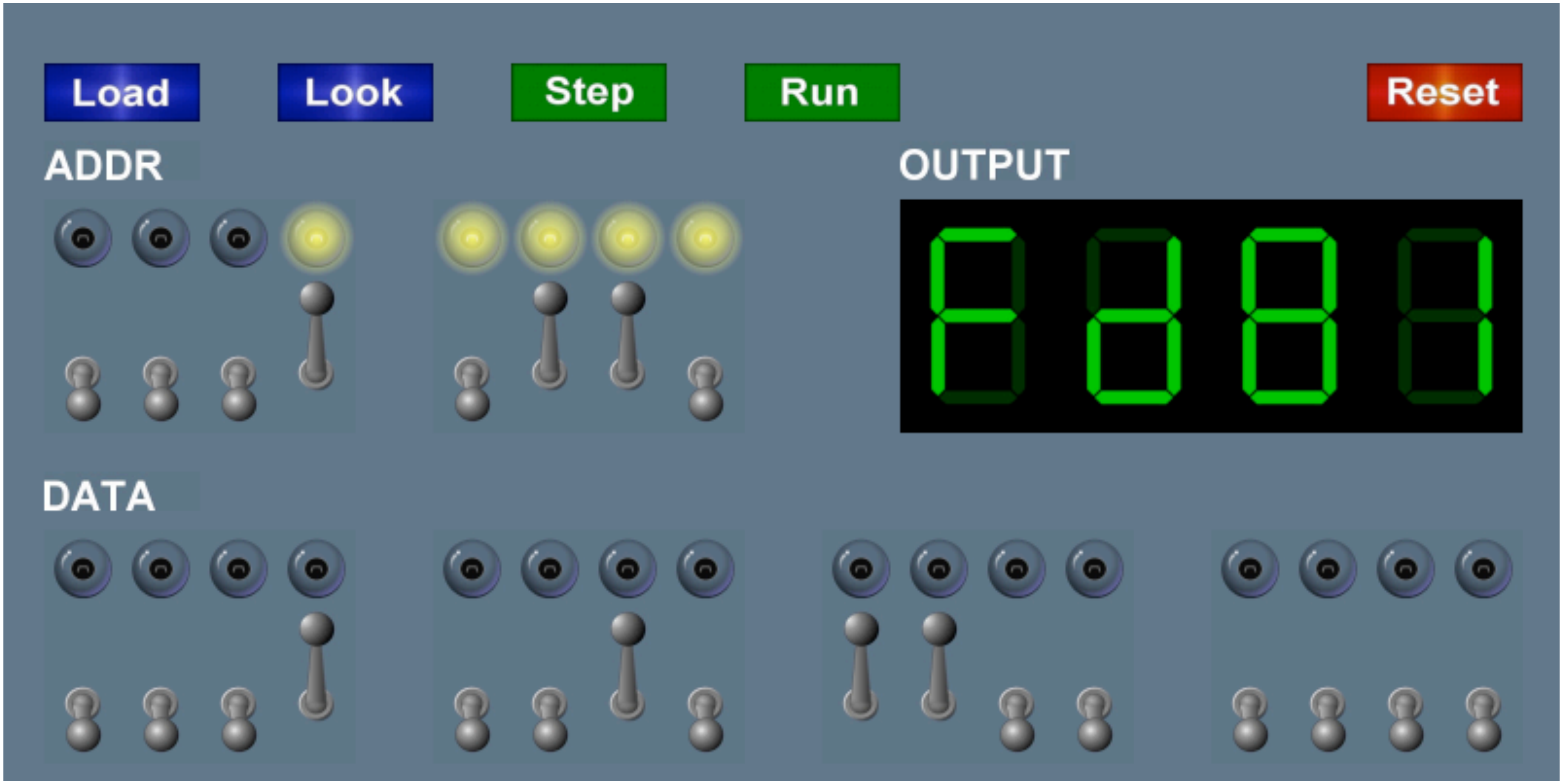
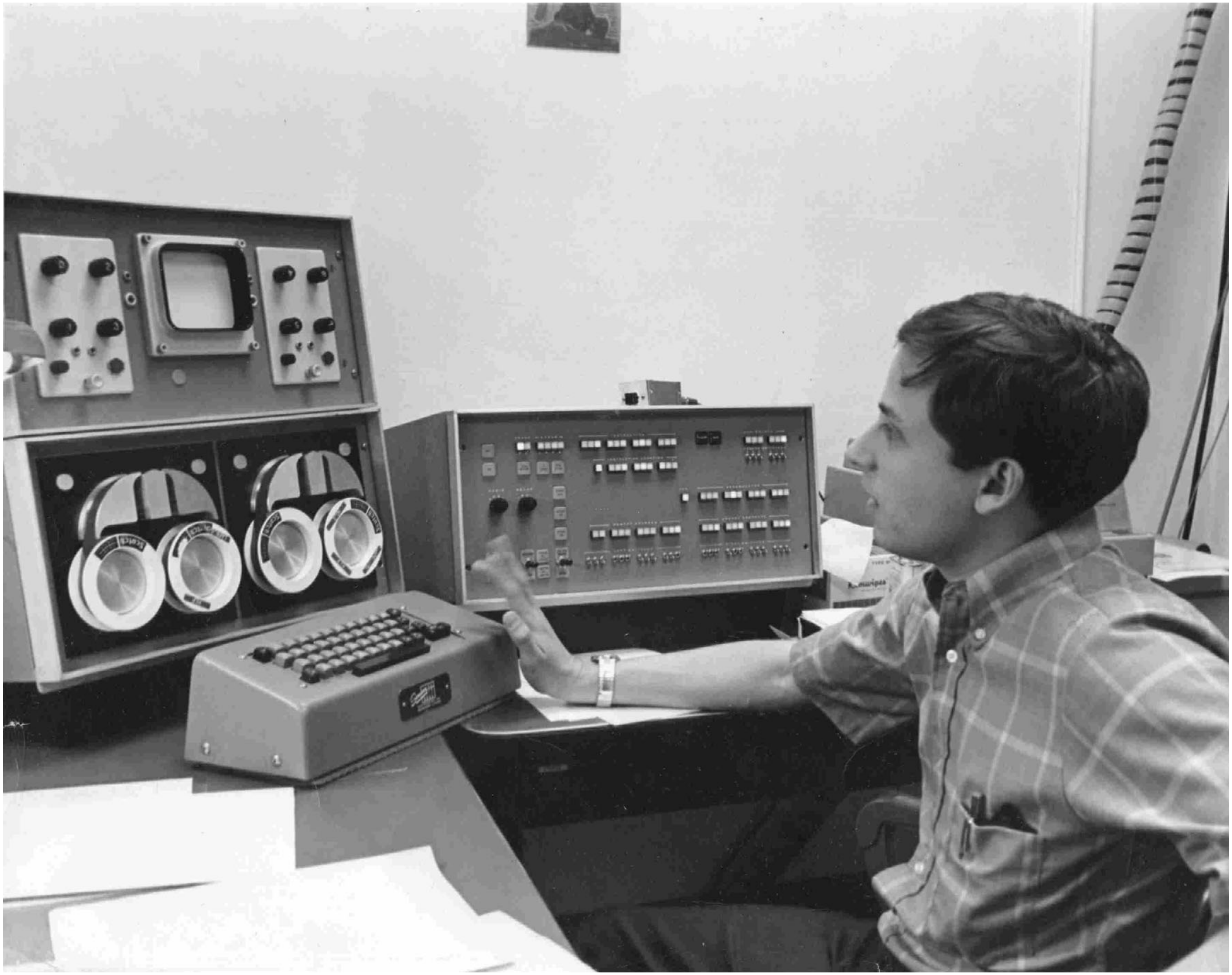

CLOSED
THROTTLE
OPEN

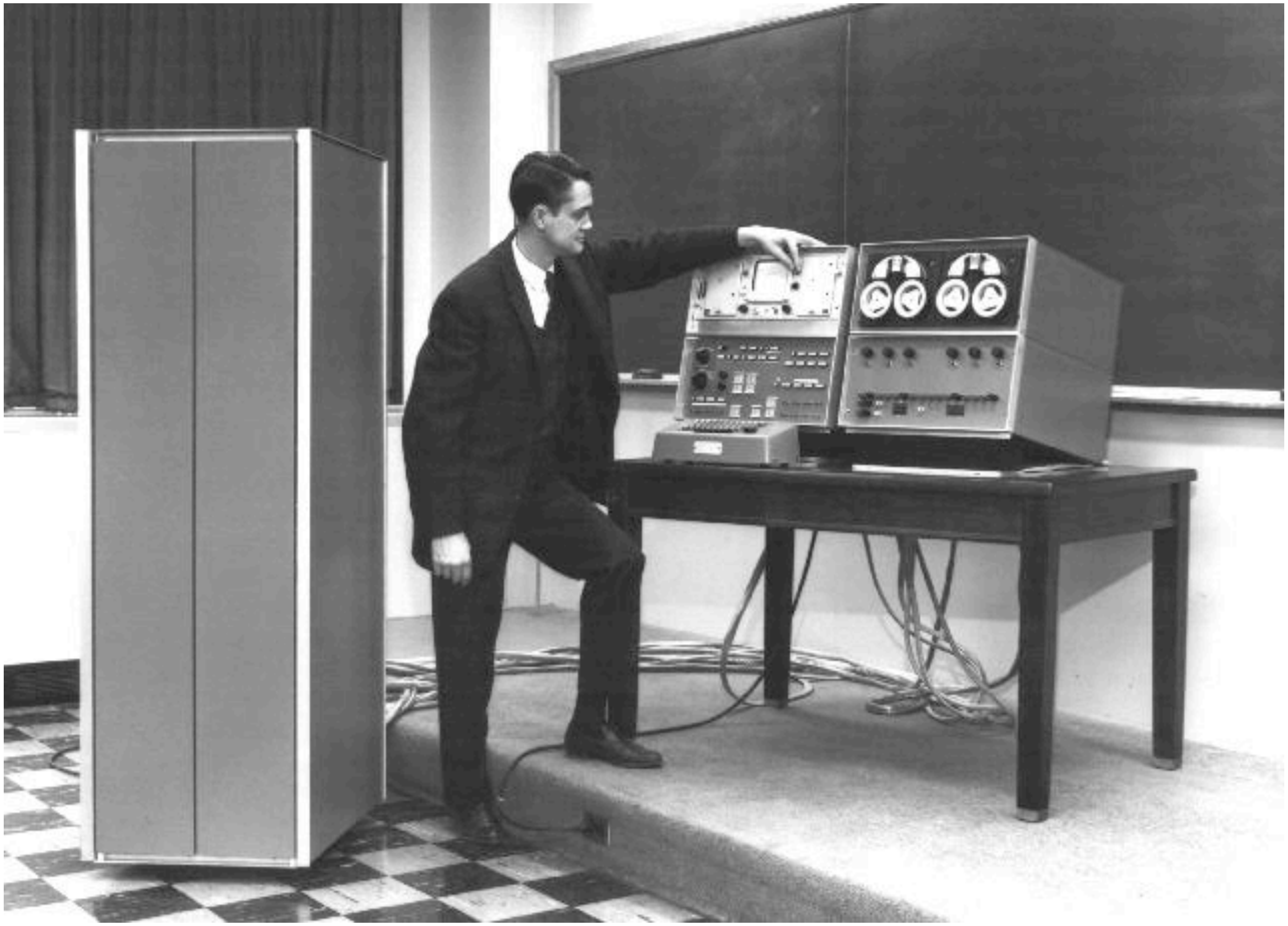
TOY II



LINC



LINC



What We've Learned About TOY

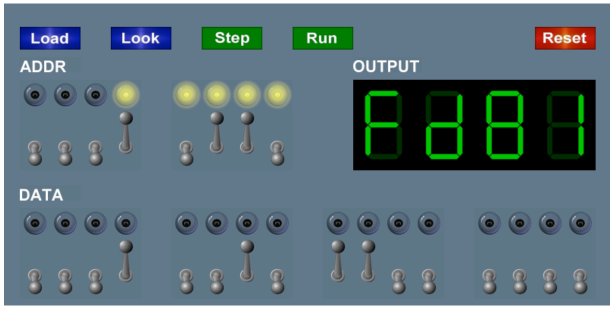
Data representation. Binary and hex.

TOY.

- Box with switches and lights.
- 16-bit memory locations, 16-bit registers, 8-bit pc.
- 4,328 bits = (255 × 16) + (15 × 16) + (8) = 541 bytes!
- von Neumann architecture.

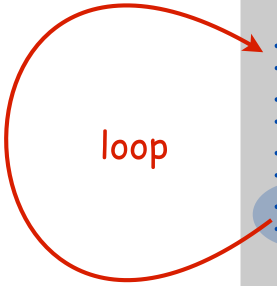
TOY instruction set architecture. 16 instruction types.

TOY machine language programs. Variables, arithmetic, loops.



Quick Review: Multiply

```
0A: 0003   3      ←   inputs
0B: 0009   9
0C: 0000   0      ←   output


0D: 0000   0      ←   constants
0E: 0001   1



10: 8A0A   RA ← mem[0A]                   a
11: 8B0B   RB ← mem[0B]                   b
12: 8C0D   RC ← mem[0D]                   c = 0

13: 810E   R1 ← mem[0E]                   always 1

14: CA18   if (RA == 0) pc ← 18           while (a != 0) {
15: 1CCB   RC ← RC + RB                      c = c + b
16: 2AA1   RA ← RA - R1                      a = a - 1
17: C014   pc ← 14                        }

18: 9C0C   mem[0C] ← RC
19: 0000   halt
```



multiply.toy

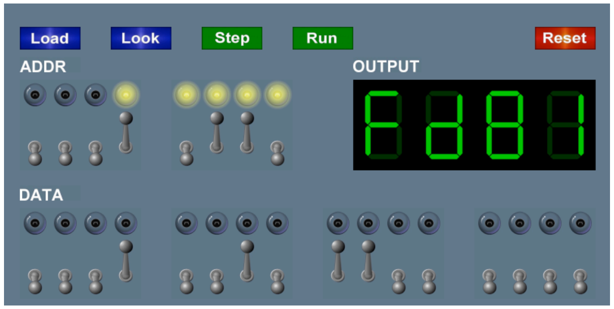
What We Do Today

Data representation. Negative numbers.

Input and output. Standard input, standard output.

Manipulate addresses. References (pointers) and arrays.

TOY simulator in Java and implications.



Data Representation

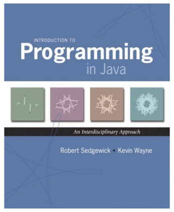
Digital World

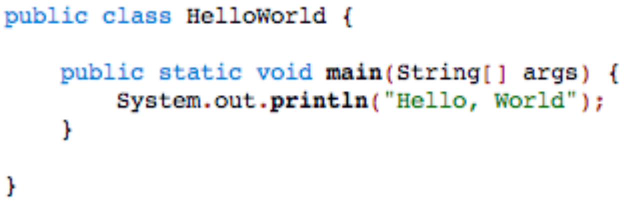
Data is a sequence of bits. (interpreted in different ways)

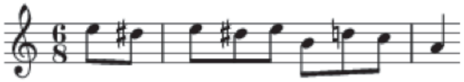
- Integers, real numbers, characters, strings, …
- Documents, pictures, sounds, movies, Java programs, …

Ex. `01110101`

- As binary integer: 1 + 4 + 16 + 32 + 64 = 117 (base ten).
- As character: 117th Unicode character = `'u'`.
- As music: 117/256 position of speaker.
- As grayscale value: 45.7% black.





```
public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello, World");
    }

}
```

Adding and Subtracting Binary Numbers

Decimal and binary addition.

```
                          carries

      1                         1   1
    013          0   0   0   0   1   1   0   1
 +  092       +  0   1   0   1   1   1   0   0
 ------       ----------------------------------
    105          0   1   1   0   1   0   0   1
```

Subtraction. Add a negative integer.

e.g., 6 - 4 = 6 + (-4))

Q. OK, but how to represent negative integers?

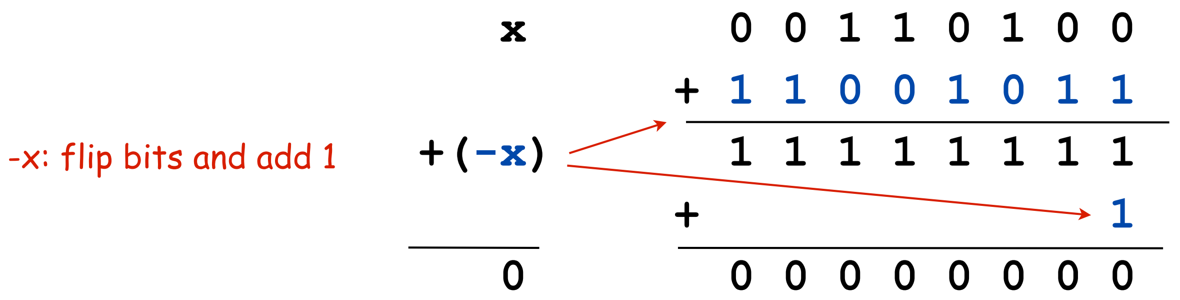
Representing Negative Integers

TOY words are 16 bits each.

- We could use 16 bits to represent 0 to $2^{16} - 1$.
- We want negative integers too.
- Reserving half the possible bit-patterns for negative seems fair.

Highly desirable property. If x is an integer, then the representation of -x, when added to x, yields zero.

$$\begin{array}{r} \mathbf{x} \\ +(-\mathbf{x}) \\ \hline 0 \end{array} \qquad \begin{array}{rcccccccc} & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ + & ? & ? & ? & ? & ? & ? & ? & ? \\ \hline & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$



-x: flip bits and add 1

$$\begin{array}{r} \mathbf{x} \\ \\ +(-\mathbf{x}) \\ \\ \hline 0 \end{array} \qquad \begin{array}{rcccccccc} & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ + & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ \hline & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ + & & & & & & & & 1 \\ \hline & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

“Two's Complement Integers

To compute -x from x:

- Start with x.

leading bit determines sign

+4	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

- Flip bits.

-5	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1

- Add one.

-4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0

Two's Complement Integers

		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
dec	hex	binary															
+32767	**7FFF**	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
									· · ·								
+4	**0004**	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
+3	**0003**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
+2	**0002**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
+1	**0001**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
+0	**0000**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-1	**FFFF**	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
-2	**FFFE**	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
-3	**FFFD**	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
-4	**FFFC**	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
									· · ·								
-32768	**8000**	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Properties of Two's Complement Integers

Properties.

- Leading bit (bit 15 in Toy) signifies sign.
- Addition and subtraction are easy.
- `0000000000000000` represents zero.
- Negative integer -x represented by 2^{16} - x.
- Not symmetric: can represent -32,768 but not 32,768.

Java. Java's `int` data type is a 32-bit two's complement integer.

Ex. `2147483647` + 1 equals `–2147483648`.

Representing Other Primitive Data Types in TOY

Bigger integers. Use two 16-bit words per `int`.

Real numbers.

- Use "floating point" (like scientific notation).
- Use four 16-bit words per `double`.

Characters.

- Use ASCII code (8 bits / character).
- Pack two characters per 16-bit word.

Note. Real microprocessors add hardware support for `int` and `double`.

Standard Input and Output

Standard Output

Standard output.

- Writing to memory location `FF` sends one word to TOY stdout.
- Ex. `9AFF` writes the integer in register `A` to stdout.

```
00: 0000    0
01: 0001    1

10: 8A00    RA ← mem[00]              a = 0
11: 8B01    RB ← mem[01]              b = 1
                                      do {
12: 9AFF    write RA to stdout           print a
13: 1AAB    RA ← RA + RB                 a = a + b
14: 2BAB    RB ← RA - RB                 b = a - b
15: DA12    if (RA > 0) goto 12        } while (a > 0)
16: 0000    halt
```

Standard Output

Standard output.

- Writing to memory location `FF` sends one word to TOY stdout.
- Ex. `9AFF` writes the integer in register `A` to stdout.

```
00: 0000    0
01: 0001    1

10: 8A00    RA ← mem[00]          a = 0
11: 8B01    RB ← mem[01]          b = 1
                                  do {
12: 9AFF    write RA to stdout       print a
13: 1AAB    RA ← RA + RB             a = a + b
14: 2BAB    RB ← RA - RB             b = a - b
15: DA12    if (RA > 0) goto 12   } while (a > 0)
16: 0000    halt
```

`fibonacci.toy`

standard output

```
0000
0001
0001
0002
0003
0005
0008
000D
0015
0022
0037
0059
0090
00E9
0179
0262
03DB
063D
0A18
1055
1A6D
2AC2
452F
6FF1
```

Standard Input

Standard input.

- Loading from memory address `FF` loads one word from TOY stdin.
- Ex. `8AFF` reads an integer from stdin and store it in register `A`.

Ex: read in a sequence of integers and print their sum.

- In Java, stop reading when EOF.
- In TOY, stop reading when user enters `0000`.

```
while (!StdIn.isEmpty()) {
   a = StdIn.readInt();
   sum = sum + a;
}
StdOut.println(sum);
```

```
00: 0000   0


10: 8C00   RC <- mem[00]
11: 8AFF   read RA from stdin
12: CA15   if (RA == 0) pc ← 15
13: 1CCA   RC ← RC + RA
14: C011   pc ← 11
15: 9CFF   write RC
16: 0000   halt
```

```
00AE
0046
0003
0000
00F7
```

Standard Input and Output: Implications

Standard input and output enable you to:

- Put information from real world into machine.
- Get information out of machine.
- Process more information than fits in memory.
- Interact with the computer while it is running.

Information can be instructions!

- Booting a computer.
- Sending programs over the Internet
- Sending viruses over the Internet

Pointers

Load Address (a.k.a. Load Constant)

Load address. [opcode 7]

- Loads an 8-bit integer into a register.
- `7A30` means load the value `30` into register `A`.

Applications.

- Load a small constant into a register.
- Load an 8-bit memory address into a register.

register stores "pointer" to a memory cell

```
a = 0x30;
```

Java code

(NOTE hex literal)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	1	0	0	0	1	1	0	0	0	0
7_{16}				A_{16}				3_{16}				0_{16}			
opcode				dest d				addr							

Arrays in TOY

TOY main memory is a giant array.

- Can access memory cell `30` using load and store.
- `8C30` means load `mem[30]` into register `C`.
- Goal: access memory cell `i` where `i` is a variable.

…	…
30	0000
31	0001
32	0001
33	0002
34	0003
35	0005
36	0008
37	000D
…	…

TOY memory

Load indirect. [opcode A]

a variable index

- `AC06` means load `mem[R6]` into register `C`.

Store indirect. [opcode B]

a variable index

- `BC06` means store contents of register `C` into `mem[R6]`.

Example: Reverse an array

TOY implementation of reverse.

- Read in a sequence of integers and store in memory 30, 31, 32, …
- Stop reading if 0000.
- Print sequence in reverse order.

Java version:

```
int n = 0;
while (!StdIn.isEmpty())
{
   a[n] = StdIn.readInt();
   n++;
}


n--;
while (n >= 0)
{
   StdOut.println(a[n]);
   n--;
}
```

(We'll just assume a[] is big enough)

TOY Implementation of Reverse

TOY implementation of reverse.

- Read in a sequence of integers and store in memory `30`, `31`, `32`, …
- Stop reading if `0000`.
- Print sequence in reverse order.

```
10: 7101  R1 ← 0001                  constant 1
11: 7A30  RA ← 0030                  a[]
12: 7B00  RB ← 0000                  n



                                     while(true) {
13: 8CFF  read RC                       c = StdIn.readInt();
14: CC19  if (RC == 0) goto 19          if (c == 0) break;
15: 16AB  R6 ← RA + RB                  memory address of a[n]
16: BC06  mem[R6] ← RC                  a[n] = c;
17: 1BB1  RB ← RB + R1                  n++;
18: C013  goto 13                    }
```

read in the data

TOY Implementation of Reverse

TOY implementation of reverse.

- Read in a sequence of integers and store in memory `30`, `31`, `32`, …
- Stop reading if `0000`.
- → Print sequence in reverse order.

```
10: 7101   R1 ← 0001                   constant 1
11: 7A30   RA ← 0030                   a[]
12: 7B00   RB ← 0000                   n



                                       while(true) {
19: CB20   if (RB == 0) goto 20           if (n == 0) break;
1A: 16AB   R6 ← RA + RB                   memory address of a[n]
1B: 2661   R6 ← R6 - R1                   n--;
1C: AC06   RC ← mem[R6]                   c = a[n];
1D: 9CFF   write RC                       StdOut.print(c);
1E: 2BB1   RB ← RB - R1                   n--;
1F: C019   goto 19                     }
```

print in reverse order

Unsafe Code at any Speed

Q. What happens if we make array start at `00` instead of `30`?

```
10: 7101   R1 ← 0001                  constant 1
11: 7A00   RA ← 0000                  a[]
12: 7B00   RB ← 0000                  n

                                      while(true) {
13: 8CFF   read RC                      c = StdIn.readInt();
14: CC19   if (RC == 0) goto 19         if (c == 0) break;
15: 16AB   R6 ← RA + RB                 address of a[n]
16: BC06   mem[R6] ← RC                 a[n] = c;
17: 1BB1   RB ← RB + R1                 n++;
18: C013   goto 13                    }
```

```
% more crazy8.txt
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
8888 8810
98FF C011
```

Unsafe Code at any Speed

```
00: 0000        08: 0000
01: 0000        09: 0000
02: 0000        0A: 0000
03: 0000        0B: 0000
04: 0000        0C: 0000
05: 0000        0D: 0000
06: 0000        0E: 0000
07: 0000        0F: 0000
```

standard input

```
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
8888 8810
98FF C011
```

```
10: 7101   R1 ← 0001                      constant 1
11: 7A00   RA ← 0000                      a[]
12: 7B00   RB ← 0000                      n
                                          while(true) {
13: 8CFF   read RC                            c = StdIn.readInt();
14: CC19   if (RC == 0) goto 19               if (c == 0) break;
15: 16AB   R6 ← RA + RB                       address of a[n]
16: BC06   mem[R6] ← RC                       a[n] = c;
17: 1BB1   RB ← RB + R1                      n++;
18: C013   goto 13                        }
```

Unsafe Code at any Speed

standard input

```
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
8888 8810
98FF C011
```

```
00: 0001         08: 0001
01: 0001         09: 0001
02: 0001         0A: 0001
03: 0001         0B: 0001
04: 0001         0C: 0001
05: 0001         0D: 0001
06: 0001         0E: 0001
07: 0001         0F: 0001
```

Machine is Owned

```
10: 8888
11: 8810   R8 ← mem[10]                  while (true) { x = 8888;
12: 98FF   write R8                        writeln(x);

13: C011   goto 11                       }
14: CC19   if (RC == 0) goto 19              if (c == 0) break;
15: 16AB   R6 ← RA + RB                      address of a[n]
16: BC06   mem[R6] ← RC                      a[n] = c;
17: 1BB1   RB ← RB + R1                     n++;
18: C013   goto 13                       }
```

What Can Happen When We Lose Control (in C or C++)?

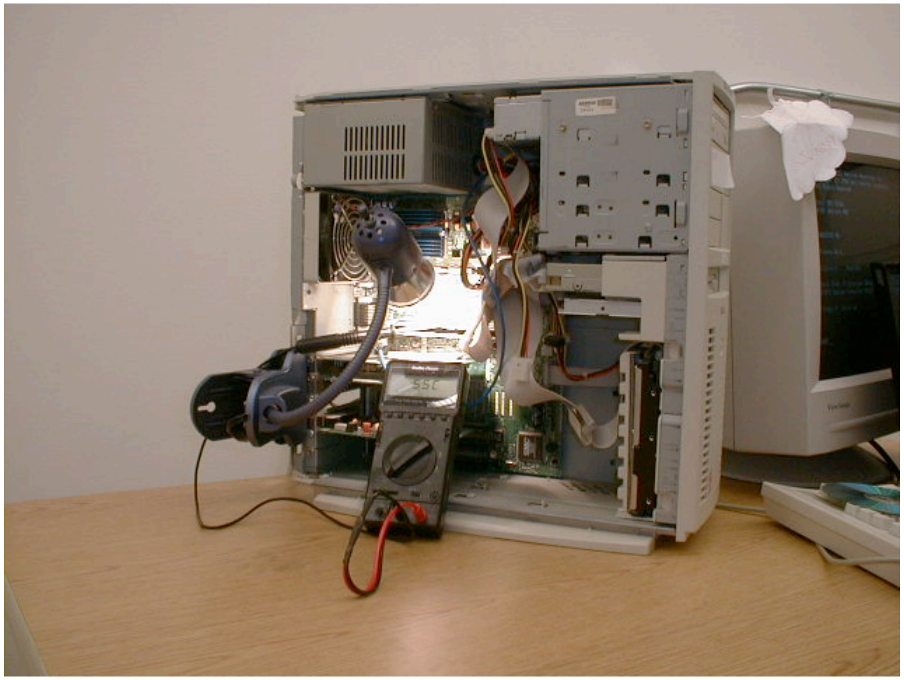
Buffer overflow.

- Array `buffer[]` has size 100.
- User might enter 200 characters.
- Might lose control of machine behavior.

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

unsafe C program

Consequences. Viruses and worms.



shine 50W bulb at DRAM
[Appel-Govindavajhala '03]

Java enforces security.

- Type safety.
- Array bounds checking.
- Not foolproof.

Buffer Overflow Attacks

Stuxnet worm. [July 2010]

- Step 1. Natanz centrifuge fuel-refining plant employee plugs in USB flash drive.
- Step 2. Data becomes code by exploiting Window buffer overflow; machine is Owned.
- Step 3. Uranium enrichment in Iran stalled.

More buffer overflow attacks: Morris worm, Code Red, SQL Slammer, iPhone unlocking, Xbox softmod, JPEG of death [2004], . . .

Lesson.

- Not easy to write error-free software.
- Embrace Java security features.
- Keep your OS patched.

Dumping

Q. Work all day to develop operating system. How to save it?

A. Write short program `dump.toy` and run it to dump contents of memory onto tape.

```
00: 7001    R1 ← 0001
01: 7210    R2 ← 0010                      i = 10
02: 73FF    R3 ← 00FF
                                           do {
03: AA02    RA ← mem[R2]                      a = mem[i]
04: 9AFF    write RA                          print a
05: 1221    R2 ← R2 + R1                      i++
06: 2432    R4 ← R3 - R2
07: D403    if (R4 > 0) goto 03            } while (i < 255)
08: 0000    halt
```

dump.toy

Booting

Q. How do you get it back?

A. Write short program `boot.toy` and run it to read contents of memory from tape.

```
00: 7001   R1 ← 0001
01: 7210   R2 ← 0010                      i = 10
02: 73FF   R3 ← 00FF
                                          do {
03: 8AFF   read RA                           read a
04: BA02   mem[R2] ← RA                      mem[i] = a
05: 1221   R2 ← R2 + R1                      i++
06: 2432   R4 ← R3 - R2
07: D403   if (R4 > 0) goto 03            } while (i < 255)
08: 0000   halt
```

`boot.toy`

Simulating the TOY machine

TOY Simulator

Goal. Write a program to "simulate" the behavior of the TOY machine.

- TOY simulator in Java.

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

      // READ .toy FILE into mem[]

      while (true)
      {
         int inst = mem[pc++];   // fetch, increment
         // DECODE
         // EXECUTE
      }
   }
}
```

```
% more add-stdin.toy
10: 8C00
11: 8AFF
12: CA15
13: 1CCA
14: C011
15: 9CFF
16: 0000

% java TOY add-stdin.toy
00AE
0046
0003
0000
00F7
```

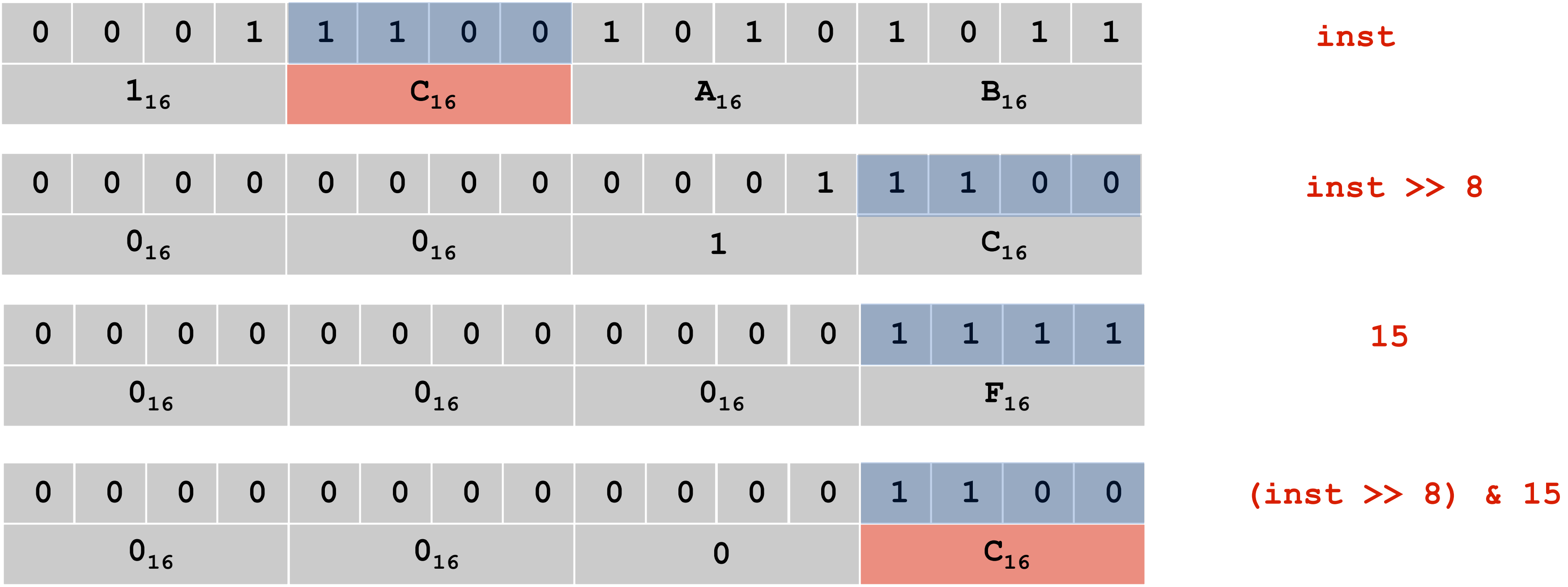
← TOY program

← standard input

← standard output

TOY Simulator: Decode

Ex. Extract destination register of `1CAB` by shifting and masking.



inst

0	0	0	1	1	1	0	0	1	0	1	0	1	0	1	1
1_{16}				C_{16}				A_{16}				B_{16}			

inst >> 8

0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0
0_{16}				0_{16}				1				C_{16}			

15

0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
0_{16}				0_{16}				0_{16}				F_{16}			

(inst >> 8) & 15

0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
0_{16}				0_{16}				0				C_{16}			

```
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)
```

TOY Simulator: 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: Omitted Details

Omitted details.

- Register 0 is always 0.
  - reset `R[0]=0` after each fetch-execute step

- Standard input and output.
  - if `addr` is `FF` and opcode is load (indirect) then read in data
  - if `addr` is `FF` and opcode is store (indirect) then write out data

- TOY registers are 16-bit integers; program counter is 8-bit.
  - Java `int` is 32-bit; Java `short` is 16-bit
  - use casts and bit-whacking

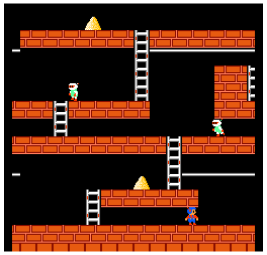
Complete implementation. See `TOY.java` on booksite.

Simulation

Building a new computer? Need a plan for old software.

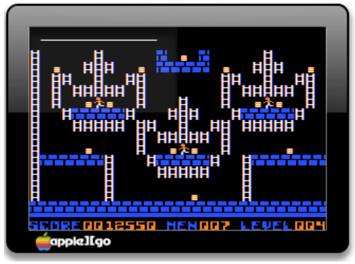
Two possible approaches

- Rewrite software (costly, error-prone, boring, and time-consuming).
- Simulate old computer on new computer.



Lode Runner

Apple IIe



Mac OS X Apple IIe emulator widget running Lode Runner

Ancient programs still running on modern computers.

- Payroll
- Power plants
- Air traffic control
- Ticketron.
- Games.