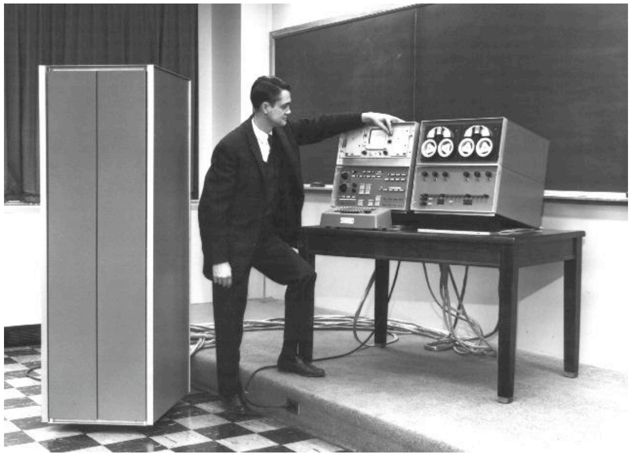
TOY II



What We've Learned About TOY

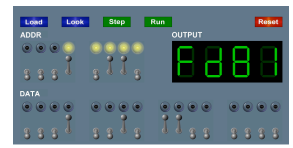
Data representation. Binary and hex.

TOY.

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

TOY instruction set architecture. 16 instruction types.

TOY machine language programs. Variables, arithmetic, loops.



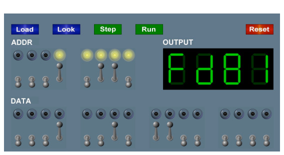
What We Do Today

Data representation. Negative numbers.

Input and output. Standard input, standard output.

Manipulate addresses. References (pointers) and arrays.

TOY simulator in Java.



Negative Numbers

How to add and subtract binary numbers

Binary addition facts:

- 0 + 0 = 0
- 0 + 1 = 1 + 0 = 1
- 1 + 1 = 10
- 1 + 1 + 1 = 11 (needed for carries)

Bigger numbers example:

```
        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
```

OK, but: subtract?

- Subtract by adding a negative integer (e.g., 6 - 4 = 6 + (-4))
- OK, but: 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, is zero.

```
     x       0 0 1 1 0 1 0 0
 +(-x)     + ? ? ? ? ? ? ? ?
     0       0 0 0 0 0 0 0 0
```

```
     x       0 0 1 1 0 1 0 0
           + 1 1 0 0 1 0 1 1
 +(-x)       1 1 1 1 1 1 1 1
           +               1
     0       0 0 0 0 0 0 0 0
```

flip bits and add 1

"Two's Complement" Integers

Properties:

- Leading bit (bit 15) signifies sign.
- Negative integer -N represented by 2^{16} - N.
- Trick to compute -N:

1. Start with N.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
+4	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

2. Flip bits.

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

3. Add 1.

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

Two's Complement Integers

dec	hex	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		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

Nice properties:

- 0000000000000000 represents 0.
- -0 and +0 are the same.
- Addition is easy (see next slide).
- Checking for arithmetic overflow is easy.

Not-so-nice properties.

- Can represent one more negative integer than positive integer. (-32,768 = -2^{15} but not 32,768 = 2^{15}).

Two's Complement Arithmetic

Addition is carried out as if all integers were positive.

- It usually works.

-3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
+																
4	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
=																
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Two's Complement Arithmetic

Addition is carried out as if all integers were positive.

- It usually works.
- But overflow can occur:
  - carry into sign (left most) bit with no carry out

+32,767	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
+																
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
=																
-32,767	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Representing Other Primitive Data Types

Negative integers. TOY uses two's complement integers.

Big integers.

- Can use "multiple precision."
- Use two 16-bit words per integer.

Real numbers.

- Can use "floating point" (like scientific notation).
- Double word for extra precision.

Characters.

- Can use ASCII code (8 bits / character).
- Can pack two characters into one 16-bit word.

Standard Input and 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 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 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 a 8-bit memory address into a register.

register stores "pointer" to a memory cell

```
a = 0x30;
```

Java code

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.

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]`.

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

TOY memory

```
for (int i = 0; i < N; i++)
   a[i] = StdIn.readInt();

for (int i = 0; i < N; i++)
   StdOut.println(a[N-i-1]);
```


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.

```
19: CB20   if (RB == 0) goto 20        while (n > 0) {
1A: 16AB   R6 ← RA + RB                   address of a[n]
1B: 2661   R6 ← R6 – R1                   address of a[n-1]
1C: AC06   RC ← mem[R6]                   c = a[n-1];
1D: 9CFF   write RC                       StdOut.println(c);
1E: 2BB1   RB ← RB – R1                   n--;
1F: C019   goto 19                     }
20: 0000   halt
```

print in reverse order

Unsafe Code at Any Speed

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

A. Self modifying program; can overflow buffer and run arbitrary code!

```
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
```


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

Buffer overrun.

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

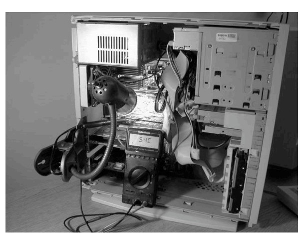
```
#include <stdio.h>
int main(void) {
   char buffer[100];
   scanf("%s", buffer);
   printf("%s\n", buffer);
   return 0;
}
```

unsafe C program

Consequences. Viruses and worms.

Java enforces security.

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



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

Buffer Overrun Example: JPEG of Death

Microsoft Windows JPEG bug. [September, 2004]

- Step 1. User views malicious JPEG in IE or Outlook.
- Step 2. Machine is Owned.
- Data becomes code by exploiting buffer overrun in GDI+ library.

Fix. Update old library with patched one.

↖ but many applications install independent copies of GDI library

Moral.

- Not easy to write error-free software.
- Embrace Java security features.
- Don't try to maintain several copies of the same file.
- 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: 7101   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: 7101   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

TOY Simulator

TOY Simulator

Goal. Write a program to "simulate" the behavior of the TOY machine.
- TOY simulator in Java.

```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 IN .toy FILE

      while (true) {
         // FETCH INSTRUCTION and DECODE
         ...
         // EXECUTE
         ...
      }
   }
}
```

```
% java TOY add-stdin.toy
A012
002B
A03D
```

TOY Simulator: Fetch

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

0 0 0 1	1 1 0 0	1 0 1 0	1 0 1 1	inst
1_{16}	C_{16}	A_{16}	B_{16}	
0 0 0 0	0 0 0 0	0 0 0 1	1 1 0 0	inst >> 8
0_{16}	0_{16}	1	C_{16}	
0 0 0 0	0 0 0 0	0 0 0 0	1 1 1 1	15
0_{16}	0_{16}	0_{16}	F_{16}	
0 0 0 0	0 0 0 0	0 0 0 0	1 1 0 0	(inst >> 8) & 15
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

Consequences of simulation.

- Test out new machine or microprocessor using simulator.
  - cheaper and faster than building actual machine
- Easy to add new functionality to simulator.
  - trace, single-step, breakpoint debugging
  - simulator more useful than TOY itself
- Reuse software from old machines.

Ancient programs still running on modern computers.

- Ticketron.
- Lode Runner on Apple IIe.