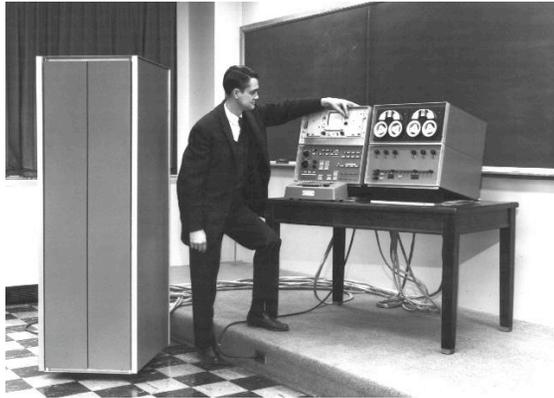


TOY II



LINC

Data representation.

- Binary and hex.

TOY: what's in it, how to use it.

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

TOY instruction set architecture.

- 16 instruction types.

Sample TOY machine language programs.

- Arithmetic.
- Loops.

What We Do Today

Binary add, subtract.

Standard input, standard output.

Manipulate addresses.

- References (pointers).
- Arrays.

TOY simulator in Java.

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:

$$\begin{array}{r}
 1 \\
 013 \\
 + 092 \\
 \hline
 105
 \end{array}
 \qquad
 \begin{array}{r}
 \text{carries} \rightarrow 1 \ 1 \\
 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \\
 + 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \\
 \hline
 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1
 \end{array}$$

OK, but: subtract?

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

How to Represent Negative Integers

TOY words are 16 bits each.

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

Highly desirable property:

- If X is a positive integer, then the representation of -X, when added to X, had better yield zero.

$$\begin{array}{r}
 X \quad \quad \quad 0\ 0\ 1\ 1\ 0\ 1\ 0\ 0 \\
 +(-X) \quad \quad +\ ?\ ?\ ?\ ?\ ?\ ?\ ?\ ? \\
 \hline
 0 \quad \quad \quad 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0
 \end{array}$$

$$\begin{array}{r}
 X \quad \quad \quad 0\ 0\ 1\ 1\ 0\ 1\ 0\ 0 \\
 +(-X) \quad \quad +\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 1 \\
 \hline
 0 \quad \quad \quad 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0
 \end{array}$$

"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
----	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

6

7

Two's Complement Integers

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

8

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.

$$-N = \sim N + 1$$

Not-so-nice properties.

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

9

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

10

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	1	1	1	1	
+																					
2	0	0	0	0	0	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	0	0	0	0	0	1

11

Standard Output

Standard output.

- Writing to memory location `FF` sends one word to TOY stdout.
- `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

12: 9AFF  print RA           while(a > 0) {
13: 1AAB  RA <- RA + RB         print a
14: 2BAB  RB <- RA - RB         a = a + b
15: DA12  if (RA > 0) goto 12    b = a - b
16: 0000  halt
    
```

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

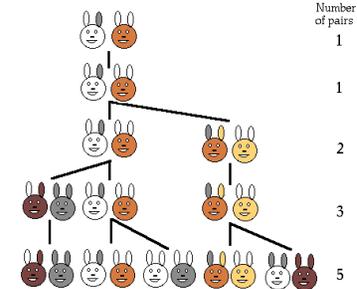
12

Fibonacci Numbers

Infinite series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

Fibonacci Rabbits:



L. P. Fibonacci (1170 - 1250)

13

Standard Input

Standard input.

- Loading from memory address FF loads one word from TOY stdin.
- 8AFF reads in an integer from stdin and stores 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;
}
System.out.println(sum);
```

00:	0000	0
10:	8C00	RC ← mem[00]
11:	8AFF	read RA
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

14

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

15

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 = 30;

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							

16

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].

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

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

Reverse.java

17

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
11: 7A30 RA ← 0030
12: 7B00 RB ← 0000

13: 8CFF read RC
14: CC19 if (RC == 0) goto 19
15: 16AB R6 ← RA + RB
16: BC06 mem[R6] ← RC
17: 1BB1 RB ← RB + R1
18: C013 goto 13
    
```

constant 1
a[]
n

```

while(true) {
    c = StdIn.readInt();
    if (c == 0) break;
    address of a[n]
    a[n] = c;
    n++;
}
    
```

read in the data

18

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
1A: 16AB R6 ← RA + RB
1B: 2661 R6 ← R6 - R1
1C: AC06 RC ← mem[R6]
1D: 9CFF write RC
1E: 2BB1 RB ← RB - R1
1F: C019 goto 19
20: 0000 halt
    
```

```

while (n > 0) {
    address of a[n]
    address of a[n-1]
    c = a[n-1];
    System.out.println(c);
    n--;
}
    
```

19

Unsafe Code at any Speed

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

- Self modifying program.
- Exploit buffer overflow and run arbitrary code!

```

10: 7101 R1 ← 0001
11: 7A00 RA ← 0000
12: 7B00 RB ← 0000

13: 8CFF read RC
14: CC19 if (RC == 0) goto 19
15: 16AB R6 ← RA + RB
16: BC06 mem[R6] ← RC
17: 1BB1 RB ← RB + R1
18: C013 goto 13
    
```

constant 1
a[]
n

```

while(true) {
    c = StdIn.readInt();
    if (c == 0) break;
    address of a[n]
    a[n] = c;
    n++;
}
    
```

Crazy 8s Input

```

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

20

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.
- Majority of viruses and worms caused by similar errors.

```

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

unsafe C program

Robert Morris Internet Worm.

- Cornell grad student injected worm into Internet in 1988.
- Exploited buffer overrun in finger daemon `fingerd`.

Java enforces security.

- Type safety.
- Array bounds checking.
- Not foolproof. [Appel '03: shine 50W bulb at DRAM.]

21

Buffer Overrun Example: JPEG of Death

Microsoft Windows JPEG bug. [September, 2004]

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



Fix.

- Update old library with patched one.
- Caveat: many applications install independent copies of GDI library.

Moral.

- Not easy to write error-free software.
- Be grateful for security features built in to Java.
- Don't try to maintain several copies of the same file.
- Keep your operating system patched.

Dumping

Dumping.

- Work all day to develop operating system.
- How do you save it for tomorrow?
 - leave computer on?
 - write short program `dump.toy`
 - run `dump.toy` to dump contents of memory onto tape

```
00: 7001  R1 ← 0001
01: 7210  R2 ← 0010          i = 10
02: 73FF  R3 ← 00FF

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

`dump.toy`

22

23

Booting



Booting.

- How do you get it back?
 - turn on computer, old memory values gone
 - write short program `boot.toy`
 - read contents of memory from tape by running `boot.toy`
 - use original program

```
00: 7001  R1 ← 0001
01: 7210  R2 ← 0010          i = 10
02: 73FF  R3 ← 00FF

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

`boot.toy`

24

TOY Simulator

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

- ➔ ▪ TOY simulator in Java.
- TOY simulator in TOY!

```
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

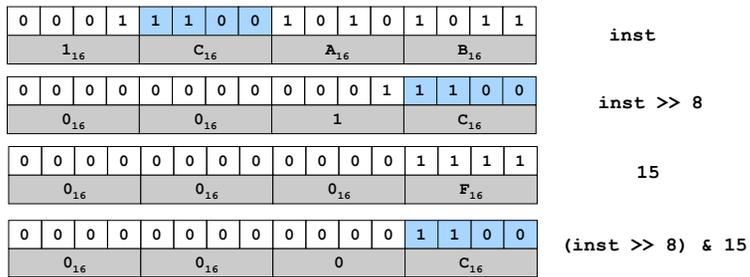
        // READ IN .toy FILE

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

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

25

Extract destination register of $1CAB$ by shifting and masking.



```

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;
}
    
```

TOY Simulator: Missing Details

Omitted details.

- Register 0 is always 0.
 - reset to 0000 after each fetch-execute step
- Standard input and output.
 - if addr is FF and opcode is load then read in data
 - if addr is FF and opcode is store then write out data
 - (load and store indirect have a similar problem)
- TOY registers are 16-bit integers; program counter is 8-bit.
 - Java int is 32 bits

See `TOY.java` for full details.

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.

