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.

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.
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:

<table>
<thead>
<tr>
<th>Binary</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1013</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>092</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

Properties:
- Leading bit (bit 15) signifies sign.
- Negative integer $-N$ represented by $2^{16} - N$.
- Trick to compute $-N$:
  1. Start with $N$.
  2. Flip bits.
  3. Add 1.

<table>
<thead>
<tr>
<th>dec</th>
<th>hex</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>+4</td>
<td>0004</td>
<td>0000000000000100</td>
</tr>
<tr>
<td>+3</td>
<td>0003</td>
<td>0000000000000111</td>
</tr>
<tr>
<td>+2</td>
<td>0002</td>
<td>0000000000000101</td>
</tr>
<tr>
<td>+1</td>
<td>0001</td>
<td>0000000000000001</td>
</tr>
<tr>
<td>+0</td>
<td>0000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>-1</td>
<td>FFFF</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>-2</td>
<td>FFFE</td>
<td>1111111111111110</td>
</tr>
<tr>
<td>-3</td>
<td>FFFD</td>
<td>1111111111111110</td>
</tr>
<tr>
<td>-4</td>
<td>FFFC</td>
<td>1111111111111110</td>
</tr>
</tbody>
</table>

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
     0 0 0 0 0 0 0 0
```

```
x   0 0 1 1 0 1 0 0
+(-x) + 1 1 0 0 1 0 1 1
     +    1 1 1 1 1 1 1 1
flip bits and add 1
     0 0 0 0 0 0 0 0
```

"Two's Complement" Integers

<table>
<thead>
<tr>
<th>dec</th>
<th>hex</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>+32767</td>
<td>7FFF</td>
<td>0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dec</th>
<th>hex</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>+4</td>
<td>0004</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>+3</td>
<td>0003</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>+2</td>
<td>0002</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>+1</td>
<td>0001</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>+0</td>
<td>0000</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>-1</td>
<td>FFFF</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>-2</td>
<td>FFFE</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0</td>
</tr>
<tr>
<td>-3</td>
<td>FFFD</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0</td>
</tr>
<tr>
<td>-4</td>
<td>FFFC</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0</td>
</tr>
<tr>
<td>-32768</td>
<td>8000</td>
<td>1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>
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.
- But overflow can occur:
  - carry into sign (left most) bit with no carry out

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

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);

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.

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 a 8-bit memory address into a register.

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

Store indirect. [opcode B]  
- 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++)
    StdOut.println(a[N-i-1]);

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.
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
```

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
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
```

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

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 0wned.
  - Data becomes code by exploiting buffer overrun in GDI+ library.

Fix. Update old library with patched one.

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.
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.

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>7101</td>
<td>R1 ← 0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>7210</td>
<td>R2 ← 0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02</td>
<td>73FF</td>
<td>R3 ← 00FF</td>
<td></td>
<td>i = 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>03</td>
<td>AA02</td>
<td>RA ← mem[R2]</td>
<td>a = mem[i]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>9AFF</td>
<td>write RA</td>
<td>print a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>1221</td>
<td>R2 ← R2 + R1</td>
<td>i++</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>2432</td>
<td>R4 ← R3 - R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>D403</td>
<td>if (R4 &gt; 0) goto 03</td>
<td></td>
<td>while (i &lt; 255)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>0000</td>
<td>halt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`dump.toy`

Q. How do you get it back?
A. Write short program `boot.toy` and run it to read contents of memory from tape.

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>7101</td>
<td>R1 ← 0001</td>
<td></td>
<td>i = 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>7210</td>
<td>R2 ← 0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02</td>
<td>73FF</td>
<td>R3 ← 00FF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>03</td>
<td>8AFF</td>
<td>read RA</td>
<td></td>
<td>read a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>BA02</td>
<td>mem[R2] ← RA</td>
<td>mem[i] = a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>1221</td>
<td>R2 ← R2 + R1</td>
<td>i++</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>2432</td>
<td>R4 ← R3 - R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>D403</td>
<td>if (R4 &gt; 0) goto 03</td>
<td></td>
<td>while (i &lt; 255)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>0000</td>
<td>halt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`boot.toy`

**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
            ...  // FETCH
            // EXECUTE
            ...  // EXECUTE
        }
    }
}
```

% java TOY add-stdin.toy
A012
002B
A03D
TOY Simulator: Fetch

**Fetch.** Extract destination register of \texttt{1CAB} by shifting and masking.

\[
\begin{array}{|cccccccccccc|}
\hline
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
1_{16} & C_{16} & A_{16} & B_{16} \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
0_{16} & 0_{16} & 1 & C_{16} \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0_{16} & 0_{16} & 0_{15} & F_{16} \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0_{16} & 0_{16} & 0 & C_{16} \\
\hline
\end{array}
\]

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

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 \(\text{addr}\) is FF and opcode is load (indirect) then read in data
  - if \(\text{addr}\) is FF and opcode is store (indirect) then write out data

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

**Complete implementation.** See \texttt{TOY.java} on booksite.

TOY Simulator: Execute

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

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.