What We've Learned About TOY

Data representation.
- Binary and hex.

TOY: what’s in it, how to use it.
- Box with switches and lights.
- 4,328 bits = (255 × 16) + (15 × 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:

```
  1011101
+ 0100110
  1010111
```

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.

\[
\begin{array}{c}
X \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \\
(-X) \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \\
\hline
0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0
\end{array}
\]

“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 \(N\).
  2. Flip bits.
  3. Add 1.

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>0000000000000001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>0000000000000010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>0000000000000011</td>
</tr>
<tr>
<td>+4</td>
<td>0100</td>
<td>0000000000000100</td>
</tr>
<tr>
<td>+3</td>
<td>0101</td>
<td>0000000000000101</td>
</tr>
<tr>
<td>+2</td>
<td>0110</td>
<td>0000000000000110</td>
</tr>
<tr>
<td>+1</td>
<td>0111</td>
<td>0000000000000111</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td>0000000000001000</td>
</tr>
<tr>
<td>-1</td>
<td>FFFE</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>-2</td>
<td>FFFD</td>
<td>1111111111111110</td>
</tr>
<tr>
<td>-3</td>
<td>FFFC</td>
<td>111111111111111</td>
</tr>
<tr>
<td>-4</td>
<td>FFFC</td>
<td>111111111111111</td>
</tr>
<tr>
<td>-32768</td>
<td>8000</td>
<td>1000000000000000</td>
</tr>
</tbody>
</table>

Ascii 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

```
-3 1 1 1 1 1 1 1 1 1 1 0 1
+ 4 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 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
+ 2 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 1
```

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
   while(a > 0) {
   12: 9AFF print RA
   13: 1AAB RA ← RA + RB
       a = a + b
   14: 2BAB RB ← RA - RB
       b = a - b
   15: DAI2 if (RA > 0) goto 12
   16: 0000 halt
```

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
   while(a > 0) {
   12: 9AFF print RA
   13: 1AAB RA ← RA + RB
       a = a + b
   14: 2BAB RB ← RA - RB
       b = a - b
   15: DAI2 if (RA > 0) goto 12
   16: 0000 halt
```
Standard Input

Standard input.
- Loading from memory address FE loads one word from TOY stdin.
- 8AFF reads 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.

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

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

Load address. (opcode 7)
- Loads an 8-bit integer into a register.
- 7A30 means load 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

TOY main memory is a giant array.
- Can access memory cell 30 using load and store.
- BC30 means load mem[30] into register C.
- Goal: access memory cell i where i is a variable.

Load indirect. (opcode A)
- A06 means load mem[R6] into register C.

Store indirect. (opcode B)
- BC06 means store contents of register C into mem[R6].

```java
a = 30;
```

Arrays in TOY

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.

Information can be instructions!
- 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.

Can access memory cell
- register stores “pointer” to a memory cell

Load a small constant into a register.
- Load a 8-bit memory address into a register.
  - register stores “pointer” to a memory cell

```
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
Unsafe Code at any Speed

What happens if we make array start at 00 instead of 30?  
- Stop reading if 0000  
- Print sequence in reverse order.

### 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: 7A00  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
```

### Java Implementation of Reverse

```
public class Reverse {
    public static void main(String[] args) {
        int[] arr = new int[100];
        int n = 0;
        while (true) {
            if (n == 0) break;
            arr[n] = c;
            n++;
        }
    }
}
```

### Unsafe C Implementation of Reverse

```
int main() {
    char buffer[100];
    scanf("%s", buffer);
    printf("%s
", buffer);
    return 0;
}
```

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.

**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.]

**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.

**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.]
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.

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

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

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
TOY Simulator: Fetch

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

<table>
<thead>
<tr>
<th>$1C_{16}$</th>
<th>$C_{16}$</th>
<th>$A_{16}$</th>
<th>$R_{16}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1$</td>
<td>$0$</td>
<td>$0$</td>
<td>$1$</td>
</tr>
</tbody>
</table>

$0_{16}$ $1_{16}$ $0_{16}$ $1_{16}$ $1_{16}$ $0_{16}$ $1_{16}$ $1_{16}$

$\text{inst} = \text{mem}[\text{pc}++]$; // fetch and increment

$\text{int inst} = \text{inst} >> 8$; // opcode (bits 12-15)

$\text{int d} = (\text{inst} >> 9) \& 15$; // dest $d$ (bits 08-11)

$\text{int s} = (\text{inst} >> 4) \& 15$; // source $s$ (bits 04-07)

$\text{int t} = (\text{inst} >> 0) \& 15$; // source $t$ (bits 00-03)

$\text{int addr} = (\text{inst} >> 0) \& 255$; // addr (bits 00-07)

TOY Simulator: Execute

if ($\text{op} == 0$) break; // halt

switch ($\text{op}$) {
    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: $R[d] = \text{addr}$; break;
    case 8: $R[d] = \text{mem[addr]}$; break;
    case 9: $\text{mem[addr]} = R[d]$; break;
    case 10: $R[d] = \text{mem[R[t]}]$; break;
    case 11: $\text{mem[R[t]}] = R[d]$; break;
    case 12: if ($R[d] == 0$) $\text{pc} = \text{addr}$; break;
    case 13: if ($R[d] > 0$) $\text{pc} = \text{addr}$; break;
    case 14: $\text{pc} = R[d]$; break;
    case 15: $R[d] = \text{pc}; \text{pc} = \text{addr}$; break;
}

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.

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.