5. The TOY Machine II

Laboratory Instrument Computer (LINC)
What We've Learned About TOY

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

TOY programming.
- TOY instruction set architecture: 16 instruction types.
- 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.
Data Representation
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: $1 + 4 + 16 + 32 + 64 = 117_{10}$.
- As character: $117^{th}$ Unicode character = 'u'.
- As music: $117/256$ position of speaker.
- As grayscale value: 45.7% black.
Adding and Subtracting Binary Numbers

Decimal and binary addition.

\[\begin{array}{c}
\text{11} \\
013 \\
+ 092 \\
\hline
105
\end{array} \quad \begin{array}{c}
\text{11} \\
00001101 \\
+ 01011100 \\
\hline
01101001
\end{array}\]

Subtraction. Add a negative integer.

\[\text{e.g., } 6 - 4 = 6 + (-4)\]

Q. 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 \(x\), is zero.

\[
\begin{array}{c}
x & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
0 & \text{0 0 0 0 0 0 0 0 0} \\
\end{array}
\]

\[
\begin{array}{c}
x & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
+1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
\text{flip bits and add 1} & \text{1 1 1 1 1 1 1 1 1 1} \\
0 & \text{0 0 0 0 0 0 0 0 0} \\
\end{array}
\]
Two's Complement Integers

To compute \(-x\) from \(x\):

- Start with \(x\).

  \( +4 \)
  \[
  \begin{array}{cccccccccccccccc}
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
  \end{array}
  \]

  leading bit

- Flip bits.

  \( -5 \)
  \[
  \begin{array}{cccccccccccccccc}
  1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
  \end{array}
  \]

- Add one.

  \( -4 \)
  \[
  \begin{array}{cccccccccccccccc}
  1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
  \end{array}
  \]
## 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>1111111111111111</td>
</tr>
<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>0000000000000100</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>1111111111111101</td>
</tr>
<tr>
<td>-4</td>
<td>FFFC</td>
<td>1111111111111100</td>
</tr>
<tr>
<td>-32768</td>
<td>8000</td>
<td>1000000000000000</td>
</tr>
</tbody>
</table>
Properties of Two's Complement Integers

Properties:

- Leading bit (bit 15) signifies sign.
- Addition and subtraction are easy.
- 0000000000000000 represents zero.
- Checking for arithmetic overflow is easy.
- 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.

[Image: http://xkcd.com/571]
Representing Other Primitive Data Types in TOY

**Bigger integers.** Use two 16-bit TOY words per 32-bit Java `int`.

**Real numbers.**
- Use IEEE floating point (like scientific notation).
- Use four 16-bit TOY words per 64-bit Java `double`.

**Characters.** Use one 16-bit TOY word per 16-bit Java Unicode `char`.

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

```
M = 01001101_2 = 4D_{16}
O = 01001111_2 = 4F_{16}
M = 01001101_2 = 4D_{16}
```
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
    13: 1AAB  RA ← RA + RB  a = a + b
    14: 2BAB  RB ← RA - RB  b = a - b
}

15: DA12  if (RA > 0) goto 12
16: 0000  halt
```

fibonacci.toy
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.

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00: 0000</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10: 8C00</td>
<td>RC &lt;- mem[00]</td>
<td></td>
</tr>
<tr>
<td>11: 8AFF</td>
<td>read RA from stdin</td>
<td></td>
</tr>
<tr>
<td>12: CA15</td>
<td>if (RA == 0) pc &lt;- 15</td>
<td></td>
</tr>
<tr>
<td>13: 1CCA</td>
<td>RC &lt;- RC + RA</td>
<td></td>
</tr>
<tr>
<td>14: C011</td>
<td>pc &lt;- 11</td>
<td></td>
</tr>
<tr>
<td>15: 9CFF</td>
<td>write RC</td>
<td></td>
</tr>
<tr>
<td>16: 0000</td>
<td>halt</td>
<td></td>
</tr>
</tbody>
</table>
Standard Input and Output: Implications

Standard input and output enable you to:

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

**standard input**

- redirected from punchcard
- by default: flip switch, press button

**standard output**

- by default: LED
- redirected to punchcard
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.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccccccccccccc}
7_{16} & A_{16} & 3_{16} & 0_{16} \\
\text{opcode} & \text{dest d} & \text{addr} \\
\end{array}
\]
Arrays in TOY

TOY main memory is a giant array.

- Can access memory cell 30 using load and store.
- 8C30 means load \texttt{mem[30]} into register \texttt{C}.
- Goal: access memory cell \(i\) where \(i\) is a variable.

Load indirect. [opcode A]

- AC06 means load \texttt{mem[R6]} into register \texttt{C}.

Store indirect. [opcode B]

- BC06 means store contents of register \texttt{C} into \texttt{mem[R6]}.

```java
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, ... until reading 0000.
- Print sequence in reverse order.

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

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

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, ... until reading 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!

```
00   R1  0001    constant 1
01   RA  0000    a[]
02   RB  0000    n

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

Consequences. Viruses and worms.

Java enforces security.
- Type safety.
- Array bounds checking.
- Not foolproof.

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

Unsafe C program

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

**Lesson.**

- Not easy to write error-free software.
- Embrace Java security features.
- Keep your OS patched.
Buffer Overflow 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 in mem[10] to mem[FF]. 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

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
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 `mem[10]` to `mem[FF]` 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
Extra Slides
Two's Complement Arithmetic

Addition is carried out as if all integers were positive.
- It usually works.

-3
\[
\begin{array}{cccccccccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
\end{array}
\]

+ 4
\[
\begin{array}{cccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\end{array}
\]

= 1
\[
\begin{array}{cccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]
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 or carry out out of sign bit with no carry in

+32,767

\[ \begin{array}{cccccccccccccccc}
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array} \]

+ 

2

\[ \begin{array}{cccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\end{array} \]

= 

\[ \begin{array}{cccccccccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{array} \]
# Java and TOY

Correspondence between Java constructs and TOY mechanisms.

<table>
<thead>
<tr>
<th>Java</th>
<th>TOY</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>load, store</td>
</tr>
<tr>
<td>arithmetic expressions</td>
<td>add, subtract</td>
</tr>
<tr>
<td>logical expressions</td>
<td>xor, and, shifts</td>
</tr>
<tr>
<td>loops (for, while)</td>
<td>jump absolute, branch</td>
</tr>
<tr>
<td>branches (if-else, switch)</td>
<td>branch if zero, positive</td>
</tr>
<tr>
<td>arrays, linked lists</td>
<td>indirect addressing</td>
</tr>
<tr>
<td>function call</td>
<td>jump and link, jump indirect</td>
</tr>
<tr>
<td>recursion</td>
<td>implement stack with arrays</td>
</tr>
<tr>
<td>whitespace</td>
<td>no-op 1000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>