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 bits = \((255 \times 16) + (15 \times 16) + (8)\) = 541 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.

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```
Adding and Subtracting Binary Numbers

Decimal and binary addition.

\[
\begin{align*}
1 & \quad 11 \\
013 & \quad 00001101 \\
+ 092 & \quad + 01011100 \\
105 & \quad 01101001
\end{align*}
\]

carries

Subtraction. Add a negative integer.

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

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.

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

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

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

- Start with \(x\).

\[
\begin{array}{c}
+4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
-5 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
-4 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
\end{array}
\]
## 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>0 1 1 1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>+4</td>
<td>0004</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 1 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 1 1</td>
</tr>
<tr>
<td>+2</td>
<td>0002</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 1 0</td>
</tr>
<tr>
<td>+1</td>
<td>0001</td>
<td>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</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</td>
</tr>
<tr>
<td>-2</td>
<td>FFFE</td>
<td>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 0 1</td>
</tr>
<tr>
<td>-4</td>
<td>FFFC</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 0 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</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 -32768 but not 32768.

**Java.** Java's int data type is a 32-bit two's complement integer.
**Ex.** 2147483647 + 1 equals -2147483648.

![Comic of sheep on a fence with an equation leading to a sheep in a boat.](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 Unicode (16 bits per char).
- Use one 16-bit TOY word per 16-bit Java `double`.

**Note.** Real microprocessors add hardware support for `int` and `double`. 
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
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
16: 0000  halt

fibbonacci.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.

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>8C00</td>
<td>RC ← mem[00]</td>
</tr>
<tr>
<td>11</td>
<td>8AFF</td>
<td>read RA from stdin</td>
</tr>
<tr>
<td>12</td>
<td>CA15</td>
<td>if (RA == 0) pc ← 15</td>
</tr>
<tr>
<td>13</td>
<td>1CCA</td>
<td>RC ← RC + RA</td>
</tr>
<tr>
<td>14</td>
<td>C011</td>
<td>pc ← 11</td>
</tr>
<tr>
<td>15</td>
<td>9CFF</td>
<td>write RC</td>
</tr>
<tr>
<td>16</td>
<td>0000</td>
<td>halt</td>
</tr>
</tbody>
</table>
Standard input and output enable you to:

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

```
Java code
a = 0x30;
```

---

<table>
<thead>
<tr>
<th></th>
<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></td>
<td>0</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>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>7_{16}</th>
<th>A_{16}</th>
<th>3_{16}</th>
<th>0_{16}</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>dest d</td>
<td>addr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Arrays in TOY

TOY main memory is a giant array.
- Can access memory cell 30 using load and store.
- 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].

```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, ...
- 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
c = StdIn.readInt();
if (c == 0) break;

14: CC19    if (RC == 0) goto 19

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  
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!

```
const 1
a[]
n
while(true) {
  c = StdIn.readInt();
  if (c == 0) break;
  address of a[n]
  a[n] = c;
  n++;
}

% 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 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
", buffer);
    return 0;
}

unsafe C program

shine 50W bulb at DRAM
[Appel-Govindavajhala ’03]
Buffer Overflow Example: JPEG of Death

Stuxnet worm. [July 2010]
- Step 1. Natanz centrifuge fuel-refining plant employee plugs in USB flash drive.
- Step 2. Machine is Owned; data becomes code by exploiting Windows buffer overflow.
- Step 3. Uranium enrichment in Iran stalled.

Buffer overflow attacks. Morris worm, Code Red, SQL Slammer, iPhone unlocking, Xbox softmod, GDI+ library for JPEG, ... 

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: 7001  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
Q. How do you get it back?

A. Write short program `boot.toy` and run it to read contents of memory from tape.

```
| 00: 7001 | R1 ← 0001 |
| 01: 7210 | R2 ← 0010 |
| 02: 73FF | R3 ← 00FF |
| 03: 8AFF | read RA   |
| 04: BA02 | mem[R2] ← RA |
| 05: 1221 | R2 ← R2 + R1 |
| 06: 2432 | R4 ← R3 - R2 |
| 07: D403 | if (R4 > 0) goto 03 |
| 08: 0000 | halt |
```

`boot.toy`
TOY Simulator
Goal. Write a program to "simulate" the behavior of the TOY machine.

- TOY simulator in Java.
- TOY simulator in TOY!

```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

standard input
standard output
TOY Simulator: Fetch

Fetch. 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]; pc = addr; break;
    case 15: R[d] = pc; pc = addr; break;
}
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)
- Reuse software from old machines.

Ancient programs still running on modern computers.
- Ticketron.
- Lode Runner on Apple IIe.