5. The TOY Machine II

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.

Data Representation

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 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: 117th Unicode character = ‘u’.
- As music: 117/256 position of speaker.
- As grayscale value: 45.7% black.

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.

To compute $-x$ from $x$:
- Start with $x$.
- Flip bits.
- Add one.

Decimal and binary addition.

\[
\begin{array}{c}
1 \\
013 \\
+ 092 \\
105
\end{array}
\]

Subtraction. Add a negative integer.

e.g., $6 - 4 = 6 + (-4))$
Two's Complement Integers

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 \(-2147483648\).

Standard Input and Output

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 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 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 } while (a > 0)
16: 0000 halt
```

fibonacci.toy

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 Input and Output: Implications

- 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.

Pointers

redirected from punchcard
by default: flip switch, press button
redirected to punchcard
by default: LED

Enter

**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 an 8-bit memory address into a register.

```
opcode    dest d
0         0
2         2
1         1
5         5
7         7

a = 0x30;
```

**Java code**
```
register stores "pointer" to a memory cell
```

**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].

**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;
    memory address of a[n]
    a[n] = c;
    n++;
}
```

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

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

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!

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.
Booting

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  i = 10
01: 7210  R2 ← 0010
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`

TOY Simulator

**Goal.** Write a program to "simulate" the behavior of the TOY machine.
- TOY simulator in Java.
- TOY simulator in TOY!

```java
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)
```

```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
      ...  
    }
  }
}
```
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;
}