TOY II

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.

Quick Review: Multiply

```
0A: 0003 3 ← inputs
0B: 0009 9 ← output
0C: 0000 0 ← constants
0D: 0000
0E: 0001 1

10: 8A0A RA ← mem[0A] a
11: 8B0B RB ← mem[0B] b
12: 8C0D RC ← mem[0D] c = 0
13: 810E R1 ← mem[0E] always 1
14: CA18 if (RA == 0) pc ← 18 while (a != 0) {
   15: 1CCB RC ← RC + RB c = c + b
   16: 2AA1 RA ← RA - R1 a = a - 1
   17: C014 pc ← 14 }
18: 9C0C mem[0C] ← RC
19: 0000 halt
```

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 and implications.
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: \(1 + 4 + 16 + 32 + 64 = 117\) (base ten).
- As character: 117th 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.

<table>
<thead>
<tr>
<th>Dec</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>013</td>
<td>000001101</td>
</tr>
<tr>
<td>+ 092</td>
<td>+ 01011100</td>
</tr>
<tr>
<td>105</td>
<td>01101001</td>
</tr>
</tbody>
</table>

Subtraction. Add a negative integer.

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 \(x\), yields zero.
Two's Complement Integers

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

- Start with \(x\).

  \(+4\) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

- Flip bits.

  \(-5\) 1 1 1 1 1 1 1 1 1 1 0 1 1

- Add one.

  \(-4\) 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0

Properties of Two's Complement Integers

- Leading bit (bit 15 in Toy) signifies sign.
- Addition and subtraction are easy.
- 0000000000000000 represents zero.
- Negative integer \(-x\) represented by \(2^{16} - x\).
- Not symmetric: can represent \(-32,768\) but not 32,768.

Java. Java's \(\text{int}\) data type is a 32-bit two's complement integer.
Ex. \(2147483647 + 1\) equals \(-2147483648\).

Representing Other Primitive Data Types in TOY

Bigger integers. Use two 16-bit words per \(\text{int}\).

Real numbers.
- Use "floating point" (like scientific notation).
- Use four 16-bit words per \(\text{double}\).

Characters.
- Use ASCII code (8 bits / character).
- Pack two characters per 16-bit word.

Note. Real microprocessors add hardware support for \(\text{int}\) and \(\text{double}\).
Standard Input and Output

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.

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

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 an 8-bit memory address into a register.

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.

Example: Reverse an array

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.

Java version:
```java
int i = 0;
while (!StdIn.isEmpty())
{
    a[i] = StdIn.readInt();
    i++;
}
i--;
while (i >= 0)
{
    StdOut.println(a[i]);
    i--;
}
```

(We'll just assume a[] is big enough)
TOY Implementation of Reverse

• Read in a sequence of integers and store in memory $30, 31, 32, \ldots$
• Stop reading if 0000.
• Print sequence in reverse order.

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

while(true) {
    c = StdIn.readInt();
    if (c == 0) break;
    R6 ← RA + RB memory address of a[n]
    mem[R6] ← RC a[n] = c;
    RB ← RB + R1 n++;
    goto 13
}
```

print in reverse order

Unsafe Code at any Speed

Q. What happens if we make array start at 00 instead of 30?

```
10: 7101 R1 ← 0001 constant 1
11: 7A00 RA ← 0000 a[
12: 7B00 RB ← 0000 n

while(true) {
    c = StdIn.readInt();
    if (c == 0) break;
    R6 ← RA + RB memory address of a[n]
    mem[R6] ← RC a[n] = c;
    RB ← RB + R1 n++;
    goto 13
}
```

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.
• 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 [2004], . . .

Lesson.

- Not easy to write error-free software.
- Embrace Java security features.
- 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.

```
00: 7001  R1 ← 0001
01: 7210  R2 ← 0010    i = 10
02: 73FF  R3 ← 00FF
03: AA02  RA ← mem[R2]  do {
04: 9AFF  write RA       a = mem[i]
          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       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

Simulating the TOY machine
**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[] mem = new int[256]; // main memory
        while (true) {
            int inst = mem[pc++]; // fetch, increment
            // EXECUTE
            switch (inst) {
                case 0x1CAB:
                    mem[pc] = mem[pc] + mem[t];
                    break;
                case 0x10:
                    mem[pc] = mem[t];
                    break;
                case 0x00:
                    mem[pc] = mem[pc] & mem[t];
                    break;
                case 0x08:
                    mem[pc] = mem[pc] & mem[t];
                    break;
                case 0x09:
                    mem[addr] = mem[pc];
                    break;
                case 10:
                    mem[pc] = mem[mem[pc]];
                    break;
                case 11:
                    mem[mem[pc]] = mem[pc];
                    break;
                case 12:
                    if (mem[pc] == 0) pc = addr;
                    break;
                case 13:
                    if (mem[pc] > 0) pc = addr;
                    break;
                case 14:
                    pc = mem[pc];
                    break;
                case 15:
                    mem[pc] = pc;
                    break;
            }
        }
    }
}
```

**Extract destination register of 1CAB by shifting and masking.**

<table>
<thead>
<tr>
<th>inst</th>
<th>inst &gt;&gt; 8</th>
<th>inst &gt;&gt; 8) &amp; 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>10000001</td>
<td>00001000</td>
</tr>
<tr>
<td>0110</td>
<td>01110011</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>10101100</td>
<td>00101010</td>
</tr>
<tr>
<td>0101</td>
<td>10000010</td>
<td></td>
</tr>
</tbody>
</table>

**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

Building a new computer? Need a plan for old software.

Two possible approaches
- Rewrite software (costly, error-prone, boring, and time-consuming).
- Simulate old computer on new computer.

Ancient programs still running on modern computers.
- Payroll
- Power plants
- Air traffic control
- Ticketron.
- Games.