TOY II

LINC

LINC

Introduction to Computer Science • Sedgewick and Wayne • Copyright © 2007 • http://www.cs.Princeton.EDU/IntroCS
What We’ve Learned About TOY

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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0A: 0003</td>
<td>RA ← mem[OA]</td>
</tr>
<tr>
<td>0B: 0009</td>
<td>RB ← mem[OB]</td>
</tr>
<tr>
<td>0C: 0000</td>
<td>RC ← mem[OD]</td>
</tr>
<tr>
<td>0D: 0000</td>
<td>R1 ← mem[OE]</td>
</tr>
<tr>
<td>0E: 0001</td>
<td>always 1</td>
</tr>
<tr>
<td>10: 8A0A</td>
<td>if (RA == 0) pc ← 18</td>
</tr>
<tr>
<td>11: 8B0B</td>
<td>RC ← RC + RB</td>
</tr>
<tr>
<td>12: 8C0D</td>
<td>R1 ← RA - R1</td>
</tr>
<tr>
<td>13: C014</td>
<td>pc ← 14</td>
</tr>
<tr>
<td>14: C118</td>
<td>mem[OC] ← RC</td>
</tr>
<tr>
<td>15: 0000</td>
<td>halt</td>
</tr>
</tbody>
</table>

Quick Review: Multiply

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0A: 0003</td>
<td>3 ← inputs</td>
</tr>
<tr>
<td>0B: 0009</td>
<td>9 ← output</td>
</tr>
<tr>
<td>0C: 0000</td>
<td>0 ← constants</td>
</tr>
<tr>
<td>0D: 0000</td>
<td>0 ← output</td>
</tr>
<tr>
<td>0E: 0001</td>
<td>1</td>
</tr>
</tbody>
</table>

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 and implications.
Digital World

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.

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.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

```
  0 0 1 1 0 1 0 0
-5 + 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
  0 0 1 1 0 1 0 0
-4 + 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
  0 0 1 1 0 1 0 0
-3 + 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
  0 0 1 1 0 1 0 0
-2 + 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
  0 0 1 1 0 1 0 0
-1 + 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
  0 0 1 1 0 1 0 0
 0 + 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Adding and Subtracting Binary Numbers

Decimal and binary addition.

```
  1 0 1 1 0 0 0 0
+ 0 1 0 1 1 0 0
  1 1 1 0 1 1 0 0
```

Subtraction. Add a negative integer.

e.g., $6 - 4 = 6 + (-4)$

```
  1 1 0 1 1 0 0
+ 0 1 0 1 1 0 0
  1 1 1 0 1 1 0 0
```

Q. OK, but how to represent negative integers?

“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 0
• Flip bits.
  -5 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
• Add one.
  -4 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

loading bit determines sign

-x: flip bits and add 1
"Two's Complement Integers"
Two's Complement Integers

Properties of Two's Complement Integers

Properties.
- 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 `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 `int`.

Real numbers.
- Use "floating point" (like scientific notation).
- Use four 16-bit words per `double`.

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

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
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

• 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

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.

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] a variable index
• AC06 means load mem[R6] into register C.

Store indirect. [opcode B] a variable index
• BC06 means store contents of register C into mem[R6].

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 n = 0;
while (!StdIn.isEmpty())
{
    a[n] = StdIn.readInt();
    n++;
}

n--;
while (n >= 0)
{
    StdOut.println(a[n]);
    n--;
}
```

(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, ...
• 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) {
    13: 8CFF read RC c = StdIn.readInt();
    14: CC19 if (RC == 0) goto 19
        if (c == 0) break;
    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
}

print in reverse order

-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) {
    13: 8CFF read RC c = StdIn.readInt();
    14: CC19 if (RC == 0) goto 19
        if (c == 0) break;
    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
}
Unsafe Code at any Speed

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.

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.
Bootimg

Q. How do you get it back?
A. Write short program boot.toy and run it to read contents of memory from tape.

```plaintext
00: 7001 R1 ← 0001
01: 7210 R2 ← 0010
02: 73FF R3 ← 00FF
     i = 10
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[] R = new int[16]; // registers
        int[] mem = new int[256]; // main memory
        // READ .toy FILE into mem[]
        while (true) {
            int inst = mem[pc++]; // fetch, increment
            // EXECUTE
            int op = (inst >> 12); // opcode (bits 12-15)
            int d = (inst >> 8) & 15; // dest d (bits 08-15)
            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)
            // execute
            // standard input
            // standard output
        }
    }
}
```

TOY Simulator: Decode

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

```plaintext
0 0 0 1 1 1 0 0 1 0 1 0 1 0 1 1
    inst
15

0 0 0 0 0 0 0 0 0 1 1 0 0
    inst >> 8
0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0
15

0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
    (inst >> 8) & 15
0 0 0 0 0 0 0 0 0 0 1 1 0 0
15
```

% more add-stdin.toy
10: 8C00 // TOY program
11: 8AFF
12: CA15
13: 1CCA
14: C011
15: 9CFF
16: 0000
% java TOY add-stdin.toy
TOY Simulator: Execute

```java
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]; break;
    case 15: R[d] = pc; pc = addr; break;
}
```

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.

TOY Simulator: Omitted Details

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.