Lecture A2: TOY Programming

What We’ve Learned About TOY

TOY: what’s in it, how to use it.
- Von Neumann architecture.
- Box with switches and lights.

Data representation.
- Binary and hexadecimal.

TOY instructions.
- Instruction set architecture.

Sample TOY machine language programs.
- \( 1 + 2 + 3 + \ldots + n \).
- LFBSR.
- Polynomial evaluation.

What We Do Today

Represent data other than positive integers.
- Negative numbers.

Represent data structures.
- Arrays.

Make function calls.

Relate TOY, C, and “real computers”.

Representing Negative Numbers (Two’s Complement)

| Number | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| +32767 | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |
| +3     | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| +2     | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| +1     | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| 0      | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| -1     | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |
| -2     | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |
| -3     | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  |
| -4     | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  |
| -32768 | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

***

***
Two's Complement Integers

Properties:
- Leading bit (bit 15) signifies sign.
- Negative integer \(-N\) represented by \(2^{16} - N\).
- Trick to compute \(-N\):
  1. Start with \(N\).
  2. Flip bits.
  3. Add 1.

<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>+4</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>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Nice properties:
- \(0000000000000000\) represents 0.
- \(-0\) and \(+0\) are the same.
- Addition is easy (see next slide).

Not-so-nice properties.
- Can represent one more negative integer than positive integer
  \((-32,768 = -2^{15} \text{ but not } 32,768 = 2^{15})\).

Alternatives other than two's complement exist.
- Many C compilers use two's complement.
- But not all, so do not assume they do.
- Unsafe C code to test if \(a\) is odd: \(\text{if } (a \& 1)\)

Two's Complement Arithmetic

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

\[-3 \quad 1\ |
\quad 1\ |
\quad 1\ |
\quad 1\ |
\quad 1\ |
\quad 1\ |
\quad 1\ |
\quad 1\ |
\quad 1\ |
\quad 0\ |
\quad 1\ |
\]

\[+ \quad 4\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 1\ |
\quad 0\ |
\]

\[= \quad 1\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 1\ |
\]

Two's Complement Arithmetic

Addition is carried out as if all integers were positive.
- It usually works.
- But overflow can occur:
  - carry into sign bit with no carry out

\[+32,767 \quad 0\ |
\quad 1\ |
\quad 1\ |
\quad 1\ |
\quad 1\ |
\quad 1\ |
\quad 1\ |
\quad 1\ |
\quad 1\ |
\quad 1\ |
\quad 1\ |
\quad 1\ |
\quad 1\ |
\quad 1\ |
\]

\[+ \quad 2\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 1\ |
\]

\[= \quad 1\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 1\ |
\]

\[-32,767 \quad 1\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 0\ |
\quad 1\ |
\]
Representing Other Primitive Data Types

Big integers.
- Can use "multiple precision."
- Use two 16-bit words per integer.

Real numbers.
- Can use "floating point" (like scientific notation).
- Double word for extra precision.

Character strings.
- Can use ASCII code (8 bits / character).
- Can pack two characters into one 16-bit word.

Indexed Addressing

Static addressing.
- So far, all load/store addresses hardwired inside instruction.
- Ex. 9234: \( R2 \leftarrow \text{mem}[34] \)
- Need more flexibility to implement arrays, functions, etc.

Indexed (dynamic) addressing.
- Want to be able to make memory index a variable, instead of hardwiring '34'.

Solution.
- Put memory address in register. (C "pointer")
- Use CONTENTS of register as address.
- Augment instruction format to use address register.

Review: Format 2 Instructions

Register-memory / register-immediate.
- Bits 12-15 encode opcode.
- Bits 8-11 encode destination register.
- Bits 0-7 encode memory address or arithmetic constant.

Ex: 9234 means
- Load contents of memory location \( 34_{16} \) into register \( R2 \).
- \( R2 \leftarrow \text{mem}[34] \)

Indexed Addressing

Bits 11 signifies "indexed addressing."
- If Bit 11 is 0 then Format 2 as usual.
- If Bit 11 is 1 then replace addr by \( R1 + R2 \)
- 9234 means \( R2 \leftarrow \text{mem}[34] \)
- 9A34 means \( R2 \leftarrow \text{mem}[R3 + R4] \)
Why "Stealing" Bit 11 is OK

Bits 11 signifies "indexed addressing."
- We only have 8 registers.
- Only 3 bits (8-10) needed to distinguish among 8 values.
- Can "steal" bit 11.

Could we do the same for Format 1 instructions?

Sample C Program: Array

Goal: put Fibonacci numbers into array \( a[\cdot] \).
- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

```c
int main(void) {
    int n, i, j, k, a[16];
    n = 15;
    a[0] = 1; a[1] = 1;
    i = 0; j = 1; k = 2;
    do {
        a[k] = a[i] + a[j];
        i++; j++; k++;
        n--;
    } while (n > 0)
    return 0;
}
```

Sample TOY Program 3: Array

```
use indexed addressing three times
```

Food for Thought

What happens if \( \text{mem}[12] = \text{B210} \) instead of \( \text{B2D0} \)?

```toy
use indexed addressing three times
```

Overwrites \( \text{mem} \) location 12, then 13, then 14, ...

mystery.toy
Branches and Loops

Press GO, TOY machine either:
- Executes some instructions and halts.
- Gets caught in an infinite loop.

Infinite loop.
- Puzzles and/or panics programmers. Why doesn’t compiler detect and tell me?
- Control structures (while, for) help manage control flow and avoid looping.
- Can always top machine by pulling plug! (Ctrl-c)

Function Calls

Functions can be used and written by different people.

Issues:
- How to pass parameter values?
- How to know where to return? (may have multiple calls)

Solution: adhere to CALLING conventions.
- Agreement between function and calling program on where to store parameters and return address.
- Assume parameter value(s) in certain register(s).
- Assume return value in specific register.
- Use indexed jump to return.

Other possible solutions.

TOY Program 4: Function Call

Goal: create function to compute \( a^b \).

Calling convention. Store:
- 0 in R0
- a in R1
- b in R2
- addr in R4
- result in R3

How to compute \( a^b \)?
- Set R3 = 1.
- Loop b times.
  - multiply R3 by a each time

Client program to compute \( x^4 + y^5 \). Assume
- \( x \) in memory location D0
- \( y \) in memory location D1

\[
\begin{align*}
10: & \text{ B000 R0 <- 0} \\
11: & \text{ 91D0 R1 <- x} \\
12: & \text{ B204 R2 <- 4} \\
13: & \text{ 8420 R3 <- x^4 (using function)} \\
14: & \text{ 1530 R5 <- R3} \\
15: & \text{ 91D1 R1 <- y} \\
16: & \text{ B205 R2 <- 5} \\
17: & \text{ 8420 R3 <- y^5 (using function)} \\
18: & \text{ 1535 R5 <- x^4 + y^5}
\end{align*}
\]
How To Build a TOY Machine

Hardware.
- See Lecture A3-A5.

Simulate in software.
- Write a program to "simulate" the behavior of the TOY machine.
  - Java TOY simulator.
  - C TOY simulator.

TOY SIMULATOR: toy.c

```c
int main(void) {
    short int inst, R[8], mem[256];
    unsigned char pc = 0X10;
    int i, op, addr, r0, r1, r2, c;
    for (i = 0; i < 256; i++)
        mem[i] = 0;
    while (scanf("%hX%hX", &i, &inst) != EOF)
        mem[i] = inst;
    do {
        inst = mem[pc++];
        op = (inst >> 12) & 15;
        r0 = (inst >>  8) &  7;           
        r1 = (inst >>  4) &  7;           
        r2 = (inst >>  0) &  7;           
        addr = (inst >>  0) & 255;
        if ((inst >> 11) & 1)
            addr = (R[r1] + R[r2]) & 255;
        . . .
    } while (op != 0)
    return 0;
}
```

Simulation

Consequences of simulation.
- Test out new machine (or microprocessor) using simulator.
  - cheaper and faster than building actual machine
- Easy to add other functions to simulator.
  - trace, single-step, breakpoint debugging
  - simulator more powerful than TOY itself
- Reuse software for old machines.

Ancient programs still running on modern computers.
- Ticketron - 5 cents per ticket.
- Lode Runner on Apple IIe.

Apple IIe Simulator
C and TOY

Correspondence between C constructs and TOY mechanisms.

<table>
<thead>
<tr>
<th>C</th>
<th>TOY</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>load, store</td>
</tr>
<tr>
<td>arithmetic expressions</td>
<td>add, multiply, subtract</td>
</tr>
<tr>
<td>logical expressions</td>
<td>xor, and, shifts</td>
</tr>
<tr>
<td>loops (for, while)</td>
<td>jump and count</td>
</tr>
<tr>
<td>branches (if-else, switch)</td>
<td>jump if positive, jump</td>
</tr>
<tr>
<td>arrays, linked lists</td>
<td>indexed addressing</td>
</tr>
<tr>
<td>function call</td>
<td>jump and link</td>
</tr>
<tr>
<td>recursion</td>
<td>implement stack with arrays</td>
</tr>
<tr>
<td>whitespace</td>
<td>D000</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Bootstrapping

Translate TOY program into C?

Translate C program to TOY?

Translate TOY simulator into TOY? (!)

Bootstrapping.
- Build "first" machine.
- Implement simulator of itself.
  - C compiler written in C
- Modify simulator to try new designs. (still going on!)