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 + . . . n.
  - LFBSR.
  - Polynomial evaluation.
What We Do Today

Represent data other than positive integers.
  ■ Negative numbers.

Manipulate addresses.
  ■ Indexed addressing and "pointers."

Represent data structures.
  ■ Arrays.

Implement functions.

Relate TOY, C, and "real computers".
Representing Negative Numbers (Two’s Complement)

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+32767</td>
<td>0</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>
<td>0</td>
</tr>
<tr>
<td>+3</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>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>+2</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>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>+1</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>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<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>1</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>
</tr>
<tr>
<td>-2</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>-3</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-32768</td>
<td>1</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>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Two’s Complement Integers Properties

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 = \(-2^{15}\) 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:

\[ -3 + 4 = 1 \]

\[
\begin{array}{cccccccccccccccc}
-3 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
+ & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
= & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}
\]
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 | 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 |
|        + |                |
|        = |                |
| -32,767 | 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 |
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: \texttt{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. \textit{(C "pointer")}
- Use CONTENTS of register as address.
- Augment instruction format to use address register.

```c
d[0] = 1;
d[1] = 1;
for (i = 2; i < 16; i++)
    d[i] = d[i-1] + d[i-2];
```
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 \( R2 \leftarrow \text{mem}[R3 + R4] \)

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

\( 9_{16} \) \quad \text{opcode} \quad \begin{array}{c} A_{16} \end{array} \quad \begin{array}{c} 3_{16} \end{array} \quad \begin{array}{c} 4_{16} \end{array} \quad \text{dest} \quad \text{regA} \quad \text{regB} \\

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

\( 9_{16} \) \quad \text{opcode} \quad \begin{array}{c} 2_{16} \end{array} \quad \begin{array}{c} 3_{16} \end{array} \quad \begin{array}{c} 34_{16} \end{array} \quad \text{dest} \quad \text{addr}
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?

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

<table>
<thead>
<tr>
<th>(9_{16})</th>
<th>(2_{16})</th>
<th>(34_{16})</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>dest</td>
<td>addr</td>
</tr>
</tbody>
</table>
Sample C Program: Array

Goal: put Fibonacci numbers into array a[].

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . .

```c
int main(void) {
    int n, i, j, k, d[16];
    n = 10;
    d[0] = 1; d[1] = 1;
    i = 0; j = 1; k = 2;
    do {
        d[k] = d[i] + d[j];
        i++; j++; k++;
    } while (--n > 0)
    return 0;
}
```

implement in TOY using indexed addressing

do-while more natural to implement in TOY
Sample TOY Program 3: Array

use indexed addressing three times

fibonacci.toy

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>B10A</td>
<td>R1 &lt;- 000A</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>B001</td>
<td>R0 &lt;- 0001</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>B2D0</td>
<td>R2 &lt;- 00D0</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>A0D0</td>
<td>mem[D0] &lt;- 1</td>
<td>a[0] = 1</td>
</tr>
<tr>
<td>14</td>
<td>A0D1</td>
<td>mem[D1] &lt;- 1</td>
<td>a[1] = 1</td>
</tr>
<tr>
<td>15</td>
<td>B300</td>
<td>R3 &lt;- 0</td>
<td>i = 0</td>
</tr>
<tr>
<td>16</td>
<td>B401</td>
<td>R4 &lt;- 1</td>
<td>j = 1</td>
</tr>
<tr>
<td>17</td>
<td>B502</td>
<td>R5 &lt;- 2</td>
<td>k = 2</td>
</tr>
<tr>
<td>18</td>
<td>9E23</td>
<td>R6 &lt;- mem[R2 + R3]</td>
<td>a[i]</td>
</tr>
<tr>
<td>19</td>
<td>9F24</td>
<td>R7 &lt;- mem[R2 + R4]</td>
<td>a[j]</td>
</tr>
<tr>
<td>1A</td>
<td>1667</td>
<td>R6 &lt;- R6 + R7</td>
<td></td>
</tr>
<tr>
<td>1B</td>
<td>AE25</td>
<td>mem[R2 + R5] &lt;- R6</td>
<td>a[k]</td>
</tr>
<tr>
<td>1C</td>
<td>1330</td>
<td>R3++</td>
<td>i++</td>
</tr>
<tr>
<td>1D</td>
<td>1440</td>
<td>R4++</td>
<td>j++</td>
</tr>
<tr>
<td>1E</td>
<td>1550</td>
<td>R5++</td>
<td>k++</td>
</tr>
<tr>
<td>1F</td>
<td>7118</td>
<td>to 18 if --R1 &gt; 0</td>
<td></td>
</tr>
</tbody>
</table>
**Food for Thought**

What happens if we change B10A to B1AA?

<table>
<thead>
<tr>
<th>Mystery.toy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>10</strong>: B10A</td>
</tr>
<tr>
<td><strong>11</strong>: B001</td>
</tr>
<tr>
<td><strong>12</strong>: B2D0</td>
</tr>
<tr>
<td><strong>13</strong>: A0D0</td>
</tr>
<tr>
<td><strong>14</strong>: A0D1</td>
</tr>
<tr>
<td><strong>15</strong>: B300</td>
</tr>
<tr>
<td><strong>16</strong>: B401</td>
</tr>
<tr>
<td><strong>17</strong>: B502</td>
</tr>
<tr>
<td><strong>18</strong>: 9E23</td>
</tr>
<tr>
<td><strong>19</strong>: 9F24</td>
</tr>
<tr>
<td><strong>1A</strong>: 1667</td>
</tr>
<tr>
<td><strong>1B</strong>: AE25</td>
</tr>
<tr>
<td><strong>1C</strong>: 1330</td>
</tr>
<tr>
<td><strong>1D</strong>: 1440</td>
</tr>
<tr>
<td><strong>1E</strong>: 1550</td>
</tr>
<tr>
<td><strong>1F</strong>: 7118</td>
</tr>
</tbody>
</table>
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)

infinite loop

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10:</td>
<td>B101</td>
<td>R1 ← 0001</td>
</tr>
<tr>
<td>11:</td>
<td>5010</td>
<td>to 10</td>
</tr>
</tbody>
</table>
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)

One solution: adhere to CALLING conventions.
- Agreement between function and calling program on where to store parameters and return address.
- Assume parameter value(s) in specific register(s).
- Assume return value(s) in specific register(s).
- Save return address (jump-and-link).
- Use indexed jump to return.
TOY Program 4: Function Call

Goal: create function to compute $a^b$.

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

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

```
20: B301    R3 <- 0001
21: 1223    R2++
22: 5024    jump to 24
23: 3331    R3 <- R3 * R1
24: 7223    to 23 if --R2 > 0
25: 5804    jump to addr in R4
```

Handle b = 0

pc <- R0 + R4 = R4
Client program to compute $x^4 + y^5$. Assume
x in memory location D0
y in memory location D1

 opcode 8
 jump and link

 R4 <- 14
 pc <- 20

 function.toy

<table>
<thead>
<tr>
<th>Opcode</th>
<th>R4</th>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:</td>
<td>B000</td>
<td></td>
<td></td>
<td></td>
<td>R0 &lt;- 0</td>
<td></td>
</tr>
<tr>
<td>11:</td>
<td>91D0</td>
<td></td>
<td></td>
<td>R1</td>
<td>R1 &lt;- x</td>
<td></td>
</tr>
<tr>
<td>12:</td>
<td>B204</td>
<td></td>
<td></td>
<td>R2</td>
<td>R2 &lt;- 4</td>
<td></td>
</tr>
<tr>
<td>13:</td>
<td>8420</td>
<td></td>
<td></td>
<td>R3</td>
<td>R3 &lt;- x$^4$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(using function)</td>
<td></td>
</tr>
<tr>
<td>14:</td>
<td>1530</td>
<td></td>
<td></td>
<td>R5</td>
<td>R5 &lt;- R3</td>
<td></td>
</tr>
<tr>
<td>15:</td>
<td>91D1</td>
<td></td>
<td></td>
<td>R1</td>
<td>R1 &lt;- y</td>
<td></td>
</tr>
<tr>
<td>16:</td>
<td>B205</td>
<td></td>
<td></td>
<td>R2</td>
<td>R2 &lt;- 5</td>
<td></td>
</tr>
<tr>
<td>17:</td>
<td>8420</td>
<td></td>
<td></td>
<td>R3</td>
<td>R3 &lt;- y$^5$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(using function)</td>
<td></td>
</tr>
<tr>
<td>18:</td>
<td>1535</td>
<td></td>
<td></td>
<td>R5</td>
<td>R5 &lt;- x$^4$ + y$^5$</td>
<td></td>
</tr>
</tbody>
</table>
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.
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;
}
Shifting and Masking

Extract destination register.
- Given 16 bit integer in C, isolate bits 8-10.
- Use bit operations in C.

\[
\text{inst} = B204_{16} = 45572_{10}
\]

\[
\begin{array}{cccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}
\]

\[
\text{(inst >> 8)}
\]

\[
\begin{array}{cccccccccccccccc}
9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0
\end{array}
\]

\[
7
\]

\[
\begin{array}{cccccccccccccccc}
8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1
\end{array}
\]

\[
\text{(inst >> 8) & 7}
\]

\[
\begin{array}{cccccccccccccccc}
8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0
\end{array}
\]
switch (op) {
    case 0: break;
    case 1: R[r0] = R[r1] + R[r2]; break;
    case 2: R[r0] = R[r1] - R[r2]; break;
    case 3: R[r0] = R[r1] * R[r2]; break;
    case 4: printf("%04X\n", R[r0]); break;
    case 5: pc = addr; break;
    case 6: if (R[r0] > 0) pc = addr; break;
    case 7: if (--R[r0]) pc = addr; break;
    case 8: R[r0] = pc; pc = addr; break;
    case 9: R[r0] = mem[addr]; break;
    case 10: mem[addr] = R[r0]; break;
    case 11: R[r0] = addr; break;
    case 12: R[r0] = R[r1] ^ R[r2]; break;
    case 13: R[r0] = R[r1] & R[r2]; break;
    case 14: R[r0] = R[r0] >> addr; break;
    case 15: R[r0] = R[r0] << addr; break;
}
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.
- Lode Runner on Apple IIe.
## 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.
  • Modify simulator to try new designs. (still going on!)