Lecture A3: The TOY Machine

Overview

TOY Machine.

Machine language instructions.

Example machine language programs.

TOY notes available in course packet (Course Documents).

What is TOY?

An imaginary machine similar to:

- Ancient computers.

What is TOY?

An imaginary machine similar to:

- Ancient computers.
- Today’s microprocessors.
What is TOY?

An imaginary machine similar to:
- Ancient computers.
- Today’s microprocessors.

Why study?
- Simplified machine.
  - easier to understand
  - captures essence of modern microprocessors
- Machine language programming.
  - how do C programs relate to computer?
  - still (a few) situations today where it is really necessary
- Computer architecture.
  - how do machines work?

Inside the Box

Switches.
- Input data and programs.

Lights.
- View data.

Registers.
- Fastest form of storage.
- Use as scratch space during computation.
- 8 registers.
  - each stores 16 bits

Program counter (PC).
- An extra register.
- Keeps track of next instruction to be executed.

Inside the Box

Memory.
- Store data and programs.
- 256 "words".
  - each word stores 16 bits

ALU (arithmetic-logic unit).
- Execute instructions and manipulate data.

Data and Programs Encoded in Binary

Each bit consists of two states:
- Switch is ON or OFF.
- High voltage or low voltage.
- 1 or 0.
- True or false.

How to represent integers?
- Use binary encoding.
- Ex: $6375_{10} = 0001100011100111_2$

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

$6375 = 2^{12} + 2^{11} + 2^7 + 2^6 + 2^5 + 2^2 + 2^1 + 2^0$

$= 4096 + 2048 + 128 + 64 + 32 + 4 + 2 + 1$
**Shorthand Notation**

Use hexadecimal (base 16) representation.
- Binary code, four bits at a time.

### Hexadecimal digits

<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>F</td>
<td>E</td>
<td>D</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Shorthand Notation**

Use hexadecimal (base 16) representation.
- Binary code, four bits at a time.
- Ex: $6375_{10} = 0001100011101110 = 18E7_{16}$

### Machine “Dump”

Everything is encoded in binary (hex).
- Integers, machine instructions, text, reals, etc.

Contents of machine in hexadecimal “dump”.
- Record of what program has done.
- Determines (with PC) what program will do.

### Program and Data

**Program**: 
- Sequence of instructions.

**Instruction**: 
- 16-bit word (interpreted one way).
- Changes contents of registers, memory, and PC in specified, well-defined ways.

**Data**: 
- 16-bit word (interpreted other ways).

**Program counter (PC)**: 
- Stores memory address of “next instruction.”

### Registers

<table>
<thead>
<tr>
<th>R7</th>
<th>R6</th>
<th>R5</th>
<th>R4</th>
<th>R3</th>
<th>R2</th>
<th>R1</th>
<th>R0</th>
</tr>
</thead>
<tbody>
<tr>
<td>F778</td>
<td>0770</td>
<td>0018</td>
<td>0401</td>
<td>0002</td>
<td>0030</td>
<td>00A0</td>
<td></td>
</tr>
</tbody>
</table>

### Main Memory

<table>
<thead>
<tr>
<th>00</th>
<th>0000 0000 0000 0000 0000 0000 0000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>08</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>10</td>
<td>9222 9120 1121 1220 1121 1221 7221 0000</td>
</tr>
<tr>
<td>18</td>
<td>0000 0001 0002 0003 0004 0005 0006 0007</td>
</tr>
<tr>
<td>20</td>
<td>0008 0009 000A 000B 000C 000D 000E 000F</td>
</tr>
<tr>
<td>28</td>
<td>0000 0000 0000 FE10 FACE CAFE ACED CEDE</td>
</tr>
<tr>
<td>38</td>
<td>1234 5678 9ABC DEFO 0000 0000 F00D 0000</td>
</tr>
<tr>
<td>40</td>
<td>0000 0000 EE00 EE00 EE00 EE00 EE00 EE00</td>
</tr>
<tr>
<td>48</td>
<td>8182 F185 FF00 0000 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>
How to Use the TOY Machine

To run a program:
- Load the program and data.
  (set switches, press LOAD for each word)
- Set switches to address of first instruction.
- Press GO.

Basic Cycle

GO button:
- Loads PC from address switches.
- Initializes fetch-increment-execute cycle.
- Machine runs until halt instruction reached.

Basic cycle:
- FETCH (get instruction from memory into ALU).
- INCREMENT program counter.
- EXECUTE (may require data from or to memory).

Output:
- System call can write output to output device (tty).
- Reads contents of memory word in lights.

TOY Instructions

Each instruction consists of 16 bits.
- Leftmost four bits (first hex digit) encode instruction type or "opcode."
- Divide up remaining 12 bits to denote component of each instruction.
- Format 1 and Format 2 instructions.
  - different ways of dividing up the 12 bits

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Format 1 Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: halt</td>
<td>0: halt</td>
</tr>
<tr>
<td>1: add</td>
<td>1: add</td>
</tr>
<tr>
<td>2: subtract</td>
<td>2: subtract</td>
</tr>
<tr>
<td>3: multiply</td>
<td>3: multiply</td>
</tr>
<tr>
<td>4: system call</td>
<td>4: system call</td>
</tr>
<tr>
<td>5: jump</td>
<td>5: jump</td>
</tr>
<tr>
<td>6: jump if greater</td>
<td>6: jump if greater</td>
</tr>
<tr>
<td>7: jump and count</td>
<td>7: jump and count</td>
</tr>
<tr>
<td>8: jump and link</td>
<td>8: jump and link</td>
</tr>
<tr>
<td>9: load</td>
<td>9: load</td>
</tr>
<tr>
<td>A: store</td>
<td>A: store</td>
</tr>
<tr>
<td>B: load address</td>
<td>B: load address</td>
</tr>
<tr>
<td>C: xor</td>
<td>C: xor</td>
</tr>
<tr>
<td>D: and</td>
<td>D: and</td>
</tr>
<tr>
<td>E: shift right</td>
<td>E: shift right</td>
</tr>
<tr>
<td>F: shift left</td>
<td>F: shift left</td>
</tr>
</tbody>
</table>

Format 1

Register-register:
- Bits 12-15 encode opcode.
- Bits 8-11 encode destination register.
- Bits 4-7 encode source register A.
- Bits 0-3 encode source register B.

Ex: $1234_{16}$ means
- Add registers R3 and R4.
- Put result in register R2.
- $R2 \leftarrow R3 + R4$
Format 1: AND

Logic operations are BITWISE:
- \(1234_{16} \& FAD2_{16} = 1210_{16}\)

\[
\begin{array}{cccc}
0 & 0 & 0 & 1 \\
1_{16} & 2_{16} & 3_{16} & 4_{16}
\end{array}
\]

\&

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
F_{16} & A_{16} & D_{16} & 2_{16}
\end{array}
\]

= 

\[
\begin{array}{cccc}
0 & 0 & 0 & 1 \\
0_{16} & 1_{16} & 0_{16} & 0_{16}
\end{array}
\]

Format 1: XOR

Logic operations are BITWISE:
- \(1234_{16} \^{} FAD2_{16} = E8E6_{16}\)

\[
\begin{array}{cccc}
0 & 0 & 0 & 1 \\
1_{16} & 2_{16} & 3_{16} & 4_{16}
\end{array}
\]

\^{}

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
F_{16} & A_{16} & D_{16} & 2_{16}
\end{array}
\]

= 

\[
\begin{array}{cccc}
1 & 1 & 0 & 1 \\
E_{16} & 8_{16} & E_{16} & 6_{16}
\end{array}
\]

Other Logical Operations

Any logical operation can be implemented with AND and XOR.
- Recall: can build any logical operation with AND, OR, NOT.

Build OR from AND and XOR.
- \(x \oplus (x \& y)\)

Build NOT from XOR.
- \(1 \oplus x = x'\)

Format 2

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: \(B234\) means
- Load the value \(34_{16}\) into register R2.
  - R2 ← 0034

<table>
<thead>
<tr>
<th>Format 2 Instructions</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>jump</td>
<td>jump if greater</td>
<td>jump and count</td>
<td>jump and link</td>
<td>load</td>
<td>store</td>
<td>load address</td>
<td>shift left</td>
</tr>
<tr>
<td></td>
<td>shift right</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>opcode</th>
<th>dest</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_{16}</td>
<td>2_{16}</td>
<td>34_{16}</td>
</tr>
</tbody>
</table>
**Format 2**

Ex: B234 means
- Load the value 3416 into register R2.
- R2 ← 0034

Ex: 9234 means
- Load contents of memory location 3416 into register R2.
- R2 ← mem[34]

Ex: A234 means
- Store the contents of register R2 into memory location 3416.
- mem[34] ← R2

**Format 2 Instructions**

<table>
<thead>
<tr>
<th>Ex</th>
<th>Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>B234</td>
<td>Load the value 3416 into register R2.</td>
</tr>
<tr>
<td>9234</td>
<td>Load contents of memory location 3416 into register R2.</td>
</tr>
<tr>
<td>A234</td>
<td>Store the contents of register R2 into memory location 3416.</td>
</tr>
</tbody>
</table>

**Shift**

- Shift bits left or right.
- Pad with zeros.
- Ex (shift right): 923416 >> 716 = 012416
discard

**Sample TOY Program 0**

TOY code for C expression: t = b*b - 4*a*c.
- Suppose memory locations D0-D3 used to store data:
  - D0 stores a
  - D1 stores b
  - D2 stores c
  - D3 stores t
- Suppose memory locations 10-19 used to store instructions:
  - 10: 91D1 R1 ← b
  - 11: 3111 R1 ← b*b
  - 12: B204 R2 ← 4
  - 13: 93D0 R3 ← a
  - 14: 94D2 R4 ← c
  - 15: 3223 R2 ← 4*a
  - 16: 3224 R2 ← (4*a)*c
  - 17: 2112 R2 ← (b*b) - (4*a*c)
  - 18: A1D3 t ← (b*b - 4*a*c)
  - 19: 0000 halt

Set PC to 10; Press GO.
- TOY computes the value.

**Sample TOY Program 0**

Step-by-step trace:

<table>
<thead>
<tr>
<th>Ex</th>
<th>Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>10: 91D1</td>
<td>R1 ← b</td>
</tr>
<tr>
<td>11: 3111</td>
<td>R1 ← b*b</td>
</tr>
<tr>
<td>12: B204</td>
<td>R2 ← 4</td>
</tr>
<tr>
<td>13: 93D0</td>
<td>R3 ← a</td>
</tr>
<tr>
<td>14: 94D2</td>
<td>R4 ← c</td>
</tr>
<tr>
<td>15: 3223</td>
<td>R2 ← 4*a</td>
</tr>
<tr>
<td>16: 3224</td>
<td>R2 ← (4*a)*c</td>
</tr>
<tr>
<td>17: 2112</td>
<td>R2 ← (b<em>b) - (4</em>a*c)</td>
</tr>
<tr>
<td>18: A1D3</td>
<td>t ← (b<em>b - 4</em>a*c)</td>
</tr>
<tr>
<td>19: 0000</td>
<td>halt</td>
</tr>
</tbody>
</table>

C compiler produces code like this.

Stay tuned for what happens if:
- Result of subtraction is negative.
- Numbers are too big.
- We need the square root (!)
Sample TOY Program 1: More Arithmetic

TOY code to compute $1 + 2 + 3 + 4 + 5 + 6 = 21$.

- Suppose memory locations 10-1E used to store instructions:
  - 10: B001 B200 B101 1221 1110 1221 1110 1221 1110 1221 1110 0000 0000

Set PC to 10; Press GO.

TOY computes the value.

Sample TOY Program 1: More Arithmetic

Step-by-step trace.

<table>
<thead>
<tr>
<th>sum1.toy</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10: B001</td>
<td>R0 &lt;- 0001</td>
<td>0000</td>
</tr>
<tr>
<td>11: B200</td>
<td>R2 &lt;- 0000</td>
<td>0000</td>
</tr>
<tr>
<td>12: B001</td>
<td>R1 &lt;- 0001</td>
<td>0001</td>
</tr>
<tr>
<td>13: 1221</td>
<td>R2 &lt;- R2 + R1</td>
<td>0001</td>
</tr>
<tr>
<td>14: 1110</td>
<td>R1 &lt;- R1 + R0</td>
<td>0002</td>
</tr>
<tr>
<td>15: 1221</td>
<td>R2 &lt;- R2 + R1</td>
<td>0003</td>
</tr>
<tr>
<td>16: 1110</td>
<td>R1 &lt;- R1 + R0</td>
<td>0003</td>
</tr>
<tr>
<td>17: 1221</td>
<td>R2 &lt;- R2 + R1</td>
<td>0006</td>
</tr>
<tr>
<td>18: 1110</td>
<td>R1 &lt;- R1 + R0</td>
<td>0004</td>
</tr>
<tr>
<td>19: 1221</td>
<td>R2 &lt;- R2 + R1</td>
<td>000A</td>
</tr>
<tr>
<td>1A: 1110</td>
<td>R1 &lt;- R1 + R0</td>
<td>0005</td>
</tr>
<tr>
<td>1B: 1221</td>
<td>R2 &lt;- R2 + R1</td>
<td>0007</td>
</tr>
<tr>
<td>1C: 1110</td>
<td>R1 &lt;- R1 + R0</td>
<td>0006</td>
</tr>
<tr>
<td>1D: 1221</td>
<td>R2 &lt;- R2 + R1</td>
<td>0015</td>
</tr>
<tr>
<td>1E: 0000</td>
<td>halt</td>
<td></td>
</tr>
</tbody>
</table>

Sample TOY Program 1: Loop

TOY code to compute $N + (N - 1) + \ldots + 2 + 1$ for any value of $N$ loaded into R1 initially.

- Suppose memory locations 10-17 used to store instructions:
  - 10: B106 B200 B001 1221 2110 6113 0000 0000

Set PC to 10; Press GO.

TOY computes the value.

Sample TOY Program 1: Loop

Step-by-step trace.

<table>
<thead>
<tr>
<th>sum2.toy</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10: B106</td>
<td>R1 &lt;- 0006</td>
<td>0006</td>
</tr>
<tr>
<td>11: B200</td>
<td>R2 &lt;- 0000</td>
<td>0000</td>
</tr>
<tr>
<td>12: B001</td>
<td>R0 &lt;- 0001</td>
<td>0001</td>
</tr>
<tr>
<td>13: 1221</td>
<td>R2 &lt;- R2 + R1</td>
<td>0006</td>
</tr>
<tr>
<td>14: 2110</td>
<td>R1 &lt;- R1 - R0</td>
<td>0005</td>
</tr>
<tr>
<td>15: 6113</td>
<td>jump if (R1 &gt; 0)</td>
<td>0006</td>
</tr>
<tr>
<td>16: 2110</td>
<td>R1 &lt;- R1 - R0</td>
<td>0004</td>
</tr>
<tr>
<td>15: 6113</td>
<td>jump if (R1 &gt; 0)</td>
<td>000B</td>
</tr>
<tr>
<td>13: 1221</td>
<td>R2 &lt;- R2 + R1</td>
<td>0006</td>
</tr>
<tr>
<td>14: 2110</td>
<td>R1 &lt;- R1 - R0</td>
<td>0003</td>
</tr>
<tr>
<td>15: 6113</td>
<td>jump if (R1 &gt; 0)</td>
<td>000F</td>
</tr>
<tr>
<td>13: 1221</td>
<td>R2 &lt;- R2 + R1</td>
<td>0003</td>
</tr>
<tr>
<td>14: 2110</td>
<td>R1 &lt;- R1 - R0</td>
<td>0002</td>
</tr>
<tr>
<td>15: 6113</td>
<td>jump if (R1 &gt; 0)</td>
<td>0012</td>
</tr>
<tr>
<td>13: 1221</td>
<td>R2 &lt;- R2 + R1</td>
<td>0014</td>
</tr>
<tr>
<td>14: 2110</td>
<td>R1 &lt;- R1 - R0</td>
<td>0001</td>
</tr>
<tr>
<td>15: 6113</td>
<td>jump if (R1 &gt; 0)</td>
<td>0015</td>
</tr>
<tr>
<td>13: 1221</td>
<td>R2 &lt;- R2 + R1</td>
<td>0006</td>
</tr>
<tr>
<td>14: 2110</td>
<td>R1 &lt;- R1 - R0</td>
<td>0000</td>
</tr>
<tr>
<td>15: 6113</td>
<td>jump if (R1 &gt; 0)</td>
<td>0015</td>
</tr>
<tr>
<td>16: 0000</td>
<td>halt</td>
<td></td>
</tr>
</tbody>
</table>
Program 2: Horner’s Method

Problem:
- Evaluate $2x^3 + 3x^2 + 9x + 7$ at $x = 10$.
- Assume “data” stored in locations 30 - 34
  - $x$    a    b    c    d
  - 30: 000A 0002 0003 0009 0007 0000 0000 0000

First try:
- Compute $x^3$, multiply by a; compute $x^2$, multiply by b, …
  (cumbersome, inefficient)

Efficient algorithm (Horner’s method):
- Rewrite $ax^3 + bx^2 + cx + d$ as $( (a x + b) x + c ) x + d$.
- Does polynomial evaluation for arbitrary $x$.
- Many applications (e.g., convert from decimal to hex).
- One raison d’être for early machines.

Program 3: Bit Manipulation

Example 3.
- Suppose memory location D0 is used to store LFBSR value:
  - D0: 0684
- Suppose memory locations 10-1B used to store instructions:
  - 10: 92D0 93D0 B001 E203 D220 E30A D330 C323
  - 18: 92D0 F201 1223 A2D0 0000 0000 0000 0000

Set PC to 10.  Press GO.  What happens?
- TOY simulates 1 step of LFBSR.
A Little History

ENIAC. (Eckert and Mauchly, 1946)
- First general purpose electronic computer.
- 30 x 50 x 8.5 ft.
- 17,468 vacuum tubes.
- 300 multiplication per second.
- Conditional jumps, programmable.
  - code: set switches
  - data: punch cards
  - used to compute artillery firing tables

Basic Characteristics of TOY Machine

TOY is a general purpose computer.
- Sufficient power to perform and computation.
- Limited only by amount of memory (and time).

Stored-program computer. (von Neumann memo, 1944)
- Data and instructions encoded in binary.
- Data and instructions stored in SAME memory.
- Can change program (control) without rewiring.
  - immediate applications
  - profound implications
- EDSAC (Wilkes 1949).
  - first stored-program computer
Next Time

Other primitive data types.
  • Negative numbers.

More TOY programming.
  • Arrays.
  • Function calls.

Simulating a TOY machine in C.