Lecture A1: The TOY Machine

What is TOY?

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

Why study?
- Machine language programming.
  - how do C programs relate to computer?
  - still (a few) situations today where it is really necessary
- Computer architecture.
  - how is a computer put together?
  - how does it work?
- Simplified machine.
  - captures essence of real computers

Inside the Box

Input/Output: (Fill/LSB)
- Switches.
  - Input data and programs.
- Lights.
  - View data.

Computation: (XOR,Shift)
- Arithmetic-logic unit (ALU).
  - Manipulate data.

State: (Register)
- Registers.
  - Fastest form of storage.
  - Use as scratch space during computation.
  - 16 registers.
    - each stores 16 bits
  - Register 0 is always 0.

Program counter (PC).
- An extra 8-bit register.
- Keeps track of next instruction to be executed.

Memory.
- Store data and programs.
- 256 “words.”
  - TOY word is 16 bits
- FF for stdin / stdout.

Data and Programs Are 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: \(6375_{10} = 000110001101111_{2}\)

<table>
<thead>
<tr>
<th>Dec</th>
<th>Bin</th>
<th>Dec</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>

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

\[6375_{10} = +2^{12} +2^{11} +2^7 +2^6 +2^5 +2^3 +2^1 +2^0 = 4096 +2048 +128 +64 +32 +4 +2 +1\]
Shorthand Notation

Use hexadecimal (base 16) representation.

- Binary code, four bits at a time.

Ex: $6375_{10} = 000110011100111_{2} = 18E7_{16}$

<table>
<thead>
<tr>
<th>Dec</th>
<th>Bin</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>

$$6375_{10} = 1 \times 16^3 + 8 \times 16^2 + 14 \times 16^1 + 7 \times 16^0 = 4096 + 2048 + 224 + 7$$

Program and Data

Program:
- Sequence of instructions.

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

Data:
- 16-bit word (interpreted other way).

Program counter (PC):
- Stores memory address of "next instruction."

Machine "Core" Dump

EVERYTHING is encoded in binary.

- Integers.
- Machine instructions.
- Text.
- Reals.
- ...

Machine contents at a particular place and time.

- Record of what program has done.
- Completely determines what program will do.

TOY Instruction Set Architecture

TOY instruction set architecture (ISA).

- Interface that specifies behavior of machine.
- 16 register, 256 words of main memory, 16-bit words.
- 16 instructions.

Each instruction consists of 16 bits.

- Bits 12-15 encode one of 16 instruction types or opcodes.
- Bits 8-11 encode destination register $d$.
- Bits 0-7 encode:
  - Format 1: source registers $s$ and $t$.
  - Format 2: 8-bit memory address or constant.
**Load Address (a.k.a. Load Constant)**

Load address. (opcode 7)
- Loads an 8-bit integer into a register.
- Format 2.
- 7A04 means:
  - load the value 0004 into register A
  - $R[A] \leftarrow 0004$

```c
a = 4;
```

**Load, Store**

Load. (opcode 8)
- Loads the contents of some memory location into a register.
- 8A04 means:
  - load the contents of memory location 04 into register A
  - $R[A] \leftarrow \text{mem}[04]$

Store. (opcode 9)
- Opposite of load.
- Store the contents of a register into main memory.

**Add, Subtract**

Add. (opcode 1)
- Add contents of two registers and store sum in a third.
- 1CAB means:
  - add contents of registers A and B
  - put result in register C
  - $R[C] \leftarrow R[A] + R[B]$

Subtract. (opcode 2)
- Analogous to add.

**Using the TOY Machine: Input, Output**

To enter a program or data:
- Set 8 memory address switches.
- Set 16 data switches.
- Press LOAD.
  - data written into addressed word of memory

To view the results of a program:
- Set 8 memory address switches.
- Press LOOK.
  - contents of addressed word of memory appears in lights
**Using the TOY Machine: Run**

To run the program:
- Set 8 memory address switches to address of first instruction.
- Press the RUN or STEP button.
  - loads PC from address switches
  - repeats fetch-execute cycle until halt instruction

Fetch-execute cycle.
- FETCH.
  - get instruction from memory
- EXECUTE.
  - update PC
  - move data to or from memory, registers
  - perform calculations

**Branch Zero, Branch Positive**

Branch if zero. (opcode C)
- Changes PC depending of value of some register.
- Used to implement loops, if-else.

Multiply.
- Load in integers a and b, and store c = a \times b.
- Brute-force algorithm:
  - initialize c = 0
  - add b to c, a times
- Problems.
- Analogous.

**Step-By-Step Trace of multiply.toy**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10:</td>
<td>8A0A</td>
<td>mem[0A]</td>
<td>0003</td>
<td></td>
</tr>
<tr>
<td>11:</td>
<td>8B0B</td>
<td>mem[0B]</td>
<td>0007</td>
<td></td>
</tr>
<tr>
<td>12:</td>
<td>7C00</td>
<td>00</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>13:</td>
<td>7101</td>
<td>01</td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>14:</td>
<td>CA18</td>
<td>if (R[A] == 0) goto 18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:</td>
<td>1CCB</td>
<td>R[C] += R[B]</td>
<td>0002</td>
<td></td>
</tr>
<tr>
<td>16:</td>
<td>2AA1</td>
<td>R[A]--</td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>17:</td>
<td>C014</td>
<td>goto 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18:</td>
<td>CA18</td>
<td>if (R[A] == 0) goto 18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19:</td>
<td>1CCB</td>
<td>R[C] += R[B]</td>
<td>000E</td>
<td></td>
</tr>
<tr>
<td>20:</td>
<td>2AA1</td>
<td>R[A]--</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>21:</td>
<td>C014</td>
<td>goto 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22:</td>
<td>CA18</td>
<td>if (R[A] == 0) goto 18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:</td>
<td>1CCB</td>
<td>R[C] += R[B]</td>
<td>0015</td>
<td></td>
</tr>
<tr>
<td>24:</td>
<td>2AA1</td>
<td>R[A]--</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>25:</td>
<td>C014</td>
<td>goto 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26:</td>
<td>CA18</td>
<td>if (R[A] == 0) goto 18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27:</td>
<td>1CCB</td>
<td>R[C] += R[B]</td>
<td>0012</td>
<td></td>
</tr>
<tr>
<td>28:</td>
<td>2AA1</td>
<td>R[A]--</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>29:</td>
<td>C014</td>
<td>goto 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30:</td>
<td>CA18</td>
<td>if (R[A] == 0) goto 18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:</td>
<td>1CCB</td>
<td>R[C] += R[B]</td>
<td>0014</td>
<td></td>
</tr>
<tr>
<td>32:</td>
<td>2AA1</td>
<td>R[A]--</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>33:</td>
<td>C014</td>
<td>goto 14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Branch if positive. (opcode D)**

```c
int a = 7, b = 8;
int c = 0;
while (a != 0) {
    c += b;
    a--;
}
```
An Efficient Multiplication Algorithm

Inefficient multiply.
- Brute force multiplication algorithm loops $a$ times.
- In worst case, 65,535 additions!

"Grade-school" multiplication.
- Always 16 additions to multiply 16-bit integers.

```
Decimal | Binary
1 2 3 4 | 1 0 1 1
* 1 5 1 2 | * 1 1 0 1
2 4 6 8 | 1 0 1 1
1 2 3 4 | 0 0 0 0
6 1 7 0 | 1 0 1 1
1 2 3 4 | 1 0 1 1
0 1 8 6 5 8 0 8 | 1 0 0 0 1 1 1 1
```

Binary Multiplication

Grade school binary multiplication algorithm to compute $c = a \times b$.
- Initialize $c = 0$.
- Loop over $i$ bits of $b$.
  - if $b_i = 0$, do nothing
  - if $b_i = 1$, shift a left $i$ bits and add to $c$

Implement with built-in TOY shift instructions.

```
int c = 0;
for (i = 15; i >= 0; i--) {
  if( (b >> i) & 1 )
    c += (a << i);
}
```

Bitwise AND

Logical AND. (opcode B)
- Logic operations are BITWISE.
- $B_{235_{16}} \& 0001_{16} = 0001_{16}$

```
 x  y  AND
 0  0  0
 0  1  0
 1  0  0
 1  1  1
```

Shift Left

Shift left. (opcode 5)
- Move bits to the left, padding with zeros as needed.
- $1234_{16} \ll 7_{16} = 1600_{16}$

```
0 0 0 1 0 0 1 0 0 0 1 1 0 1 0 0
16 16 16 16
= 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0
16 16 16 16
```

```
0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0
16 16 16 16
= 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0
16 16 16 16
```
### Shift Right

**Shift right. (opcode 6)**
- Move bits to the right, padding with sign bit as needed.
- \(1234_{16} \gg 7_{16} = 0124_{16}\)

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1&lt;sub&gt;16&lt;/sub&gt;</td>
<td>2&lt;sub&gt;16&lt;/sub&gt;</td>
<td>3&lt;sub&gt;16&lt;/sub&gt;</td>
<td>4&lt;sub&gt;16&lt;/sub&gt;</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>
</tbody>
</table>

- **discard**
- **sign bit**
- **pad with 0's**

**>> \(7\) =**

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 0<sub>16</sub> | 1<sub>16</sub> | 2<sub>16</sub> | 4<sub>16</sub> |

### Binary Multiplication

**multiply-fast.toy**

```
OA: 0007 7  input a
OB: 0008 8  input b
10: 8A0A R[A] ← mem[OA]  
11: 8B0B R[B] ← mem[OB]  
12: 7101 R[1] ← 0001    always 1
14: 7C00 R[C] ← 0000    result
   C21D if (R[2] == 0) goto 1D while (i > 0) {
   15: 2221 R[2]-- i--  
   a << i  
   b >> i  
   \(b_i = i^{th} \text{ bit of } b\)
1A: C43A if (R[4] == 0) goto 1C  
   \(\text{if } b_i \text{ is } 1\)  
   1B: 1CC3 R[C] += R[3]  
   \(\text{add } a << i \text{ to sum}\)
   1C: C015 goto 15  
   }  
   1D: 9C0C mem[OC] ← R[C]
```

### 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 multiplications per second.
- Conditional jumps, programmable.
  - code: set switches
  - data: punch cards

### Basic Characteristics of TOY Machine

**TOY is a general-purpose computer.**
- Sufficient power to perform ANY 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
- Outgrowth of Turing's work.

All modern computers are general-purpose computers and have same (von Neumann) architecture.
Harvard vs. Princeton

Harvard architecture.
- Separate program and data memories.
- Can’t load game from disk (data) and execute (program).
- Used in some microcontrollers.

von Neumann architecture.
- Program and data stored in same memory.
- Used in almost all computers.

What’s the difference between Harvard and Princeton?

Lecture A1: Extra Slides

TOY Cheat Sheet

<table>
<thead>
<tr>
<th>#</th>
<th>Operation</th>
<th>Fmt</th>
<th>Pseudocode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>halt</td>
<td></td>
<td><code>exit(0)</code></td>
</tr>
<tr>
<td>1</td>
<td>add</td>
<td>1</td>
<td><code>R[d] ↔ R[s] + R[t]</code></td>
</tr>
<tr>
<td>2</td>
<td>subtract</td>
<td>1</td>
<td><code>R[d] ↔ R[s] - R[t]</code></td>
</tr>
<tr>
<td>3</td>
<td>and</td>
<td>1</td>
<td><code>R[d] ↔ R[s] &amp; R[t]</code></td>
</tr>
<tr>
<td>4</td>
<td>xor</td>
<td>1</td>
<td><code>R[d] ↔ R[s] ^ R[t]</code></td>
</tr>
<tr>
<td>5</td>
<td>shift left</td>
<td>1</td>
<td><code>R[d] ↔ R[s] &lt;&lt; R[t]</code></td>
</tr>
<tr>
<td>6</td>
<td>shift right</td>
<td>1</td>
<td><code>R[d] ↔ R[s] &gt;&gt; R[t]</code></td>
</tr>
<tr>
<td>7</td>
<td>load addr</td>
<td>2</td>
<td><code>R[d] ↔ addr</code></td>
</tr>
<tr>
<td>8</td>
<td>load</td>
<td>2</td>
<td><code>R[d] ↔ mem[addr]</code></td>
</tr>
<tr>
<td>9</td>
<td>store</td>
<td>2</td>
<td><code>mem[addr] ↔ R[d]</code></td>
</tr>
<tr>
<td>A</td>
<td>load indirect</td>
<td>1</td>
<td><code>R[d] ↔ mem[R[t]]</code></td>
</tr>
<tr>
<td>B</td>
<td>store indirect</td>
<td>1</td>
<td><code>mem[R[t]] ↔ R[d]</code></td>
</tr>
<tr>
<td>C</td>
<td>branch zero</td>
<td>2</td>
<td>if (R[d] == 0) pc ↔ addr</td>
</tr>
<tr>
<td>D</td>
<td>branch positive</td>
<td>2</td>
<td>if (R[d] &gt; 0) pc ↔ addr</td>
</tr>
<tr>
<td>E</td>
<td>jump register</td>
<td>2</td>
<td>pc ↔ R[d]</td>
</tr>
<tr>
<td>F</td>
<td>jump and link</td>
<td>2</td>
<td>R[d] ↔ pc; pc ↔ addr</td>
</tr>
</tbody>
</table>

Useful TOY "Idioms"

Jump absolute.
- Jump to a fixed memory address.
  - branch if zero with destination
  - register 0 is always 0

Register assignment.
- No instruction that transfers contents of one register into another.
- Pseudo-instruction that simulates assignment:
  - add with register 0 as one of two source registers

No-op.
- Instruction that does nothing.
- Plays the role of whitespace in C programs.
  - numerous other possibilities!