17. A Computing Machine

A TOY computing machine

TOY is an imaginary machine similar to:
• Ancient computers.
• Today’s smartphone processors.
• Countless other devices designed and built over the past 50 years.

Smartphone processor, 2010s

PDP-8, 1970s

Reasons to study TOY

Prepare to learn about computer architecture
• How does your computer’s processor work?
• What are its basic components?
• How do they interact?

Learn about machine-language programming.
• How do Java programs relate to your computer?
• Key to understanding Java references.
• Intellectual challenge of a new programming regime.
• Still necessary in some modern applications.

Learn fundamental abstractions that have informed processor design for decades.
Bits and words

Everything in TOY is encoded with a sequence of bits (value 0 or 1).
- Why? Easy to represent two states (on and off) in real world.
- Bits are organized in 16-bit sequences called words.

More convenient for humans: hexadecimal notation (base 16)
- 4 hex digits in each word.
- Convert to and from binary 4 bits at a time.

Inside the box

Components of TOY machine
- Memory
- Registers
- Arithmetic and logic unit (ALU)
- Program counter (PC)
- Instruction register (IR)

Arithmetic and logic unit (ALU)

ALU
- TOY’s computational engine
- A calculator, not a computer
- Hardware that implements all data-type operations
- How? Stay tuned for computer architecture lectures

Memory

Holds data and instructions
- 256 words
- 16 bits in each word
- Connected to registers
- Words are addressable

Use hexadecimal for addresses
- Number words from 00 to FF
- Think in hexadecimal
- Use array notation
- Example: M[2A] = C024

Table of 256 words completely specifies contents of memory.
Registers

- 16 words, addressable in hex from 0 to F (use names R[0] through R[15])
- Scratch space for calculations and data movement.
- Connected to memory and ALU
- By convention, R[0] is always 0. Often simplifies code (stay tuned)
  In our code, we often also keep 0001 in R[1].

Q. Why not just connect memory directly to ALU?
A. Too many different memory names (addresses).

Q. Why not just connect memory locations to one another?
A. Too many different connections.

Table of 16 words completely specifies contents of registers.

Program counter and instruction register

TOY operates by executing a sequence of instructions.

Critical abstractions in making this happen
- Program Counter (PC). Memory address of next instruction.
- Instruction Register (IR). Instruction being executed.

Fetch-increment-execute cycle
- Fetch: Get instruction from memory into IR.
- Increment: Update PC to point to next instruction.
- Execute: Move data to or from memory, change PC, or perform calculations, as specified by IR.

The state of the machine

Contents of memory, registers, and PC at a particular time
- Provide a record of what a program has done.
- Completely determines what the machine will do.

ALU and IR hold intermediate states of computation

Image sources:
- http://commons.wikimedia.org/wiki/File:Abacus_5.jpg
17. A Computing Machine

- Overview
- Data types
- Instructions
- Operating the machine
- Machine language programming

### TOY data type (original design): Unsigned integers

**Values.** 0 to \(2^{16} - 1\), encoded in binary (or, equivalently, hex).

<table>
<thead>
<tr>
<th>decimal</th>
<th>hex</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>65535</td>
<td>FFF</td>
<td>1111111111111111</td>
</tr>
</tbody>
</table>

**Example.** \(6375_{10}\).

**Operations.**
- Add.
- Subtract.
- Test if 0.

**Warning.** TOY ignores overflow.

### TOY data type (better design): two’s complement

**Values.** \(-2^{15} \text{ to } 2^{15} - 1\), encoded in 16-bit two’s complement.

<table>
<thead>
<tr>
<th>decimal</th>
<th>hex</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>-32767</td>
<td>7FFF</td>
<td>1011111111111111</td>
</tr>
<tr>
<td>-32766</td>
<td>7FFE</td>
<td>1011111111111110</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>32768</td>
<td>8000</td>
<td>0000000000000000</td>
</tr>
</tbody>
</table>

**Operations.**
- Add.
- Subtract.
- Test if positive, negative, or 0.

**16 bit two’s complement**
- 16-bit binary representation of \(x\) for positive \(x\).
- 16-bit binary representation of \(2^{16} - |x|\) for negative \(x\).

**Useful properties**
- Leading bit (bit 15) signifies sign.
- \(0000000000000000\) represents zero.
- Add/subtract is the same as for unsigned.

---

**Warning.** slight annoyance: one extra negative value
Two's complement: conversion

**To convert from decimal to two's complement**
- If greater than +32,767 or less than −32,768 report error.
- Convert to 16-bit binary.
- If not negative, done.
- If negative, *flip all bits and add 1*.

**Examples**

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Two's Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>+13</td>
<td>0000000000001101</td>
</tr>
<tr>
<td>−13</td>
<td>1111111111101110</td>
</tr>
<tr>
<td>+256</td>
<td>0000000000010000</td>
</tr>
<tr>
<td>−256</td>
<td>1111111111110000</td>
</tr>
</tbody>
</table>

**To convert from two's complement to decimal**
- If sign bit is 1, *flip all bits and add 1* and output minus sign.
- Convert to decimal.

**Examples**

<table>
<thead>
<tr>
<th>Two's Complement</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>+1</td>
</tr>
<tr>
<td>FFFF</td>
<td>−1</td>
</tr>
<tr>
<td>FF0D</td>
<td>−243</td>
</tr>
<tr>
<td>00F3</td>
<td>+243</td>
</tr>
</tbody>
</table>

**To add/subtract**
- Use same rules as for unsigned binary.
- (Still) ignore overflow.

**Examples**

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Two's Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>0000000000001101</td>
</tr>
<tr>
<td>12</td>
<td>0000000000001110</td>
</tr>
</tbody>
</table>

Overflow in two's complement

- 32,767 + 1 = 224 − 1
- 1000000000000000 = 32,768
- 8000 = 214 − 32,768
- −32,768... = −214... = −32,768...

---

TOY data type: Bitwise operations

**Operations**
- Bitwise AND.
- Bitwise XOR.
- Shift left.
- Shift right.

**AND**

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>x AND y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**XOR**

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>x XOR y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Shift left**

- Fill with 0s if leading bit is 0.
- Fill with 1s if leading bit is 1.

**Shift right**

- Fill with 0s if leading bit is 0.
- Fill with 1s if leading bit is 1.

Special note: Shift left/right operations also implement multiply/divide by powers of 2 for integers.
17. A Computing Machine

- Overview
- Data types
- Instructions
- Operating the machine
- Machine language programming

### TOY instructions

**ANY** 16-bit (4 hex digit) value defines a TOY instruction.

First hex digit specifies which instruction.

Each instruction changes machine state in a well-defined way.

<table>
<thead>
<tr>
<th>category</th>
<th>opcodes</th>
<th>implements</th>
<th>changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>operations</td>
<td>1 2 3 4 5 6</td>
<td>data-type operations</td>
<td>registers</td>
</tr>
<tr>
<td>data movement</td>
<td>7 8 9 AB</td>
<td>data moves between registers and memory</td>
<td>registers, memory</td>
</tr>
<tr>
<td>flow of control</td>
<td>0 C D E F</td>
<td>conditionals, loops, and functions</td>
<td>PC</td>
</tr>
</tbody>
</table>

**Encoding instructions**

ANY 16-bit (4 hex digit) value defines a TOY instruction.

Two different instruction formats

- **Type RR**: Opcode and 3 registers.
- **Type A**: Opcode, 1 register, and 1 memory address.

**Examples**

- **1CAB**: Add R[A] to R[B] and put result in R[C].
- **8A15**: Load into R[A] data from M[15].

<table>
<thead>
<tr>
<th>opcode</th>
<th>instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 RR</td>
<td>halt</td>
</tr>
<tr>
<td>1 RR</td>
<td>add</td>
</tr>
<tr>
<td>2 RR</td>
<td>subtract</td>
</tr>
<tr>
<td>3 RR</td>
<td>bitwise and</td>
</tr>
<tr>
<td>4 RR</td>
<td>bitwise xor</td>
</tr>
<tr>
<td>5 RR</td>
<td>shift left</td>
</tr>
<tr>
<td>6 RR</td>
<td>shift right</td>
</tr>
<tr>
<td>7 A</td>
<td>load address</td>
</tr>
<tr>
<td>8 A</td>
<td>load</td>
</tr>
<tr>
<td>9 A</td>
<td>store</td>
</tr>
<tr>
<td>A RR</td>
<td>load indirect</td>
</tr>
<tr>
<td>B RR</td>
<td>store indirect</td>
</tr>
<tr>
<td>C A</td>
<td>branch if zero</td>
</tr>
<tr>
<td>D A</td>
<td>branch if positive</td>
</tr>
<tr>
<td>E RR</td>
<td>jump register</td>
</tr>
<tr>
<td>F A</td>
<td>jump and link</td>
</tr>
</tbody>
</table>

**A TOY program**

Add two integers
- Load operands from memory into registers.
- Add the registers.
- Put result in memory.

- Load into R[A] data from M[15]
- Load into R[B] data from M[16]
- Add R[A] and R[B] and put result into R[C]
- Store R[C] into M[17]
- Halt

Q. How can you tell whether a word is an instruction?
A. If the PC has its address, it is an instruction!
Same program with different data

Add two integers
- Load operands from memory into registers.
- Add the registers.
- Put result in memory.

Q. How can you tell whether a word is data?
A. If it is added to another word, it is data!

Outside the box

User interface
- Switches.
- Lights.
- Control Buttons.

First step: Turn on the machine!
**Loading a program into memory**

To load an instruction
- Set 8 memory address switches.
- Set 16 data switches to instruction encoding.
- Press LOAD to load instruction from switches into addressed memory word.

**Loading instructions into memory**

To load an instruction
- Set 8 memory address switches.
- Set 16 data switches to instruction encoding.
- Press LOAD to load instruction from switches into addressed memory word.
Loading instructions into memory

To load an instruction
- Set 8 memory address switches.
- Set 16 data switches to instruction encoding.
- Press LOAD to load instruction from switches into addressed memory word.

Loading data into memory

To load data, use the same procedure as for instructions
- Set 8 memory address switches.
- Set 16 data switches to data encoding.
- Press LOAD to load data from switches into addressed memory word.

Looking at what's in the memory

To double check that you loaded the data correctly
- Set 8 memory address switches.
- Press LOOK to examine the addressed memory word.
Running a program

To run a program, set the address switches to the address of first instruction and press RUN.

[ data lights may flash, but all (and RUN light) go off when HALT instruction is reached ]

To see the output, set the address switches to the address of expected result and press LOOK.

To run the program again, enter different data and press RUN again.

Switches and lights

Q. Did people really program this way?

A. Yes!

17. A Computing Machine

- Overview
- Data types
- Instructions
- Operating the machine
- Machine language programming
Machine language programming

TOY instructions support the same basic programming constructs as Java.
- Primitive data types.
- Assignment statements.
- Conditional and loops.
- Arrays (next lecture).
- Standard input and output (next).

and can support advanced programming constructs, as well.
- Functions and libraries (see text).
- Linked structures (see text).

Conditionals and loops

To control the flow of instruction execution
- Test a register's value.
- Change the PC, depending on the value.

Example: Absolute value of R[A]

10 DA12 If R[A] > 0 set PC to 12 (skip 11)
11 2A0A Subtract R[A] from 0 and put result into R[A]
12 ...

Example: Typical while loop (assumes R[1] is 0001)

10 CA15 If R[A] is 0 set PC to 15
11 ...
12 ...
13 2AA1 Decrement R[A] by 1
14 C010 Set PC to 10
15 ...

Standard input and output

An immediate problem
- We're not going to be able to address real-world problems with just switches and lights for I/O.
- One solution: Paper tape.

Punched paper tape
- Encode 16-bit words in two 8-bit rows.
- To write a word, punch a hole for each 1.
- To read a word, shine a light behind the tape and sense the holes.

TOY mechanism
- Connect hardware to memory location FF.
- To write the contents of a register to stdout, store to FF.
- To read from stdin into a register, load from FF.
Flow of control and standard output example: Fibonacci numbers

Register trace

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>8</th>
<th>13</th>
<th>21</th>
<th>34</th>
<th>55</th>
<th>89</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td>89</td>
<td>144</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td>89</td>
<td>144</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
# Fibonacci numbers

```c
int a = 1;  
int b = 1;  
int i = N; 
while (i > 0) {
    std::cout << a << std::endl; 
    int c = a + b; 
    a = b; 
    b = c; 
    i = i - 1; 
}
```

TOY reference card

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operation</th>
<th>Format</th>
<th>Pseudo-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>halt</td>
<td>—</td>
<td>halt</td>
</tr>
<tr>
<td>1</td>
<td>add</td>
<td>RR</td>
<td>R[d] = R[x] + R[y]</td>
</tr>
<tr>
<td>2</td>
<td>subtract</td>
<td>RR</td>
<td>R[d] = R[x] - R[y]</td>
</tr>
<tr>
<td>3</td>
<td>bitwise and</td>
<td>RR</td>
<td>R[d] = R[x] &amp; R[y]</td>
</tr>
<tr>
<td>4</td>
<td>bitwise xor</td>
<td>RR</td>
<td>R[d] = R[x] ^ R[y]</td>
</tr>
<tr>
<td>5</td>
<td>shift left</td>
<td>RR</td>
<td>R[d] = R[x] &lt;&lt; R[y]</td>
</tr>
<tr>
<td>6</td>
<td>shift right</td>
<td>RR</td>
<td>R[d] = R[x] &gt;&gt; R[y]</td>
</tr>
<tr>
<td>7</td>
<td>load addr</td>
<td>RR</td>
<td>R[d] = add</td>
</tr>
<tr>
<td>8</td>
<td>load</td>
<td>RR</td>
<td>R[d] = M[addr]</td>
</tr>
<tr>
<td>9</td>
<td>store</td>
<td>RR</td>
<td>M[addr] = R[d]</td>
</tr>
<tr>
<td>A</td>
<td>load indirect</td>
<td>RR</td>
<td>R[d] = M[R[t]]</td>
</tr>
<tr>
<td>B</td>
<td>store indirect</td>
<td>RR</td>
<td>M[R[t]] = R[d]</td>
</tr>
<tr>
<td>C</td>
<td>branch zero</td>
<td>RR</td>
<td>if (R[d] == 0) PC = addr</td>
</tr>
<tr>
<td>D</td>
<td>branch positive</td>
<td>RR</td>
<td>if (R[d] &gt; 0) PC = addr</td>
</tr>
<tr>
<td>E</td>
<td>jump register</td>
<td>RR</td>
<td>PC = R[d]</td>
</tr>
<tr>
<td>F</td>
<td>jump and link</td>
<td>RR</td>
<td>R[d] = PC + 1; PC = addr</td>
</tr>
</tbody>
</table>

Pop quiz 1 on TOY

Q: What is the interpretation of 1A75 as a TOY instruction?

1A75 as a two's complement integer value?

Pop quiz 2 on TOY

Q: How does one flip all the bits in a TOY register?

0FFF as a TOY instruction?

OFFF as a two's complement integer value?
Pop quiz 3 on TOY

Q. What does the following TOY program leave in R[2]?

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>7 0 0 0</th>
<th>R[C] = 1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>7 1 0 1</td>
<td>R[1] = 1</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>7 2 0 1</td>
<td>R[2] = 1</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>2 C C 1</td>
<td>R[C] = R[C] - 1</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>D C 1 3</td>
<td>IF (R[C] &gt; 0) PC = 13</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0 0 0 0</td>
<td>HALT</td>
<td></td>
</tr>
</tbody>
</table>

TOY vs. your laptop

Two different computing machines
- Both implement basic data types, conditionals, loops, and other low-level constructs.
- Both can have arrays, functions, and other high-level constructs.
- Both have infinite input and output streams.

Q. Is 256 words enough to do anything useful?

A. Yes! (See book, and stay tuned for next lecture.)

A. Yes! It is a Turing Machine, with a read/write I/O device (see theory lectures).

OK, we definitely want a faster version with more memory when we can afford it...