11. A Computing Machine

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

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

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 computer?
• Key to understanding Java references.
• Still necessary in 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)
- PC and IR

**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.*

Table of 256 words *completely specifies* contents of memory.

**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.
Registers

- 16 words, addressable in hex from 0 to F (use names R0 through RF)
- Scratch space for calculations and data movement.
- Connected to memory and ALU
- By convention, R0 is always 0.

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

11. A Computing Machine

- Overview
- Data types
- Instructions
- Operating the machine
- Machine language programming
**TOY data type**

A **data type** is a set of values and a set of operations on those values.

**TOY’s data type** is 16-bit 2s complement integers.

**Two kinds of operations**
- Arithmetic.
- Bitwise.

All other types of data must be implemented with **software**
- 32-bit and 64-bit integers.
- 32-bit and 64-bit floating point values.
- Characters and strings.
- ...

*All values are represented in 16-bit words.*

---

**TOY data type (better design): 2s complement**

**Values.** $-2^{15}$ to $2^{15} - 1$, encoded in 16-bit 2s complement.

<table>
<thead>
<tr>
<th>decimal</th>
<th>hex</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>+32,767</td>
<td>7FF</td>
<td>0111111111111111</td>
</tr>
<tr>
<td>+32,766</td>
<td>7FE</td>
<td>0111111111111110</td>
</tr>
<tr>
<td>+32,765</td>
<td>7FD</td>
<td>0111111111111101</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>+3</td>
<td>0003</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>+2</td>
<td>0002</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>+1</td>
<td>0001</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>-1</td>
<td>FFFE</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>-2</td>
<td>FFE</td>
<td>1111111111111110</td>
</tr>
<tr>
<td>-3</td>
<td>FFD</td>
<td>1111111111111101</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>-32,766</td>
<td>8002</td>
<td>1000000000000000</td>
</tr>
<tr>
<td>-32,765</td>
<td>8001</td>
<td>1000000000000000</td>
</tr>
<tr>
<td>-32,764</td>
<td>8000</td>
<td>1000000000000000</td>
</tr>
</tbody>
</table>

---

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

**16 bit 2s complement**
- 16-bit binary representation of x for positive x.
- 16-bit binary representation of $2^{15} - |x|$ for negative x.

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

---

**2s complement: conversion**

**To convert from decimal to 2s 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.**

**To convert from 2s complement to decimal**
- If sign bit is 1, **flip all bits and add 1** and output minus sign.
- Convert to decimal.

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

---

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

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

**Example.** $6375_{10}$.

<table>
<thead>
<tr>
<th>decimal</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>6374</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>6375</td>
<td>1111111111111110</td>
</tr>
</tbody>
</table>

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

**Example.** $18E7 + 18E7 = 31CE$

<table>
<thead>
<tr>
<th>hex</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>10E4</td>
<td>0111111111111110</td>
</tr>
<tr>
<td>10E4</td>
<td>0111111111111110</td>
</tr>
<tr>
<td>0000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>+</td>
<td>0000 0000 0000 0000</td>
</tr>
<tr>
<td>=</td>
<td>0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

**Warning.** TOY ignores overflow.
Overflow in 2's complement

\[ 32,767_{10} = 2^{15} - 1 = 01111111111111_2 = 7FF_{16} \]
\[ \text{largest positive number} \]
\[ +1 = 0000000000000001 \]
\[ \text{+0001} \]
\[ = 1000000000000000 = 8000 \]
\[ \text{smallest negative number} \]
\[ -2^{15} = -32,768_{10} \]

TOY data type: Bitwise operations

\[
\begin{array}{cccccccc}
0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
\text{AND} & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
\text{XOR} & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
\text{Shift left} & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
\text{Shift right} & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\end{array}
\]

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

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

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 well-defined ways.

<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 A B</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 1: Opcode and 3 registers.

<table>
<thead>
<tr>
<th>opcode</th>
<th>destination Rd</th>
<th>source Rs</th>
<th>source Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 A B C D E</td>
<td>0 1 2 3 4 5 6 7 8 9 A B C D E</td>
<td>0 1 2 3 4 5 6 7 8 9 A B C D E</td>
<td></td>
</tr>
</tbody>
</table>

• Type 2: Opcode, 1 register, and 1 memory address.

<table>
<thead>
<tr>
<th>opcode</th>
<th>destination Rd</th>
<th>address ADDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 A B C D E</td>
<td>0 1 2 3 4 5 6 7 8 9 A B C D E</td>
<td></td>
</tr>
</tbody>
</table>

Examples

1 C A B  add RA to RB and put result in RC
8 B 0 1  load contents of memory location 01 into RB

opcode | instruction
---|----------------
0 1 | halt
1 1 | add
2 1 | subtract
3 1 | and
4 1 | xor
5 1 | shift left
6 1 | shift right
7 2 | load address
8 2 | load
9 2 | store
A 1 | load indirect
B 1 | store indirect
C 2 | branch if zero
D 2 | branch if positive
E 2 | jump register
F 2 | jump and link

A TOY program

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

Load into RA data from mem[01]
Load into RB data from mem[02]
Add RA and RB and put result into RC
Store RC into mem[03]
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.

PC

Memory

| 00 | F FF E |
| 01 | 0 0 0 5 |
| 02 | 0 0 0 3 |
| 03 | 0 0 0 0 |
| 04 | ... |

10 | 8 A 0 1 |
11 | 8 B 0 2 |
12 | 1 C A B |
13 | 9 C 0 3 |
14 | 0 0 0 0 |

Registers

| A | F FF E |
| B | 0 0 0 5 |
| C | 0 0 0 3 |
| ... |

RA ← mem[01]  
RB ← mem[02]  
RC ← RA + RB  
mem[03] ← RC  
Halt
Outside the box

User interface
• Switches.
• Lights.
• Control Buttons.

First step: Turn on the machine!

Outside the box

Loading data into memory

To load data
• 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.

Loading instructions into memory

Use the same procedure as for data
• 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

Use the same procedure as for data
- Set 8 memory address switches.
- Set 16 data switches to instruction encoding.
- Press LOAD to load instruction from switches into addressed memory word.

LOAD  LOOK  STEP  RUN  ON
ADDRESS
DATA

LOAD  LOOK  STEP  RUN  ON
ADDRESS
DATA

LOAD  LOOK  STEP  RUN  ON
ADDRESS
DATA

LOAD  LOOK  STEP  RUN  ON
ADDRESS
DATA

LOAD  LOOK  STEP  RUN  ON
ADDRESS
DATA

LOAD  LOOK  STEP  RUN  ON
ADDRESS
DATA
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.

Machine language programming

TOY instructions support the same basic programming constructs that you learned in Java.
- Primitive data types.
- Assignment statements.
- Conditionals and loops.
- Standard input and output (this section).
- Arrays (this section).

and can support advanced constructs, as well.
- Functions and libraries.
- Objects.

11. A Computing Machine

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

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 RA

```
10  DA12  if RA > 0 set PC to 12 (skip 11)
11  2A0A  Subtract RA from 0 (R0) and put result into RA
12  ...  
```

Example: Typical while loop (assumes R1 is 0001)

```
10  CA15  if RA = 0 set PC to 15
11  ...  
12  ...  
13  2AA1  Decrement RA by 1
14  C010  Set PC to 10
15  ...  
while (a != 0) {
  ...  
  ...  
  a--;  
}
To infinity and beyond!
**Standard input and output**

An immediate problem
- We can’t be using switches and lights all the time!
- One solution: Paper tape.

Need to bolt new I/O devices to the side of the machine.

**Flow control and standard output example: Fibonacci numbers**

<table>
<thead>
<tr>
<th>Memory</th>
<th>Register trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 A</td>
<td>C 1 1 2 3 5 8 13 21 34 55 89</td>
</tr>
<tr>
<td>01 00 1</td>
<td>A 9 7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>

To implement an array
- Keep items in an array contiguous starting at mem address a.
- Access a[i] at mem[a+i].

To access an array element, use indirection
- Keep array address in a register.
- Add index
- Indirect load/store uses contents of a register.

**Arrays**

Example: Indirect store

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 7680</td>
<td>Load the address 80 into R6</td>
<td>array starts at mem location 80</td>
</tr>
<tr>
<td>13 7800</td>
<td>Set RB to 0</td>
<td>b is the index</td>
</tr>
<tr>
<td>16 1568</td>
<td>RS ← R6 + RB</td>
<td>compute address of a[b]</td>
</tr>
<tr>
<td>17 BC05</td>
<td>mem[RS] ← RC</td>
<td>a[b] ← c</td>
</tr>
<tr>
<td>18 1BB1</td>
<td>RB ← RB + 1</td>
<td>increment b</td>
</tr>
</tbody>
</table>
Arrays example: Read an array from standard input

To implement an array
- Keep items in an array contiguous starting at mem location a.
- Access a[i] at mem[a+i].

```plaintext
PC
10    7101 R1 ← 1
11    8AFF RA ← N
12    7680 R6 ← 80
13    7B00 RB ← 0
14    CA1B if (RA ← 0) PC ← 1B
15    8CFF read RC from stdin
16    156B R5 ← R6 + RB
17    BC0S mem[Rs] ← RC
18    1BB1 RB ← RB + 1
19    2A41 RA ← RA − 1
20    C014 PC ← 14
```

Stay tuned.
Full trace in next lecture.

TOY reference card

<table>
<thead>
<tr>
<th>op code</th>
<th>operation</th>
<th>format</th>
<th>pseudo-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>HALT</td>
<td>HALT</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>add</td>
<td>R[d] ← R[s] + R[t]</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>subtract</td>
<td>R[d] ← R[s] − R[t]</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>and</td>
<td>R[d] ← R[s] &amp; R[t]</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>xor</td>
<td>R[d] ← R[s] ^ R[t]</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>shift left</td>
<td>R[d] ← R[s] &lt;&lt; R[t]</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>shift right</td>
<td>R[d] ← R[s] &gt;&gt; R[t]</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>load addr</td>
<td>2 R[d] ← ADDR</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>load</td>
<td>2 R[d] ← mem[ADDR]</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>store</td>
<td>2 mem[ADDR] ← R[d]</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>load indirect</td>
<td>1 R[d] ← mem[R[t]]</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>store indirect</td>
<td>1 mem[R[t]] ← R[d]</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>branch zero</td>
<td>2 if (R[d] = 0) PC ← ADDR</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>branch positive</td>
<td>2 if (R[d] &gt; 0) PC ← ADDR</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>jump register</td>
<td>2 PC ← R[d]</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>jump and link</td>
<td>2 R[d] ← PC; PC ← ADDR</td>
<td></td>
</tr>
</tbody>
</table>

Pop quiz 1 on TOY

Q. What is the interpretation of

1A75 as a TOY instruction?
0FFF as a TOY instruction?
8888 as a TOY instruction?

1A75 as a 2s complement integer value?
0FFF as a 2s complement integer value?
8888 as a 2s complement integer value? (Answer in base 16).
Pop quiz 2 on TOY

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

Pop quiz 3 on TOY

Q. What does the following TOY program leave in R2?

```
10 7 C 0 A  RC ← 1010
11 7 1 0 1  R1 ← 1
12 7 2 0 1  R2 ← 1
13 1 2 2 2  R2 ← R2 + R2
14 2 C C 1  RC ← RC - 1
15 D C 1 3  if (RC > 0) PC ← 13
16 0 0 0 0  HALT
```