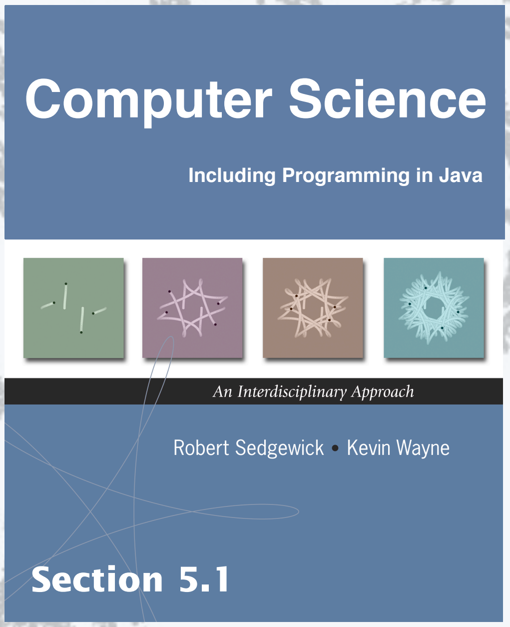
COMPUTER SCIENCE
SEDGEWICK / WAYNE


Computer Science
Including Programming in Java
An Interdisciplinary Approach
Robert Sedgewick • Kevin Wayne
Section 5.1

http://introcs.cs.princeton.edu

11. A Computing Machine

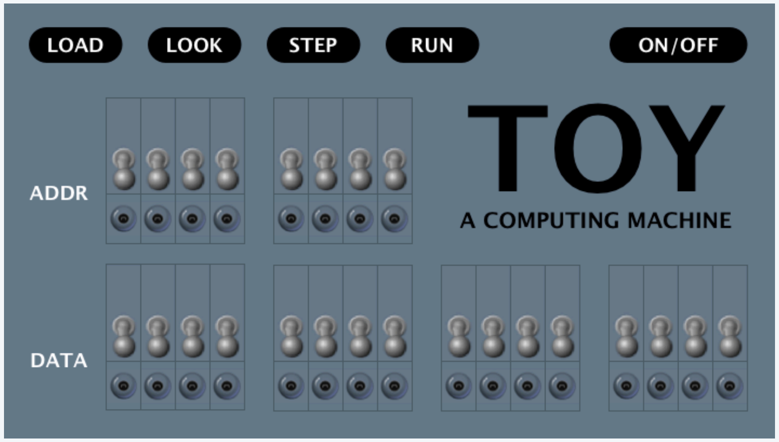
11. A Computing Machine

- **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

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 computer?
- Key to understanding Java references.
- Still necessary in modern applications.

multimedia, computer games, embedded devices, scientific computing,...

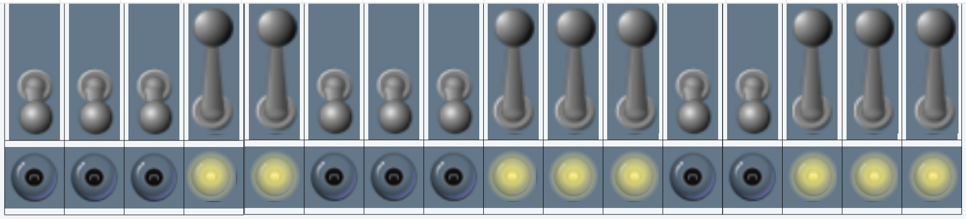
Learn fundamental abstractions that have informed processor design for decades.

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.*

0	0	0	1	1	0	0	0	1	1	1	0	0	1	1	1



More convenient for humans: *hexadecimal notation* (base 16)

- 4 *hex digits* in each word.
- Convert to and from binary 4 bits at a time.

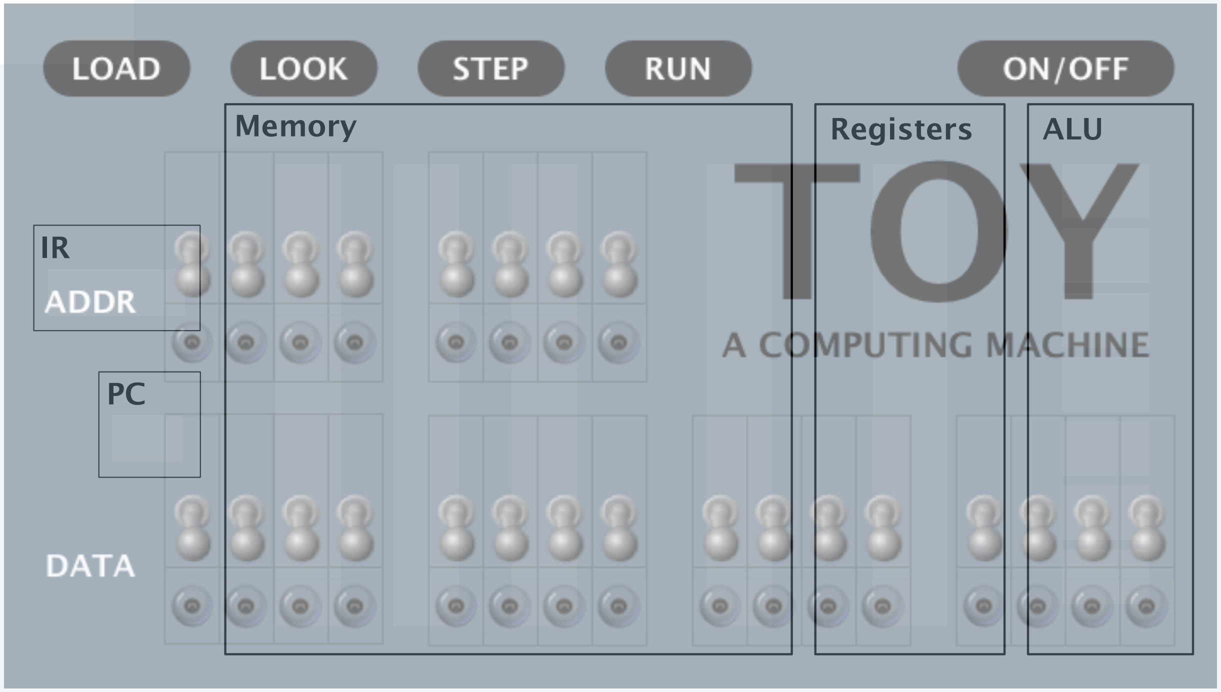
0	0	0	1	1	0	0	0	1	1	1	0	0	1	1	1
1				8				E				7			

binary	*hex*
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

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*.

Memory

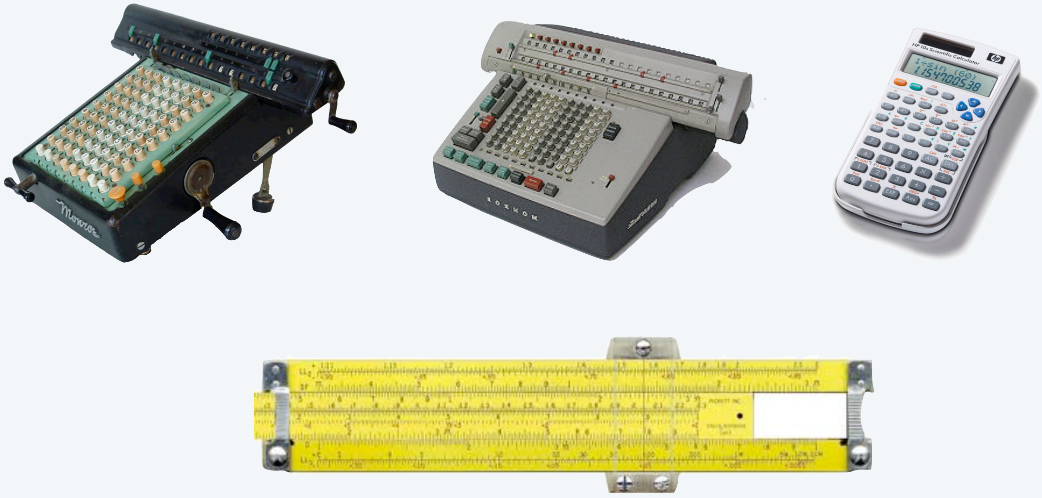
00	0000	10	8A01	20	7101			F0	F0F0
01	FFFE	11	8B02	21	8AFF			F1	0505
02	000D	12	1CAB	22	7680			F2	000D
03	0003	13	9C03	23	7B00			F3	1000
04	0001	14	0001	24	CA2B			F4	0101
05	0000	15	0010	25	8CFF			F5	0010
06	0000	16	0100	26	156B			F6	0001
07	0000	17	1000	27	BC05	. . .		F7	0010
08	0000	18	0100	28	2AA1			F8	0100
09	0000	19	0010	29	2BB1			F9	1000
0A	0000	1A	0001	2A	C024			FA	0100
0B	0000	1B	0010	2B	0000			FB	0010
0C	0000	1C	0100	2C	0000			FC	0001
0D	0000	1D	1000	2D	0000			FD	0010
0E	0000	1E	0100	2E	0000			FE	0100
0F	0000	1F	0010	2F	0000			FF	0100

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

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.

← often simplifies code (stay tuned)

In our code, we often also keep 0001 in R1.

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.

Registers

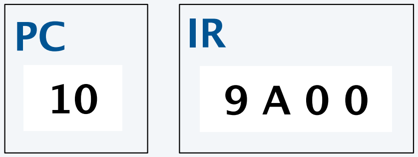
R0	0 0 0 0
R1	0 0 0 5
R2	0 0 0 8
R3	0 0 0 D
R4	0 0 0 1
R5	0 0 0 0
R6	F A C E
R7	0 0 0 0
R8	F 0 0 1
R9	0 0 0 0
RA	0 0 0 0
RB	0 0 0 0
RC	0 0 0 0
RD	0 0 0 0
RE	0 0 0 0
RF	0 0 0 0

Program counter and instruction register

TOY operates by executing a sequence of instructions.

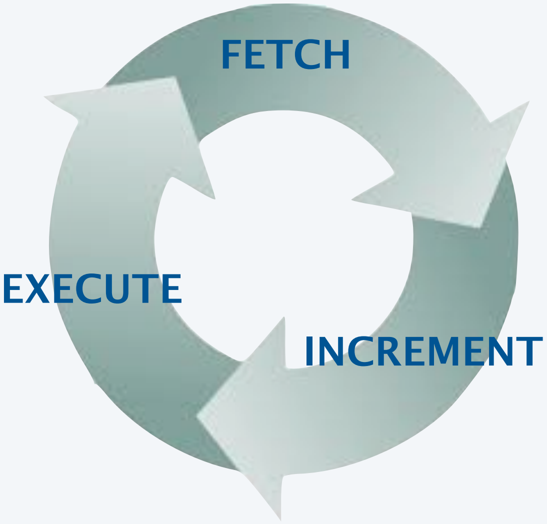
Critical abstractions in making this happen

- 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.

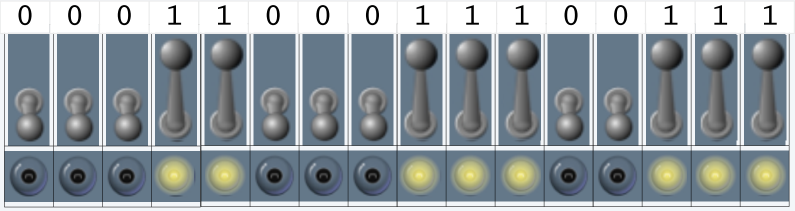
Two kinds of operations

- 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 (original design): Unsigned integers

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

Example. 6375_{10} .

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
binary	0	0	0	1	1	0	0	0	1	1	1	0	0	1	1	1
				2^{12}	$+2^{11}$				$+2^7$	$+2^6$	$+2^5$			$+2^2$	$+2^1$	$+2^0$

hex	1	8	E	7
	1×16^3	$+ 8 \times 16^2$	$+ 14 \times 16$	$+ 7$
	4096	+ 2048	+ 224	+ 7

Operations.

- Add.
- Subtract.
- Test if 0.

Example. 18E7 + 18E7 = 31CE

	0	0	0	1	1	0	0	0	1	1	1	0	0	1	1	1
+	0	0	0	1	1	0	0	0	1	1	1	0	0	1	1	1
=	0	0	1	1	0	0	0	1	1	1	0	0	1	1	1	0

Warning. TOY ignores overflow.

TOY data type (better design): 2s complement

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

includes negative integers!

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 $2^{16} - |x|$ for negative x.

Useful properties

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

decimal	*hex*	*binary*
+32,767	7FFF	0111111111111111
+32,766	7FFE	0111111111111110
+32,765	7FFD	0111111111111101
...		
+3	0003	
+2	0002	
+1	0001	
0	0000	
−1	FFFF	
−2	FFFE	
−3	FFFD	
...		
−32,766	8002	1000000000000010
−32,767	8001	1000000000000001
−32,768	8000	1000000000000000

slight annoyance: one extra negative value

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.

Examples

$+13_{10}$	0000000000001011	000D
-13_{10}	1111111111110101	FFF5
$+256_{10}$	0000000100000000	0100
-256_{10}	1111111100000000	FF00

Examples

0001	0000000000000001	1_{10}
FFFF	1111111111111111	-1_{10}
FF0D	1111111100001101	-243_{10}
00F3	0000000011110011	$+243_{10}$

Example

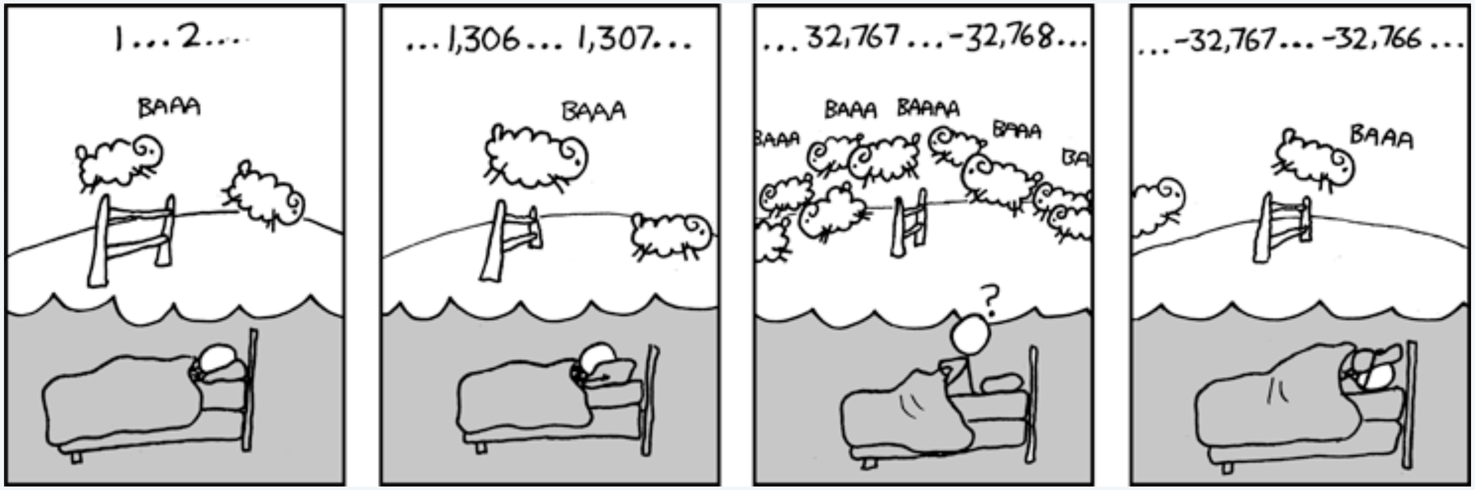
-256_{10}	1111111100000000	FF00
$+13_{10}$	+0000000000001011	+000D
$= -243_{10}$	=1111111100001101	=FF0D

Overflow in 2s complement

$32{,}767_{10} = 2^{15} - 1$	0111111111111111	7FFF	
+1	+ 0000000000000001	+ 0001	
	= 1000000000000000	= 8000	$= -2^{15} = -32{,}768_{10}$

largest (positive) number

smallest (negative) number



http://xkcd.com/571/

TOY data type: Bitwise operations

Operations

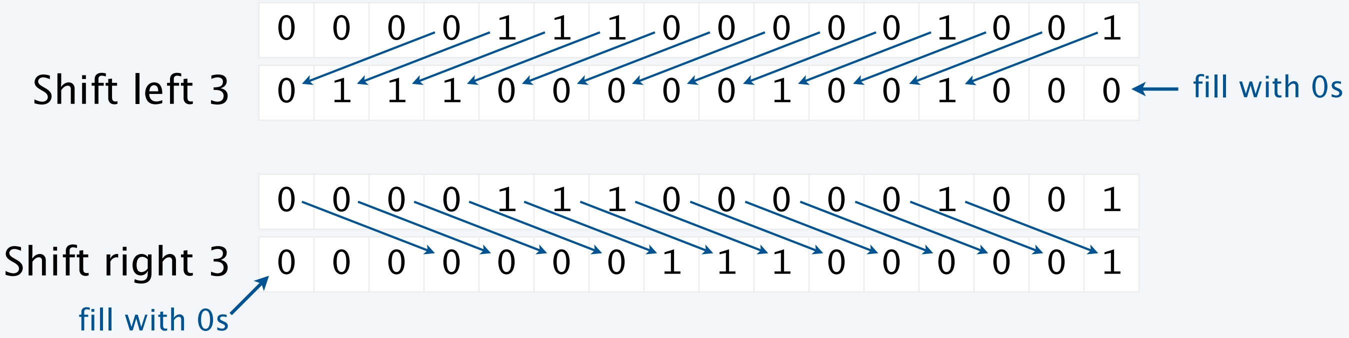
- Bitwise AND.
- Bitwise XOR.
- Shift left.
- Shift right.

	0	1	0	1	1	0	0	1	0	1	0	0	1	0	0	0
AND	0	0	0	1	1	1	1	1	0	0	0	0	0	1	0	1
=	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0

x	*y*	*x* AND *y*
0	0	0
0	1	0
1	0	0
1	1	1

	0	1	0	1	1	0	0	1	0	1	0	0	1	0	0	0
XOR	0	0	0	1	1	1	1	1	0	0	0	0	0	1	0	1
=	0	1	0	0	0	1	1	0	0	1	0	0	1	1	0	1

x	*y*	*x* XOR *y*
0	0	0
0	1	1
1	0	1
1	1	0



	0	0	0	0	1	1	1	0	0	0	0	0	1	0	0	1
Shift left 3	0	1	1	1	0	0	0	0	0	1	0	0	1	0	0	0

fill with 0s

	0	0	0	0	1	1	1	0	0	0	0	0	1	0	0	1
Shift right 3	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	1

fill with 0s

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

shift right fills with 1s if leading bit is 1

COMPUTER SCIENCE
SEDGEWICK / WAYNE

11. 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 well-defined ways.

category	*opcodes*	*implements*	*changes*
operations	**1 2 3 4 5 6**	data-type operations	registers
data movement	**7 8 9 A B**	data moves between registers and memory	registers, memory
flow of control	**0 C D E F**	conditionals, loops, and functions	PC

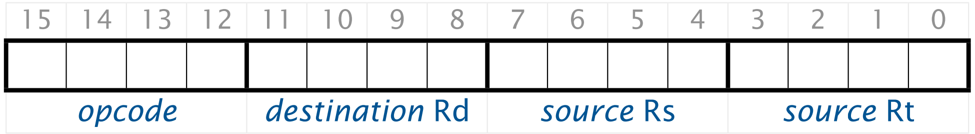
opcode	*instruction*
0	halt
1	add
2	subtract
3	and
4	xor
5	shift left
6	shift right
7	load address
8	load
9	store
A	load indirect
B	store indirect
C	branch if zero
D	branch if positive
E	jump register
F	jump and link

Encoding instructions

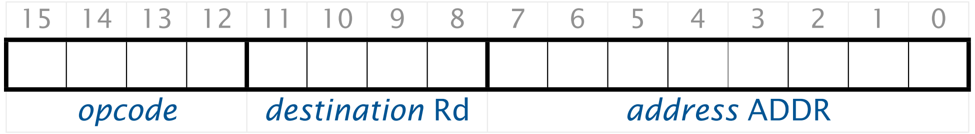
ANY 16-bit (4 hex digit) value defines a TOY instruction.

Two different instruction formats

- Type 1: Opcode and 3 registers.



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



Examples

1CAB	*add* RA *to* RB *and put result in* RC
8B01	*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.

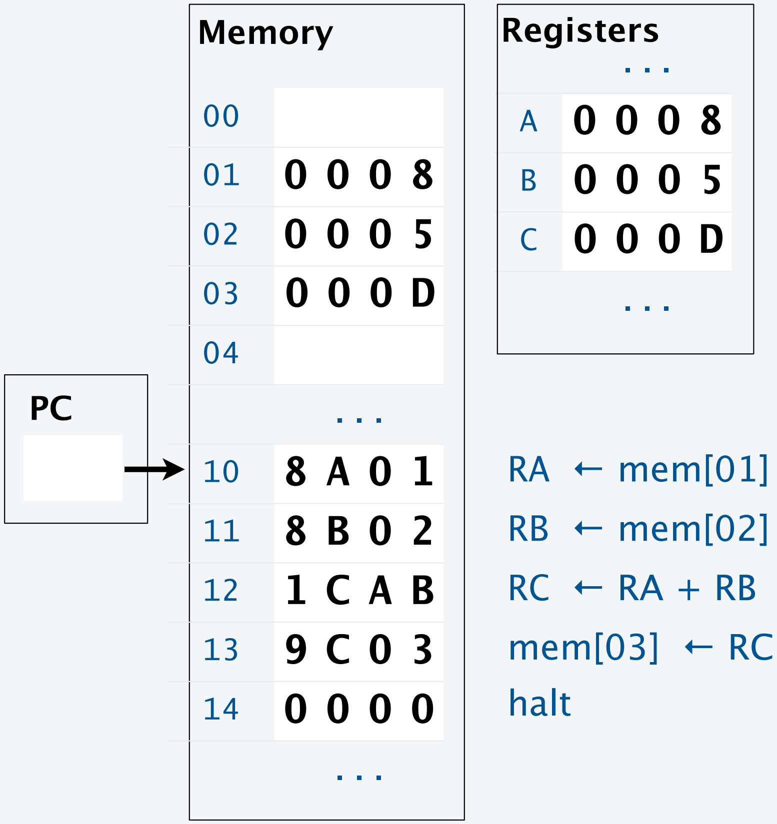
Load into RA data from `mem[01]`

Load into RB data from `mem[02]`

Add RA and RB and put result into RC

Store RC into `mem[03]`

Halt



PC

Memory

00		
01	0 0 0 8	
02	0 0 0 5	
03	0 0 0 D	
04		
	. . .	
10	8 A 0 1	RA ← mem[01]
11	8 B 0 2	RB ← mem[02]
12	1 C A B	RC ← RA + RB
13	9 C 0 3	mem[03] ← RC
14	0 0 0 0	halt
	. . .	

Registers

	. . .
A	0 0 0 8
B	0 0 0 5
C	0 0 0 D
	. . .

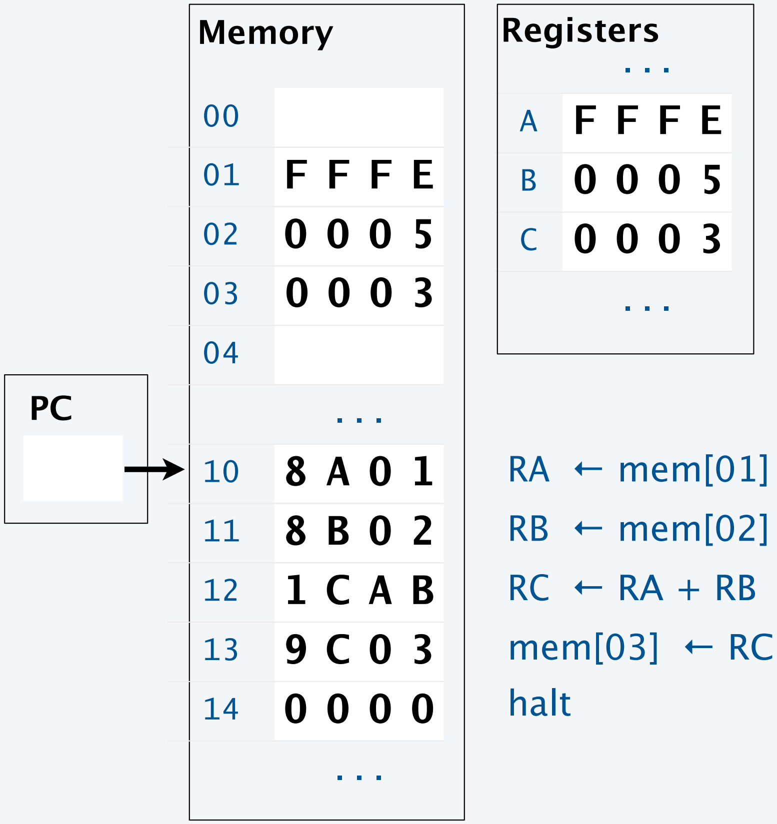
Q. How can you tell whether a word is an instruction?

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	
01	F F F E
02	0 0 0 5
03	0 0 0 3
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 F F E
B	0 0 0 5
C	0 0 0 3
	. . .

RA ← mem[01]

RB ← mem[02]

RC ← RA + RB

mem[03] ← RC

halt

11. A Computing Machine

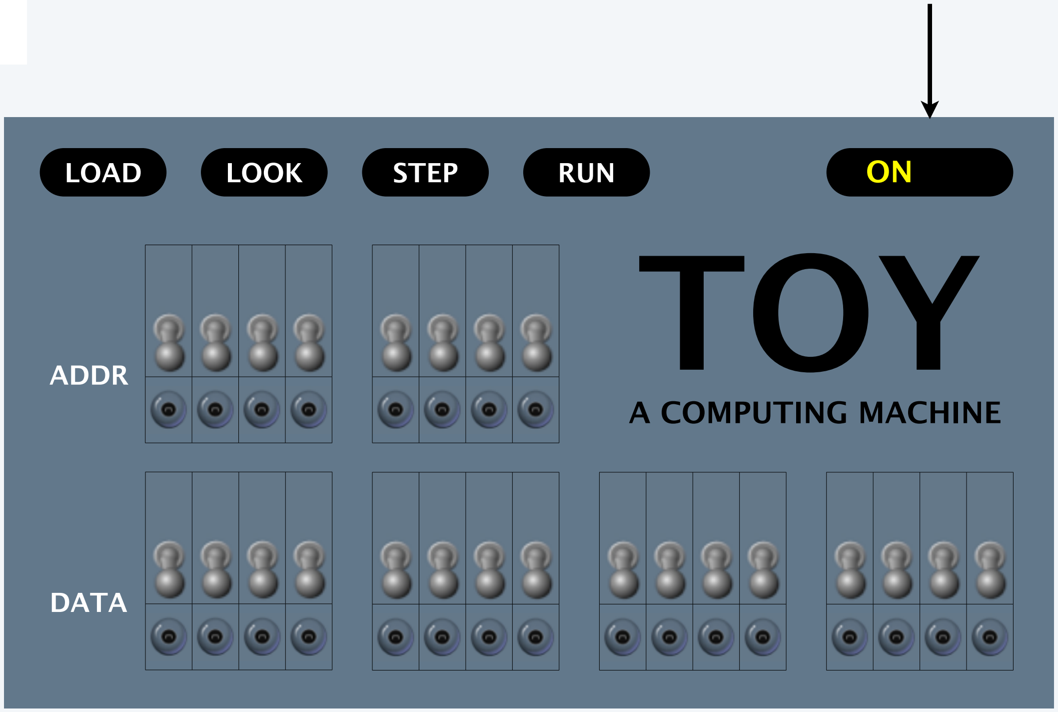
- Overview
- Data types
- Instructions
- **Operating the machine**
- Machine language programming

Outside the box

User interface

- Switches.
- Lights.
- Control Buttons.

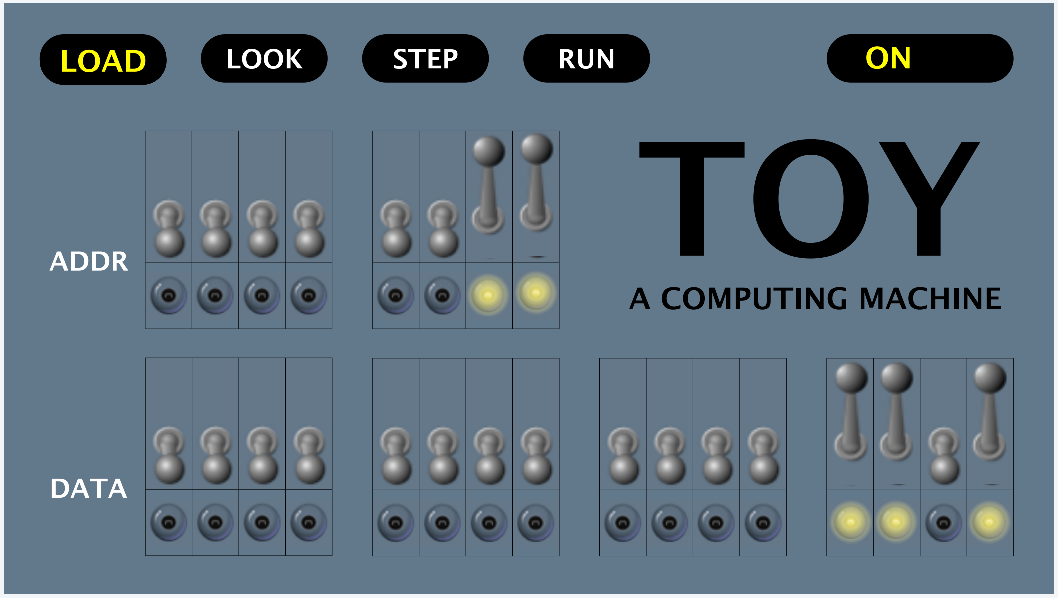
First step: Turn on the machine!

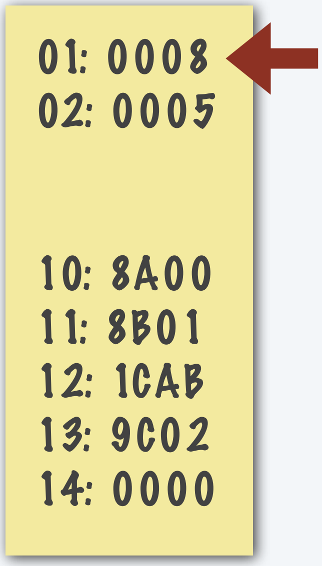


Loading data into memory

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.

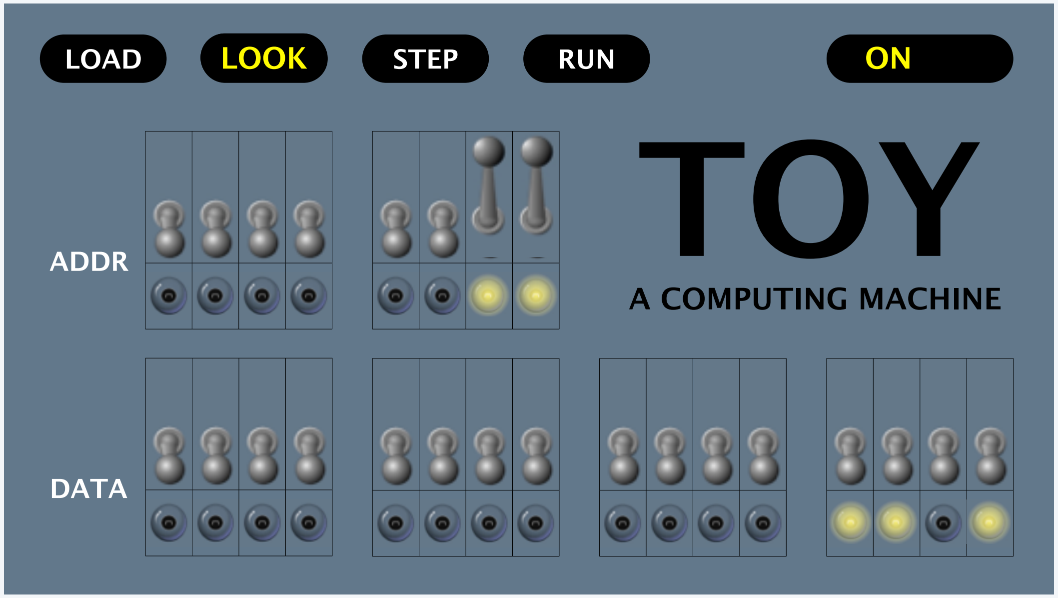




Looking at what's in the memory

To double check that you loaded the data correctly

- Set 8 memory address switches.
- Press LOOK to examine the addressed memory word.



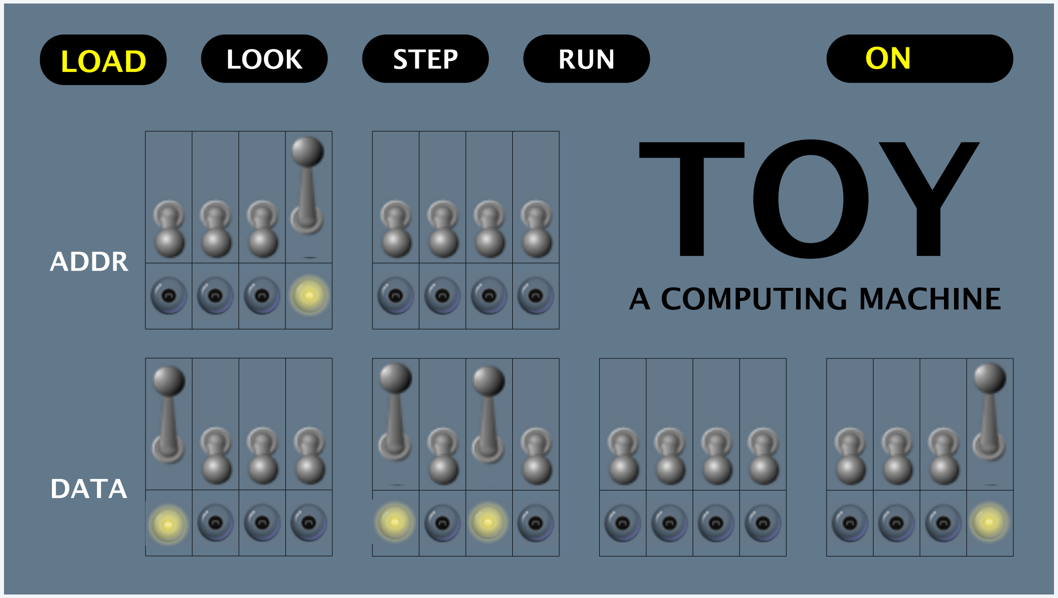
01: 0008
02: 0005

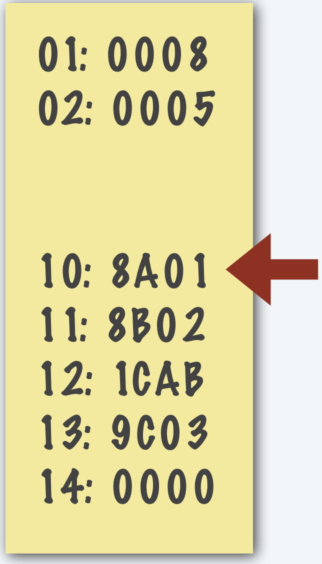
10: 8A01
11: 8B02
12: 1CAB
13: 9C03
14: 0000

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.

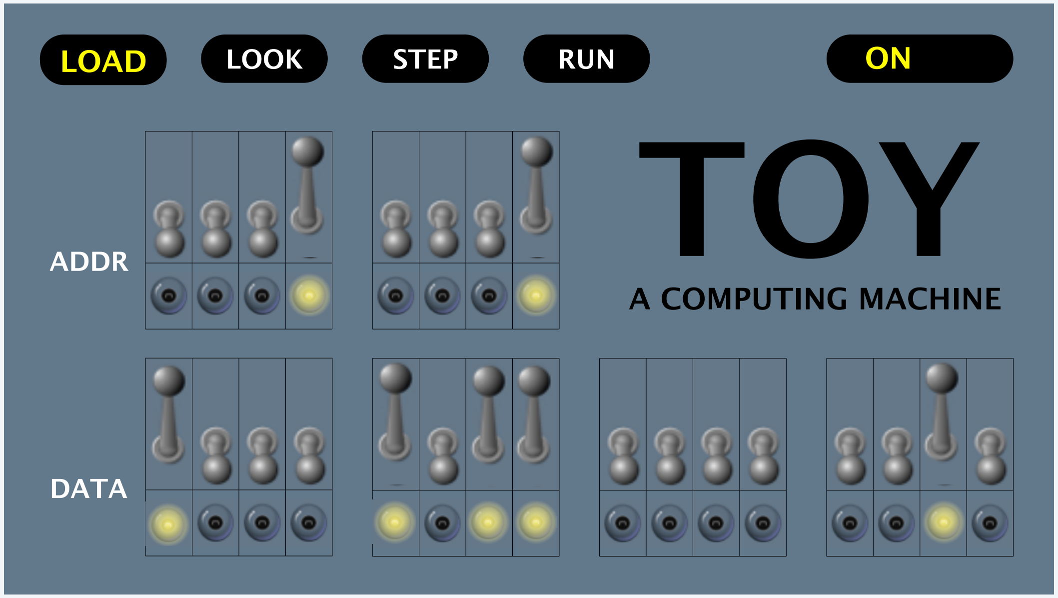




Loading instructions into memory

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.

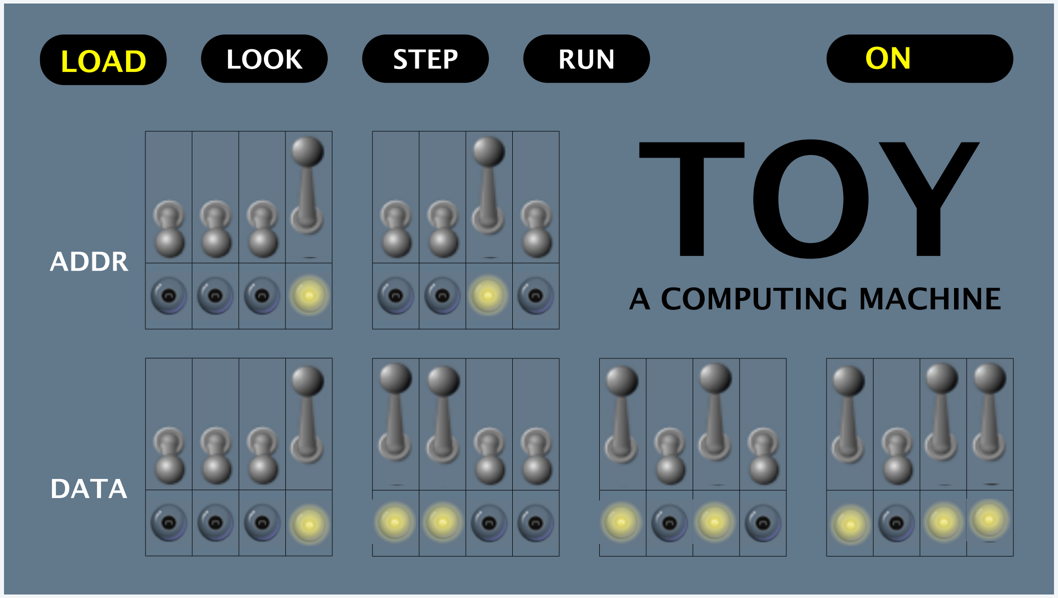


```
01: 0008
02: 0005


10: 8A01
11: 8B02
12: 1CAB
13: 9C03
14: 0000
```

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.



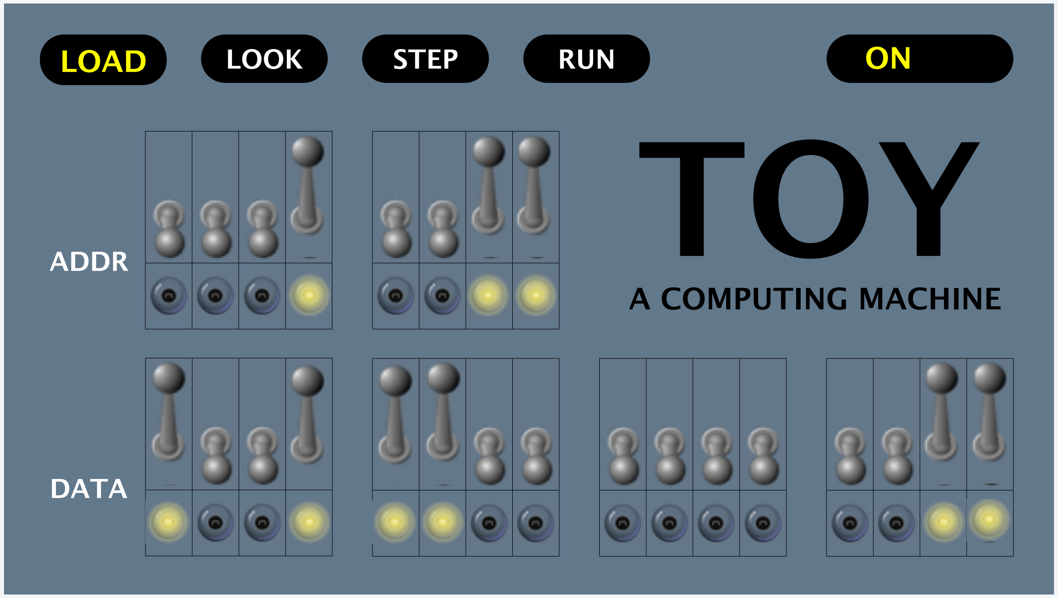
01: 0008
02: 0005

10: 8A01
11: 8B02
12: 1CAB
13: 9C03
14: 0000

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.

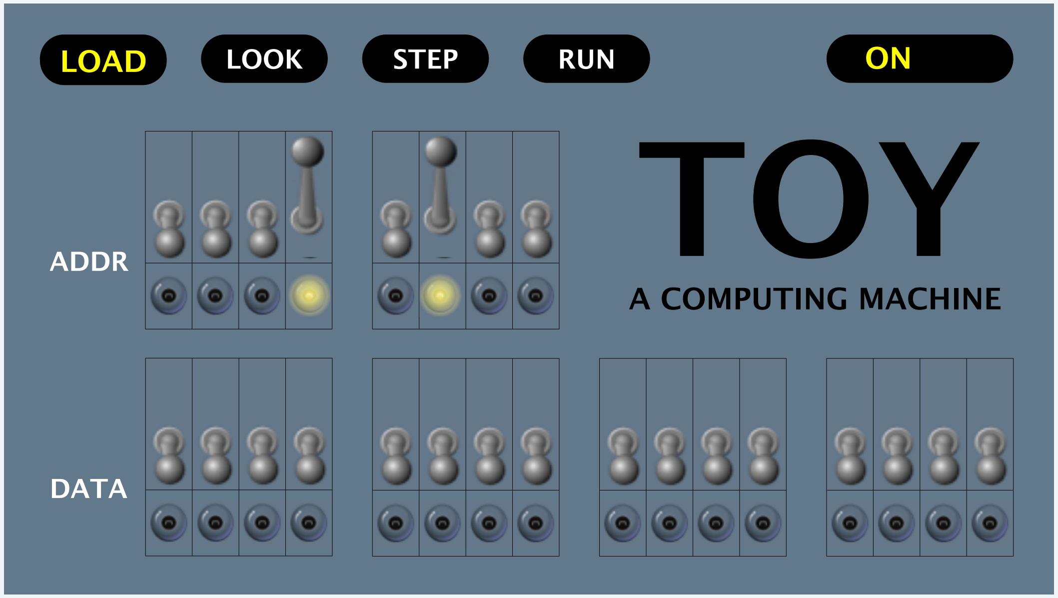


```
01: 0008
02: 0005


10: 8A01
11: 8B02
12: 1CAB
13: 9C03
14: 0000
```

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.



01: 0008
02: 0005

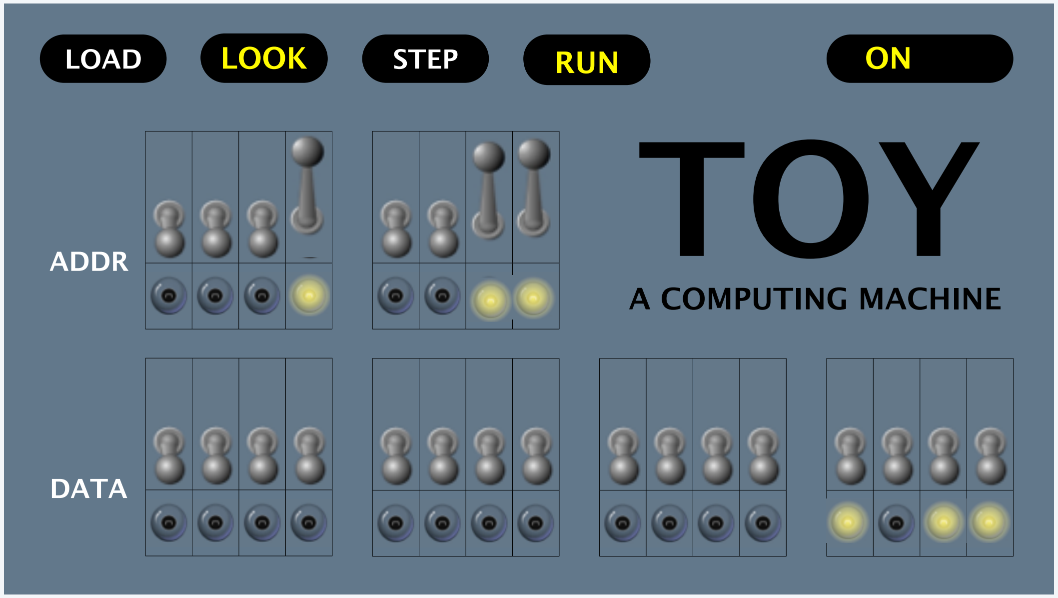
10: 8A01
11: 8B02
12: 1CAB
13: 9C03
14: 0000

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.



```
01: 0008
02: 0005
03: 000D

10: 8A01
11: 8B02
12: 1CAB
13: 9C03
14: 0000
```

11. A Computing Machine

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

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.

Conditionals and loops

To control the flow of instruction execution

- Test a register's value.
- Change the PC, depending on the value.

opcode	*instruction*
C	branch if zero
D	branch if positive

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	**...**	

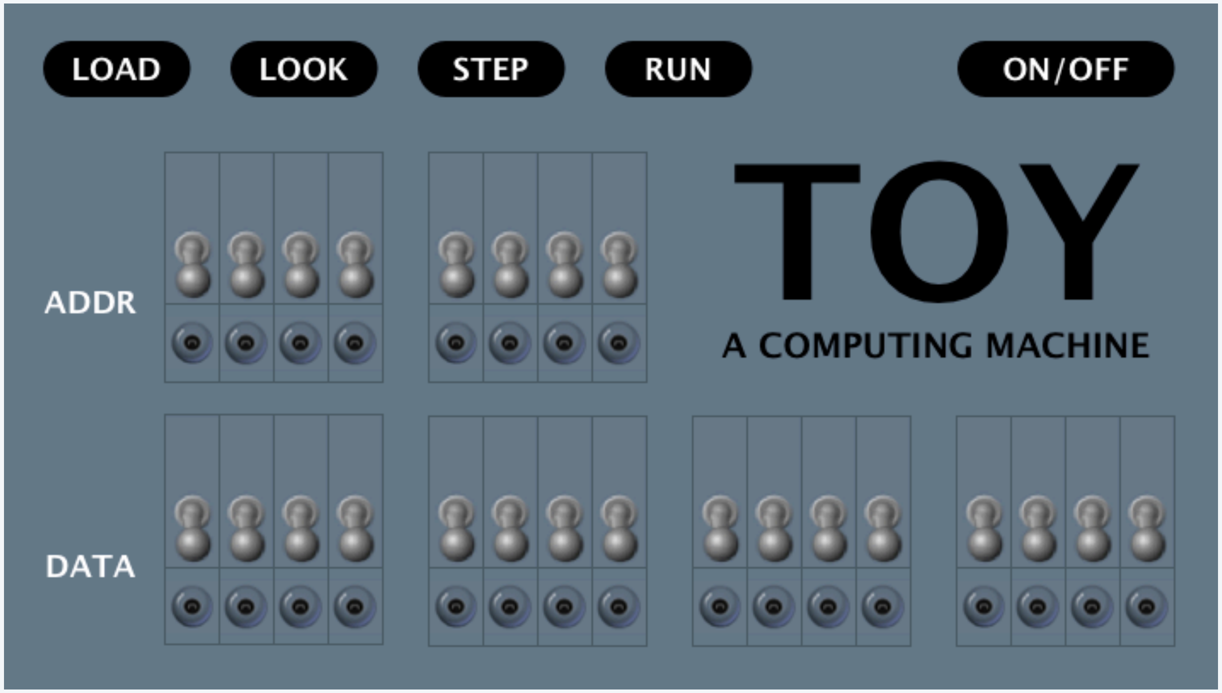
```
while (a != 0) {
  ...
  ...
    a--;
}
```

To infinity and beyond!

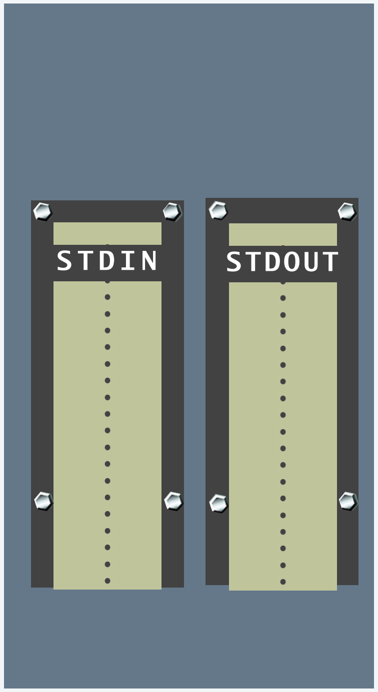
Standard input and output

An immediate problem

- We can't be using switches and lights all the time!
- One solution: Paper tape.



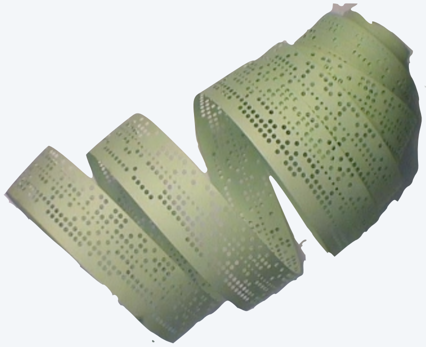
Need to bolt new I/O devices to the side of the machine.

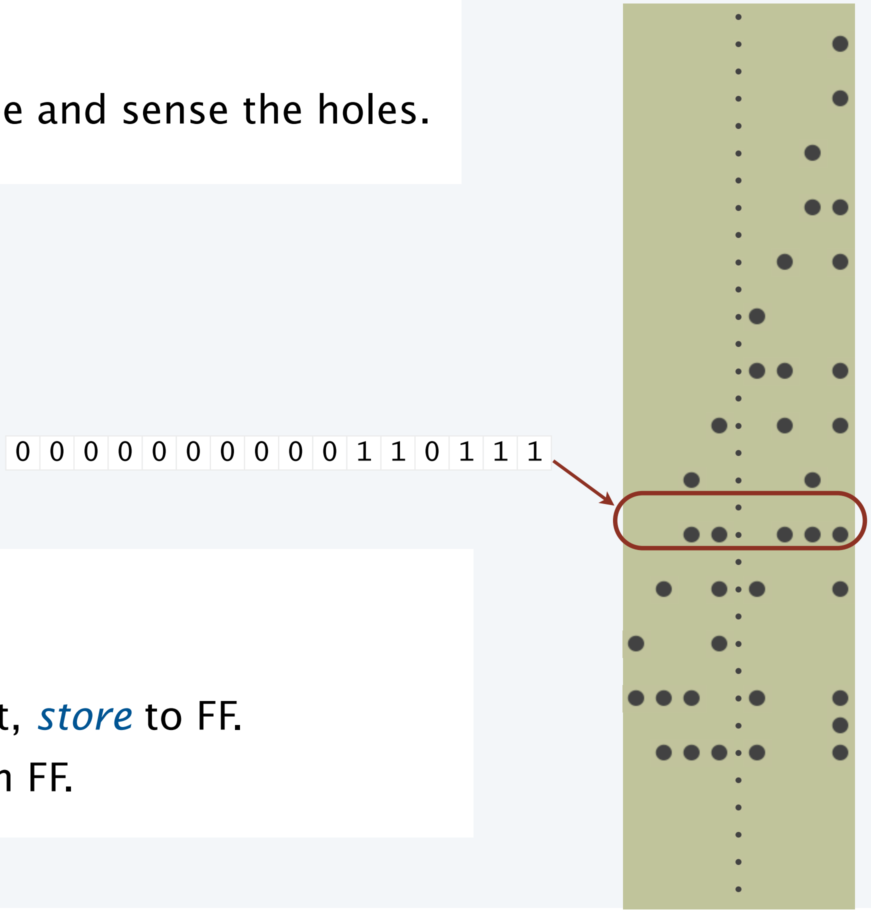


Standard input and output

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.





0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	1

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 control and standard output example: Fibonacci numbers

N	*F_N*
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55

Memory

PC → 10

00	**0 0 0 A**	
01	**0 0 0 1**	
02		
	...	
10	**8 1 0 1**	R1 ← 1
11	**8 C 0 1**	RC ← 1
12	**2 B B B**	RB ← 0
13	**8 A 0 0**	RA ← mem[00]
14	**C A 1 A**	if (RA == 0) PC ← 1A
15	**9 C F F**	write RC to stdout
16	**1 C B C**	RC ← RB + RC
17	**2 B C B**	RB ← RC - RB
18	**2 A A 1**	RA ← RA - 1
19	**C 0 1 4**	PC ← 14
1A	**0 0 0 0**	halt
	...	

Register trace

C	1	1	2	3	5	8	13	21	34	55	89
B	0	1	1	2	3	5	8	13	21	34	55
A	A	9	8	7	6	5	4	3	2	1	0

```
int c = 1;
int b = 0;
int a = N;
while (a != 0) {
   StdOut.print(c);
   c = b + c;
   b = c - b;
   a--;
}
```

Arrays

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.

opcode	*instruction*
7	load address
A	load indirect
B	store indirect

Example: Indirect store

12	**7680**	Load the address 80 into R6	array starts at mem location 80
13	**7B00**	Set RB to 0	b is the index
...			
16	**156B**	`R5 ← R6 + RB`	`compute address of a[b]`
17	**BC05**	`mem[R5] ← RC`	`a[b] ← c`
18	**1BB1**	`RB ← RB + 1`	increment b
...			

Array of length 11

80	**0000**
81	**0001**
82	**0001**
83	**0002**
84	**0003**
85	**0005**
86	**0008**
87	**000D**
88	**0015**
89	**0022**
8A	**0037**

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]`.

PC →

10	7 1 0 1	R1 ← 1
11	8 A F F	RA ← N
12	7 6 8 0	R6 ← 80
13	7 B 0 0	RB ← 0
14	C A 1 B	if (RA == 0) PC ← 1B
15	8 C F F	read RC from stdin
16	1 5 6 B	R5 ← R6 + RB
17	B C 0 5	mem[R5] ← RC
18	1 B B 1	RB ← RB + 1
19	2 A A 1	RA ← RA - 1
1A	C 0 1 4	PC ← 14
1B		[begin array processing code]

. . .

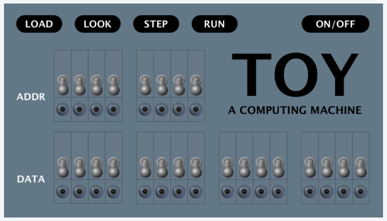
```
int a = StdIn.read();
arr = new int[];
int b = 0;
while (a != 0) {
   int c = StdIn.read();

   arr[b] = c;
   b++;
   a--;
}
```

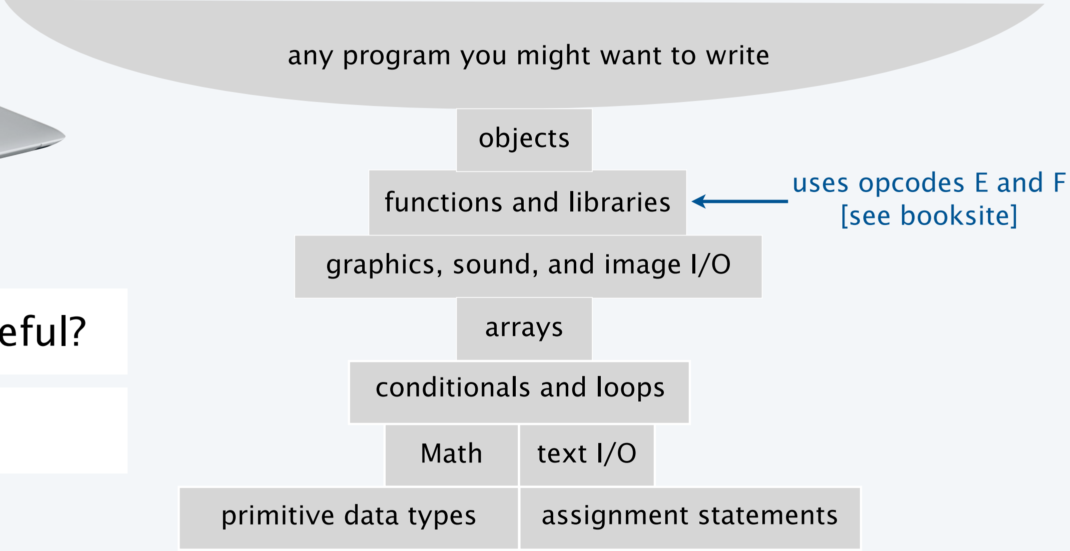
Stay tuned.
Full trace in next lecture.

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.





any program you might want to write

objects

functions and libraries ← uses opcodes E and F [see booksite]

graphics, sound, and image I/O

arrays

conditionals and loops

Math | text I/O

primitive data types | assignment statements

Q. Is 256 words enough to do anything useful?

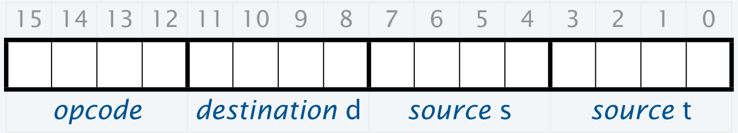
A. Yes! (Stay tuned for next lecture.)

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

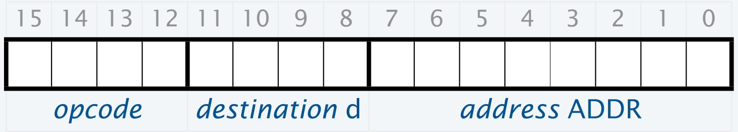
TOY reference card

opcode	*operation*	*format*	*pseudo-code*
0	HALT	1	`HALT`
1	add	1	`R[d] ← R[s] + R[t]`
2	subtract	1	`R[d] ← R[s] - R[t]`
3	and	1	`R[d] ← R[s] & R[t]`
4	xor	1	`R[d] ← R[s] ^ R[t]`
5	shift left	1	`R[d] ← R[s] << R[t]`
6	shift right	1	`R[d] ← R[s] >> R[t]`
7	load addr	2	`R[d] ← ADDR`
8	load	2	`R[d] ← mem[ADDR]`
9	store	2	`mem[ADDR] ← R[d]`
A	load indirect	1	`R[d] ← mem[R[t]]`
B	store indirect	1	`mem[R[t]] ← R[d]`
C	branch zero	2	`if (R[d] == 0) PC ← ADDR`
D	branch positive	2	`if (R[d] > 0) PC ← ADDR`
E	jump register	2	`PC ← R[d]`
F	jump and link	2	`R[d] ← PC; PC ← ADDR`

Format 1



Format 2



ZERO R0 is always 0.

STANDARD INPUT Load from FF.

STANDARD OUTPUT Store to FF.

Pop quiz 1 on TOY

Q. What is the interpretation of

1A75 as a TOY instruction?

1A75 as a 2s complement integer value?

0FFF as a TOY instruction?

0FFF as a 2s complement integer value?

8888 as a TOY instruction?

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 ?

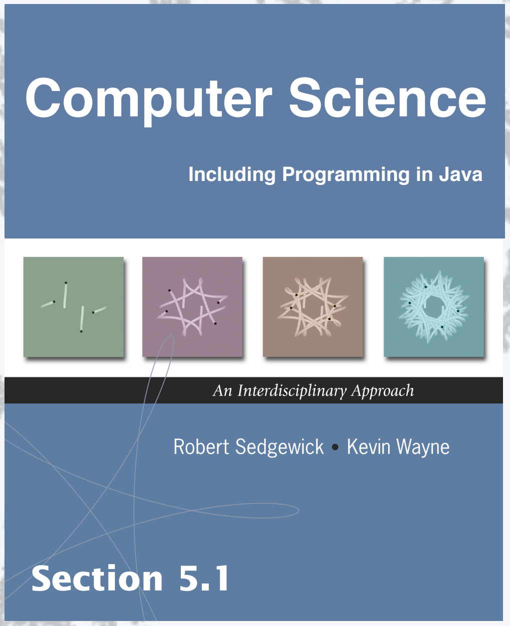
Pop quiz 3 on TOY

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

10	**7 C 0 A**	RC ← 10_{10}
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

COMPUTER SCIENCE
SEDGEWICK / WAYNE

11. A Computing Machine



Computer Science

Including Programming in Java

An Interdisciplinary Approach

Robert Sedgewick • Kevin Wayne

Section 5.1

http://introcs.cs.princeton.edu