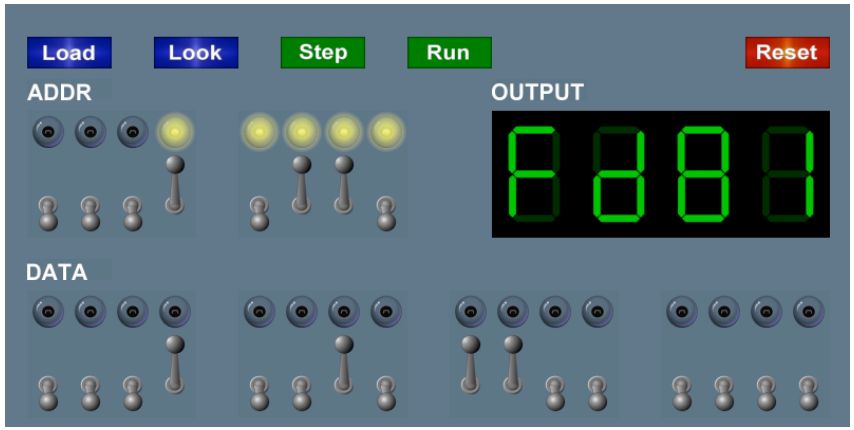
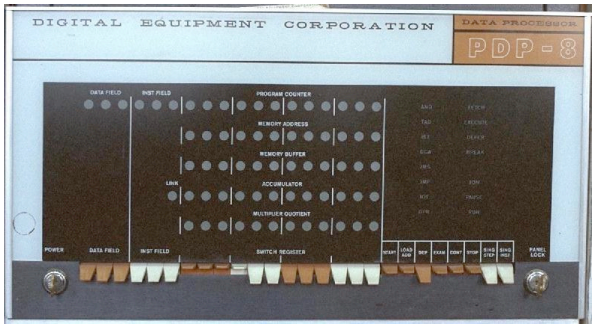
5: The TOY Machine



An imaginary machine similar to:

- Ancient computers.



An imaginary machine similar to:

- Ancient computers.
- Today's microprocessors.

Pentium Celeron

An imaginary machine similar to:

- Ancient computers.
- Today's microprocessors.

Why study TOY?

- Machine language programming.
  - how do Java programs relate to computer?
  - key to understanding Java references (and C pointers)
- Computer architecture.
  - how is a computer put together?
  - how does it work?
- Optimized for understandability, not cost or performance.

Inside the Box

Switches. Input data and programs.

Lights. View data.

Memory.
- Stores data and programs.
- 256 "words." (16 bits each)
- Special word reserved for stdin / stdout.

Registers.
- Fastest form of storage.
- Scratch space during computation.
- 16 registers. (16 bits each)
- Register 0 is always 0.

Arithmetic-logic unit (ALU). Manipulate data stored in registers.

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

Standard input, standard output. Interact with outside world.

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: $6375_{10} = 0001100011100111_2$

Dec	Bin
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Dec	Bin
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	0	1	1	1	0	0	1	1	1

$6375_{10} = +2^{12} +2^{11} +2^{7} +2^{6} +2^{5} +2^{2} +2^{1} +2^{0}$

$= 4096 + 2048 + 128 + 64 + 32 + 4 + 2 + 1$

Shorthand Notation

Use hexadecimal (base 16) representation.
- Binary code, four bits at a time.
- Ex: $6375_{10} = 0001100011100111_2 = 18E7_{16}$

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

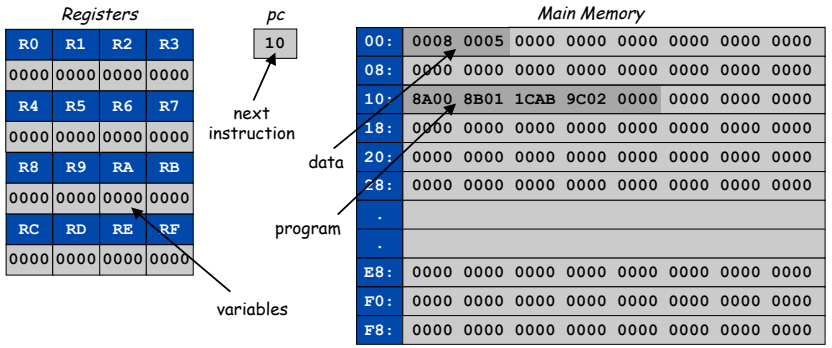
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	0	1	1	1	0	0	1	1	1
1				8				E				7			

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

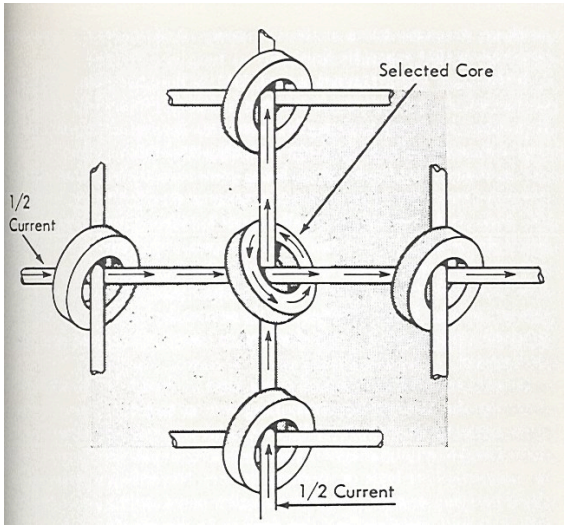
$= 4096 + 2048 + 224 + 7$

Machine "Core" Dump

Machine contents at a particular place and time.
- Record of what program has done.
- Completely determines what machine will do.



Why do they call it "core"?



Reference: http://www.columbia.edu/acis/history/core.html

A Sample Program

A sample program.

- Adds 8 + 5 = D.

RA	RB	RC
0000	0000	0000

Registers

pc
10

```
00: 0008   8          add.toy
01: 0005   5
02: 0000   0

10: 8A00   RA ← mem[00]
11: 8B01   RB ← mem[01]
12: 1CAB   RC ← RA + RB
13: 9C02   mem[02] ← RC
14: 0000   halt
```

Memory

Since PC = 10, machine interprets 8A00 as an instruction.

Load

Load. (opcode 8)

- Loads the contents of some memory location into a register.
- `8A00` means load the contents of memory cell `00` into register `A`.

RA	RB	RC
0000	0000	0000

Registers

pc
10

```
00: 0008   8          add.toy
01: 0005   5
02: 0000   0

10: 8A00   RA ← mem[00]
11: 8B01   RB ← mem[01]
12: 1CAB   RC ← RA + RB
13: 9C02   mem[02] ← RC
14: 0000   halt
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
8_{16}				A_{16}				00_{16}							
opcode				dest d				addr							

Load

Load. (opcode 8)

- Loads the contents of some memory location into a register.
- `8B01` means load the contents of memory cell `01` into register `B`.

RA	RB	RC
0008	0000	0000

Registers

pc
11

```
00: 0008   8          add.toy
01: 0005   5
02: 0000   0

10: 8A00   RA ← mem[00]
11: 8B01   RB ← mem[01]
12: 1CAB   RC ← RA + RB
13: 9C02   mem[02] ← RC
14: 0000   halt
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	1	1	0	0	0	0	0	0	0	1
8_{16}				B_{16}				01_{16}							
opcode				dest d				addr							

Add

Add. (opcode 1)

- Add contents of two registers and store sum in a third.
- `1CAB` means add the contents of registers `A` and `B` and put the result into register `C`.

RA	RB	RC
0008	0005	0000

Registers

pc
12

```
00: 0008   8          add.toy
01: 0005   5
02: 0000   0

10: 8A00   RA ← mem[00]
11: 8B01   RB ← mem[01]
12: 1CAB   RC ← RA + RB
13: 9C02   mem[02] ← RC
14: 0000   halt
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	1	0	1	0	1	0	1	1
1_{16}				C_{16}				A_{16}				B_{16}			
opcode				dest d				source s				source t			

Store

Store. (opcode 9)

- Stores the contents of some register into a memory cell.
- `9C02` means store the contents of register `C` into memory cell `02`.

RA	RB	RC
0008	0005	000D

Registers

pc
13

```
00: 0008   8          add.toy
01: 0005   5
02: 0000   0

10: 8A00   RA ← mem[00]
11: 8B01   RB ← mem[01]
12: 1CAB   RC ← RA + RB
13: 9C02   mem[02] ← RC
14: 0000   halt
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	1	0	0	0	0	0	0	0	0	1	0
9_{16}				C_{16}				02_{16}							
opcode				dest d				addr							

Halt

Halt. (opcode 0)

- Stop the machine.

RA	RB	RC
0008	0005	000D

Registers

pc
14

```
00: 0008   8          add.toy
01: 0005   5
02: 000D   D

10: 8A00   RA ← mem[00]
11: 8B01   RB ← mem[01]
12: 1CAB   RC ← RA + RB
13: 9C02   mem[02] ← RC
14: 0000   halt
```


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):

- Holds memory address of "next instruction."

Instructions

➡	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 zero
	D:	branch positive
	E:	jump register
	F:	jump and link

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

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	0	1	1	1	0	1	0	0	0	0	0	0	1	0	0
Format 1	opcode				dest d				source s				source t			
Format 2	opcode				dest d				addr							

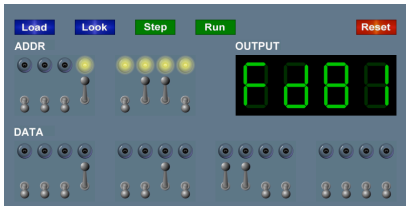
Interfacing with the TOY Machine

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 appears in lights.



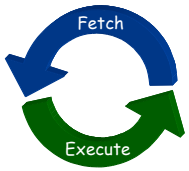
Using the TOY Machine: Run

To run the program:

- Set 8 memory address switches to address of first instruction.
- Press LOOK to set PC to first instruction.
- Press RUN button to repeat fetch-execute cycle until halt opcode.

Fetch-execute cycle.

- FETCH: get instruction from memory.
- EXECUTE: update PC, move data to or from memory and registers, perform calculations.



Programming in TOY

Hello, World. Add two numbers.

To harness the power of TOY, need loops and conditionals.

- Manipulate PC to control program flow.

Branch if zero. (opcode C)

- Changes PC depending of value of some register.
- Used to implement: `for`, `while`, `if-else`.

Branch if positive. (opcode D)

- Analogous.

An Example: Multiplication

Multiply.

- No direct support in TOY hardware.
- Load in integers a and b, and store c = a × b.
- Brute-force algorithm:
  - initialize c = 0
  - add b to c, a times

```
int a = 3;
int b = 9;
int c = 0;

while (a != 0) {
   c = c + b;
   a = a - 1;
}
```

Java

Issues ignored: slow, overflow, negative numbers.

Multiply

```
0A: 0003   3      ⇐ inputs
0B: 0009   9
0C: 0000   0      ⇐ output

0D: 0000   0      ⇐ constants
0E: 0001   1


10: 8A0A   RA ← mem[0A]              a
11: 8B0B   RB ← mem[0B]              b
12: 8C0D   RC ← mem[0D]              c = 0

13: 810E   R1 ← mem[0E]              always 1

14: CA18   if (RA == 0) pc ← 18      while (a != 0) {
15: 1CCB   RC ← RC + RB                 c = c + b
16: 2AA1   RA ← RA - R1                 a = a - 1
17: C014   pc ← 14                   }

18: 9C0C   mem[0C] ← RC
19: 0000   halt
```

loop

multiply.toy

Step-By-Step Trace

```
                                    R1      RA      RB      RC
10: 8A0A   RA ← mem[0A]                     0003
11: 8B0B   RB ← mem[0B]                             0009
12: 8C0D   RC ← mem[0D]                                     0000
13: 810E   R1 ← mem[0E]             0001
14: CA18   if (RA == 0) pc ← 18
15: 1CCB   RC ← RC + RB                                     0009
16: 2AA1   RA ← RA - R1                     0002
17: C014   pc ← 14
14: CA18   if (RA == 0) pc ← 18
15: 1CCB   RC ← RC + RB                                     0012
16: 2AA1   RA ← RA - R1                     0001
17: C014   pc ← 14
14: CA18   if (RA == 0) pc ← 18
15: 1CCB   RC ← RC + RB                                     001B
16: 2AA1   RA ← RA - R1                     0000
17: C014   pc ← 14
14: CA18   if (RA == 0) pc ← 18
18: 9C0C   mem[0C] ← RC
19: 0000   halt
```

multiply.toy

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

```
   1 2 3 4
 * 1 5 1 2
 ---------
   2 4 6 8
 1 2 3 4
 6 1 7 0
 1 2 3 4
 ---------
 0 1 8 6 5 8 0 8
```

Binary

```
     1 0 1 1
   * 1 1 0 1
   ---------
     1 0 1 1
   0 0 0 0
 1 0 1 1
 1 0 1 1
 ---------
 1 0 0 0 1 1 1 1
```

Binary Multiplication

Grade school binary multiplication algorithm to compute c = a × b.

- Initialize c = 0.
- Loop over i bits of b. ⬅ b_i = i^{th} bit of b
  - if b_i = 0, do nothing
  - if b_i = 1, shift a left i bits and add to c

```
    1 0 1 1     a
  * 1 1 0 1     b
  ---------
    1 0 1 1     a << 0
  0 0 0 0
+ 1 0 1 1       a << 2
+ 1 0 1 1       a << 3
---------------
1 0 0 0 1 1 1 1 c
```

Implement with built-in TOY shift instructions.

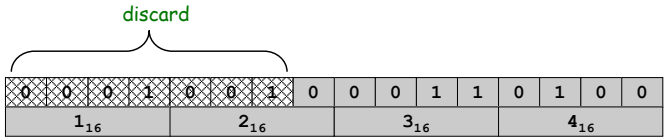
```
int c = 0;
for (int i = 15; i >= 0; i--)
   if (((b >> i) & 1) == 1)
      c = c + (a << i);
```

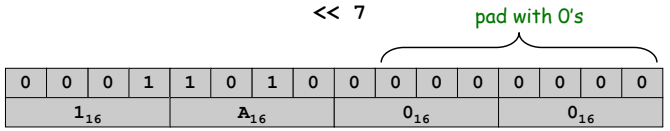
⬅ b_i = i^{th} bit of b

Shift Left

Shift left. (opcode 5)

- Move bits to the left, padding with zeros as needed.
- 1234_{16} << 7_{16} = $1A00_{16}$

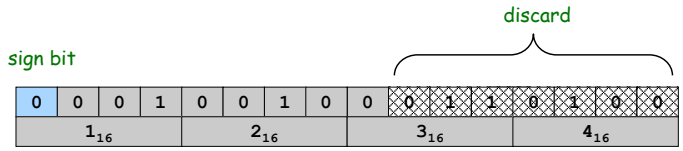


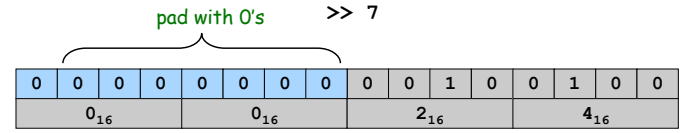


Shift Right

Shift right. (opcode 6)

- Move bits to the right, padding with sign bit as needed.
- 1234_{16} >> 7_{16} = 0024_{16}



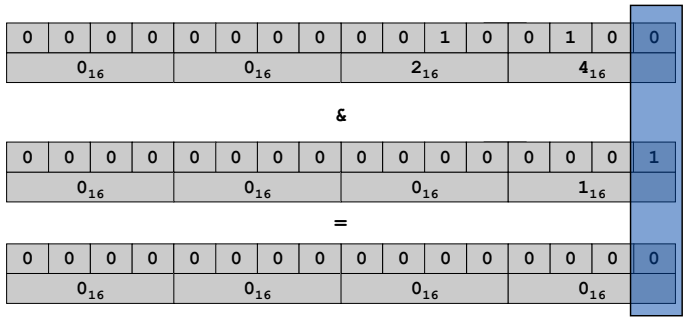


Bitwise AND

Logical AND. (opcode B)

- Logic operations are BITWISE.
- 0024_{16} & 0001_{16} = 0000_{16}

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



Shifting and Masking

Shift and mask: get the 7th bit of 1234.

- Compute 1234_{16} >> 7_{16} = 0024_{16}.
- Compute 0024_{16} && 1_{16} = 0_{16}.

0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0
1_{16}				2_{16}				3_{16}				4_{16}			

>> 7

0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
0_{16}				0_{16}				2_{16}				4_{16}			

&

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0_{16}				0_{16}				0_{16}				1_{16}			

=

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0_{16}				0_{16}				0_{16}				0_{16}			

Binary Multiplication

```
0A: 0007    3
0B: 0008    9   <=  inputs
0C: 0000    0   <=  output
0D: 0000    0
0E: 0001    1   <=  constants
0F: 0010   16

10: 8A0A   RA ← mem[0A]              a
11: 8B0B   RB ← mem[0B]              b
12: 8C0D   RC ← mem[0D]              c = 0
13: 810E   R1 ← mem[0E]              always 1
14: 820F   R2 ← mem[0F]              i = 16   <= 16 bit words

                                     do {
15: 2221   R2 ← R2 -  R1                i--
16: 53A2   R3 ← RA << R2                a << i
17: 64B2   R4 ← RB >> R2                b >> i
18: 3441   R4 ← R4 &  R1                b_i = i^th bit of b
19: C41B   if (R4 == 0) goto 1B         if b_i is 1
1A: 1CC3   RC ← RC + R3                    add a << i to sum
1B: D215   if (R2 > 0) goto 15       } while (i > 0);

1C: 9C0C   mem[0C] ← RC                          multiply-fast.toy
```

loop

branch

TOY Reference Card

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Format 1	opcode				dest d				source s				source t			
Format 2	opcode				dest d				addr							

#	Operation	Fmt	Pseudocode
0:	halt	1	exit(0)
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

Register 0 always 0.
Loads from mem[FF] from stdin.
Stores to mem[FF] to stdout.

Useful TOY "Idioms"

Jump absolute.

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

```
17: C014   pc ← 14
```

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

```
17: 1230   R[2] ← R[3]
```

No-op.

- Instruction that does nothing.
- Plays the role of whitespace in C programs.
  - numerous other possibilities!

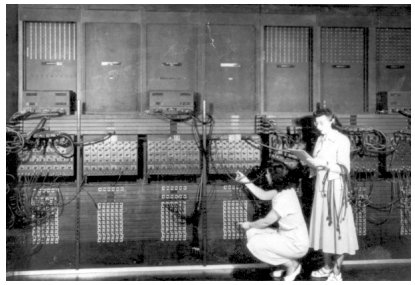
```
17: 1000   no-op
```

A Little History

Electronic Numerical Integrator and Calculator (ENIAC).

- First widely known general purpose electronic computer.
- 30 tons, 30 x 50 x 8.5 ft, 17,468 vacuum tubes, 300 multiply/sec.
- Conditional jumps, programmable.
- Programming: change switches and cable connections.
- Data: enter numbers using punch cards.

John Mauchly (left) and J. Presper Eckert (right)
http://cs.swau.edu/~durkin/articles/history_computing.html



ENIAC, Ester Gerston (left), Gloria Gordon (right)
US Army photo: http://ftp.arl.mil/ftp/historic-computers

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 without rewiring.

Outgrowth of Alan Turing's work. (stay tuned)

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

John von Neumann

Maurice Wilkes (left)
EDSAC (right)