

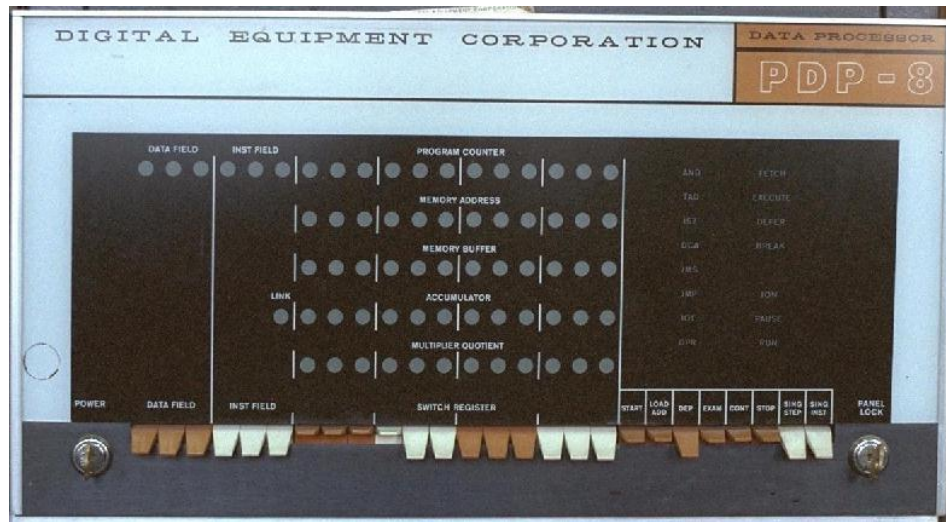
# 5. The TOY Machine



# What is TOY?

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.
- Still some situations today where it is really necessary.

↑  
multimedia, computer games, scientific computing, SSE, AVX

## Computer architecture.

- How does it work?
- How is a computer put together?

TOY machine. Optimized for **simplicity**, not cost or performance.

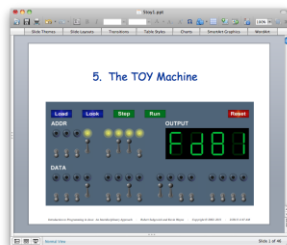
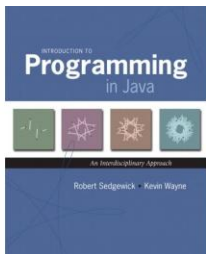
# Data and Programs Are Encoded in Binary

Each bit consists of two states:

- 1 or 0; true or false.
- Switch is on or off; wire has high voltage or low voltage.

Everything stored in a computer is a sequence of bits.

- **Data** and **programs**.
- Text, documents, pictures, sounds, movies, executables, ...



# Binary Encoding

How to represent integers?

- Use **binary** encoding.
- Ex:  $6375_{10} = 0001100011100111_2$

Dec	Bin	Dec	Bin
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	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


$$\begin{aligned} 6375_{10} &= +2^{12} +2^{11} && +2^7 +2^6 +2^5 && +2^2 +2^1 +2^0 \\ &= 4096 +2048 && +128 +64 +32 && +4 +2 +1 \end{aligned}$$

# Hexadecimal Encoding

## How to represent integers?

- Use **hexadecimal** encoding.
- Ex:  $6375_{10} = 0001100011100111_2$   
 $= 18E7_{16}$

binary code,  
4 bits at a time

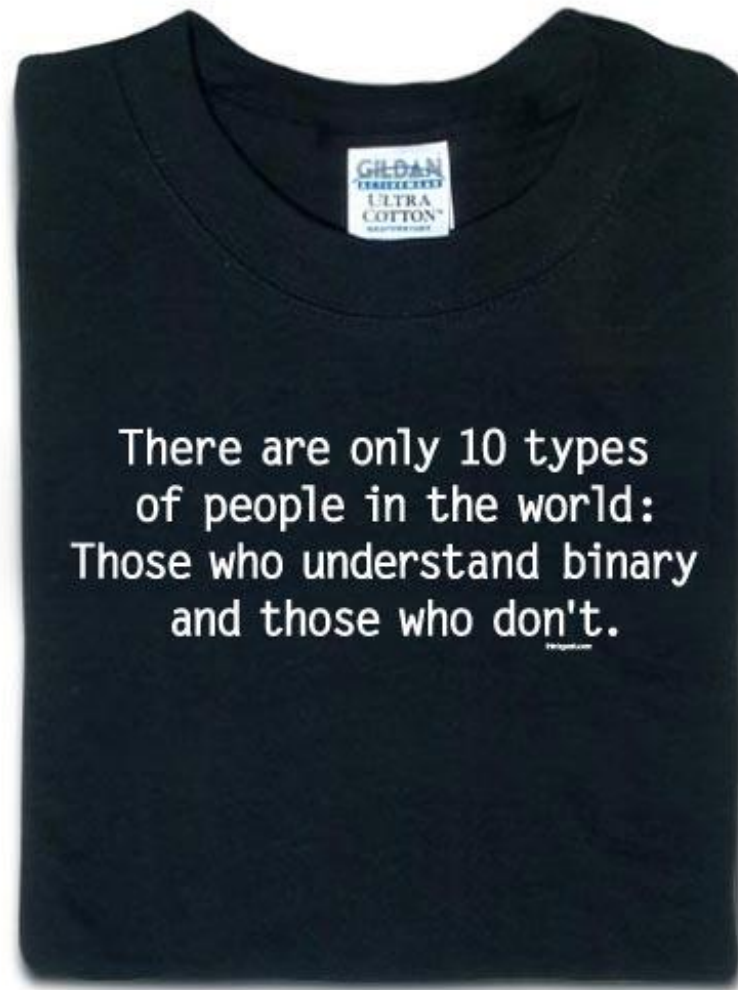


Dec	Bin	Hex	Dec	Bin	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

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			

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

# Binary People



<http://www.thinkgeek.com/tshirts-apparel/unisex/frustrations/5aa9>

# Inside the TOY Box

**Switches.** Input data and programs.

**Lights.** View data.

**Memory.**

- Stores data and programs.
- 256 16-bit "words."
- Special word for stdin / stdout.

**Program counter (PC).**

- An extra 8-bit register.
- Next instruction to be executed.

**Registers.**

- Fastest form of storage.
- Scratch space during computation.
- 16 16-bit registers.
- Register 0 is always 0.

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

**Standard input, standard output.**

Interact with outside world.

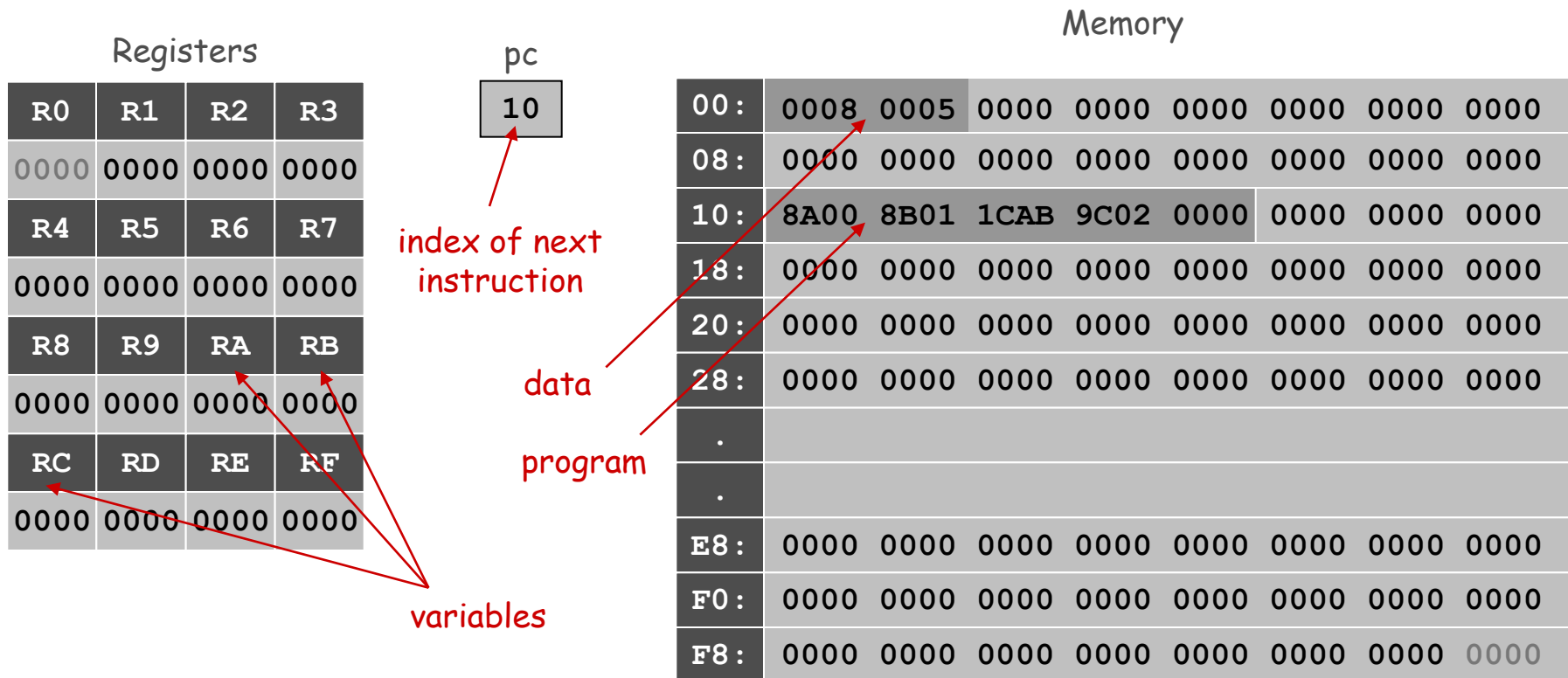




# TOY Machine "Core" Dump

A **core dump** is the contents of machine at a particular place and time.

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



# A Sample Program

A sample program. Adds  $0008 + 0005 = 000D$ .

RA	RB	RC
0000	0000	0000

Registers

pc
10

Program counter

TOY memory  
(program and data)

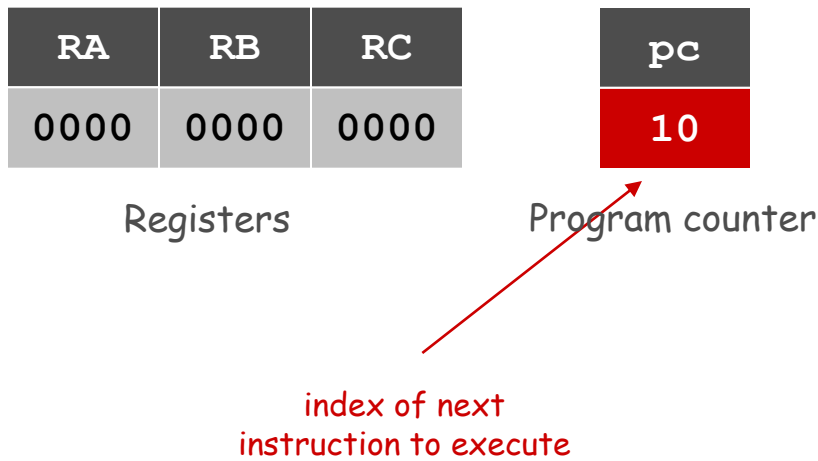
comments

00: 0008	8	
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

add.toy

# A Sample Program

**Program counter.** The `pc` is initially 10, so the machine interprets 8A00 as an instruction.



00:	0008	8
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

add.toy

# 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

Program counter

00:	0008	8
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

add.toy

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 <sub>16</sub>				A <sub>16</sub>				00 <sub>16</sub>							
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

Program counter

00:	0008	8
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

add.toy

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 <sub>16</sub>				B <sub>16</sub>				01 <sub>16</sub>							
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

Program counter

00:	0008	8
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

add.toy

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

Program counter

00:	0008	8
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

add.toy

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 <sub>16</sub>				C <sub>16</sub>				02 <sub>16</sub>							
opcode				dest d				addr							

# Halt

Halt. [opcode 0]

- Stop the machine.

RA	RB	RC
0008	0005	000D

Registers

pc
14

Program counter

```
00: 0008    8
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
```

add.toy



# Program and Data

**Program.** Sequence of 16-bit integers, interpreted one way.

**Data.** Sequence of 16-bit integers, interpreted other way.

**Program counter (pc).** Holds memory address of the next "instruction" and determines which integers get interpreted as instructions.

**16 instruction types.** Changes contents of registers, memory, and pc in specified, well-defined ways.

## 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

← add, subtract, ...

[Format 2] 8-bit memory address or constant

← load, store, ...

	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.



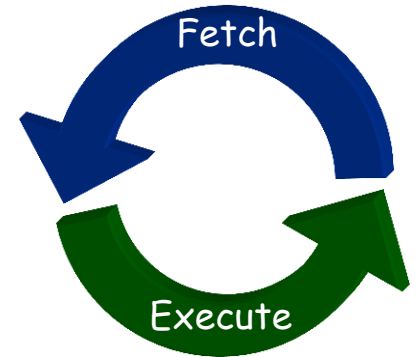
# Interfacing with the TOY Machine

To execute the program:

- Set 8 memory address switches to address of first instruction.
- Press `Look` to set `pc` to first instruction.
- Press `Run` 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.



# Flow Control

## Flow control.

- To harness the power of TOY, need loops and conditionals.
- Manipulate `pc` to control program flow.

```
if (boolean expression) {  
    statement 1 ;  
    statement 2 ;  
}
```

```
while (boolean expression) {  
    statement 1 ;  
    statement 2 ;  
}
```

## Branch if zero. [opcode C]

- Changes `pc` depending on whether value of some register is **zero**.
- Used to implement: `for`, `while`, `if-else`.

## Branch if positive. [opcode D]

- Changes `pc` depending on whether value of some register is **positive**.
- Used to implement: `for`, `while`, `if-else`.

# An Example: Multiplication

**Multiply.** Given integers  $a$  and  $b$ , compute  $c = a \times b$ .

**TOY multiplication.** No direct support in TOY hardware.

**Brute-force multiplication algorithm:**

- Initialize  $c$  to 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;
}
```

brute force multiply in 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 = 3;
11: 8B0B    RB ← mem[0B]    b = 9;
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
```

multiply.toy

# Step-By-Step Trace

	<u>R1</u>	<u>RA</u>	<u>RB</u>	<u>RC</u>
10: 8A0A		0003		
11: 8B0B			0009	
12: 8C0D				0000
13: 810E	0001			
14: CA18				
15: 1CCB				0009
16: 2AA1		0002		
17: C014				
14: CA18				
15: 1CCB				0012
16: 2AA1		0001		
17: C014				
14: CA18				
15: 1CCB				001B
16: 2AA1		0000		
17: C014				
14: CA18				
18: 9C0C				
19: 0000				

multiply.toy



# A Little History

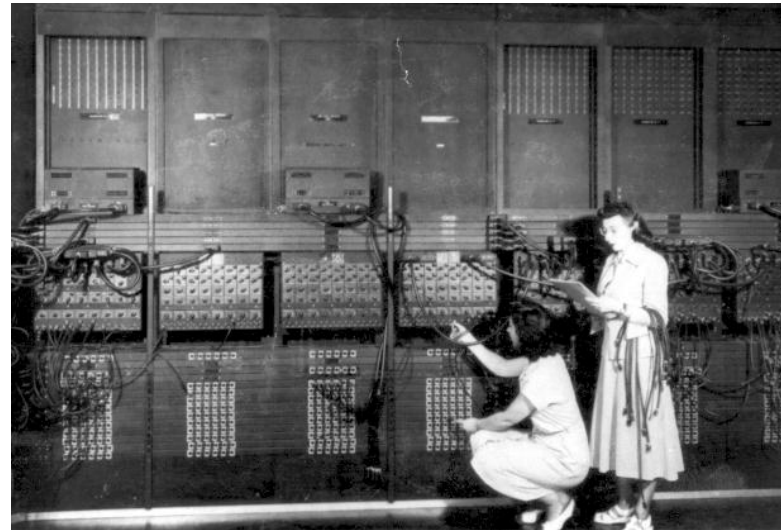
## Electronic Numerical Integrator and Calculator (ENIAC).

- First widely known general purpose electronic computer.
- Conditional jumps, programmable.
- Programming: change switches and cable connections.
- Data: enter numbers using punch cards.

30 tons  
30 x 50 x 8.5 ft  
17,468 vacuum tubes  
300 multiply/sec  
15,000 watts



John Mauchly (left) and J. Presper Eckert (right)  
[http://cs.swau.edu/~durkin/articles/history\\_computing.html](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 program encoded in binary.
- Data and program stored in **same** memory.
- Can change program without rewiring.

Outgrowth of Alan Turing's work.

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



John von Neumann



Maurice Wilkes (left)  
EDSAC (right)

# 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							

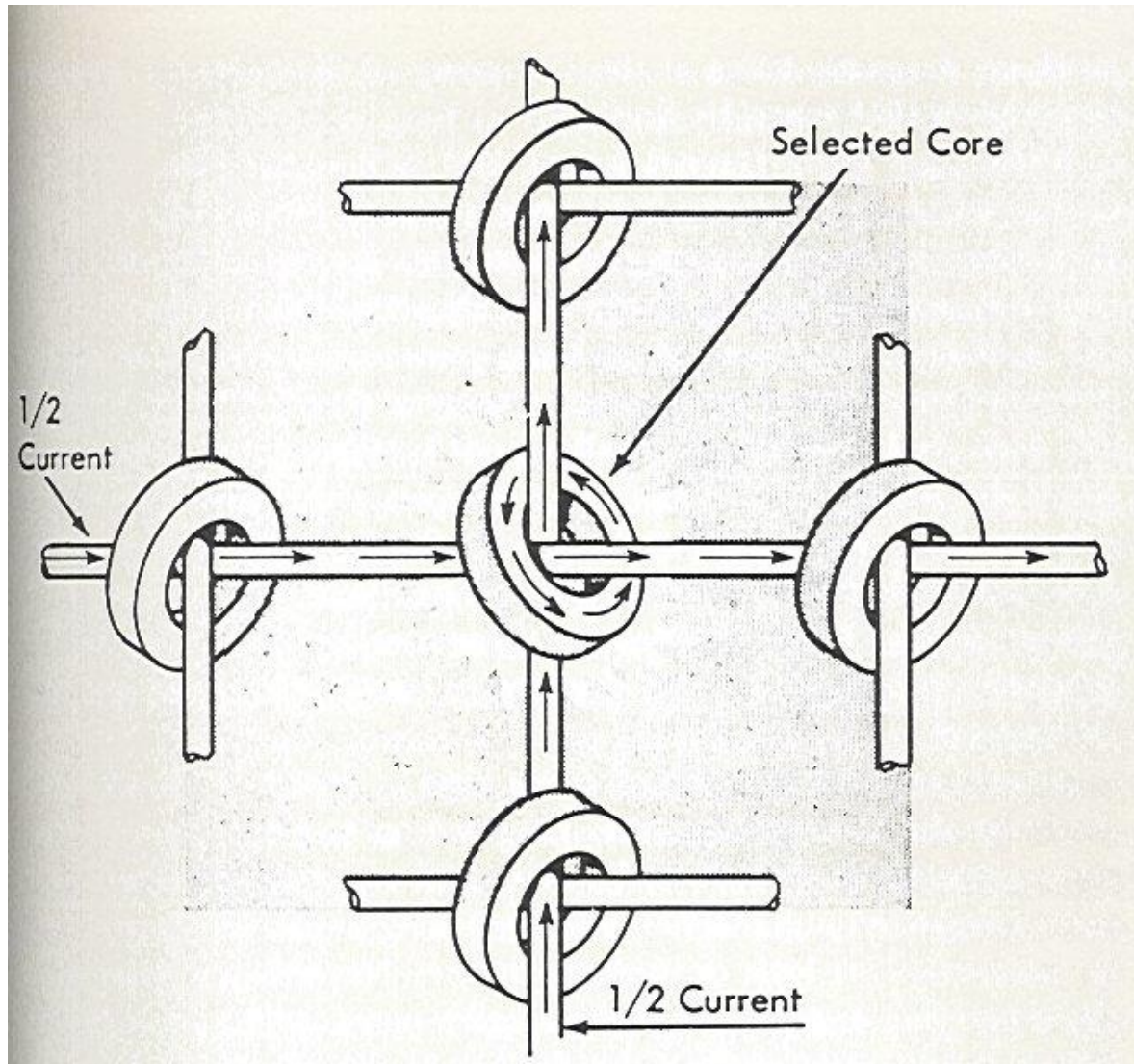
#	Operation	Fmt	Pseudocode
0:	halt	1	exit(0)
1:	add	1	$R[d] \leftarrow R[s] + R[t]$
2:	subtract	1	$R[d] \leftarrow R[s] - R[t]$
3:	and	1	$R[d] \leftarrow R[s] \& R[t]$
4:	xor	1	$R[d] \leftarrow R[s] \wedge R[t]$
5:	shift left	1	$R[d] \leftarrow R[s] \ll R[t]$
6:	shift right	1	$R[d] \leftarrow R[s] \gg R[t]$
7:	load addr	2	$R[d] \leftarrow \text{addr}$
8:	load	2	$R[d] \leftarrow \text{mem}[\text{addr}]$
9:	store	2	$\text{mem}[\text{addr}] \leftarrow R[d]$
A:	load indirect	1	$R[d] \leftarrow \text{mem}[R[t]]$
B:	store indirect	1	$\text{mem}[R[t]] \leftarrow R[d]$
C:	branch zero	2	if ( $R[d] == 0$ ) $\text{pc} \leftarrow \text{addr}$
D:	branch positive	2	if ( $R[d] > 0$ ) $\text{pc} \leftarrow \text{addr}$
E:	jump register	2	$\text{pc} \leftarrow R[d]$
F:	jump and link	2	$R[d] \leftarrow \text{pc}; \text{pc} \leftarrow \text{addr}$

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

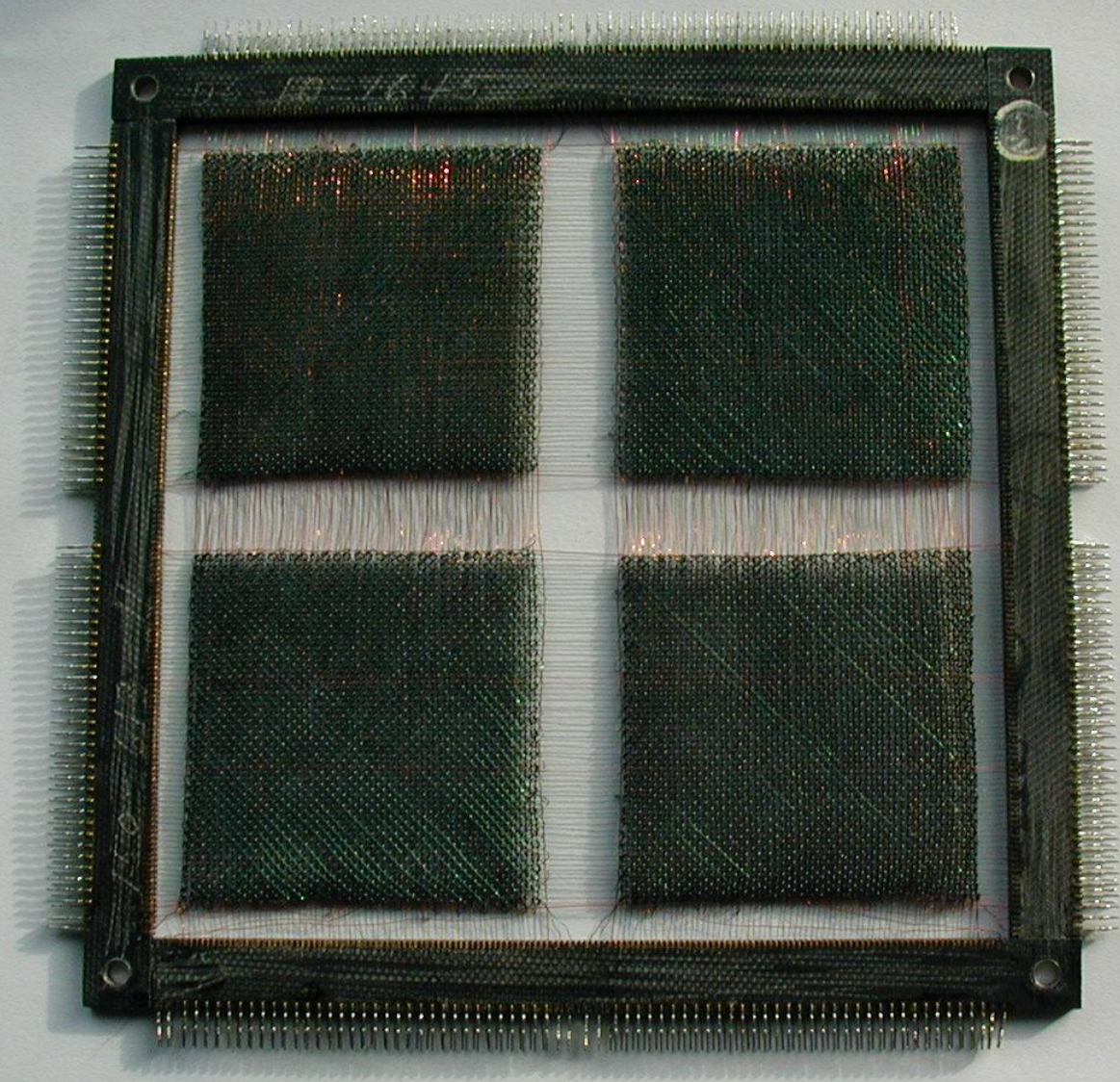
16-bit registers.  
16-bit memory.  
8-bit program counter.

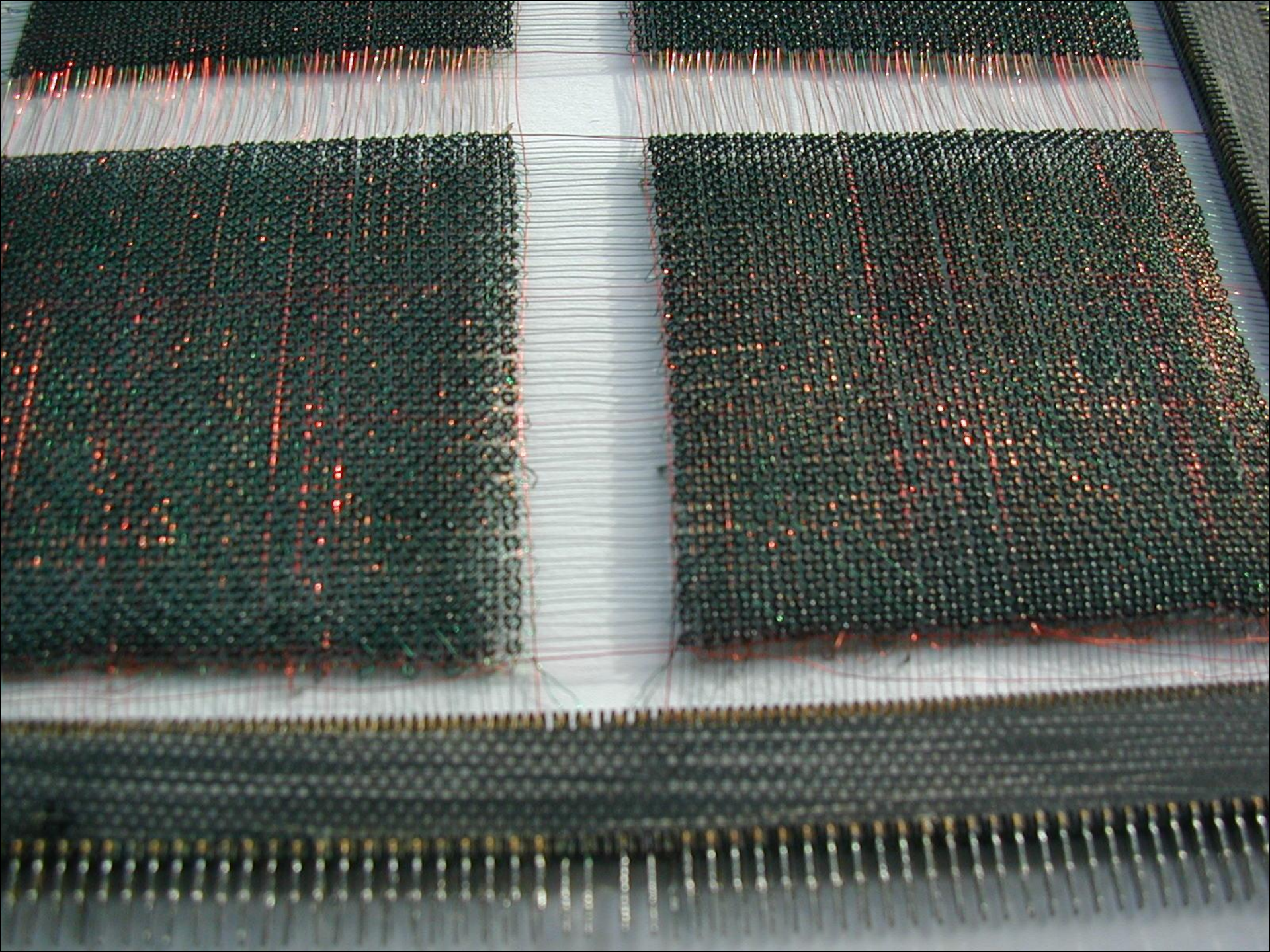
# 5: Supplemental Notes

# Why do They Call it "Core"?



<http://www.columbia.edu/acis/history/core.html>





# 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

$$\begin{array}{r} \phantom{0000} 1\ 2\ 3\ 4 \\ * \phantom{000} 1\ 5\ 1\ 2 \\ \hline \phantom{0000} 2\ 4\ 6\ 8 \\ \phantom{000} 1\ 2\ 3\ 4 \\ \phantom{00} 6\ 1\ 7\ 0 \\ \phantom{0} 1\ 2\ 3\ 4 \\ \hline 0\ 1\ 8\ 6\ 5\ 8\ 0\ 8 \end{array}$$

Binary

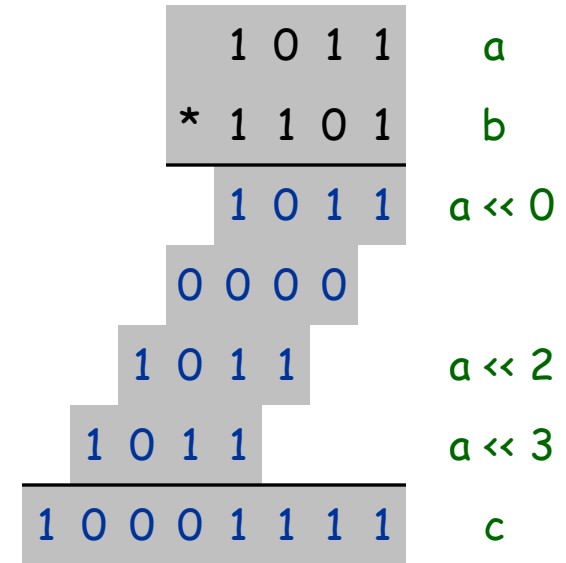
$$\begin{array}{r} \phantom{0000} 1\ 0\ 1\ 1 \\ * \phantom{000} 1\ 1\ 0\ 1 \\ \hline \phantom{0000} 1\ 0\ 1\ 1 \\ \phantom{000} 0\ 0\ 0\ 0 \\ \phantom{00} 1\ 0\ 1\ 1 \\ \phantom{0} 1\ 0\ 1\ 1 \\ \hline 1\ 0\ 0\ 0\ 1\ 1\ 1\ 1 \end{array}$$



# Binary Multiplication

Grade school binary multiplication algorithm to compute  $c = a \times b$ .

- Initialize  $c = 0$ .
- Loop over  $i$  bits of  $b$ .  $\leftarrow b_i = i^{\text{th}}$  bit 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.

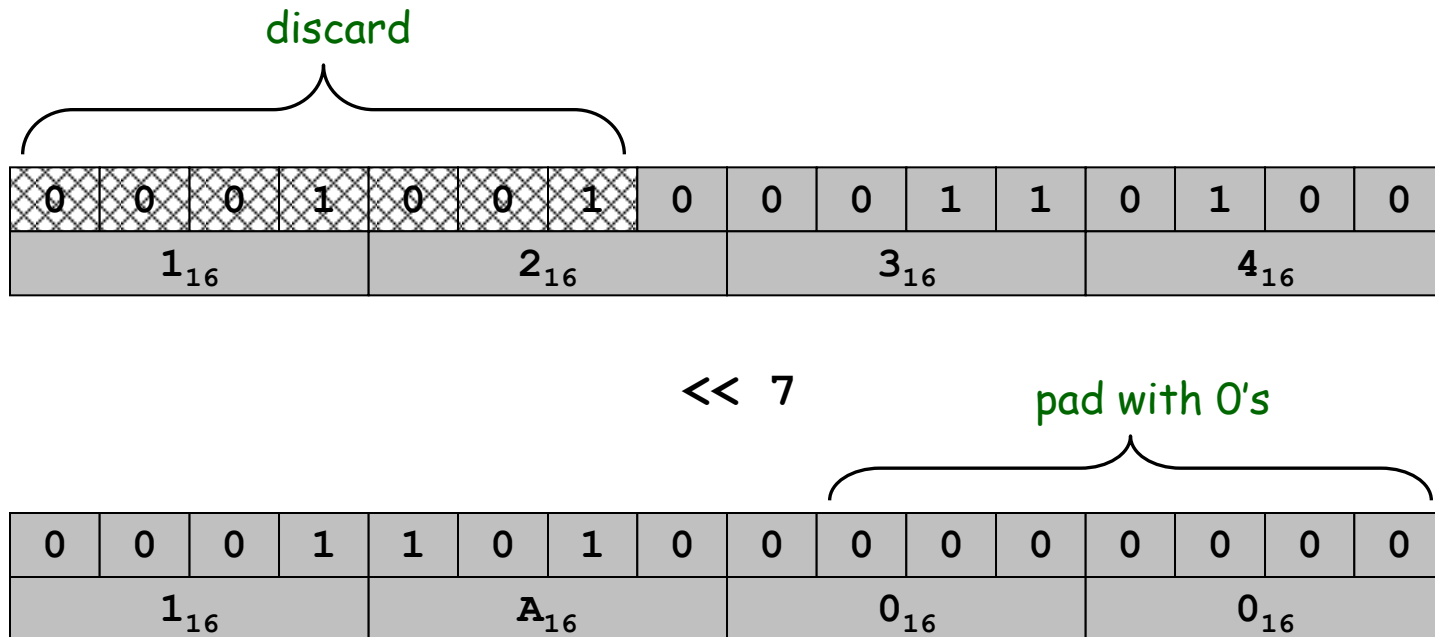
```
int c = 0;
for (int i = 15; i >= 0; i--)
    if (((b >> i) & 1) == 1)
        c = c + (a << i);
```

$\leftarrow b_i = i^{\text{th}}$  bit of  $b$

# Shift Left

## Shift left. (opcode 5)

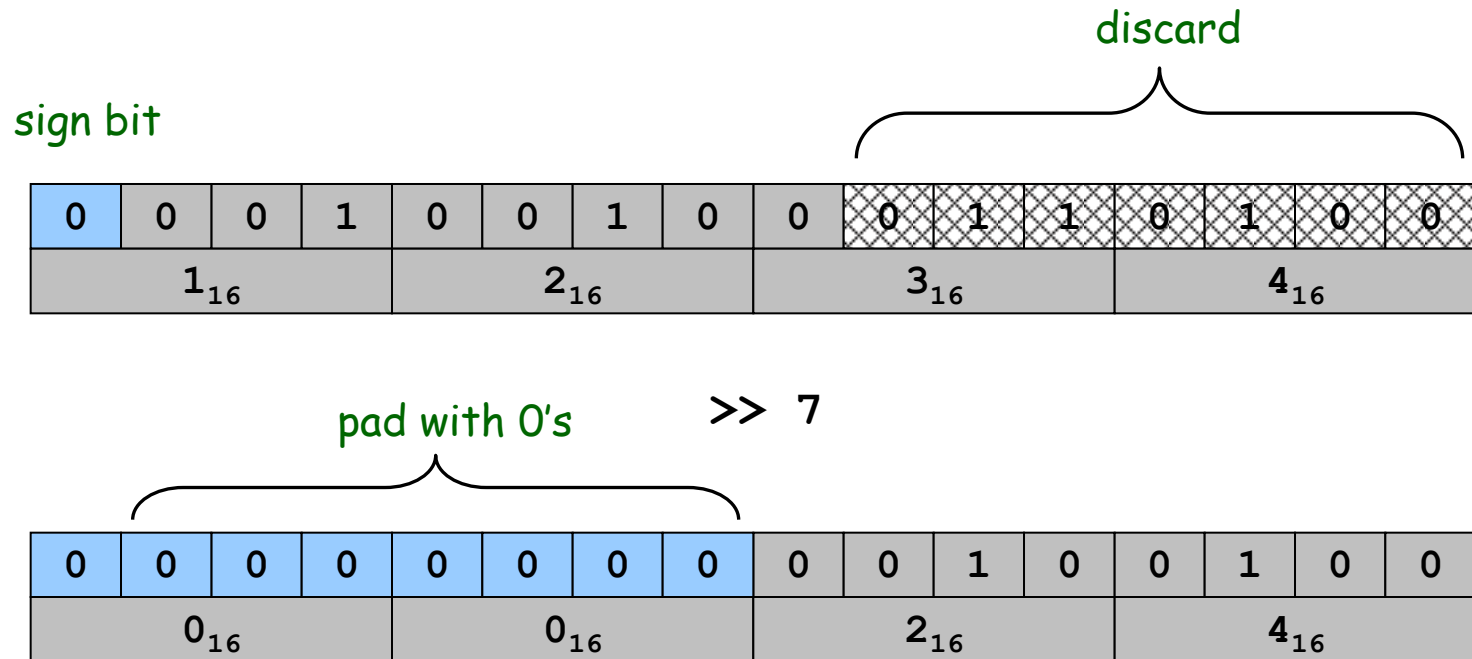
- Move bits to the left, padding with zeros as needed.
- $1234_{16} \ll 7_{16} = 1A00_{16}$



# Shift Right

Shift right. (opcode 6)

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

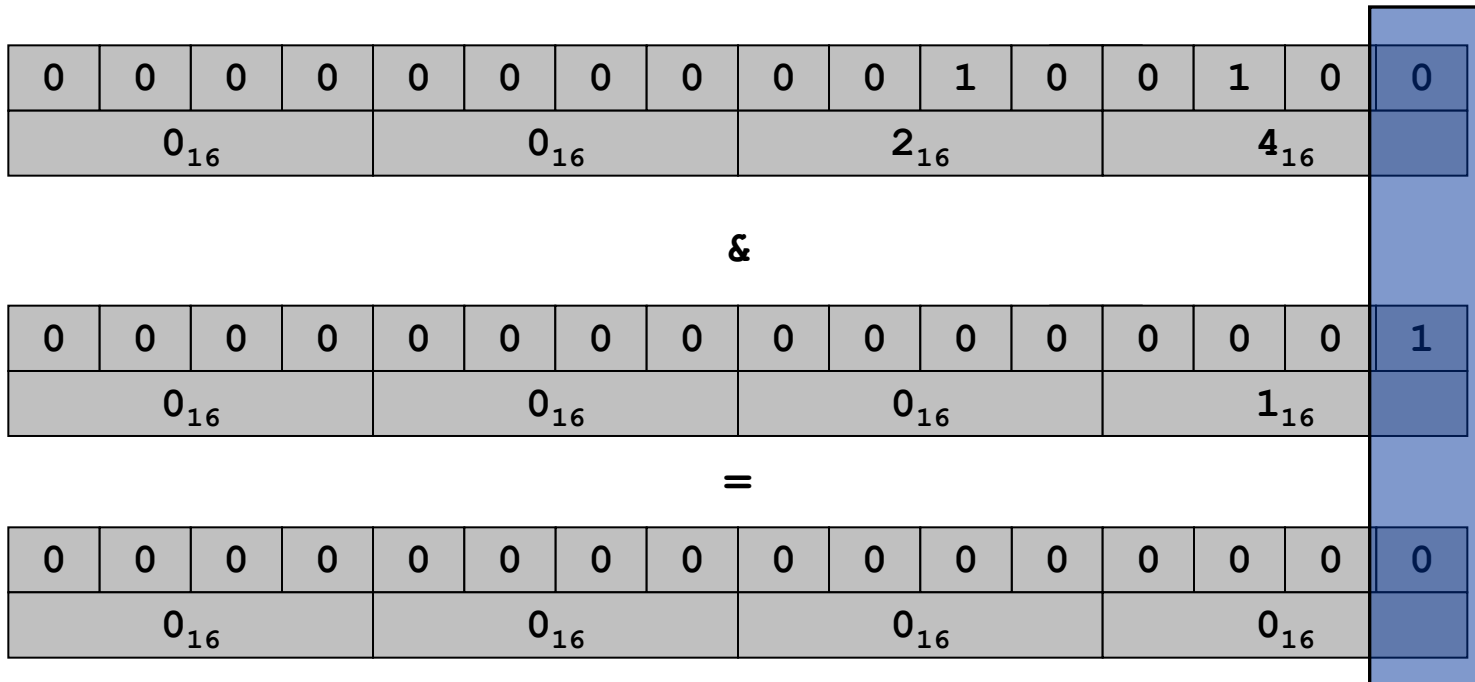


# Bitwise AND

x	y	&
0	0	0
0	1	0
1	0	0
1	1	1

Logical AND. (opcode B)

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



# Shifting and Masking

**Shift and mask:** get the 7<sup>th</sup> bit of 1234.

- Compute  $1234_{16} \gg 7_{16} = 0024_{16}$ .
- Compute  $0024_{16} \&\& 1_{16} = 0_{16}$ .

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

$\gg 7$

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: 0003    3
0B: 0009    9
0C: 0000    0
0D: 0000    0
0E: 0001    1
0F: 0010   16
  
```

← inputs

← output

← constants

```

10: 8A0A    RA ← mem[0A]
11: 8B0B    RB ← mem[0B]
12: 8C0D    RC ← mem[0D]
13: 810E    R1 ← mem[0E]
14: 820F    R2 ← mem[0F]
  
```

```

a
b
c = 0
always 1
i = 16 ← 16 bit words
  
```

loop

branch

```

15: 2221    R2 ← R2 - R1
16: 53A2    R3 ← RA << R2
17: 64B2    R4 ← RB >> R2
18: 3441    R4 ← R4 & R1
19: C41B    if (R4 == 0) goto 1B
1A: 1CC3    RC ← RC + R3
1B: D215    if (R2 > 0) goto 15

1C: 9C0C    mem[0C] ← RC
  
```

```

do {
    i--
    a << i
    b >> i
    bi = ith bit of b
    if bi is 1
        add a << i to sum
} while (i > 0);
  
```

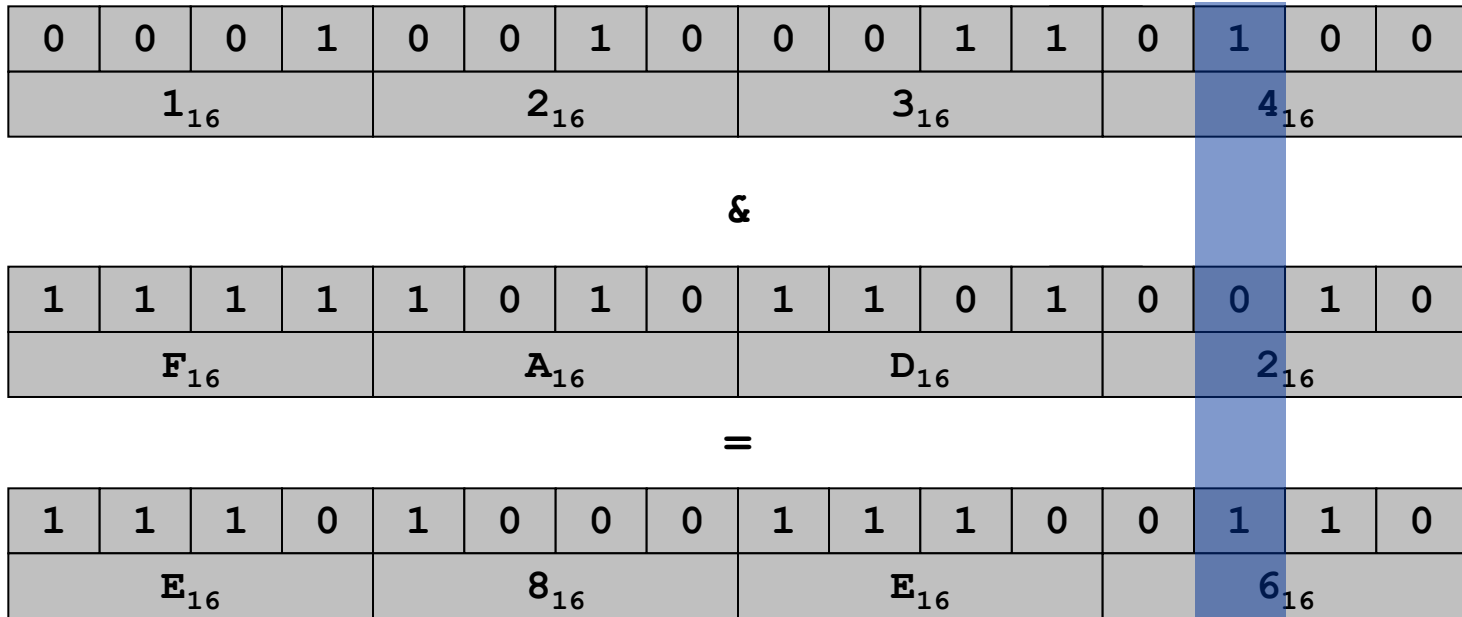
multiply-fast.toy

# Bitwise XOR

## Bitwise XOR. (opcode 4)

- Logic operations are BITWISE.
- $1234_{16} \wedge FAD2_{16} = E8E6_{16}$

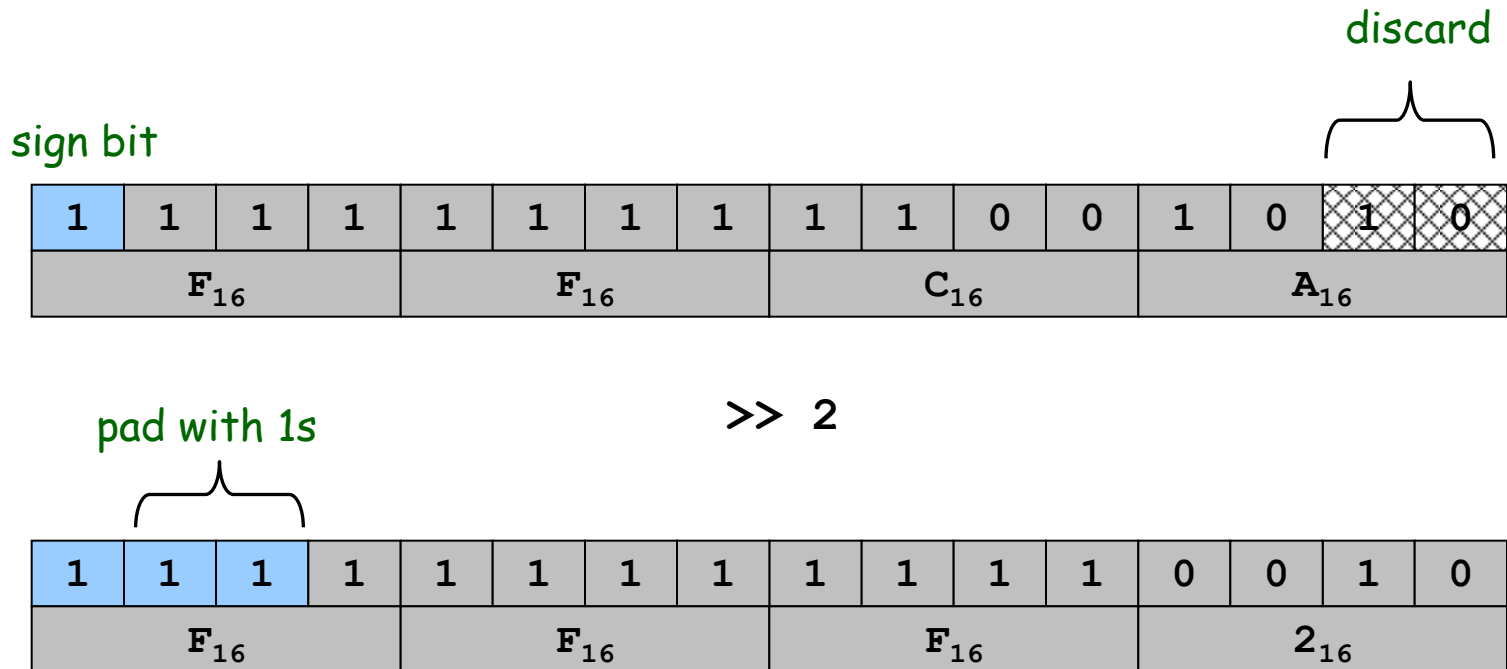
x	y	^
0	0	0
0	1	1
1	0	1
1	1	0



# Shift Right (Sign Extension)

Shift right. (opcode 6)

- Move bits to the right, padding with sign bit as needed.
- $FFCA_{16} \gg 2_{16} = FFF2_{16}$
- $-53_{10} \gg 2_{10} = -13_{10}$





# 5: Extra Slides

# 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

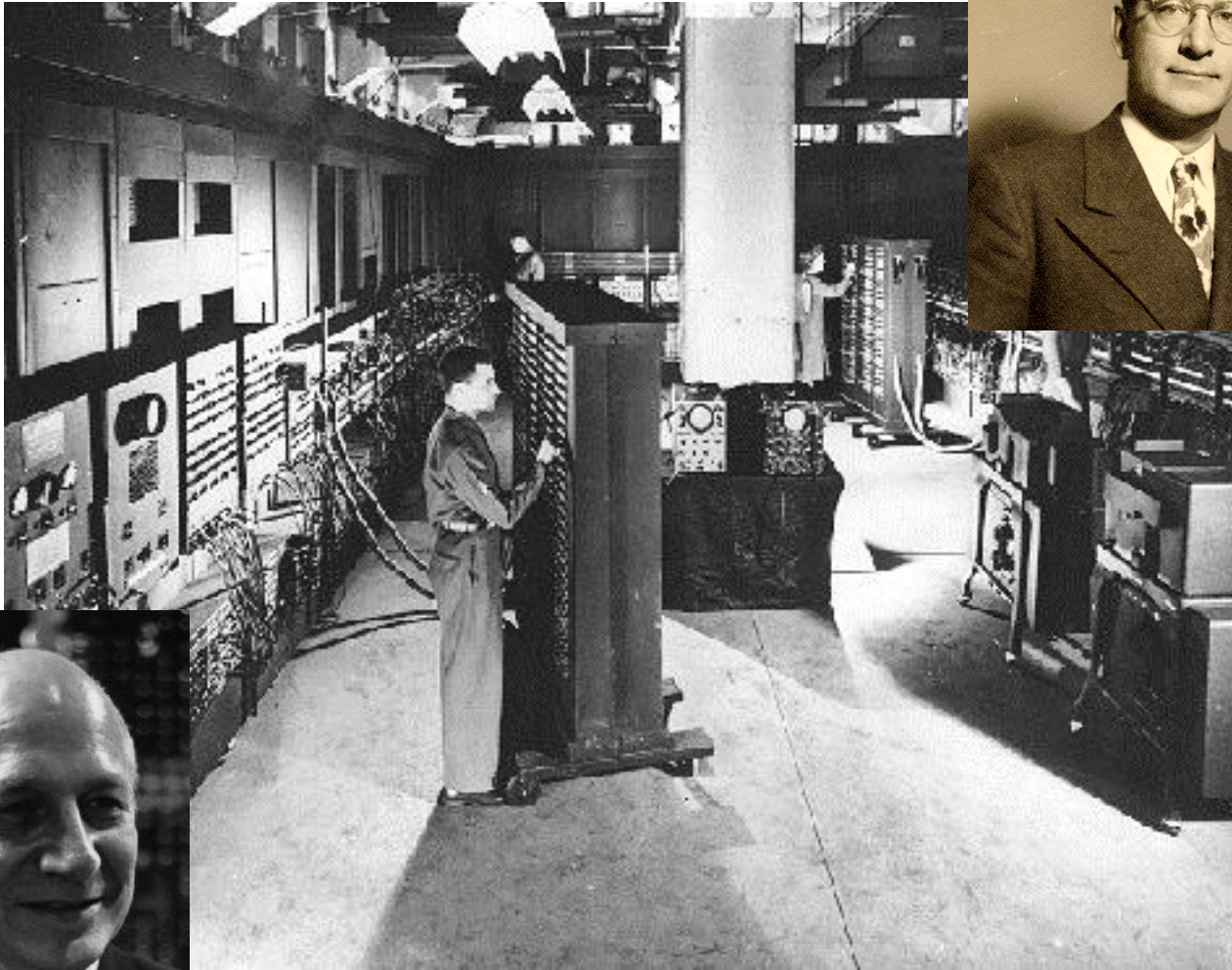
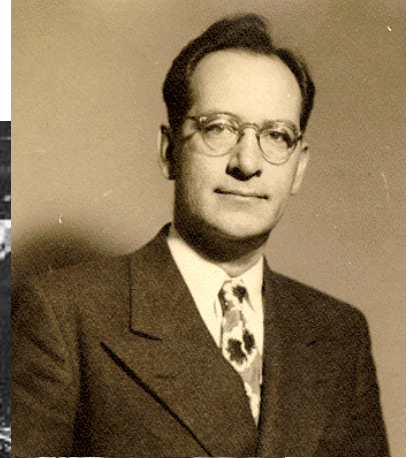
```
17: 1230  R[2] ← R[3]
```

## No-op.

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

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

# ENIAC



# What is TOY?

An imaginary machine similar to:

- Ancient computers.

