

# The TOY Machine



# 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.

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



John von Neumann

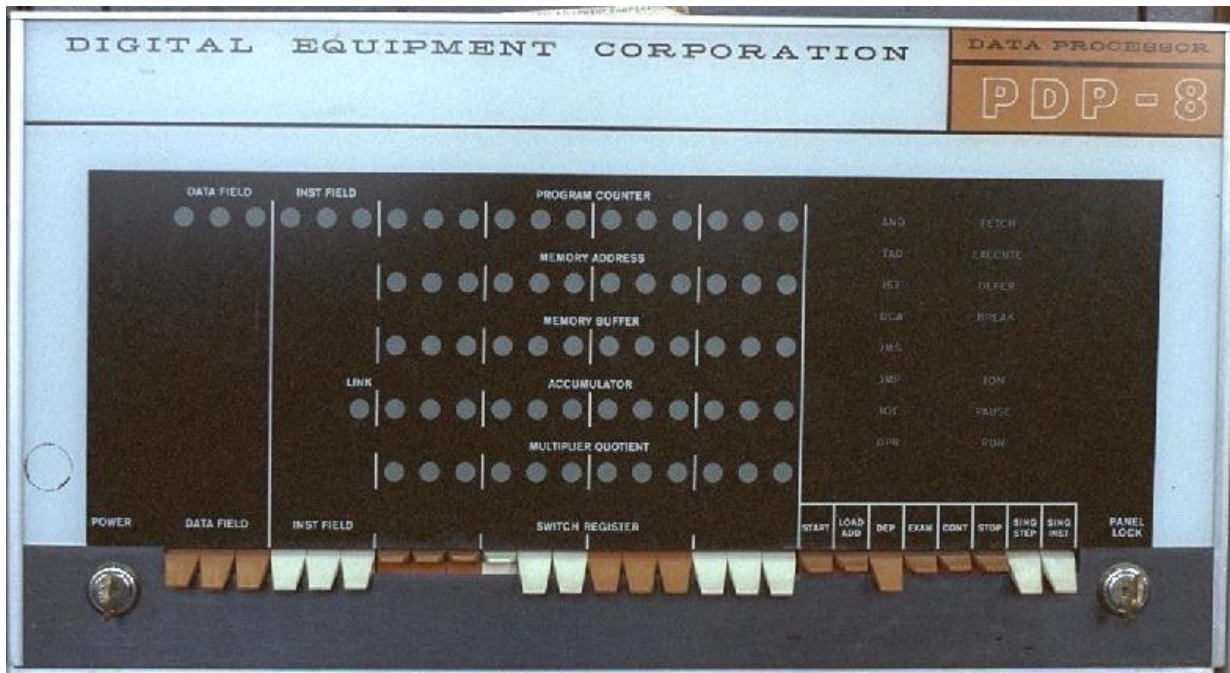


Maurice Wilkes (left)  
EDSAC (right)

# What is TOY?

An imaginary machine similar to:

- Ancient computers. (PDP-8, world's first commercially successful minicomputer. 1960s)
  - 12-bit words
  - 2K words of memory
  - Used in Apollo project



# What is TOY?

An imaginary machine similar to:

- Ancient computers.
- Today's microprocessors.



Pentium

Celeron

# What is TOY?

An imaginary machine similar to:

- ♦ Ancient computers.
- ♦ Today's microprocessors.

Why study TOY?

- ♦ Machine language programming.
  - how do high-level programs relate to computer?
  - a favor of assembly programming
- ♦ 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 for stdin / stdout.

**Program counter (PC).**

- An extra 8-bit register.
- Keeps track of next instruction to be executed.

**Registers.**

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

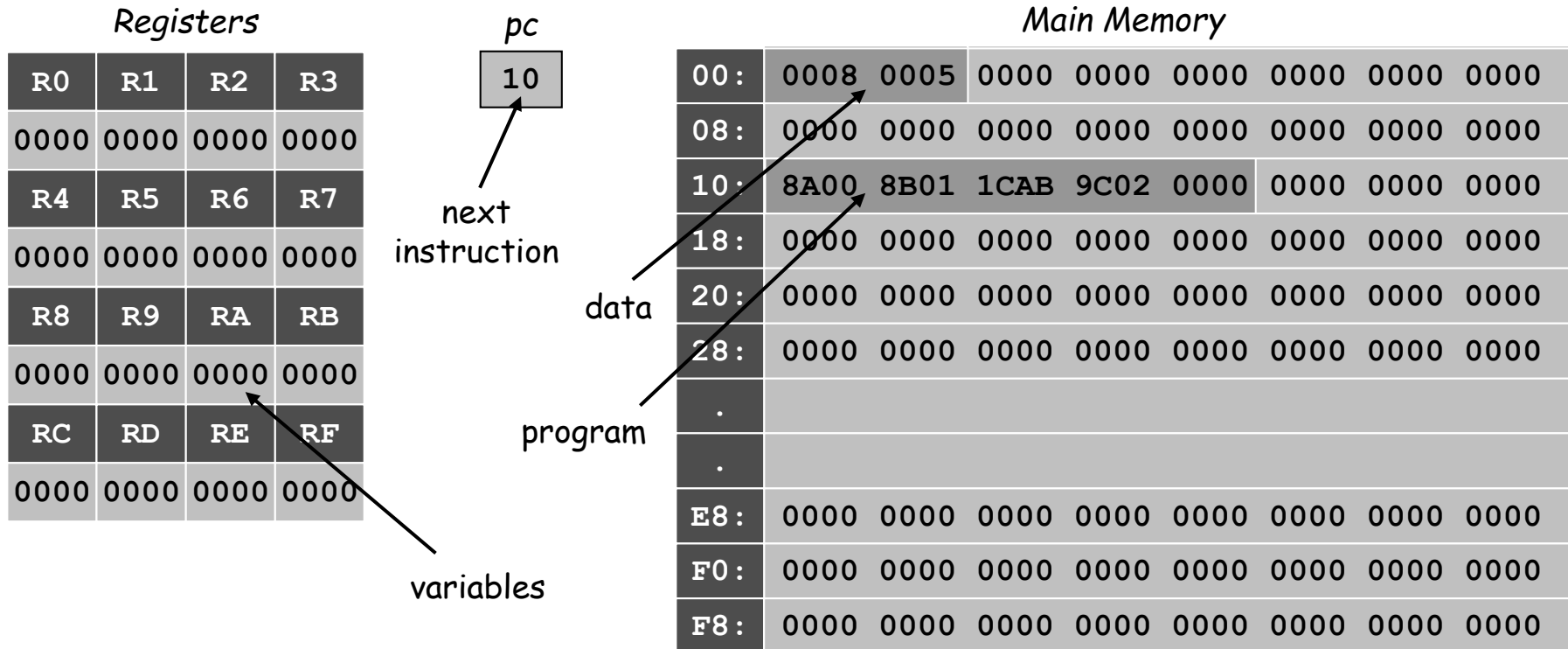
**Arithmetic-logic unit (ALU).** Manipulate data stored in registers.

**Standard input, standard output.** Interact with outside world.

# Machine "Core" Dump

Machine contents at a particular place and time.

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



# 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):

- Stores memory address of "next instruction."
- TOY usually starts at address 10.

## 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



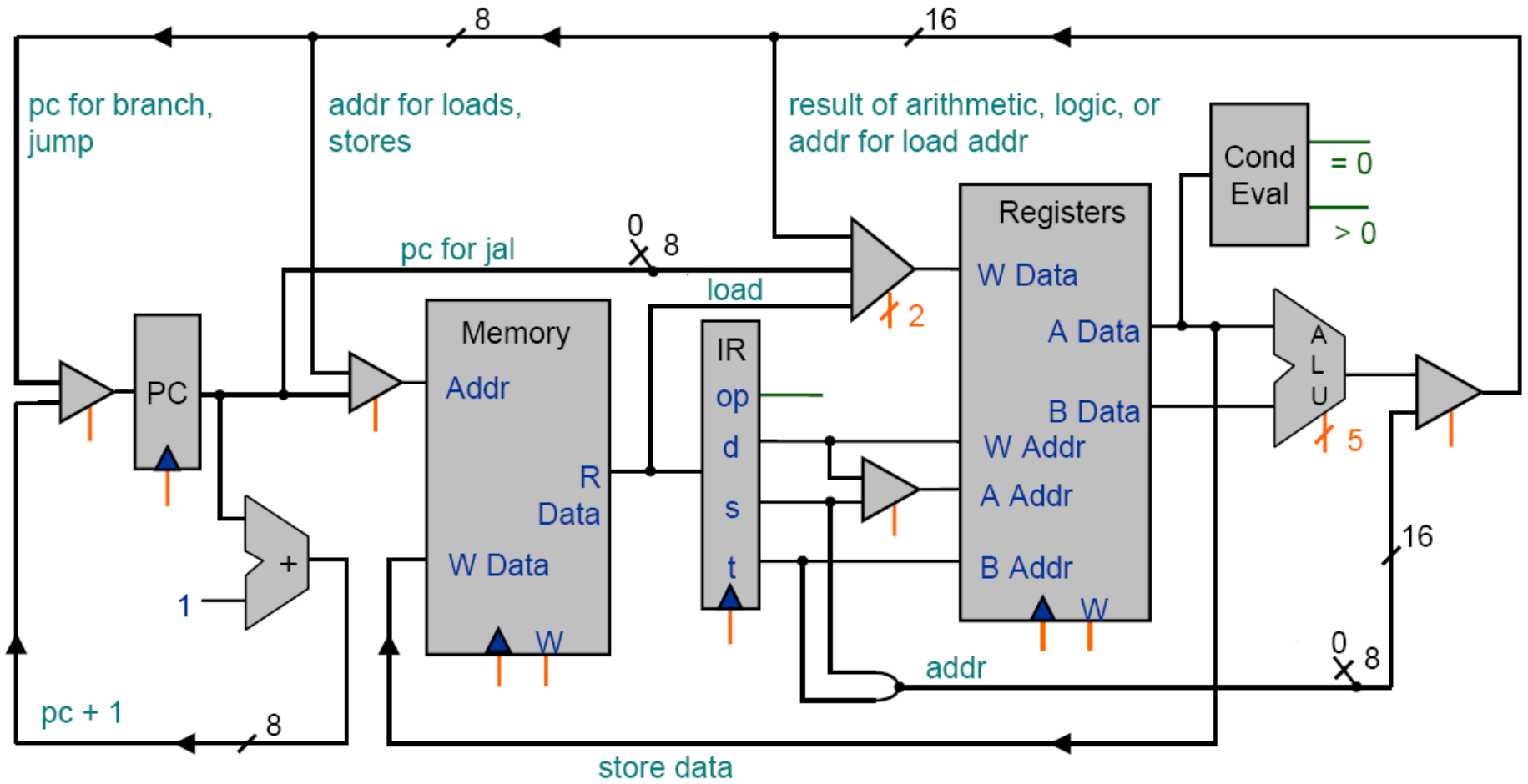
# TOY Reference Card

	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							

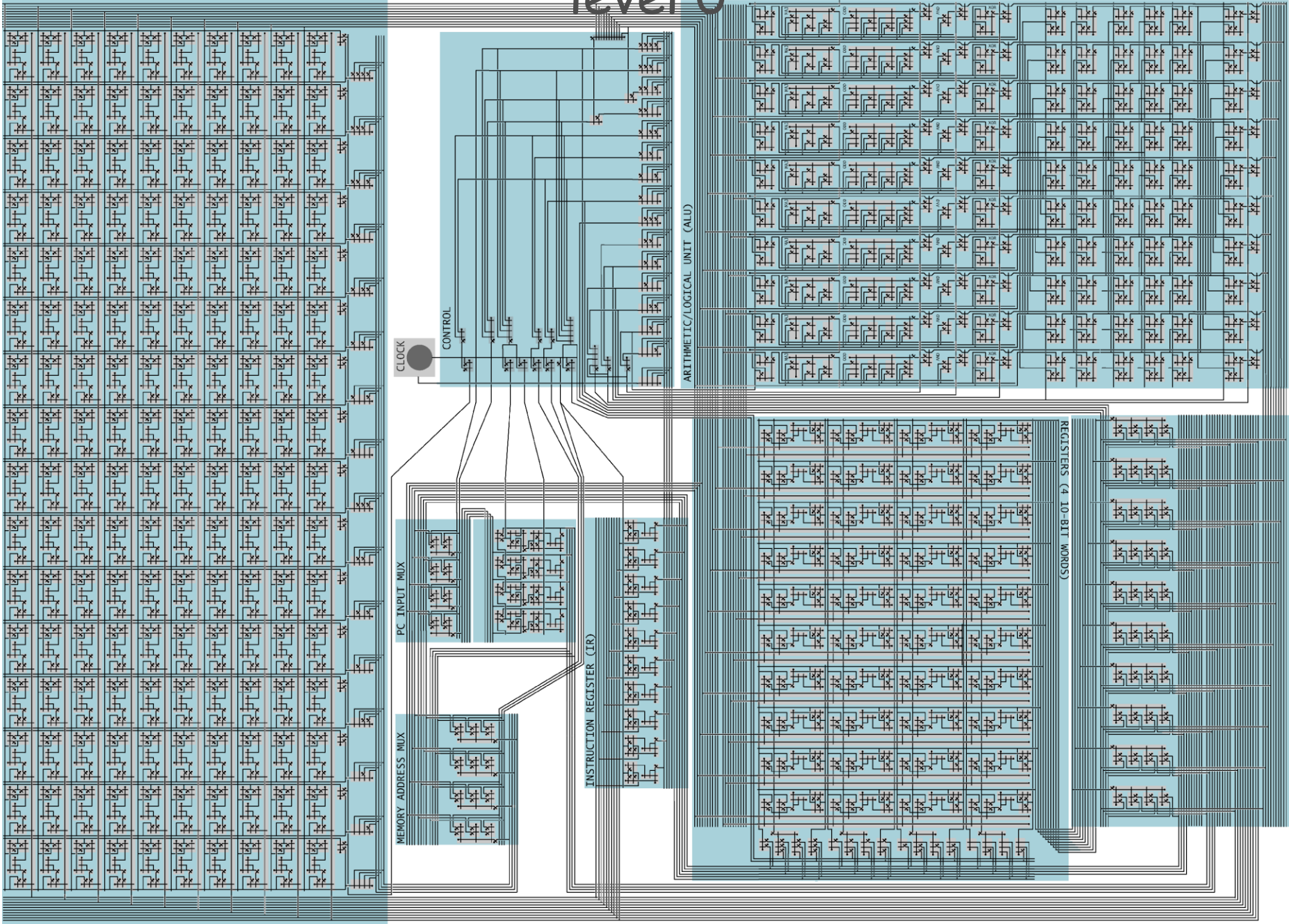
#	Operation	Fmt	Pseudocode
0:	halt	1	<code>exit(0)</code>
1:	add	1	<code>R[d] ← R[s] + R[t]</code>
2:	subtract	1	<code>R[d] ← R[s] - R[t]</code>
3:	and	1	<code>R[d] ← R[s] &amp; R[t]</code>
4:	xor	1	<code>R[d] ← R[s] ^ R[t]</code>
5:	shift left	1	<code>R[d] ← R[s] &lt;&lt; R[t]</code>
6:	shift right	1	<code>R[d] ← R[s] &gt;&gt; R[t]</code>
7:	load addr	2	<code>R[d] ← addr</code>
8:	load	2	<code>R[d] ← mem[addr]</code>
9:	store	2	<code>mem[addr] ← R[d]</code>
A:	load indirect	1	<code>R[d] ← mem[R[t]]</code>
B:	store indirect	1	<code>mem[R[t]] ← R[d]</code>
C:	branch zero	2	<code>if (R[d] == 0) pc ← addr</code>
D:	branch positive	2	<code>if (R[d] &gt; 0) pc ← addr</code>
E:	jump register	1	<code>pc ← R[t]</code>
F:	jump and link	2	<code>R[d] ← pc; pc ← addr</code>

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

# TOY Architecture (level 1)



# level 0



# Programming in TOY

Hello, World. Add two numbers.

- Adds  $8 + 5 = 13$ .

# 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

10: 8A00    RA ← mem[00]
11: 8B01    RB ← mem[01]
12: 1CAB    RC ← RA + RB
13: 9CFF    mem[FF] ← 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	pc
0000	0000	0000	10

Registers

00: 0008	8	add.toy
01: 0005	5	
10: 8A00	RA ← mem[00]	
11: 8B01	RB ← mem[01]	
12: 1CAB	RC ← RA + RB	
13: 9CFF	mem[FF] ← 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 <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	pc
0008	0000	0000	11

Registers

00: 0008	8	add.toy
01: 0005	5	
10: 8A00	RA	← mem[00]
11: 8B01	RB	← mem[01]
12: 1CAB	RC	← RA + RB
13: 9CFF	mem[FF]	← 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 adds 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	
10: 8A00	RA	← mem[00]
11: 8B01	RB	← mem[01]
12: 1CAB	RC	← RA + RB
13: 9CFF	mem[FF]	← 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 <sub>16</sub>				C <sub>16</sub>				A <sub>16</sub>				B <sub>16</sub>			
opcode				dest d				source s				source t			



# Store

## Store. (opcode 9)

- Stores the contents of some register into a memory cell.
- `9CFF` means store the contents of register `c` into memory cell `FF` (stdout).

RA	RB	RC	pc
0008	0005	000D	13

Registers

00:	0008	8	add.toy
01:	0005	5	
10:	8A00	RA	← mem[00]
11:	8B01	RB	← mem[01]
12:	1CAB	RC	← RA + RB
13:	9CFF	mem[FF]	← 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
10: 8A00    RA ← mem[00]
11: 8B01    RB ← mem[01]
12: 1CAB    RC ← RA + RB
13: 9CFF    mem[FF] ← RC
14: 0000    halt
```

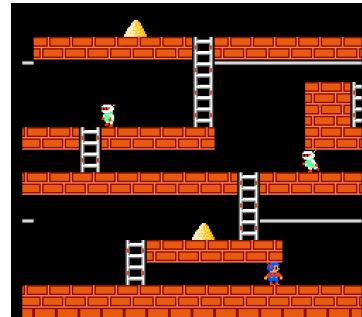
# Simulation

## Consequences of simulation.

- ♦ Test out new machine or microprocessor using simulator.
  - cheaper and faster than building actual machine
- ♦ Easy to add new functionality to simulator.
  - trace, single-step, breakpoint debugging
  - simulator more useful than TOY itself
- ♦ Reuse software from old machines.

## Ancient programs still running on modern computers.

- ♦ Lode Runner on Apple IIe.
- ♦ Gameboy simulator on PCs.



# 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:

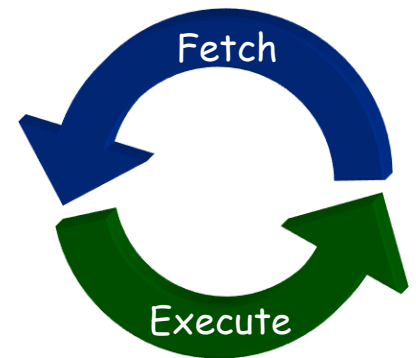
- ♦ Set 8 memory address switches.
- ♦ Press LOOK: contents of addressed word appears in lights.



# 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.



# Branch in TOY

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  $c = a \times b$ .
- 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

```
int a = 3;  
int b = 9;  
int c = 0;  
  
while (a != 0) {  
    c = c + b;  
    a = a - 1;  
}
```



# 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: 9CFF  mem[FF] ← 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:	9CFF				
19:	0000				

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

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

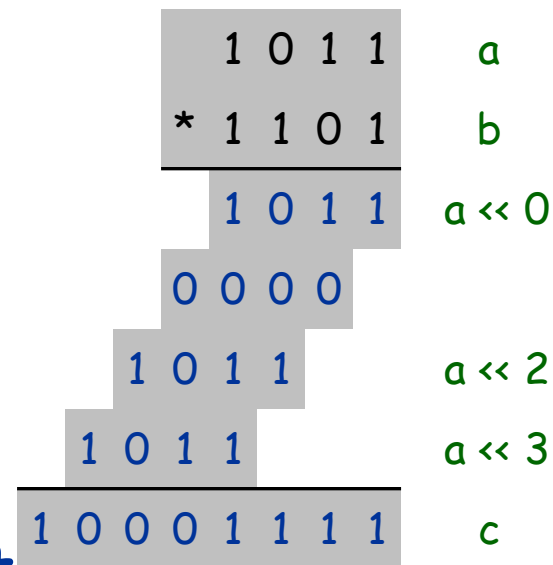
Binary

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

# Binary Multiplication

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

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



Implement with built-in TOY shift instructions.

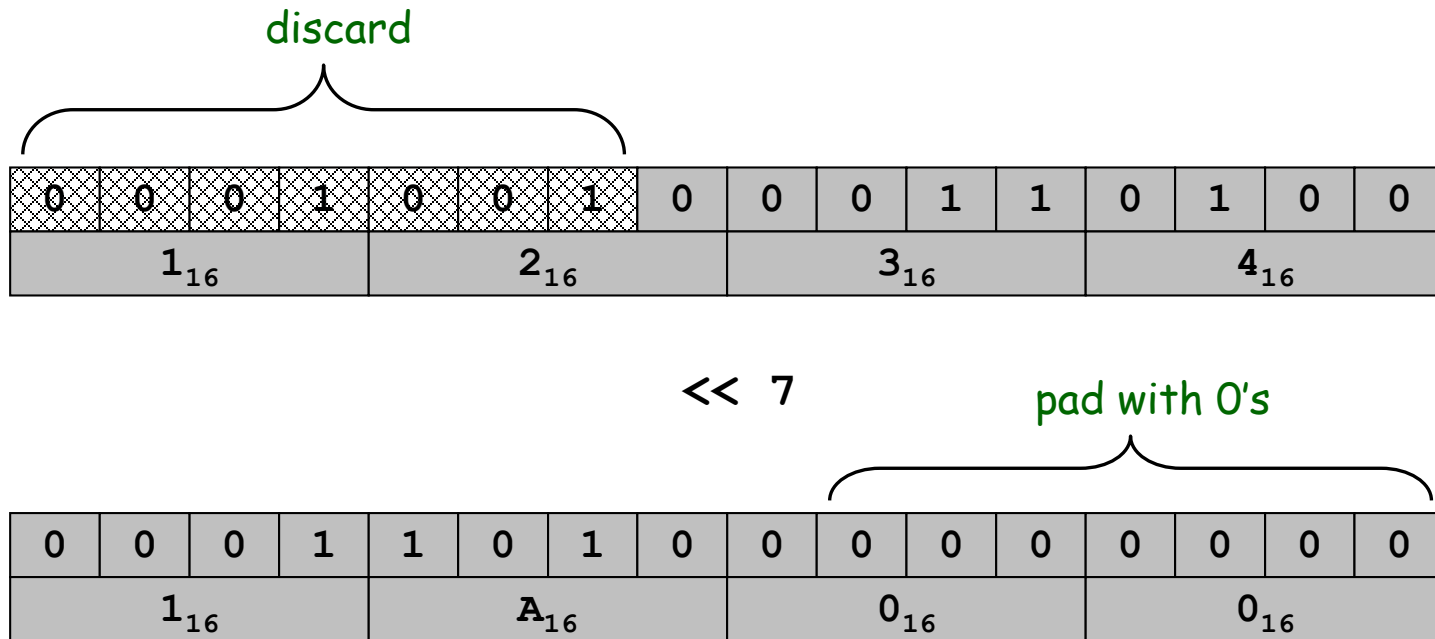
```
int c = 0;
for (int i = 15; i >= 0; i--)
    if (((b >> i) & 1) == 1)
        c = c + (a << i);
```

←  $b_i = i^{\text{th}}$  bit of  $b$

# Shift Left

## Shift left. (opcode 5)

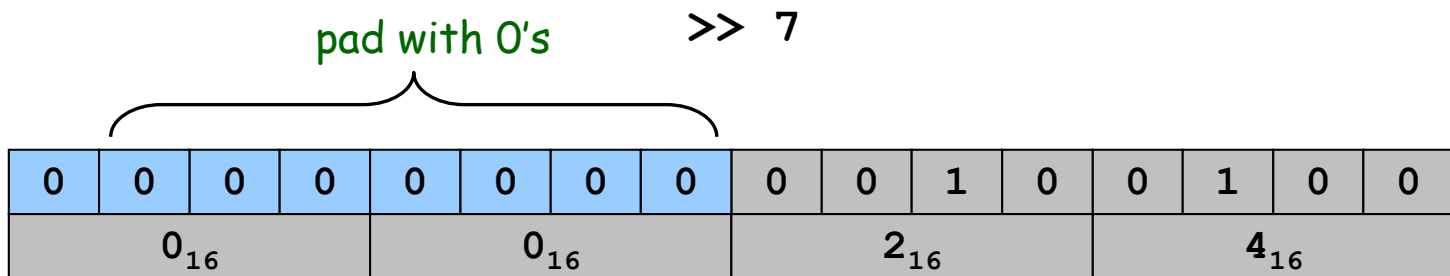
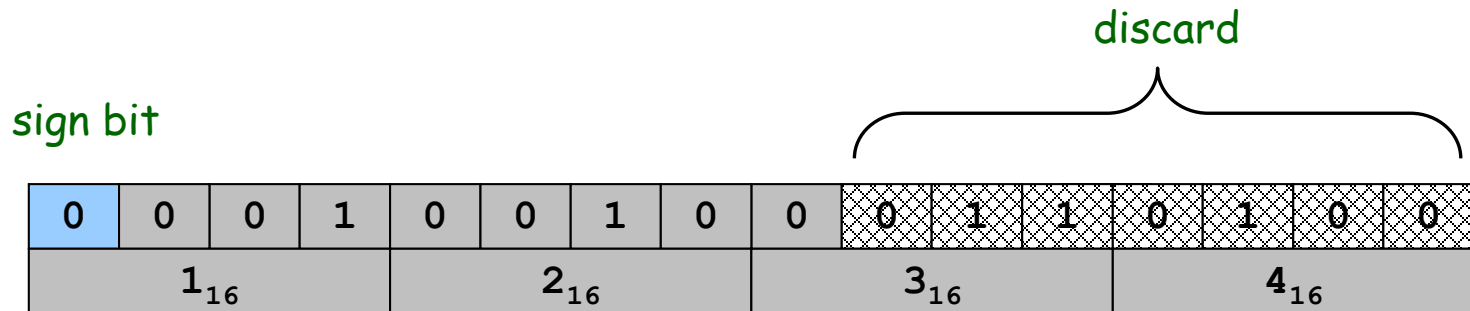
- Move bits to the left, padding with zeros as needed.
- $1234_{16} \ll 7_{16} = 1A00_{16}$



# Shift Right

## Shift right. (opcode 6)

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



# Shift Right (Sign Extension)

## Shift right. (opcode 6)

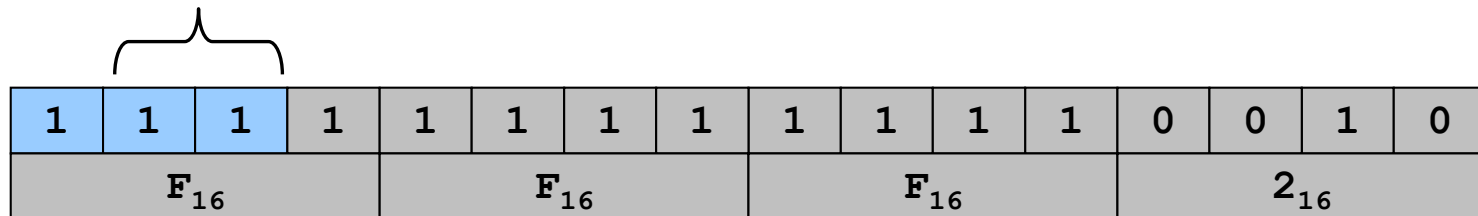
- Move bits to the right, padding with sign bit as needed.
- $\text{FFCA}_{16} \gg 2_{16} = \text{FFF2}_{16}$
- $-53_{10} \gg 2_{10} = -13_{10}$

sign bit



pad with 1s

$\gg 2$

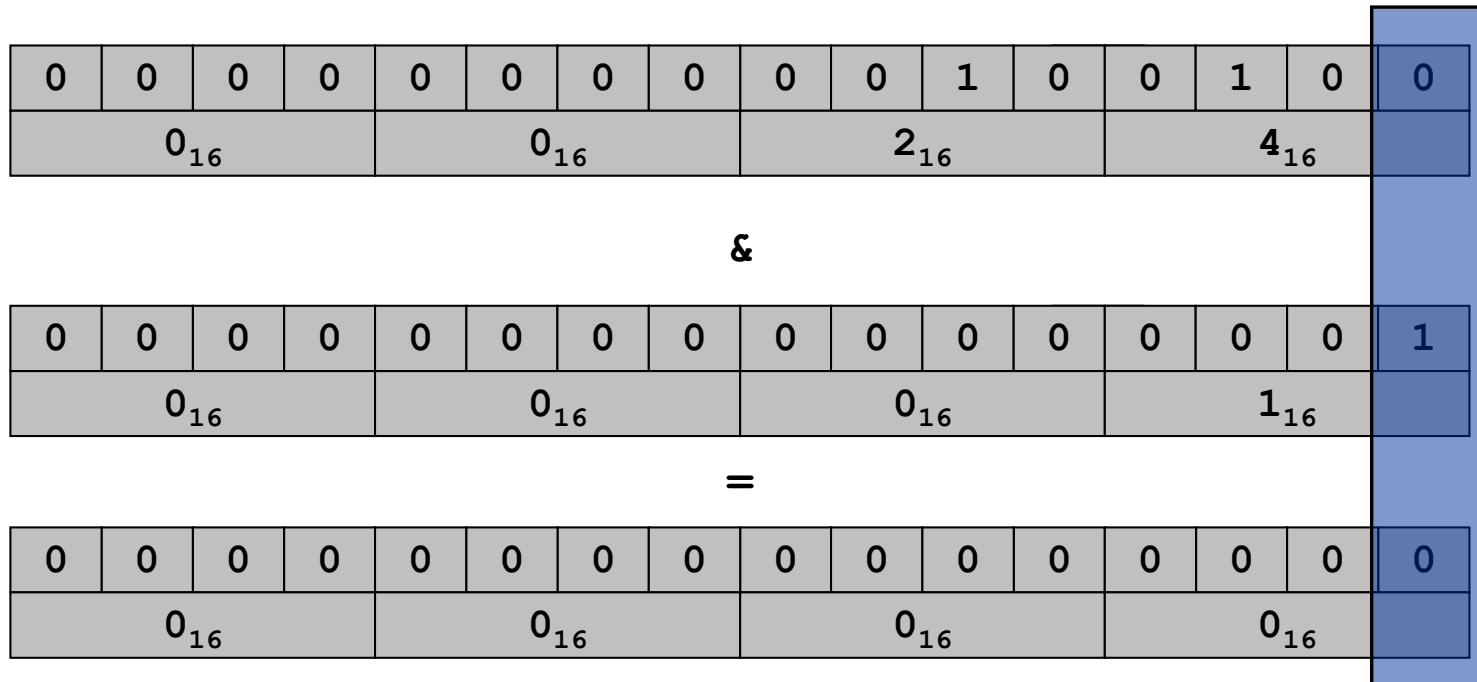


# Bitwise AND

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

## Logical AND. (opcode 3)

- 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

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

# 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: 9CFE    mem[FF] ← 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

# 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
```

# Standard Input and Output: Implications

## Standard input and output enable you to:

- Process more information than fits in memory.
- Interact with the computer while it is running.

## Standard output.

- Writing to memory location `FF` sends one word to TOY stdout.
- `9AFF` writes the integer in register `A` to stdout.

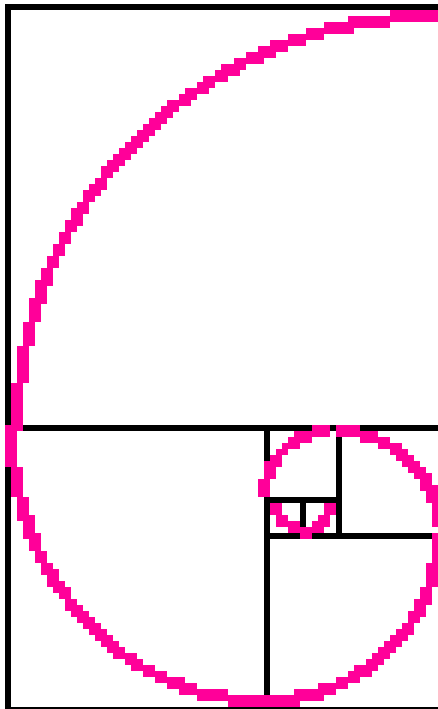
## Standard input.

- Loading from memory address `FF` loads one word from TOY stdin.
- `8AFF` reads in an integer from stdin and store it in register `A`.

# Fibonacci Numbers

Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . .

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$



# Standard Output

```
00: 0000    0
01: 0001    1

10: 8A00    RA ← mem[00]    a = 0
11: 8B01    RB ← mem[01]    b = 1
                    do {
12: 9AFF    print RA        print a
13: 1AAB    RA ← RA + RB    a = a + b
14: 2BAB    RB ← RA - RB    b = a - b
15: DA12    if (RA > 0) goto 12    } while (a > 0)
16: 0000    halt
```

fibonacci.toy

```
0000
0001
0001
0002
0003
0005
0008
000D
0015
0022
0037
0059
0090
00E9
0179
0262
03DB
063D
0A18
1055
1A6D
2AC2
452F
6FF1
```

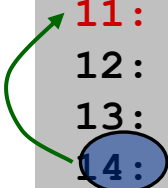
# Standard Input

Ex: read in a sequence of integers and print their sum.

- In Java, stop reading when EOF.
- In TOY, stop reading when user enters 0000.

```
while(!StdIn.isEmpty()) {  
    a = StdIn.readInt();  
    sum = sum + a;  
}  
System.out.println(sum);
```

00:	0000	0
10:	8C00	RC ← mem[00]
11:	8AFF	read RA
12:	CA15	if (RA == 0) pc ← 15
13:	1CCA	RC ← RC + RA
14:	C011	pc ← 11
15:	9CFF	write RC
16:	0000	halt



00AE
0046
0003
0000
00F7



# Load Address (a.k.a. Load Constant)

## Load address. (opcode 7)

- Loads an 8-bit integer into a register.
- 7A30 means load the value 30 into register A.

## Applications.

- Load a small constant into a register.
- Load a 8-bit memory address into a register.
  - register stores "pointer" to a memory cell

```
a = 30;
```

Java code

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

# Arrays in TOY

TOY main memory is a giant array.

- Can access memory cell 30 using load and store.
- 8c30 means load `mem[30]` into register c.
- Goal: access memory cell `i` where `i` is a variable.

Load indirect. (opcode A)

- AC06 means load `mem[R6]` into register c.

↑  
a variable index (like a pointer)

Store indirect. (opcode B)

- BC06 means store contents of register c into `mem[R6]`.

↑  
a variable index

```
for (int i = 0; i < N; i++)
    a[i] = StdIn.readInt();

for (int i = 0; i < N; i++)
    System.out.println(a[N-i-1]);
```

Reverse.java

# TOY Implementation of Reverse

## TOY implementation of reverse.

- ♦ Read in a sequence of integers and store in memory  
30, 31, 32, ...
- ♦ Stop reading if 0000.
- ♦ Print sequence in reverse order.

# TOY Implementation of Reverse

## TOY implementation of reverse.

- ♦ Read in a sequence of integers and store in memory  
30, 31, 32, ...
- ♦ Stop reading if 0000.
- ♦ Print sequence in reverse order.

```
10: 7101  R1 ← 0001
11: 7A30  RA ← 0030
12: 7B00  RB ← 0000
```

```
constant 1
a[]
n
```

```
13: 8CFF  read RC
14: CC19  if (RC == 0) goto 19
15: 16AB  R6 ← RA + RB
16: BC06  mem[R6] ← RC
17: 1BB1  RB ← RB + R1
18: C013  goto 13
```

```
while(true) {
    c = StdIn.readInt();
    if (c == 0) break;
    address of a[n]
    a[n] = c;
    n++;
}
```

read in the data

# TOY Implementation of Reverse

## TOY implementation of reverse.

- Read in a sequence of integers and store in memory  
30, 31, 32, ...
- ➔ • Stop reading if 0000.
- Print sequence in reverse order.

```
19: CB20  if (RB == 0) goto 20          while (n > 0) {
1A: 16AB  R6 ← RA + RB                    address of a[n]
1B: 2661  R6 ← R6 - R1                address of a[n-1]
1C: AC06  RC ← mem[R6]              c = a[n-1];
1D: 9CFF  write RC                    System.out.println(c);
1E: 2BB1  RB ← RB - R1                n--;
1F: C019  goto 19                      }
20: 0000  halt                        print in reverse order
```

# Unsafe Code at any Speed

What happens if we make array start at 00 instead of 30?

- Self modifying program.
- Exploit buffer overrun and run arbitrary code!

```
10: 7101  R1 ← 0001
11: 7A00  RA ← 0000
12: 7B00  RB ← 0000
```

```
13: 8CFF  read RC
14: CC19  if (RC == 0) goto 19
15: 16AB  R6 ← RA + RB
16: BC06  mem[R6] ← RC
17: 1BB1  RB ← RB + R1
18: C013  goto 13
```



```
constant 1
a[]
n

while(true) {
    c = StdIn.readInt();
    if (c == 0) break;
    address of a[n]
    a[n] = c;
    n++;
}
```

Crazy 8s Input

```
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
8888 8810
98FF C011
```

# What Can Happen When We Lose Control?

## Buffer overrun.

- Array `buffer[]` has size 100.
- User might enter 200 characters.
- Might lose control of machine behavior.
- Majority of viruses and worms caused by similar errors.

```
#include <stdio.h>
int main(void) {
    char buffer[100];
    scanf("%s", buffer);
    printf("%s\n", buffer);
    return 0;
}
```

unsafe C program

## Robert Morris Internet Worm.

- Cornell grad student injected worm into Internet in 1988.
- Exploited buffer overrun in finger daemon `fingerd`.

# Function Call: A Failed Attempt

Goal:  $x \times y \times z$ .

- Need two multiplications:  $x \times y$ ,  $(x \times y) \times z$ .
  - ✎ Solution 1: write multiply code 2 times.
  - ✎ Solution 2: write a TOY function.

## A failed attempt:

- Write multiply loop at 30-36.
- Calling program agrees to store arguments in registers A and B.
- Function agrees to leave result in register C.
- Call function with jump absolute to 30.
- Return from function with jump absolute.

## Reason for failure.

- ✎ Need to return to a VARIABLE memory address.

function?	
10:	8AFF
11:	8BFF
12:	C030
13:	1AC0
14:	8BFF
15:	C030
16:	9CFF
17:	0000
30:	7C00
31:	7101
32:	CA36
33:	1CCB
34:	2AA1
35:	C032
36:	C013?



# Multiplication Function

## Calling convention.

- Jump to line 30.
- Store a and b in registers A and B.
- Return address in register F.
- Put result  $c = a \times b$  in register C.
- Register 1 is scratch.
- Overwrites registers A and B.

### function.toy

```
30: 7C00  R[C] ← 00
31: 7101  R[1] ← 01
32: CA36  if (R[A] == 0) goto 36
33: 1CCB  R[C] += R[B]
34: 2AA1  R[A]--
35: C032  goto 32
36: EF00  pc ← R[F]
```

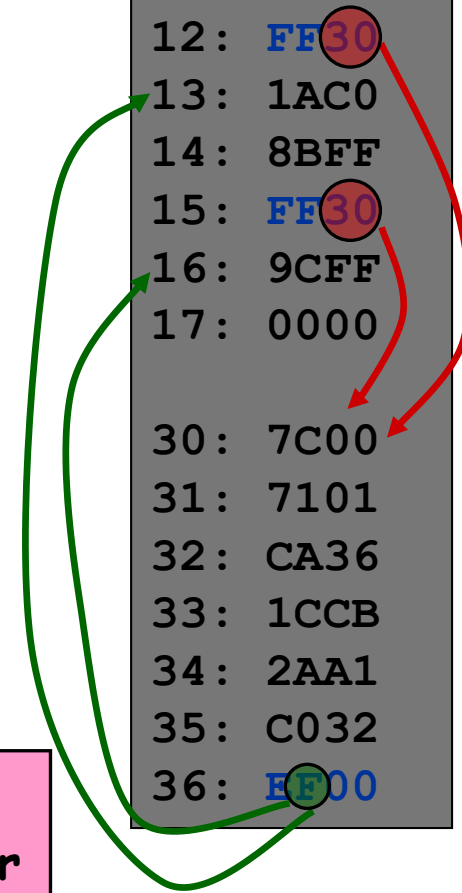
return

opcode E  
jump register

### function

```
10: 8AFF
11: 8BFF
12: FF30
13: 1AC0
14: 8BFF
15: FF30
16: 9CFF
17: 0000

30: 7C00
31: 7101
32: CA36
33: 1CCB
34: 2AA1
35: C032
36: EF00
```



# Multiplication Function Call

## Client program to compute $x \times y \times z$ .

- Read  $x, y, z$  from standard input.
- Note: PC is incremented before instruction is executed.
  - value stored in register F is correct return address

### function.toy (cont)

10:	8AFF	read R[A]	x
11:	8BFF	read R[B]	y
→ 12:	FF30	R[F] ← pc; goto 30	x * y
13:	1AC0	R[A] ← R[C]	(x * y)
14:	8BFF	read R[B]	z
→ 15:	FF30	R[F] ← pc; goto 30	(x * y) * z
16:	9CFF	write R[C]	
17:	0000	halt	

opcode F  
jump and link

← R[F] ←  
13

← R[F] ←  
16

# Function Call: One Solution

## Contract between calling program and function:

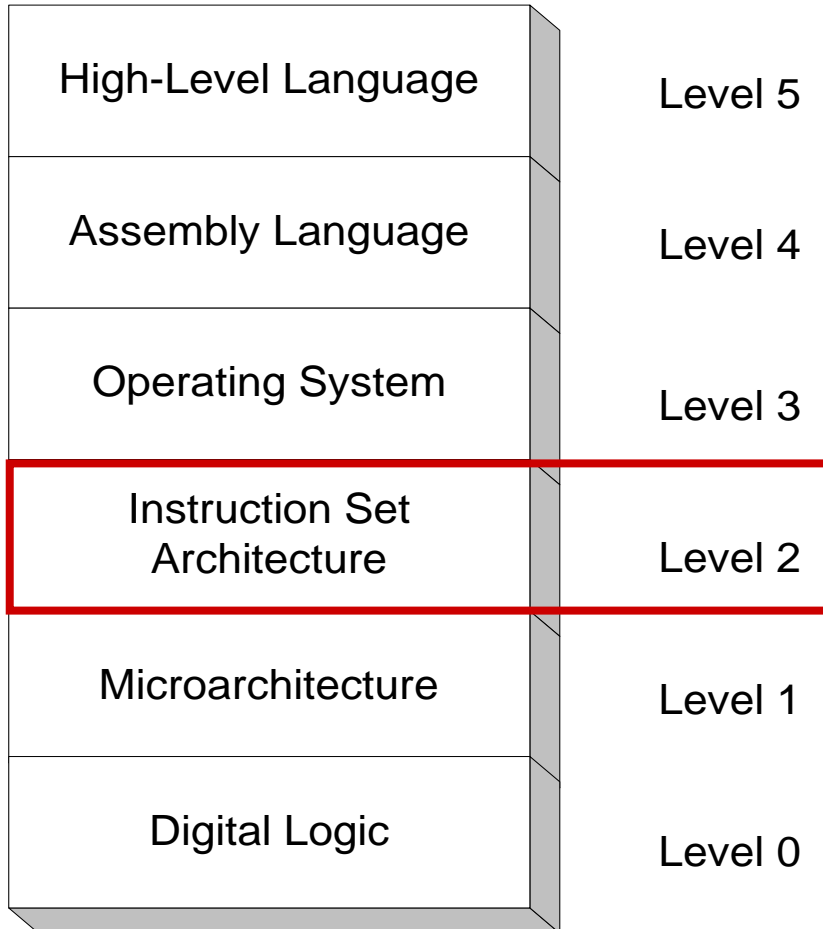
- Calling program stores function parameters in specific registers.
- Calling program stores return address in a specific register.
  - jump-and-link
- Calling program sets PC to address of function.
- Function stores return value in specific register.
- Function sets PC to return address when finished.
  - jump register

## What if you want a function to call another function?

- Use a different register for return address.
- More general: store return addresses on a stack.

# Virtual machines

## Abstractions for computers



	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 <u>addr</u>	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 $(R[d] == 0)$ $pc \leftarrow \text{addr}$
D:	branch positive	2	if $(R[d] > 0)$ $pc \leftarrow \text{addr}$
E:	jump register	1	$pc \leftarrow R[t]$
F:	jump and link	2	$R[d] \leftarrow pc; pc \leftarrow \text{addr}$

10: C020  
 20: 7101  
 21: 7A00  
 22: 7C00  
 23: 8DFF  
 24: CD29  
 25: 12AC  
 26: BD02  
 27: 1CC1  
 28: C023  
 29: FF2B  
 2A: 0000

# Problems with programming using machine code

- ◆ Difficult to remember instructions
- ◆ Difficult to remember variables
- ◆ Hard to calculate addresses/relocate variables or functions
- ◆ Need to handle instruction encoding

Table B.1 ARM instruction decode table.

Instruction classes (indexed by op)	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
AND   EOR   SUB   RSB   ADD   ADC   SBC   RSC	cond	0	0	0	0	op	S	Rn	Rd	shift_size	shift	0	Rm																							
AND   EOR   SUB   RSB   ADD   ADC   SBC   RSC	cond	0	0	0	0	op	S	Rn	Rd	Rs	0	shift	1	Rm																						
MUL	cond	0	0	0	0	0	0	0	S	Rd	0	0	0	0	Rs	1	0	0	1	Rm																
MLA	cond	0	0	0	0	0	0	1	S	Rd	Rn	Rs	1	0	0	1	Rm																			
UMAAL	cond	0	0	0	0	0	1	0	0	RdHi	RdLo	Rs	1	0	0	1	Rm																			
UMULL   UMLAL   SMULL   SMLAL	cond	0	0	0	0	1	op	S	RdHi	RdLo	Rs	1	0	0	1	Rm																				
STRH   LDRH   post	cond	0	0	0	0	U	0	0	op	Rn	Rd	0	0	0	0	1	0	1	1	Rm																
LDRD   STRD   LDRSB   LDRSH   post	cond	0	0	0	0	U	0	0	op	Rn	Rd	0	0	0	0	1	1	op	1	Rm																
LDRD   STRD   LDRSB   LDRSH   post	cond	0	0	0	0	U	1	0	op	Rn	Rd	immed	1	1	op	1																				
MRS Rd, cpsr   MRS Rd, spsr	cond	0	0	0	1	op	0	0	1	1	1	1	Rd	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0				
MSR cpsr, Rm   MSR spsr, Rm	cond	0	0	0	1	op	1	0	f	s	x	c	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0			
BLX	cond	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	0	0	1	0	Rm														
SMLaxy	cond	0	0	0	1	0	0	0	0	Rd	Rn	Rs	1	y	x	0	Rm																			
SMLawy	cond	0	0	0	1	0	0	1	0	Rd	Rn	Rs	1	y	0	0	Rm																			
SMULwy	cond	0	0	0	1	0	0	1	0	Rd	0	0	0	0	Rs	1	y	1	0	Rm																
SMLALxy	cond	0	0	0	1	0	1	0	0	RdHi	RdLo	Rs	1	y	x	0	Rm																			
SMULxy	cond	0	0	0	1	0	1	1	0	Rd	0	0	0	0	Rs	1	y	x	0	Rm																
TST   TEQ   CMP   CMN	cond	0	0	0	1	op	1	Rn	0	0	0	0	shift_size	shift	0	Rm																				
ORR   BIC	cond	0	0	0	1	op	0	S	Rn	Rd	shift_size	shift	0	Rm																						
MOV   MVN	cond	0	0	0	1	op	1	S	0	0	0	0	Rd	shift_size	shift	0	Rm																			
BX   BLX	cond	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	0	op	1	Rm															
CLZ	cond	0	0	0	1	0	1	1	0	1	1	1	1	Rd	1	1	1	1	0	0	1	Rm														
QADD   QSUB   QDADD   QDSUB	cond	0	0	0	1	op	1	0	Rn	Rd	0	0	0	0	1	0	1	0	Rm																	
BKPT	1	1	1	0	0	0	0	1	0	0	1	0	immed[15:4]			0	1	1	1	immed																
TST   TEQ   CMP   CMN	cond	0	0	0	1	op	1	Rn	0	0	0	0	Rs	0	shift	1	Rm																			
ORR   BIC	cond	0	0	0	1	op	0	S	Rn	Rd	Rs	0	shift	1	Rm																					
MOV   MVN	cond	0	0	0	1	op	1	S	0	0	0	0	Rd	Rs	0	shift	1	Rm																		
SWP   SWPB	cond	0	0	0	1	op	0	0	Rn	Rd	0	0	0	0	1	0	0	1	Rm																	
STREX	cond	0	0	0	1	0	0	0	0	Rn	Rd	1	1	1	1	0	0	1	Rm																	
LDRDX	cond	0	0	0	1	1	0	0	1	Rn	Rd	1	1	1	1	0	0	1	1	1	Rm															

Table B.1 ARM instruction decode table. (Continued.)

Instruction classes (indexed by op)	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
STRH   LDRH   pre	cond	0	0	0	1	U	0	W	op	Rn	Rd	0	0	0	0	1	0	1	1	Rm														
STRH   LDRH   pre	cond	0	0	0	1	U	1	W	op	Rn	Rd	immed	1	0	1	1	0	1	1	immed														
LDRD   STRD   LDRSB   LDRSH   pre	cond	0	0	0	1	U	0	W	op	Rn	Rd	0	0	0	0	1	1	op	1	Rm														
LDRD   STRD   LDRSB   LDRSH   pre	cond	0	0	0	1	U	1	W	op	Rn	Rd	immed	1	1	op	1																		
AND   EOR   SUB   RSB   ADD   ADC   SBC   RSC	cond	0	0	1	0	op	S	Rn	Rd	rotate	immed																							
MSR cpsr, #imm   MSR spsr, #imm	cond	0	0	1	1	op	1	0	f	s	x	c	1	1	1	1	rotate	immed																
TST   TEQ   CMP   CMN	cond	0	0	1	1	op	1	Rn	0	0	0	0	rotate	immed																				
ORR   BIC	cond	0	0	1	1	op	0	S	Rn	Rd	rotate	immed																						
MOV   MVN	cond	0	0	1	1	op	1	S	0	0	0	0	Rd	rotate	immed																			
STR   LDR   STRB   LDRB   post	cond	0	1	0	0	U	op	T	op	Rn	Rd																							
STR   LDR   STRB   LDRB   post	cond	0	1	0	1	U	op	W	op	Rn	Rd																							
STR   LDR   STRB   LDRB   post	cond	0	1	1	0	U	op	T	op	Rn	Rd	shift_size	shift	0	Rm																			
{ S   Q   SH }   { U   UQ   UH }   ADD16	cond	0	1	1	0	op	Rn	Rd	1	1	1	1	0	0	0	1	Rm																	
{ S   Q   SH }   { U   UQ   UH }   ADDSUBX	cond	0	1	1	0	op	Rn	Rd	1	1	1	1	0	0	1	1	Rm																	
{ S   Q   SH }   { U   UQ   UH }   SUBADDX	cond	0	1	1	0	op	Rn	Rd	1	1	1	1	0	1	0	1	Rm																	
{ S   Q   SH }   { U   UQ   UH }   SUB16	cond	0	1	1	0	op	Rn	Rd	1	1	1	1	0	1	1	1	Rm																	
{ S   Q   SH }   { U   UQ   UH }   ADD8	cond	0	1	1	0	op	Rn	Rd	1	1	1	1	1	0	0	1	Rm																	
{ S   Q   SH }   { U   UQ   UH }   SUB8	cond	0	1	1	0	op	Rn	Rd	1	1	1	1	1	1	1	1	Rm																	
PKHBT   PKHTB	cond	0	1	1	0	1	0	0	0	Rn	Rd	shift_size	op	0	1	Rm																		
{ S   U }   SAT	cond	0	1	1	0	1	op	1	immed5	Rd	shift_size	sh	0	1	Rm																			
{ S   U }   SAT16	cond	0	1	1	0	1	op	1	0	immed4	Rd	1	1	1	1	0	0	1	1	Rm														
SEL	cond	0	1	1	0	1	0	0	0	Rd	Rd	1	1	1	1	0	0	1	1	Rm														
REV   REV16     REVSH	cond	0	1	1	0	1	op	1	1	1	1	1	1	1	Rd	1	1	1	0	op	1	1	Rm											
{ S   U }   XTAB16	cond	0	1	1	0	1	op	0	0	Rn!	1	1	1	1	Rd	rot	0	0	0	1	1	Rm												
{ S   U }   XTAB16	cond	0	1	1	0	1	op	0	0	1	1	1	1	Rd	rot	0	0	0	1	1	1	Rm												
{ S   U }   XTAB	cond	0	1	1	0	1	op	1	0	Rn!	1	1	1	1	Rd	rot	0	0	0	1	1	1	Rm											
{ S   U }   XTAB	cond	0	1	1	0	1	op	1	0	1	1	1	1	Rd	rot	0	0	0	1	1	1	1	Rm											
{ S   U }   XTAB	cond	0	1	1	0	1	op	1	0	1	1	1	1	Rd	rot	0	0	0	1	1	1													

# Virtual machines

## Abstractions for computers

