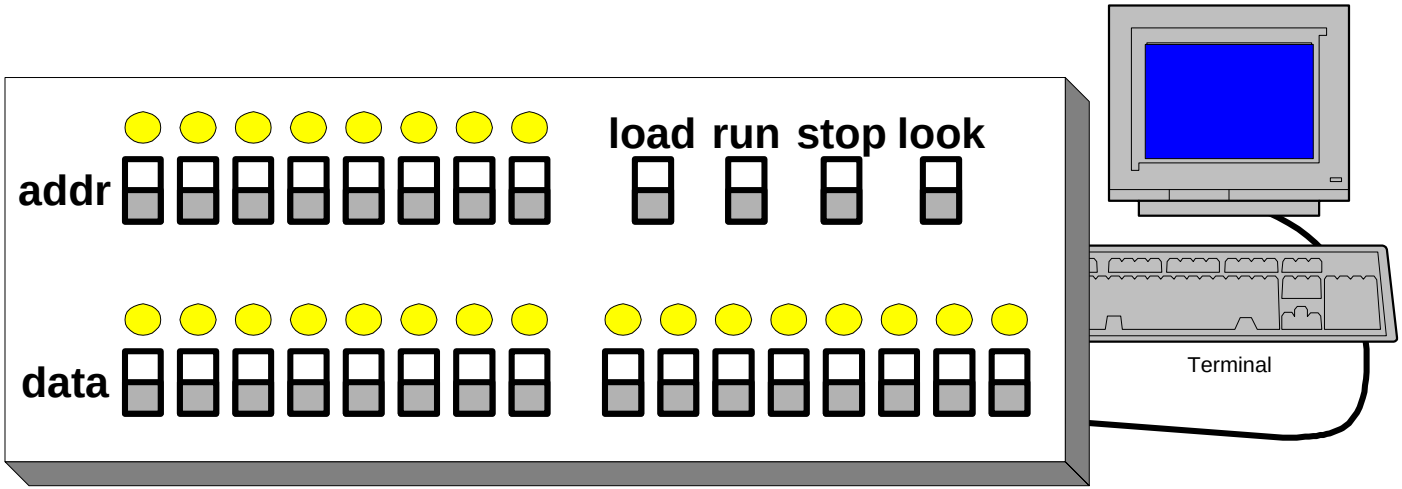
Lecture 7. The TOY Machine

- **TOY is an imaginary machine similar to**

  early computers

  1980s microprocessers

- **Box with switches, lights, terminal**



- **TOY helps introduce**

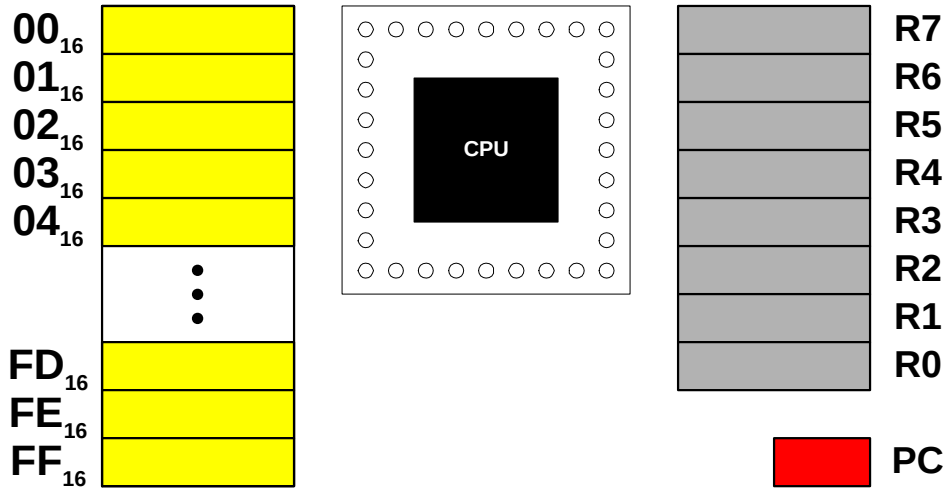
  machine-language **programming (how a C program is 'mapped' onto a machine)**

  computer *architecture* (how the machine works)

- **With enough memory and time, TOY can compute *anything* a supercomputer can**

Inside the Box

- 1 central processing unit (CPU)
- 256 16-bit ***words*** of memory
- 8 16-bit ***registers***
- 1 8-bit ***program counter*** (PC) register — the address of the 'current' instruction
- Machine consists of 'On/Off' switches and lights



- Numbers are encoded in base 2, e.g., $6375_{10} = 0001\ 1000\ 1110\ 0111_2$
- Operation
  1. Load the program and the data into memory using the *addr*, *data*, and *load* switches
  2. Set the *addr* switches to the address of the first instruction
  3. Press *run*
  4. To examine memory — the 'output' — set *addr* switches to the desired address, press *look*, read the *data* lights
- ***Everything*** is encoded in binary — data, machine instructions, text, addresses, …

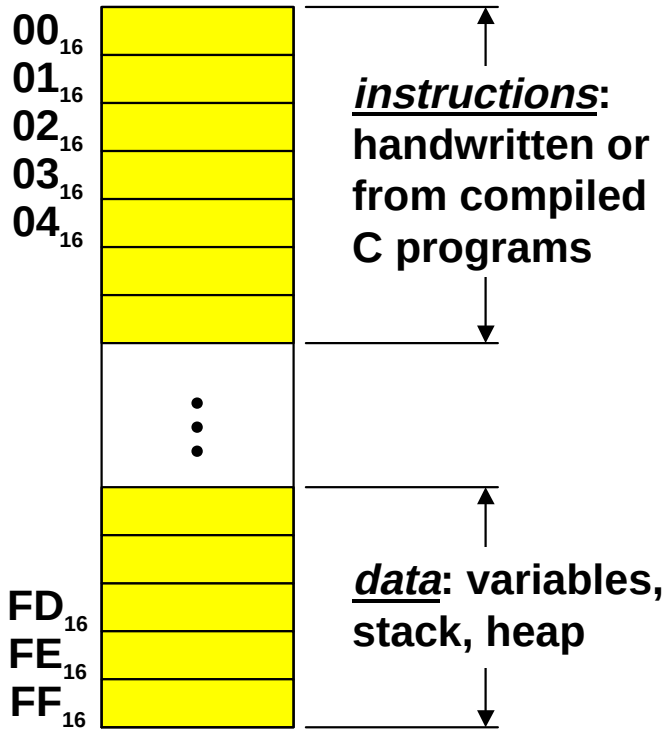
Memory

- **'Dump' of machine state in hexadecimal includes the registers, PC, and memory**

```
PC = 000C

R0 = 0000  R1 = 0037  R2 = 0001  R3 = FFFF
R4 = 0000  R5 = 0000  R6 = 0008  R7 = 00FF

00: 0000 0000 0000 0000 0000 0000 0000 0000
08: 0000 0000 0000 0000 0000 0000 0000 0000
10: 9222 9120 1121 A120 1121 A121 7211 0000
18: 0000 0000 0000 0000 0000 0000 0000 0000
20: 0000 0001 0010 0000 0000 0000 0000 0000
28: 0000 0000 0000 0000 0000 0000 0000 0000
...
E8: 0000 0000 0000 0000 0000 0000 0000 0000
F0: 0000 0000 0000 0000 0000 0000 0000 0000
F8: 0000 0000 0000 0000 0000 0000 0000 0000
```



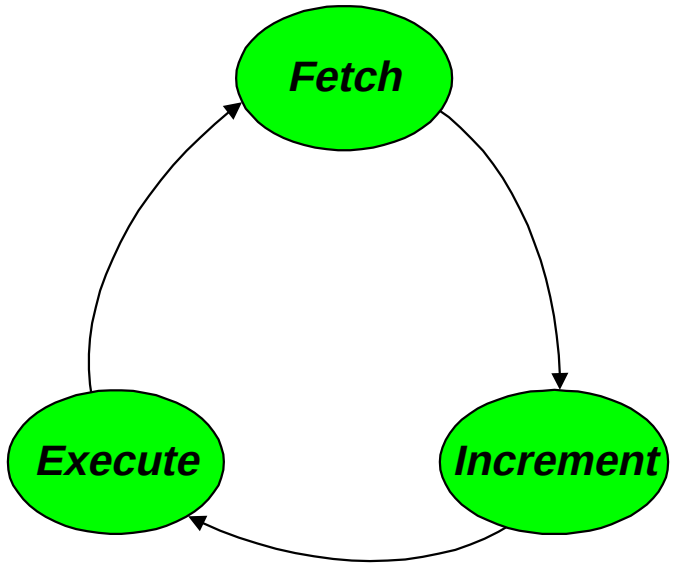
- **Programmers still look at dumps, even in the 90s**
- **Machine state**

  records what a program has done

  determines what the machine will do

Basic Cycle

- When you press *Run*
  1. ***Fetch***: load the instruction at the address given by the PC into the CPU
  2. ***Increment*** the PC by 1
  3. ***Execute*** the instruction, which may load/store data from/to memory
  4. Continue this ***fetch-increment-execute cycle*** until a halt is executed



- Instructions make well-defined changes to the registers, memory, and the PC

Digression: Number Systems

- **The general form of an integer in *base* b is**

$$x = x_n b^n + x_{n-1} b^{n-1} + \ldots + x_1 b^1 + x_0 b^0$$

The x_i are the *positional coefficients*

- **Modern computers use binary arithmetic — base 2**

$$\begin{aligned} 140_{10} &= 1 \times 10^2 + 4 \times 10^1 + 0 \times 10^0 \\ &= 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &= 10001100_2 \end{aligned}$$

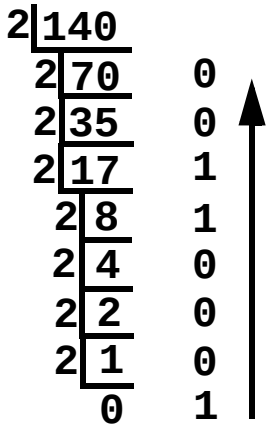
- **Base 2 is easily converted to base 8 (octal) and base 16 (hexadecimal)**

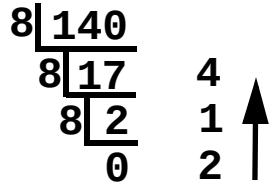
$$\begin{aligned} 140_{10} &= 2 \times 8^2 + 1 \times 8^1 + 4 \times 8^0 = 214_8 \\ &= 8 \times 16^1 + \mathrm{C} \times 16^0 = 8\mathrm{C}_{16} \end{aligned}$$

Digits in base *2* **0 1**
8 **0 1 2 3 4 5 6 7**
10 **0 1 2 3 4 5 6 7 8 9**
16 **0 1 2 3 4 5 6 7 8 9 A=10 B=11 C=12 D=13 E=14 F=15**

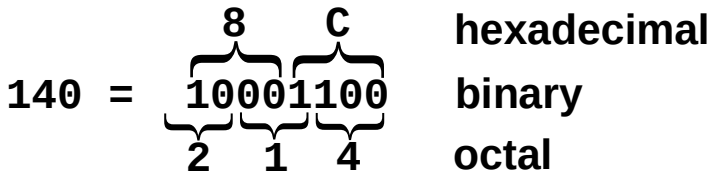
Conversions

- To convert from decimal to binary, divide by 2 repeatedly, read remainders up





- Easier to convert to octal, then to binary, then to hexadecimal



Boolean Functions

- **16 possible Boolean functions of two binary variables; *some* have names, and C operators**

Truth table			
0 0 1 1	x		
0 1 0 1	y		
	Name		***C expression***
0 0 0 0			
0 0 0 1	AND		x & y
0 0 1 0			
0 0 1 1			
0 1 0 0			
0 1 0 1			
0 1 1 0	XOR	‘exclusive or’	x ^ y
0 1 1 1	OR	‘inclusive or’	x | y
1 0 0 0	NOR	‘not or’	~(x | y)
1 0 0 1	EQV	‘not xor’	~(x ^ y)
1 0 1 0	NOT y	one’s complement	~y
1 0 1 1			
1 1 0 0	NOT x	one’s complement	~x
1 1 0 1			
1 1 1 0	NAND	‘not and’	~(x & y)
1 1 1 1			

- **Don’t confuse `&&` `||` `!` for `&` `|` `~`**

Machine Arithmetic

- On a machine with 16-bit words, there are $2^{16} = 65{,}536$ ***unsigned*** integers 0..65,535

0000 0000 0000 0000_2	0
0000 0000 0000 0001	1
0000 0000 0000 0010	2
0000 0000 0000 0011	3
0000 0000 0000 0100	4
…	
1111 1111 1111 1100_{10}	65,532
1111 1111 1111 1101	65,533
1111 1111 1111 1110	65,534
1111 1111 1111 1111	65,535

- There are 65,536 ***two's-complement signed integers*** –32,768..+32,767

1000 0000 0000 0000_2	$-32{,}768_{10}$
1000 0000 0000 0001	–32,767
…	
1111 1111 1111 1110	–2
1111 1111 1111 1111	–1
0000 0000 0000 0000	0
0000 0000 0000 0001	+1
0000 0000 0000 0010	+2
…	
0111 1111 1111 1110	+32,766
0111 1111 1111 1111	+32,767

Two's-Complement Arithmetic

- Adding two's-complement numbers is easy: Ignore signs, add unsigned numbers

```
   +20     010100          -20     101100
 + - 7   + 111001        + + 7   + 000111
   ---     ------          ---     ------
   +13     001101          -13     110011

   +20     010100          -20     101100
 + + 7   + 000111        + - 7   + 111001
   ---     ------          ---     ------
   +27     011011          -27     100101
```

- To ***negate*** a two's complement number: Complement all the bits, then add 1

	Start with	Complement	Increment	
+6	000110	111001	111010	-6
-6	111010	000101	000110	+6
0	000000	111111	000000	0
+1	000001	111110	111111	-1
+31	011111	100000	100001	-31
-31	100001	011110	011111	+31
-32	100000	011111	100000	-32