Lecture 5

TOY MACHINE

(a fairly realistic view of the computer's innards)

The toy programming language resembles today's machine languages.

A Motivating Example: `goto` statement in C

Syntax: `goto` *identifier*;
where *identifier* is a label within the same function.

Example:

label

```
#include <stdio.h>
void main()
{
 int i=10;

  squarer: i = i*i;
  if (i<= 1000)
      goto squarer;
  printf("%d", i);
}
```

Goto's can be used to write loops

Example: Program to sum the integers from 1 to n

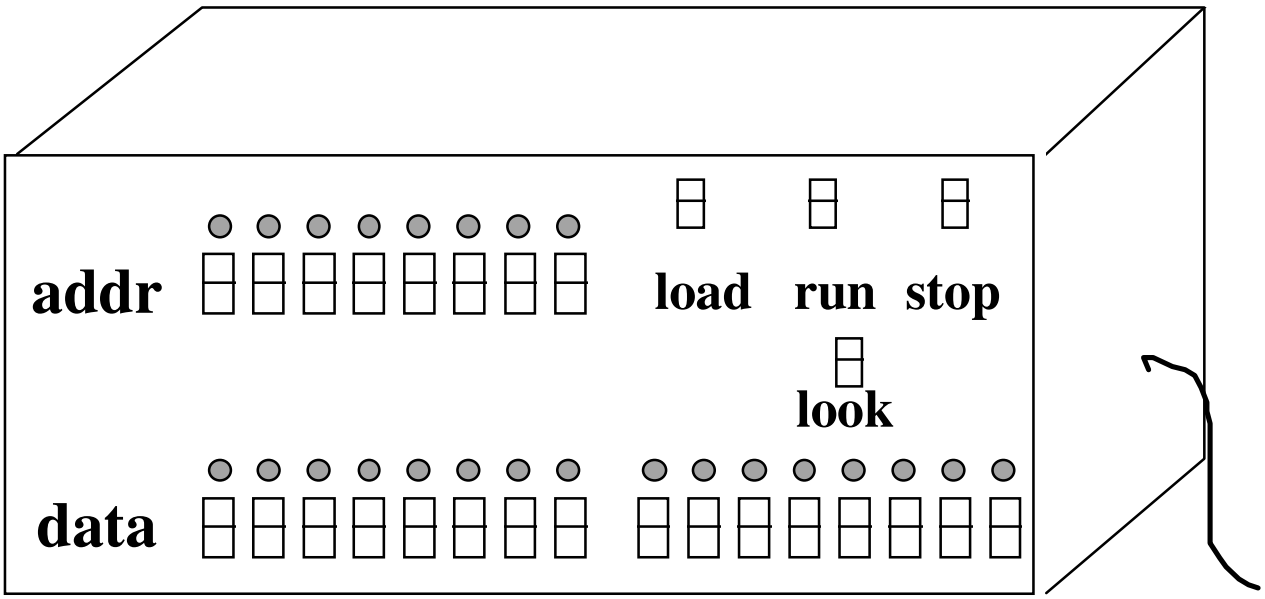
```
#include <stdio.h>
void main()
{
 int i, sum, n;
 printf("Input a +ve integer");
 scanf("%d", &n);
 i=1; sum=0;

 A: if (i >n)
      goto B;
 sum = sum +i;
 i = i+1;
 goto A;

 B: printf("The sum is %d", sum);
}
```

TOY Machine Description

Outside: Box with switches and lights



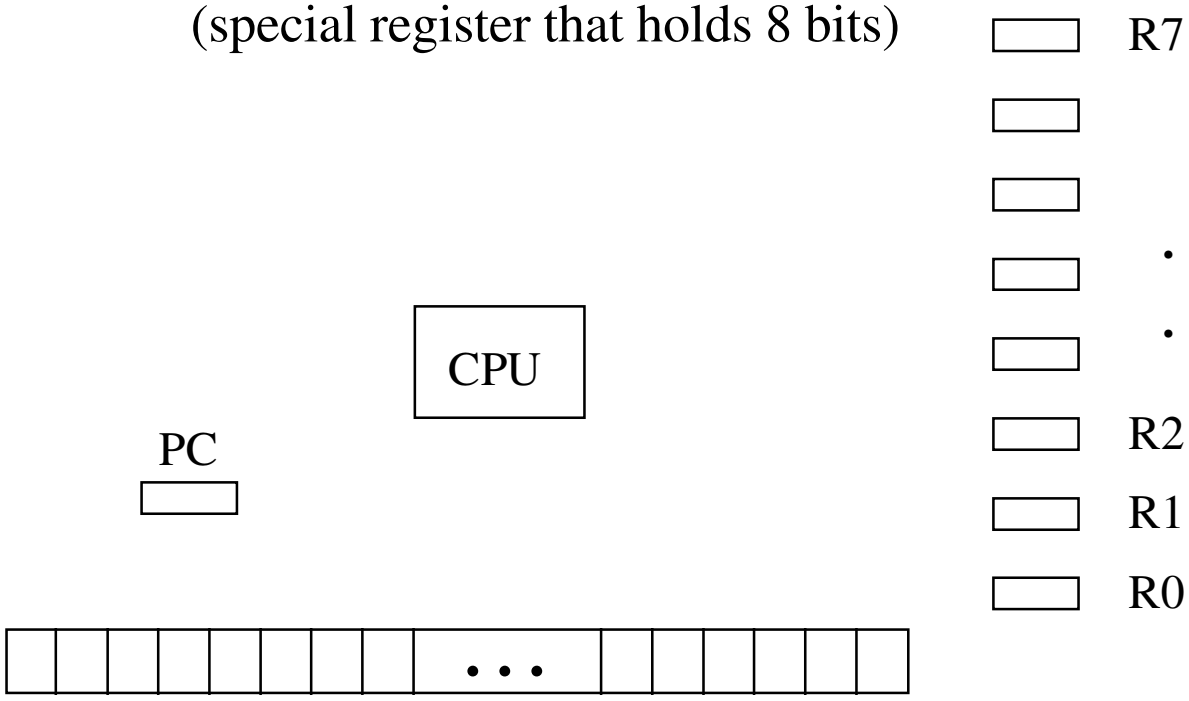
Inside the box:

- CPU
- 256 "words" of memory
- 8 "registers"

(each register/word holds 16 bits)

- Program Counter (PC);

(special register that holds 8 bits)



Operating the Toy Machine

- load the program and data using switches
- load address of first instruction into PC
- press GO

.

.

(wait until the "halt" light comes on)

- "look" at contents of memory to determine program output.

Program = Sequence of instructions

Data/Instruction Representation

Everything is represented by 16 bits:

- machine instructions
- data (the only data type is a 16-bit signed integer)

Hexadecimal ("hex") alphabet: Each letter stands for a sequence of 4 bits

"0" = 0000, "1"= 0001, .. "9"= 1001,
"A"=1010, "B"= 1011, ... "F"= 1111.

A "Dump" of TOY machine memory in hex

PC: 0010

R0: R1: R2: R3: R4: R5: R6: R7:
0000 0788 B700 0010 0401 0002 0003 00A0

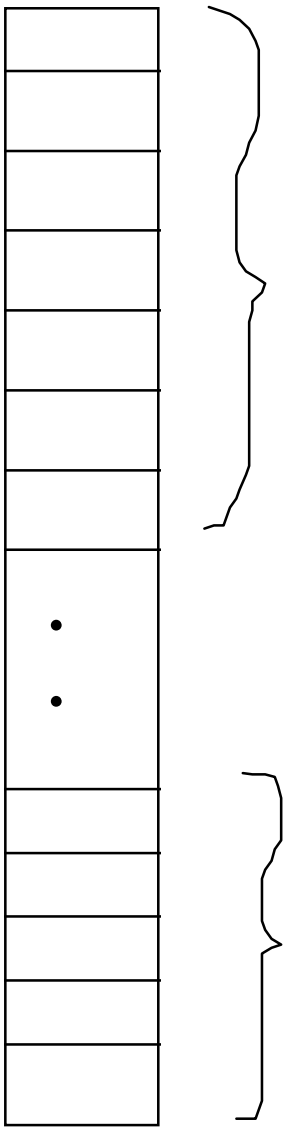
00: 0000 0000 0000 004D 0000 001A 0000 0000
08: 0000 0000 0000 0000 0000 0000 B009 0000
10: 9222 9120 1121 A120 1121 A121 7211 0000
18: 0000 0000 0000 0000 0000 0000 0000 0000

•

•

E8: 0000 0000 0000 0000 000C 0000 0000 0000
F0: 0000 0000 0000 0000 000B 0000 0000 0000
F8: 0000 0000 0000 0000 00A0 0000 0000 0000

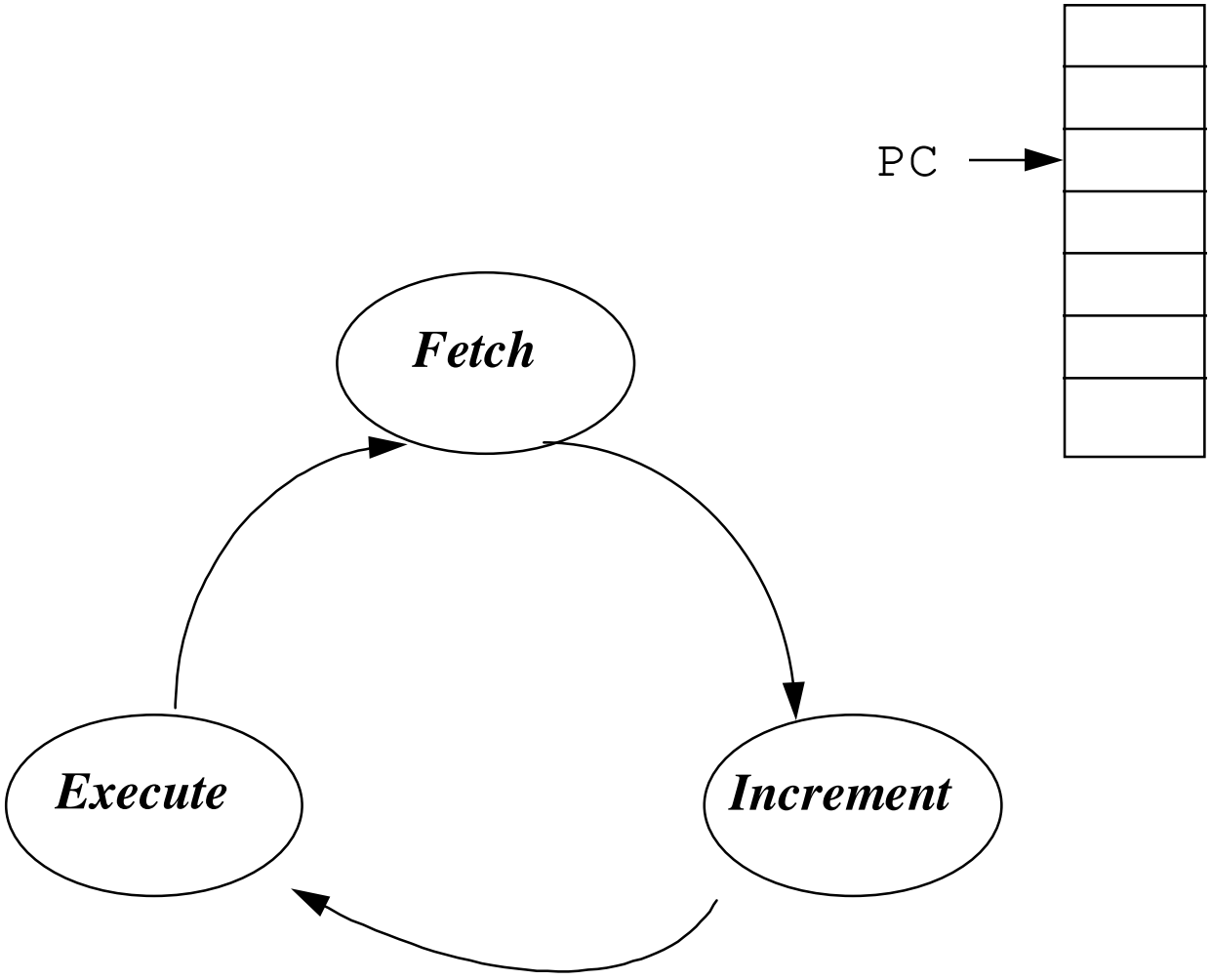
General Form of a TOY Program



instructions
(correspond to `main( )` and
user-defined functions of C)

data
(corresponds to variables)

How the program executes:
Fetch-Increment-Execute Cycle



The cycle stops when the HALT instruction is encountered

Instruction Format

Most instructions have a register address as argument
(Note: Address of a register = 3 bits)

Format 1

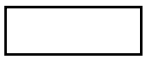
opcode dest reg reg

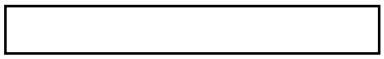
Format 2

opcode dest reg constt

Format 3

opcode dest constt

 = 4bits

 = 8 bits

Opcodes

0: halt
1: add
2: subtract
3: multiply
4: system call
5: jump
6: jump if less than
7: jump indirect
8: jump and link
9: load
A: store
B: load immediate
C: xor
D: and
E: shift right
F: shift left

system call instruction (opcode 4) is currently used only for `printf()`

Format 1 Instructions

(Arithmetic Operations)

OPCODE REG 1 REG 2 REG 3

e.g., `1 2 3 7`
stands for

```
ADD R2 R3 R7
```

which means:

```
R2 = R3 + R7;
```

The following are Format 1:

- ADD (Opcode =1)
- SUBTRACT (Opcode =2)
- MULTIPLY (Opcode =3)
- XOR, AND (Opcodes = C, D)
- SHIFT-RIGHT, SHIFT LEFT (Opcodes=E,F)

e.g., `3 x y z`
means

```
Rx = Ry * Ry
```

Format 2 Instructions

(MEMORY OPERATIONS)

OPCODE REG1 REG2 CONSTANT

e.g., `9 x y z`
stands for

```
LOAD    Rx    Ry    z
```

which means

```
Rx = M[Ry + z]
```

```
A x y z
```

stands for

```
STORE   Rx   Ry   z
```

which means

```
M[Ry + z]  = Rx
```

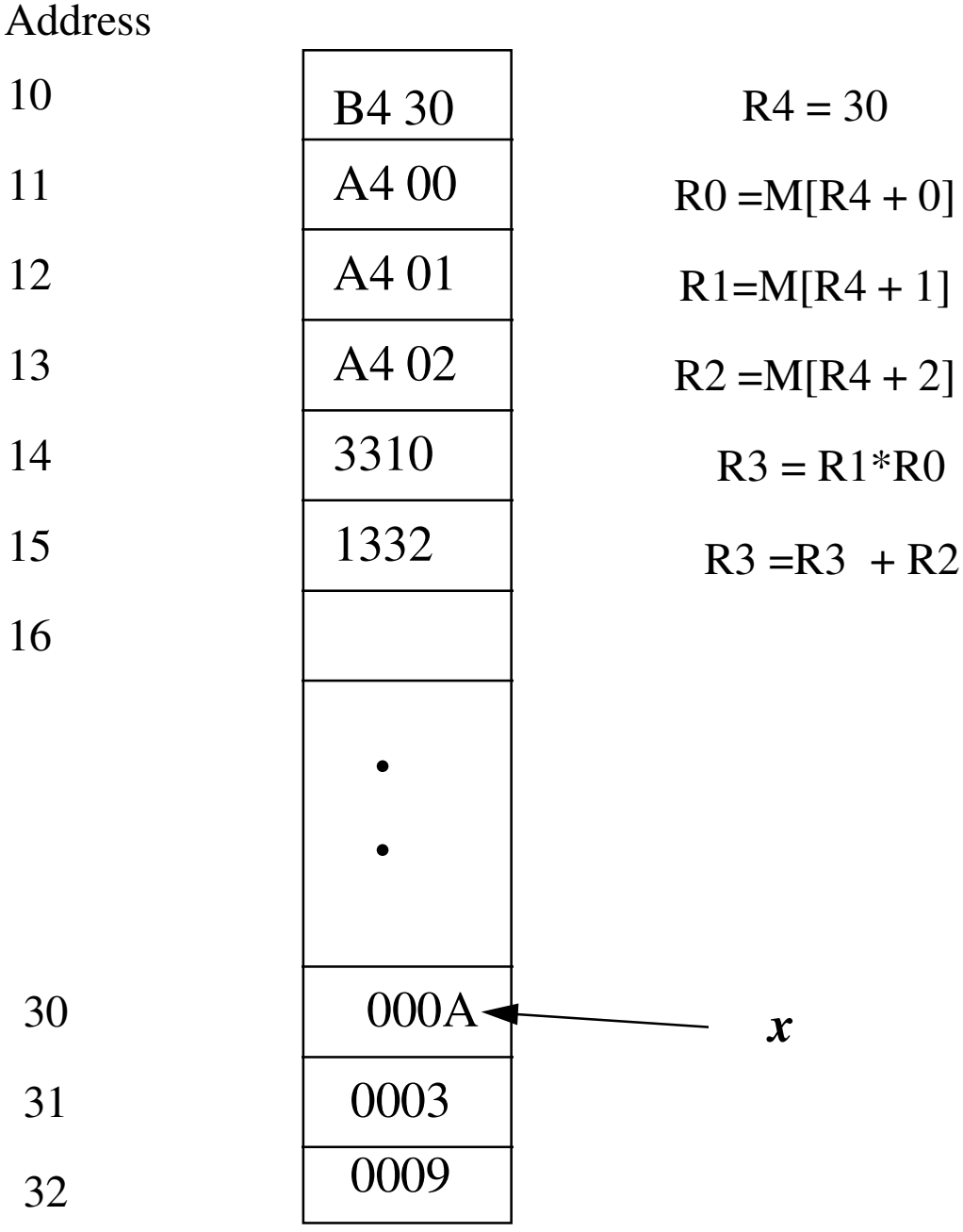
Note: ADDRESS is the contents of the register
+ 4-bit constant (the "displacement")

Format 3 Instructions

OPCODE REG CONSTANT

e.g., `B x yz`
stands for
`LOAD IMMEDIATE Rx yz`
which means
`Rx = yz`

Example: copy data from R2 into R3

```
B000          R0 = 00
1320          R3 = R2 + R0
```

Program fragment to evaluate 3x + 9 at x = 10