Objectives

1. structure solutions to problems for execution by computer

2. use a microcomputer efficiently and effectively

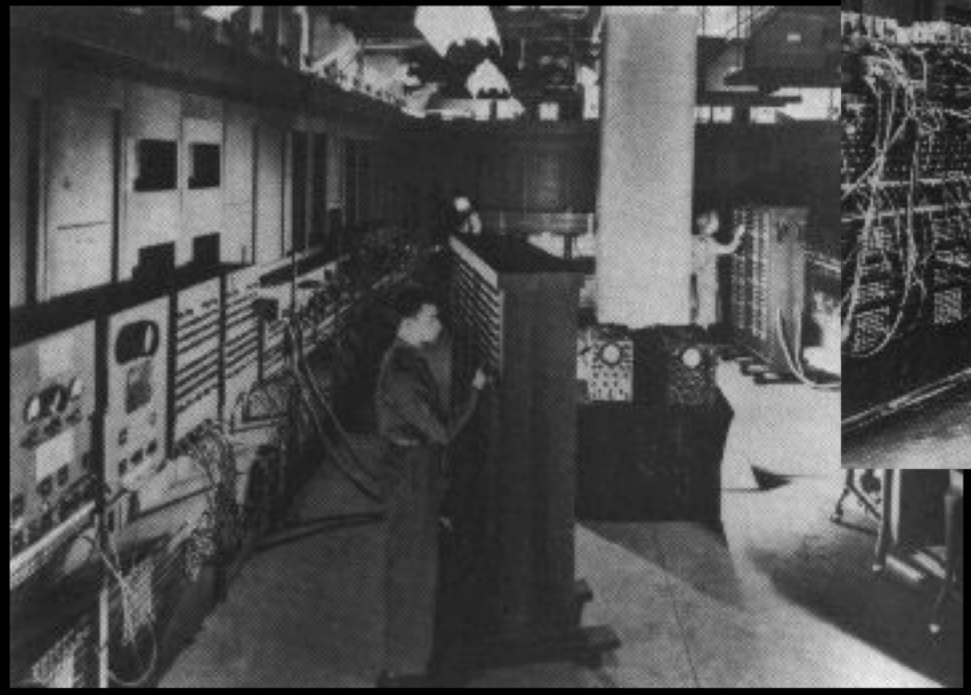
3. develop and express solutions in good programming style

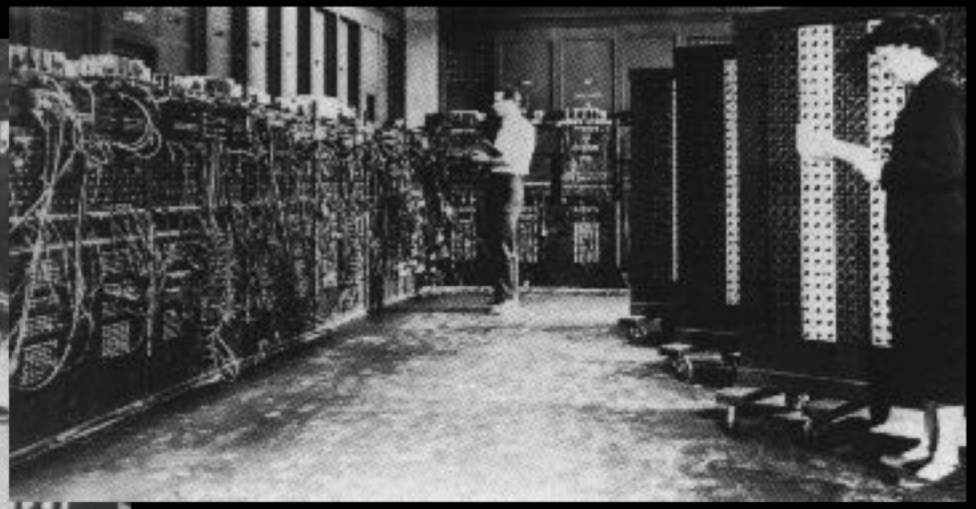
4. express solutions in well-structured programs written in C++

You will need to

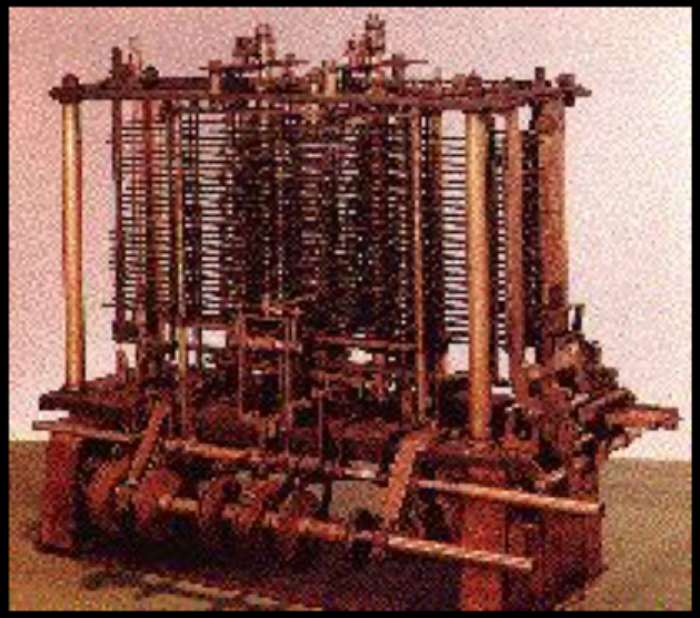
- learn the jargon
- extend your problem-solving skills
- extend your writing skills
- learn some C++
- learn good programming practice/style

Why do we need a computer language?





But Babbage had used codes years before



Unfortunately switches (on, off) or binary codes (0,1) are difficult to manipulate by humans who think in visual terms.

Higher level languages

- closer to natural language - English
- converted to machine code using compiler
- can't use English - too imprecise
  too many exceptions
  too context dependent

ou

cough

bough

enough

although

to
too
two

ghoti = fish

cough

women

tuition

Go to the table where the telephone is ringing
and pick it up

Go to the mountain with the hut on top
and sleep in it

Go to the mountain with the hut on top
and climb it

syntax

- grammar
- how symbols are put together
- what sequences of letters are meaningful

semantics

- what the sequence of symbols mean
- what does the computer do

A limited language is desirable

Why C++?

- structured
- currently popular
- employment
- object-oriented
- part of the Algol family

A bit of history

AT&T Bell Labs

Ken Thompson

UNIX™

PDP-7 –> PDP-11

thousands of lines of code

BCPL –> B

Dennis Ritchie

C

Bjarne Stroustrup

C with objects

This course does not intend to teach all of C++.

In fact, very little of the object-oriented component will be mentioned.

A subset will be discussed, not unlike C.

Try to eliminate some of the (student) difficulties with C.

Fundamental Concepts of Computing

- electronic computer is relatively new

- Babbage developed 5 essential elements:
  - INPUT
  - OUTPUT
  - STORAGE
  - PROCESS

  and
  - CONTROL

Many of programming concepts stem from Babbage's work last century.

But Babbage did not do most of the programming.

Ada Lovelace

- reusable code
- subroutines

Let's apply Babbage's logical elements.

I will say three numbers.

Add the first two together, then subtract the third.

Tell me the answer.

So

5 6 7 –> 4

3 9 4 –> 8

What was the INPUT?

- **the instructions**
- **the data**

How did you receive the input?

via the senses

Where did you STORE the instructions?

in your brain

How about the data?

same again

What did the CONTROLLING?

you did (or your brain)

What did the PROCESSING?

the adding and subtracting was done by your brain

Finally, what about OUTPUT?

you wrote it down

or

you spoke

Without output, the program is useless.

All of these elements are present in all computers:

INPUT

keyboard, mouse, touch panel, digitiser

OUTPUT

screen, printer, plotter

PROCESS

arithmetic logic unit

in central processing unit

CONTROL

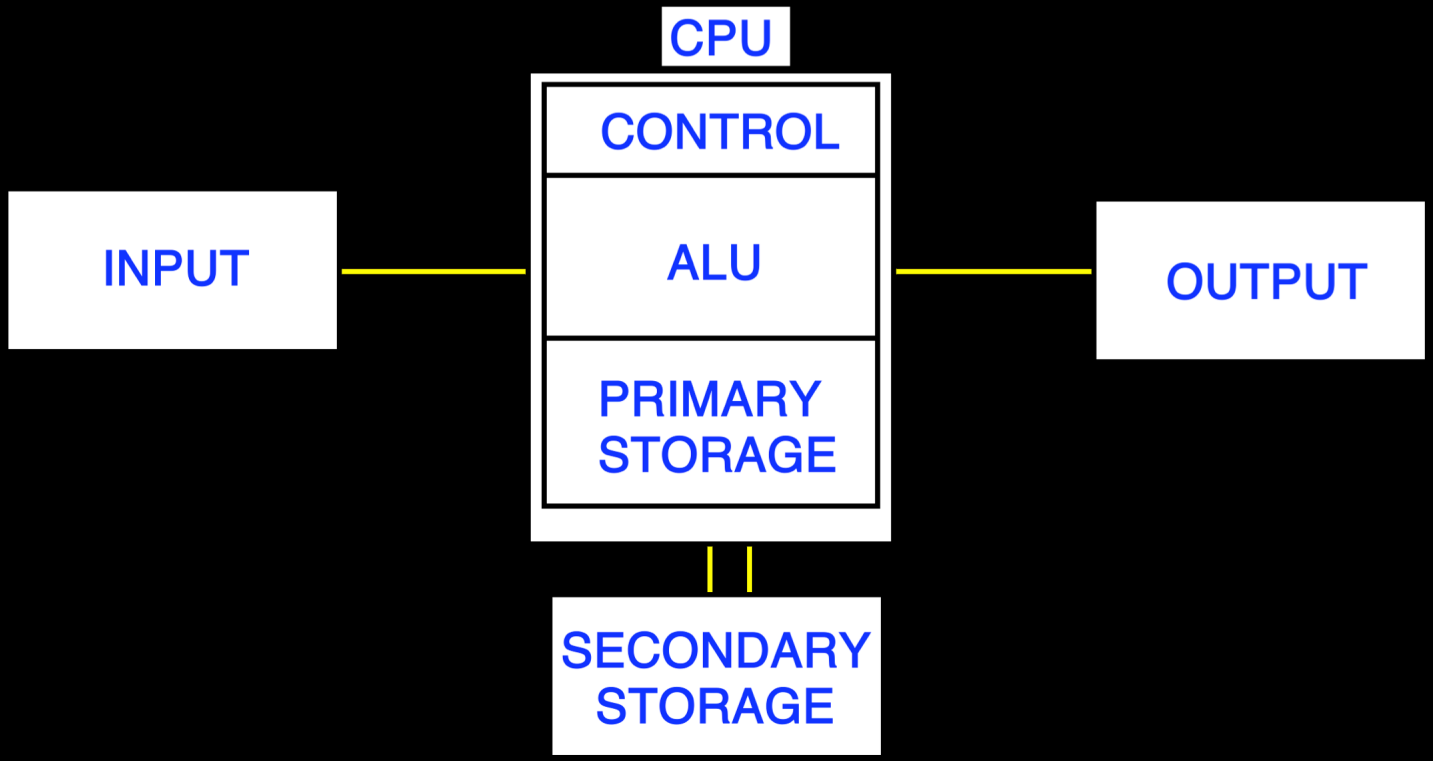
in the CPU

STORAGE

temporary (or primary) storage in the computer's memory

permanent (or secondary) storage on media such as magnetic disks, tapes, optical disks, . . .

Primary storage can be considered as a sequence of locations, each with a unique address.


CPU
CONTROL
ALU
PRIMARY STORAGE
INPUT
OUTPUT
SECONDARY STORAGE

Writing to memory overwrites any existing content of the location

– destructive write

Reading something from memory takes a copy, leaving the original contents unchanged

– (non-destructive) read

So let's look at how a high-level language refers to memory.

Here is a simple high level construct

```
TOTAL = PRINCIPAL + INTEREST
```

This is actually C++.

It contains

- 3 labels

  `TOTAL, PRINCIPAL, INTEREST`

- 2 operators `=` and `+`

```
TOTAL = PRINCIPAL + INTEREST
```

A compiler will translate this into machine code:

Label a memory location `TOTAL`, another `PRINCIPAL` and a third `INTEREST`.

Take a copy of the contents of the location named `PRINCIPAL`, and ADD its contents to the contents of the location named `INTEREST`.

ASSIGN (store) the result in the location called `TOTAL`.

What if I now say 2, 8, 3 ?

Someone will say 7.
You have a memory of earlier instructions.
I've given you new data.

You've been programmed.

You can separate instructions from data
– they look different to you.

At the machine level, program and data look just the same
– merely strings of 0's and 1's.

Just binary digits – bits.

A collection of bits (usually 8) is called a byte.

Computer memory is usually collected into words – can be handled by the computer as a single item – 32 bits

Representation of Numbers

Mathematicians differentiate between

integers

5 17 0 -456 12345

and

real numbers

5.0 23.674 -0.0013

Computers also must vary the way these numbers are stored physically.

Here is the 16-bit representation of the integer 5

0000000000000101

Each binary digit, read from right to left, represents a power of 2 (just like decimal digits represent powers of 10)

So

$5 = 1x\ 2^0 + 0 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + \ldots$

OK. So what number is represented by
010010011010 ?

Start from the right-hand end:

0×2^0 0
1×2^1 2
1×2^3 8
1×2^4 16
1×2^7 128
1×2^{10} 1024

A total of 1178

Can you go from decimal to binary?

What is 194 as a binary number?

Find the highest power of two smaller than (or equal to) the number.

1,2,4,8,16,32,64,128,256. Too far.
So 128 is largest.
So the first digit is 1. (Always.)

Subtract 128 from the number -> 66.
Now go down the powers.
64 is smaller - next digit is 1 -> 11 so far.

Subtract 64 from 66 to leave 2.
32 is too big, so next digit is 0 -> 110.
16 is too big, so next digit is 0 -> 1100.
8 is too big -> 11000.
4 is too big -> 110000.
2 is just right -> 1100001.

Remainder is 0, but we have to keep going with zeros.

1 is too big -> 11000010. DONE.
(Check by converting answer to decimal.)

Let's limit discussion to 4-bit representations (just to save space).

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

What about negative integers?

Negative Integers

Replace all 0's by 1's and 1's by 0's.

-0	=	1111
-1	=	1110
-2	=	1101
-3	=	1100
-4	=	1011
-5	=	1010
-6	=	1001
-7	=	1000

This is ones complement arithmetic.

Negative Integers

ones complement arithmetic has

two zeros +0 and -0

and can represent all values from

-7 to +7 with 4 bits

i.e. the range is from

$-(2^{n-1}-1)$ to $2^{n-1}-1$

where n = 4.

Why n-1?

Because the top bit only indicates sign.

Negative Integers

Look at any meter, say a trip meter in a car.
When the device reaches its limit, it will read

0000

One reading before this value would be

9999

(one less than 10000)

Using the same analogy we can define negative binary integers using twos complement arithmetic, by subtracting the positive value from 0.

Negative Integers

twos complement

0	=	0000
-1	=	1111
-2	=	1110
-3	=	1101
-4	=	1100
-5	=	1011
-6	=	1010
-7	=	1001
-8	=	1000

-1	=	1110
-2	=	1101
-3	=	1100
-4	=	1011
-5	=	1010
-6	=	1001
-7	=	1000

add one to ones complement

Negative Integers

We get an extra negative number in twos complement (only one zero).

How about bigger numbers?

0001010010101010
= 2+8+32+128+1024+4096 = +5290

ones complement
-5290 = 1110101101010101

twos complement (add 1 to above)
-5290 = 1110101101010110

Negative Integers

sign and magnitude

```
 0    =    0000
-1    =    1001
-2    =    1010
-3    =    1011
-4    =    1100
```

and

+5290 = 0001010010101010

means

-5290 = 1001010010101010

Using binary digits occupies a lot of space - that's why we like decimal.

But the relationship from binary to decimal is not a simple one.

Computers often use two other bases, by collecting bits into groups of 3 or 4.

Octal - base 8 - 3 bits

Hexadecimal - base 16 - 4 bits

Thus
5290 in decimal
= 0001010010101010 in binary
or 0 001 010 010 101 010
(breaking into 3s from right)

= 012252 in octal

or 0001 0100 1010 1010
(breaking into 4s from the right)

= 14AA in hex

Hex

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
then
10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
1A, 1B, 1C, 1D, 1E, 1F, 20, 21,...

The letters A-F can appear in lower case.

So
ff = 11111111 (binary) = 255 (decimal)

Representation of real numbers

much more complicated

a decimal real

123.456

$= 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$

$+ 4 \times 10^{-1} + 5 \times 10^{-2} + 6 \times 10^{-3}$

a binary real

101.0101

$= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

$+ 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4}$

$= 5.3125$

Consider that computers stored reals in decimal form and not binary.

Any decimal real can be written with a 0 before the decimal point, a non-zero digit after and a power of 10 multiplier.

123.456 becomes 0.123456 x 103
0.00789 is the same as 0.789 x 10-2

decimal fraction is the **mantissa**
power is called the **exponent**

Since we know where the decimal point is (and the base of the power is 10), we can just store the fraction and the exponent.

Let's store the exponent as 2 decimal digits – 00 to 99

What about negative exponents?

Use an offset.

Pick a middle value, say 50.

Use it to represent a power of zero.

Then 53 would be +3.

48 would be -2.

So we now have a range of multiplier from 10-50 to 1049.

(Where is the numerical value 0?)

123.456

0.123456×10^{3}

stored as +53123456

-0.00789

$-.789 \times 10^{-2}$

stored as -48789000

Back to binary reals.

exponent fraction sign bit

exponent base is 2

fraction is binary

$5.0 = 101.000 \times 2^0$

$= 10.1000 \times 2^1$

$= 1.01000 \times 2^2$

5.0 = 1.01000 x 2^2

Ensure 1 before the point.
Then don't store it.

So

sign	exponent	fraction
0	10000001	01000000000000000000000
1 bit	8 bits	23 bits

Again, the exponent is offset, this time by 127, mid-way on range from 0 to 255.

Thus
255 represents an exponent of +128
0 stands for -127

127 is the power 0

That's why +2 is 10000001

Range of multipliers is 2^{-127} to 2^{128}
or about 10^{-38} to 10^{38}.

represents

$(-1)^s \times 2e^{-127} \times 1.f$

We use scientific notation, writing
1.234×10^2 as 1.234e2
0.0001234 as 1.234e-4 or .1234e-3 or …

A Warning

Numbers we think are simple in decimal notation are like fractions such as 1/3 as decimals.

For example

$$
\begin{aligned}
0.3 &= 0.25 + 0.05 \\
&= 0.25 + 0.03125 + 0.01875 \\
&= 0.25 + 0.03125 + 0.015625 + 0.003125 \\
&= ? \; ? \; ? \\
&= 0.010011???
\end{aligned}
$$

How big can numbers be?

- depends on how many bits are used
- varies from computer to computer
- even from compiler to compiler
- there exist standards

16-bit integers

from 0000000000000000 (0)
to 1111111111111111 (65535)

BUT if we have negative values, then max value is 32767.

Smallest negative number depends on whether ones or two complement.
-32767 or -32768

(ones complement has two zeros)

32-bit integers

maximum is 2147483647

minimum is –2147483648 (2s comp)

32-bit reals

- 8-bit exponent
  could be 2^{-127} to 2^{128}
  standard reserves max and min
  so range is 2^{-126} to 2^{127} (10^{-38} to 10^{38})

- 24-bit mantissa incl. sign bit
  assumes 1 before the point
  all zeros would represent
  1.000...000 x $2^{exponent}$
  all 1s would be 1.111...111 x $2^{exponent}$

24 bit fractions equivalent to 5 to 7 decimal digits.

What about zero?

- cannot be written as 1.something times
- use smallest exponent -127 (all zeros) and a zero mantissa (all zeros)
- same as long integer zero

The maximum exponent (all ones) is used in the standard (IEEE754) to represent special values such as

+∞ called +INF

-∞ called -INF

and NAN (not a number)
used for impossible values (0/0)

5 to 7 digits is usually enough but

64-bit reals

use 11 bits of exponent
53 bits of mantissa

numbers in range 10-308 to 10308

about 16 digits of decimal precision

Also 48-bit, 80-bit, 96-bit and even 128-bit

What about letters and symbols?

useless if we have to always communicate with computers just with numbers

want letters, both upper and lower case
- punctuation marks
- others

ASCII code

American Standard Code for Information Interchange

A byte can store 256 different bit patterns

ASCII describes 128 of these (use 7 bits)

alphabetic

A	65 dec	101 octal
B	66	102
Z	90	132
a	97	141
b	98	142
z	122	172

Thus

01000110	in binary
106	octal

is

F	in ASCII

space is 40 octal (32 decimal)

Numbers 0 to 9 as characters are
60 to 71 octal
48 to 57 decimal

Some symbols

$	44 octal
+	53
-	55
*	52
=	75
/	57
?	77
.	56

printables are octal 40 to 176
below 40 are non-printables

What about the other 128 characters? (those with upper bit 1 - negative?)

foreign characters and others

§ ¡ ™ £ ¢ ∞ ¶ • ª º π ø

œ ∑ é ® † ¥ ü î å ß ∂ ƒ

© ˙ ∆ ˚ ¬ … æ Ω ≈ ç √ ∫

~ ñ µ ≤ ≥ ÷ ¿ “ ‘ « ” ’ »

not ASCII

So
any 32 bits of computer memory can be interpreted as

- a binary bit pattern
- an integer
- a real
- four characters

eg. 01000110011100100110010101100100
10634462544 octal, 46726564 hex
1181902180 decimal integer
1.551335e4 real, "Fred" characters

A Simple Computer Model

The central processing unit (CPU) contains

- Arithmetic Logic Unit (ALU)
- four fast access memory locations (registers)
  - Accumulator
  - Program Counter
  - Instruction Register
  - Data Register

Fetch/Execute Cycle

Fetch

- next instruction is fetched from memory
  - loaded into Instruction Register
  - and decoded
- OpCode stays in IR, address of operand is placed in Data Register

Execute

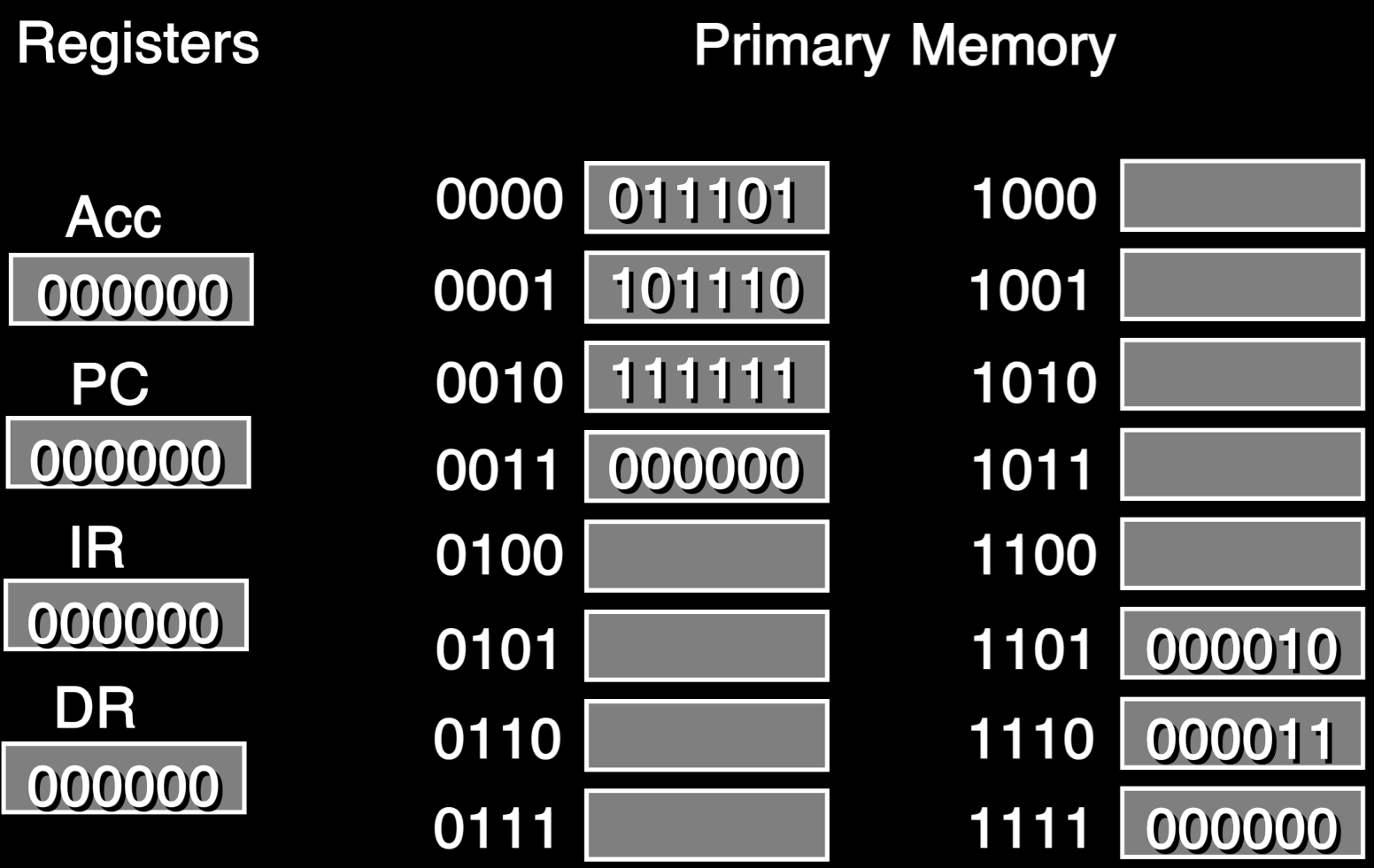
- instruction sent to ALU and performed
- PC is then incremented to location of next instruction

Our machine only has 6-bit words

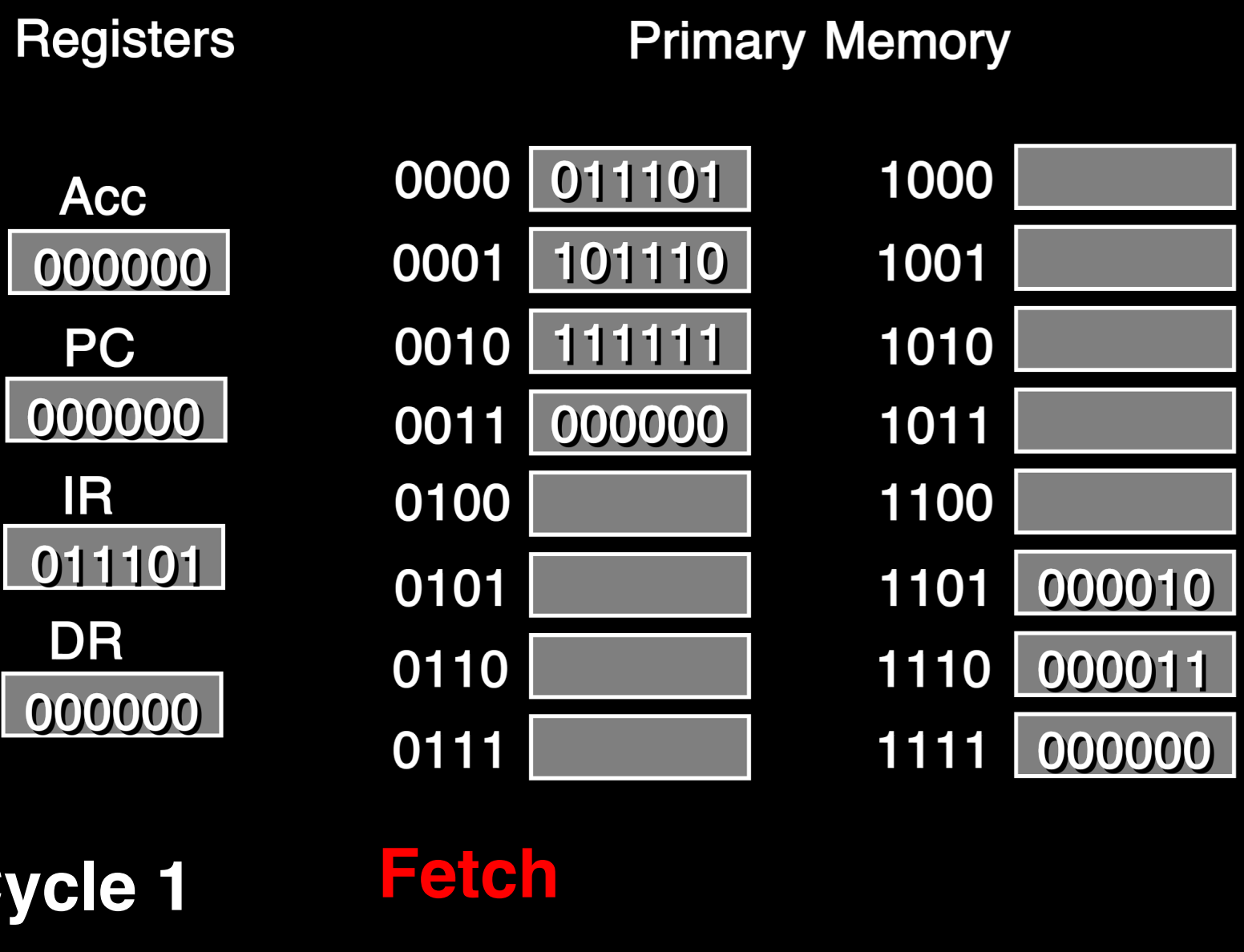
Instructions have 2-bit OpCodes
and 4-bit addresses of operands.

00 xxxx	STOP	address is ignored
01 aaaa	LOAD	put into accumulator
10 bbbb	ADD	add to accumulator
11 cccc	STORE	place acc into memory

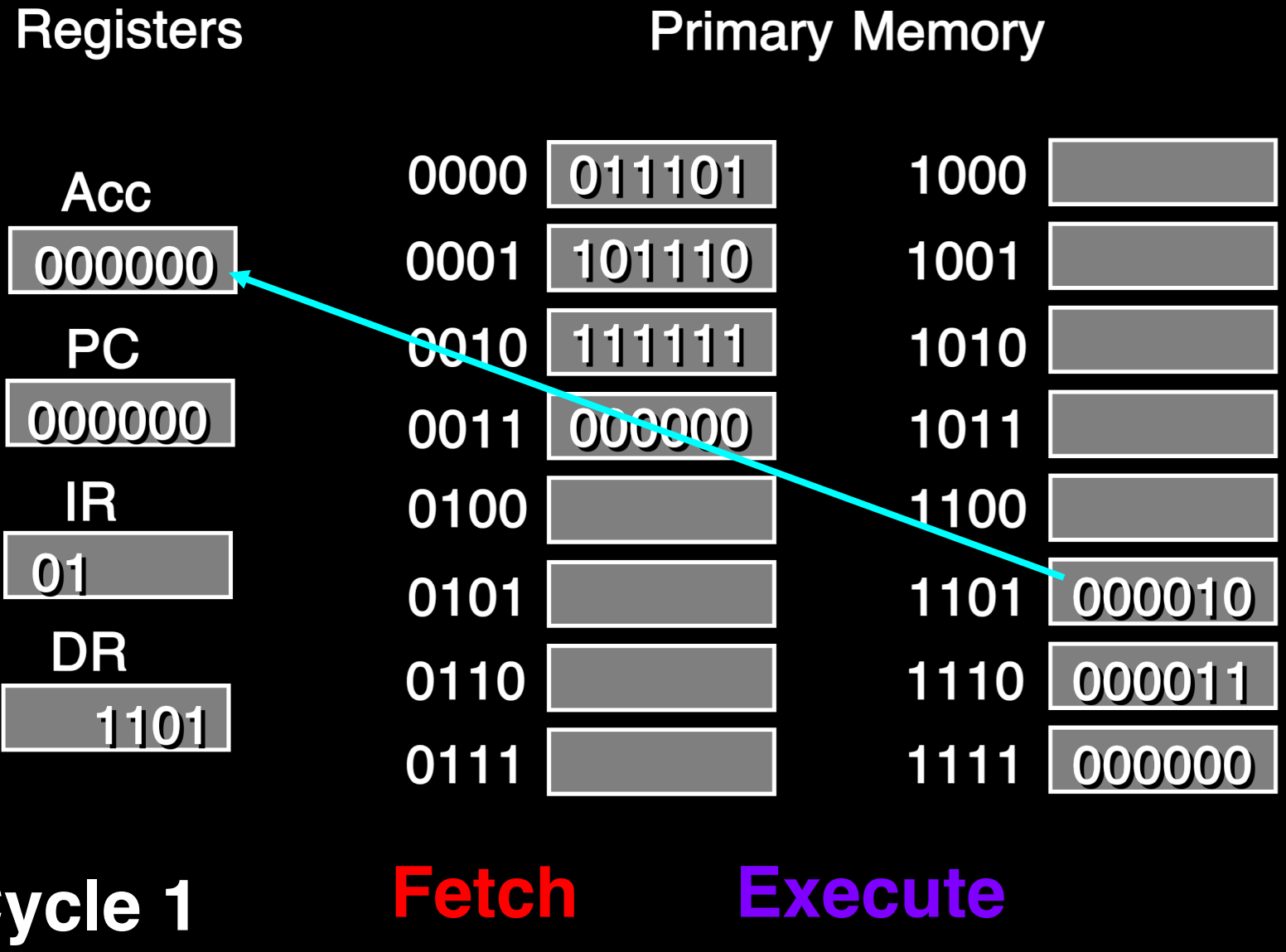
4-bit addresses –> 16 memory locations

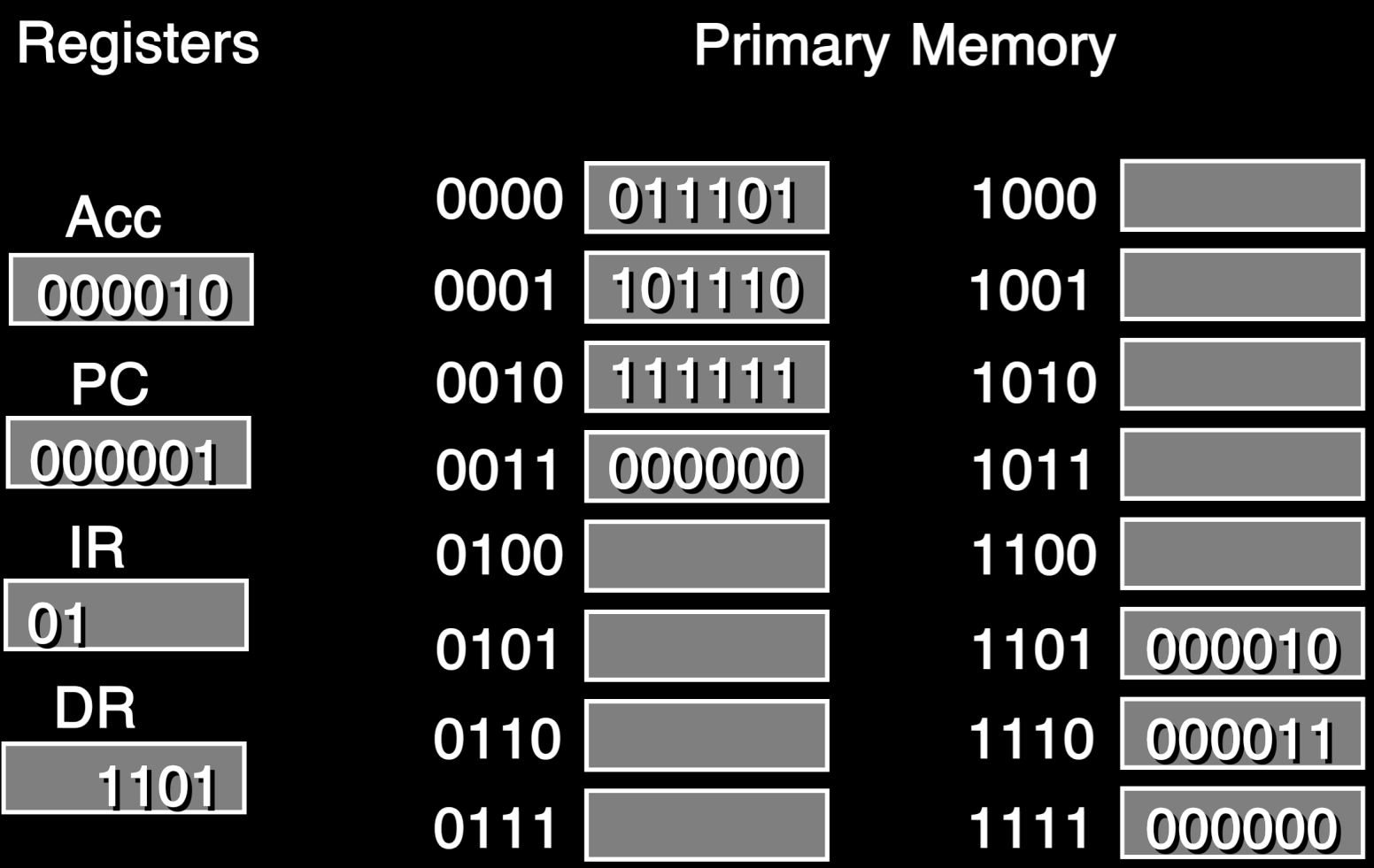

Registers
Primary Memory
Acc
000000
PC
000000
IR
000000
DR
000000
0000 011101
0001 101110
0010 111111
0011 000000
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101 000010
1110 000011
1111 000000

Cycle 1

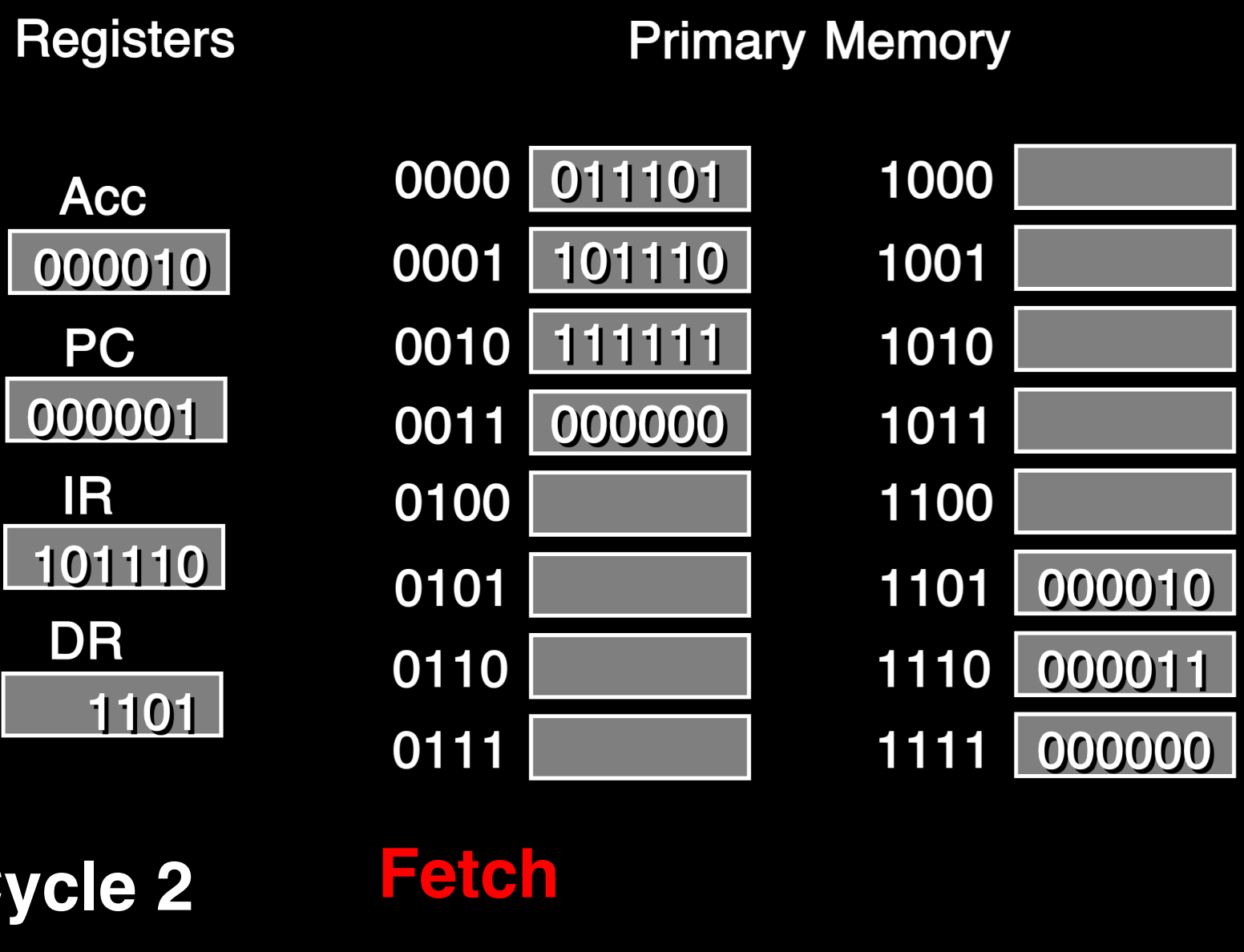


Cycle 1 **Fetch**

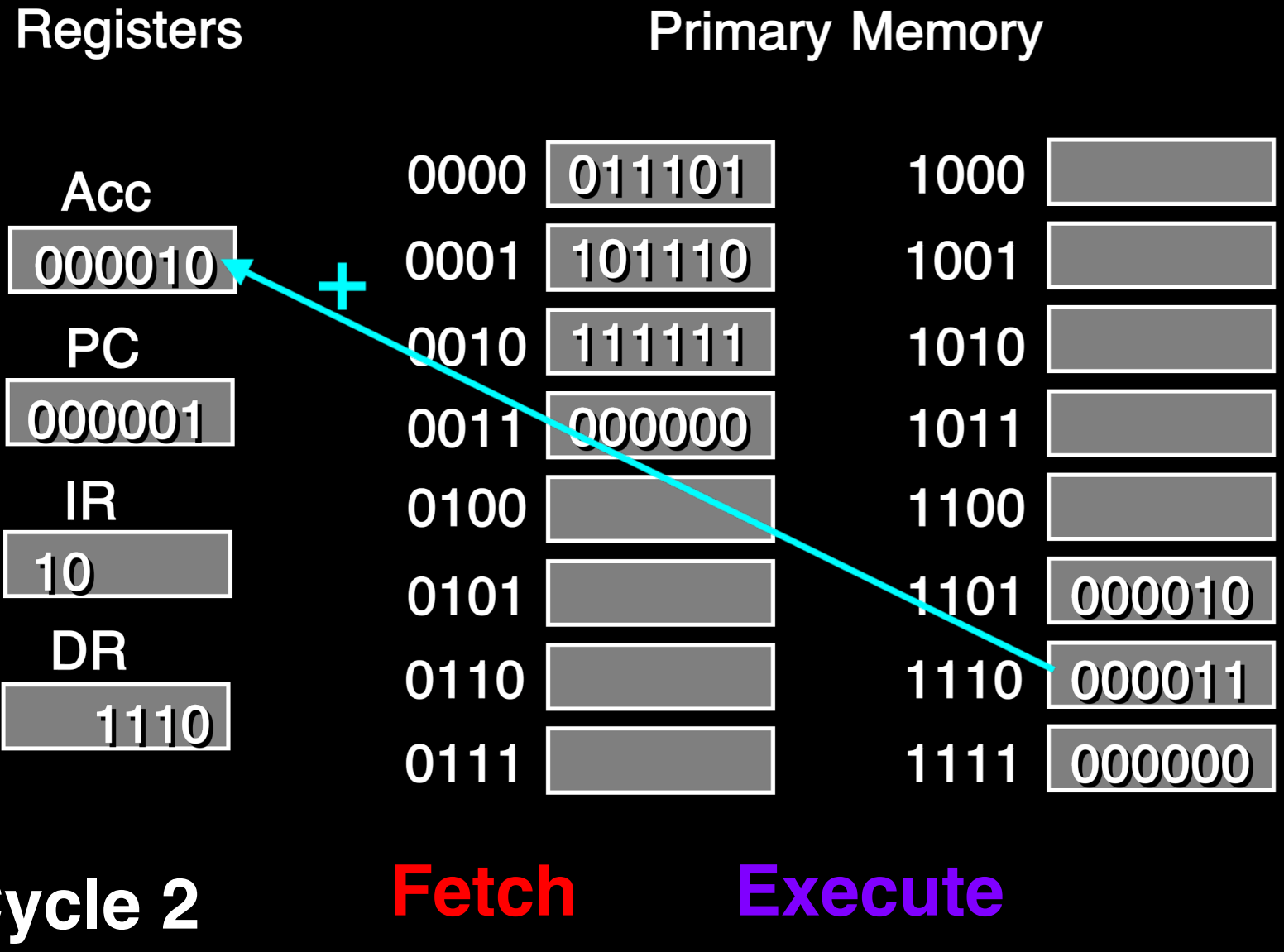

Registers
Primary Memory
Acc
000000
PC
000000
IR
01
DR
1101
0000 011101
0001 101110
0010 111111
0011 000000
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101 000010
1110 000011
1111 000000
Cycle 1
Fetch
Execute


Registers
Primary Memory
Acc
000010
PC
000001
IR
01
DR
1101
0000 011101
0001 101110
0010 111111
0011 000000
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101 000010
1110 000011
1111 000000

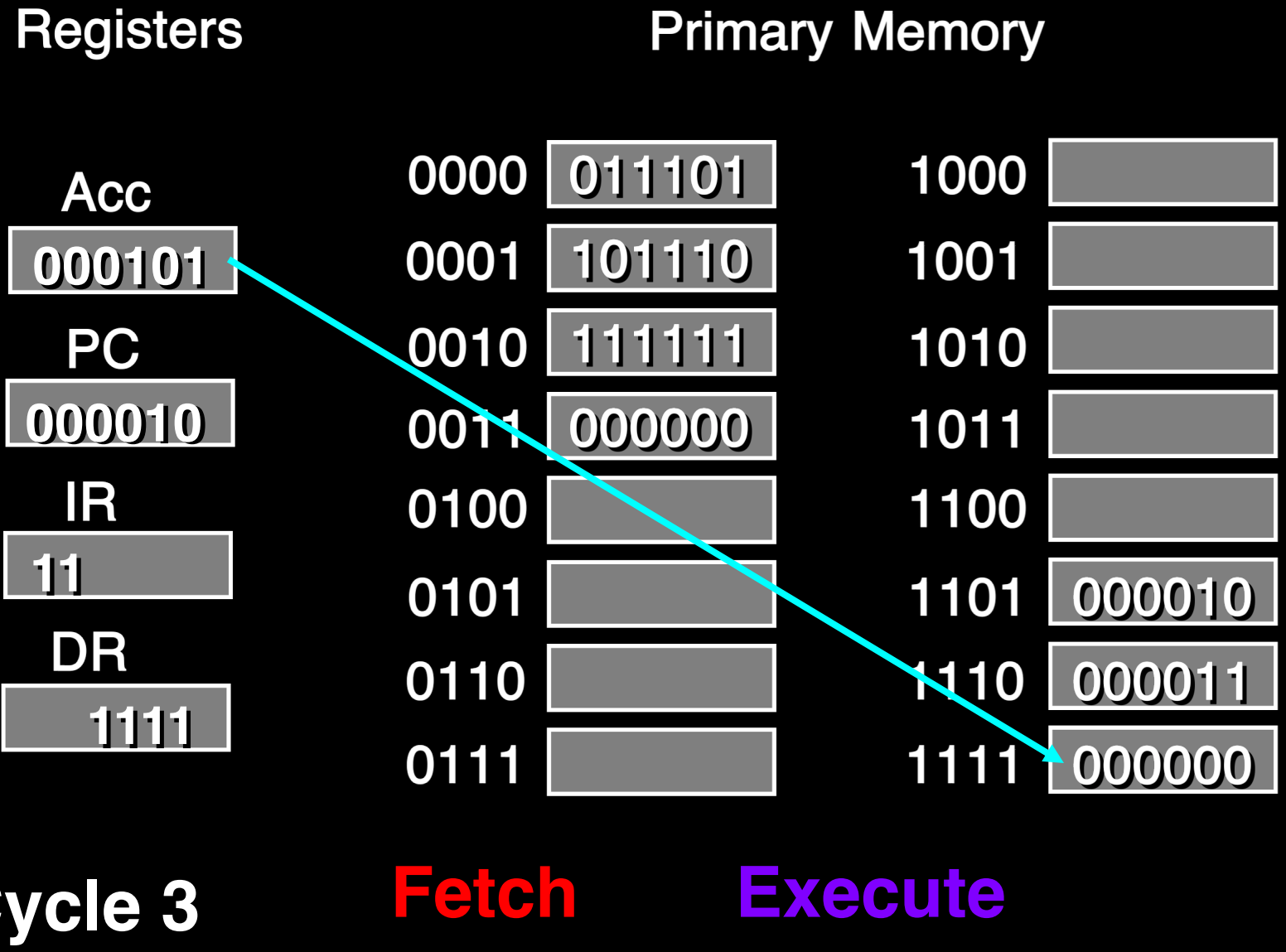
Cycle 2



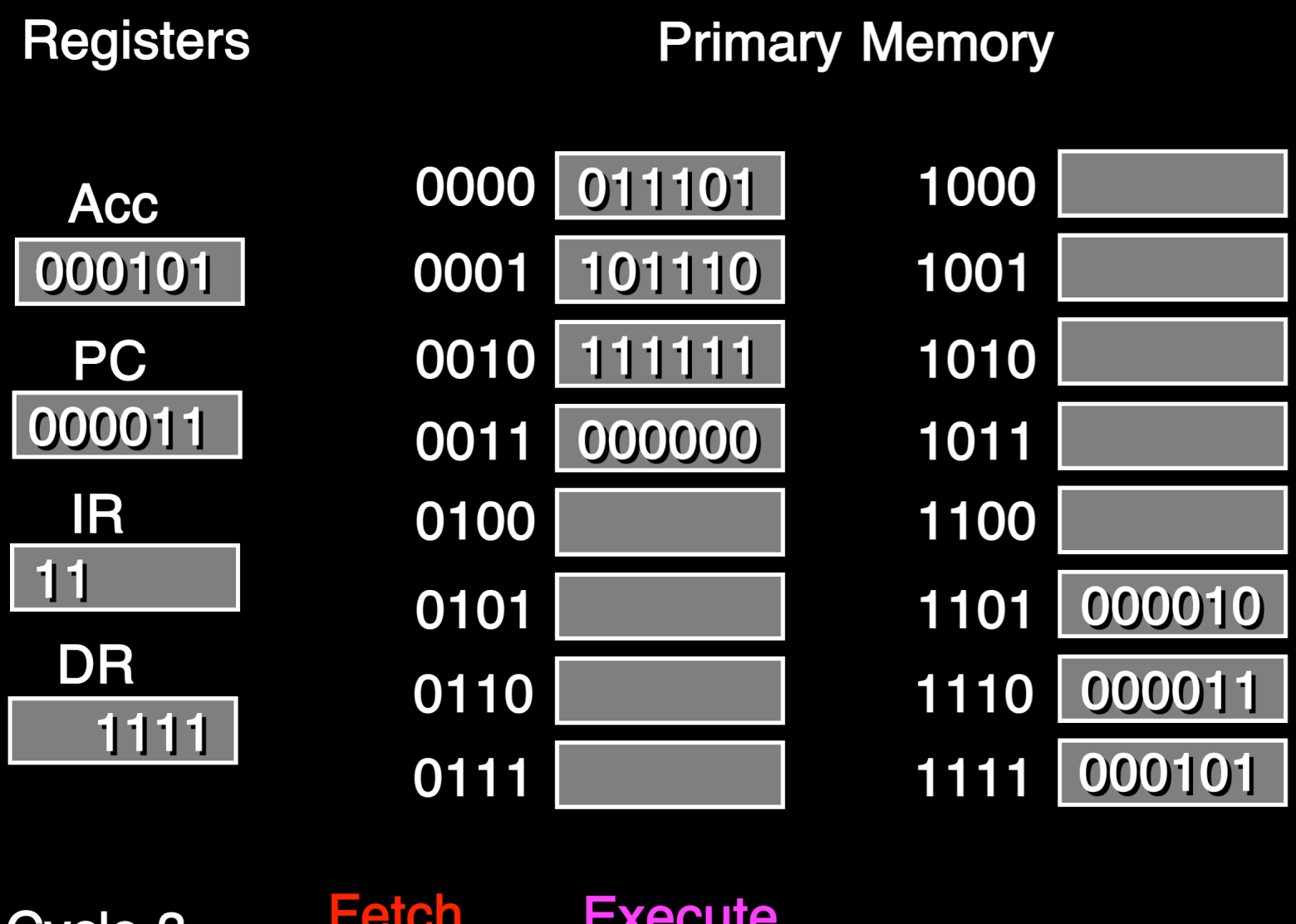
Cycle 2 **Fetch**



Cycle 2 **Fetch** **Execute**



Cycle 3 **Fetch** **Execute**


Registers
Primary Memory
Acc
000101
PC
000011
IR
11
DR
1111
0000 011101
0001 101110
0010 111111
0011 000000
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101 000010
1110 000011
1111 000101
Cycle 3
Fetch
Execute

High Level Languages

syntax

allowable arrangement of symbols

semantics

meaning of arrangement

compiler

checks syntax

converts to machine code via semantics

uses libraries to support regular ops.