Computer Systems

Seminar 3

Petru Eles
petel@ida.liu.se
tf: 281396

Some few slides are based on material from the course book as well as from the book “Computer Systems: A Programmer’s Perspective” by Bryant & O’Hallaron

Binary Representation

- Information is represented as a sequence of binary digits: Bits

- What the actual bits represent depends on the context:
  - Numerical value (integer, floating point, fixed point)
  - Sequence of characters (text)
  - Executable instruction

Binary Representation

- Information is represented as a sequence of binary digits: Bits

- What the actual bits represent depends on the context:
  - Numerical value (integer, floating point, fixed point)
  - Sequence of characters (text)
  - Executable instruction

- Depending on the context, operations performed are:
  - Logical computation (context: logic)

    1: true
    0: false

    Operations: And, Or, Exclusive-Or (Xor), Not
  - Numerical Computation (context: numbers)

    1
    0

    Operations: Addition, Subtraction, Multiplication, Division

Logical Computation: Boolean Algebra

And

&	0	1
0	0	0
1	0	1

Or

|	0	1
0	0	1
1	1	1

Xor

^	0	1
0	0	1
1	1	0

Not

~	
0	1
1	0

Logical Computation: Boolean Algebra

And

&	0	1
0	0	0
1	0	1

Or

|	0	1
0	0	1
1	1	1

Xor

^	0	1
0	0	1
1	1	0

Not

~	
0	1
1	0

- It applies similarly to bit vectors (operations apply bitwise):

```
  11100101        11100101        11100101
& 01101101      | 01101101      ^ 01101101      ~ 01101101
----------      ----------      ----------      ----------
  01100101        11101101        10001000        10010010
```

Arithmetical Computation

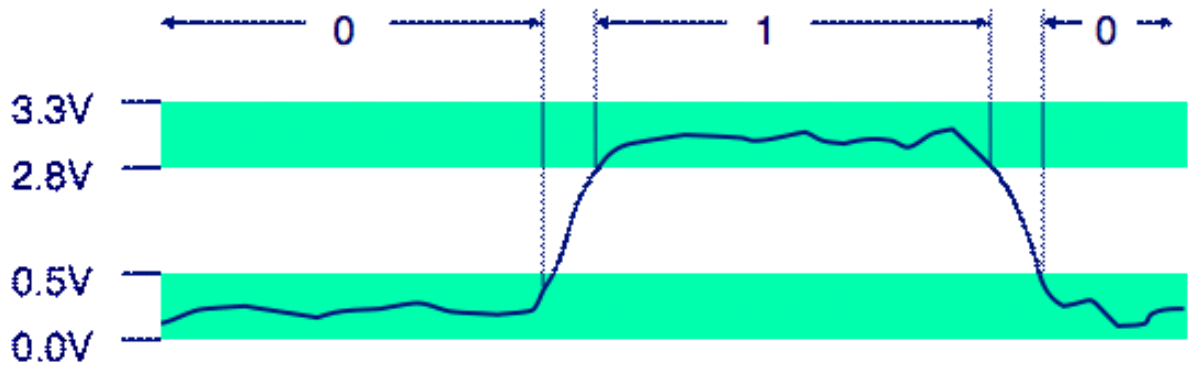
Adding two one bit numbers

A	B	Σ	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

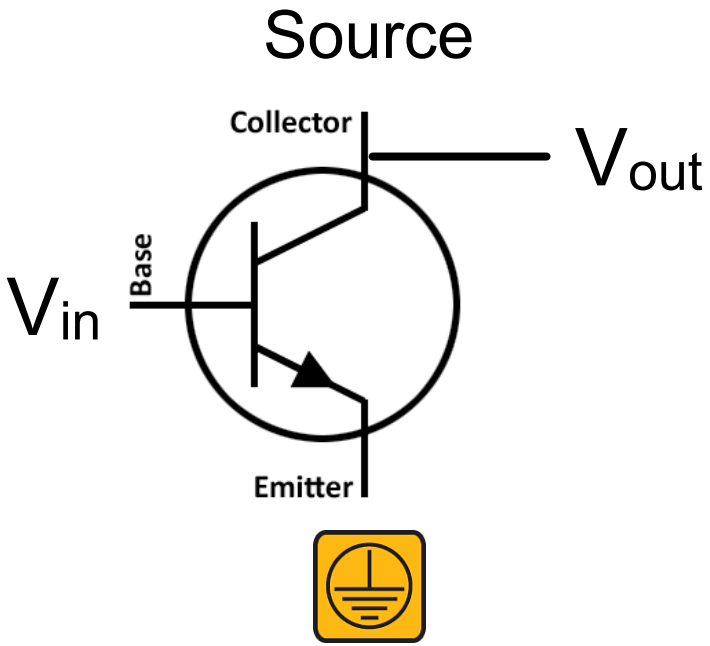
How is this Done in Computers?

- Logic values are represented by voltage levels:
  - High Voltage (e.g. 3.3V):1
  - Low Voltage (e.g. 0V): 0

At the output of a circuit we can have the following signal; this circuit produces the sequence 0, 1, 0 (or false, true, false):



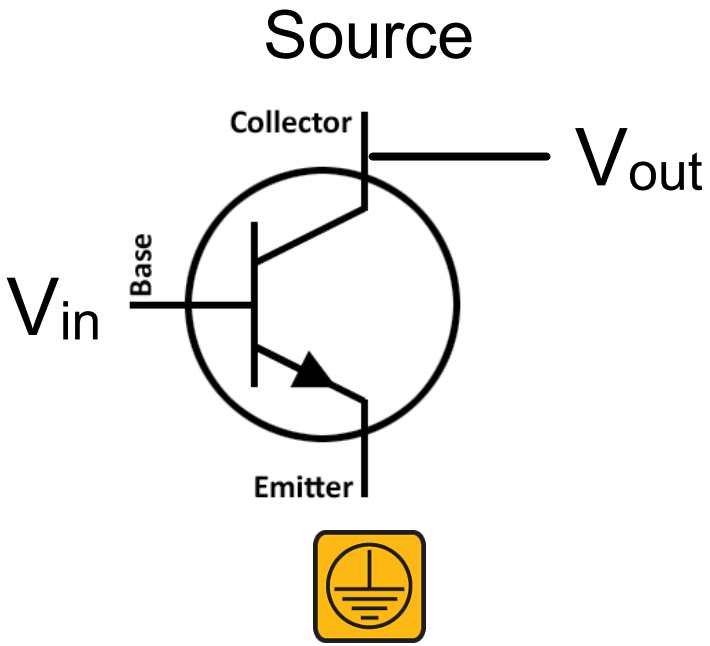
The Basic Building Block: The Transistor



V_{in}	Source	V_{out}
H	H	L
L	H	H

H: high voltage level (1, true)
L: low voltage level (0, false)

The Basic Building Block: The Transistor

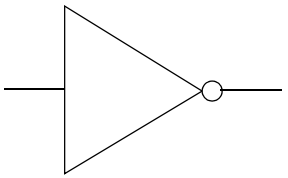


Observe!
This implements logic *Not* from V_{in} to V_{out}!

~	
0	1
1	0

V_{in}	Source	V_{out}
H	H	L
L	H	H

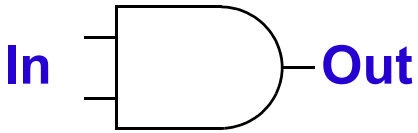
Such a circuit is called a *Not gate*
(also *inverter*):



H: high voltage level (1, true)
L: low voltage level (0, false)

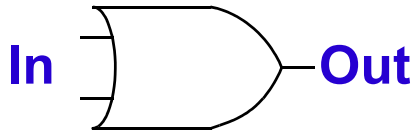
Gates for Boolean Operations

- Gates are electronic devices that perform Boolean operations.
  - Gates are built as small electronic circuits based on transistors;
  - Gates are the basic building blocks out of which VLSI (very Large Scale Integration) circuits are built; today computers are implemented as VLSI circuits, *with up to billions of transistors on a chip*.



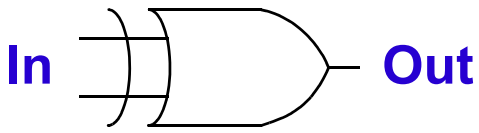
And

In_1	In_2	Out
0	0	0
0	1	0
1	0	0
1	1	1



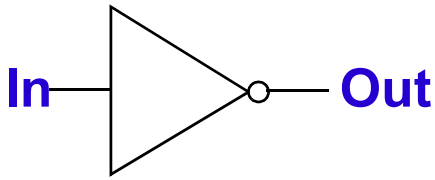
Or

In_1	In_2	Out
0	0	0
0	1	1
1	0	1
1	1	1



Xor

In_1	In_2	Out
0	0	0
0	1	1
1	0	1
1	1	0



Not

In	Out
0	1
1	0

Gates for Boolean Operations

- Gates are electronic devices that perform Boolean operations.
  - Gates are built as small electronic circuits based on transistors;
  - Gates are the basic building blocks out of which VLSI (very Large Scale Integration) circuits are built; today computers are implemented as VLSI circuits, *with up to billions of transistors on a chip*.

- Any logical function can be implemented as a combination of such gates.

Implementing Arithmetical Computation

- The one bit adder:

A	B	Sum	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Sum = A Xor B
C_{out} = A And B

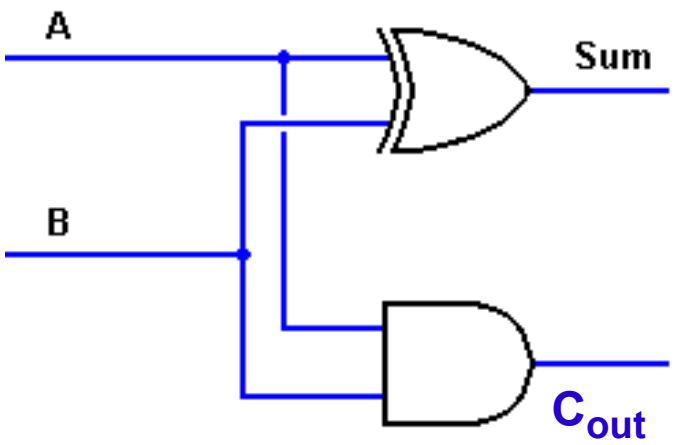
- This is just a truth table capturing a logical function; thus, it can be implemented with a combination of logical gates!

Implementing Arithmetical Computation

- The one bit adder:

A	B	Sum	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Here is the circuit:



Sum = A Xor B
C_{out} = A And B

- This is just a truth table capturing a logical function; thus, it can be implemented with a combination of logical gates!

This is called a *Half Adder* (does not consider input carry).

Implementing Arithmetical Computation

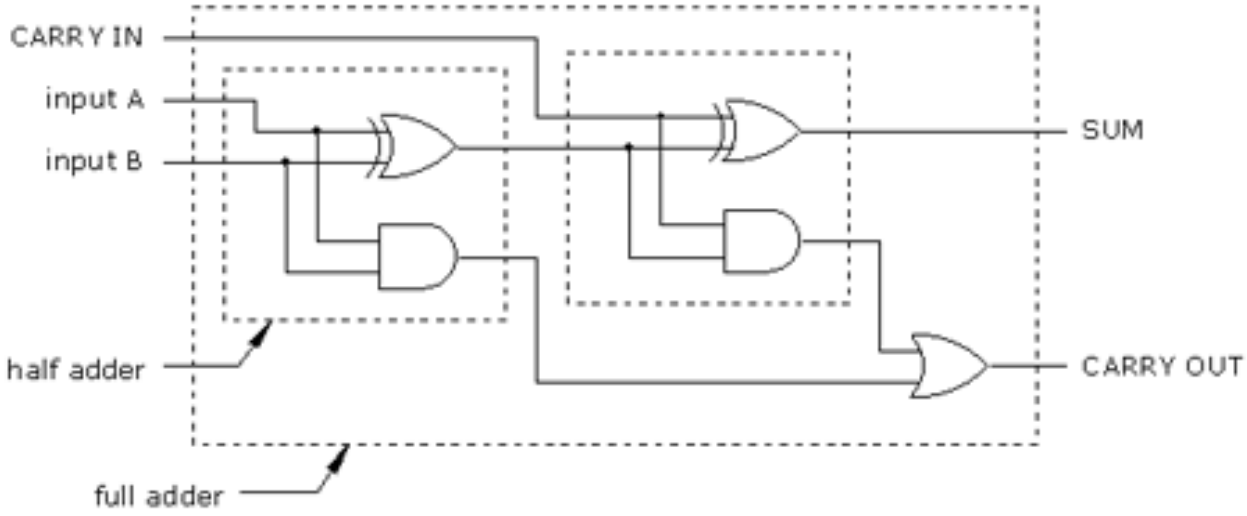
- The Full Adder (adds two bits and input carry):

C_{in}	A	B	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Implementing Arithmetical Computation

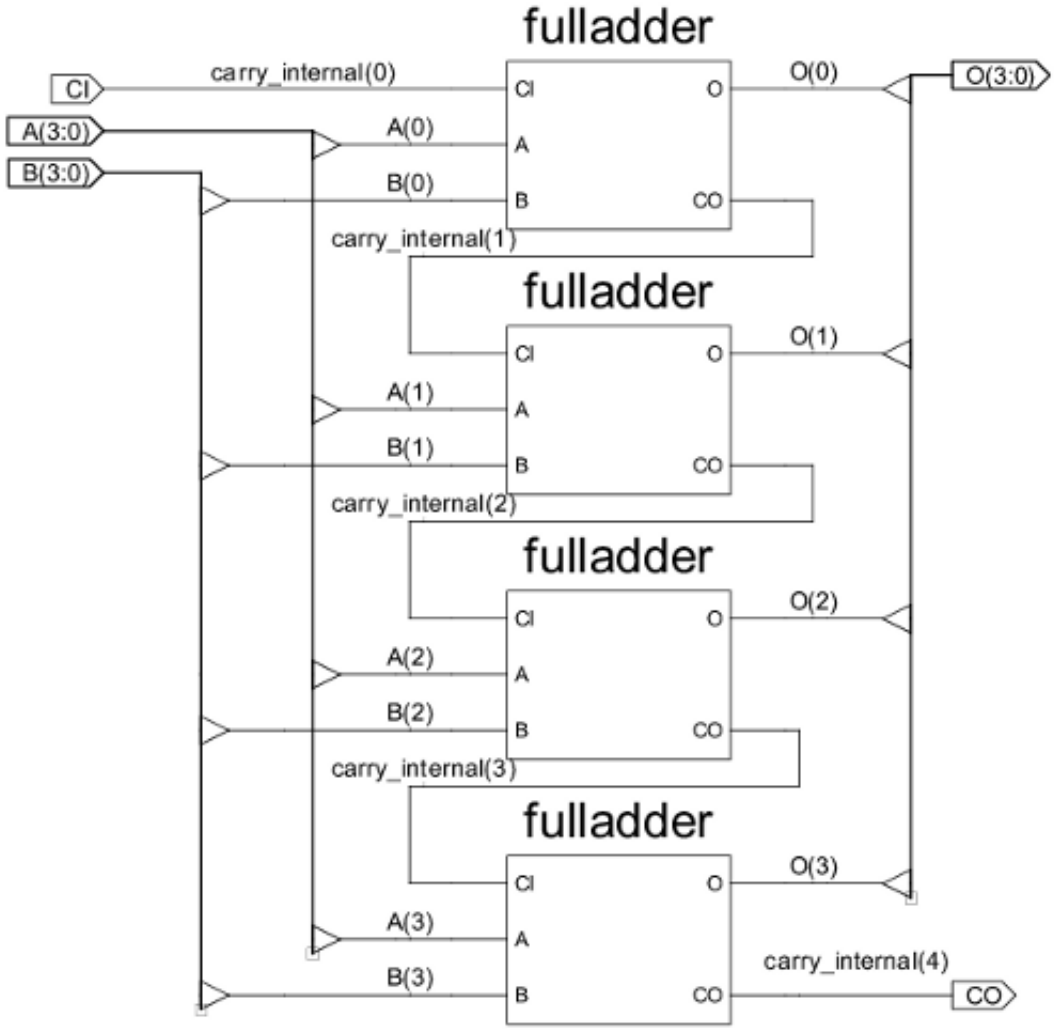
- The Full Adder (adds two bits and input carry):

C_{in}	A	B	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Implementing Arithmetical Computation

- A four bits adder: adds two four bits numbers and an input carry:



- By further cascading full adders, one can build 8, 16, 32, 64, ... bit adders.

- In a similar way, circuits for other arithmetic operations can be implemented.

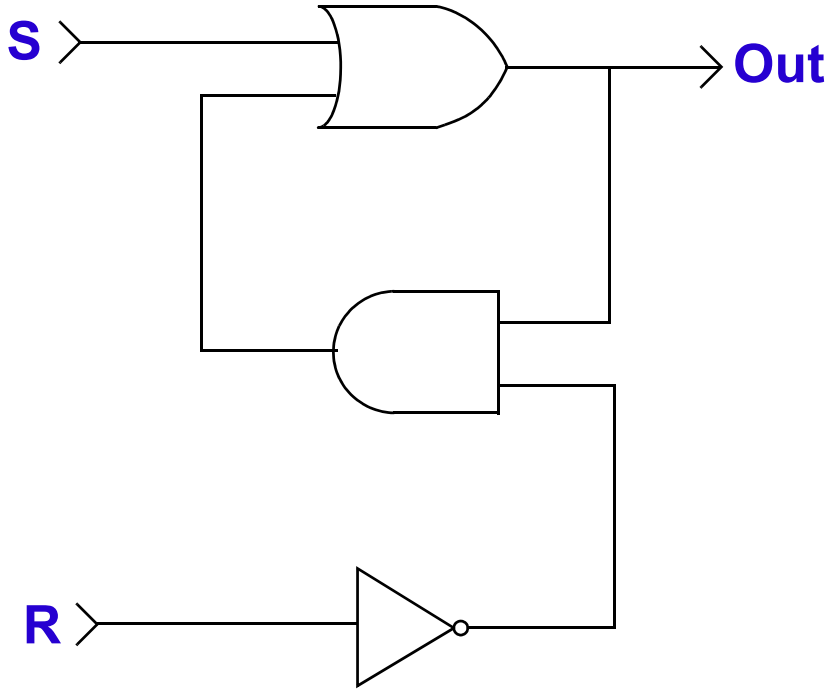
How to Store a BIT?

- Circuits like those shown in the previous slides are called “combinatorial”: they produce an output that only depends on the input; the output is maintained as long as that input is applied.

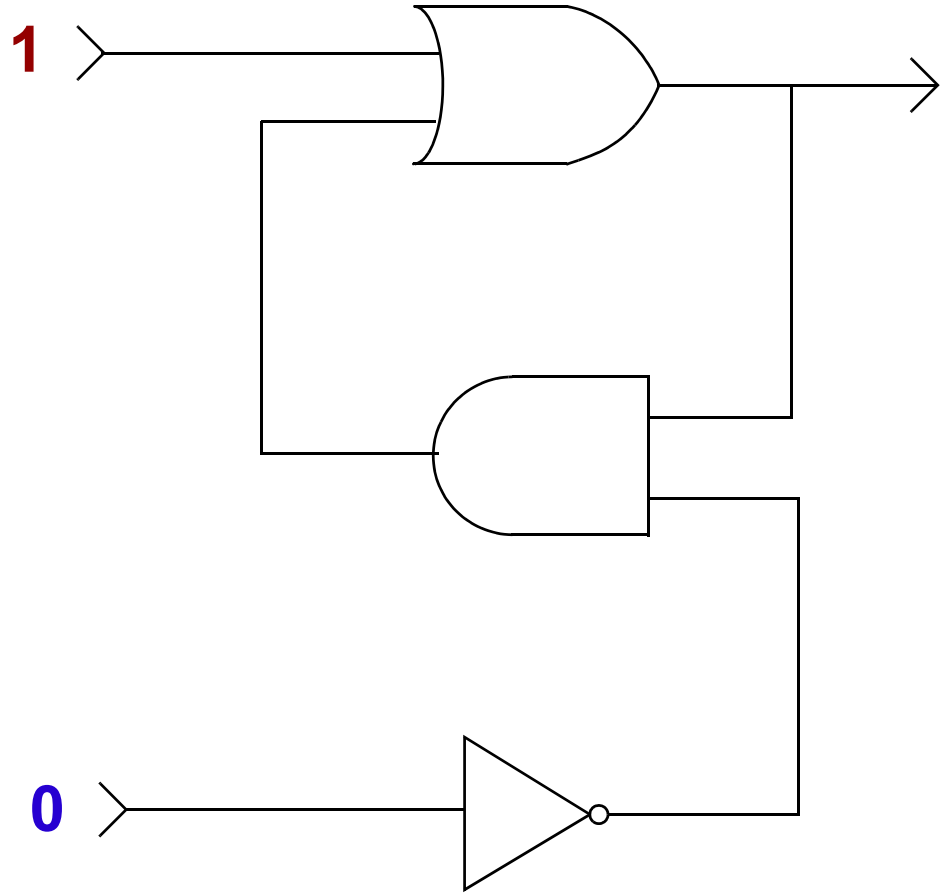
- What about a circuit that is able to store a bit?
  You can write 1 or 0 to the circuit, and the output will keep the value also after the input has disappeared.

Flip-Flops

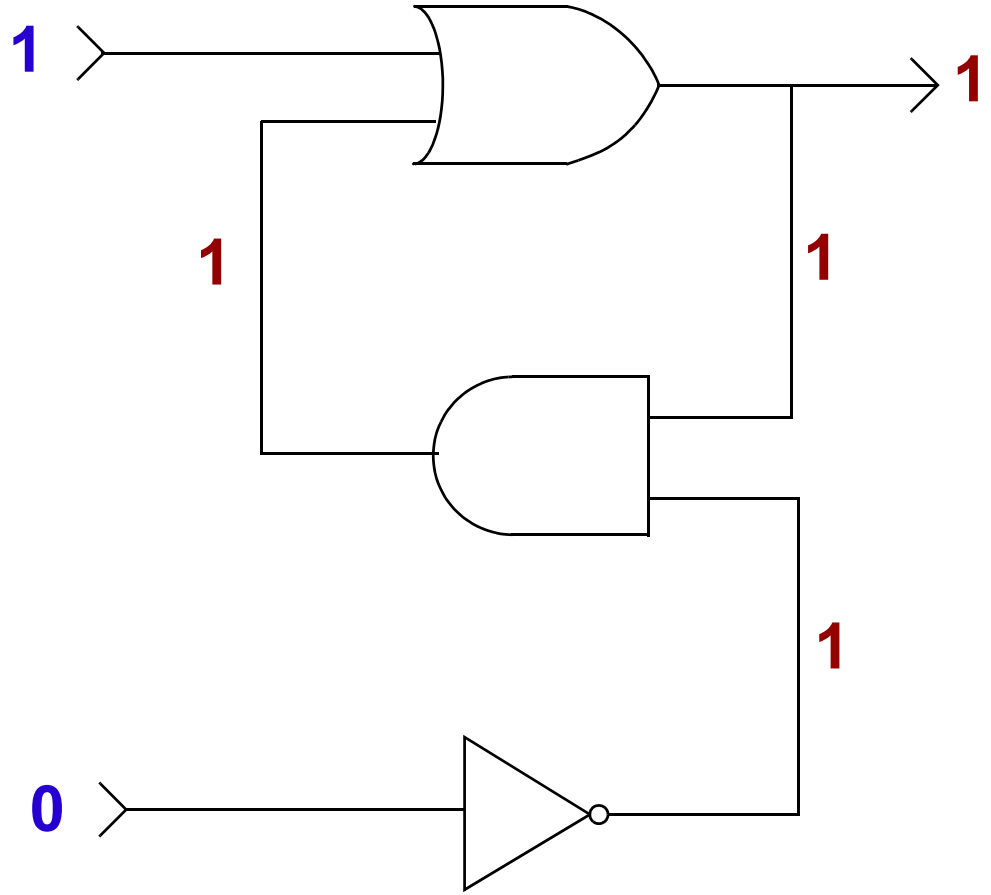
- The circuit below has two inputs. One (called *S*) for setting it to 1 (H), the other (called *R*) for setting to 0 (L).
  - When there arrives an input 1 to *S*, the output becomes 1; it will stay 1, until there comes an input 1 to R.
  - Once an input 1 arrived to R, the output switches 0, and stays so until an 1 arrives to S.



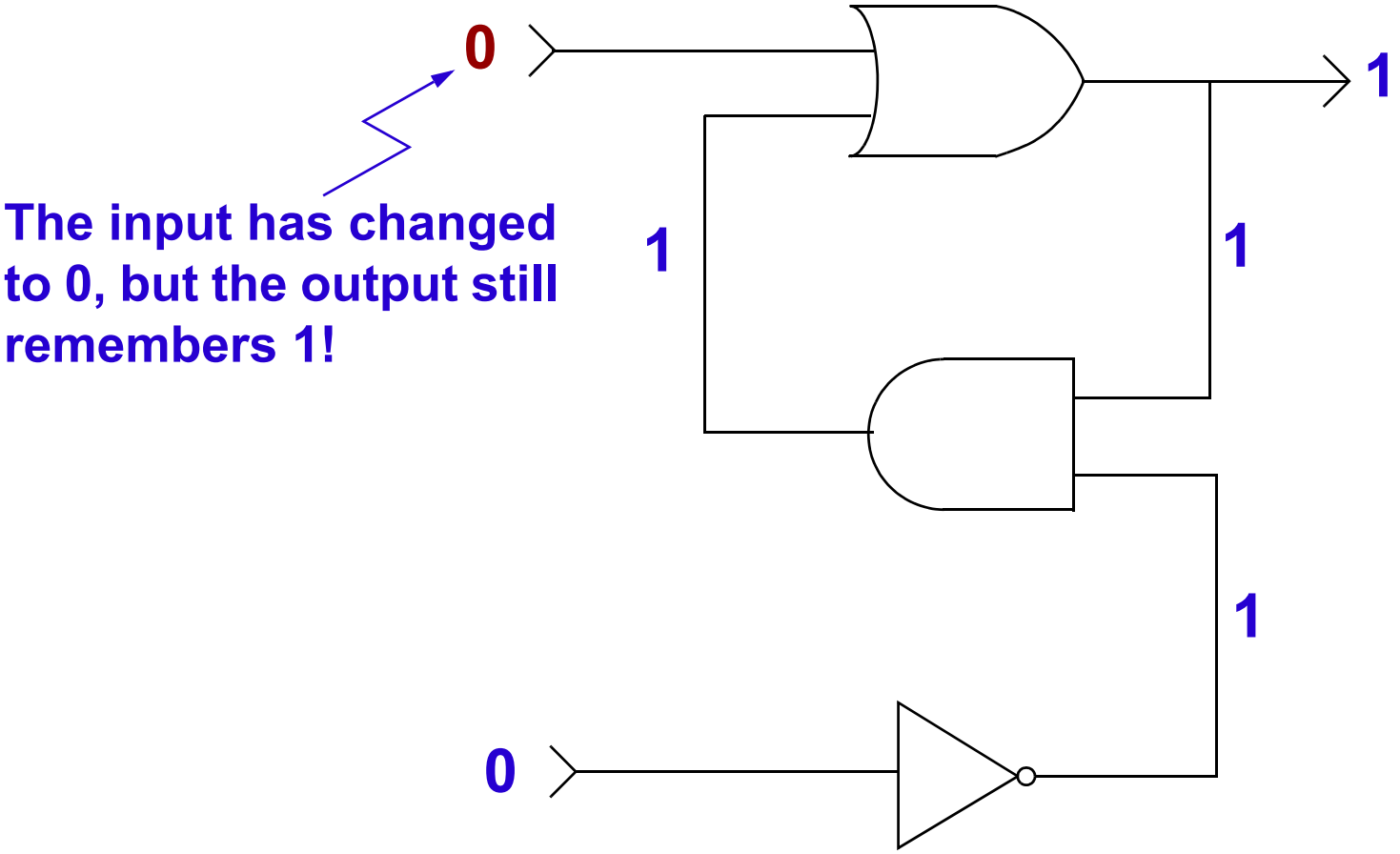
Setting the Flip-Flop to 1



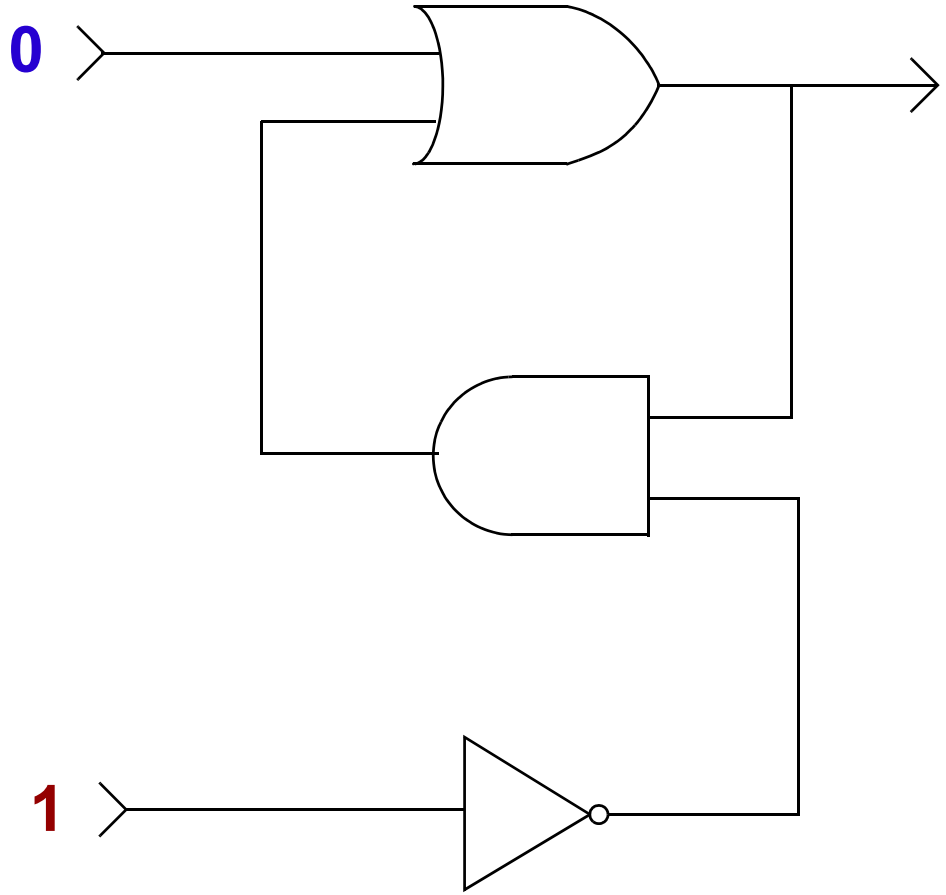
Setting the Flip-Flop to 1



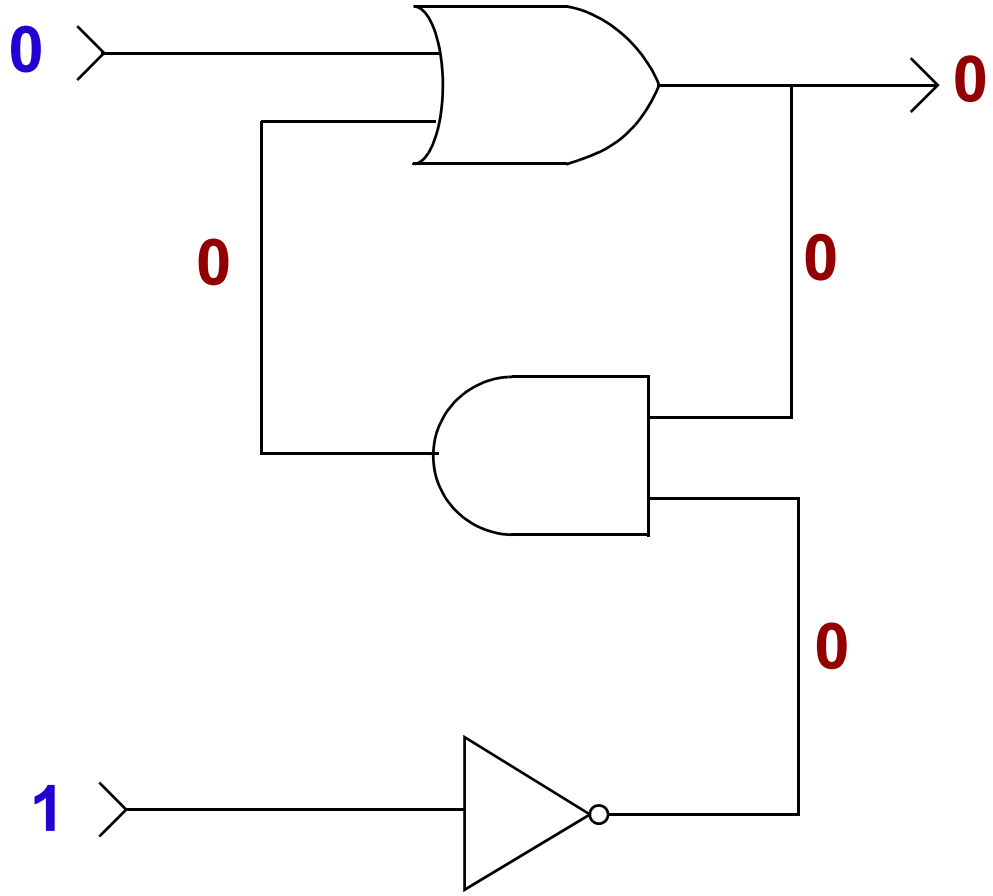
Setting the Flip-Flop to 1



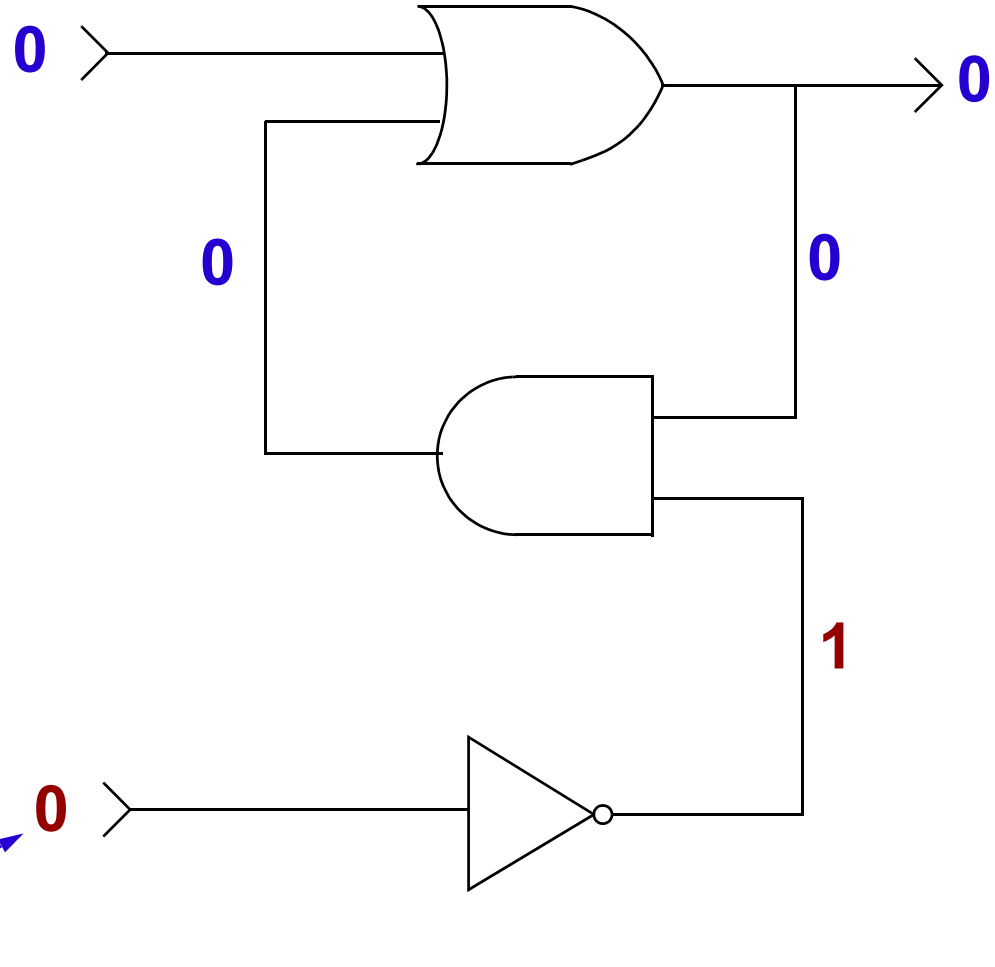
Setting the Flip-Flop to 0



Setting the Flip-Flop to 0



Setting the Flip-Flop to 0



The input has changed to 0, but the output still remembers 0!

Flip-Flops

- One flip-flop can store one bit. Using groups of several flip-flops, arbitrary long sequences of bits can be stored. This is a basic technique to store data in computers e.g. in registers.

Let's Go Over to Computers

- We have seen how data (logical and numerical) is represented in a computer.
- We have seen that it is possible to construct circuits that are able to operate on data and perform logical and arithmetical operations.
- We have seen that circuits can be built which are able to store data.

Let's Go Over to Computers

- We have seen how data (logical and numerical) is represented in a computer.
- We have seen that it is possible to construct circuits that are able to operate on data and perform logical and arithmetical operations.
- We have seen that circuits can be built which are able to store data.

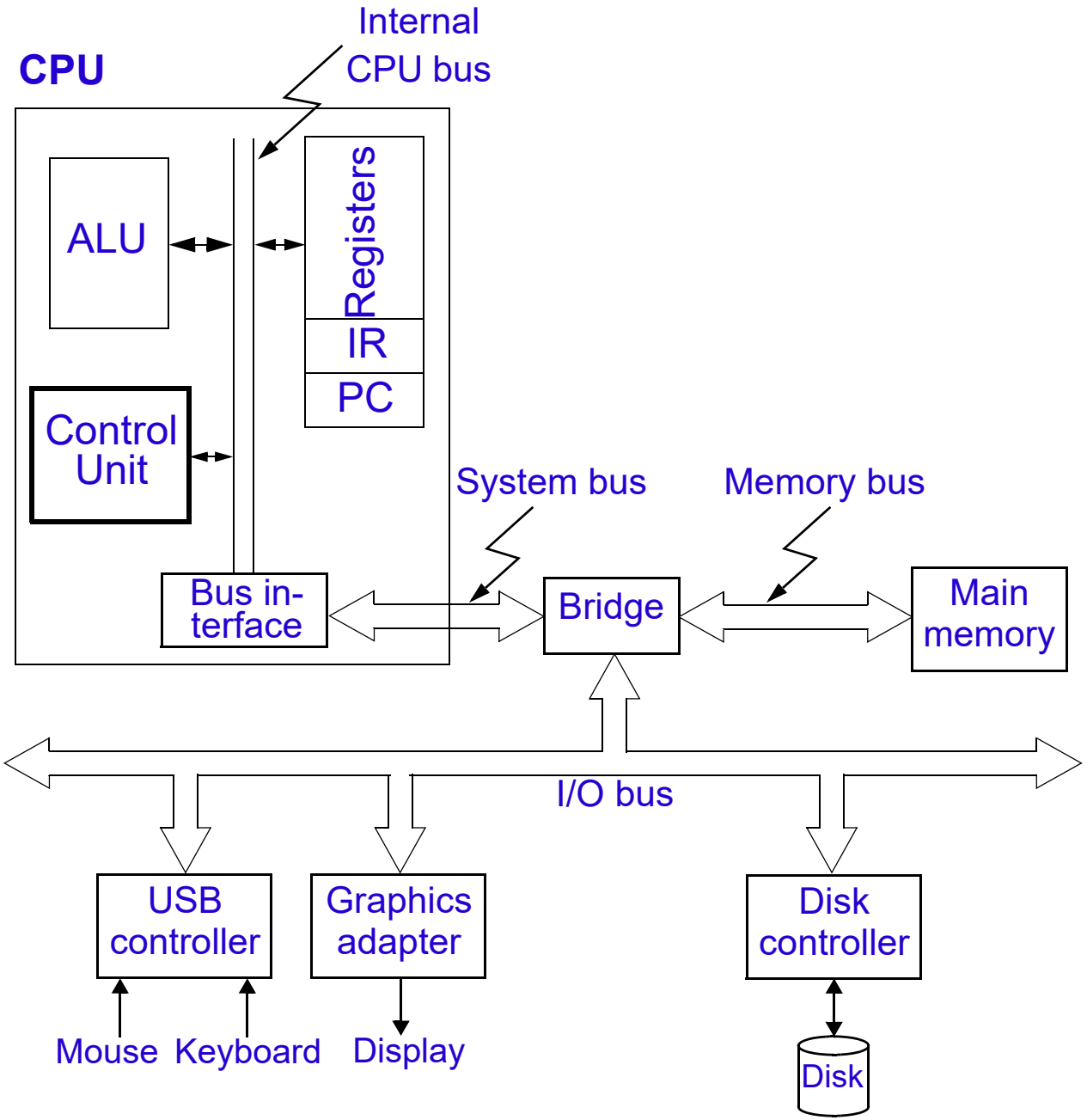
Now, let's see how a computer is built and works!

What is a Computer/Computer-System?

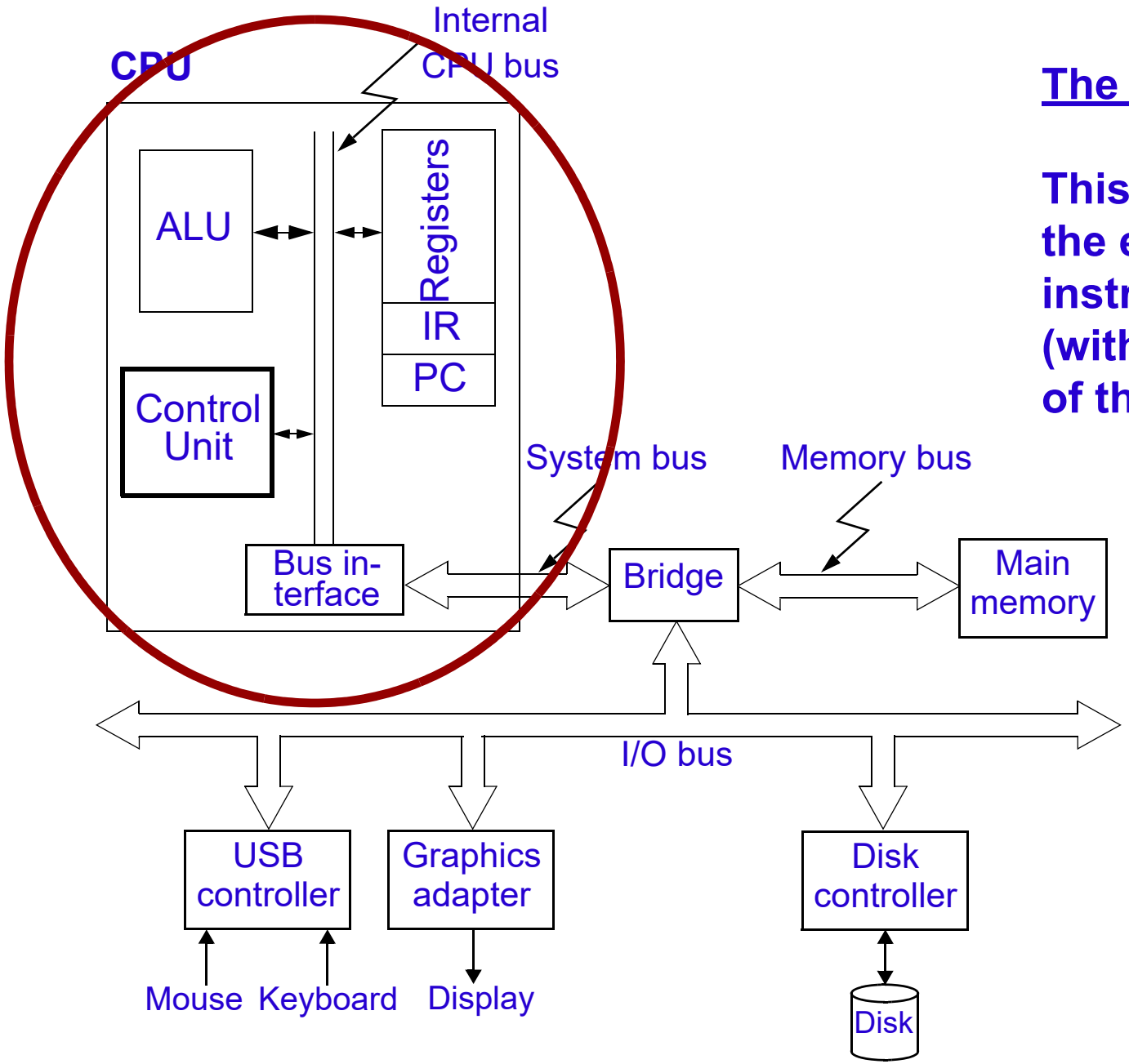
- A computer is a data processing machine which is operated automatically under the control of a list of instructions (called a program) stored in its main memory.

- Computers today are extremely complex and are built of many interconnected components; in addition to actual data processing, they have to perform tasks, such as communicate with other computers and devices, to interact with the user and the environment, etc. Therefore, we speak about *Computer Systems*.

Computer Systems



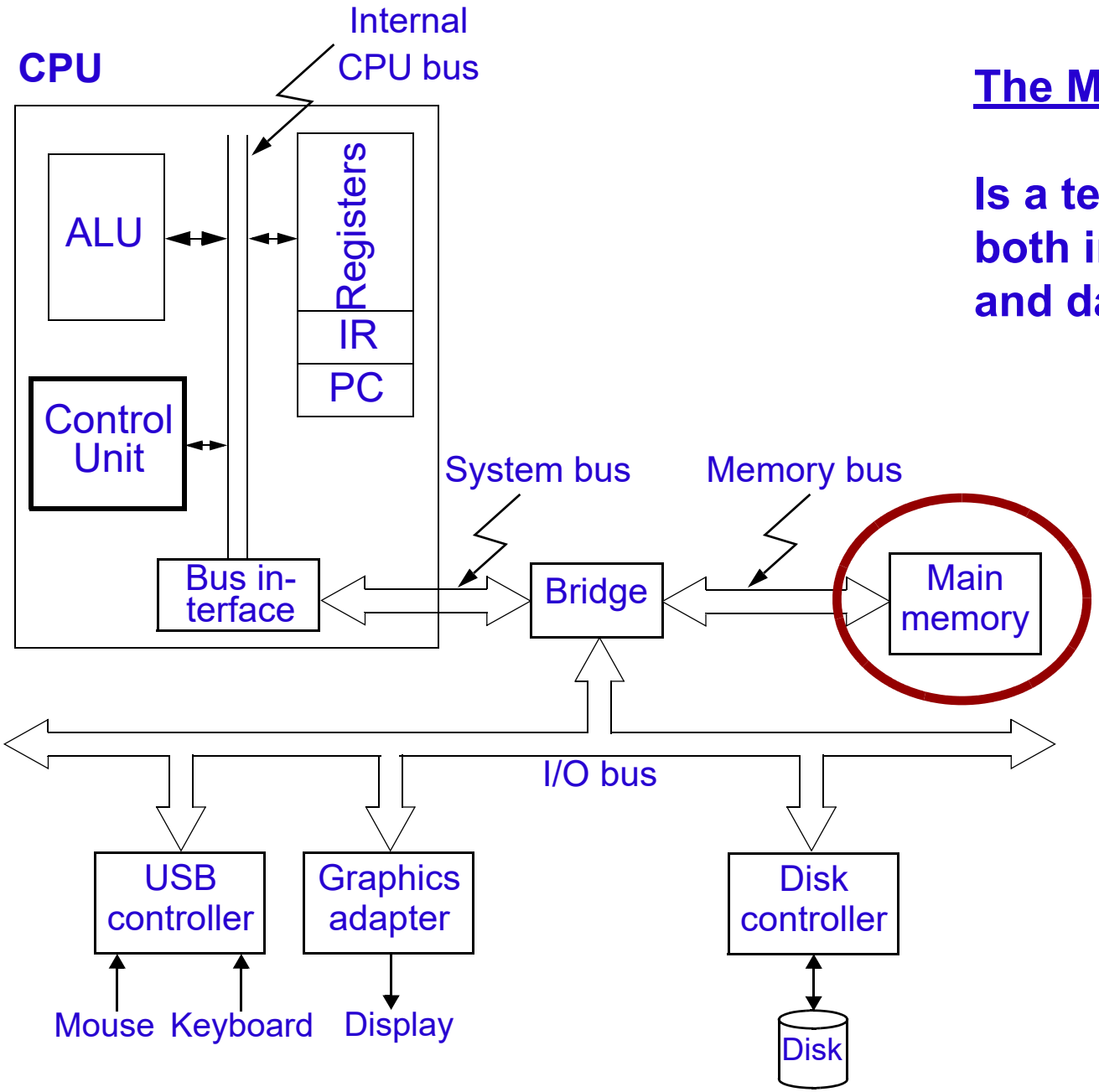
Computer Systems



The CPU (Central Processing Unit)

This is the hart of the system; it is the engine that interprets the instructions and executes them (with the help of other components of the computer system).

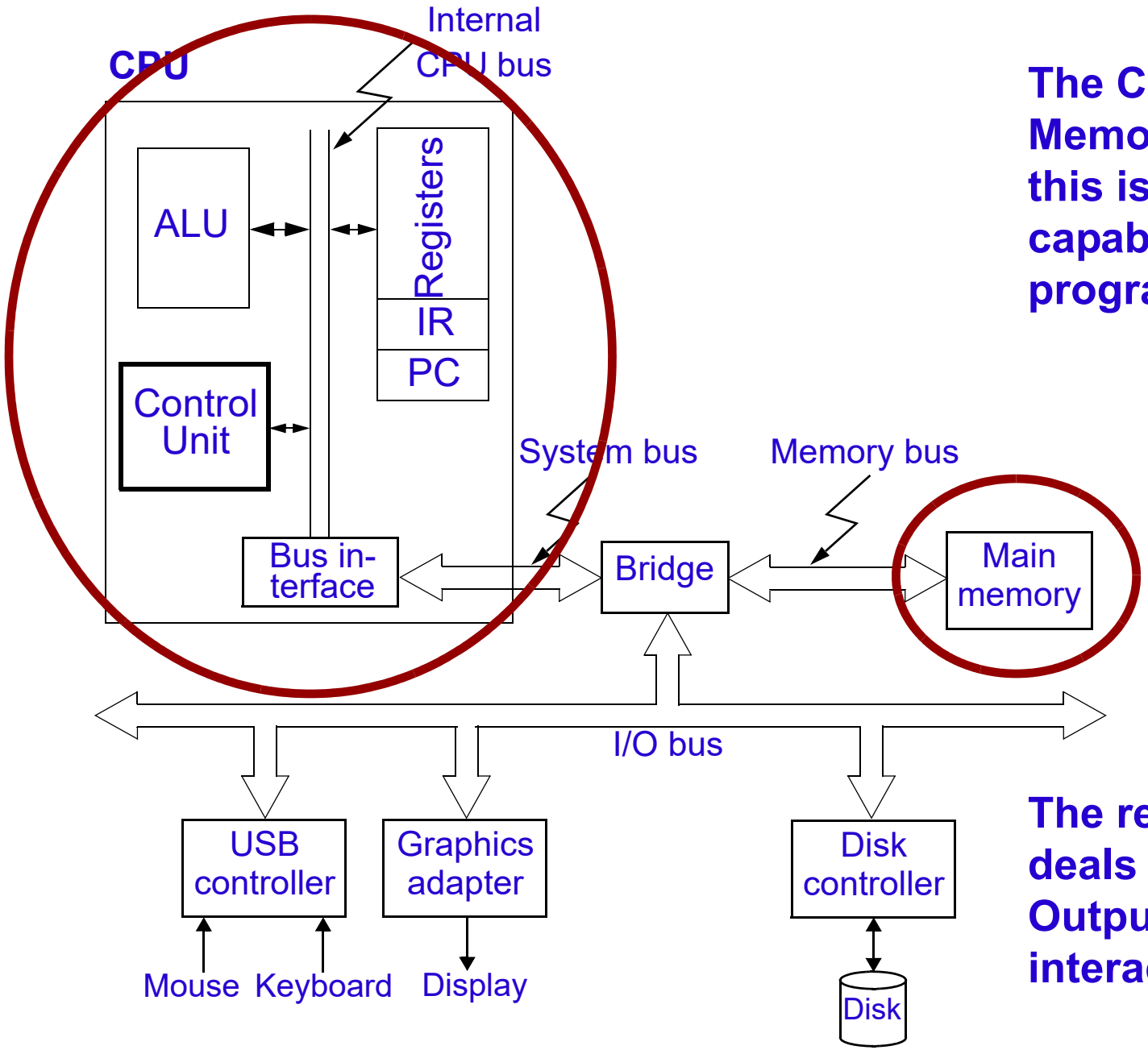
Computer Systems



The Main Memory

Is a temporary storage that stores both instructions (the program) and data.

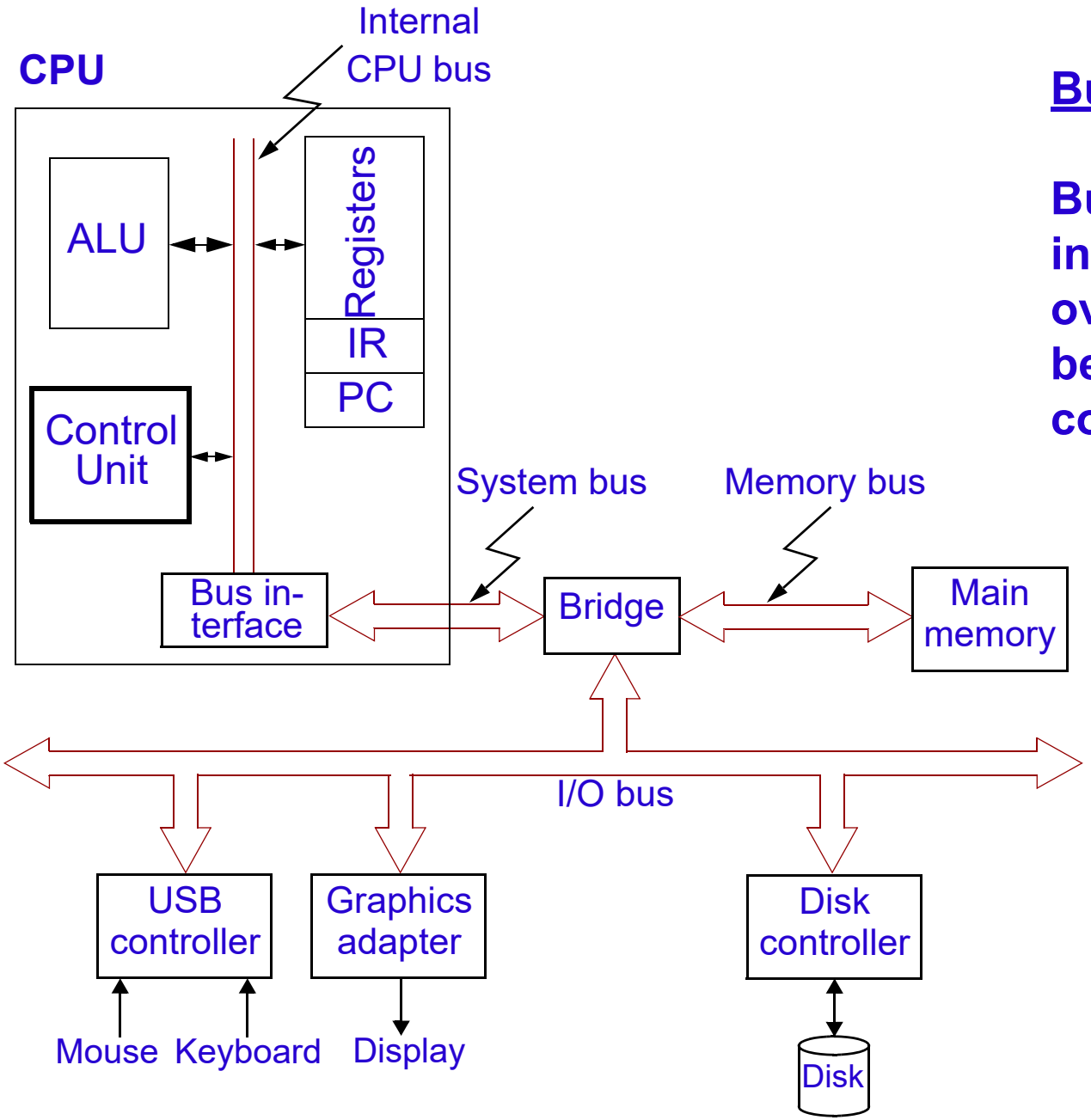
Computer Systems



The CPU together with the Main Memory build the core computer; this is the minimal structure capable of storing and executing programs.

The rest of the computer system deals with communication, Input/ Output, long term storage, and interaction with the environment.

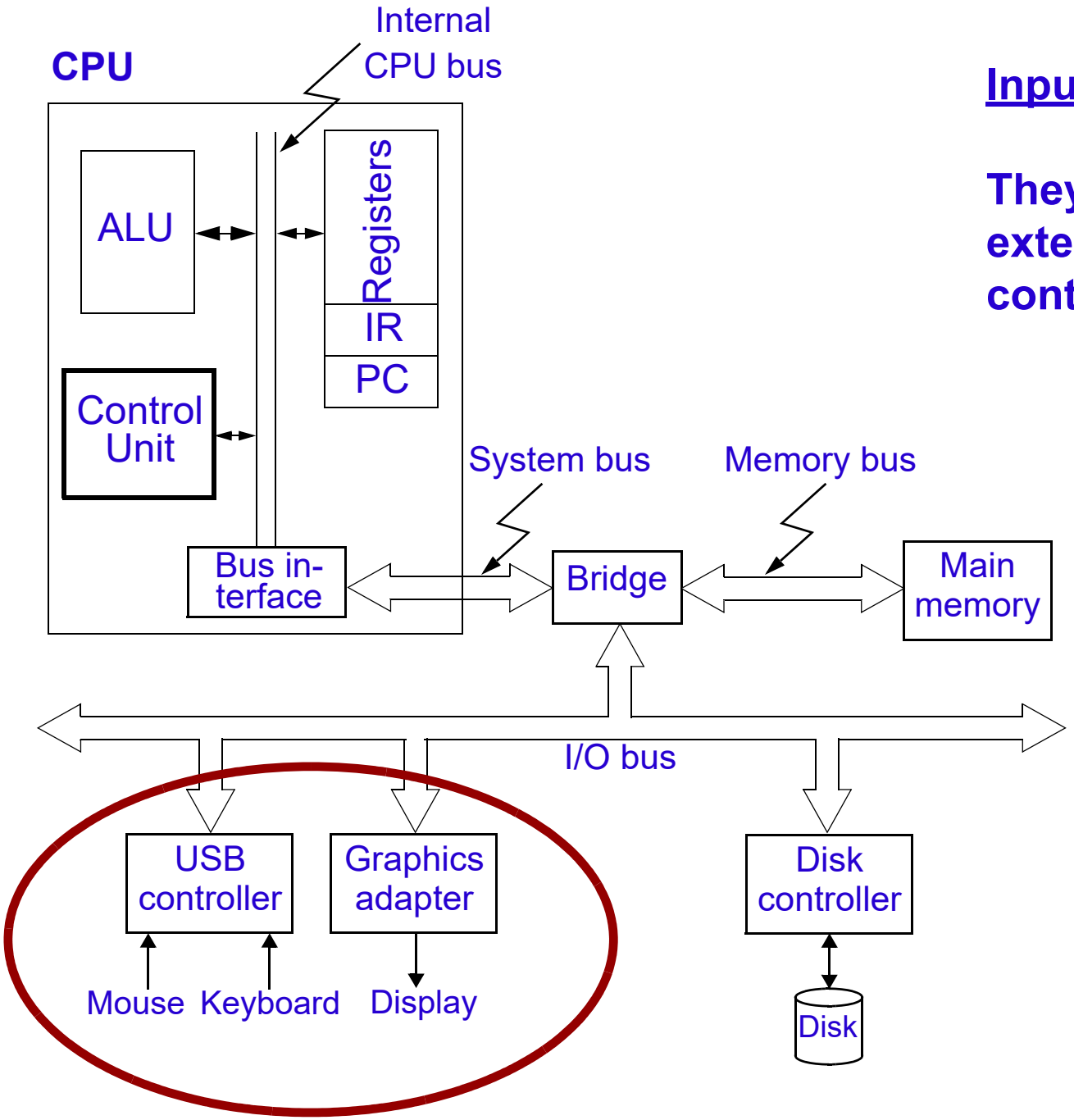
Computer Systems



Buses

Buses are the physical infrastructure (electrical wiring) over which bytes are travelling between components of the computer system.

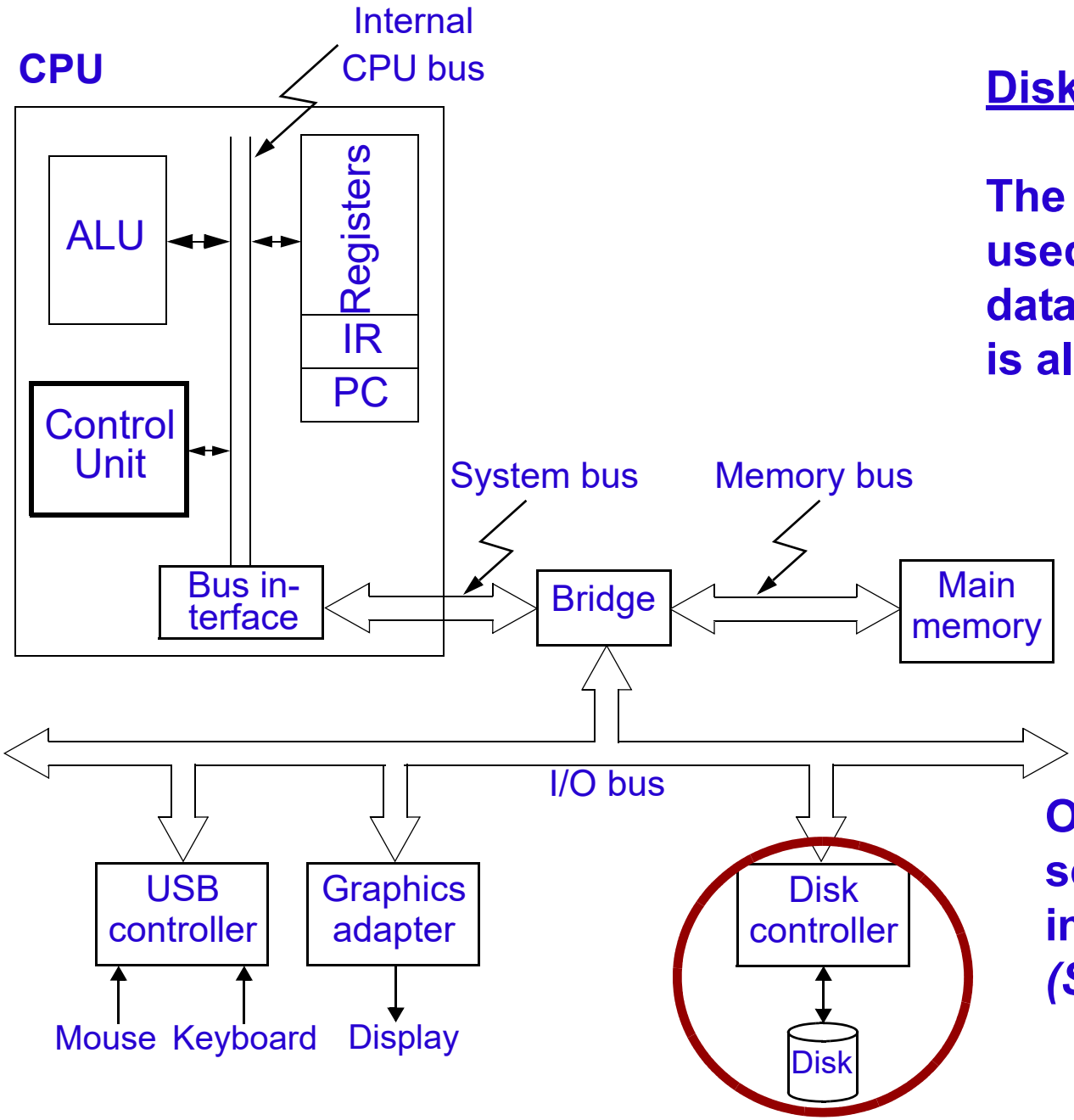
Computer Systems



Input/Output Devices

They connect the computer to the external world. Connection is via controllers/adaptors.

Computer Systems



Disk drive

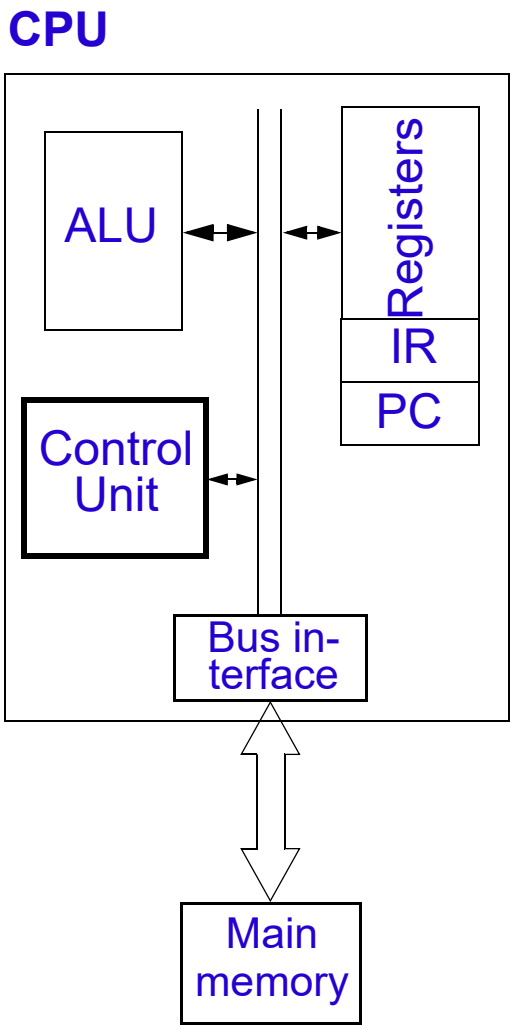
The disk drive is a special device used as a long term storage for data and programs. Such a storage is also called *Secondary Memory*.

On modern computers the secondary memory is often implemented as *solid state disk (SSD)* on *flash memory*.

How Does a Computer Work?

- All computers in use, simple or complicated, big or small, cheap or expensive work according to the same basic concept, known as the von Neumann architecture:
  - Data and instructions are both stored in the main memory (stored program concept);
  - The content of the memory is addressable by location (without regard to what is stored in that location);
  - Instructions are executed sequentially (from one instruction to the next, in order of their location in memory) unless the order is explicitly modified.

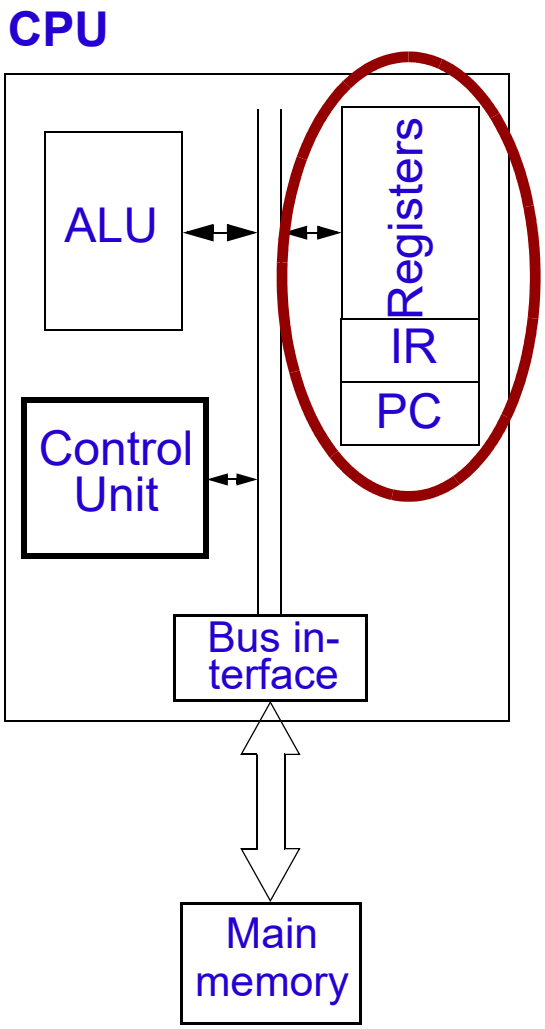
A Simple Computer Architecture



The basic organization (architecture):

- ***Central processing unit*** **(CPU) contains:**
  - ***Control unit*** **(CU) that coordinates the execution of instructions;**
  - ***Arithmetic/logic unit*** **(ALU) that performs arithmetic and logic operations;**
  - **A set of *registers.***
- ***Main memory*****.**

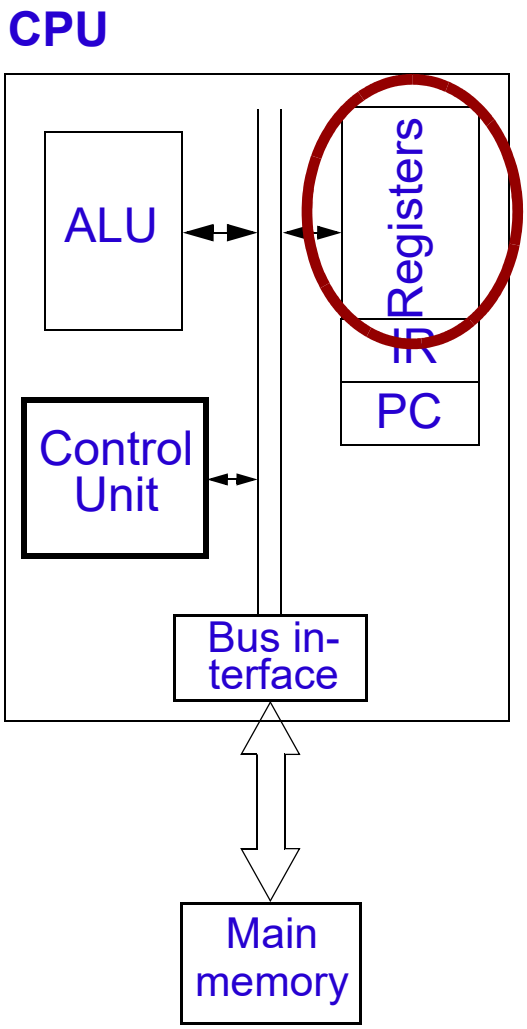
A Simple Computer Architecture



Register Organization

- The set of registers within the CPU represents the top level of the memory hierarchy inside the computer system:
  - User visible registers: can be accessed by programs, for data storing.
  - Control and Status registers: used by the Control Unit to control the operation of the CPU; not directly accessible by the programmer.

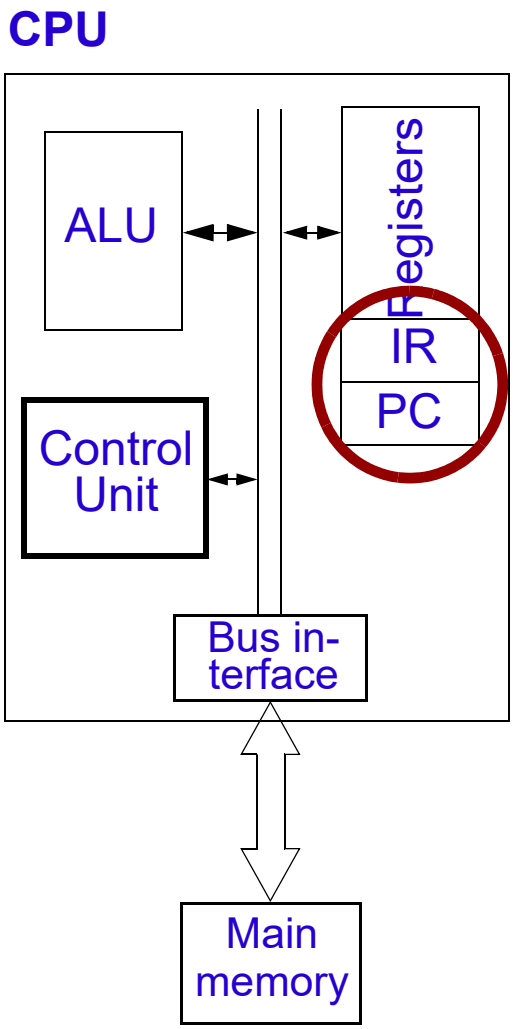
A Simple Computer Architecture



User Visible Registers

- **A set of registers which can be used without restrictions as operands for any operation and as address registers; these are so called *general-purpose registers*.**

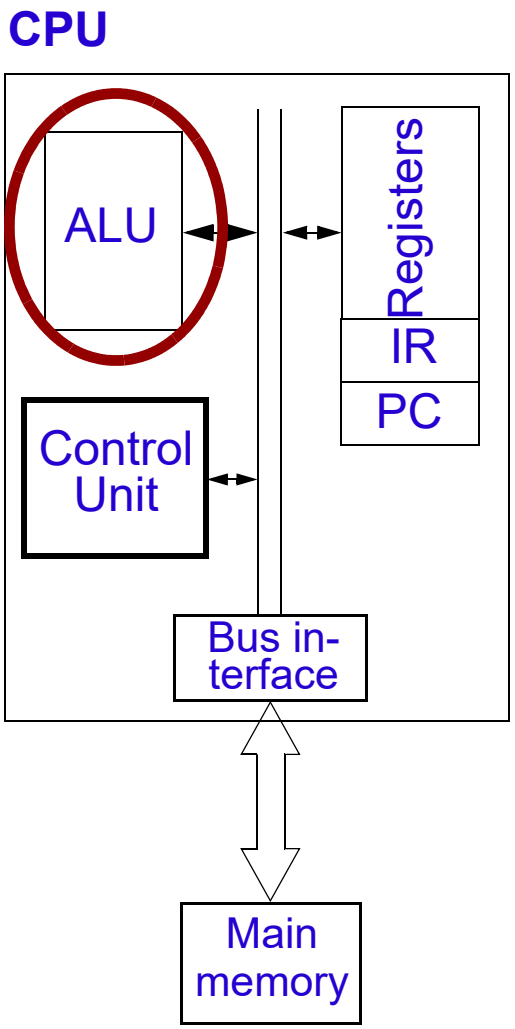
A Simple Computer Architecture



Control and Status Registers

- Program Counter (PC): holds the address of the instruction to be fetched and executed.
- Instruction Register (IR): holds the last instruction fetched.
- Program Status Word (PSW): Condition Code Flags + other bits defining the status of the CPU.
- . . . . . .

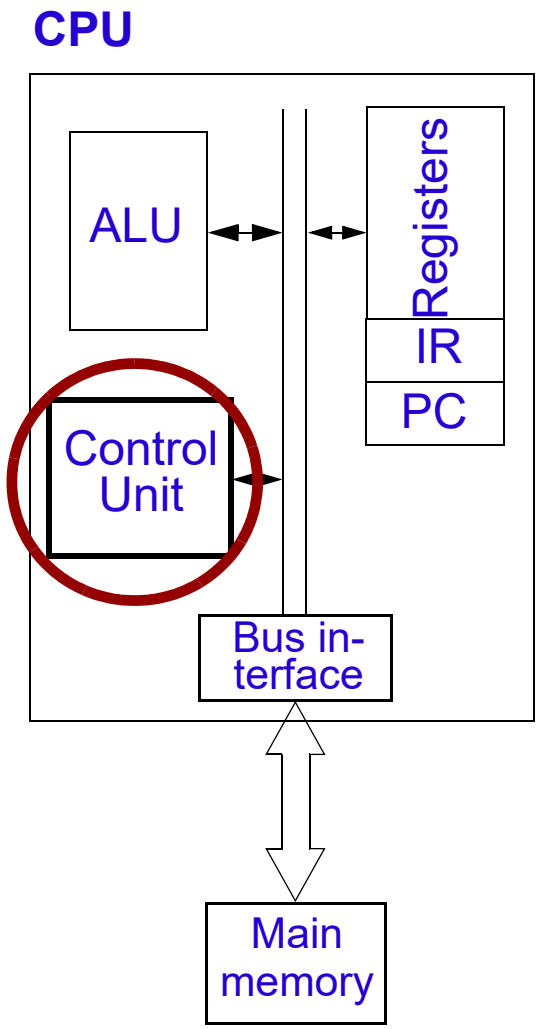
A Simple Computer Architecture



Arithmetic Logic Unit (ALU)

- Performs arithmetic and logic operations. There might be several of them in a CPU. ALUs are different, depending on the data type they operate on: integer ALU, floating point ALU, etc.

A Simple Computer Architecture



Arithmetic Logic Unit (ALU)

- Performs arithmetic and logic operations. There might be several of them in a CPU. ALUs are different, depending on the data type they operate on: integer ALU, floating point ALU, etc.

Control Unit

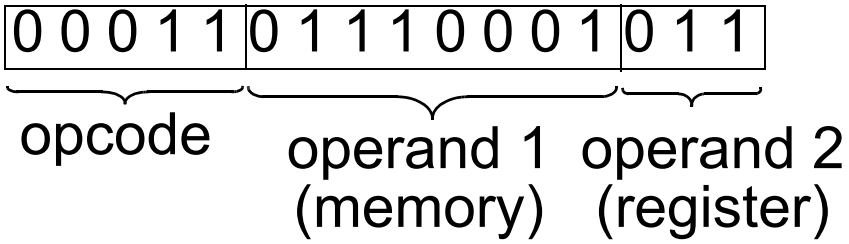
- The control unit generates the appropriate signals such that all other components of the CPU and the computer system, together, execute the current instruction.
- The current instruction to execute is stored in the instruction register (IR); it is the instruction whose memory address is stored in the program counter (PC)

Machine Instructions

- A CPU can only execute *machine instructions*,
- Each computer has a set of specific machine instructions which its CPU is able to recognize and execute.

Machine Instructions

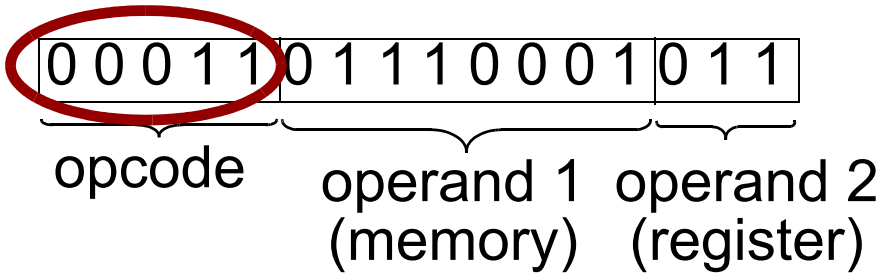
- A CPU can only execute *machine instructions*,
- Each computer has a set of specific machine instructions which its CPU is able to recognize and execute.



- A machine instruction is represented as a sequence of bits (binary digits). These bits are organized into *fields* that define:

Machine Instructions

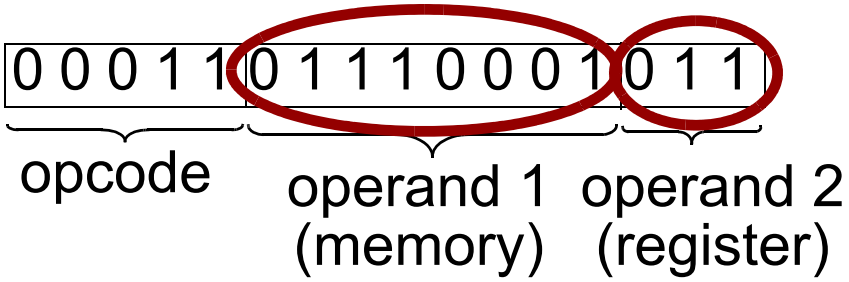
- A CPU can only execute *machine instructions*,
- Each computer has a set of specific machine instructions which its CPU is able to recognize and execute.



- A machine instruction is represented as a sequence of bits (binary digits). These bits are organized into *fields* that define:
  - What has to be done (the operation code).

Machine Instructions

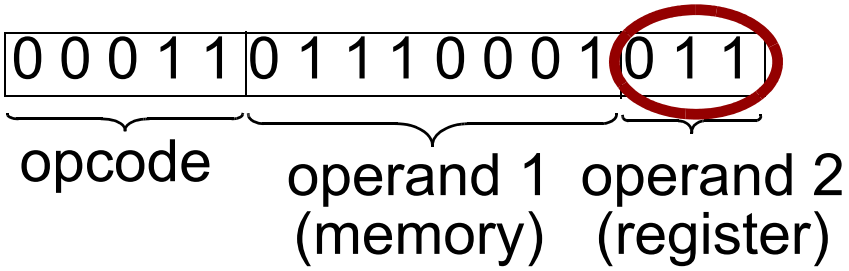
- A CPU can only execute *machine instructions*,
- Each computer has a set of specific machine instructions which its CPU is able to recognize and execute.



- A machine instruction is represented as a sequence of bits (binary digits). These bits are organized into *fields* that define:
  - What has to be done (the operation code).
  - To whom the operation applies (source operands).

Machine Instructions

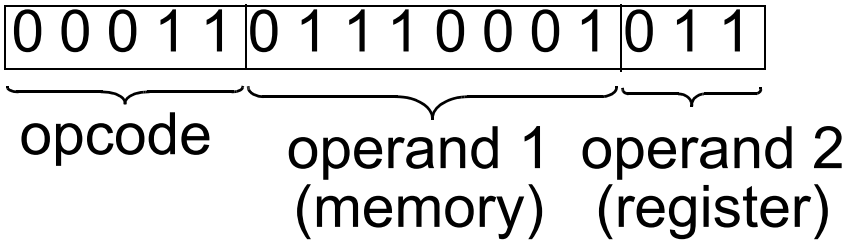
- A CPU can only execute *machine instructions*,
- Each computer has a set of specific machine instructions which its CPU is able to recognize and execute.



- A machine instruction is represented as a sequence of bits (binary digits). These bits are organized into *fields* that define:
  - What has to be done (the operation code).
  - To whom the operation applies (source operands).
  - Where does the result go (destination operand); in this example CPU it is assumed that the result of the operation is stored in the same place where the second operand was stored; no additional field is needed.

Machine Instructions

- A CPU can only execute *machine instructions*,
- Each computer has a set of specific machine instructions which its CPU is able to recognize and execute.



- The number of bits, number and length of the fields and their order is particular to each computer; this defines the *instruction format* of that computer.

Types of Machine Instructions

- Machine instructions are of four types:
  - Data transfer between memory and CPU registers
  - Arithmetic and logic operations
  - Program control (test and branch); these are those instructions that change the flow of instruction execution by *jumping* to an instruction *different* from the instruction following the current one in memory.
  - I/O transfer

You see, there are very simple things a machine instruction does!
But many machine instructions, together, perform the big thing!

Instruction Execution

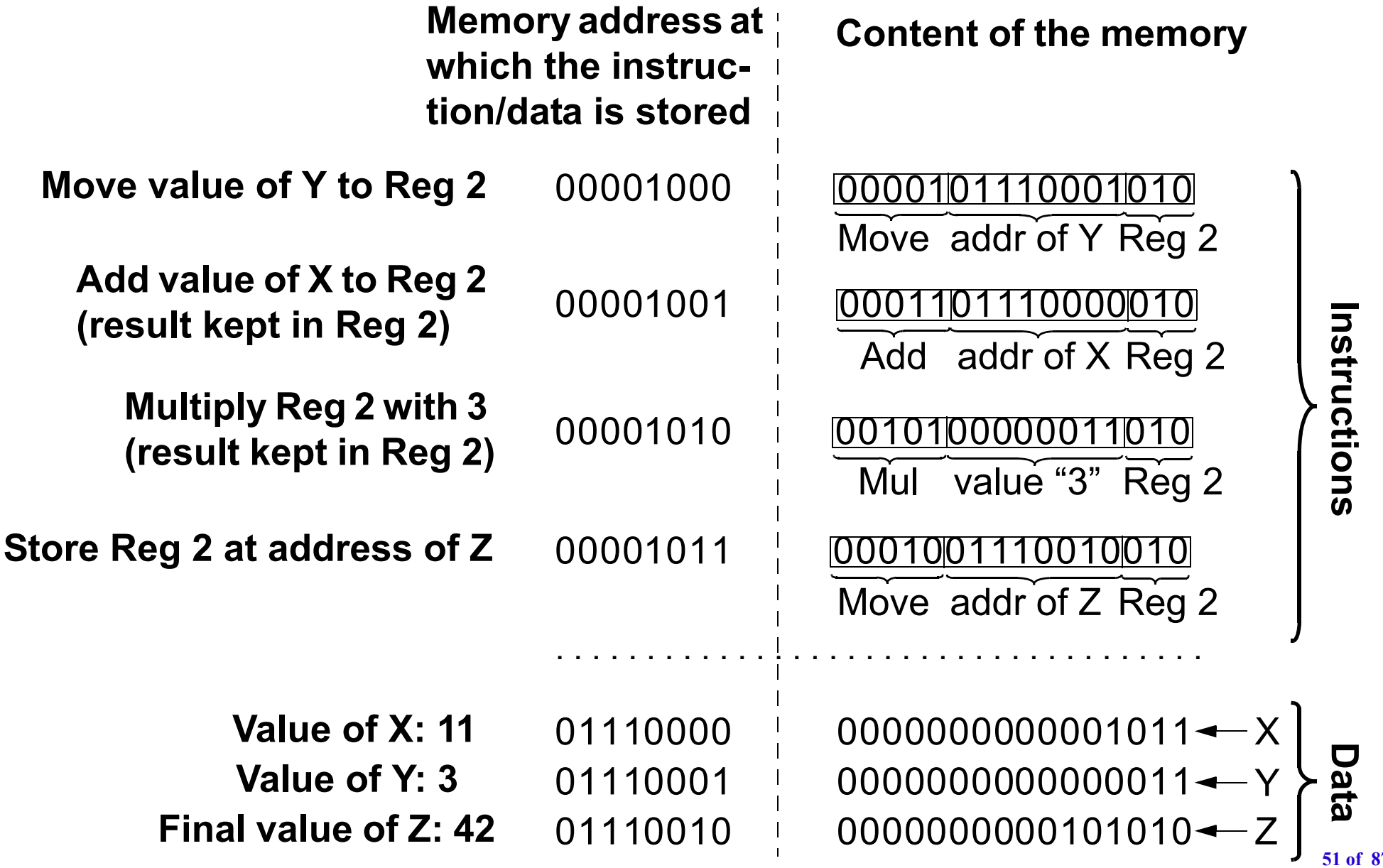
Let's imagine you write in a program the following instruction:

Z := (Y + X) * 3;

The instruction will be executed by the CPU as a sequence of four machine instructions!

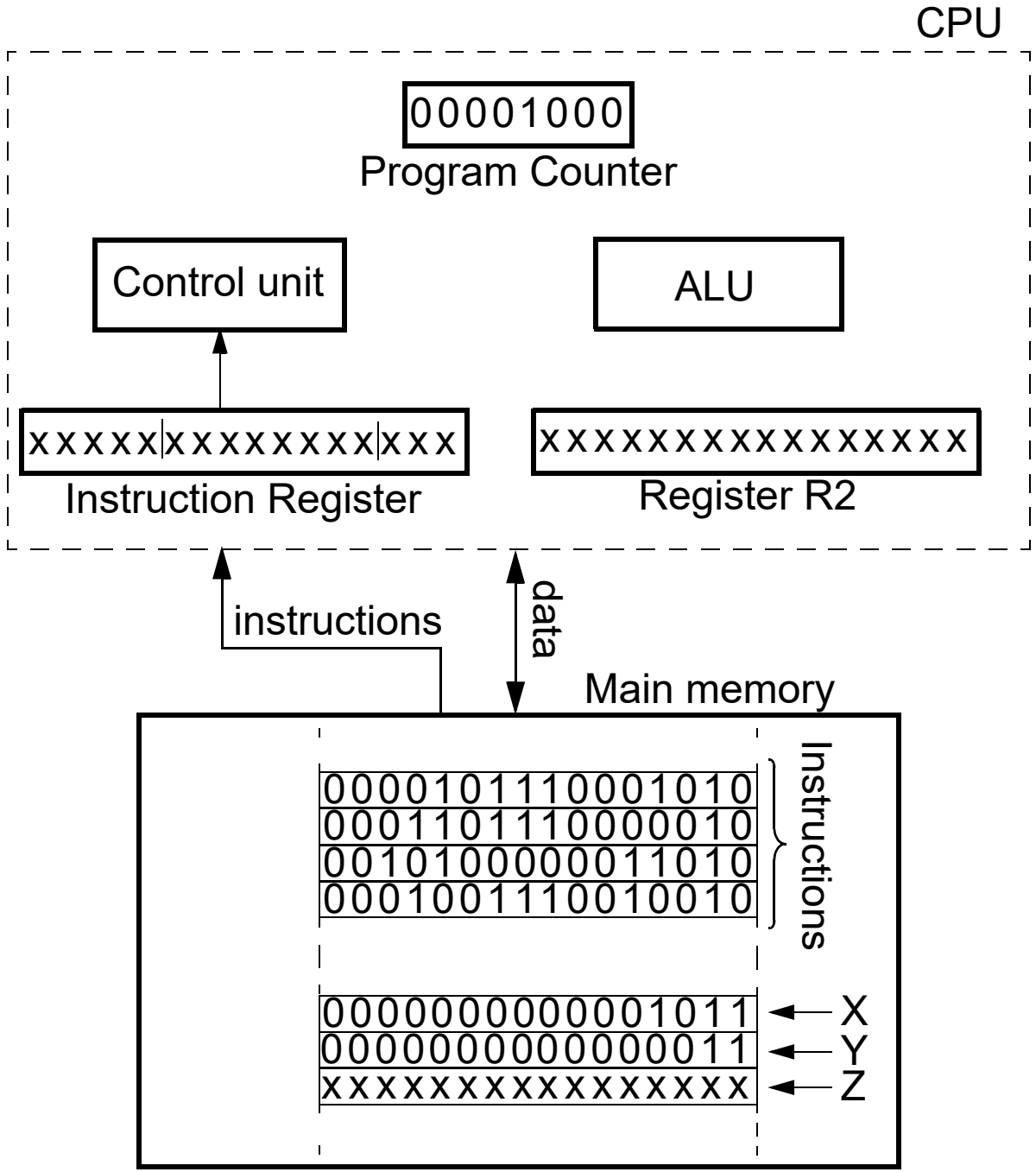
Instruction Execution

Let's imagine you write in a program the following instruction: Z := (Y + X) * 3;



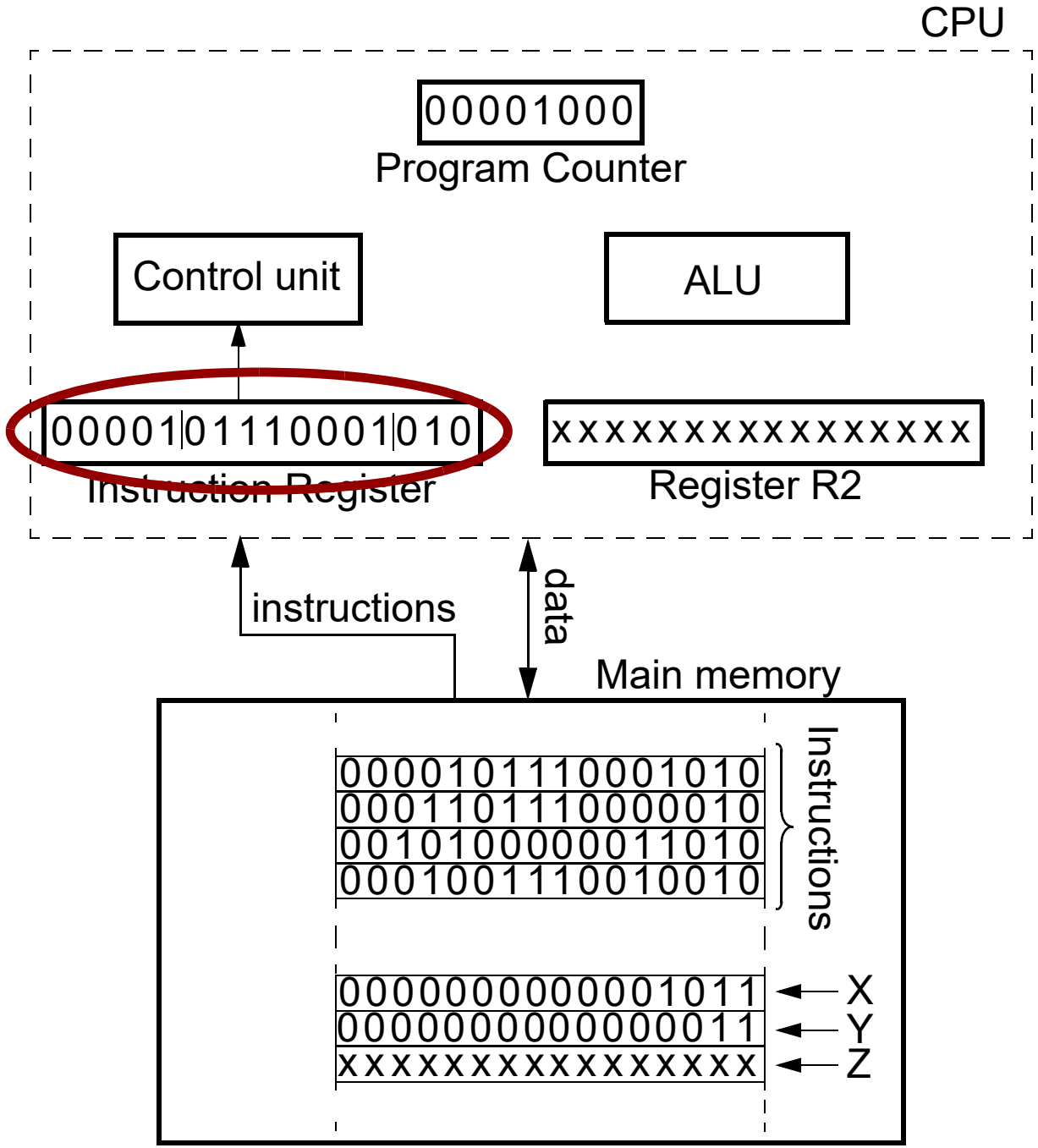
Let's Follow the Instruction Execution

Before the first instruction



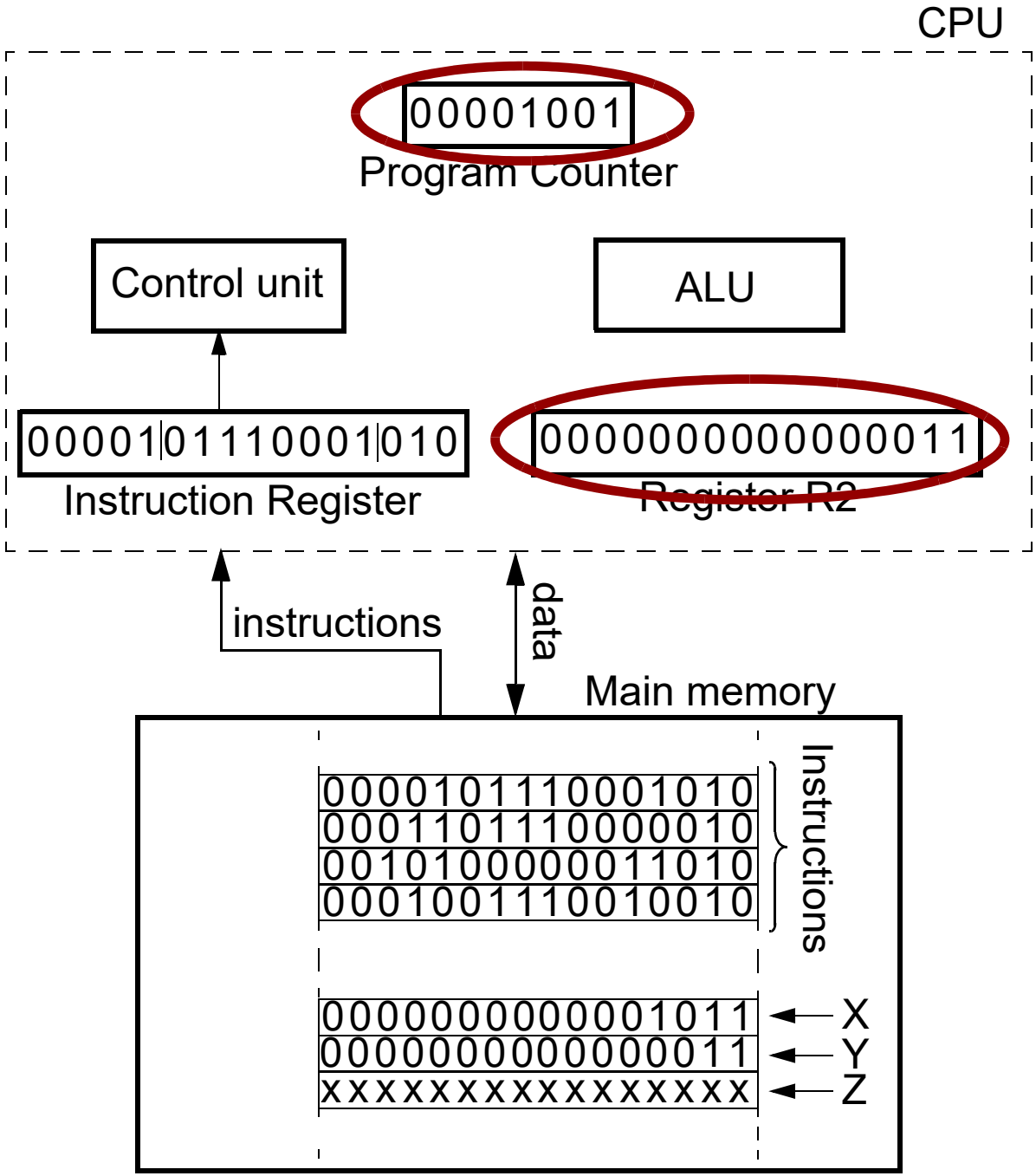
Let's Follow the Instruction Execution

Now the first instruction is fetched



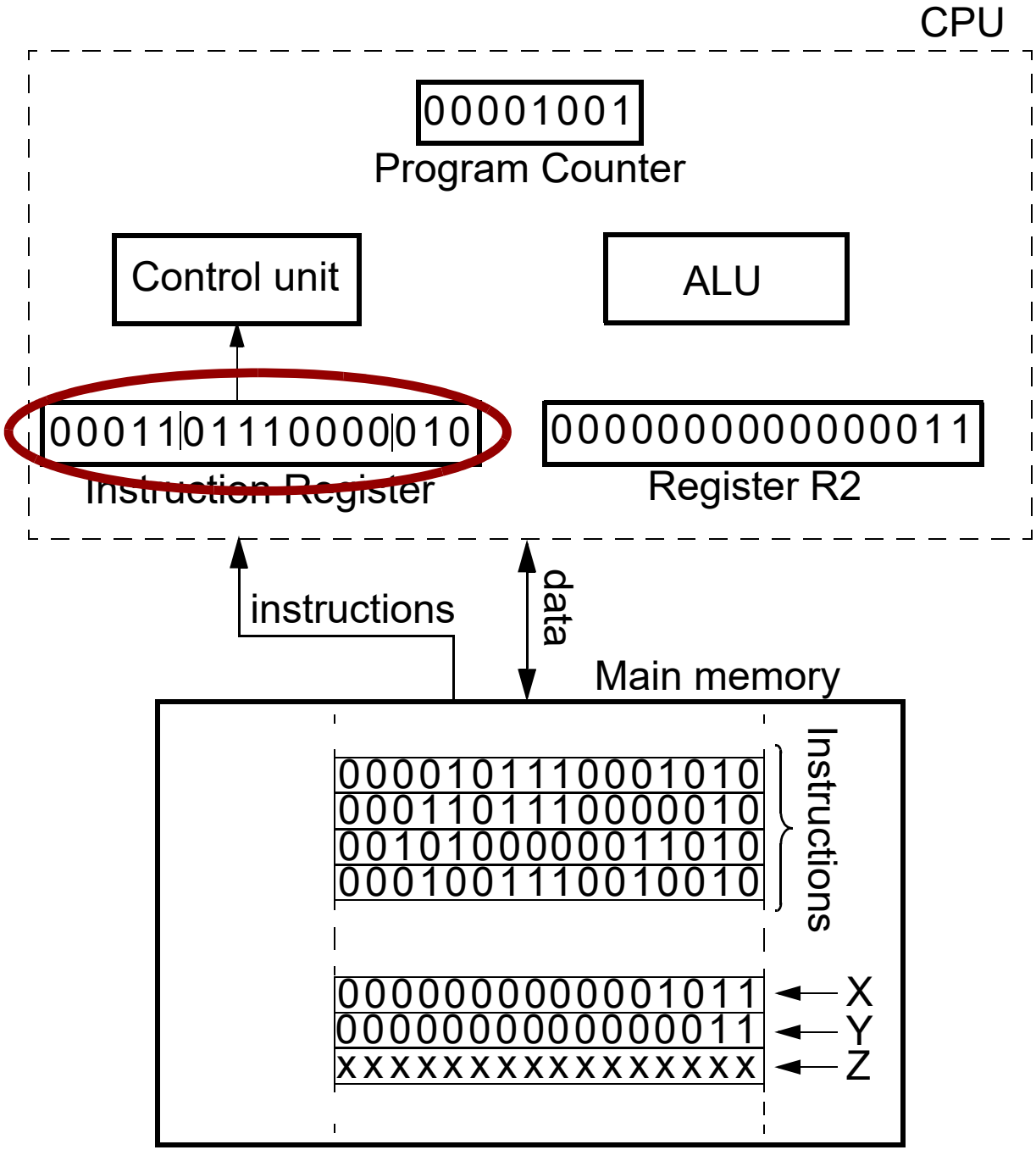
Let's Follow the Instruction Execution

After the first instruction



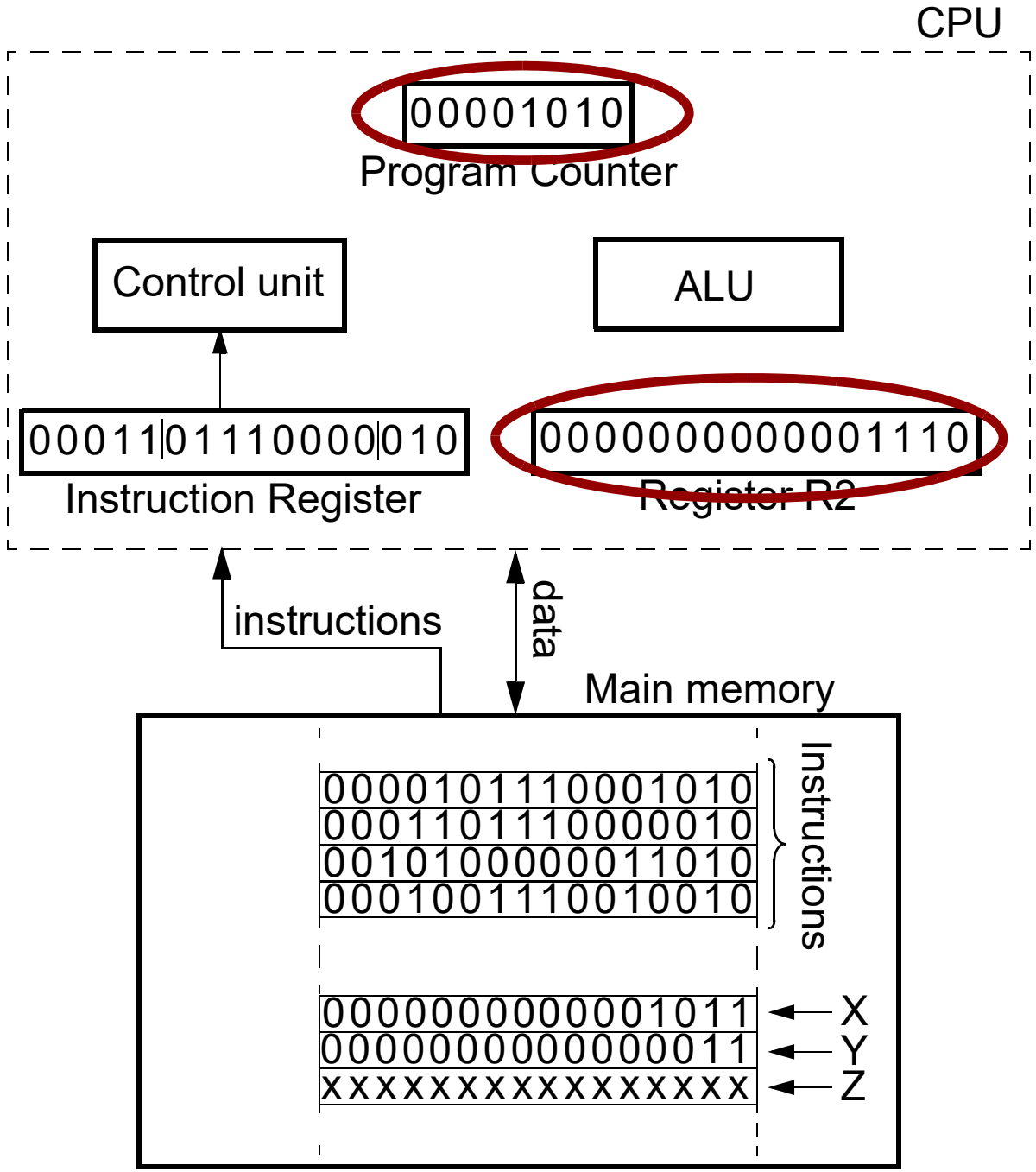
Let's Follow the Instruction Execution

Now the second instruction is fetched



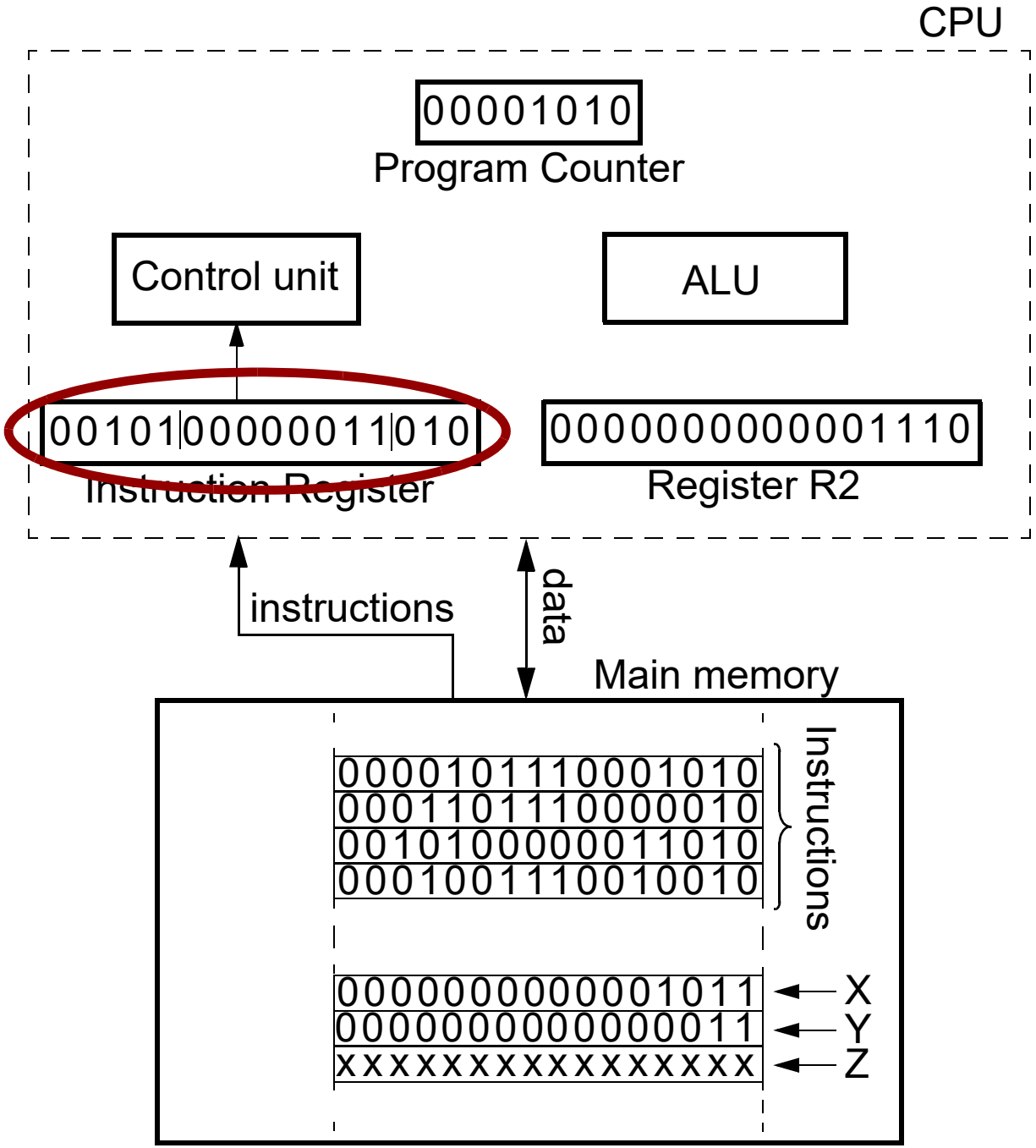
Let's Follow the Instruction Execution

After the sec-ond instruction



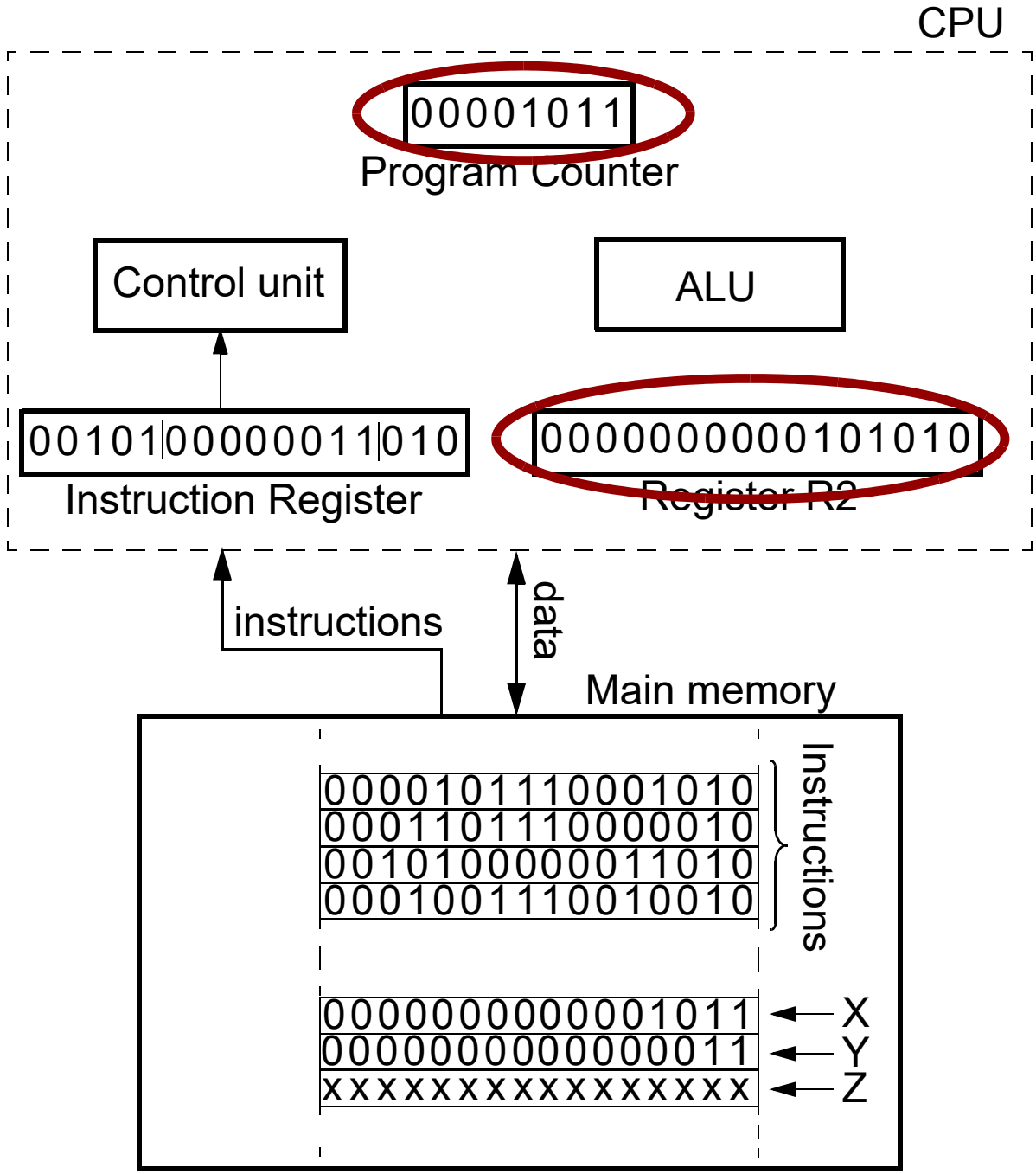
Let's Follow the Instruction Execution

Now the third instruction is fetched



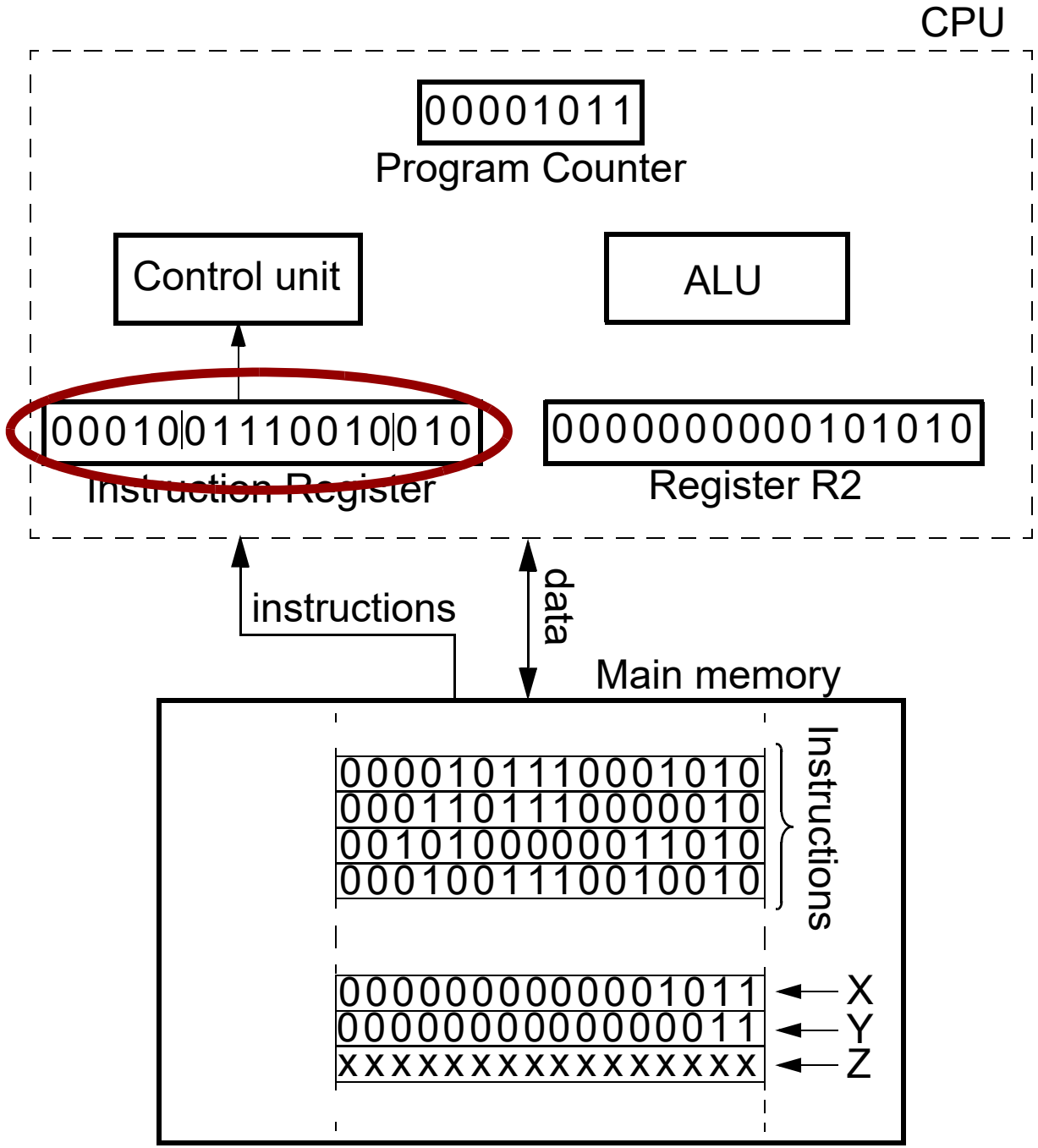
Let's Follow the Instruction Execution

After the third instruction



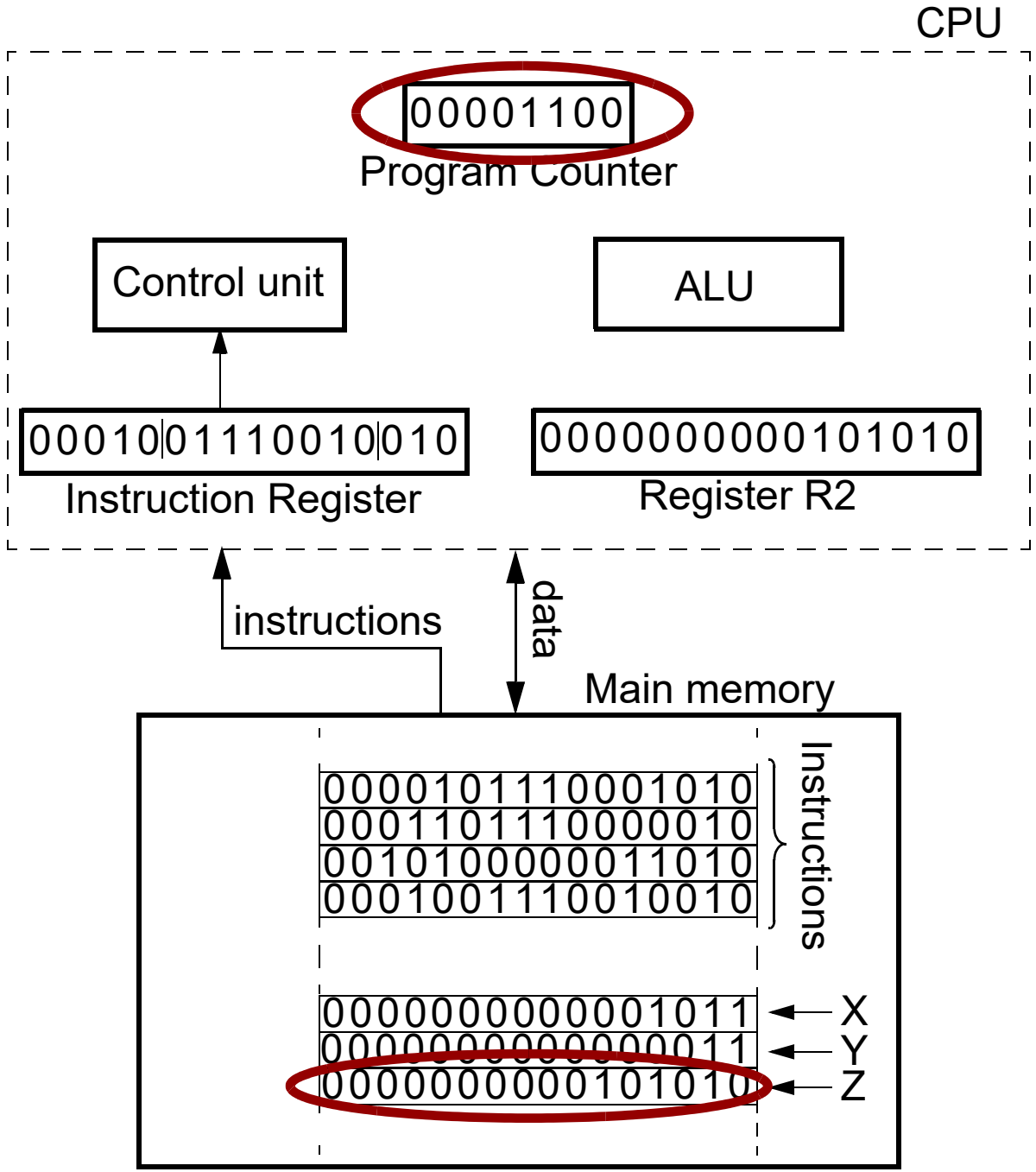
Let's Follow the Instruction Execution

Now the fourth instruction is fetched



Let's Follow the Instruction Execution

After the fourth and last instruction



Compilers

High Level Language (e.g. C, C++, Java)

We have written in our program: Z := (Y + X) * 3;

Machine instructions for the particular processor that runs the program.

What the computer executes is:

```
0000101110001010
0001101110000010
0010100000011010
0001001110010010
```

Compilers

We have written in our program: Z := (Y + X) * 3;

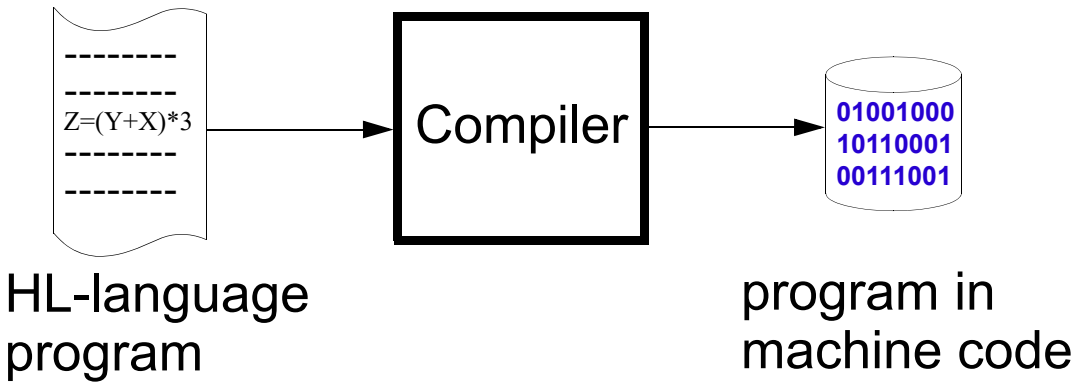
High Level Language (e.g. C, C++, Java)

What the computer executes is:

```
0000101110001010
0001101110000010
0010100000011010
0001001110010010
```

Machine instructions for the particular processor that runs the program.

Who brings us from our program to the machine instructions?



- A *compiler* is a program that translates programs written in a high level language into machine code to be executed on a certain processor.

The Machine Cycle

From the previous example you have seen that many things have to be done to execute a simple machine instruction;

- Fetch instruction
- Decode instruction
- Execute instruction

The Machine Cycle

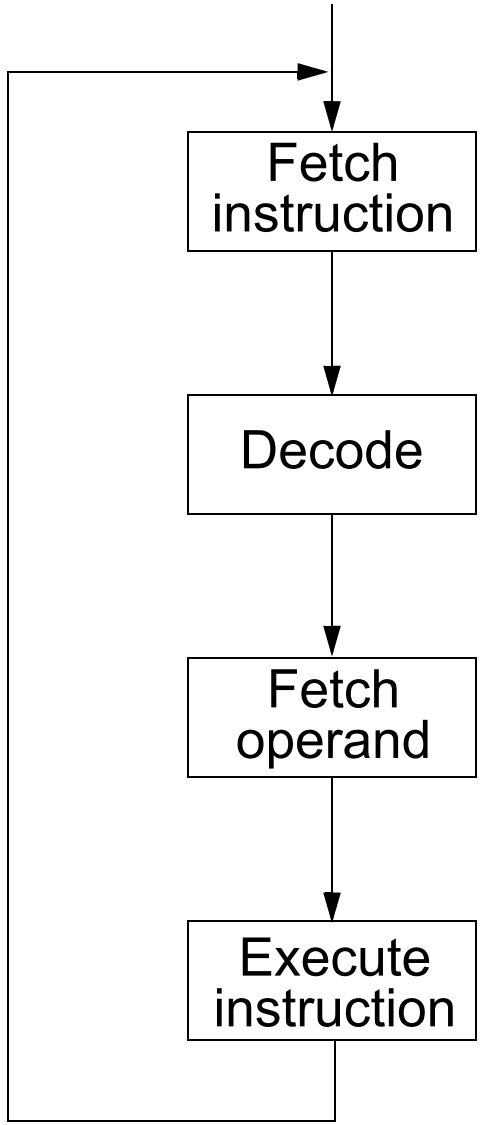
From the previous example you have seen that many things have to be done to execute a simple machine instruction;

- Fetch instruction
- Decode instruction
- Execute instruction
  - Fetch operand(s)
  - Execute instruction

The Machine Cycle

From the previous example you have seen that many things have to be done to execute a simple machine instruction;

- Fetch instruction
- Decode instruction
- Execute instruction
  - Fetch operand(s)
  - Execute instruction



Machine Cycle

Each instruction is performed as a sequence of steps; the steps corresponding to the execution of one instruction are referred together as a *machine cycle*.

The number and nature of steps in the machine cycle differ from processor to processor.

The Quest for Speed

Running faster (more instructions per time unit) has been a permanent goal of computer designers.

<u>Two main factors contribute to high performance of modern processors</u>:

1. **Fast circuit technology: smaller and faster switching transistors, allowing the processor to run at higher frequency.**
2. **Architectural features such as:**
   - **Smart memory hierarchies**
   - **Pipelining**
   - **Superscalar architectures**

   } **Several instructions are executed in parallel.**

Memory System

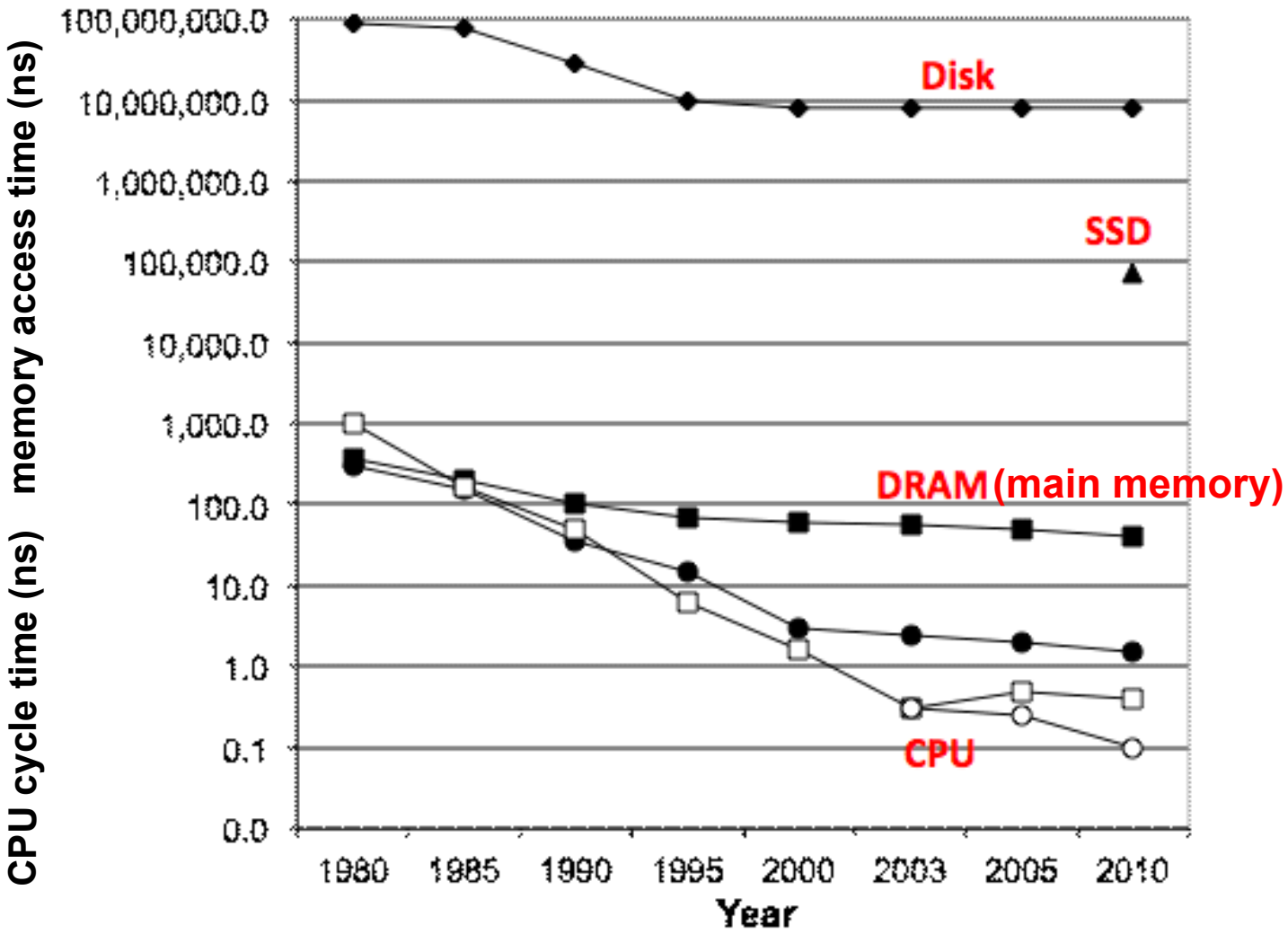
One of the most crucial aspects in designing efficient computer architectures is the memory system.

- **What do we need?**

  We need memory to fit very large programs and to work at a speed comparable to that of the microprocessors.

- **Main problem:**
  - **Processors are working at a high clock rate and they need large memories;**
  - **Memories are much slower than microprocessors; but for executing a single instruction you need several memory accesses (fetch the instruction and operands); *it doesn't help that the processor is fast, if the memory is orders of magnitude slower*.**

The CPU-Memory Gap

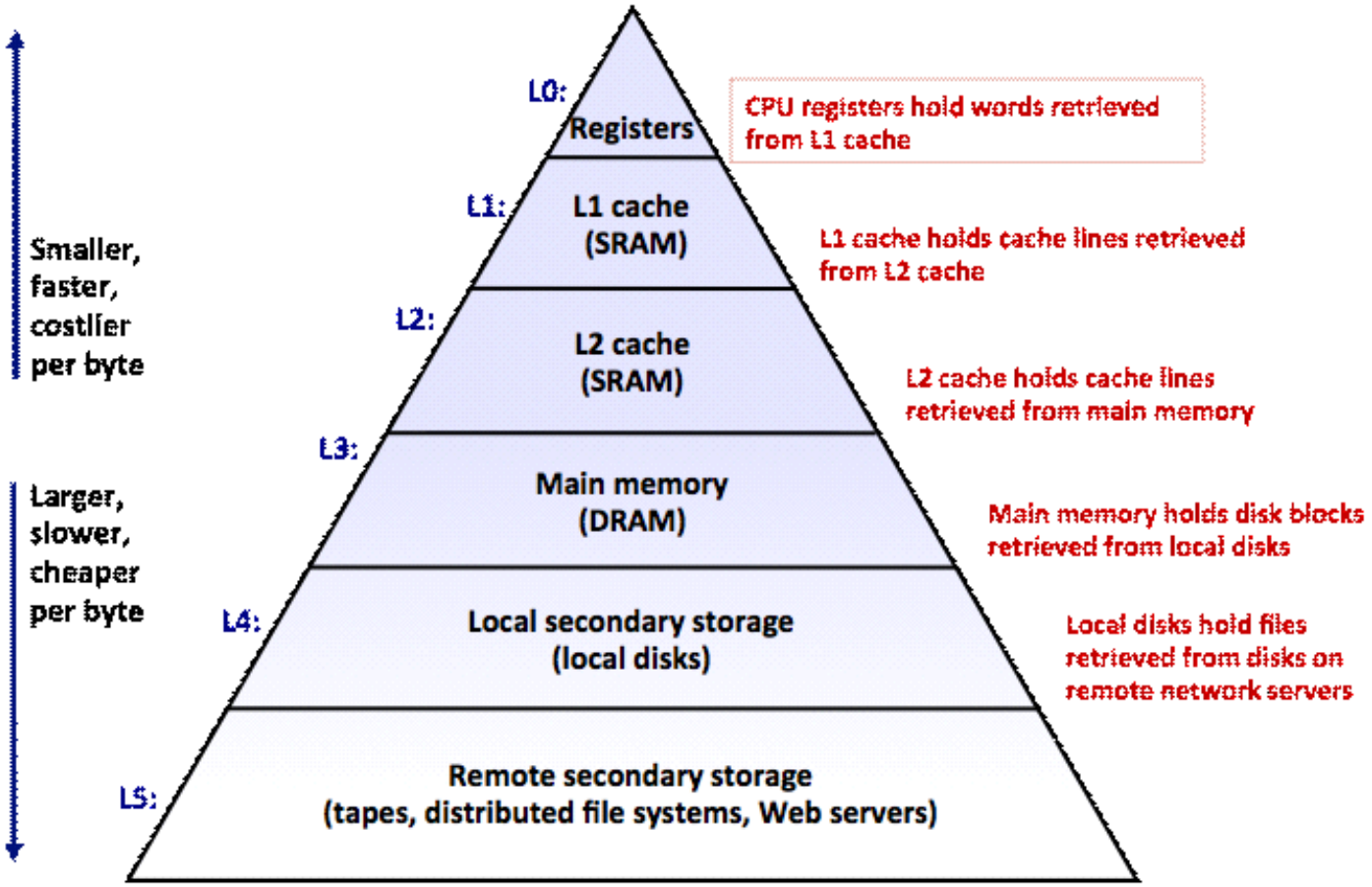


Memory Hierarchies

- Fast memories are more expensive per byte and cannot be very large (main memory is much smaller than SSD or Disk)
- It is possible to build memory structures that are as fast as the CPU, but they are very expensive and small.

Memory Hierarchies

- Fast memories are more expensive per byte and cannot be very large (main memory is much smaller than SSD or Disk)
- It is possible to build memory structures that are as fast as the CPU, but they are very expensive and small.



Memory Hierarchies

The good news:

- **It is possible to build a composite memory system which combines *small, fast memories* (from the top of the hierarchy) and *large slow memories* (from the middle and bottom of the hierarchy) and which behaves (most of the time) like a large fast memory.**

How can this work?

Memory Hierarchies

The good news:

- It is possible to build a composite memory system which combines *small, fast memories* (from the top of the hierarchy) and *large slow memories* (from the middle and bottom of the hierarchy) and which behaves (most of the time) like a large fast memory.

How can this work?

The answer is: Locality

The Principle of Locality

- During execution of a program, memory references by the processor, for both instructions and data, tend to cluster: once an area of the program is entered, there are repeated references to a small set of instructions (loop, subroutine) and data (components of a data structure, local variables or parameters on the stack).

  - Temporal locality (locality in time): If an item is referenced, it will tend to be referenced again soon.
  - Spacial locality (locality in space): If an item is referenced, items whose addresses are close by will tend to be referenced soon.

Cache Memory

A cache memory is a small, very fast memory that retains copies of recently used information (instructions and data). It operates transparently to the programmer, automatically deciding which values to keep and which to overwrite.

- Due to the property of locality, most of the time, the instruction or data required by the CPU will be available in the top cache. If not, it will be loaded from the lower level cache; once loaded the information will be written into the top level cache and replace some existing one, in order to make space for the new information.
- Which information is replaced when new one has to be written?
  - Some information is overwritten that has, for a long time, not been used by the CPU (and, thus, is less likely to be needed in the future)
- The above procedure is repeated at each level of the hierarchy.

The Quest for Speed

Running faster (more instructions per time unit) has been a permanent goal of computer designers.

<u>Two main factors contribute to high performance of modern processors</u>:

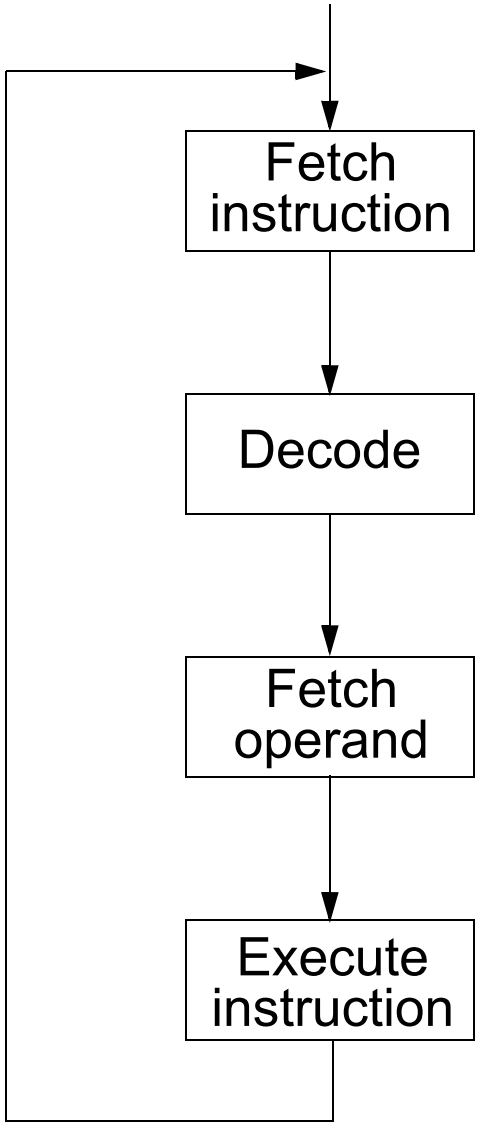
3. **Fast circuit technology: smaller and faster switching transistors, allowing the processor to run at higher frequency.**

4. **Architectural features such as:**
   - **Smart memory hierarchies**
   - **Pipelining**
   - **Superscalar architectures**

   Several instructions are executed in parallel.

Pipelining

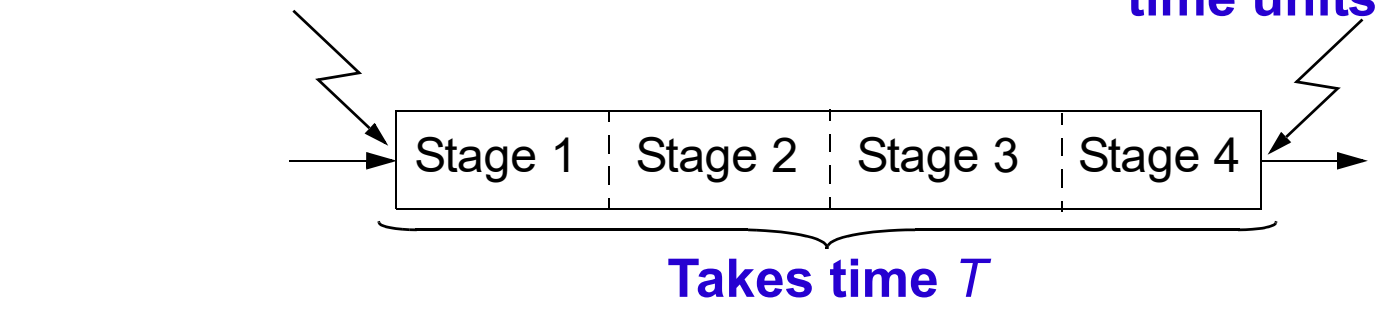
We remember the machine cycle

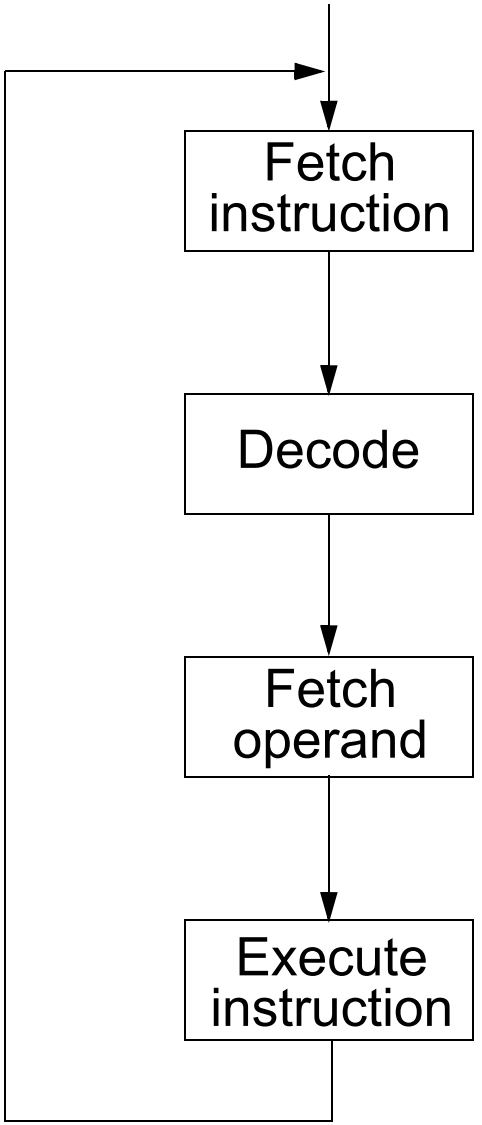


Pipelining

We remember the machine cycle

- Each step in the machine cycle is performed by a separate piece of hardware:

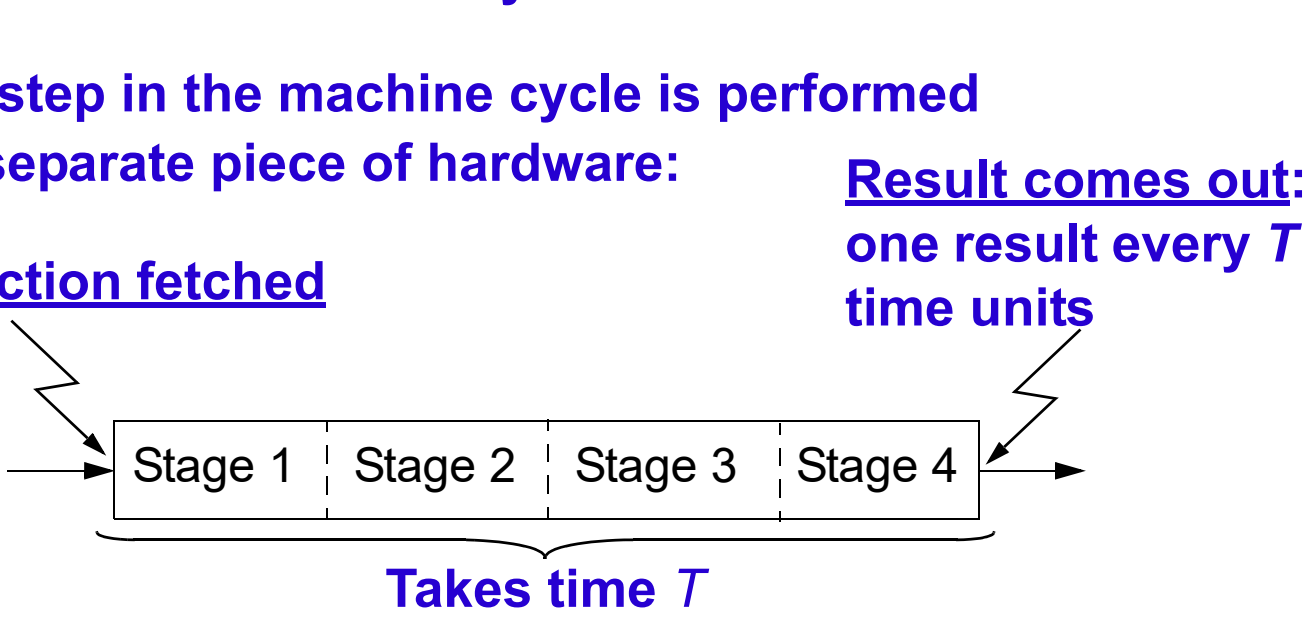




Pipelining

We remember the machine cycle

- Each step in the machine cycle is performed by a separate piece of hardware:



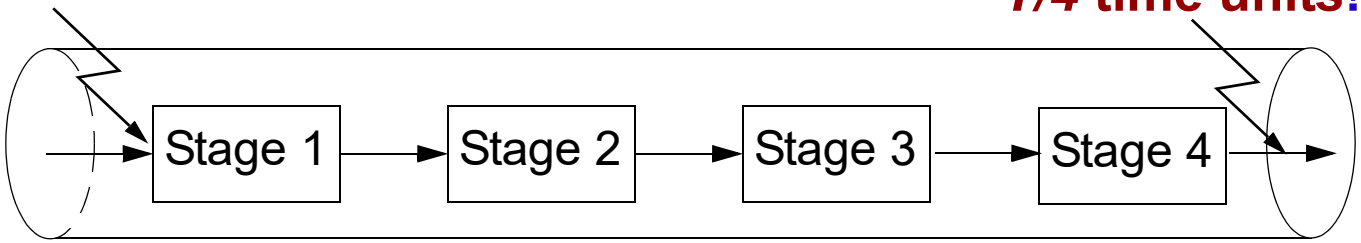
New instruction fetched

Result comes out: one result every *T* time units

- The CPU works like a pipeline (assembly line): Once a stage finished with an instruction, it hands it over to the next stage and takes over a new instruction.

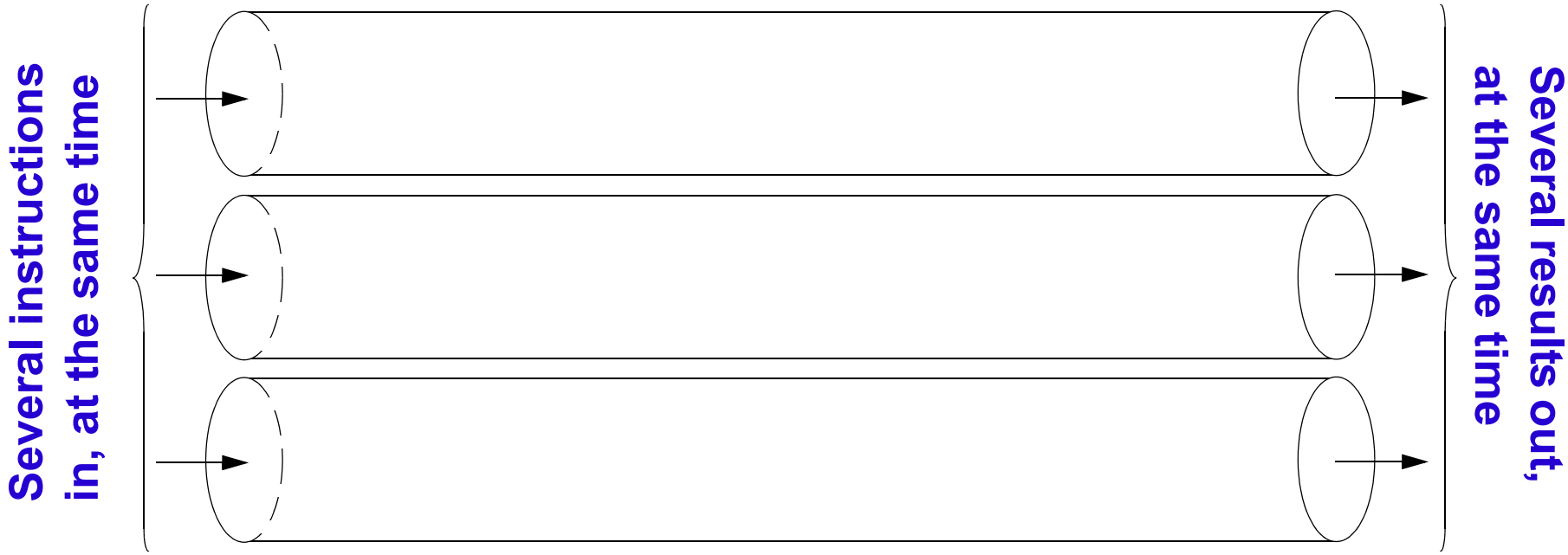
New instruction fetched

Result comes out: one result every *T/4* time units!!!



Superscalar Architectures

- You can imagine a superscalar processor as composed of several pipelines running together.
  - As opposed to simple pipelined computers, superscalars fetch several instructions and produce several results simultaneously



The Quest for Speed

Running faster (more instructions per time unit) has been a permanent goal of computer designers.

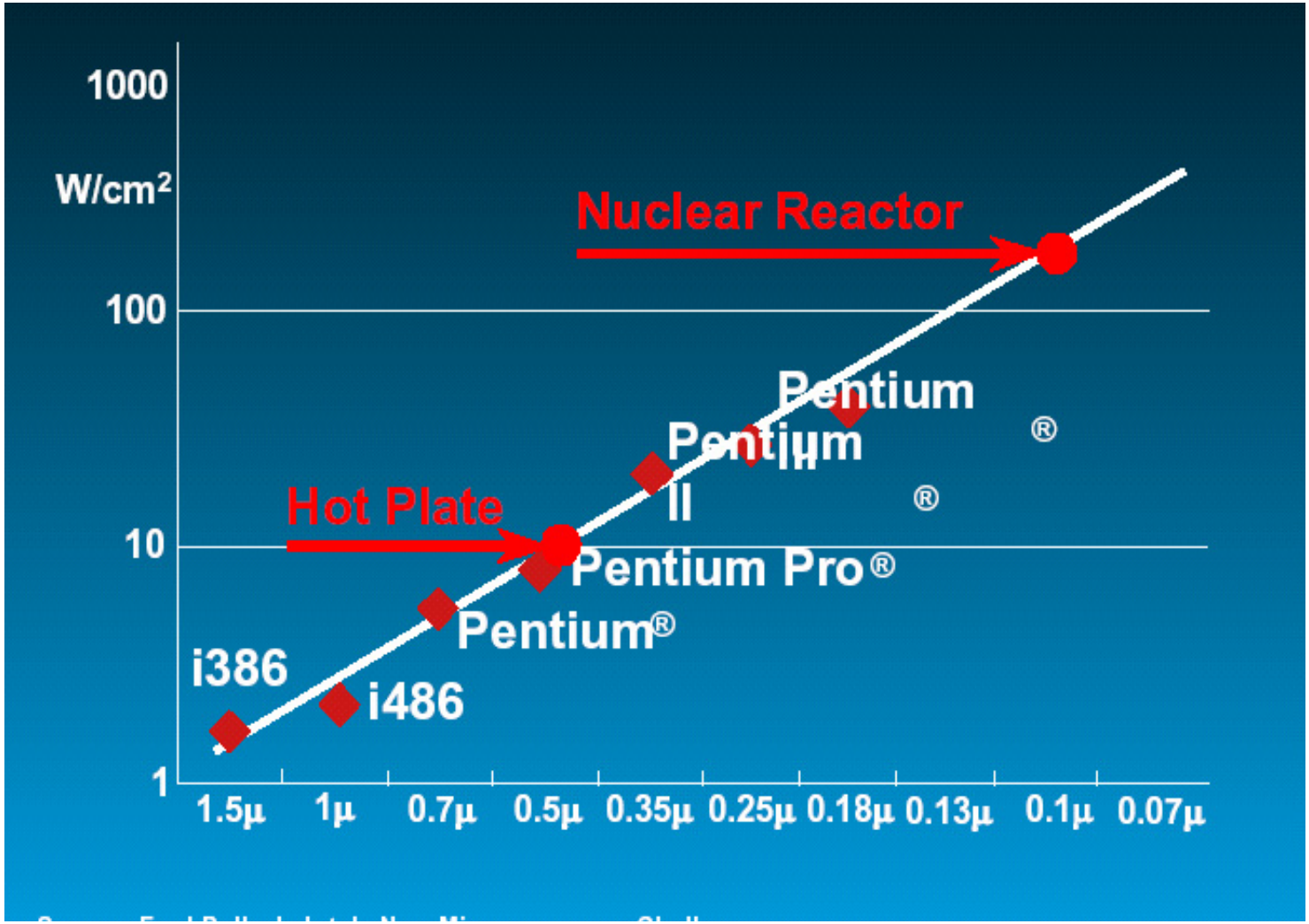
Two main factors contribute to high performance of modern processors:

5. Fast circuit technology: smaller and faster switching transistors, allowing the processor to run at higher frequency.
6. Architectural features such as:
   - Smart memory hierarchies
   - Pipelining
   - Superscalar architectures

That one has been a primary source of performance improvement all over the years. Processors running at higher and higher frequencies allowed for a continuous increase in speed. That doesn't work any more!!!

The Power Wall

We have reached the limit due to the temperature produced by the high power consumption! Further increase of the frequency is impossible!



This is the main challenge today!

New ways have to be explored in order to deliver performance!

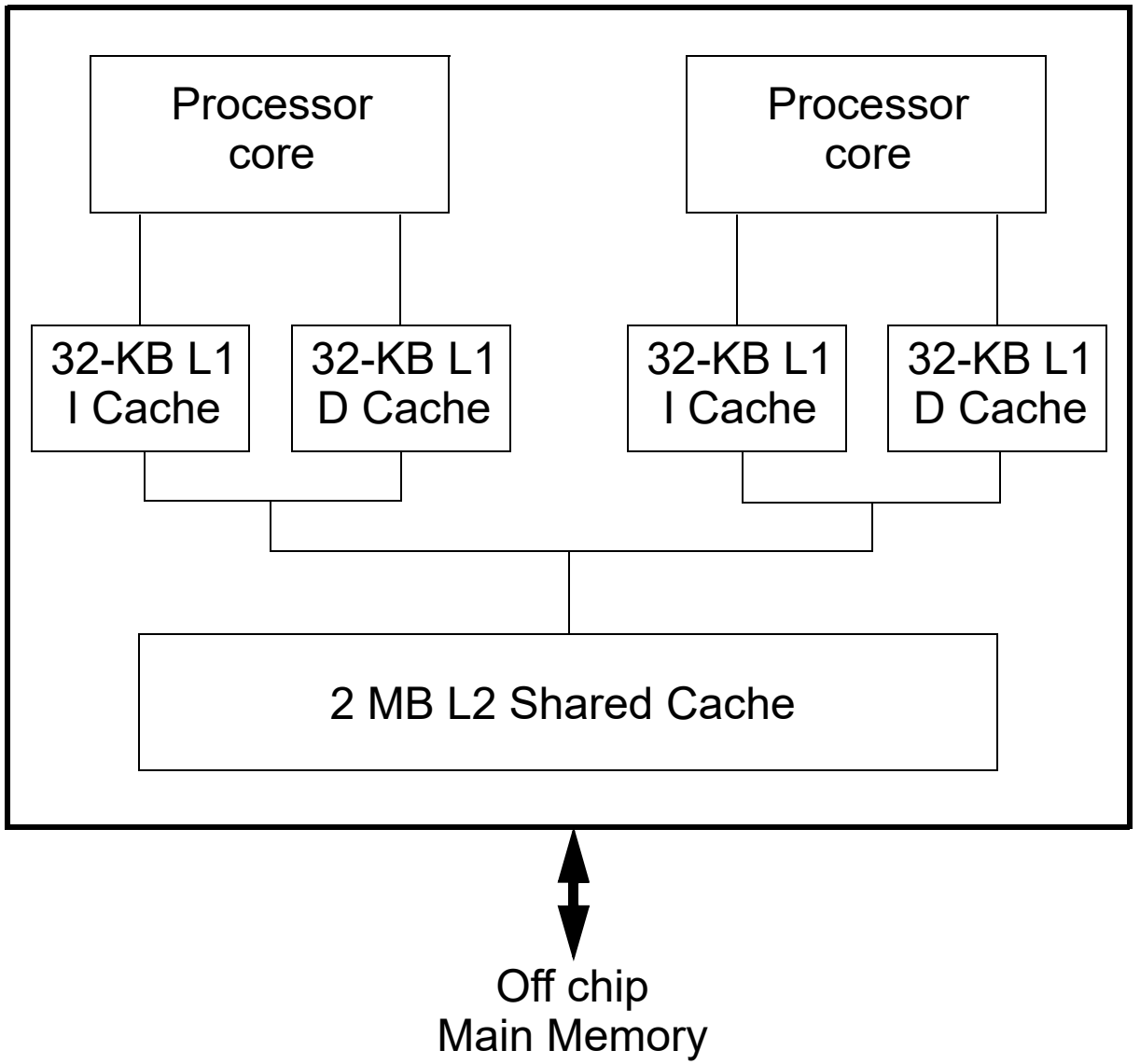
Multicore Chips

- Multicore chips: Several processors on the same chip.

- This is the only way to increase chip performance without excessive increase in power consumption:
  - Instead of increasing processor frequency, use several processors and run them in parallel, each at lower frequency.

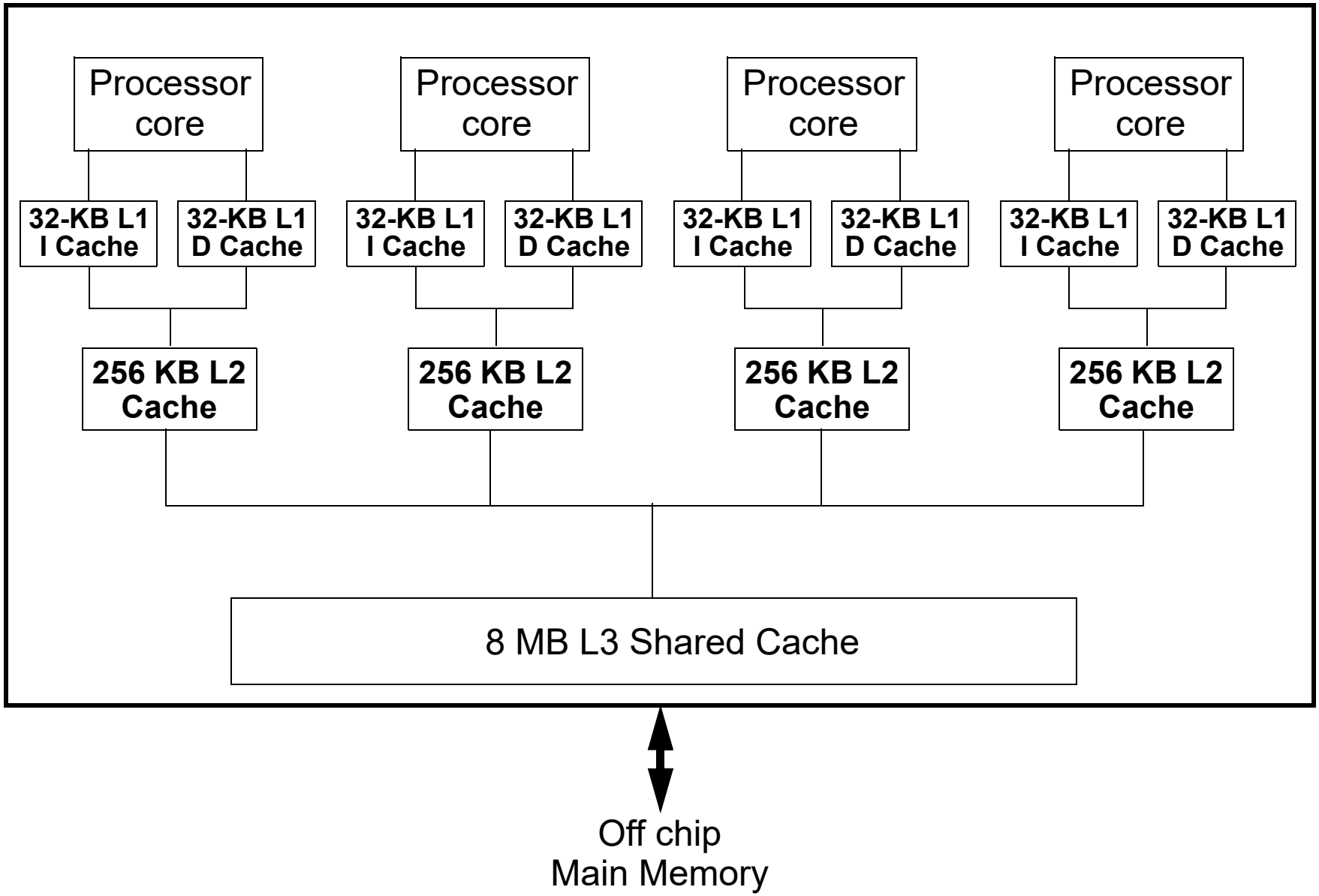
Intel Core Duo

- **Composed of two Pentium M superscalar processors.**



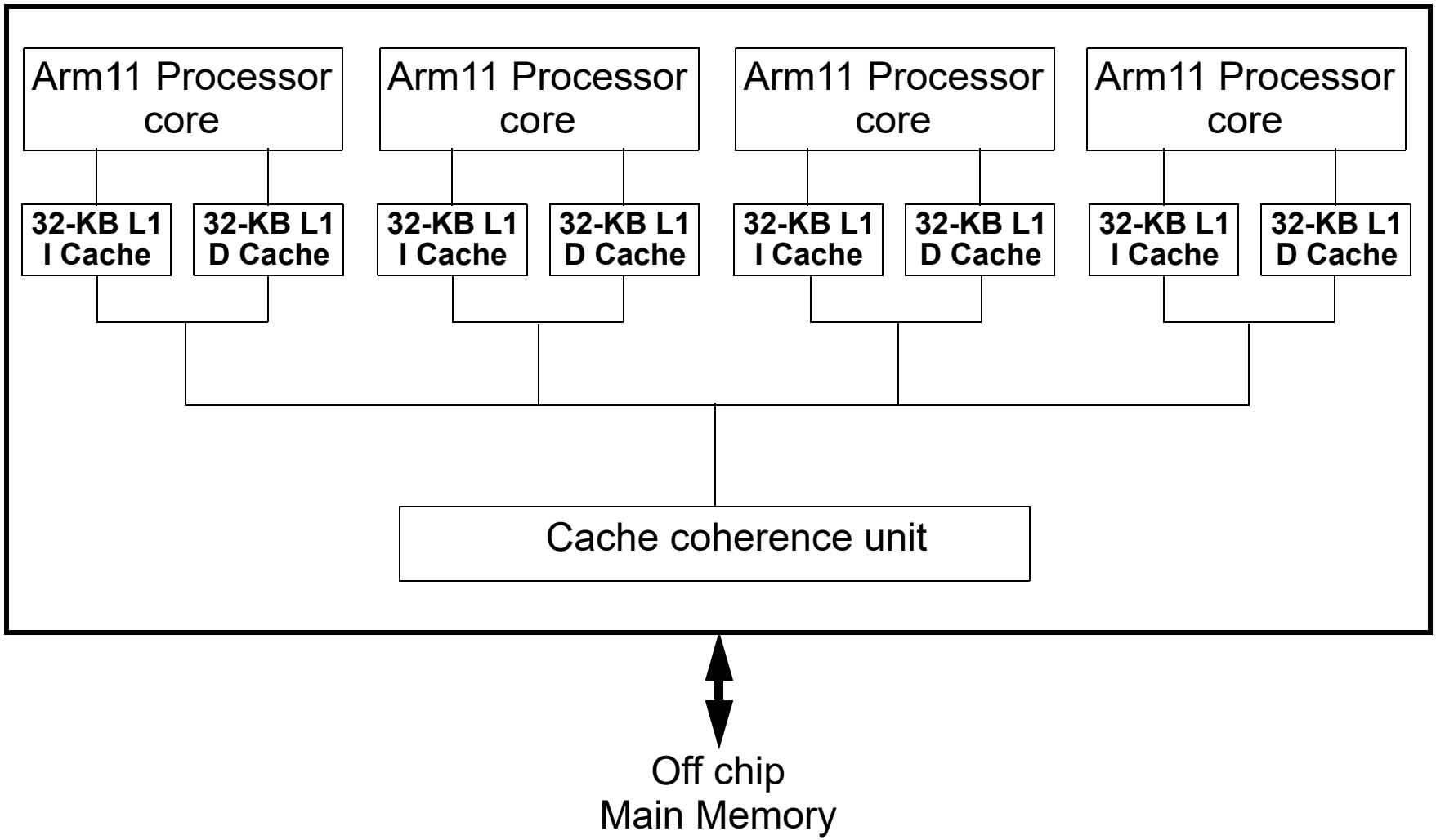
Intel Core i7

- **Composed of four x86 SMT (simultaneous multithreading) processors.**



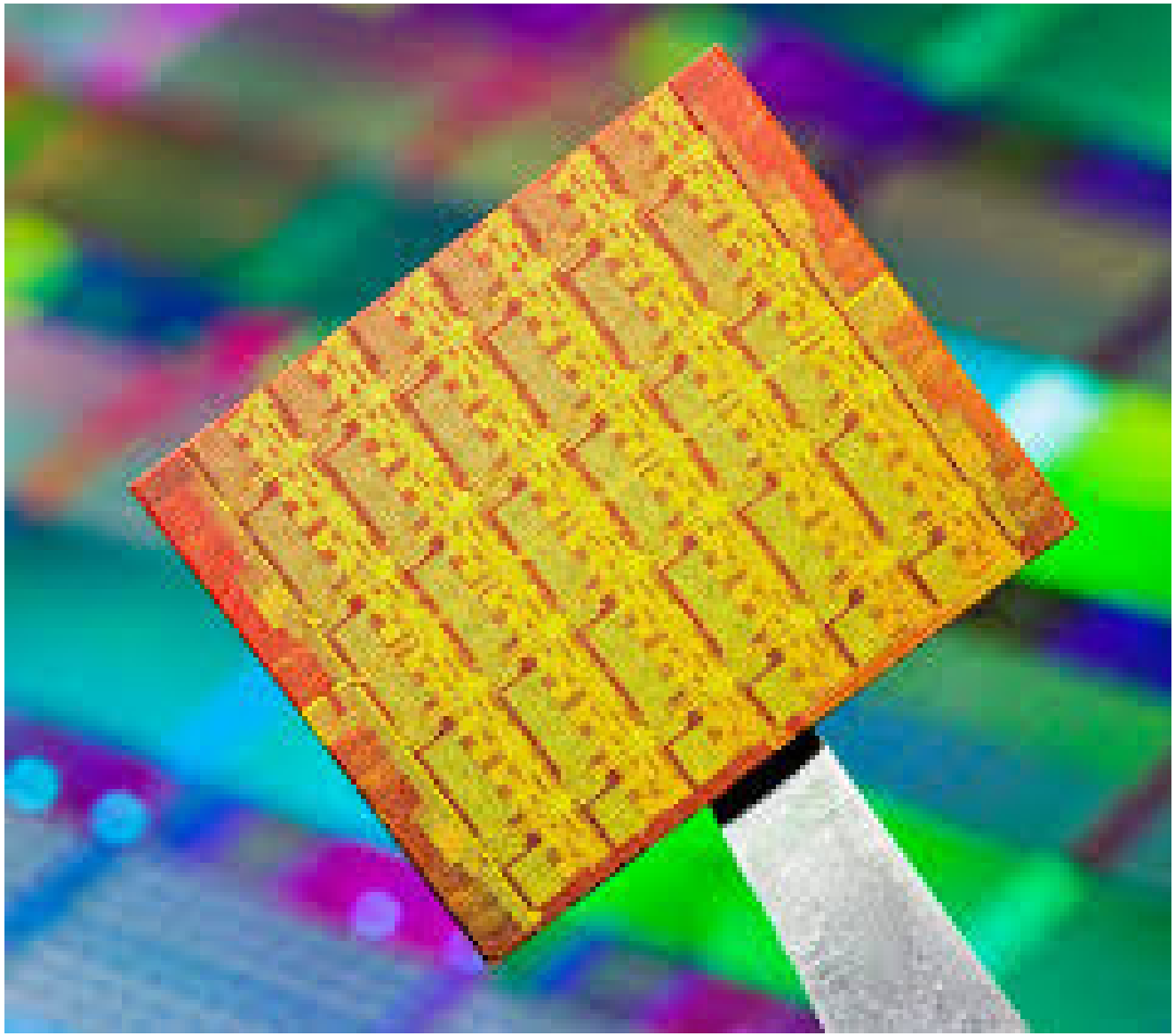
ARM11 MPcore

- **Composed of four ARM11 processor cores.**

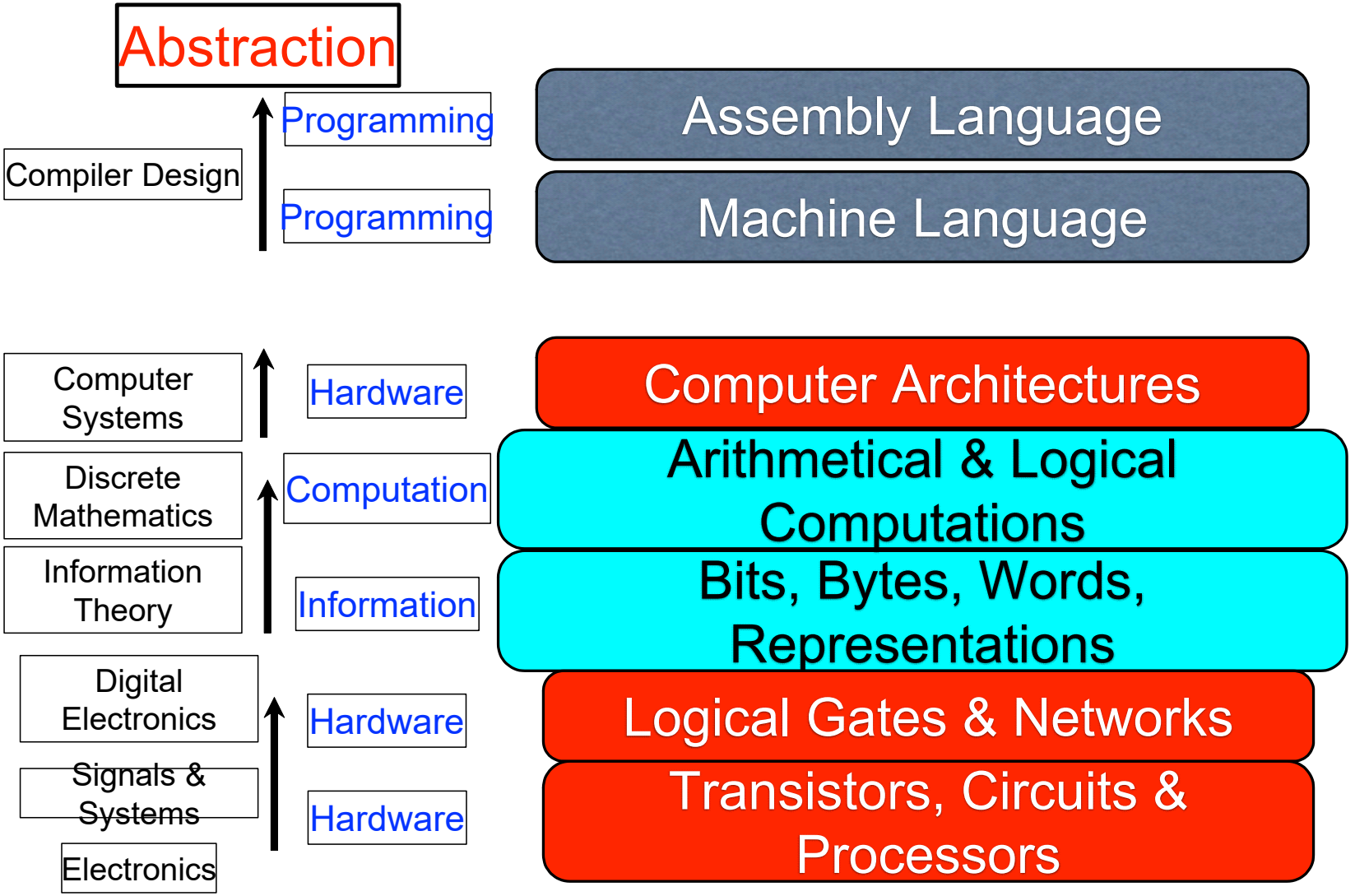


Intel's Single-Chip Cloud Computer (SCC)

- Composed of 48 P54C Pentium cores



Where are We


Abstraction
Programming
Compiler Design
Programming
Assembly Language
Machine Language
Computer Systems
Hardware
Discrete Mathematics
Computation
Information Theory
Information
Digital Electronics
Hardware
Signals & Systems
Hardware
Electronics
Computer Architectures
Arithmetical & Logical Computations
Bits, Bytes, Words, Representations
Logical Gates & Networks
Transistors, Circuits & Processors