



# COMPUTER ARCHITECTURE

## 3 Basic principles of computing



## 3 Basic principles of computing - content

- von Neumann computer model
  - von Neumann computer model
  - Operation von Neumann computer
- Flynn's classification
- The main memory in von Neumann computer
  - Memory word (location)
  - memory address
  - address space
  - The content of the memory word
  - Princeton and Harvard memory architecture
  - Access to memory
- Amdahl's law
- Languages, levels and virtual computers
  - The computer as a series of virtual computers
  - Transition from the language L2 into the language L1
  - Hardware and software of the computer
- Case: Execution of the program on the computer



## 3 Basic principles of computing - content

- von Neumann computer model
- Flynn's classification
- The main memory in von Neumann computer
- Amdahl's law
- Languages, levels and virtual computers
- Case: Execution of the program on the computer

### Basic principles of computing – goals :

- Basic understanding of computer operation
  - Von Neumann model and extensions (parallel)
- Computer system levels (HW <-> SW)
- Understanding of program execution



## 3.1 Von Neumann computer model

- Computers are built on the basis of computing model, known as „von Neumann“ model (John von Neumann 1945)

- von Neumann's:

- Computing model,
- Computer model,
- Computer,
- Architecture.

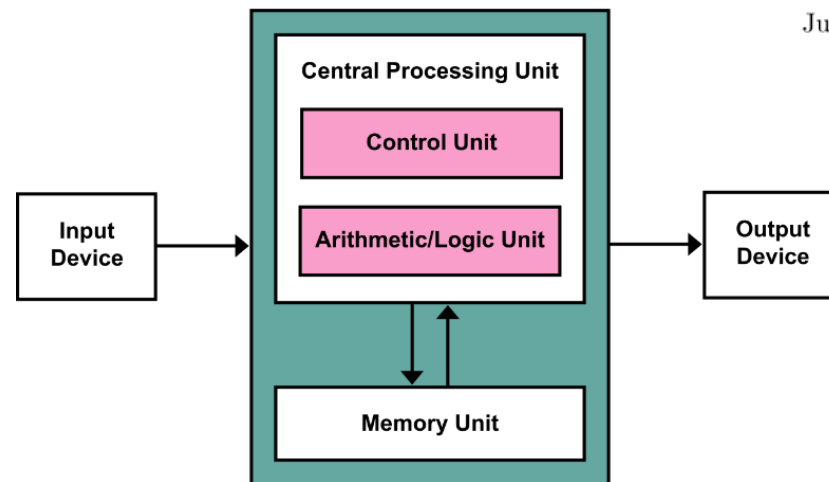
First Draft of a Report  
on the EDVAC

by

John von Neumann

Moore School of Electrical Engineering  
University of Pennsylvania

June 30, 1945



## 3.1 Von Neumann computer model

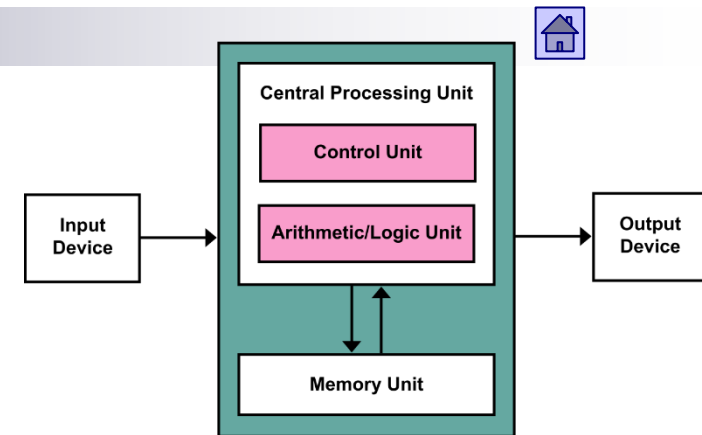
- It consists of three basic parts:

- CPU (Central Processing Unit)
- Main Memory
- Input-Output (I/O) system

- It is the machine with a **stored program** in the main memory. Instructions in the program specify what the machine will do.

- **The program leads the machine operation** - program determines how the machine will work.

- CPU takes instructions from the main memory and execute them one after the other.



First Draft of a Report  
on the EDVAC

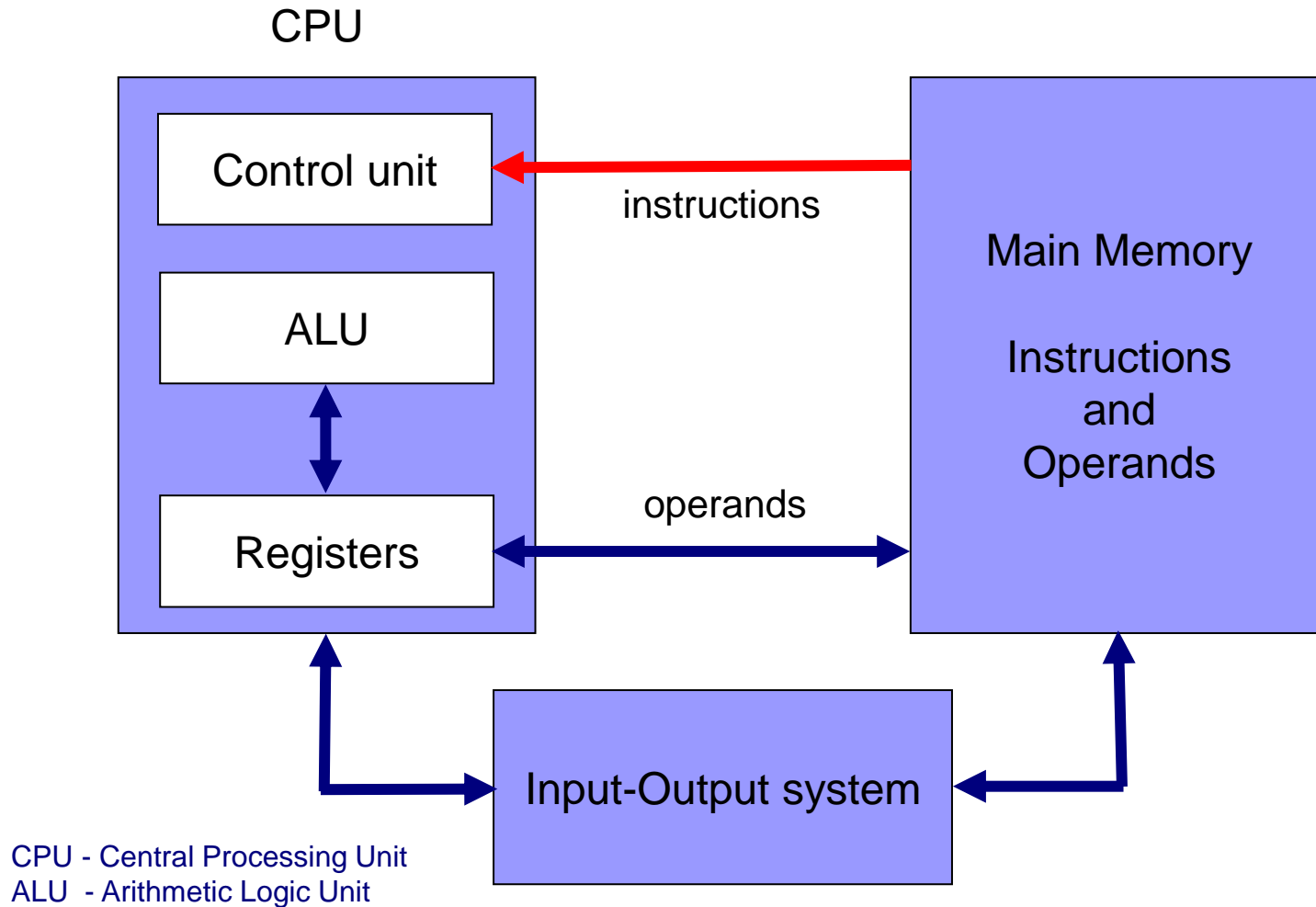
by

John von Neumann  
Moore School of Electrical Engineering  
University of Pennsylvania

June 30, 1945



## Von Neumann computer model





- **CPU** reads instructions from the main memory and executes them. In today's computers, in addition to the main, there are even more processors, thus we denote main processor as the **Central Processing Unit**. It consists of three parts:
  - Control Unit – fetches the instructions & operands, and activates operations set by instructions.
  - ALU – performs arithmetic operations (addition, ...) and logic operations (AND, ...).
  - REGISTERS - a number of connected memory cells which serve to store values.
    - Program Inaccessible registers - necessary for the operation of the CPU.
    - Program Accessible registers (architectural registers) for storing operands - a small and fast memory in the CPU.



- **Main Memory** is made up of memory words (locations). Each memory word has its own unique address.
  - It stores instructions and operands.
  - Identification „main“ again serves to distinguish it from other memory devices in today's computers (caches, virtual memory).
  
- **I/O system** serves for the transfer of information to the outside world or from the outside world. Information from the CPU and main memory is stored in a format that is not accessible to the outside world.
  - An integral part of the I/O system are the input-output devices, which transform the information into another form which is suitable for the user or represent an auxiliary (secondary) memory.





# Operation of Von Neumann computer

- Its operation is completely controlled by instructions (machine instructions), that are read by the CPU from the main memory in a order one after the other.
  
- Machine instructions are stored in the memory one after the other by increasing addresses.
  
- There has to be a deterministic procedure how to start: First instruction is usually read from certain address in memory, after the computer is turned on or pressing the RESET button.
  - The easiest way: the first or last memory location - the lowest or the highest address in the memory.



## For each instruction we distinguish two steps

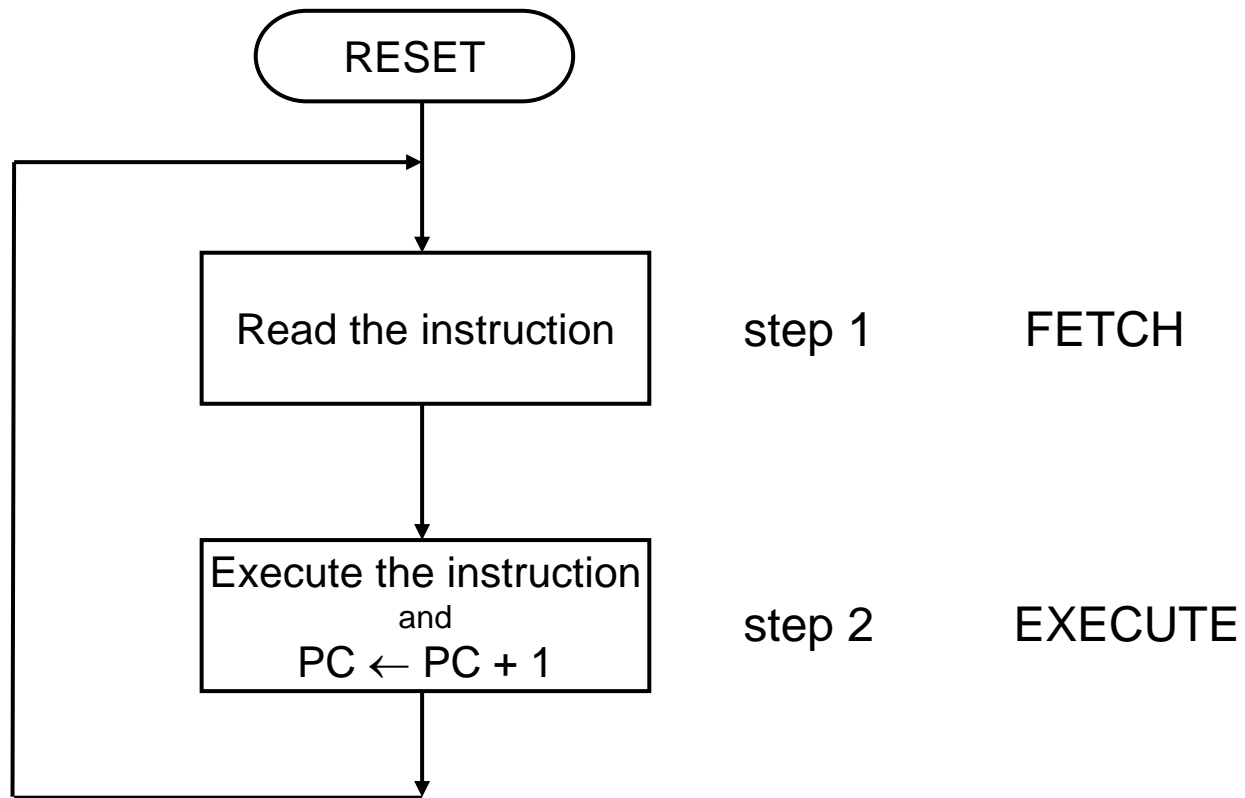
- **1. step:** Read instruction from the memory (FETCH)
  - instruction fetch cycle
  - The CPU includes the special register - the program counter (PC - Program counter) that always contains a memory address of the next instruction to be read and executed.
  
- **2. Step:** Execution of the fetched instruction (EXECUTE)
  - Execute cycle



- Each instruction contains two types of information:
  - information about the operation to be executed,
  - information on the operands, over which the operation is executed.
  
- CPU executes the operation, and ensure that the PC includes the address of the next instruction by increasing the content of the PC by 1.
  
- **Rule:** instructions are stored in memory by increasing addresses so  $PC \leftarrow PC + 1$ . This rule is the result of an agreement and specifies the order in which the instructions are usually executed.



## Operation of Von Neumann computer





Upon completion of Step 2, the CPU starts again with first step. These two steps are repeated until the computer runs.

- **Exception 1:** Jump instructions, which can write in PC other address than  $PC+1$ .
  
- **Exception 2:** Interrupt or trap  
CPU after step 2 does not fetch next instruction by rule,  $PC \leftarrow PC + 1$ , but starts another program - Interrupt Service Program (ISP). Proper return to original program execution is needed.



- Sequential instruction execution is slow and represents a basic weakness of Von Neumann based computers.
- Extensions of the basic Von Neumann model are contained in the Flynn's classification.



## 3 Basic principles of computing - content

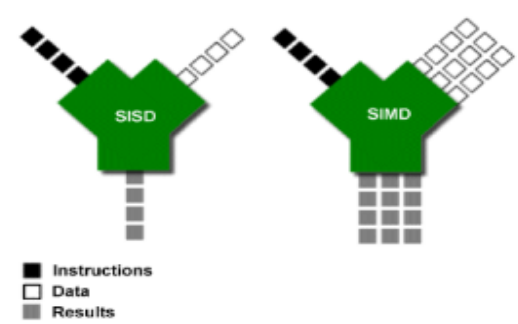
- von Neumann computer model
- Flynn's classification
- The main memory in von Neumann computer
- Amdahl's law
- Languages, levels and virtual computers
- Case: Execution of the program on the computer



## 3.2 Flynn's classification

- This classification of computers into four groups suggested M.J.Flynn in year 1966. The basic criteria in this classification are:
  - The number of instructions that are executed at the same time (instruction stream)
  - The number of operands that one instruction processes (data stream).
- According to these criteria every computer belongs to one of four classes:
- 1 SISD (Single Instruction Single Data)
  - classic Von Neumann computers without parallelism for instructions and operands
  - Intel Pentium 4





Source: ARS Technica

## ■ 2 SIMD (Single Instruction Multiple Data)

- The real vector computers (parallel computers, graphics processors)
- Instructions SSE (Streaming SIMD Extensions) for x86 architecture processors

## ■ 3 MISD (Multiple Instruction Single Data)

- Unusual architecture. More instructions on single operand - can be used where better robustness to errors is required .

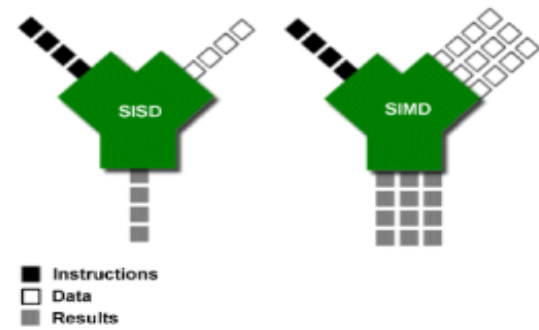
## ■ 4 MIMD (Multiple Instruction Multiple Data)

- Multiprocessor computers (parallel computers)
- This group could also include multicore superscalar computers (e.g. Intel Core i7) although they are generally attributed to SISD group because of limited number of cores.





- In MIMD computers, several instructions are executed simultaneously, each on its operands.
  
- MIMD computer is formed from more common Von Neumann computers - more CPUs that are interconnected.
  
- Multicore computers are commonly attributed to SISD group, although nowadays multi-core processors could be classified also in SIMD or MIMD groups.



Source: ARS Technica

## Case:

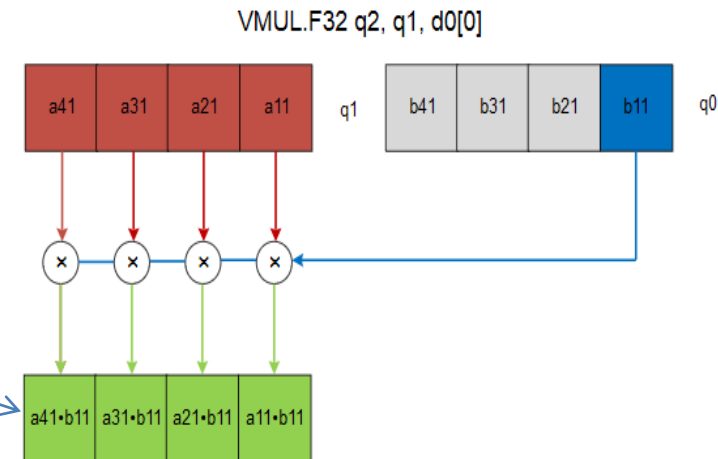
### ■ SIMD Unit inside CPU

For example, matrix multiplication: (ARM: NEON unit as a SIMD extension):

Figure 4.4. Matrix multiplication showing one column of results

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} = \begin{bmatrix} a_{11} \cdot b_{11} + a_{12} \cdot b_{21} + a_{13} \cdot b_{31} + a_{14} \cdot b_{41} & \dots & \dots & \dots \\ a_{21} \cdot b_{11} + a_{22} \cdot b_{21} + a_{23} \cdot b_{31} + a_{24} \cdot b_{41} & \dots & \dots & \dots \\ a_{31} \cdot b_{11} + a_{32} \cdot b_{21} + a_{33} \cdot b_{31} + a_{34} \cdot b_{41} & \dots & \dots & \dots \\ a_{41} \cdot b_{11} + a_{42} \cdot b_{21} + a_{43} \cdot b_{31} + a_{44} \cdot b_{41} & \dots & \dots & \dots \end{bmatrix}$$

Figure 4.5. NEON vector-by-scalar multiplication



GPU: similar philosophy, is a broader concept



## 2. Why Comp.Arch., HW ?

- Because knowledge on computer architecture and operation leads to more efficient programming (programs).
  - Case 2: program code optimization regarding the parallel execution

us/Iteration	Iterations/sec
2.02500	493827.16
0.53300	1876172.61

Code below is 4-times faster !

Reference: „Pomen poznavanja računalniške arhitekture“,  
avtor Miha Krajnc.

```
double results[st];

for(int i = 0; i < st; ++i)
{
    results[i] = a[i] * b[i];
}
```

```
float results[st];

for(int i = 0; i < (st - 8); i += 8)
{
    __m256 i_a = _mm256_load_ps(&a[i]);
    __m256 i_b = _mm256_load_ps(&b[i]);
    __m256 i_c = _mm256_mul_ps(i_a, i_b);
    _mm256_store_ps(&results[i], i_c);
}

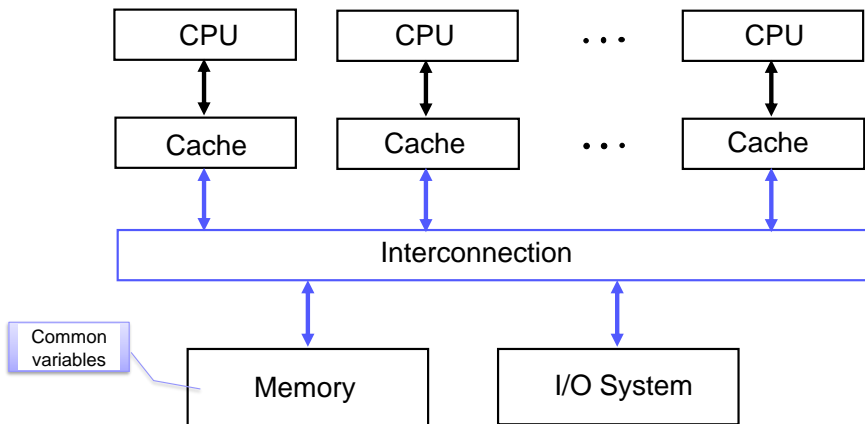
for(int i = (st - 8); i < st; ++i)
{
    results[i] = a[i] * b[i];
}
```



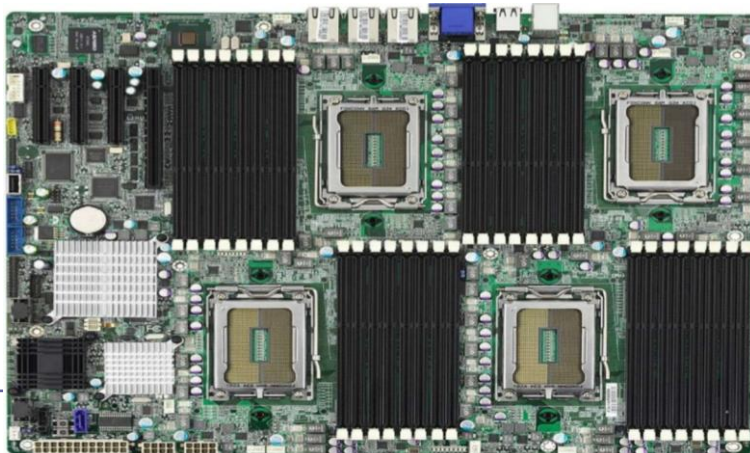
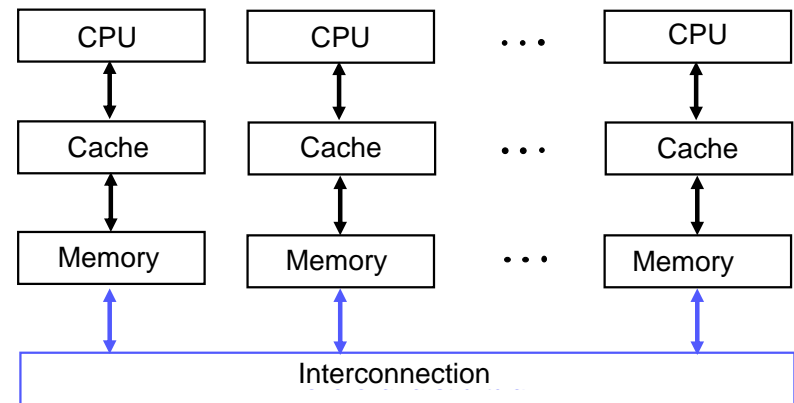
## Examples:

### ■ 4 MIMD (Multiple Instruction multiple Data)

#### Multiprocessor (closely connected)



#### Multicomputers (loosely connected)





## 3 Basic principles of computing - content

- von Neumann computer model
- Flynn's classification
- The main memory in von Neumann computer
- Amdahl's law
- Languages, levels and virtual computers
- Case: Execution of the program on the computer



## 3.3 Main Memory in Von Neumann based computer

### ■ Definition

- Main memory is a **passive device** and serves for storage of instructions and operands.
- The basic cell in the memory is a memory cell which can store 1 bit of information (the content of 0 or 1).

### ■ Memory word (memory location)

- The memory word is defined as the minimum number of bits that have their own address. Memory word is thus the smallest addressable unit of memory.
- The Memory is a one-dimensional sequence of memory words.
- The Memory word comprises a number of one-bit memory cells.
- **The length of the memory word:** the number of one-bit memory cells that make up the memory word. Nowadays, the most common word length is 1 byte (= 8 bits).



## ■ Memory Address

- It is a unique label for each memory word
- Each memory word has its own unique memory address.
- Address memory word is unchangeable.
- The number of bits that comprise the address, are denoted as **Address Length**.
- Title length of the address in bits determines an **Address Space**.

## ■ Address Space (also a memory space)

- it is the set of all addresses
- And also determines the maximum memory size.

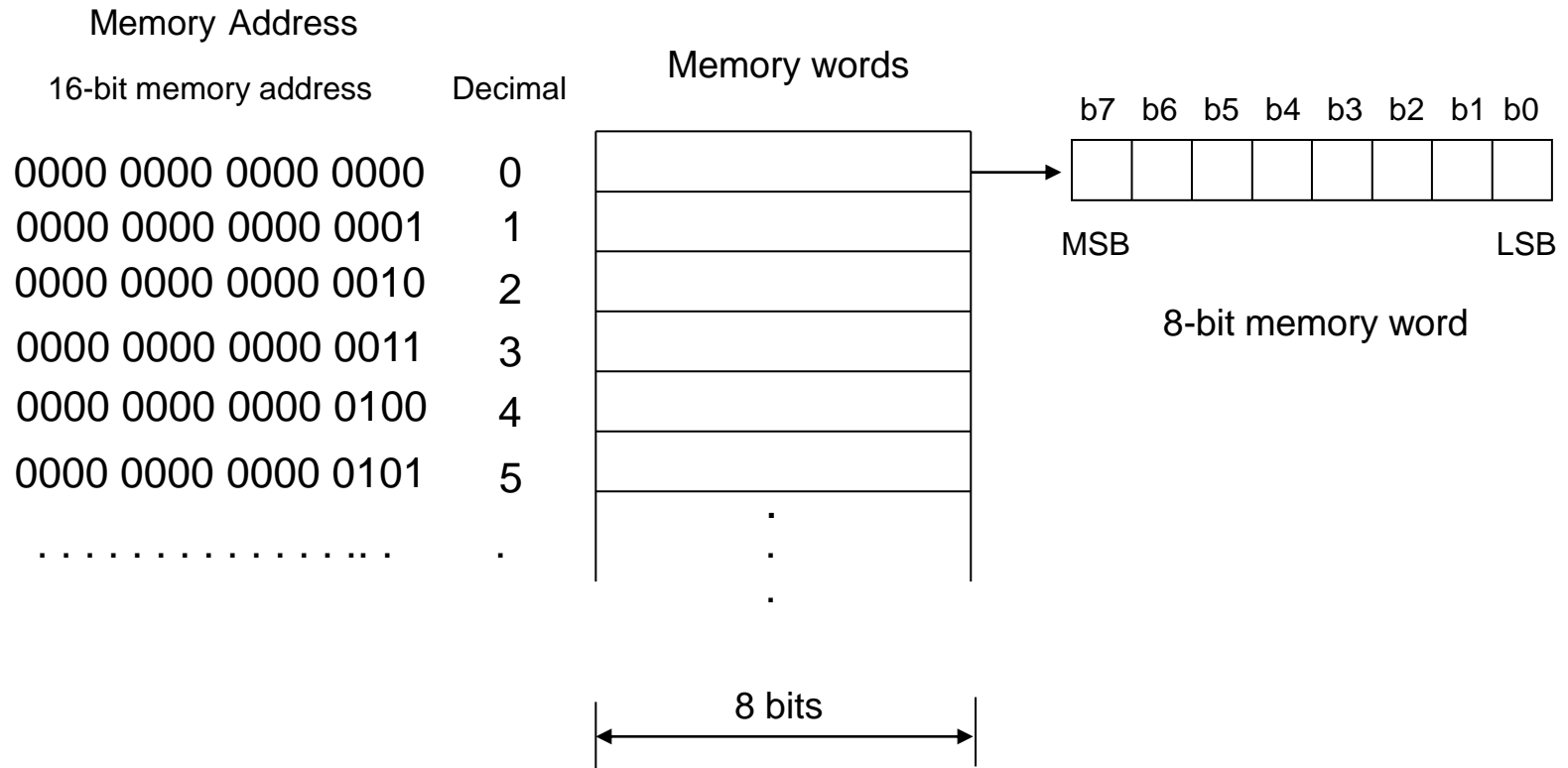




- **The content of the memory words** can change. In an 8-bit memory word can be stored for  $2^8 = 256$  different content.
- The Address of memory word is unchangeable.
- The number of memory words in the main memory is not necessarily equal to the size of the address space.
- Parts of the address space may be empty (all addresses are not used)  $\Rightarrow$  main memory is usually smaller than the maximum size.

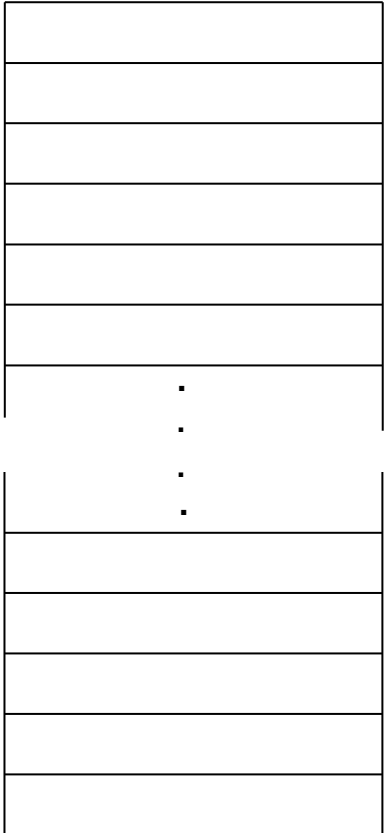


## The main memory of the Von Neumann computer





## The main memory of the Von Neumann computer

Memory Address			memory words
Binary (16-bit address)	Hex	Decimal	
0000 0000 0000 0000	0000	0	
0000 0000 0000 0001	0001	1	
0000 0000 0000 0010	0002	2	
0000 0000 0000 0011	0003	3	
0000 0000 0000 0100	0004	4	
0000 0000 0000 0101	0005	5	
.....			
.....			
.....			
.....			
.....			
.....			
.....			
.....			
.....			
.....			
.....			
1111 1111 1111 1011	FFFB	65531	
1111 1111 1111 1100	FFFC	65532	
1111 1111 1111 1101	FFFD	65533	
1111 1111 1111 1110	FFFE	65534	
1111 1111 1111 1111	FFFF	65535	



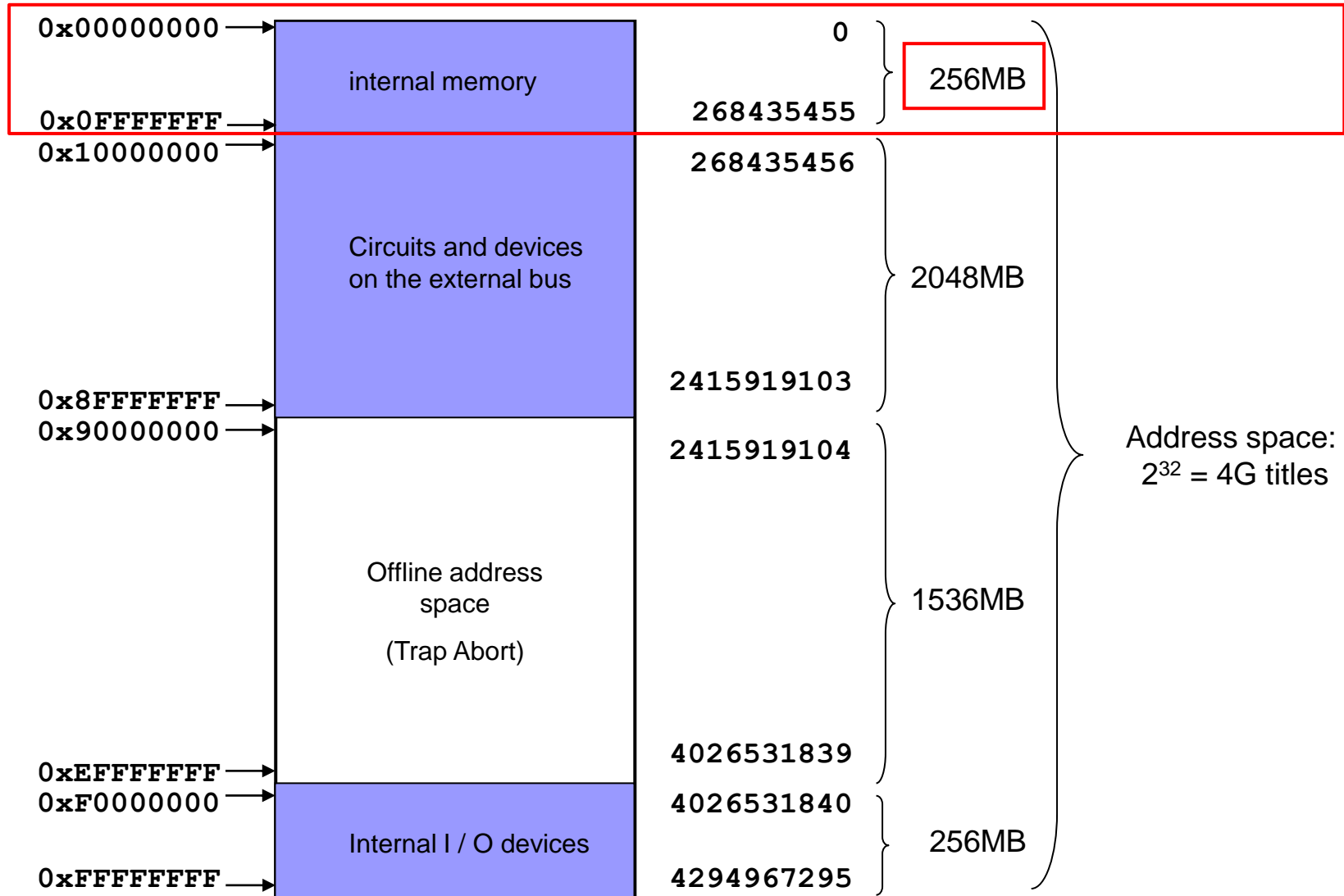
# The prefixes kilo, mega, giga et al. are only in memory size related to powers of 2!

- 1K (kilo) =  $2^{10} = 1024$  (1 KB = 1024 B)
- 1M (mega) =  $2^{20} = 1,048,576$  (1 MB = 1048576 B)
- 1G (giga) =  $2^{30} = 1073741824$  (1 GB =  $1024 * 1024 * 1024 = 1073741 824$  B)
  - The reason is technological: eg. 10-bit memory address allows  $2^{10} = 1024$  different addresses and not 1000 !
  - Proposal of IEC 1998: KiB =  $2^{10}$  B, MiB =  $2^{20}$  B GiB =  $2^{30}$  B
- Other areas (frequency, transfer capacity ...)
  - 1k (kilo) =  $10^3 = 1000$  (1 km = 1000 m)
  - 1M (mega) =  $10^6 = 1\ 000\ 000$  (100 Mb / s = 100 000 000 b/ S)
  - 1G (giga) =  $10^9 = 1,000,000,000$  (1 GHz = 1000000000 Hz)



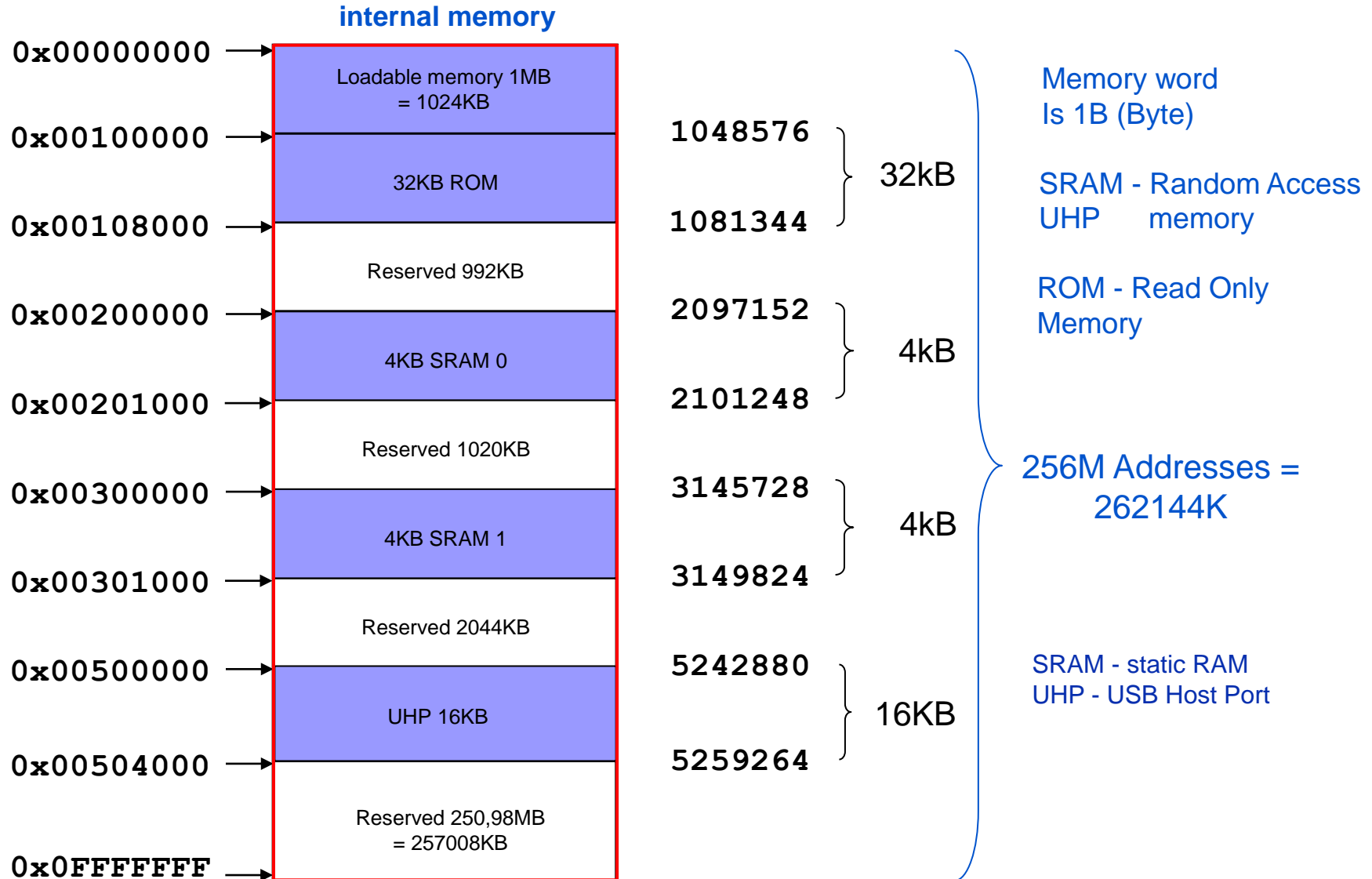
An example of the memory on the processor ARM AT91SAM9260 (32-bit memory address)

32-bit address - 8 hex characters





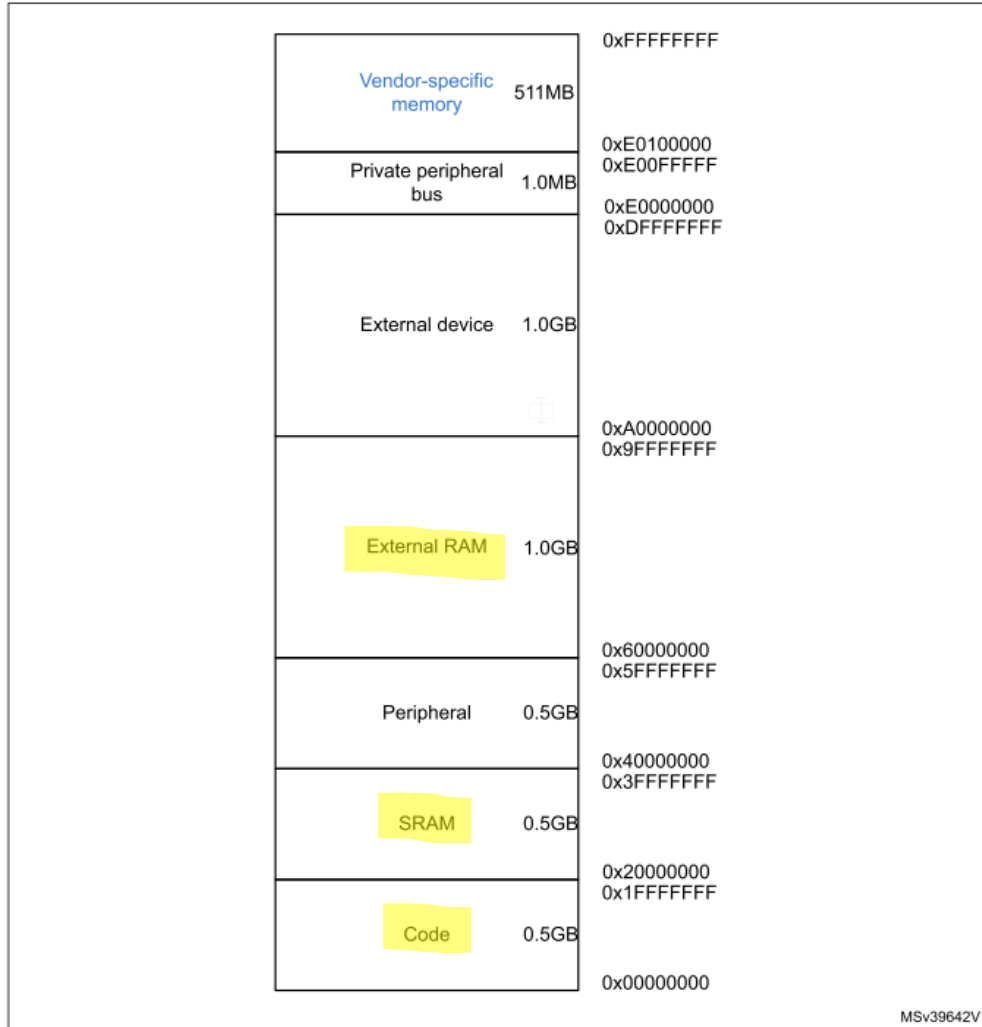
## Picture of the internal memory (the first 256 MB) of AT91SAM9260





## An example of the memory on the microcontroller STM32H750XB

**Figure 8. Processor memory map**



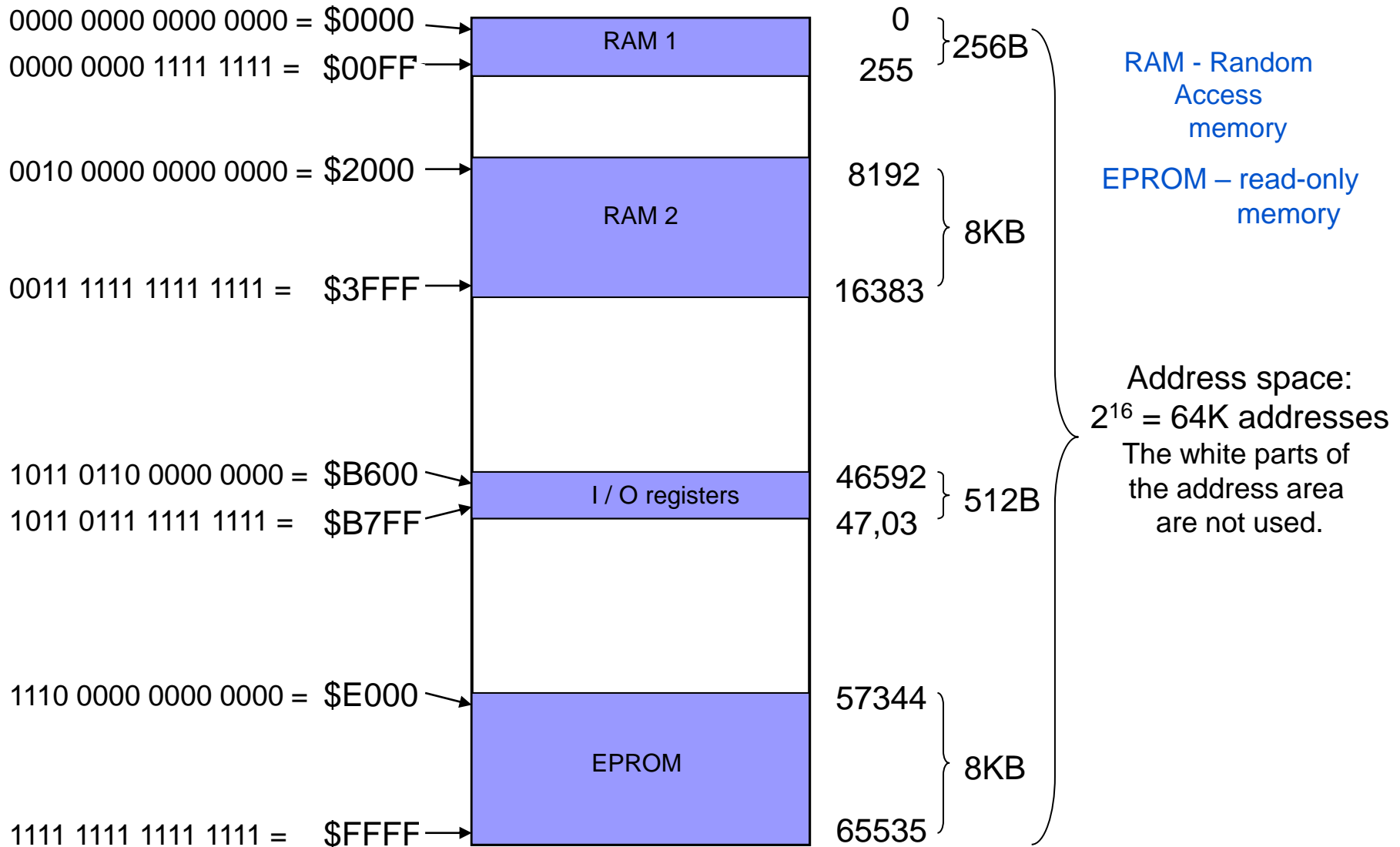
Address space:  
 $2^{32} = 4\text{G}$  addresses

```
MEMORY
{
  FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 128K
  DTCMRAM (xrw) : ORIGIN = 0x20000000, LENGTH = 128K
  RAM_D1 (xrw) : ORIGIN = 0x24000000, LENGTH = 512K
  RAM_D2 (xrw) : ORIGIN = 0x30000000, LENGTH = 288K
  RAM_D3 (xrw) : ORIGIN = 0x38000000, LENGTH = 64K
  ITCMRAM (xrw) : ORIGIN = 0x00000000, LENGTH = 64K
}
```



For example, the image of memory (memory map) the processor 68HC11 - the processor has a 16-bit memory address

The 16-bit memory address



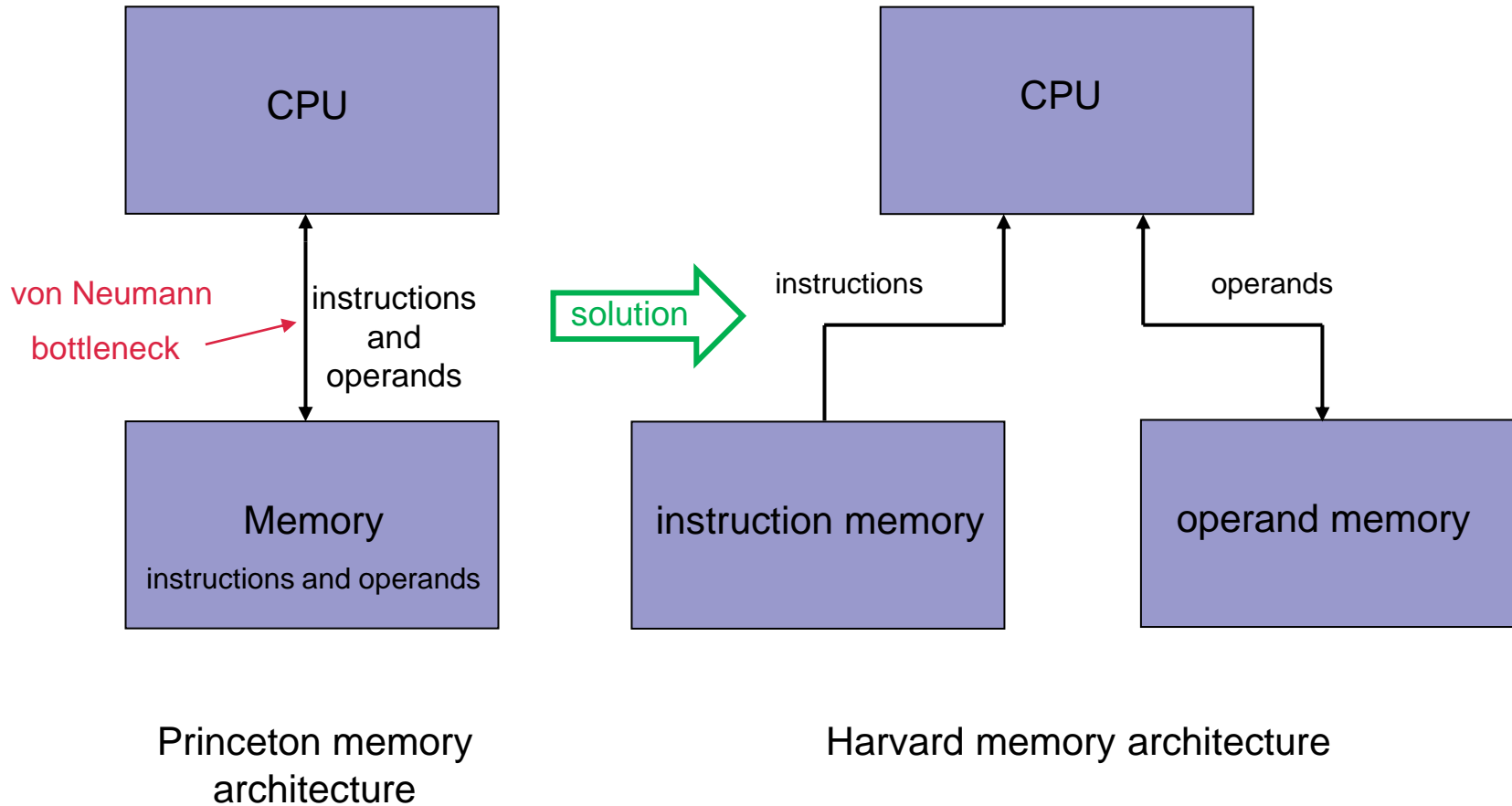




# Von Neumann bottleneck

- Transfers CPU ↔ Main Memory – produce a lot of traffic
- Von Neumann's bottleneck – is the connection between the CPU and the main memory. All instructions are transferred from the main memory to the CPU, and all operands are transferred in both directions - from memory or in the memory.
- One way to extend this bottleneck, is the split of the main memory into two parts.

# Extension of the Von Neumann bottleneck





- Memory in the Harvard architecture is divided into two separate memories.
- In one only operands are stored – „operand memory“, second only includes instructions – „instruction memory“.
- Instruction and operand memories can operate simultaneously. Thus we can achieve double speed.
- Harvard architecture is used nowadays in cache memories at the lowest level (separate operand and instruction L1 caches), but the main memory of most computers is usually uniform (Princeton architecture).



# Access to memory

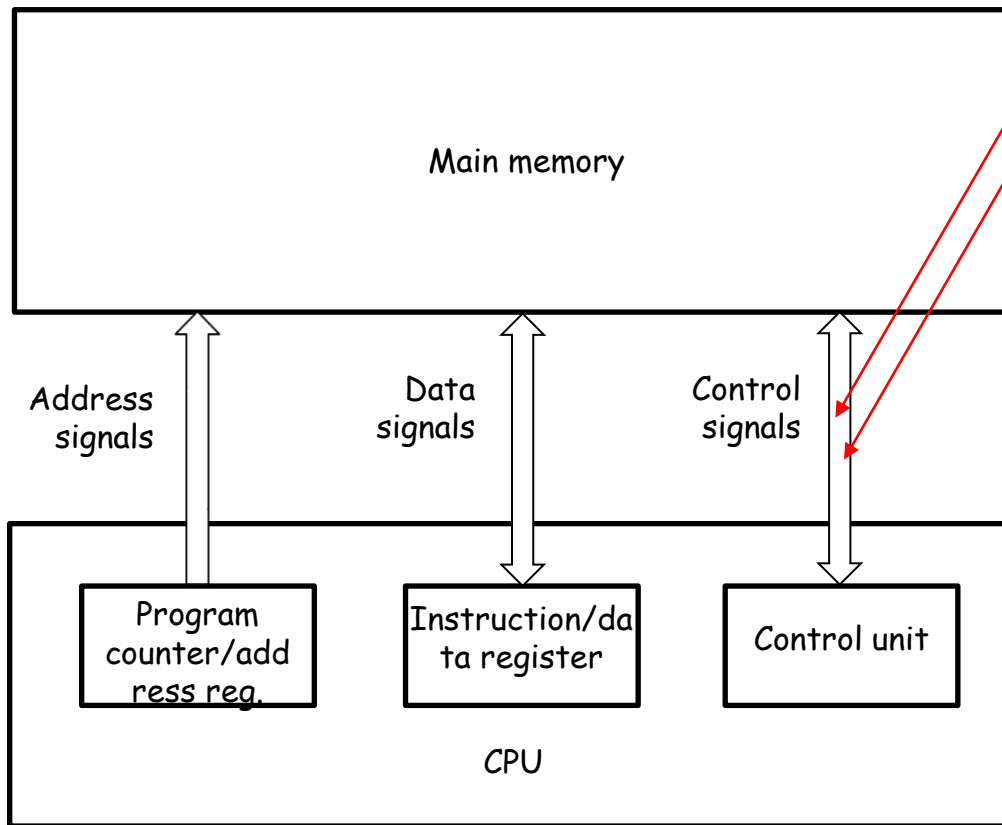
- CPU accesses to memory word by first sending its address to the memory and the signal that determines the direction of transfer.
  
- The direction of transfer - type of access
  - CPU ← main memory - reading (read access)
  
  - CPU → main memory - write (write access)



# Interconnection CPU <-> main memory?

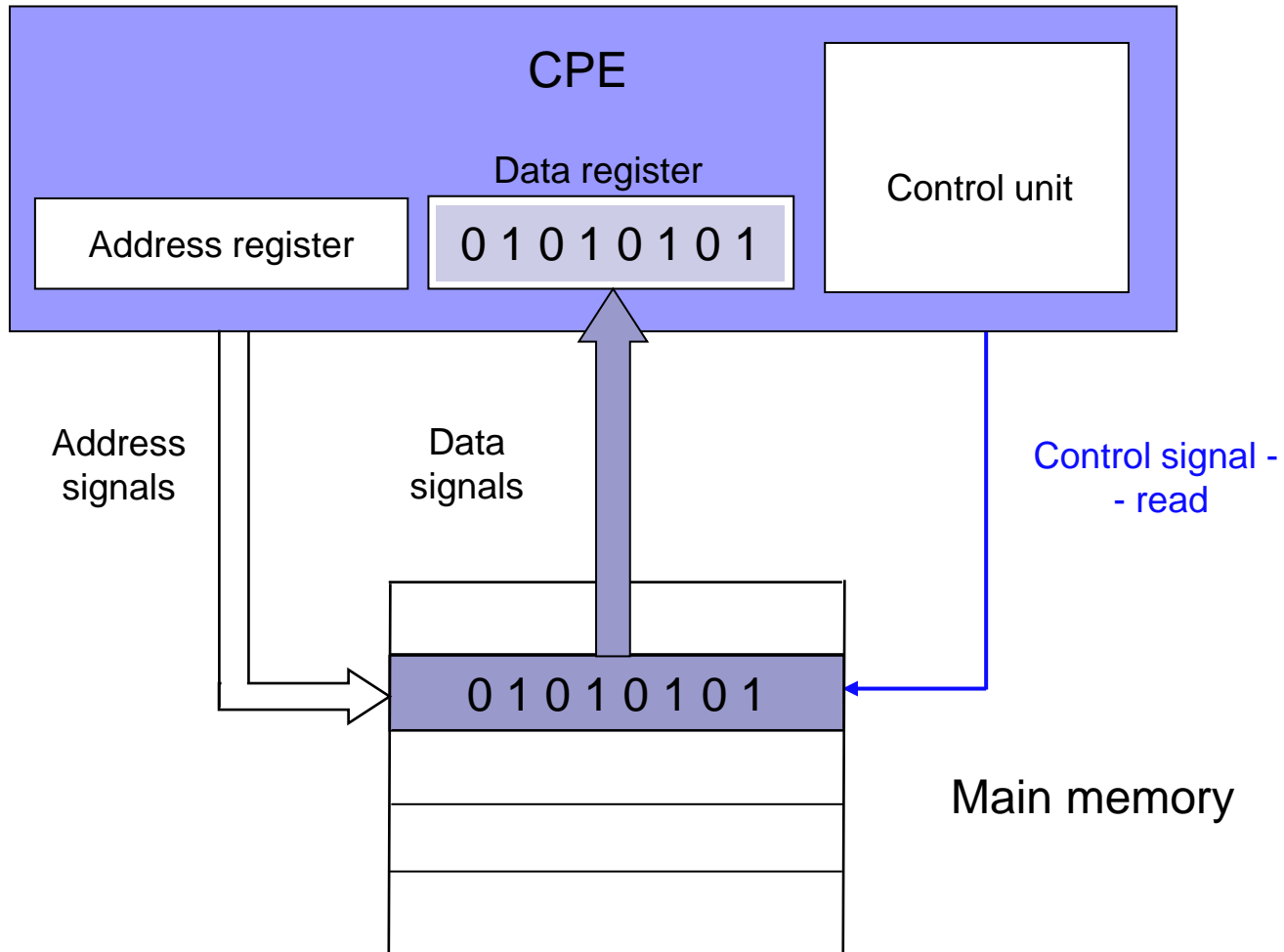
Bus = group of lines  
(Address, Data, Control buses)

Line = physical connection  
Signal = content transferred over the line (1bit)



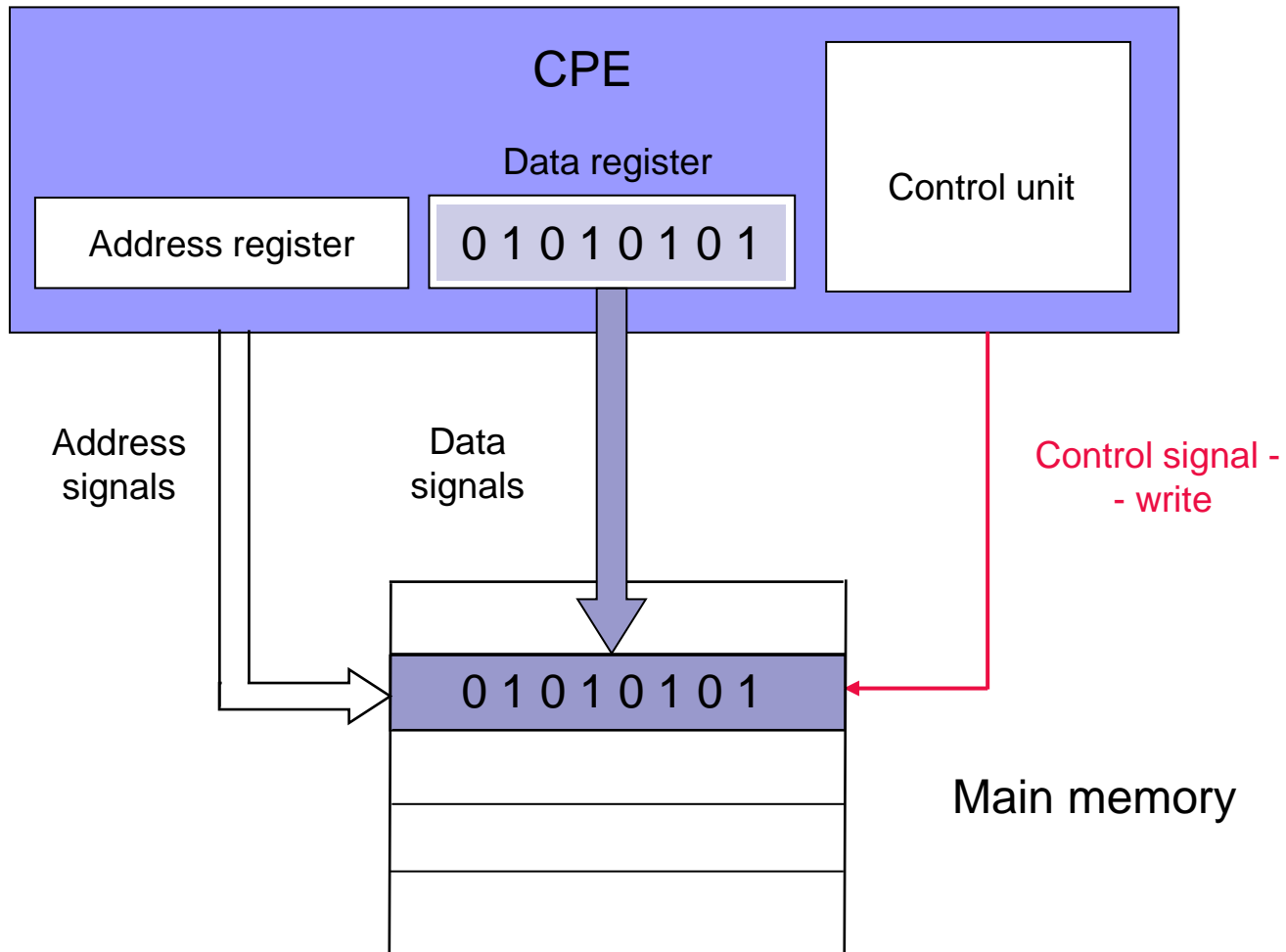


# The connection between the CPU and main memory - read access



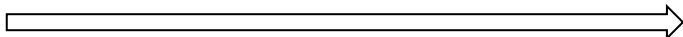


# The connection between the CPU and main memory - write access





# Summary of the memory properties in Von Neumann computer

- Memory is **one-dimensional** and organized as a sequence of **words**. Each word has its own, **unique address**.
- There is no difference between **instructions and operands in memory**.
- **Type or description** is not included in operands.
- **More read than write accesses**,
  - Ratio: approximately 80% are read (R), 20% are write accesses (W)
  - **Why?** 

example  
program

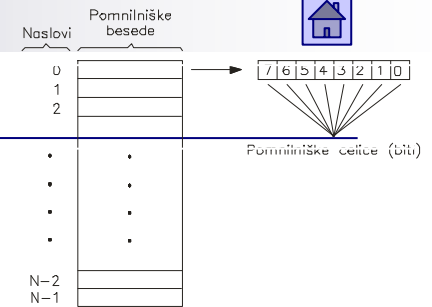
PROGRAM	
1R	adr r0,STEV1
2R	ldr r1,[r0]
1R	adr r0,STEV2
2R	ldr r2,[r0]
1R	add r3,r1,r2
1R	adr r0,REZ
1R1W	str r3,[r0]





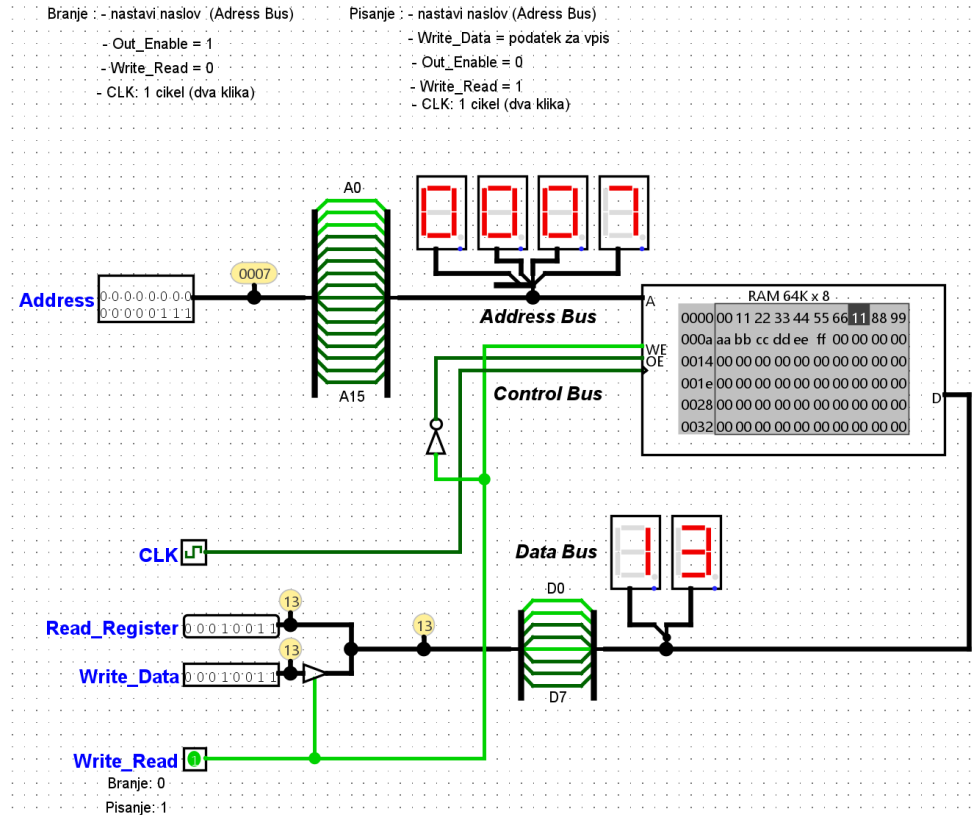
The combination of 8 bits in the memory,  
eg. 1000 1011, can represent:

- Unsigned: 139 (decimal)  
or
- number with sign: - 11 (decimal)  
or
- Extended ASCII character : <
- or
- Hardware instruction: ADDA (op.code of the machine instruction  
for processor 68HC11)  
or
- memory address 139 (decimal)  
or
- combination of bits  
or
- point in image (pixel), audio sample, . . .



# Memory Demonstration – Logisim EVO

Primer delovanja pomnilnika RAM



RAM\_pomnilnik\_demo\_EVO.circ



## 3 Basic principles of computing - content

- von Neumann computer model
- Flynn's classification
- The main memory in von Neumann computer
- Amdahl's law
- Languages, levels and virtual computers
- Case: Execution of the program on the computer



## 3.4 Amdahl's law (1967)

- G.M. Amdahl was one of the architects of the famous family of computers IBM 370
- If the computer speeds up all operations by a factor  $N$  ( $N$ -times), except the relative  $f$ -part of all operations, then increase in the speed of entire computer  $S(N)$  is

$$S(N) = \frac{1}{f + \frac{1-f}{N}} = \frac{N}{1 + (N-1) * f}$$

$f$  - the portion of operations that are not *accelerated*!

$S(N)$  = Increase in the speed of the entire system

$N$  = a scaling factor of the speed of  $(1 - f)$  portion of operations

$f$  = portion of operations, which are not accelerated

$1 - f$  = fraction of operations that are  $N$  times accelerated

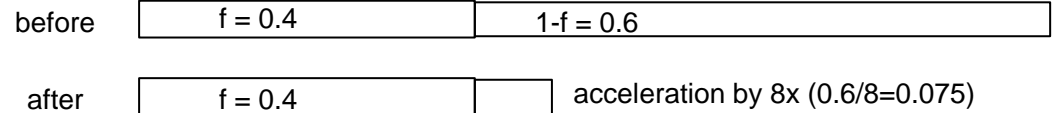


## Case 1:

- Implementation of programs on a computer would like to be accelerated so that the single-core processor is replaced with eight-core CPU (8 CPUs operating in parallel).
- How much faster will software run, if only 60% of the programs can be performed in parallel?



## Amdahlov law



- $N = 8$  (part of the programs can be performed eight times faster)
- $1 - f = 0.6$ , the proportion of the programs that have 8-fold speed up;
- $f = 0.4$ , the proportion of the programs which are not sped up (40% of the programs can not be executed in parallel)
- $S(N)$  speed up the whole SW (all programs)

$$S(N) = \frac{8}{1 + (8 - 1) * 0.4} = \frac{8}{1 + 2.8} = 2.1$$

- The speed of all programs will be increased by a factor of 2.1 (2.1 times).
- If the programs were executed before the replacement 100 seconds, will be then executed in 47.6 seconds ( $100 / 2.1 = 47.6$ ) on 8-core CPU.



before 

$f = 0.1$	$1-f = 0.9$
-----------	-------------

## Case 2:

then 

$f = 0.1$	acceleration by 2x (half-time)
-----------	--------------------------------

- Execution of the program on a computer would like to accelerated so that the 90% of all instructions will be executed two times faster.
- How many times faster will run the program on this computer?

$$S(N)=?$$



before 

f = 0.1	1-f = 0.9
---------	-----------

## Case 2:

then 

f = 0.1	acceleration by 2x (half-time)
---------	--------------------------------

- Execution of the program on a computer would like to accelerated so that the 90% of all instructions will be executed two times faster.
- How many times faster will run the program on this computer?

$$S(N) = \frac{1}{0.1 + \frac{0.9}{2}} = \frac{1}{0.1 + 0.45} = \frac{1}{0.55} = 1.818181$$

- Speed of the program execution is increased by a factor of 1.82.





## Amdahl's law:

- Parallelism is not ideal
- Importance of relative share of operations that can speed up
- Greater the share, less speed up is needed for a similar overall effect

## Parallelism:

- Only viable possibility cause by specificities of elektronic technology evolution
- Not simple from speed up and programming viewpoints
- Has a potential of greater efficiency from energy consumption viewpoint



## 3 Basic principles of computing - content

- von Neumann computer model
- Flynn's classification
- The main memory in von Neumann computer
- Amdahl's law
- Languages, levels and virtual computers
- Case: Execution of the program on the computer



## 3.5 Languages, Levels and Virtual computers

- For the vast majority of users, the details of the structure and operation of computers are insignificant.
- Computer and its features are seen mostly through the features of the programming language that you use.
- A programming language can be realized in a wide variety of computers, this means that different computers for a user who uses the same programming language look more or less the same.



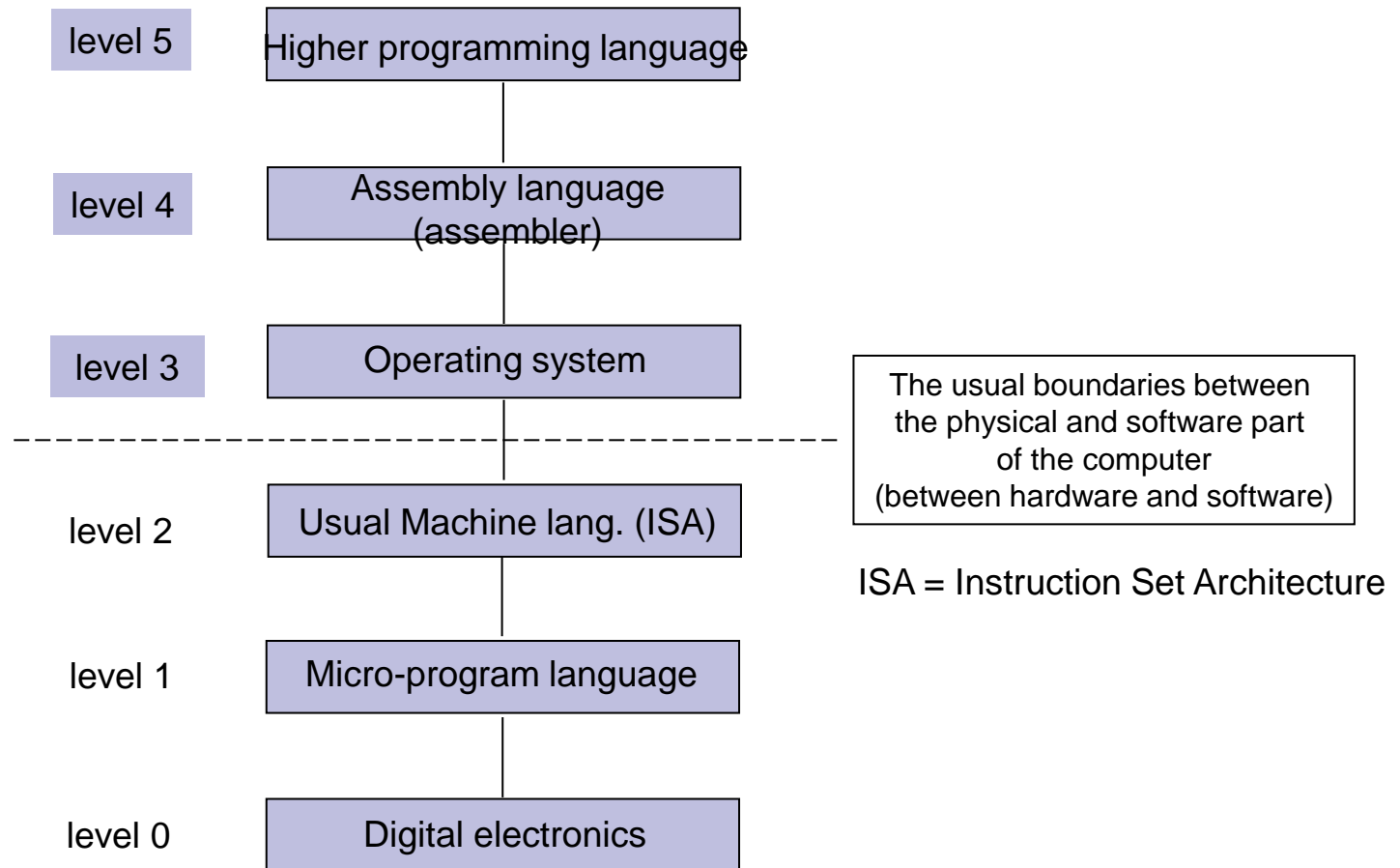
# The computer as a series of virtual computers

- The vast majority of today's computers have 6 levels.
- At each level we see a computer through a different computer programming language.
- This programming language can be represented as the „machine language of a certain virtual machine“.
- At the lowest level (level 0) Electronics (logic gates and flip-flops) directly executes the simplest (machine) instructions.





## A computer with six levels





- Level 1 can be seen in many of today's computers. RISC computers don't have first level.
  - Each instruction of „usual“ machine language is executed as a sequence of micro instructions - computer, which operate in this manner (with level 1) are denoted as micro-programmed.
  - For these computers, micro-program language is actually the real machine language.
  - Since at the beginning of the computers, this level was invisible to the user, the term „machine language“ is usually used for the level 2.
  - Micro-program on level 1 is written by the manufacturer of the CPU and actually defines the usual machine language. Usually, it can not be changed by the user.



- The user sees the computer on the level 2 through the use of conventional machine instructions, which form the conventional machine language.
  - Computer architecture is determined by the structure and properties of the computer, as seen by the programmer at this level.
  - Therefore, the name of the ISA - Instruction Set Architecture.
  - With the conventional machine language programmer has full control over all parts of the computer.
  - At early stages of evolution, the computers didn't have higher levels, and programming took place only in the normal machine language.



- Level 3 is the level of the operating system.
  - Language at this level contains all the instructions of Level 2, with the addition of new instructions to better control the computer (eg. operations with I/O devices, parallel execution of programs, diagnostic instructions).
  - The operating system is a program that facilitates computer work and serves as an interface between the user and the computer hardware.
  - With operating system we want to achieve:
    - easier work
    - better utilization of hardware capabilities of the computer (do more work in given time).





- The functions of the operating system could be implemented in the hardware Level 2, but is currently more economical to do it in software (multiple operating systems, upgrade...).
- At this level, we usually divide users with different rights to use the instructions.
- Some instructions in Level 2 are in level 3 inaccessible (available only to system programmers) to normal users .
- For most of today's programmers is level 3 the lowest level at which they can work.



- At level 4 user can see the computer through the assembly language.
  - Assembly language is only symbolic form, closer to humans, of language on Level 3 (and thus the Level 2).
  - Programs in assembly language must be translated before the execution to the language on Level 3 (or 2).
  
- Level 5 is formed of higher programming languages, which are designed to majority of computer programmers.
  - This are, for example, C, C#, C++, Java, Python, BASIC, FORTRAN, COBOL, and many others.
  - Programs written in these languages must be translated to the language on Level 4 or Level 3.



- Regarding computers, we can establish also higher levels, e.g. programs for AI, databases, ...
- Each level can be thought of as a virtual computer that has own „machine language“ as the language of this level. Therefore, a typical user at higher levels doesn't need to know the details about actual „machine level“.
- However, it is mandatory that programs written in any higher level language (for corresponding virtual machine) are converted into a sequence of machine language instructions.
- Users don't need to be fully aware of this translation, providers of HW and SW products must ensure the tools for translation from one language to another.

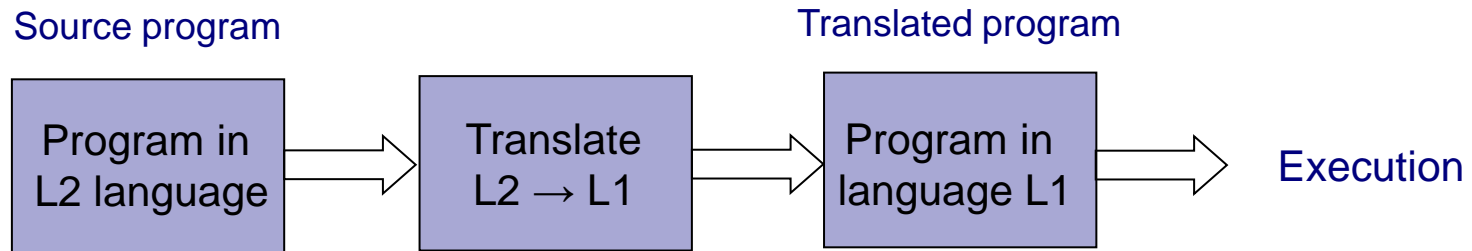


- The mechanism of transition from one language to another can be realized in two ways:
  - Translation (or compilation)
  - Interpretation.
  
- After 1990, an intermediate solution emerged:
  - partial translation (compilation).
  
- The main difference between translation (compilation) and interpretation is that in interpretation, the translated (compiled) program does not exist.



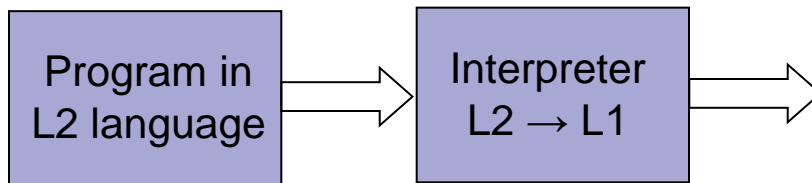
## Transition from the language L2 into the language L1

### Translation (compilation)



### Interpretation

Source program



Each instruction of the language L2 is simultaneously translated into the instructions in language L1 and executed (No translated program!)



- Compiled programs work only on the computer with machine language in which they were translated.
- Before transferring to another computer (using a different machine language L1a) we should recompile the source code of a program.
  
- By integrating a large number of different computers on the network, the portability of programs enabled by interpretation, has become very important.
  
- Partial translation is an intermediate solution between the interpretation and translation, which enables faster interpretation on target machine.



- Partial translation: Source language program in L2 is translated into an intermediate language program in L1, and then L1 program is interpreted on a target machine.
- Partial translation in the intermediate language L1 allows faster interpretation, but is still typically 10 times slower than full implementation of the program translation (compilation).
- Despite this, it still allows portability of programs at a significantly lower loss of speed than if we used the interpretation only.



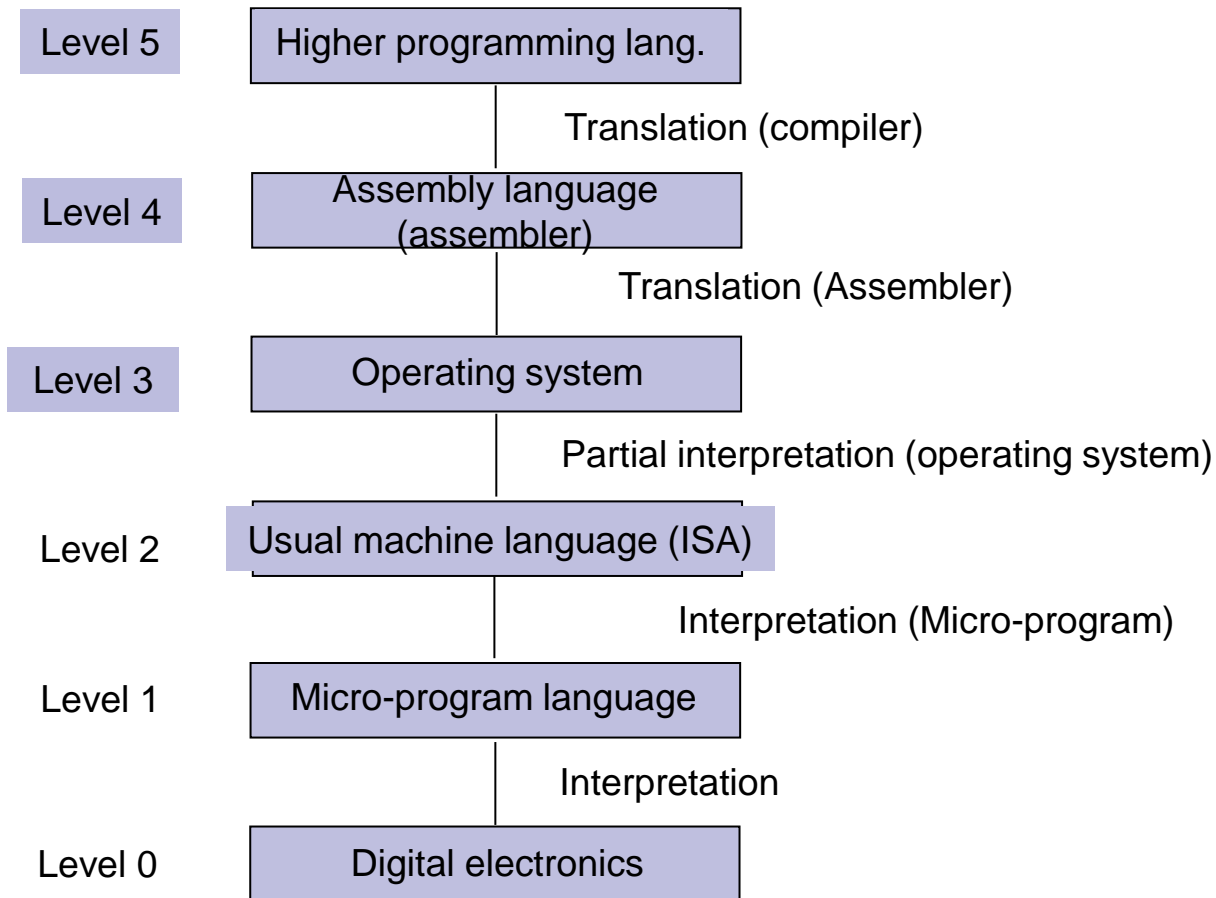
- Practical case of virtual machine: JVM (Java Virtual Machine)
  - Virtual Machine - VM (Virtual Machine) is a software implementation of the machine (computer), operating (running programs) like a real machine (computer).
  - Java programs are executed so, that they are first translated (partial translation) in an intermediate language (Java byte code), which is interpreted by the program JVM on a target machine.





# Computer with six levels (Micro-programmed)

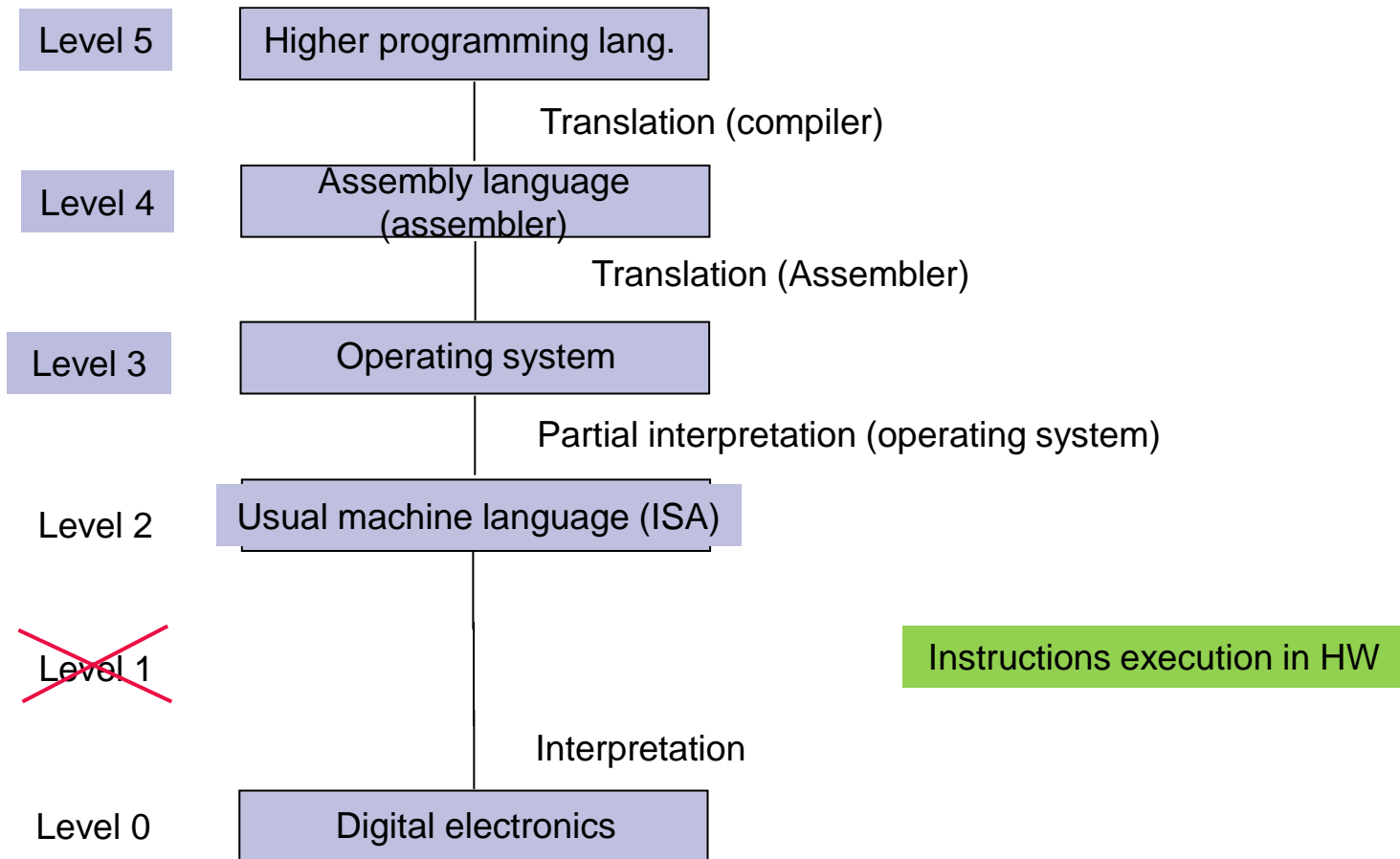
## Older generation of computers





# Computer with five levels

## Newer generation of computers





## Hardware and software on computer

- The boundary between hardware and software of the computer is not solid - it can be moved.
- Each of the levels can be realized in both hardware and software way.
- Level 2 for example: It can be realized with a program running on another computer.

Hardware and software are logically equivalent.



- Each operation carried out by the software can be realized as hardware directly.
  
- Also, each machine instruction, executed by hardware, can also be simulated with the program.
  
- Evolution of multi-level computing machines
  - Invention of Micro-programming (1951)
  - Invention of Operating system (OS) (around 1960)
  - Moving functionality in Micro-programs (around 1970)
  - Abandonment of Micro-programming (after 1984)
  - Today usually the combination of:
    - the complex instructions at normal machine level are realized in micro-program (software), simpler instructions are realized in hardware.



## 3 Basic principles of computing - content

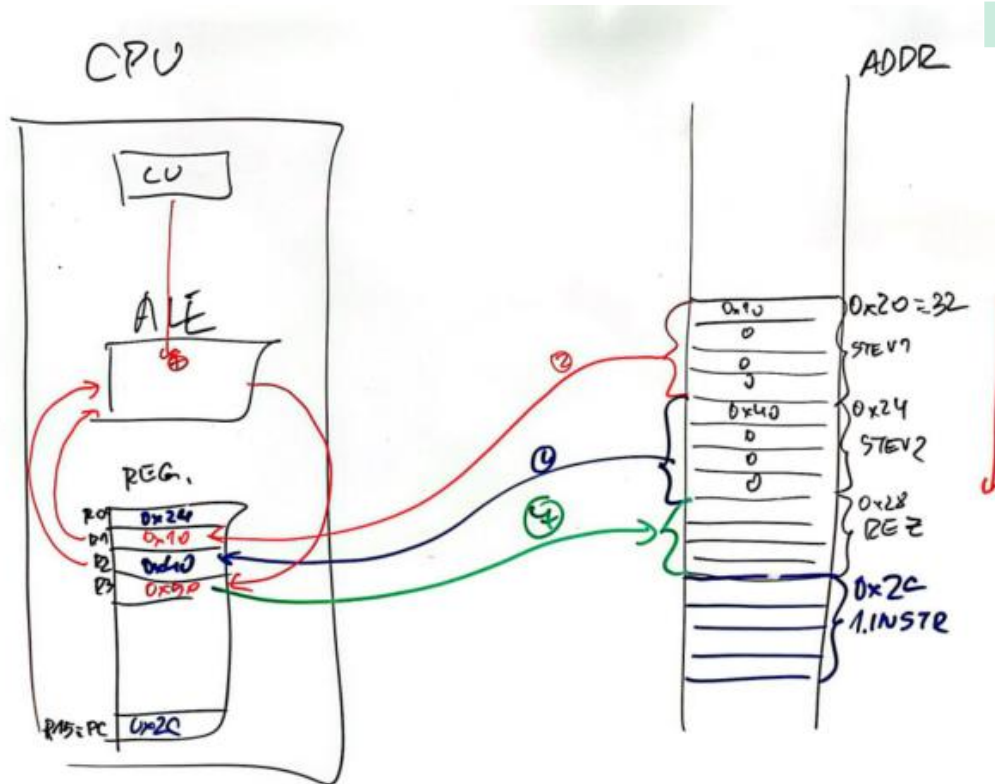
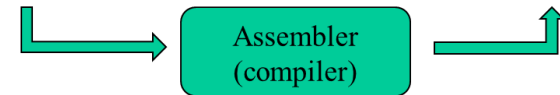
- von Neumann computer model
- Flynn's classification
- The main memory in von Neumann computer
- Amdahl's law
- Languages, levels and virtual computers
- Case: Execution of the program on the computer



## 3.6 Example of program execution on the computer

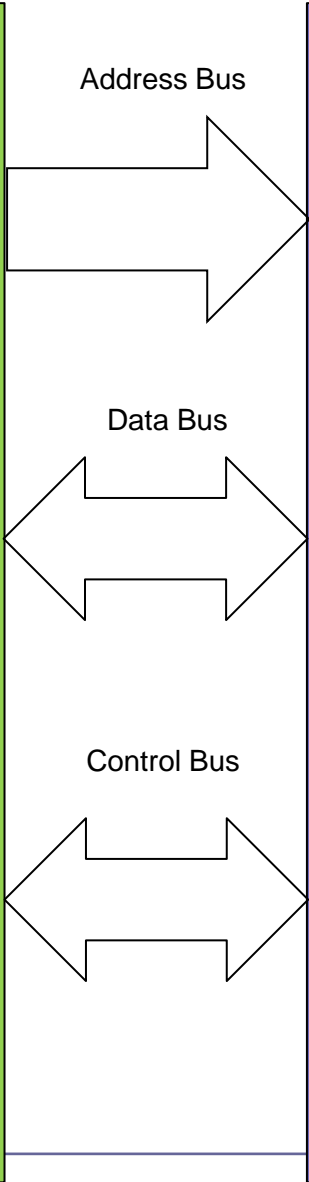
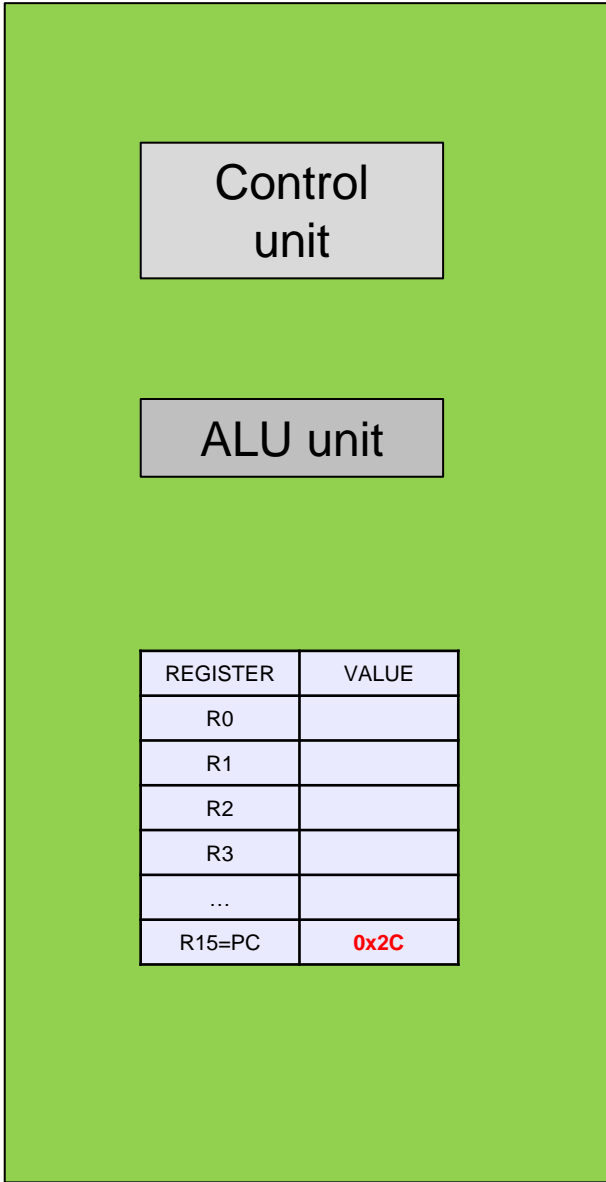
An example of adding two numbers:  
**rez: = stev1 + stev2**

Assembly language	Instruction description	Machine language
adr r0, stev1	R0 ← Addr. of stev1	0xE24F0014
ldr r1, [r0]	R1 ← M[R0]	0xE5901000
adr r0, stev2	R0 ← Addr. of stev2	0xE24F0018
ldr r2, [r0]	R2 ← M[R0]	0xE5902000
add r3, r2, r1	R3 ← R1 + R2	0xE0823001
adr r0, rez	R0 ← Addr. of rez	0xE24F0020
str r3, [r0]	M[R0] ← R3	0xE5803000



# Case execution of program

Instruction	STEP	Comment
		Initial state



CONTENT	ADDRESS	LABEL
0x40	0x20	STEVE1
0	0x21	
0	0x22	
0	0x23	
0x10	0x24	STEVE2
0	0x25	
0	0x26	
0	0x27	
0	0x28	REZ
0	0x29	
0	0x2A	
0	0x2B	
0x14	0x2C	ADR R0,STEVE1
0x10	0x2D	
0x1F	0x2E	
0xE5	0x2F	
	0x30	LDR R1,[R0]
	...	
	0x34...	ADR R0,STEVE2
	0x38...	LDR R2,[R0]
	0x3C...	ADD R3,R2,R1
	0x40...	ADR R0,REZ
	0x44...	STR R3,[R0]
	...	

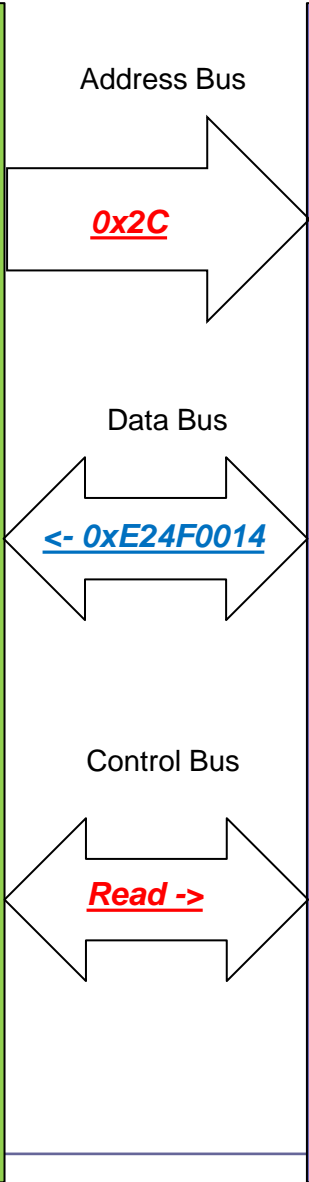
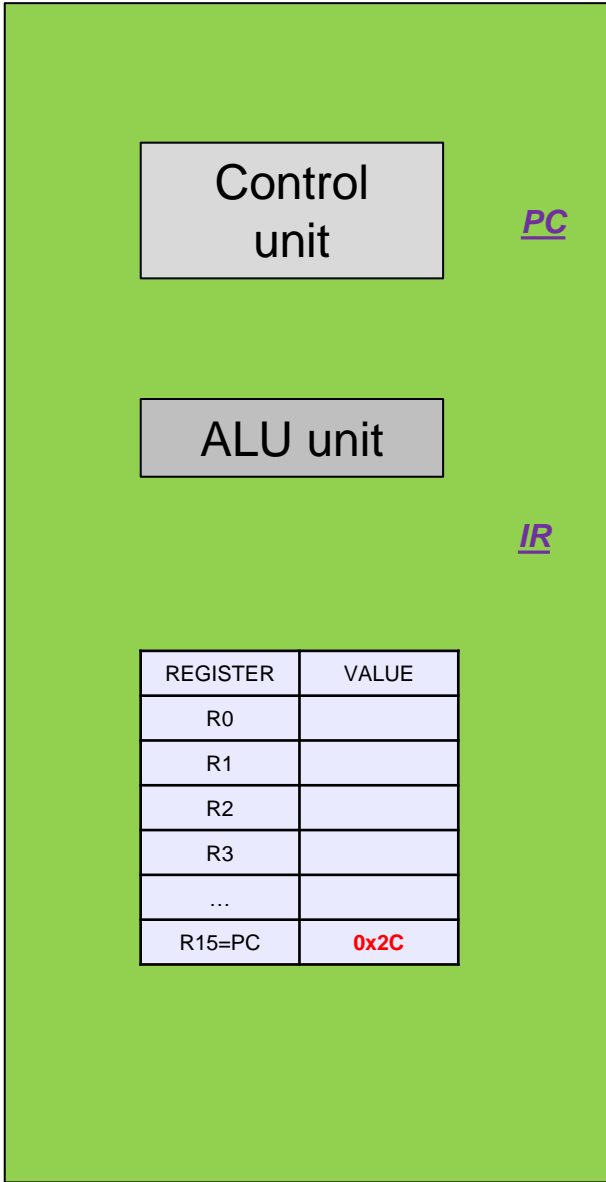
PROGRAM	
→	ADR R0,STEVE1
	LDR R1,[R0]
	ADR R0,STEVE2
	LDR R2,[R0]
	ADD R3,R1,R2
	ADR R0,REZ
	STR R3,[R0]

Machine lang	
→	0xE24F0014
	0xE5901000
	0xE24F0018
	0xE5902000
	0xE0823001
	0xE24F0020
	0xE5803000

#0

Instruction	STEP	Comment
ADR R0,STEV1	FETCH	Read 1. instruction

# Case execution of program



CONTENT	ADDRESS	LABEL
0x40	0x20	STEV1
0	0x21	
0	0x22	
0	0x23	
0x10	0x24	STEV2
0	0x25	
0	0x26	
0	0x27	
0	0x28	REZ
0	0x29	
0	0x2A	
0	0x2B	
0x14	0x2C	ADR R0,STEV1
0x00	0x2D	
0x4F	0x2E	
0xE2	0x2F	
	0x30	LDR R1,[R0]
	...	
	0x34...	ADR R0,STEV2
	0x38...	LDR R2,[R0]
	0x3C...	ADD R3,R2,R1
	0x40...	ADR R0,REZ
	0x44...	STR R3,[R0]
	...	

PROGRAM	
→	ADR R0,STEV1
	LDR R1,[R0]
	ADR R0,STEV2
	LDR R2,[R0]
	ADD R3,R1,R2
	ADR R0,REZ
	STR R3,[R0]

Machine lang	
→	0xE24F0014
	0xE5901000
	0xE24F0018
	0xE5902000
	0xE0823001
	0xE24F0020
	0xE5803000

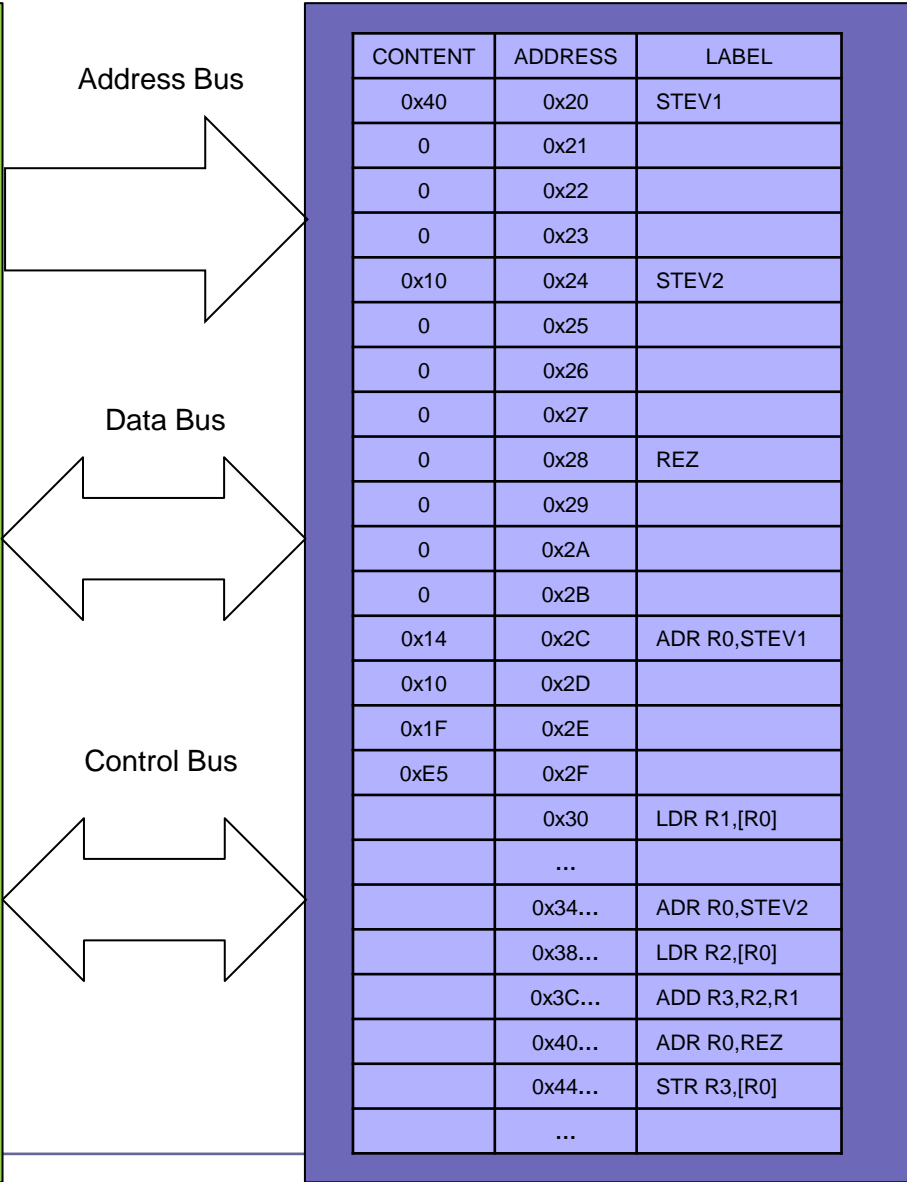
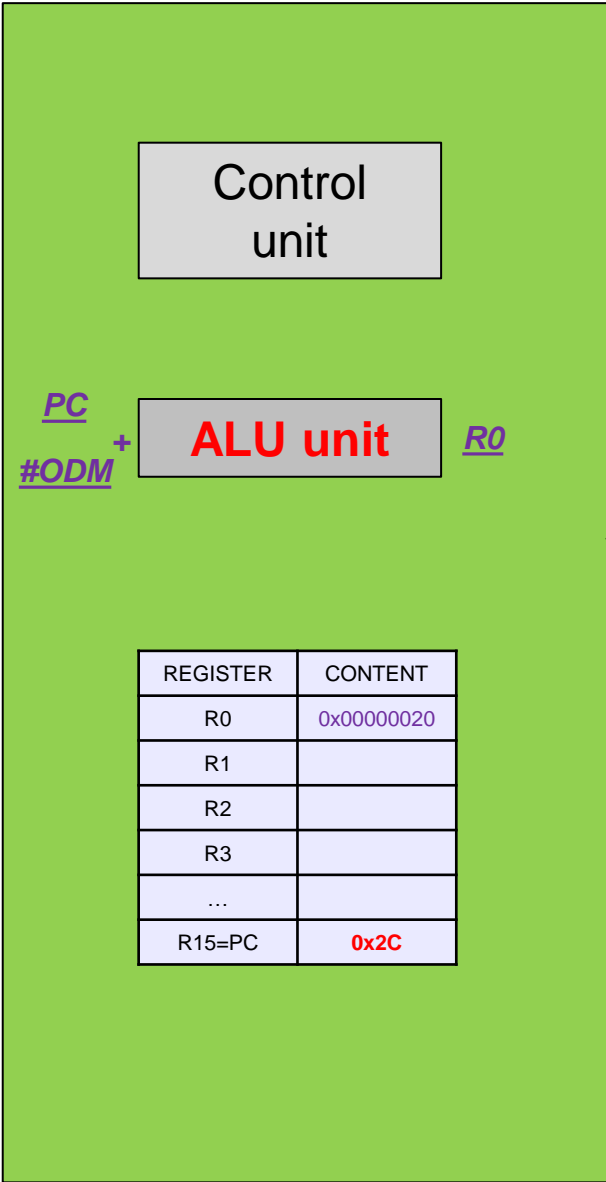
#1





Instruction	STEP	Comment
ADR R0,STEV1	EXECUTE	ALE: R0 <- PC +- ODMIK

# Case execution of program



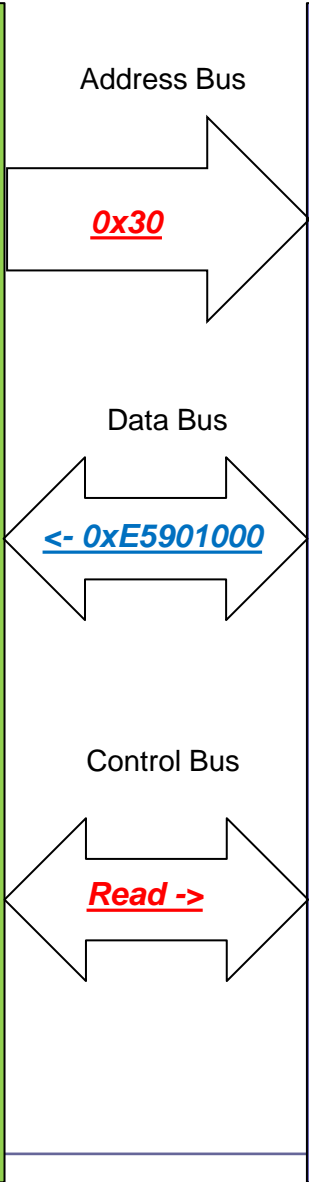
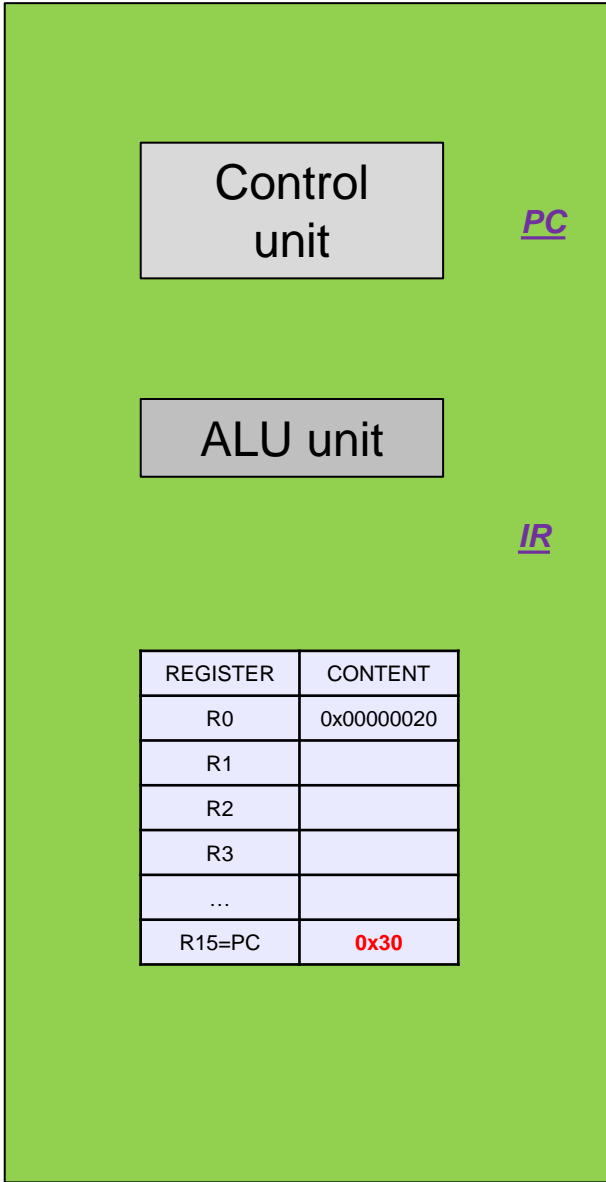
PROGRAM	
➔	ADR R0,STEV1
	LDR R1,[R0]
	ADR R0,STEV2
	LDR R2,[R0]
	ADD R3,R1,R2
	ADR R0,REZ
	STR R3,[R0]

Machine lang	
➔	0xE24F0014
	0xE5901000
	0xE24F0018
	0xE5902000
	0xE0823001
	0xE24F0020
	0xE5803000

#2

Instruction	STEP	Comment
LDR R1,[R0]	FETCH	Read 2. instruction

# Case execution of program



CONTENT	ADDRESS	LABEL
0x40	0x20	STEV1
0	0x21	
0	0x22	
0	0x23	
0x10	0x24	STEV2
0	0x25	
0	0x26	
0	0x27	
0	0x28	REZ
0	0x29	
0	0x2A	
0	0x2B	
0x14	0x2C	ADR R0,STEV1
0x10	0x2D	
0x1F	0x2E	
0xE5	0x2F	
	<b>0x30</b>	<b>LDR R1,[R0]</b>
	...	
	0x34...	ADR R0,STEV2
	0x38...	LDR R2,[R0]
	0x3C...	ADD R3,R2,R1
	0x40...	ADR R0,REZ
	0x44...	STR R3,[R0]
	...	

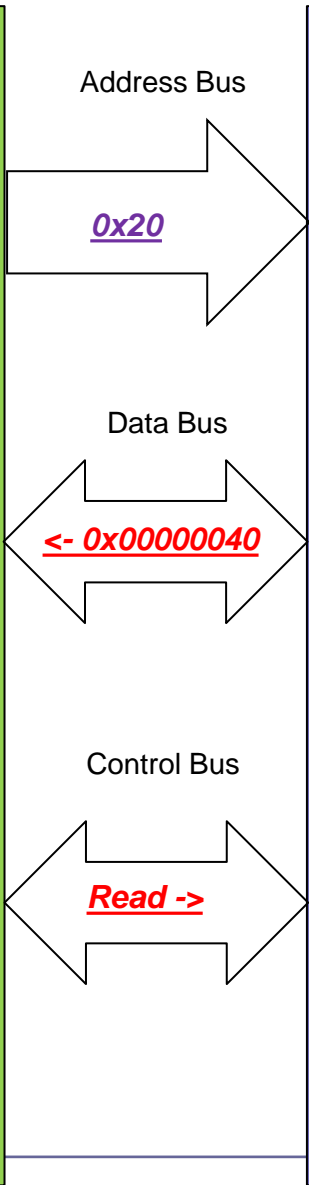
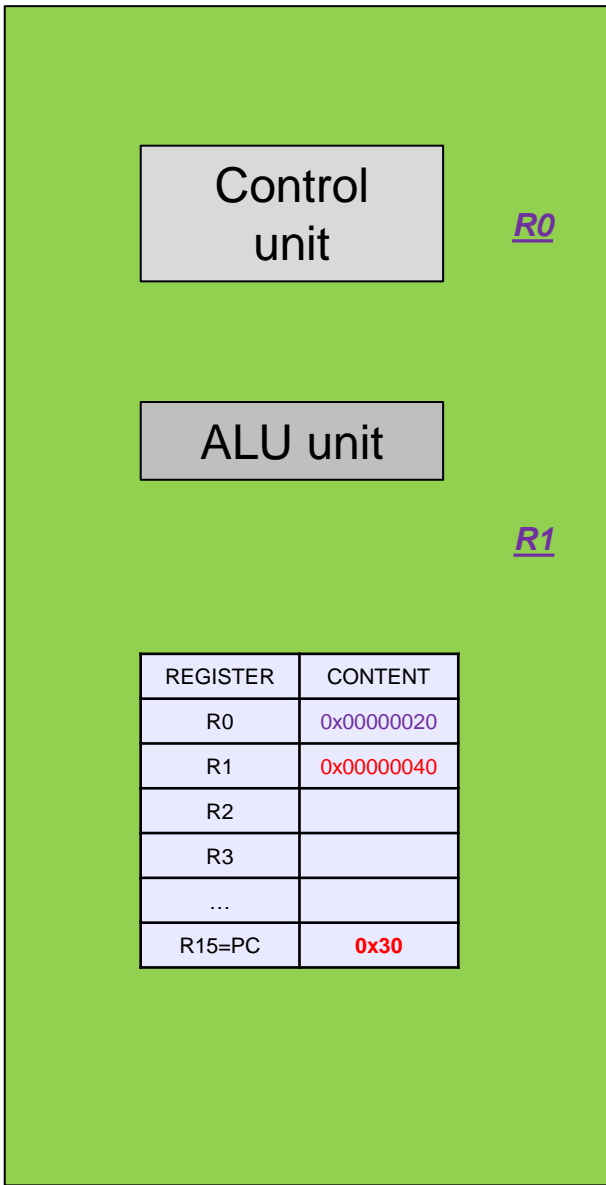
PROGRAM	
ADR	R0,STEV1
→	LDR R1,[R0]
ADR	R0,STEV2
LDR	R2,[R0]
ADD	R3,R1,R2
ADR	R0,REZ
STR	R3,[R0]

Machine lang
0xE24F0014
→ 0xE5901000
0xE24F0018
0xE5902000
0xE0823001
0xE24F0020
0xE5803000

**#3**

Instruction	STEP	Comment
LDR R1,[R0]	EXECUTE	Read operand from M[R0] to R1

# Case execution of program



CONTENT	ADDRESS	LABEL
0x40	0x20	STEVE1
0	0x21	
0	0x22	
0	0x23	
0x10	0x24	STEVE2
0	0x25	
0	0x26	
0	0x27	
0	0x28	REZ
0	0x29	
0	0x2A	
0	0x2B	
0x14	0x2C	ADR R0,STEVE1
0x10	0x2D	
0x1F	0x2E	
0xE5	0x2F	
	0x30	LDR R1,[R0]
	...	
	0x34...	ADR R0,STEVE2
	0x38...	LDR R2,[R0]
	0x3C...	ADD R3,R2,R1
	0x40...	ADR R0,REZ
	0x44...	STR R3,[R0]
	...	

PROGRAM	
ADR	R0,STEVE1
LDR	R1,[R0]
ADR	R0,STEVE2
LDR	R2,[R0]
ADD	R3,R1,R2
ADR	R0,REZ
STR	R3,[R0]

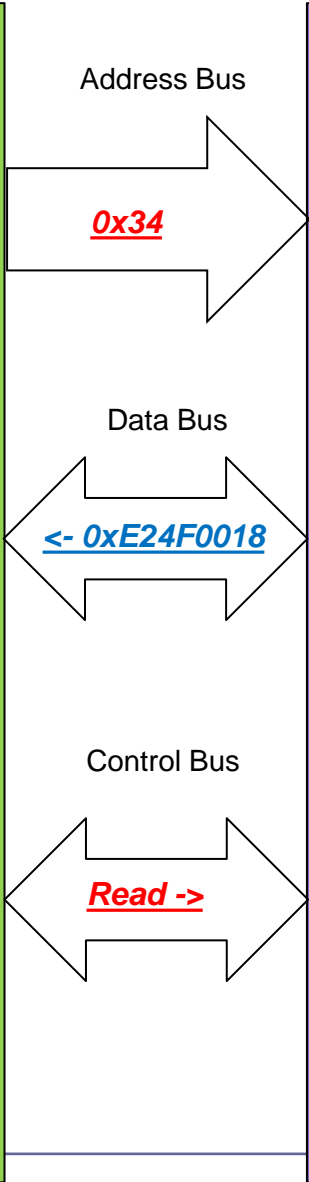
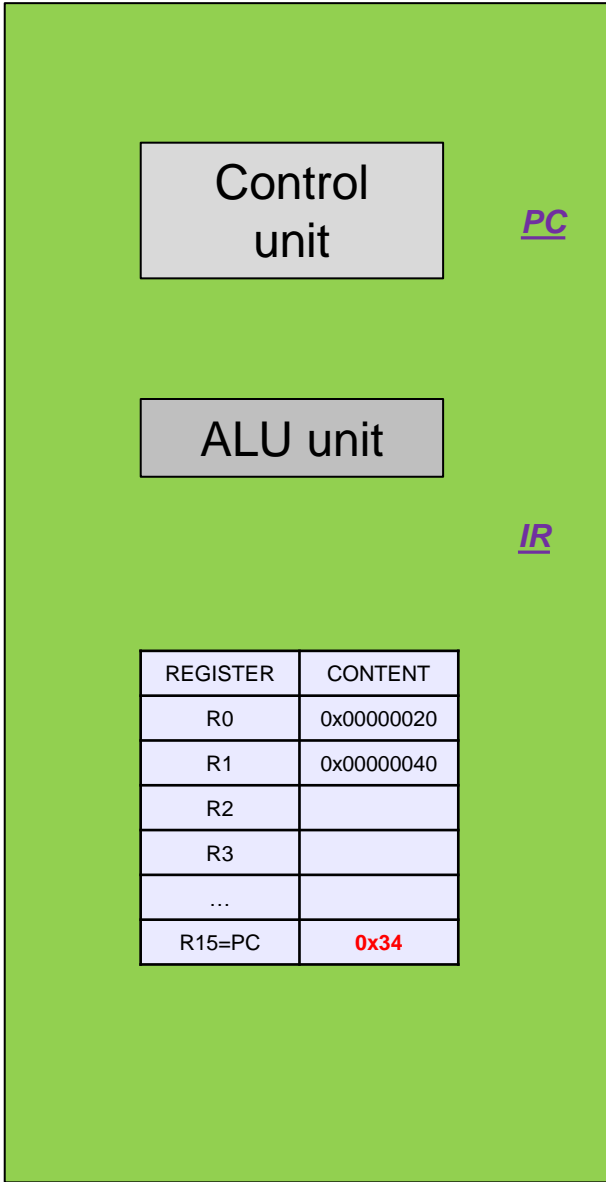
Machine lang
0xE24F0014
0xE5901000
0xE24F0018
0xE5902000
0xE0823001
0xE24F0020
0xE5803000

**#4**



Instruction	STEP	Comment
ADR R0,STEV2	FETCH	Read 3. instruction

# Case execution of program



CONTENT	ADDRESS	LABEL
0x40	0x20	STEV1
0	0x21	
0	0x22	
0	0x23	
0x10	0x24	STEV2
0	0x25	
0	0x26	
0	0x27	
0	0x28	REZ
0	0x29	
0	0x2A	
0	0x2B	
0x14	0x2C	ADR R0,STEV1
0x00	0x2D	
0x4F	0x2E	
0xE2	0x2F	
	0x30	LDR R1,[R0]
	...	
	<b>0x34...</b>	<b>ADR R0,STEV2</b>
	0x38...	LDR R2,[R0]
	0x3C...	ADD R3,R2,R1
	0x40...	ADR R0,REZ
	0x44...	STR R3,[R0]
	...	

PROGRAM	
	ADR R0,STEV1
	LDR R1,[R0]
➔	ADR R0,STEV2
	LDR R2,[R0]
	ADD R3,R1,R2
	ADR R0,REZ
	STR R3,[R0]

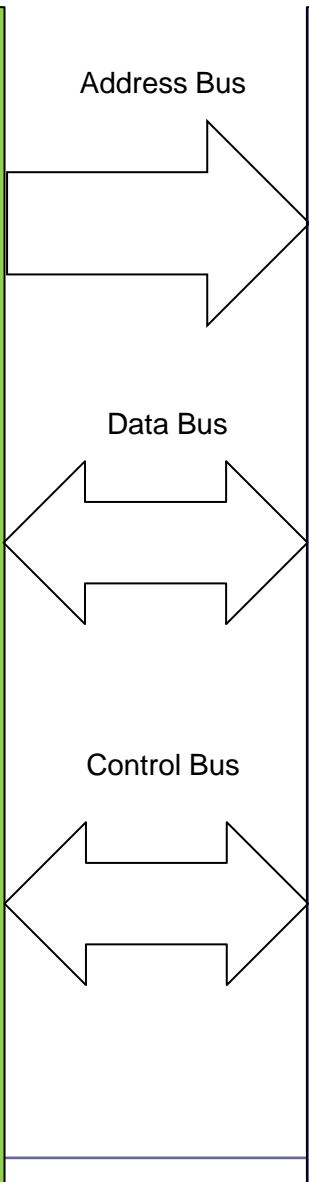
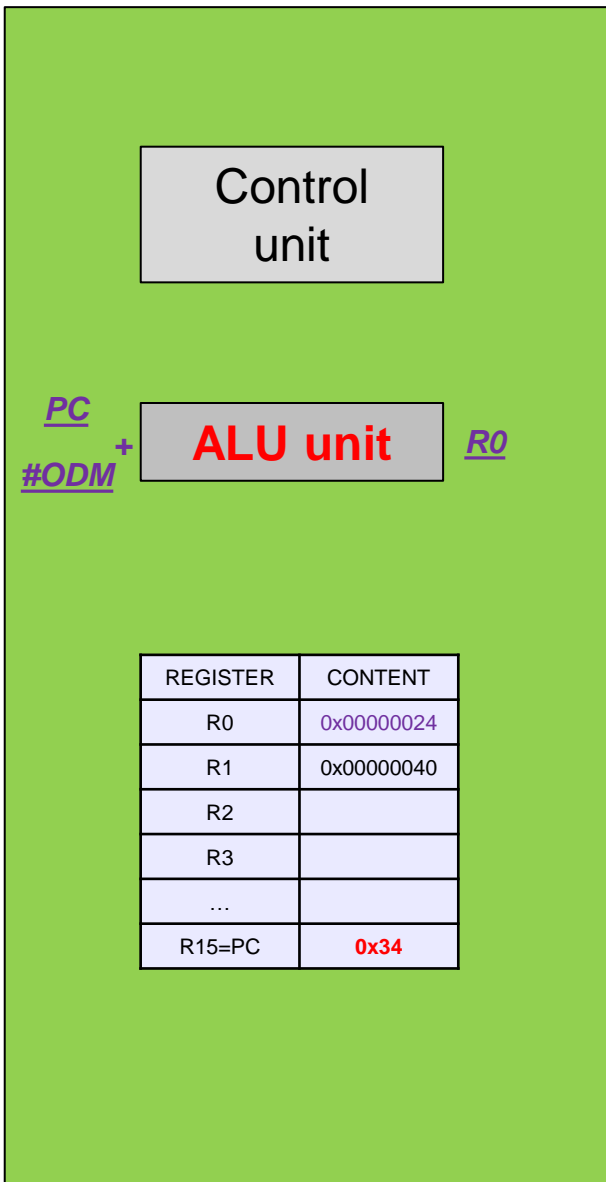
Machine lang	
	0xE24F0014
	0xE5901000
➔	0xE24F0018
	0xE5902000
	0xE0823001
	0xE24F0020
	0xE5803000

**#5**



Instruction	STEP	Comment
ADR R0,STEV2	EXECUTE	ALE: R0 <- PC +- ODMIK

# Case execution of program



CONTENT	ADDRESS	LABEL
0x40	0x20	STEV1
0	0x21	
0	0x22	
0	0x23	
0x10	0x24	STEV2
0	0x25	
0	0x26	
0	0x27	
0	0x28	REZ
0	0x29	
0	0x2A	
0	0x2B	
0x14	0x2C	ADR R0,STEV1
0x10	0x2D	
0x1F	0x2E	
0xE5	0x2F	
	0x30	LDR R1,[R0]
	...	
	0x34...	ADR R0,STEV2
	0x38...	LDR R2,[R0]
	0x3C...	ADD R3,R2,R1
	0x40...	ADR R0,REZ
	0x44...	STR R3,[R0]
	...	

PROGRAM	
ADR R0,STEV1	
LDR R1,[R0]	
ADR R0,STEV2	
LDR R2,[R0]	
ADD R3,R1,R2	
ADR R0,REZ	
STR R3,[R0]	

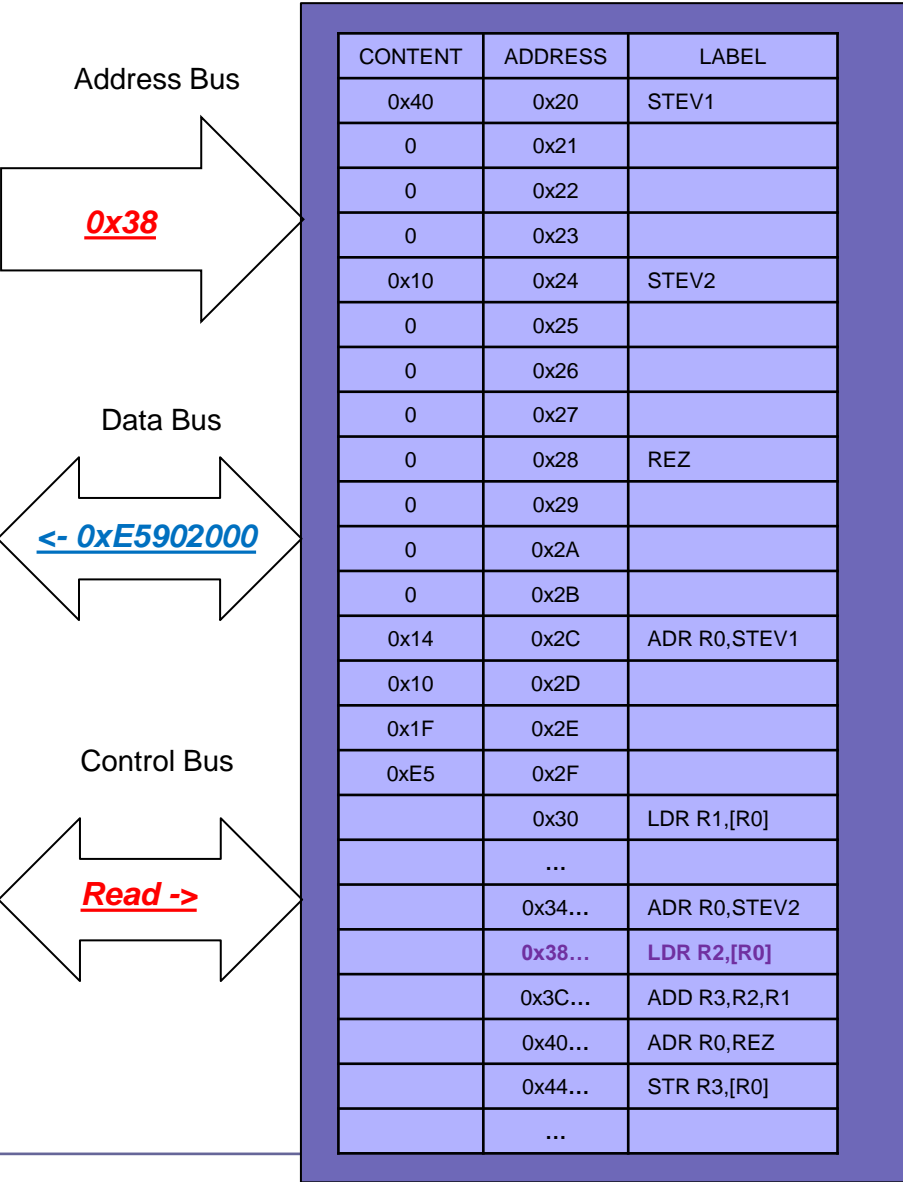
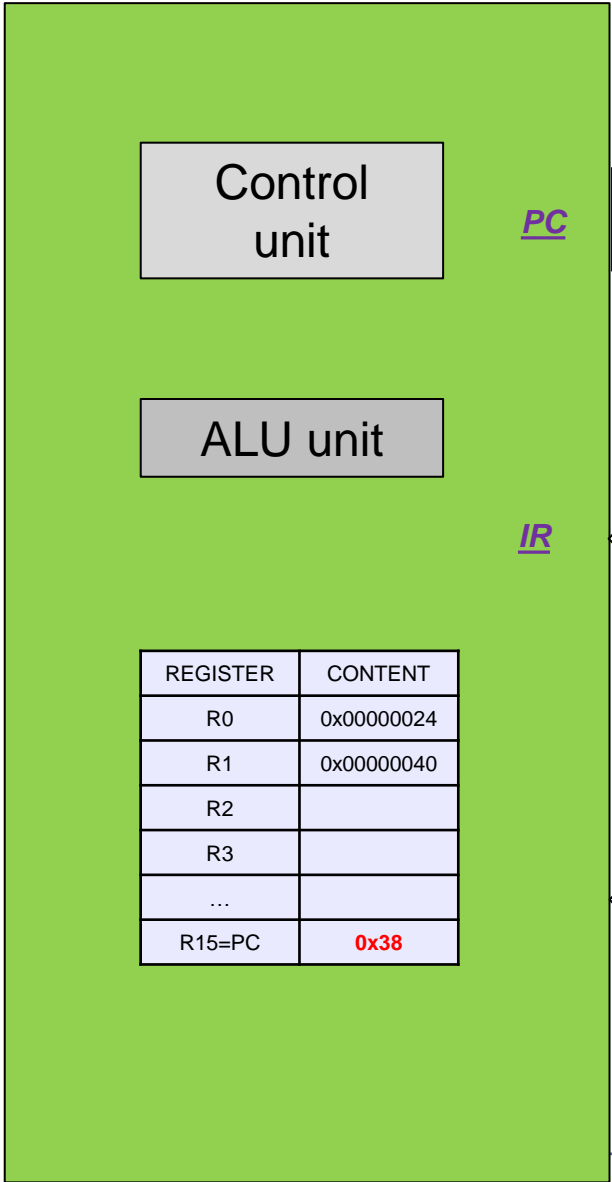
Machine lang
0xE24F0014
0xE5901000
0xE24F0018
0xE5902000
0xE0823001
0xE24F0020
0xE5803000

**#6**



Instruction	STEP	Comment
LDR R2,[R0]	FETCH	Read 4. instruction

# Case execution of program



PROGRAM	
ADR	R0,STEV1
LDR	R1,[R0]
ADR	R0,STEV2
LDR	R2,[R0]
ADD	R3,R1,R2
ADR	R0,REZ
STR	R3,[R0]

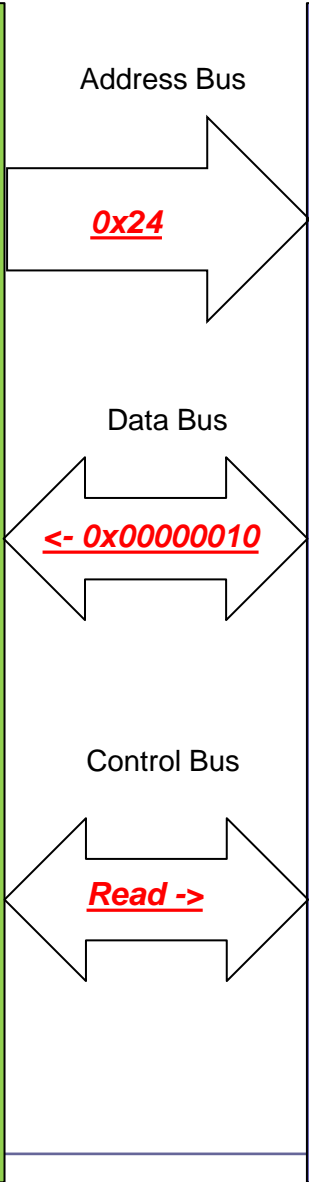
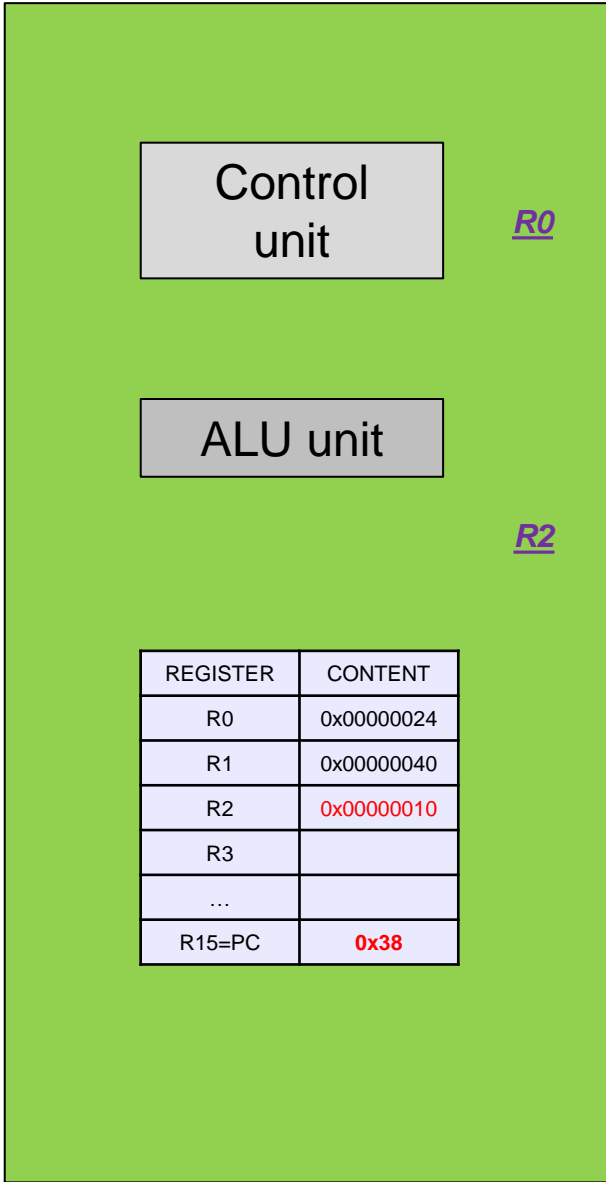
## Machine lang

- 0xE24F0014
- 0xE5901000
- 0xE24F0018
- 0xE5902000
- 0xE0823001
- 0xE24F0020
- 0xE5803000

**#7**

Instruction	STEP	Comment
LDR R2,[R0]	EXECUTE	Read operand from M[R0] to R1

# Case execution of program



CONTENT	ADDRESS	LABEL
0x40	0x20	STEVE1
0	0x21	
0	0x22	
0	0x23	
0x10	0x24	STEVE2
0	0x25	
0	0x26	
0	0x27	
0	0x28	REZ
0	0x29	
0	0x2A	
0	0x2B	
0x14	0x2C	ADR R0,STEVE1
0x10	0x2D	
0x1F	0x2E	
0xE5	0x2F	
	0x30	LDR R1,[R0]
	...	
	0x34...	ADR R0,STEVE2
	0x38...	LDR R2,[R0]
	0x3C...	ADD R3,R2,R1
	0x40...	ADR R0,REZ
	0x44...	STR R3,[R0]
	...	

PROGRAM	
ADR	R0,STEVE1
LDR	R1,[R0]
ADR	R0,STEVE2
LDR	R2,[R0]
ADD	R3,R1,R2
ADR	R0,REZ
STR	R3,[R0]

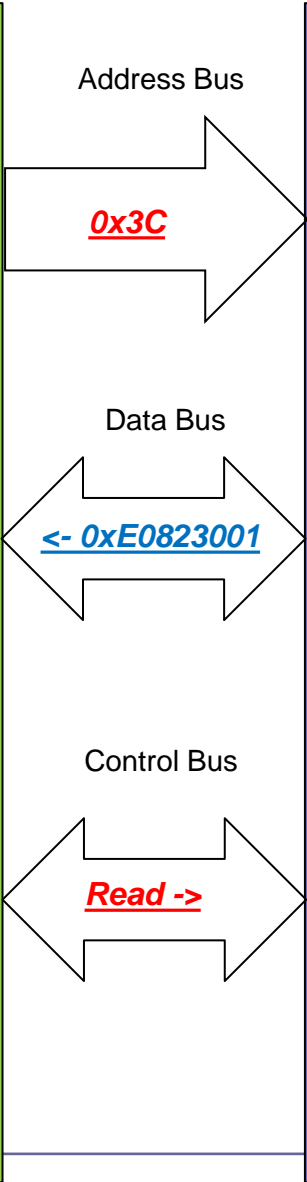
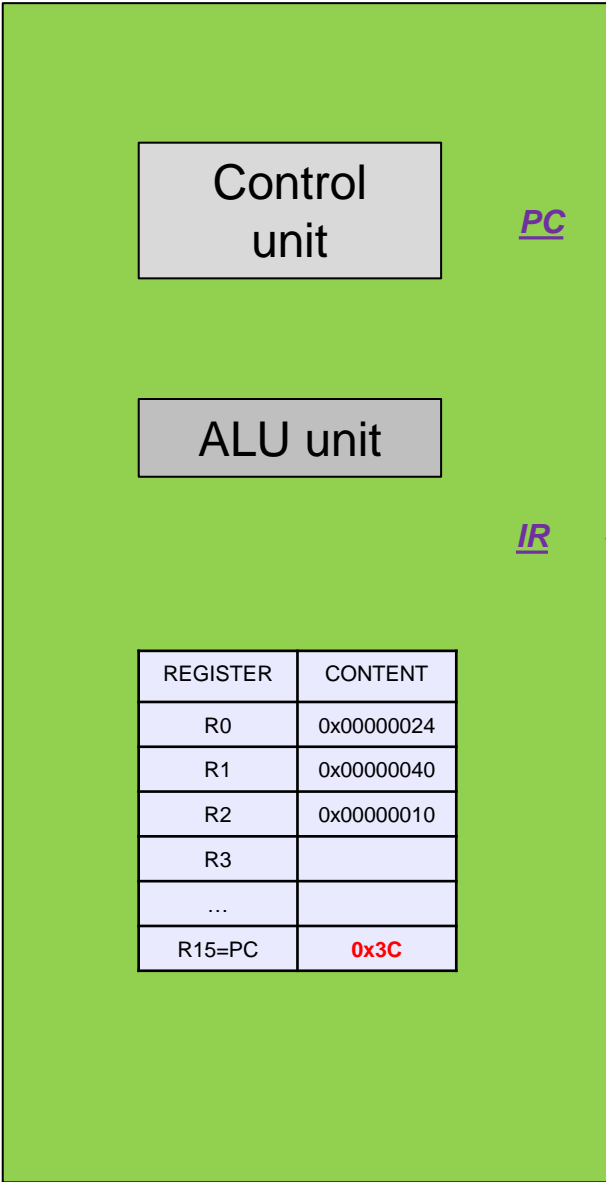
Machine lang
0xE24F0014
0xE5901000
0xE24F0018
0xE5902000
0xE0823001
0xE24F0020
0xE5803000

#8



Instruction	STEP	Comment
ADD R3,R2,R1	FETCH	Read 5. instruction

# Case execution of program



CONTENT	ADDRESS	LABEL
0x40	0x20	STEV1
0	0x21	
0	0x22	
0	0x23	
0x10	0x24	STEV2
0	0x25	
0	0x26	
0	0x27	
0	0x28	REZ
0	0x29	
0	0x2A	
0	0x2B	
0x14	0x2C	ADR R0,STEV1
0x10	0x2D	
0x1F	0x2E	
0xE5	0x2F	
	0x30	LDR R1,[R0]
	...	
	0x34...	ADR R0,STEV2
	0x38...	LDR R2,[R0]
	<b>0x3C...</b>	<b>ADD R3,R2,R1</b>
	0x40...	ADR R0,REZ
	0x44...	STR R3,[R0]
	...	

PROGRAM	
ADR	R0,STEV1
LDR	R1,[R0]
ADR	R0,STEV2
LDR	R2,[R0]
ADD	R3,R1,R2
ADR	R0,REZ
STR	R3,[R0]

Machine lang
0xE24F0014
0xE5901000
0xE24F0018
0xE5902000
<b>0xE0823001</b>
0xE24F0020
0xE5803000

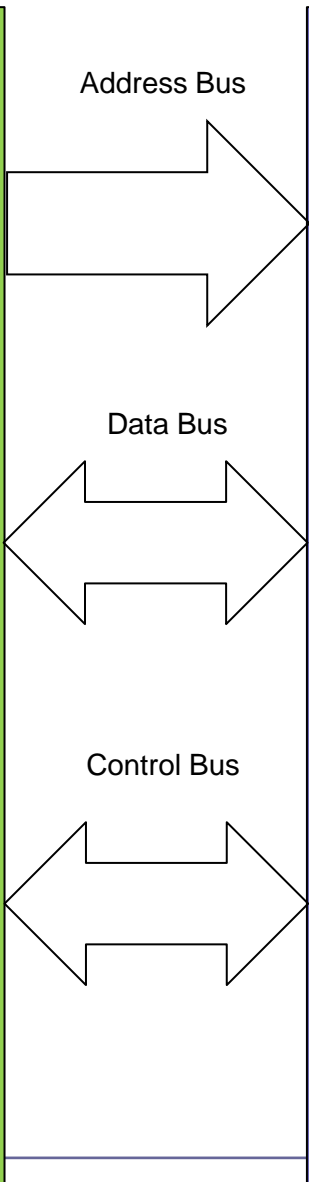
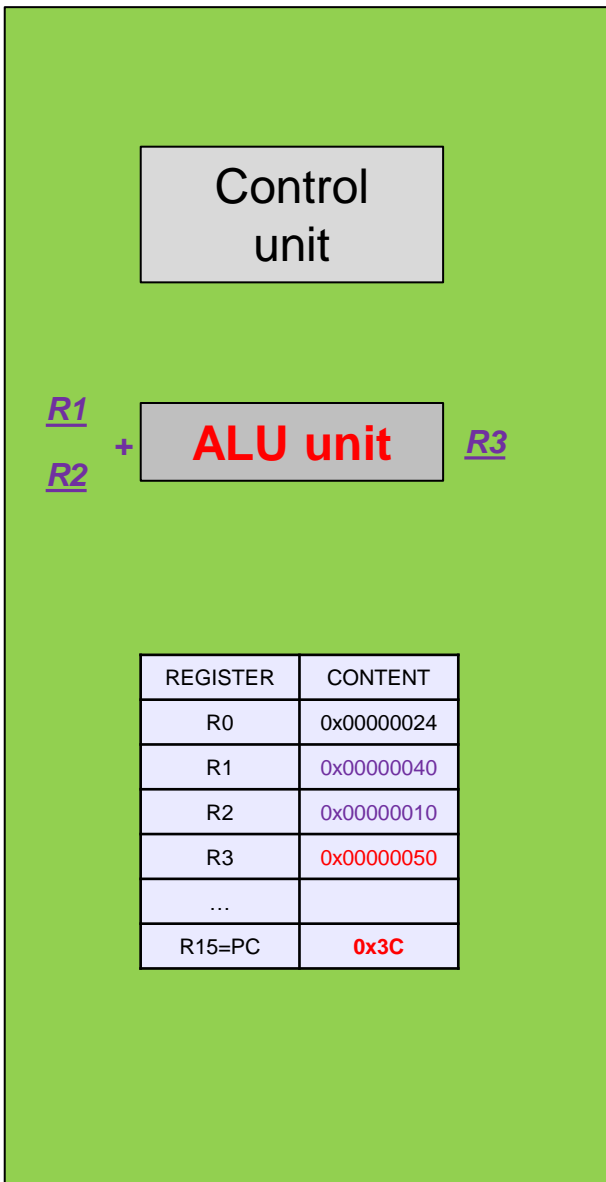
**#9**





Instruction	STEP	Comment
ADD R3,R2,R1	EXECUTE	ALE: R3 <- R2 + R1 (sum)

# Case execution of program



CONTENT	ADDRESS	LABEL
0x40	0x20	STEV1
0	0x21	
0	0x22	
0	0x23	
0x10	0x24	STEV2
0	0x25	
0	0x26	
0	0x27	
0	0x28	REZ
0	0x29	
0	0x2A	
0	0x2B	
0x14	0x2C	ADR R0,STEV1
0x10	0x2D	
0x1F	0x2E	
0xE5	0x2F	
	0x30	LDR R1,[R0]
	...	
	0x34...	ADR R0,STEV2
	0x38...	LDR R2,[R0]
	0x3C...	ADD R3,R2,R1
	0x40...	ADR R0,REZ
	0x44...	STR R3,[R0]
	...	

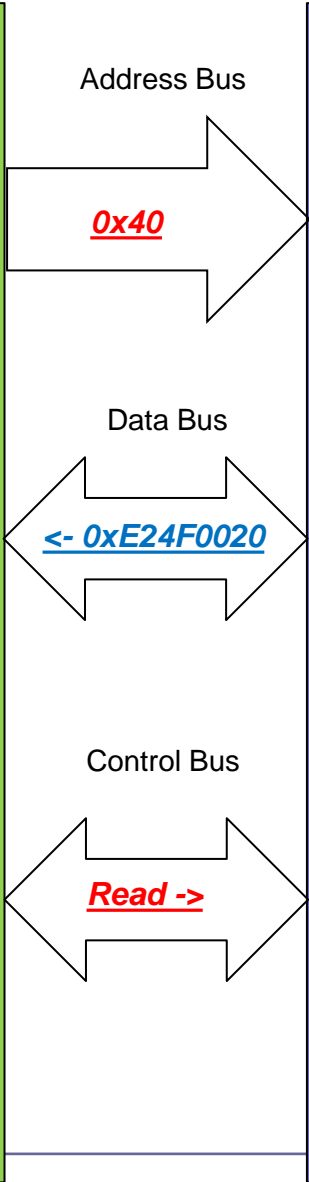
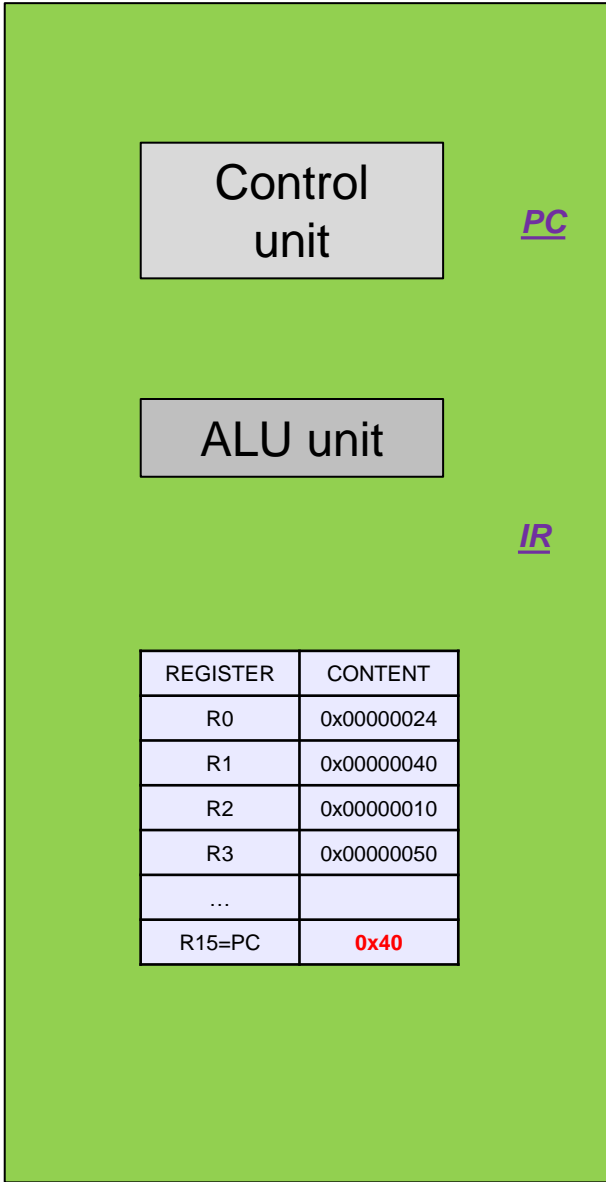
PROGRAM	
ADR	R0,STEV1
LDR	R1,[R0]
ADR	R0,STEV2
LDR	R2,[R0]
ADD	R3,R1,R2
ADR	R0,REZ
STR	R3,[R0]

Machine lang
0xE24F0014
0xE5901000
0xE24F0018
0xE5902000
0xE0823001
0xE24F0020
0xE5803000

**#10**

Instruction	STEP	Comment
ADR R0,REZ	FETCH	Read 6. instruction

# Case execution of program



CONTENT	ADDRESS	LABEL
0x40	0x20	STEV1
0	0x21	
0	0x22	
0	0x23	
0x10	0x24	STEV2
0	0x25	
0	0x26	
0	0x27	
0	0x28	REZ
0	0x29	
0	0x2A	
0	0x2B	
0x14	0x2C	ADR R0,STEV1
0x00	0x2D	
0x4F	0x2E	
0xE2	0x2F	
	0x30	LDR R1,[R0]
	...	
	0x34...	ADR R0,STEV2
	0x38...	LDR R2,[R0]
	0x3C...	ADD R3,R2,R1
	<b>0x40...</b>	<b>ADR R0,REZ</b>
	0x44...	STR R3,[R0]
	...	

PROGRAM	
ADR	R0,STEV1
LDR	R1,[R0]
ADR	R0,STEV2
LDR	R2,[R0]
ADD	R3,R1,R2
ADR	R0,REZ
STR	R3,[R0]

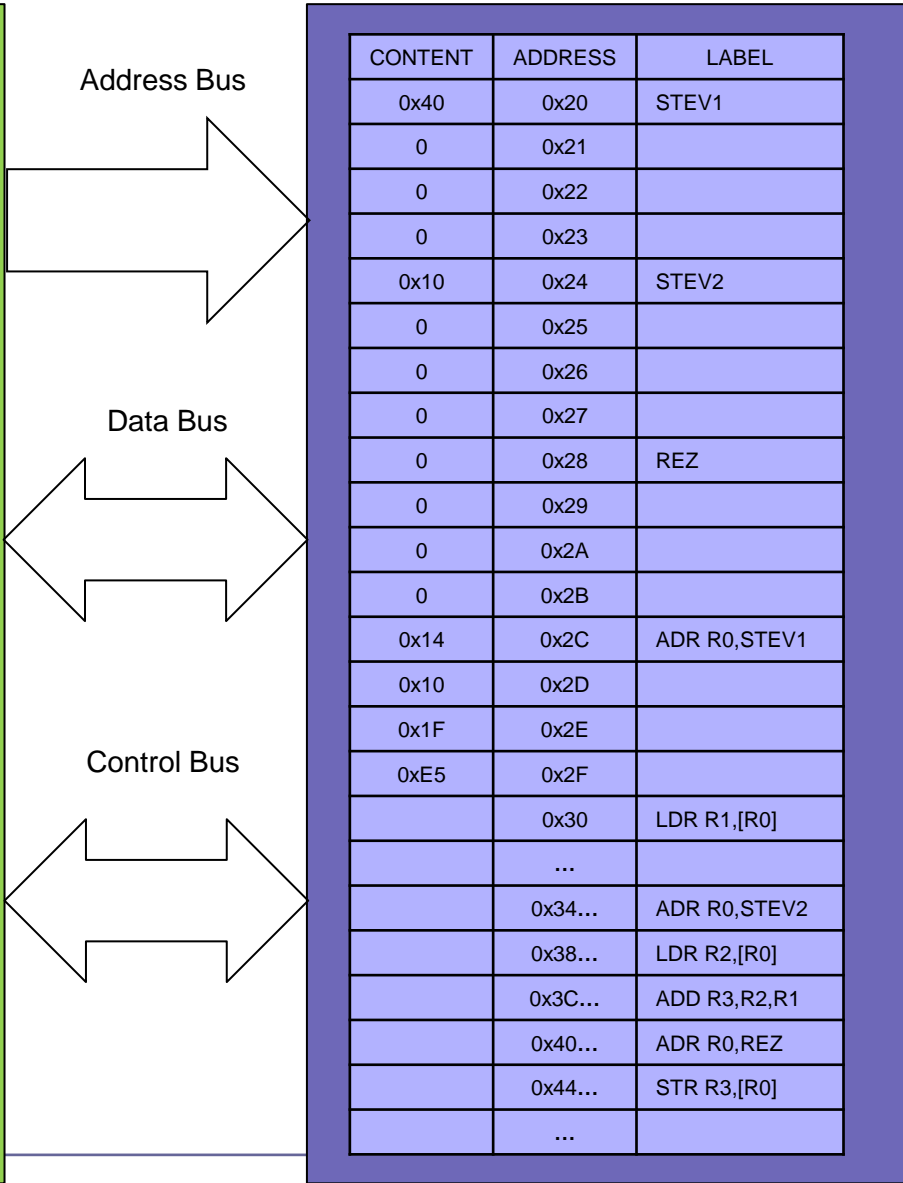
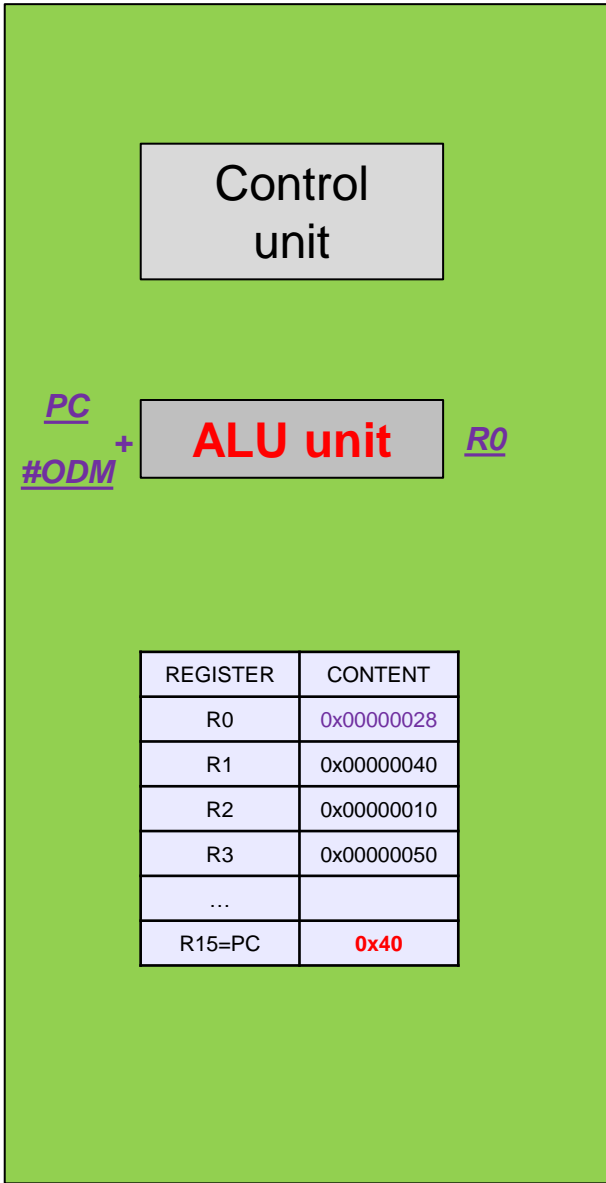
Machine lang
0xE24F0014
0xE5901000
0xE24F0018
0xE5902000
0xE0823001
<b>0xE24F0020</b>
0xE5803000

**#11**



Instruction	STEP	Comment
ADR R0,REZ	EXECUTE	ALE: R0 <- PC +- ODMIK

# Case execution of program



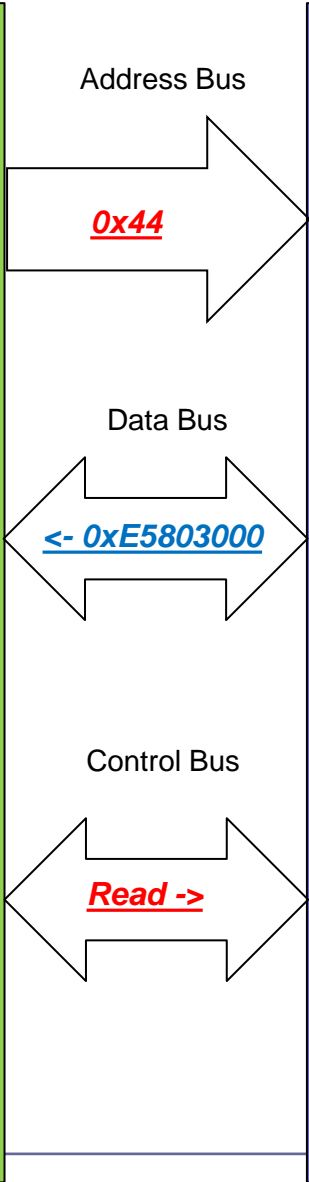
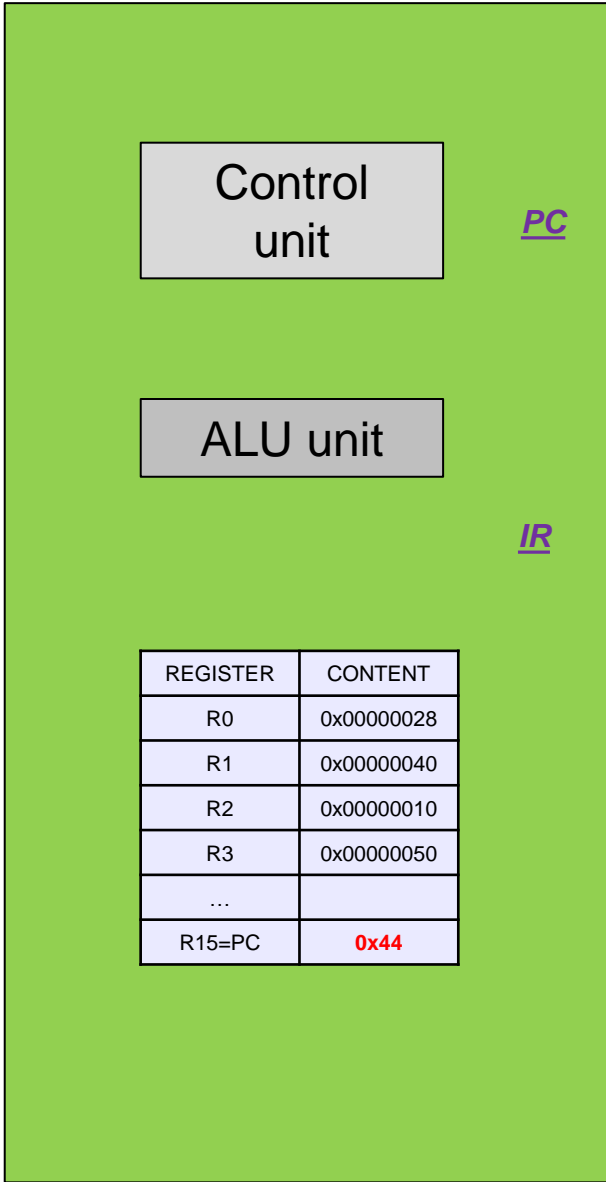
PROGRAM	
ADR	R0,STEV1
LDR	R1,[R0]
ADR	R0,STEV2
LDR	R2,[R0]
ADD	R3,R1,R2
ADR	R0,REZ
STR	R3,[R0]

Machine lang	
0xE24F0014	
0xE5901000	
0xE24F0018	
0xE5902000	
0xE0823001	
0xE24F0020	
0xE5803000	

**#12**

Instruction	STEP	Comment
STR R3,[R0]	FETCH	Read 7. instruction

# Case execution of program



CONTENT	ADDRESS	LABEL
0x40	0x20	STEV1
0	0x21	
0	0x22	
0	0x23	
0x10	0x24	STEV2
0	0x25	
0	0x26	
0	0x27	
0	0x28	REZ
0	0x29	
0	0x2A	
0	0x2B	
0x14	0x2C	ADR R0,STEV1
0x10	0x2D	
0x1F	0x2E	
0xE5	0x2F	
	0x30	LDR R1,[R0]
	...	
	0x34...	ADR R0,STEV2
	0x38...	LDR R2,[R0]
	0x3C...	ADD R3,R2,R1
	0x40...	ADR R0,REZ
	<b>0x44...</b>	<b>STR R3,[R0]</b>
	...	

PROGRAM	
ADR	R0,STEV1
LDR	R1,[R0]
ADR	R0,STEV2
LDR	R2,[R0]
ADD	R3,R1,R2
ADR	R0,REZ
STR	R3,[R0]

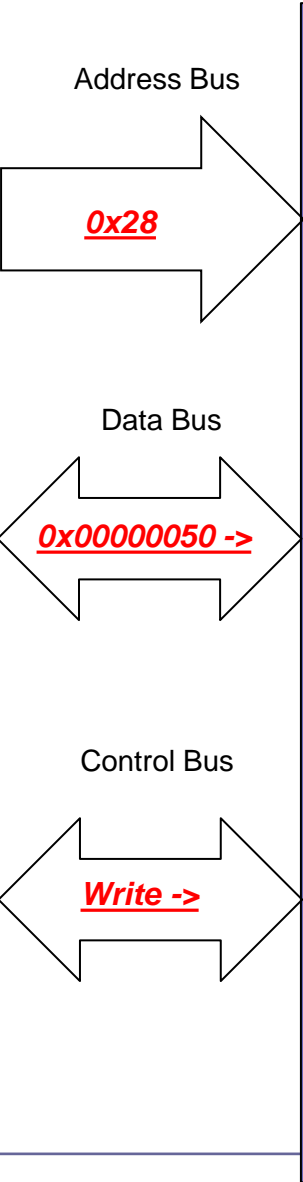
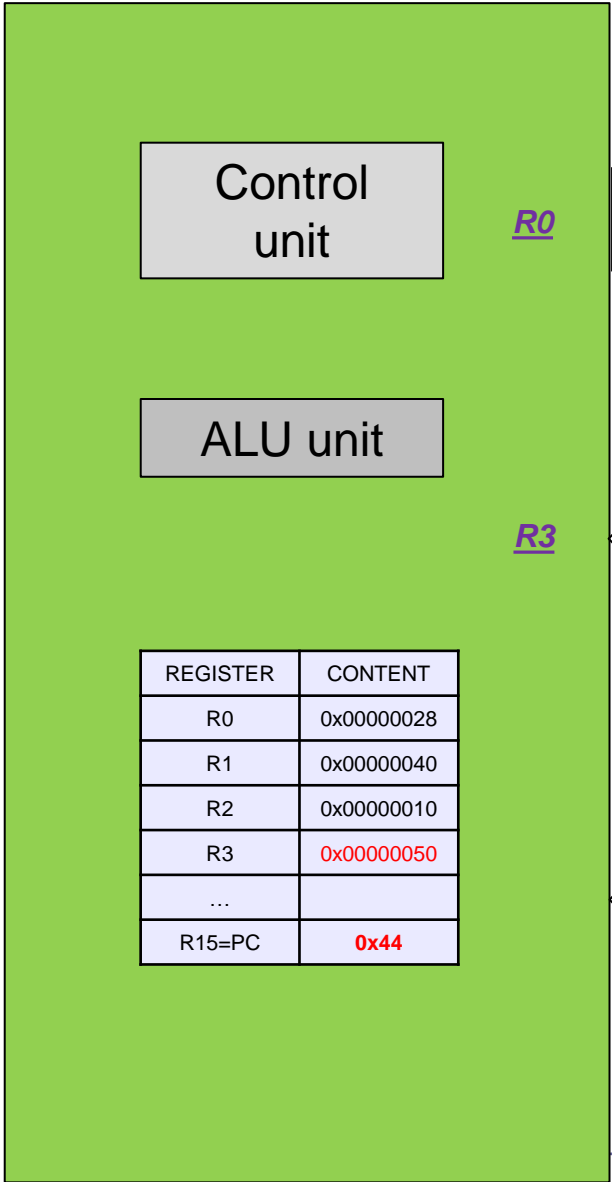
Machine lang
0xE24F0014
0xE5901000
0xE24F0018
0xE5902000
0xE0823001
0xE24F0020
0xE5803000

**#13**



Instruction	STEP	Comment
STR R3,REZ	EXECUTE	Store R3 to M[REZ]

# Case execution of program



CONTENT	ADDRESS	LABEL
0x40	0x20	STEV1
0	0x21	
0	0x22	
0	0x23	
0x10	0x24	STEV2
0	0x25	
0	0x26	
0	0x27	
0x50	0x28	REZ
0	0x29	
0	0x2A	
0	0x2B	
0x14	0x2C	ADR R0,STEV1
0x10	0x2D	
0x1F	0x2E	
0xE5	0x2F	
	0x30	LDR R1,[R0]
	...	
	0x34...	ADR R0,STEV2
	0x38...	LDR R2,[R0]
	0x3C...	ADD R3,R2,R1
	0x40...	ADR R0,REZ
	0x44...	STR R3,[R0]
	...	

PROGRAM	
ADR	R0,STEV1
LDR	R1,[R0]
ADR	R0,STEV2
LDR	R2,[R0]
ADD	R3,R1,R2
ADR	R0,REZ
STR	R3,[R0]

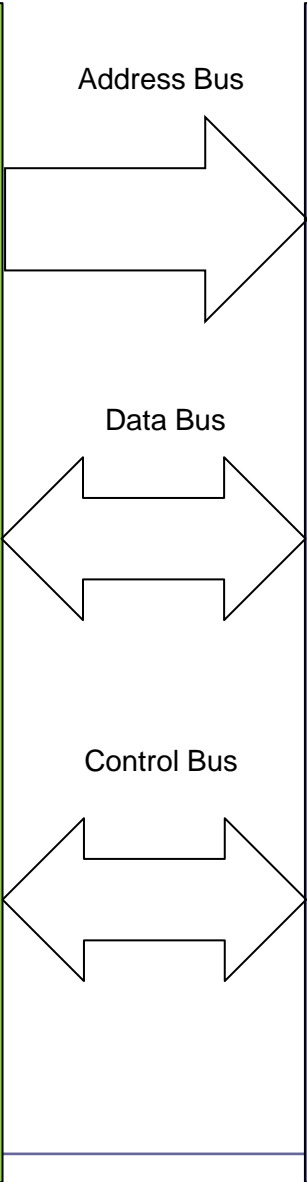
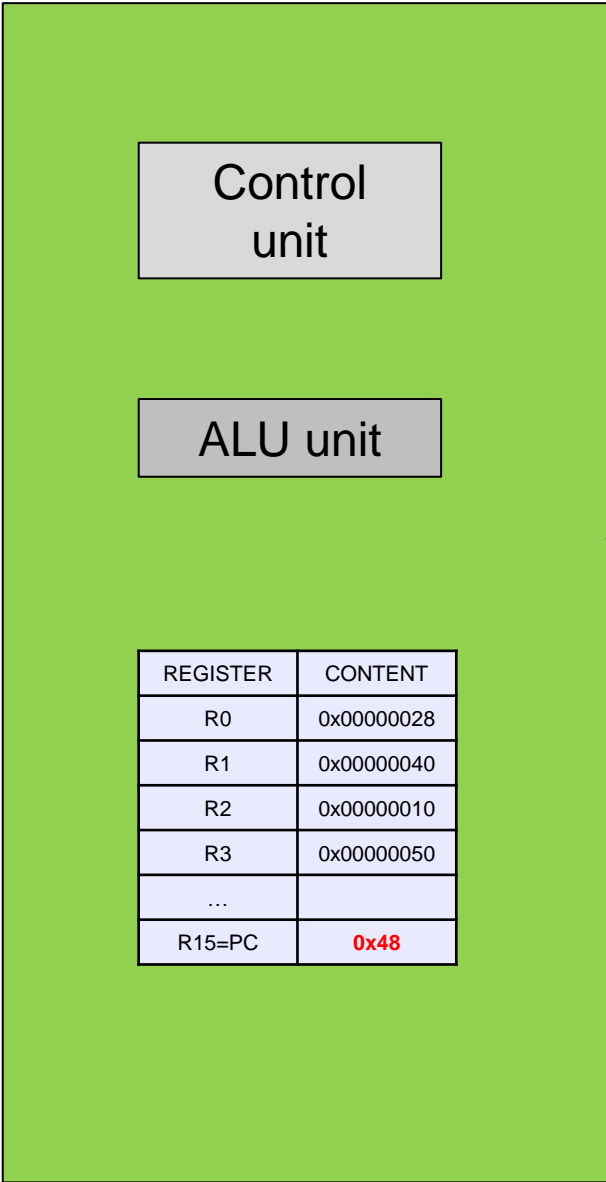
Machine lang
0xE24F0014
0xE5901000
0xE24F0018
0xE5902000
0xE0823001
0xE24F0020
0xE5803000

#14



Instruction	STEP	Comment
?	FETCH	Final state ?

# Case execution of program



CONTENT	ADDRESS	LABEL
0x40	0x20	STEV1
0	0x21	
0	0x22	
0	0x23	
0x10	0x24	STEV2
0	0x25	
0	0x26	
0	0x27	
0x50	0x28	REZ
0	0x29	
0	0x2A	
0	0x2B	
0x14	0x2C	LDR R1,STEV1
0x10	0x2D	
0x1F	0x2E	
0xE5	0x2F	
0x14	0x30	LDR R2,STEV2
0x20	...	
0x1F	0x34...	
0xE5	0x38...	
0x01	0x3C...	ADD R3,R2,R1
	0x40...	
0x18	0x44...	STR R3,REZ
	<b>0x48...</b>	<b>???</b>

PROGRAM	
ADR	R0,STEV1
LDR	R1,[R0]
ADR	R0,STEV2
LDR	R2,[R0]
ADD	R3,R1,R2
ADR	R0,REZ
STR	R3,[R0]

Machine lang
0xE24F0014
0xE5901000
0xE24F0018
0xE5902000
0xE0823001
0xE24F0020
0xE5803000

➔ **#15**

<i>CPU</i>		<i>CPU</i>	<i>BUSes</i>			<i>MEMORY</i>
<i>Description</i>	<i>CPU</i>	<i>Description</i>	<i>Address</i>	<i>Data</i>	<i>Control</i>	<i>Description</i>
ADR R0,STEV1	FETCH					
	EXECUTE					
LDR R1,[R0]	FETCH					
	EXECUTE					
ADR R0,STEV2	FETCH					
	EXECUTE					
LDR R2,[R0]	FETCH					
	EXECUTE					
ADD R3,R1,R2	FETCH					
	EXECUTE					
ADR R0,REZ	FETCH					
	EXECUTE					
STR R3,[R0]	FETCH					
	EXECUTE					