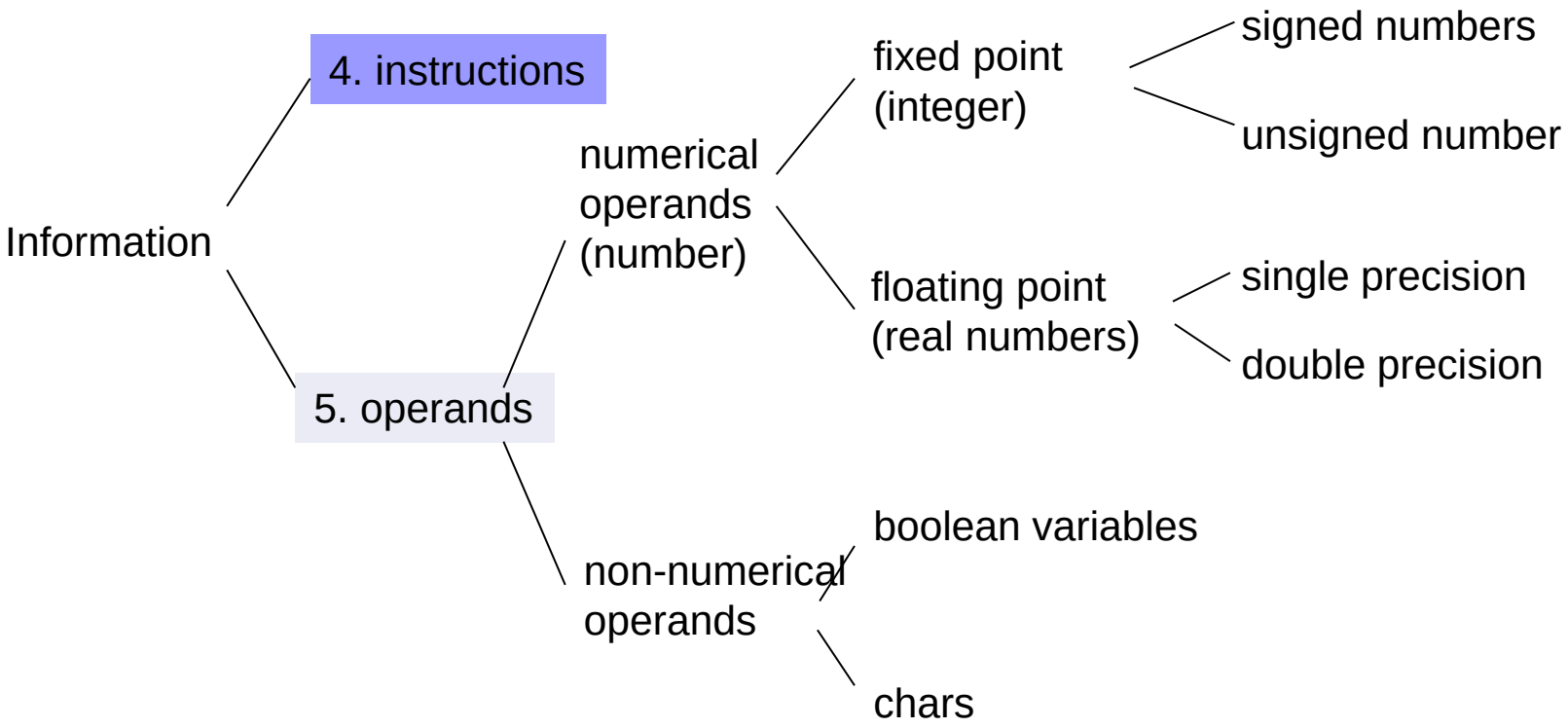
COMPUTER ARCHITECTURE

4 Instructions (machine, assembly)

Basic types of information on your computer



Instructions - content (Chapter 5 [Kodek]):

- General information about instructions
- Storage of the operands in the CPU
  - Accumulator
  - Stack
  - Register set
- Number of explicit operands in instruction
- Operands location and addressing modes
  - immediate addressing
  - direct addressing
  - indirect addressing

- Operations (types of instructions)
  - Arithmetic and logical operations (ALU operations)
  - Data transfer
  - Control operations
  - Floating-point operations
  - System operations
  - Input / Output Operations
- Instruction types and number of operands
  - Composed memory operands
  - Big-endian rule
  - Little-endian rule
  - Alignment problem
- Instruction format
- RISC – CISC computers

- Instructions = Machine instructions (= instructions of the typical machine language)

- Instruction set is important ⇒ Computer architecture

  ISA = Instruction Set Architecture

- Different computers -> different architectures -> different machine instructions

- The operation of von Neumann’s computers is completely determined by the instructions that the CPU fetches from the main memory.
- Those instructions are machine instructions (instructions of the typical machine language).
- By specifying a set of machine instructions we largely determine the computer architecture.
- That's why we talk about „instruction architecture“ (ISA – Instruction Set Architecture)

4.1 In general about instructions

- Instructions are executed by the CPU, there are two ways of executing instructions:
  - Using hard-wired logic
    - Fast execution (logic circuit directly performs hardware instructions)
    - It is difficult to change and add new instructions (new logic circuit in the CPU is needed $\Rightarrow$ new chip)
  - Microprogramming (MiMo model: elective course OR – Comp. Org.)
    - Slower execution (it is necessary to interpret to the microprogramming level - logical circuit performs microinstructions)
    - Easier to modify and add new instructions (change s needed in the microprogram only)

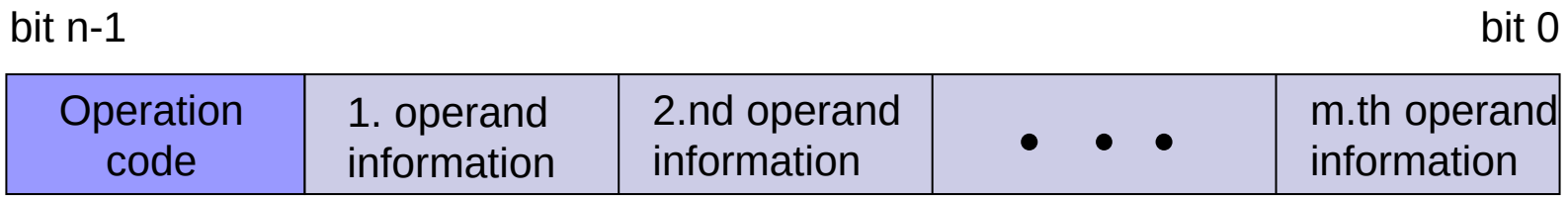
- Each instruction must contain information about two strictly separate types:
  - information on the **operation** to be executed
  - information on the **operands** on which to execute the operation
- Both types of information are determined with bits in fields into which the instruction is divided to - the length (number of bits) and the number of these fields largely differs for different computers.

- **Operation code** - the name of the field that contains information about the operation.
- Fields containing **information on the operands:**
  - Can contain operand
  - Or address information, where the operand is stored
- For some instructions, the information on the operands is already contained in the operation code.

- **Instruction format** – defines the division into fields for bough types of information, length of each field in bits and the bits meaning.

- What the instruction format looks like is dependent on:
  - Number of operations
  - Number of registers in the CPU
  - Memory address size
  - Memory word size
  - …

- Machine instruction format for length of *n* bits, with *m* explicitly defined operands:



Instruction format of length *n* - bits
with *m* - explicitly defined operands

- ARM9: Example of a 32-bit instruction (all instructions are 32-bit):

mov r_d, # imm

r_d = destination register

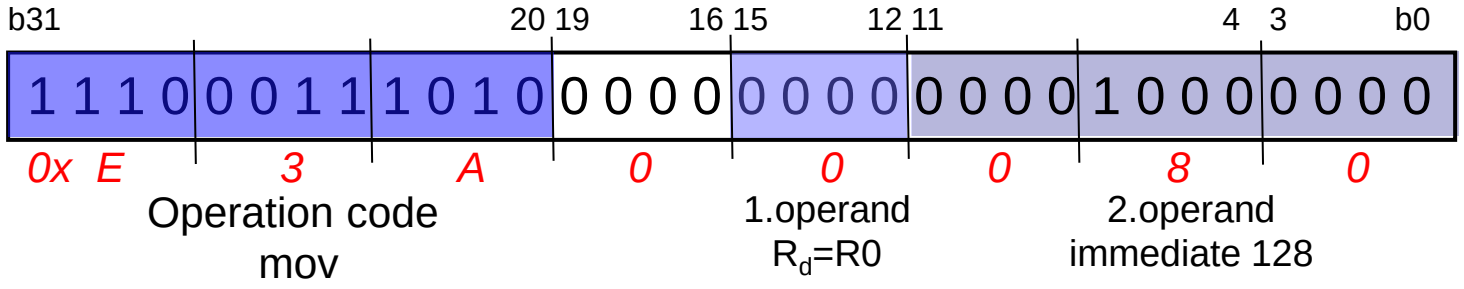
imm = immediate operand

Assembly Instruction:

mov r0, #128 **@ R0 ← 128=0x080**

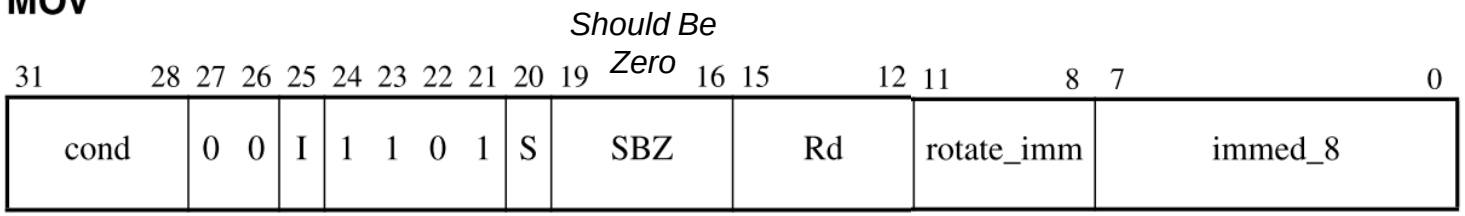
```
_START: mov   r0,#128
0x02C
        0xE3A00080
```

Machine instruction:



Instruction format (from ARM documentation) :

A4.1.35 MOV



- ARM9: example of an 32-bit instruction (all instructions are 32-bit):

add r_d, r_{s1}, r_{s2}

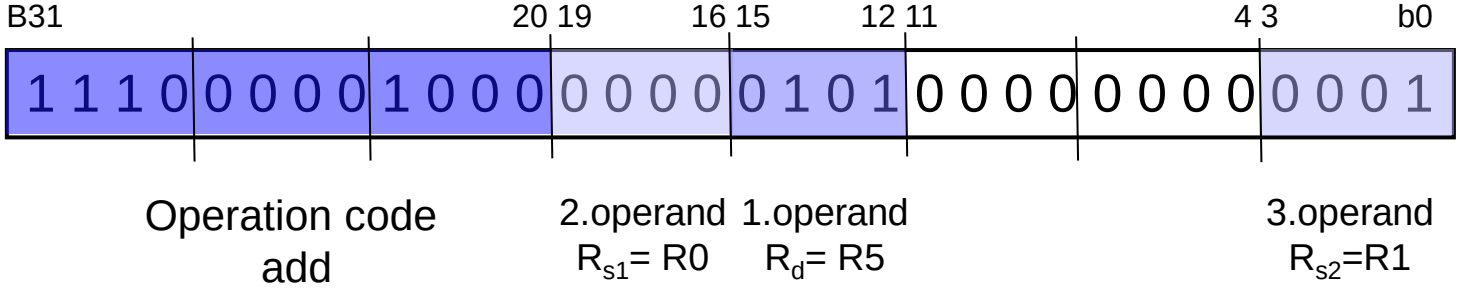
r_d = destination register

r_{sx} = source register

Assembly Instruction :

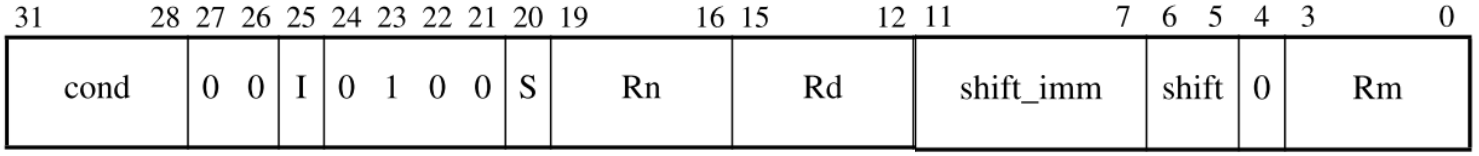
add r5, r0, r1 @ R5 ← R0 + R1

Machine instruction:



Instruction format (from documentation):

A4.1.3 ADD



- Mini MiMo : example of an 16-bit instruction (all instructions are 16-bit):

`add` r_d, r_d, r_s

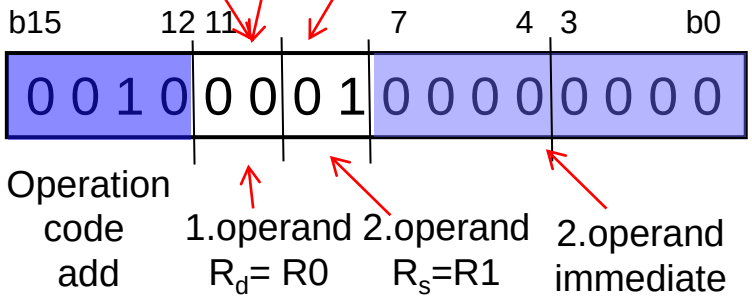
r_d = destination register

r_s = source register

Assembly Instruction :

`add r0, r0, r1` **@ R0 ← R0 + R1**

Machine instruction:



Instruction format (from documentation):

16-bitni ukazi - format:

op1	op2	Rd	Rs	immediate
2b	2b	2b	2b	8b

- The distinction between instructions and operations – execution of the same operation (eg. addition) can be achieved by a variety of instructions that have different formats (normally the information about operands is provided in different ways).

- The number of instructions is therefore usually greater than the actual number of operations.

- Example: ARM - arithmetic instructions (addition, subtraction):

```
add r0, r1, r2 @ r0 <- R1 + r2
adc r0, r1, r2 @ r0 <- R1 + r2 + C      (add with C)
sub r0, r1, r2 @ r0 <- r1 - r2
SBC r0, r1, r2 @ r0 <- r1 - r2 + C - 1  (-Not (C) = - (1-C) = C-1
RSB r0, r1, r2 @ r0 <- r2 - r1          (reverse subtract)
rsc r0, r1, r2 @ r0 <- r2 - r1 + C - 1  (Rev. sub -not (C))
```

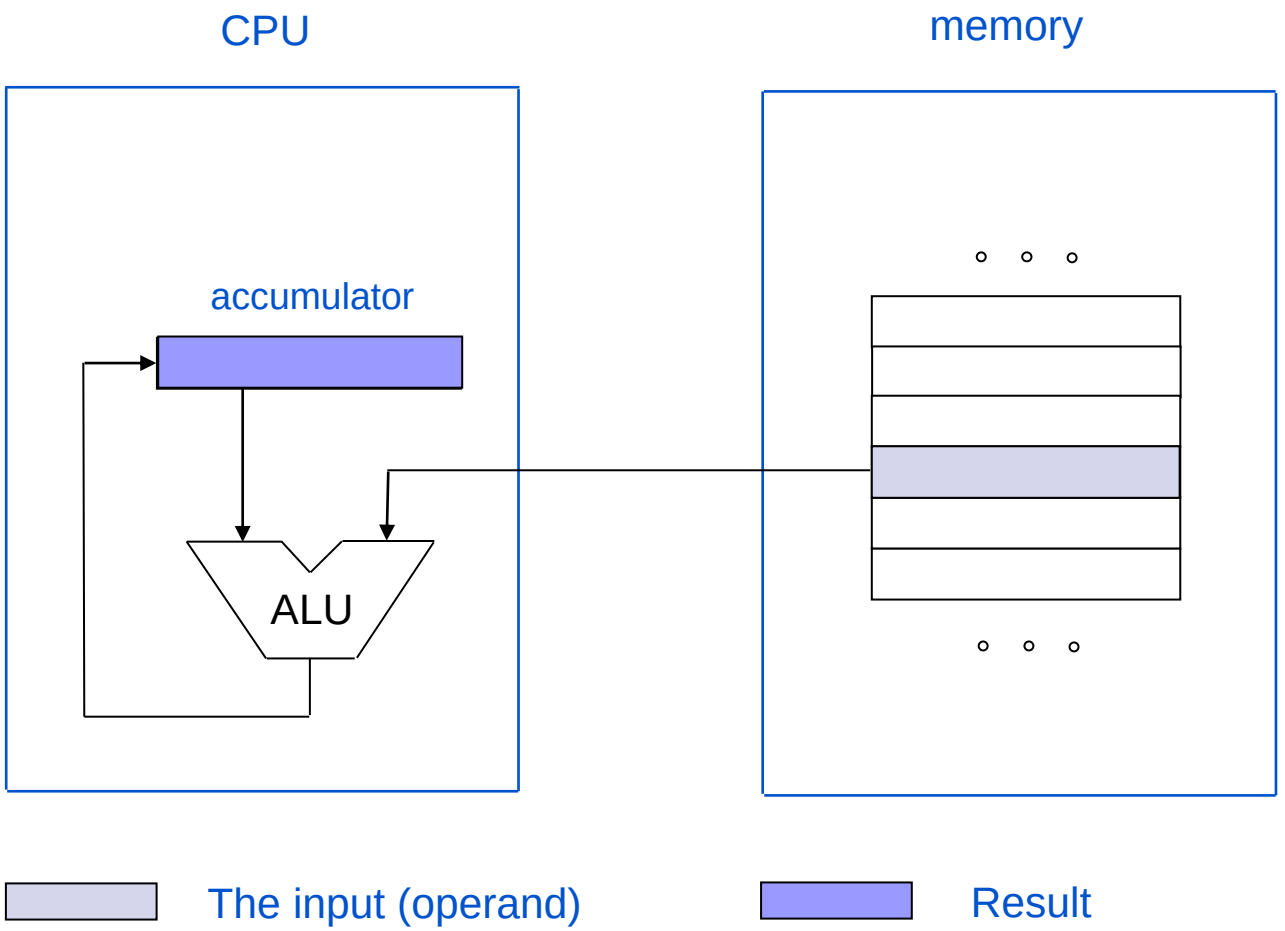

- Basic properties of instructions, according to which the instructions differ:
  - Storage of operands in the CPU
  - Number of explicit operands in instruction
  - Operands location and addressing modes
  - Operations
  - Type and length of the operands
- Decisions on each of the properties affect the structure and function of the computer.

4.2 Modes of saving operands in the CPU

- Property that the mostly affects how a user sees your computer.
- Notable are three ways to store operands in the CPU:
  - Accumulator (a single software accessible register in the CPU)
  - Stack (in the CPU)
  - Set of registers (set of software accessible registers in the CPU)

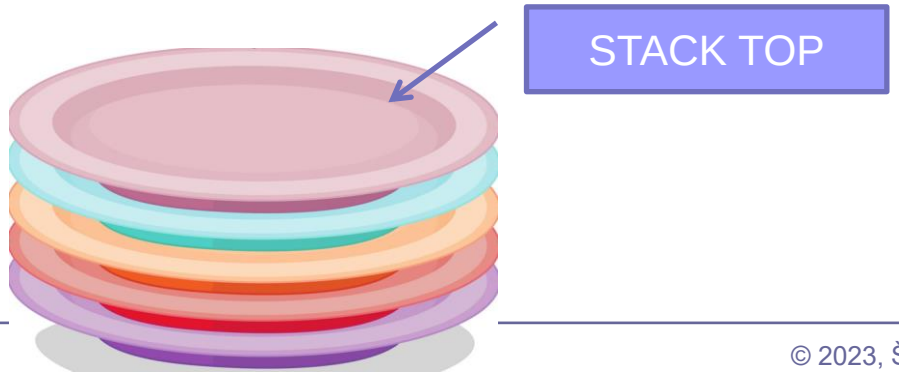
- **Accumulator:** Memory in the CPU is a single register, called accumulator.
  - Holds one operand
  - The oldest solution, its the simplest
  - For most instructions, one of the operands is in the accumulator, that's also where the result is stored.

The execution of an ALU instruction

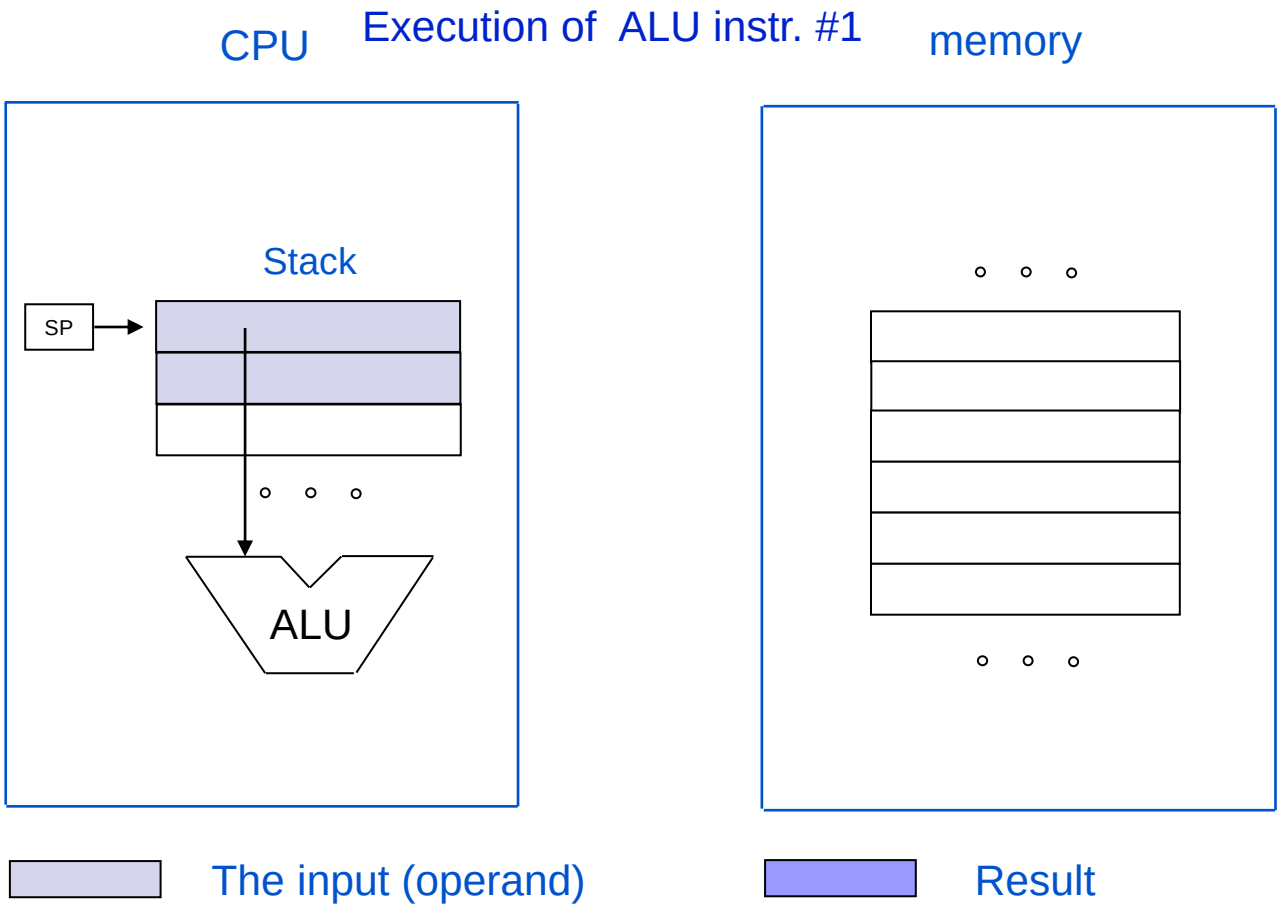


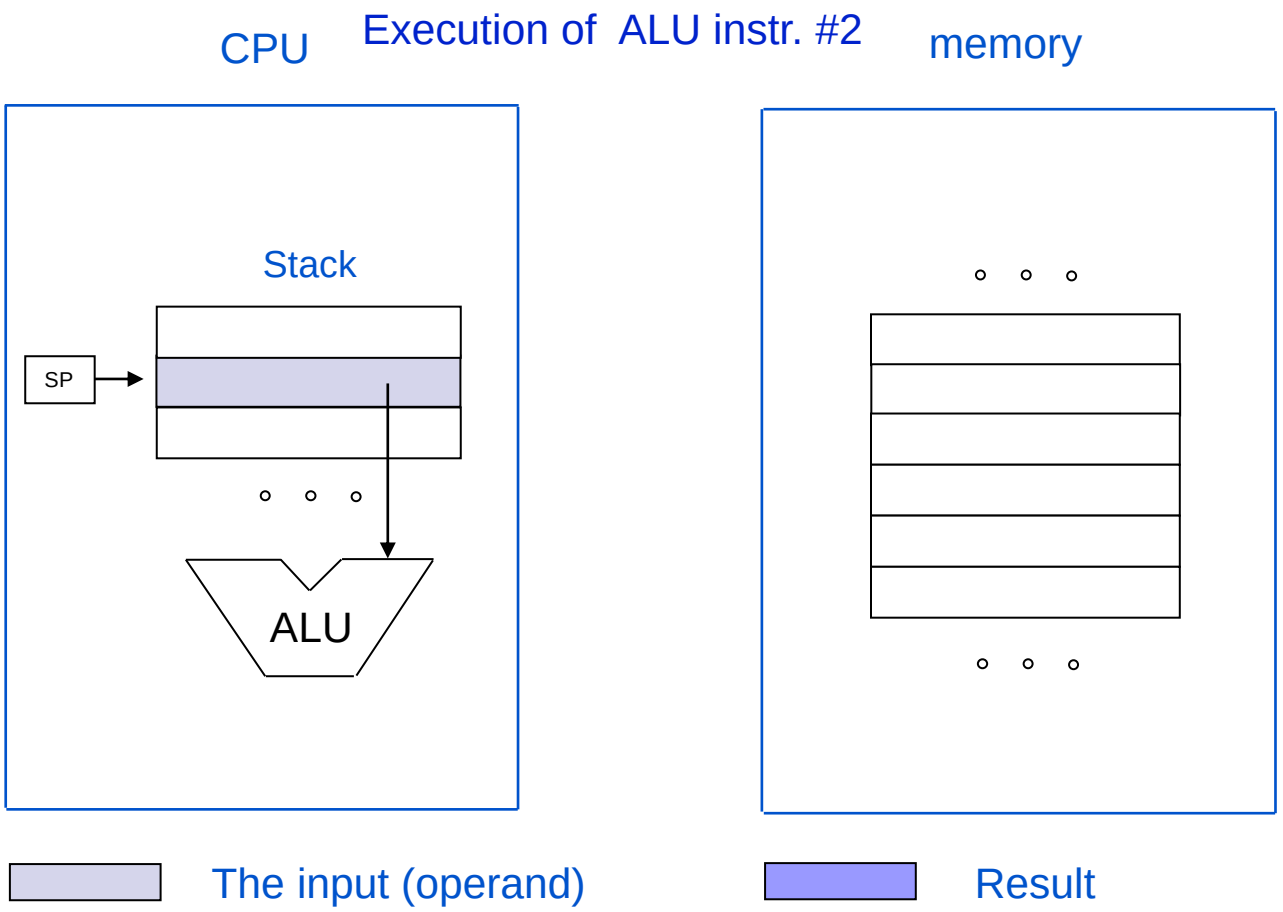
- Instruction to transfer an operand from the main memory into the accumulator or vice versa are LOAD and STORE.
- Because there is just one accumulator it’s not necessary to specify its address – shorter instructions.
- Simpler compilers, because there is no selection on the modes of saving operands.
- A lot of transfers between the CPU and main memory (more then for the other two solutions), because there is just one register.

- **Stack**: a simple way of extending CPU memory is to make it in a form of a stack.
  - In stack only the top location is accessible – top of the stack.
  - The operation of stacks is denoted as LIFO - Last In First Out.
  - Instructions to transfer operands from main memory to stack and vice-versa are PUSH and PULL or POP.
  - Great resemblance with concept of accumulator. All accumulator advantages also apply to stack, however we can also save more operands to the stack.

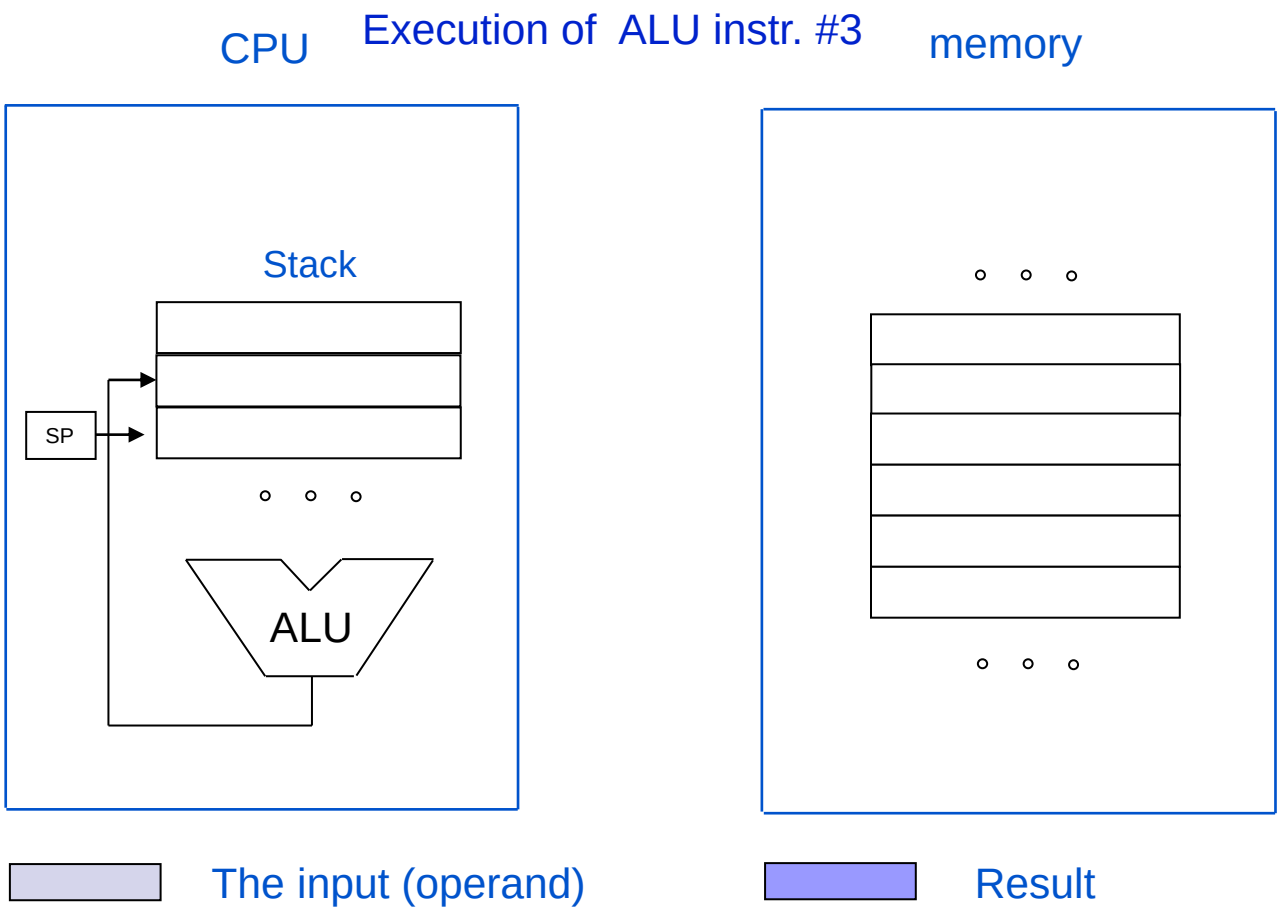


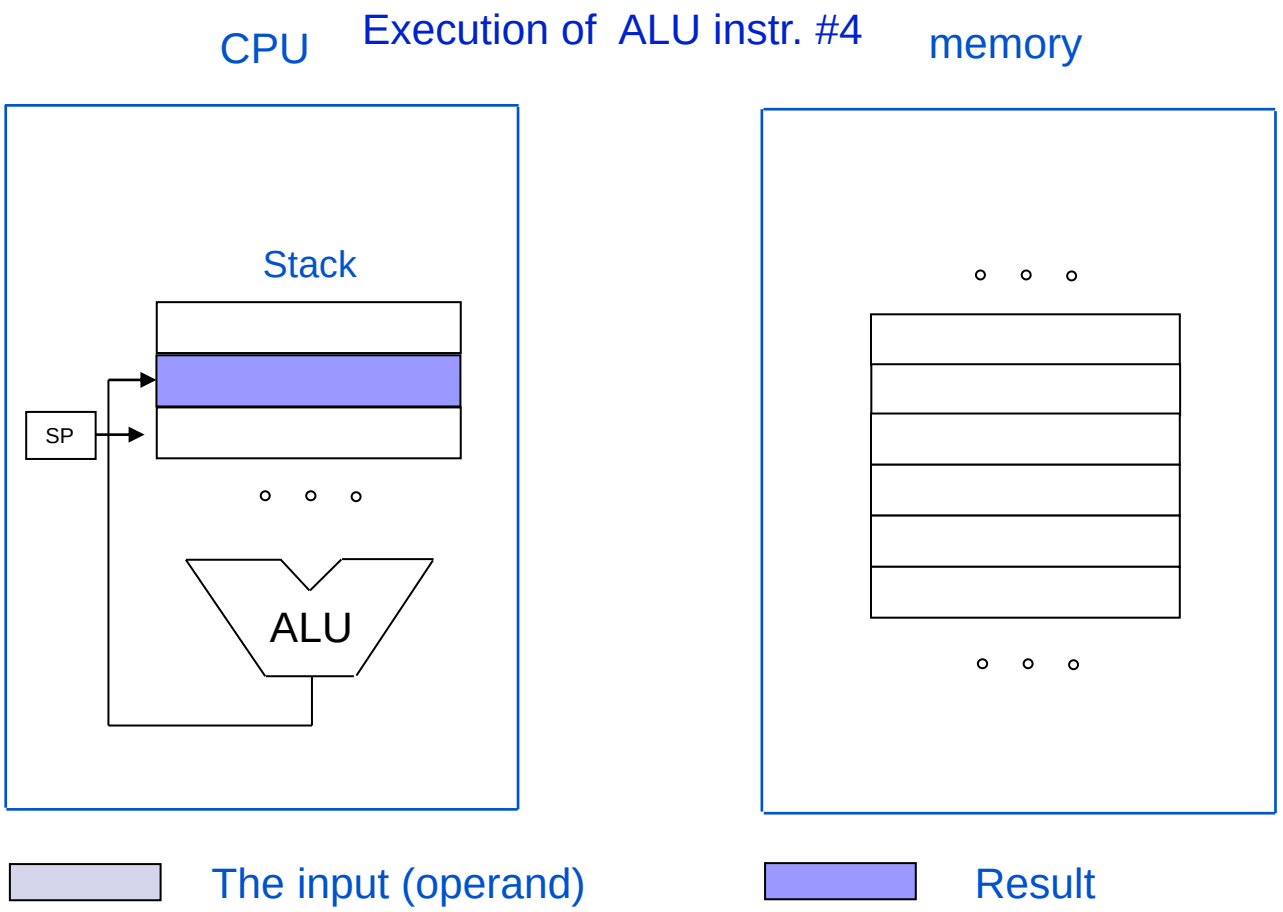
SP - address of the top of the stack in the CPU. In this case it points to the last occupied location



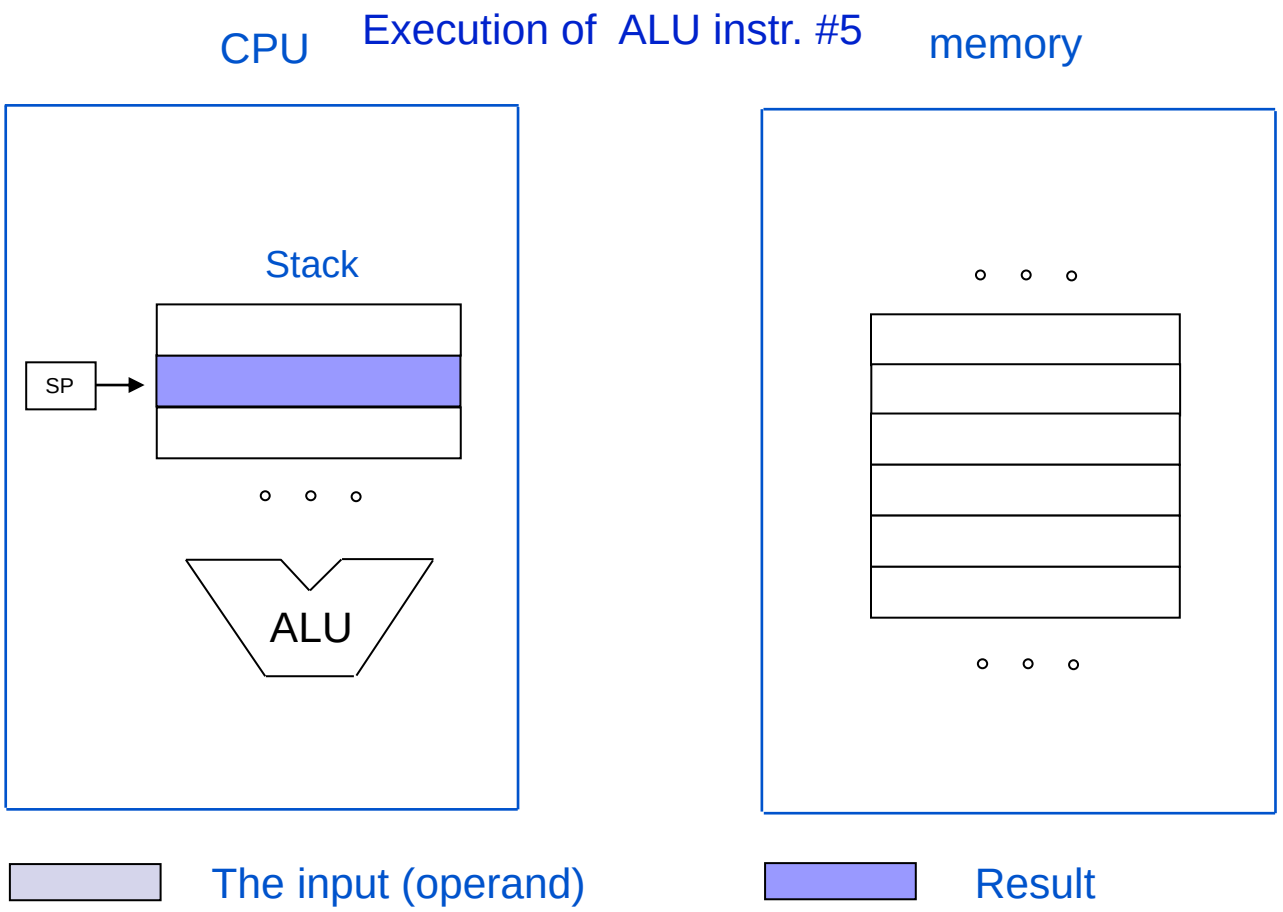


Modes of saving of the operands in the CPU - Stack





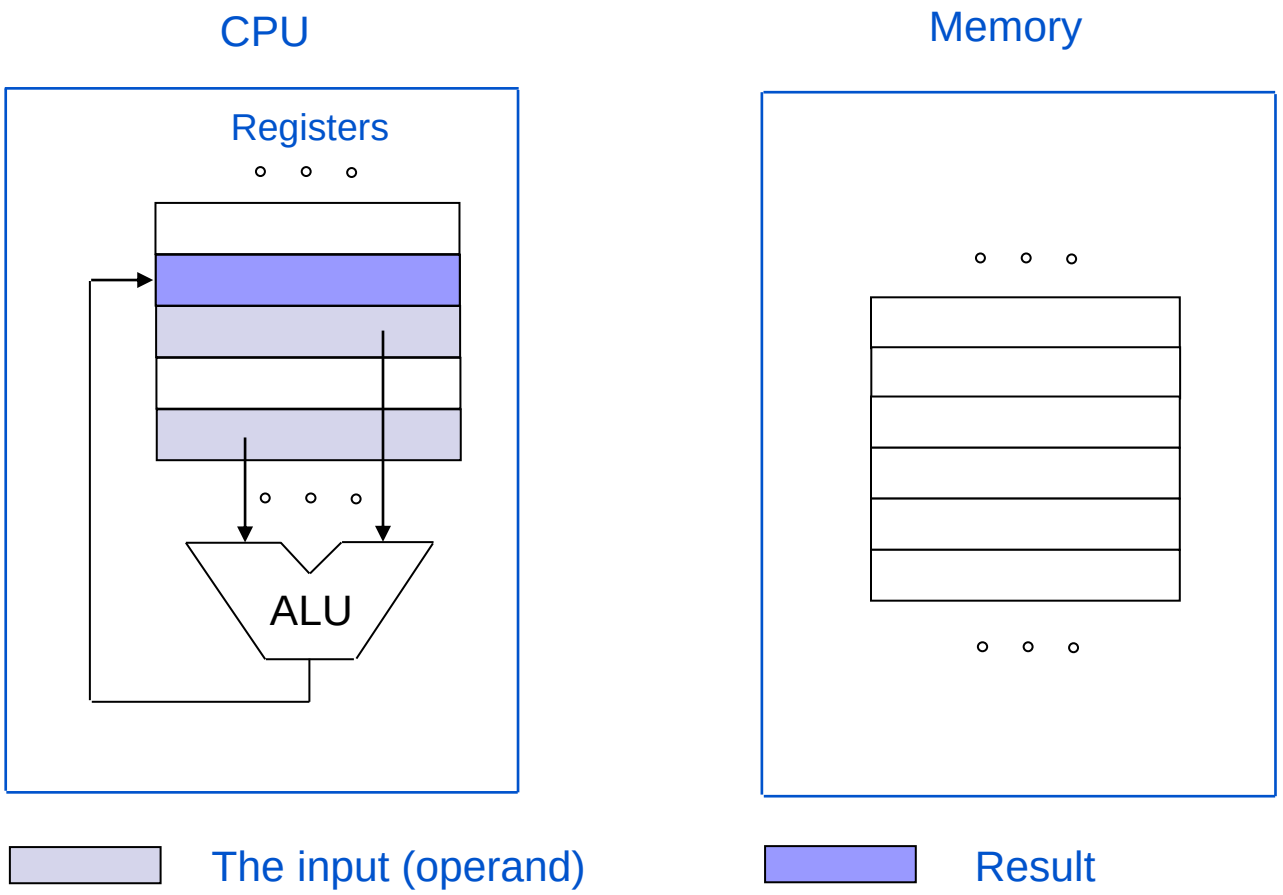
Modes of saving of the operands in the CPU - Stack



- If you read the stack, the stack pointer moves down one position and the top of the stack is new value.
- When writing to the Stack, the new value is stored on the highest free location, which becomes the top of the stack.
- On two-operand operations (eg. adding A + B), the operands A and B are read from the Stack (A must be at the top of stack and B just below it), and the result is saved on the top of the stack.
- Stack computers or computers with stack architecture were popular around the year 1960.

- **Register set**: Memory in the CPU is designed as a set of registers to which access is possible without restrictions.
  - The number of registers in today's computers is from 8 to 100 or more.
  - Each register has its own address, like words in main memory. For register address the instruction needs significantly fewer bits than for the memory address.
  - There are two types according to restrictions on registers use:
    - All registers are equivalent - general purpose registers
    - The set of registers is divided into two groups. One is used for arithmetic-logic operands, the other for calculating addresses (base or index registers)

Modes of saving of the operands in the CPU– register set



- Advantages of a set of programmatically accessible registers in CPU:
  - **Higher speed**. Because the memory in the CPU is small it can be build in a faster technology then the main memory – shorter access time. It is possible to access multiple register at the same time.
  - **Shorter instructions**. Because there is smaller number of registers then the main memory you need less bits for register address – smaller fields in instruction to describe the operands.
  - **Number of transfers** between CPU and main memory is **reduced**. Registers allow to store intermediate results (as long as we don‘t run out of free registers)

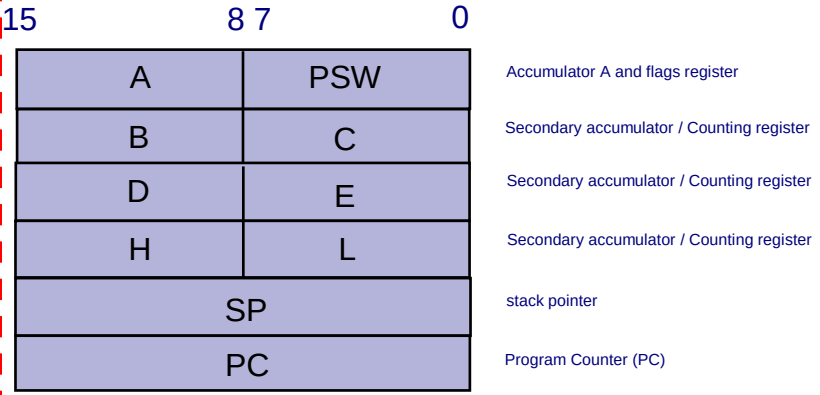
- There are methods for compilers to enable the best possible use of registers.
- Consequence: All after 1980 developed computers have register set as „memory“ in CPU.

- For most computers, usually it can be easily understood which method for storing operands they use.

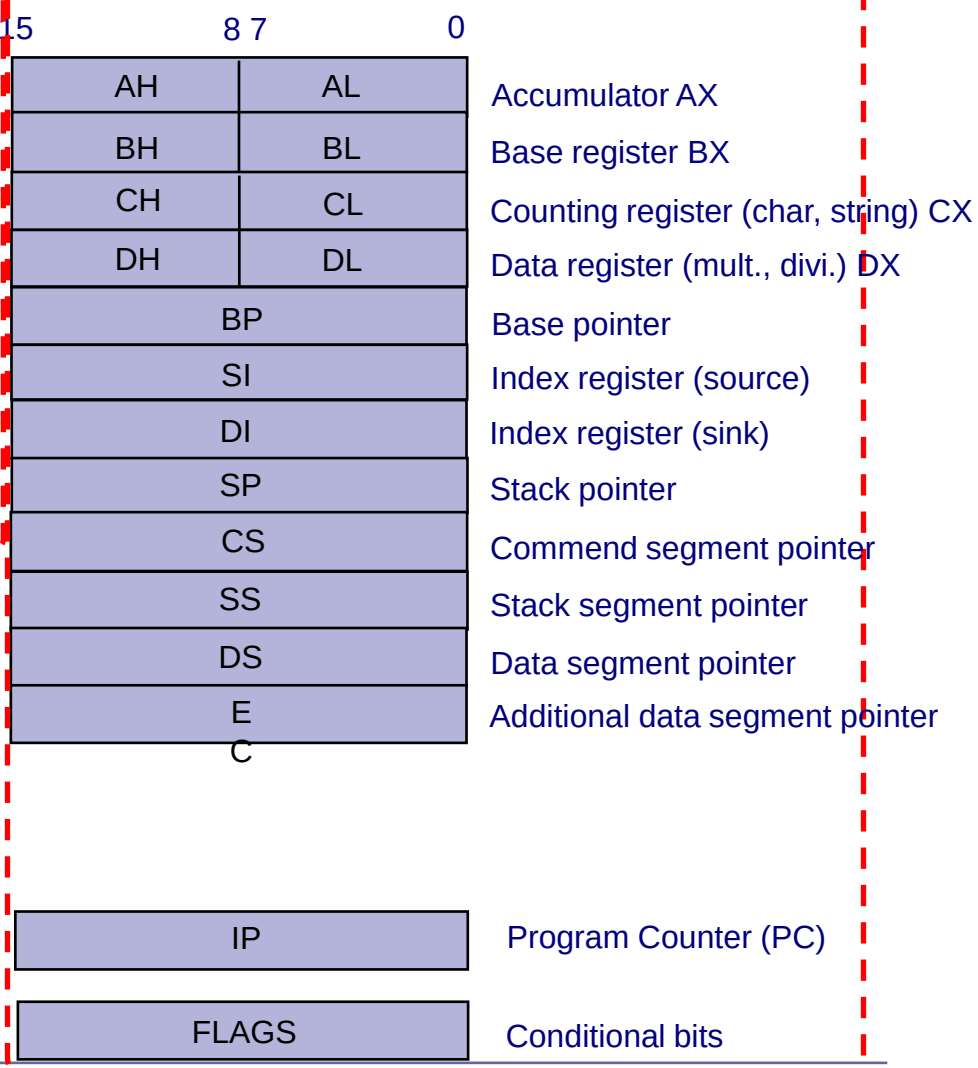
- Some computers are somewhere in between:
  - Microprocessor Intel 80x86 had in the beginning (Intel 8086 - year 1978) only one accumulator (general purpose register) and some additional registers to help with operands access.
  - Intel 80386 - year 1985: 8 general purpose registers.
  - Pentium 4 - year 2006: 16 general purpose registers in 32 additional registers (FPU, MMX, XMM).

Programmatically available registers

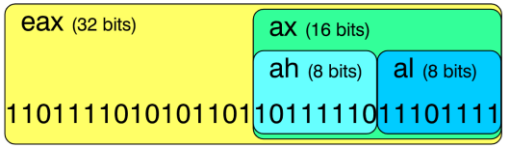
8080 - 8-bit processor (1974, 8085 1977)



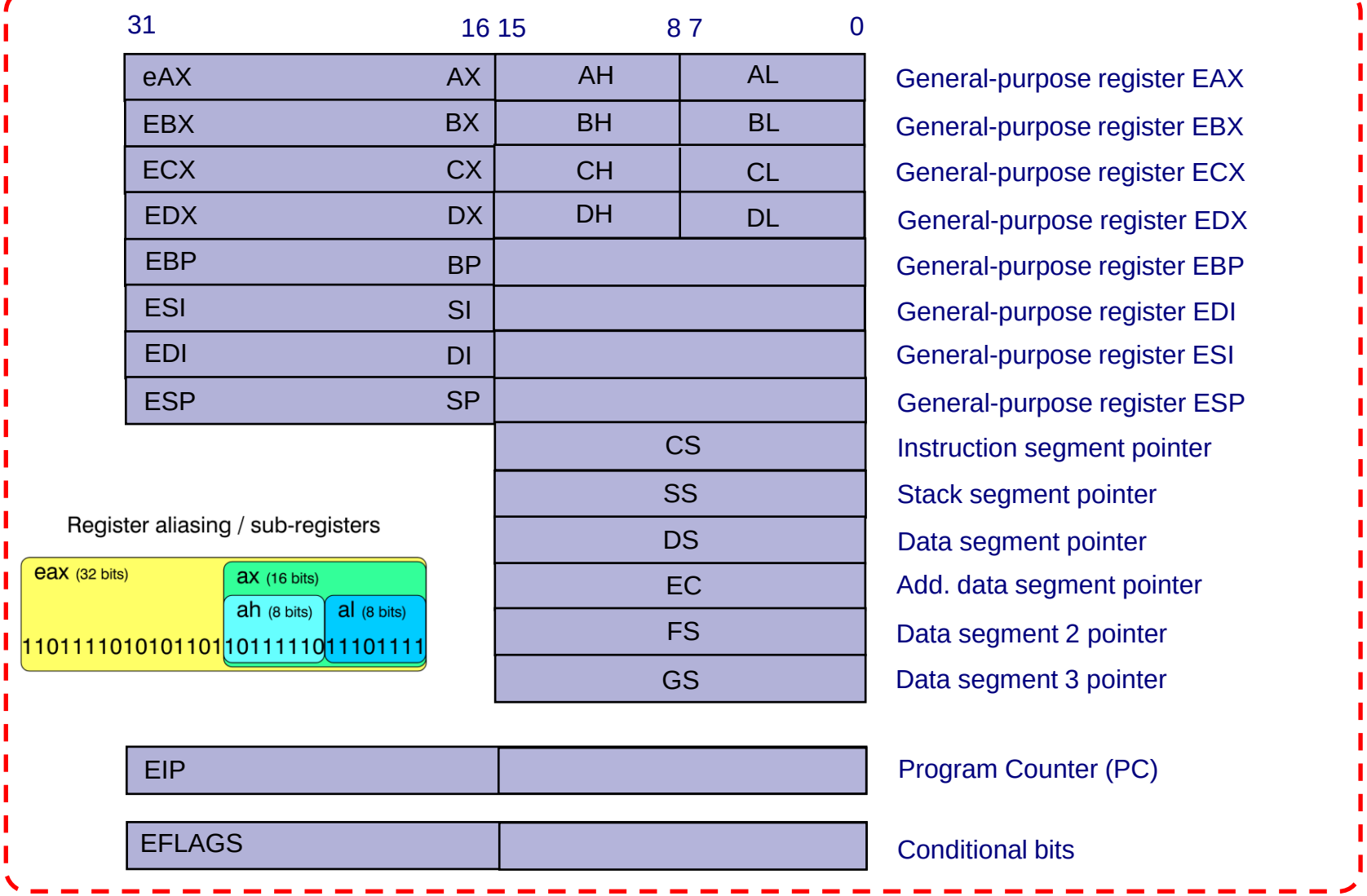
8086 - 16-bit processor (1978)



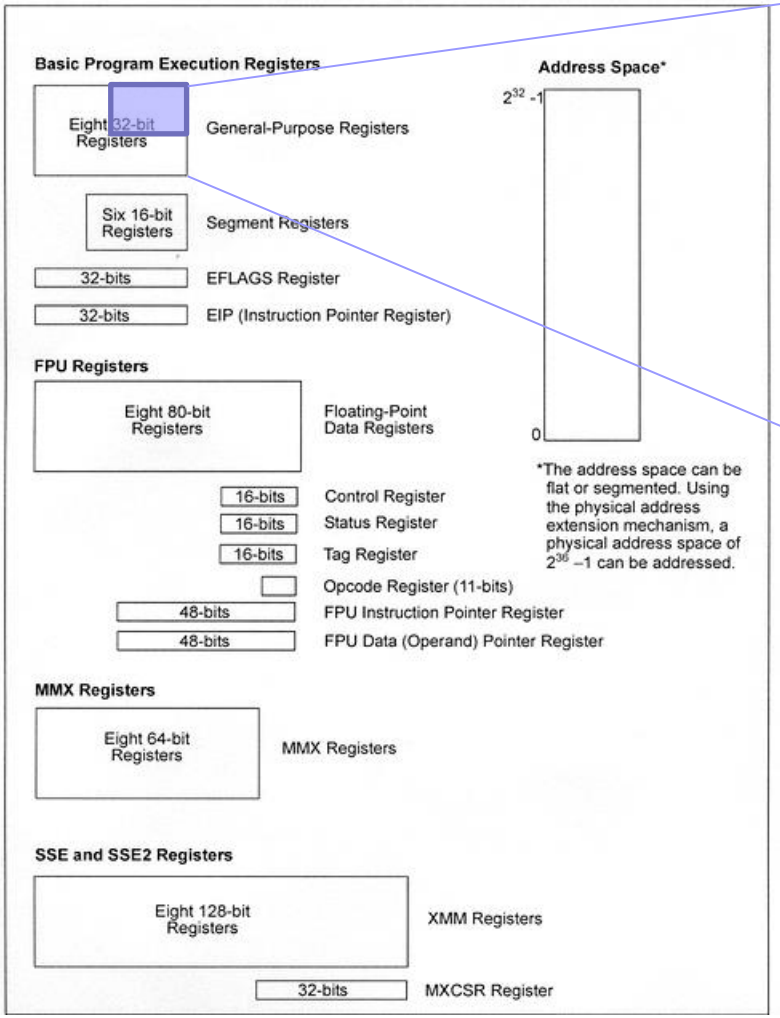
Register aliasing / sub-registers



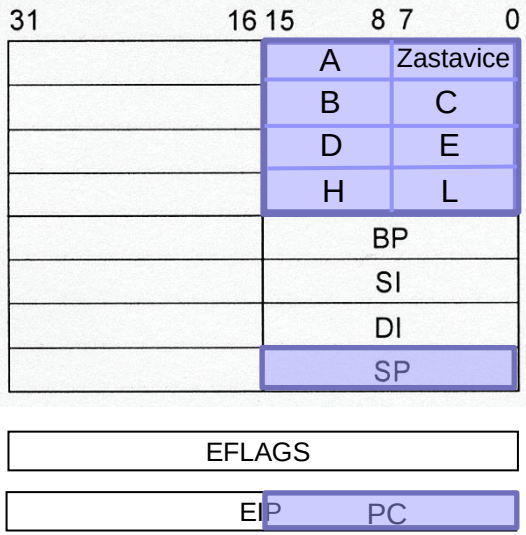
Programmatically available registers 80386 - 32-bit processor (1985 ⇒ x86 architecture)



Intel x86
Architecture *Intel® 64 in mode* IA-32

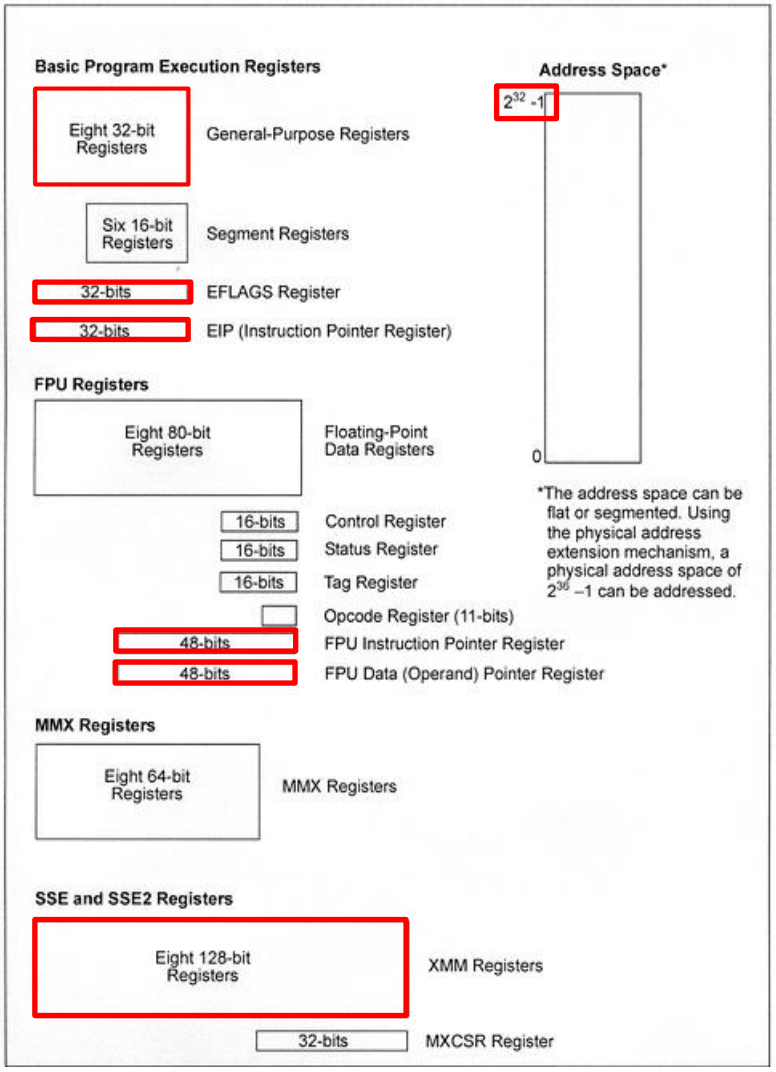


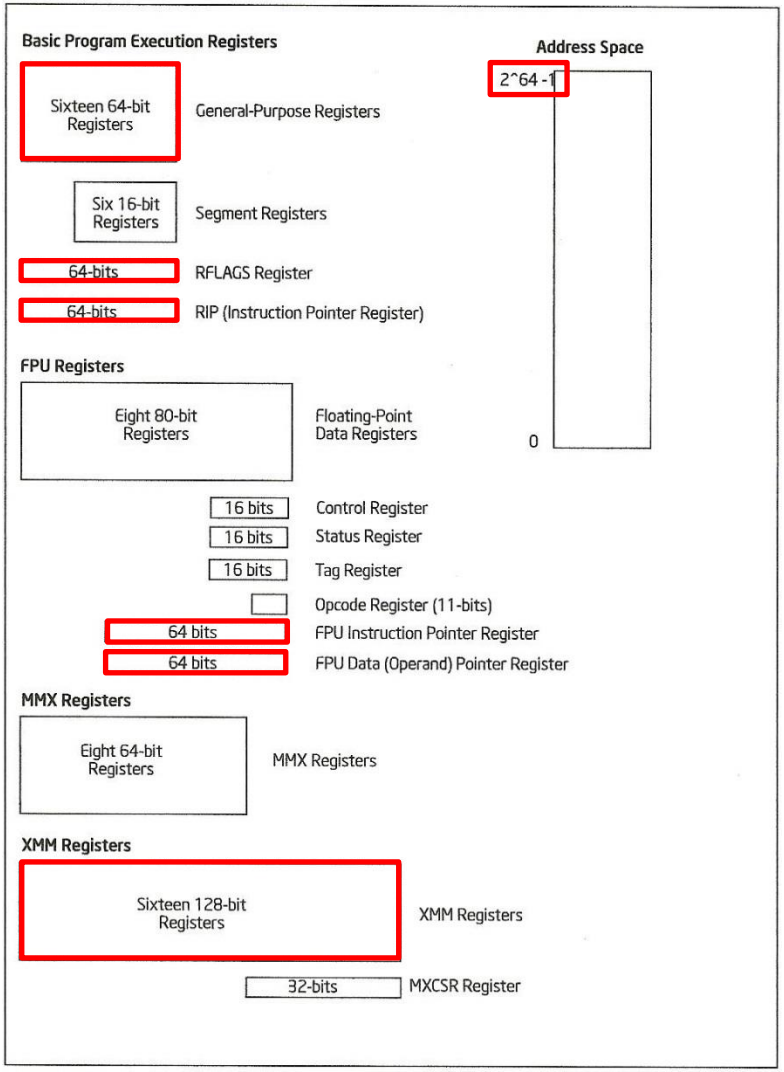
Registers of 8-bit processor Intel 8080 in year 1974 and Intel 8085 in year 1977



Software accessible registers of the Intel x86 architecture *Intel® 64*

32-bit mode operation of the IA-32 | Differences | 64-bit mode operation of IA-32e





4.3 The number of explicit operands in instruction

- The second most important instructions property that impacts the user's view of the computer.

- A small number of operands in the instruction:
  - Shorter instructions that take up less memory space
  - Less powerful instructions

- A larger number of operands in the instruction:
  - More powerful instructions
  - More complex CPU structure, longer instructions

- The number of operands in the instruction is also influenced by the type of operation carried out by the instruction.
- Elementary operations with more than three operands are rare (two input operand and result).
- Therefore, today's computers instructions contain at most three explicit operands.
- Explicit operands are usually in the instruction given by direct or indirect address information where operand is stored.

- Computer, which uses instructions with a maximum of m-operands are called **m-operand** or **m-address computer.**

- Depending on the number of operands in explicit instructions computers can be divided into five groups:

  - 3 + 1operand computers
  - 3-operand computers
  - 2-operand computers
  - 1-operand computers
  - No-operand or stack computers

  Presented in chronological order :

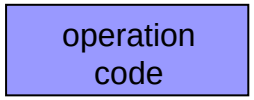
3 + 1 operand computers

- A representative of this type of computers was EDVAC
- Such computers today no longer exsist
- +1 (in „3+1“) means that the address of the next instruction is an instruction operand

- Symbolically we can describe operation with :

OP3 ← OP2 ⊕ OP1 (⊕ means any operation on the two operands)
PC ← OP4

- **No-operand (Stack) computers**
  - This group includes computers that have a CPU memory in a form of stack



  - Symbolically we can describe operation with :

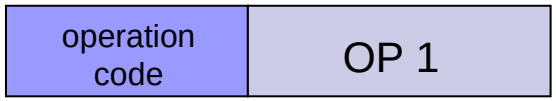
$Stack_{TOP} \leftarrow Stack_{TOP} \oplus Stack_{TOP-1}$

$PC \leftarrow PC + 1$

- Operations are performed on operands at the top of the stack, so the instruction don’t need explicit operands
- Required are at least two instructions to transfer an operand from memory to the stack (PUSH) and from the stack to memory (POP or PULL)
- As with 1-operand computers, there are in addition to stack also additional special purposes registers in no-operand computers
- Example: Computer Atlas 1961
- After 1980, there were no stack computers developed anymore

1-operand computer

- These are computers that have CPUs with a single accumulator (maybe two, for example accumulator A and B in CPU 68HC11)
- One of the operands is always located in the accumulator, the result is also stored there, so one explicit operand is enough



- Symbolically we can describe operation with :

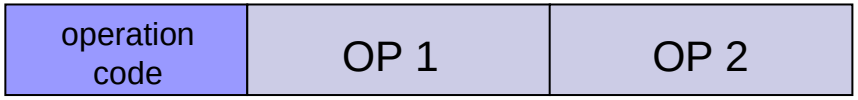
$AC \leftarrow AC \oplus OP1$ (AC is abbreviation for accumulator)
$PC \leftarrow PC + 1$

- In addition to the accumulator (one or two) 1-operand computers usually also have at least a few additional registers for special purposes (eg. the index register for storing memory address).

- In the years 1970 to 1980 were virtually all microprocessors, due to technological limits, of 1-operand type.

2-operand computers

- The result of the operation can in most cases be stored in the space of one of the two input operands.
- Thus, there is no need for the third operand and we obtain 2-operand computer from 3-operand computer



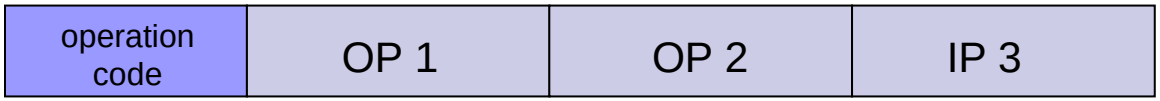
- Symbolically we can describe operation with:

$OP2 \leftarrow OP2 \oplus OP1$
$PC \leftarrow PC + 1$

- Operands can be in memory or in the CPU registers
- Many 2-operand computers allow both operands to be in the memory (in 3-operand almost never)
- Most often one of the operands is in the register and the second in memory
- If we want to keep both input operands then one must be previously stored in a different location, in which case the 2-operand computer is slower than the 3-operand
- Until 1990 2-operand computers were the most common, today 3-operand computers prevail

3-operand computers

- Using the random access memory, the address of the next instruction is no longer needed
- Implicit order of instruction execution is set by rule PC←PC + 1



- Symbolically we can describe operation with :

$OP3 \leftarrow OP2 \oplus OP1$

$PC \leftarrow PC + 1$

- Basic arithmetic operations have three operands, so computers with three explicit operands are closest to mathematic standards.

- Most computers developed after 1980 are 3-operand, but usually with the restriction that the operands are in the registers of the CPU (Load/Store computers).

Depending on the number of operands in explicit instructions computers can be divided into five groups – summary:

- 3 + 1operand computers

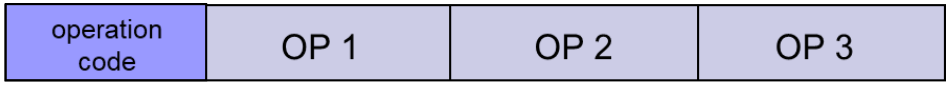
  $OP3 \leftarrow OP2 \oplus OP1$

  $PC \leftarrow OP4$

- 3-operand computers

  $OP3 \leftarrow OP2 \oplus OP1$

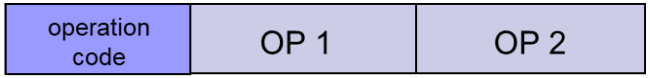
  $PC \leftarrow PC + 1$



- 2-operand computers

  $OP2 \leftarrow OP2 \oplus OP1$

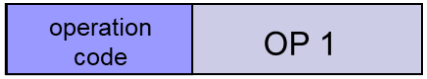
  $PC \leftarrow PC + 1$



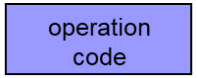
- 1-operand computers

  $AC \leftarrow AC \oplus OP1$

  $PC \leftarrow PC + 1$



- No-operand or stack computers



4.4 Operand location and addressing modes

- Operands can be stored in:
  - Programmatically accessible registers in the CPU
  - In one or more neighbour memory words in the main memory (or on any level of the memory hierarchy)
  - Operands can also be stored in one of the registers on controllers for I/O devices or I/O processors. Because of the small number of I/O instruction, we won't elaborate on them.
  - Operands can also be embedded in instructions („Immediate operand")

- **Register operands** are operands that are stored in CPU registers
  - Register operands are almost always given by the register address in which they are stored.
  - There is a special field in the instruction for the register address, but it can be also part of the operation code.
  - Only for accumulator (or stack) computers the address is not needed.
  - Because there are just a few register we only need a small number of bits to represent the address in the instruction. (eg. 16 = 2^4 registers $\Rightarrow$ 4 bit for the register address)

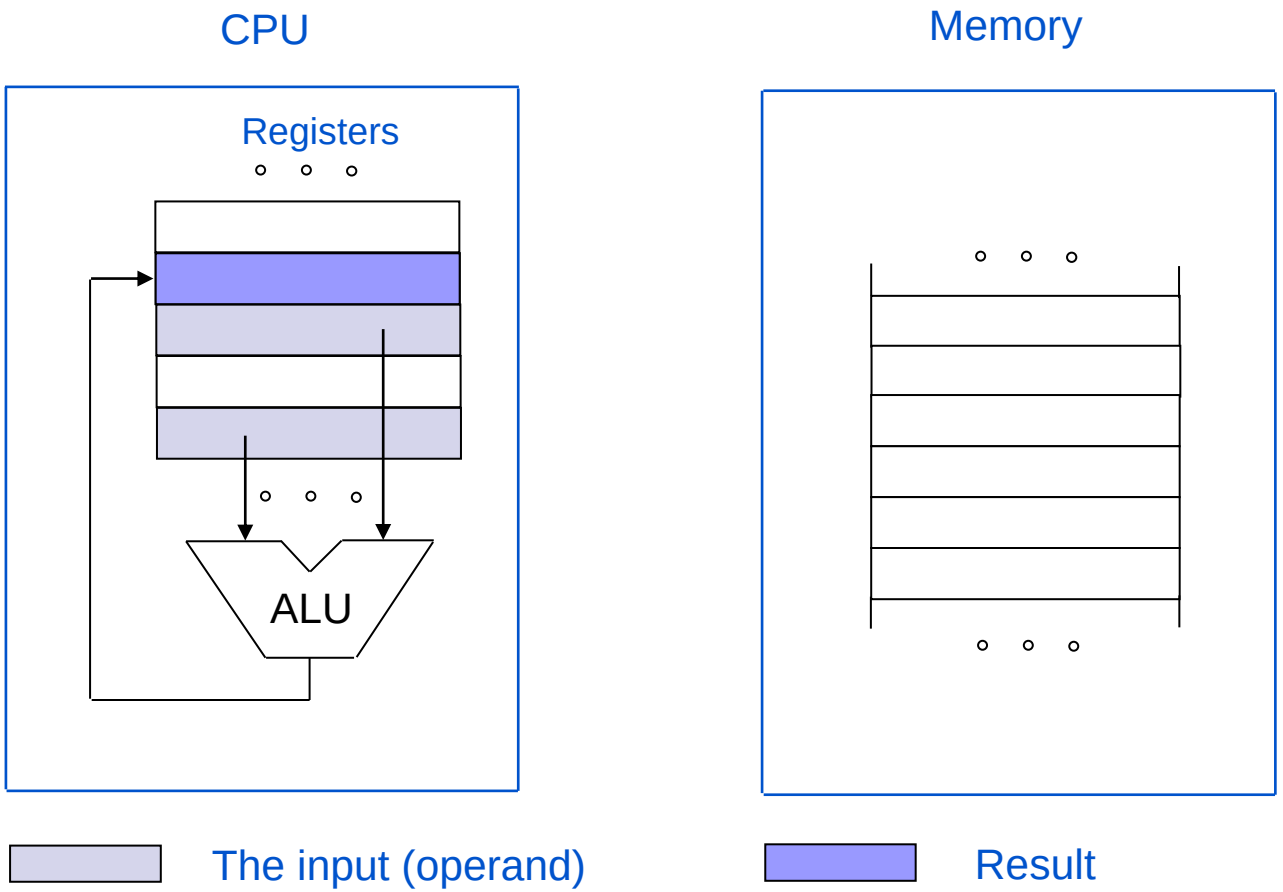
- **Memory operands** are those which are stored in the main memory (or at various levels of the memory hierarchy)
  - Addressing is much more complex then for register operands
  - There is more space in the main memory which means a larger address in the instruction (eg. main memory 4 GB = 2^{32} B $\Rightarrow$ 32 bits for the address)
  - Access to operands in memory is subject to a certain rule, which usually requires changing the address.

- **Immediate operands** are those which are stored in the instruction itself
  - They are available "immediately" - after reading the command
  - Fast path to transfer constants to registers (command read already enough) Usually the place in the command is limited - so the stock of values is also limited

- With 2 - and 3 – operand computers we, considering the location of the operands, differentiate between three types of computers:
  - Register-register computers
  - Register-memory computers
  - Memory-memory computers

- **Register-register computer**
  - All operands for ALU instructions are in CPU registers.
  - Instruction LOAD and STORE are used to transfer operands from memory to registers and vice versa, hence the name **load/store computer.**
  - Execution time of ALU instructions is always the same.
  - We usually need more instructions for the same problems as with computers that can have operands for ALU instructions in the memory.

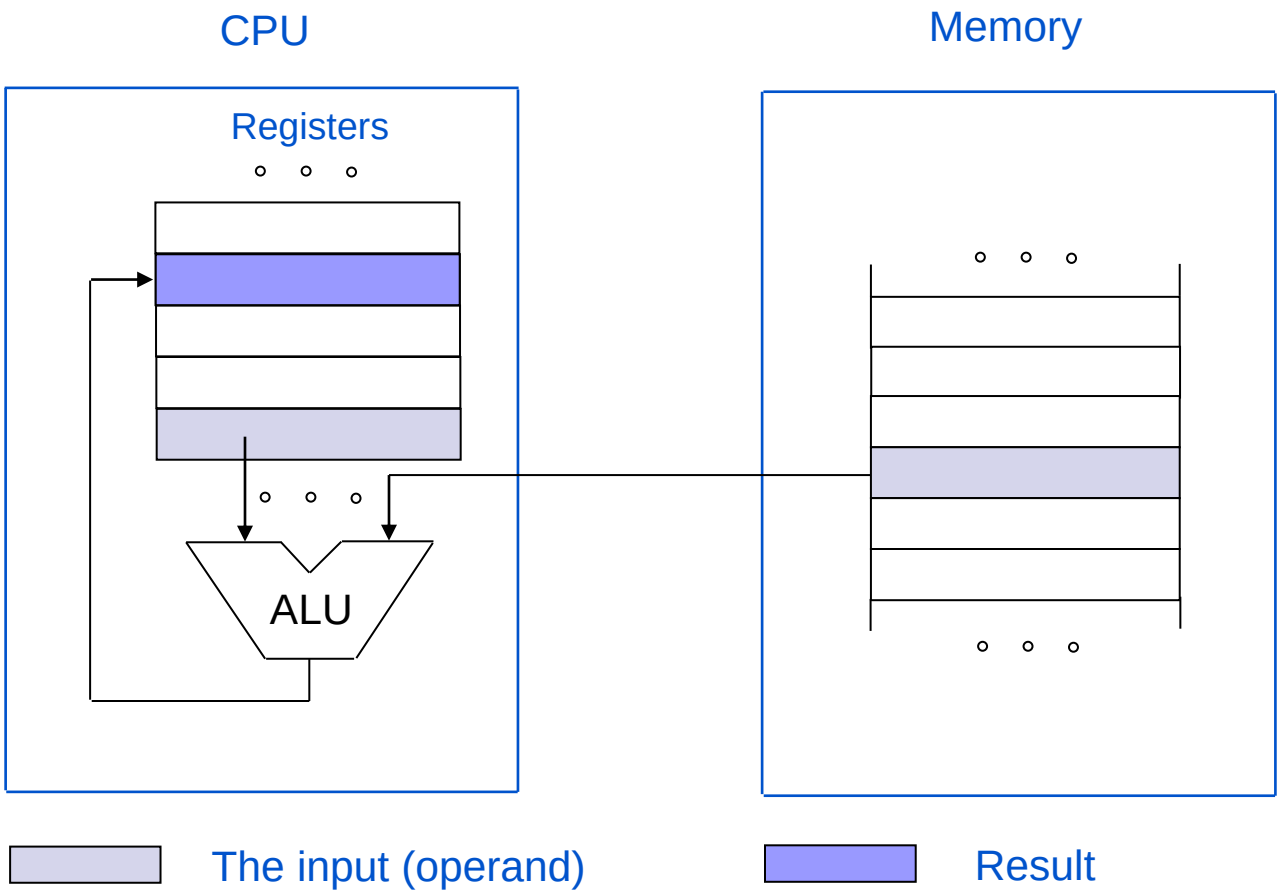
Operand location and addressing modes – register-register computer



Case: ADD R3,R2,R0

- **Register-memory computer**
  - One of the operands is in the memory or the register and the rest is always in the registers
  - These computers include as a subset also instructions from register-register computers
  - ALU instructions can use memory operands without transferring them first into registers with the LOAD instruction

Operand location and addressing modes – register-memory computer



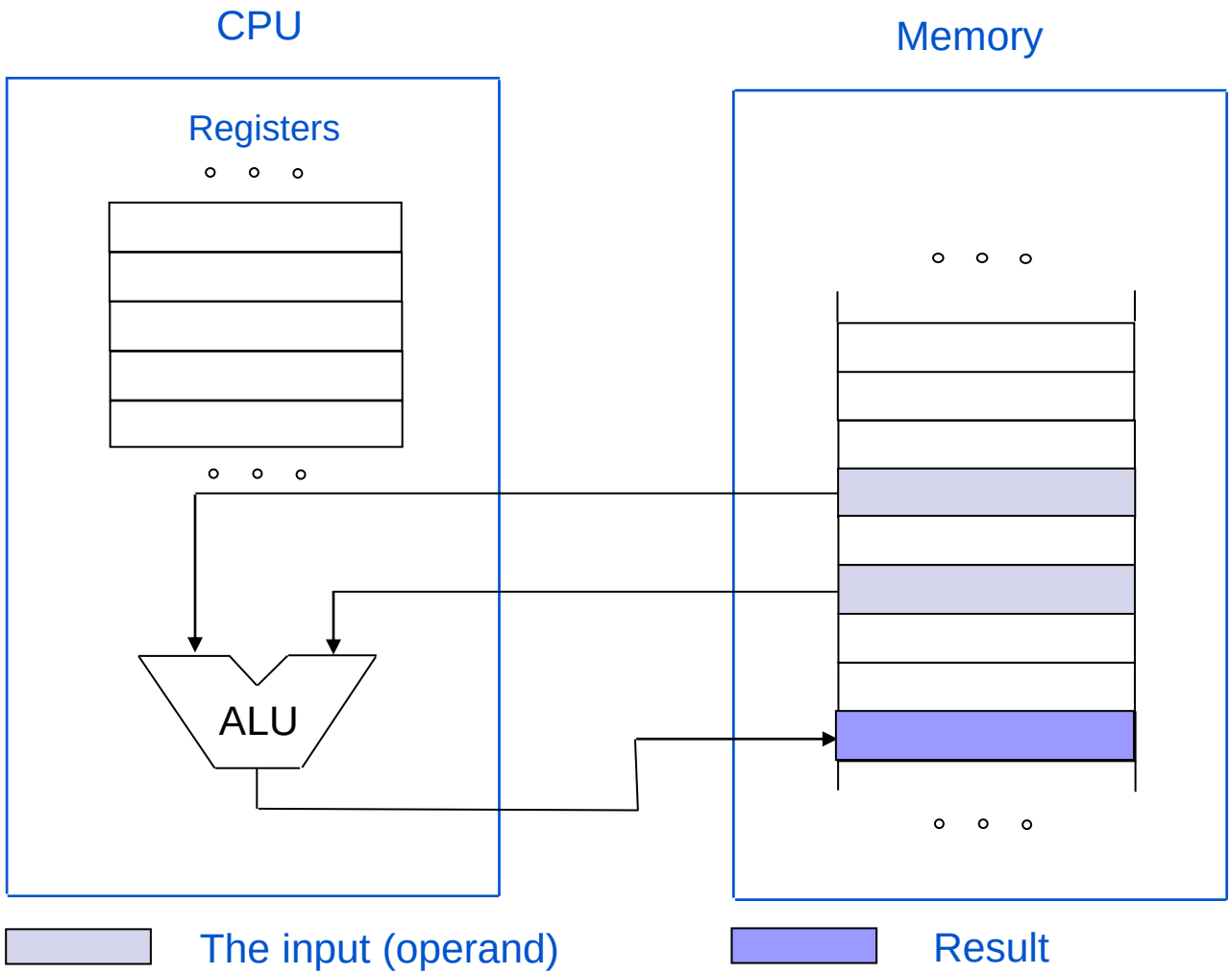
Primer: ADD R3,R0,[STEV1]

- Instructions are longer and more complicated but for the same problems we need fewer instructions.
- Execution time is dependent on operand location:
  - Operand in register – shorter execution time
  - Operand in memory – execution time is longer and is dependent on where in the memory hierarchy the operand is located.

Memory-memory computer

- Each operand can be in the memory or the registers.
- Include instructions from register-memory computers and register-register computers.
- Those computers are the most general and offer a lot of different solutions for the same problem.
- Instructions are complex and have different lengths, the gep in execution time is also wary big.

Operand location and addressing modes – memory-memory computers



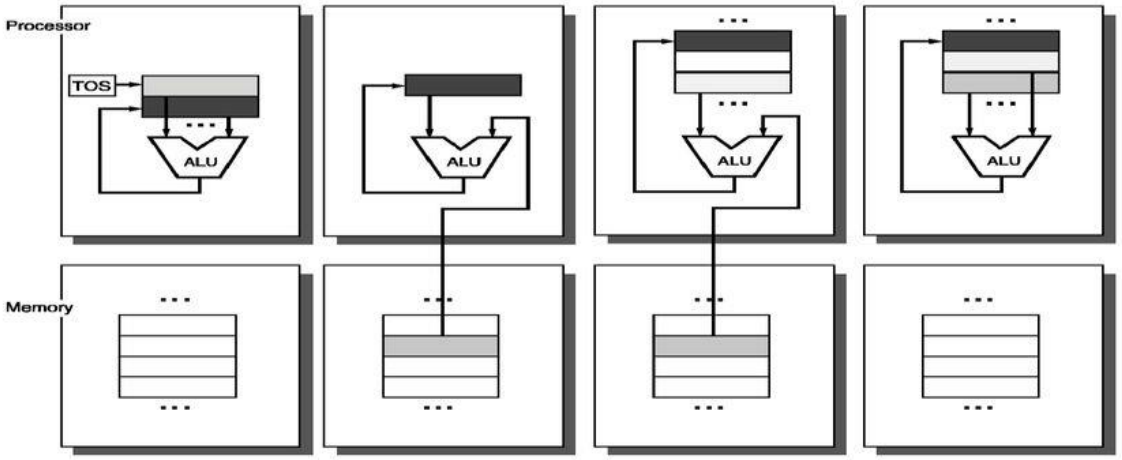
- Comparison

Four Architecture Classes

Assembly for C:=A+B:

Each instruction has an opcode and one or more operands

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R1,B	Load R2,B
Add	Store C	Store C,R1	Add R3,R1,R2
Pop C			Store C,R3



3

Addressing modes - how are operand addresses given

- To solve these problems many modes of addressing were developed.
- All addressing modes can be divided into three basic groups:
  - Immediate addressing
  - Direct addressing
  - Indirect addressing

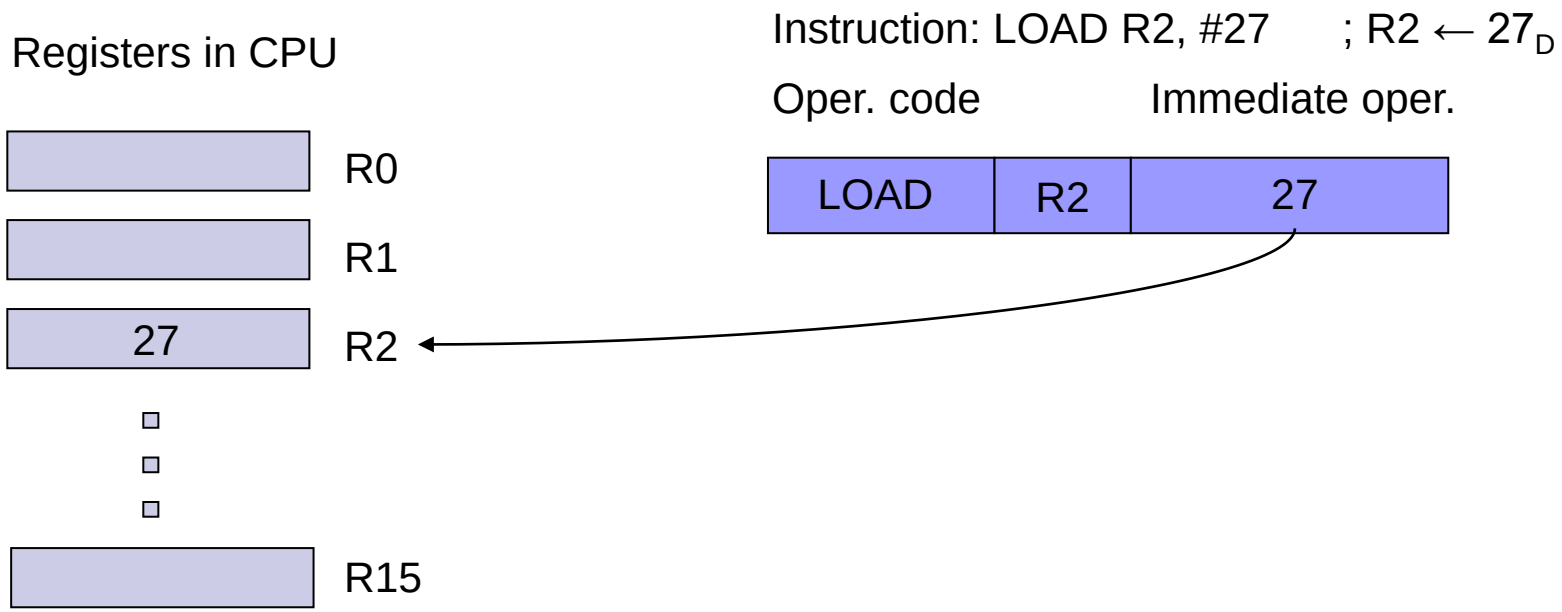
Ukaz: LOAD R2, #27 ; R2 ← 27_D

Oper. koda Takojšnji oper.

Immediate addressing

- The operand is given in the instruction by its value
- Operand is part of instruction and is transferred to the CPU along with it, so there is no need for additional memory access (access is faster)
- The operand is called **immediate operand** or **literal**
- Computers differ by the number of instructions which use immediate addressing and the length of the immediate operand (8, 16, or 32-bit)
- Some computers do not have immediate addressing.

- An example of a instruction with immediate addressing:
  - With the instruction LOAD we want for example to transfer the value 27_D into the register R2
  - # is usual the label for an immediate operand

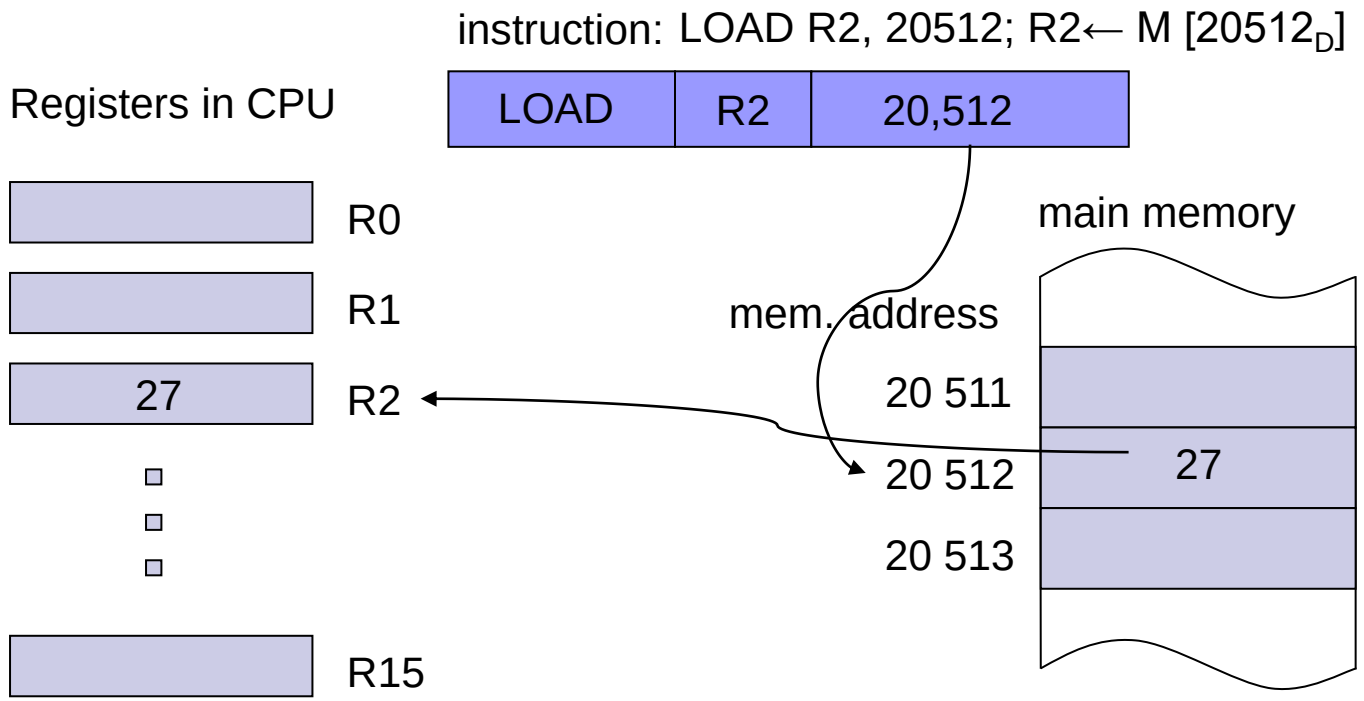


- Examples of ARM9 instructions with immediate addressing:
  - The immediate operand in the ARM9 processor is 8-bit and 4-bit for offset
  - Value $(0 \ldots 255_D) * 2^{2 * (0..12)}$

```
mov     r0, # 128          @ R0 ← 128
add     r3, r3, # 1        @ R3 ← R3 + 1
bic     r0, r0, # 0x80     @ b7 (R0) ← 0
```


- **Direct addressing** (also absolute addressing)
  - Operand address is given in the instruction
  - This mode is used especially for register operands and is in this case called direct register addressing or shorter **register addressing**
  - For memory operands, the operand address in the memory is given in the instruction
  - When the address is part of the instruction, it is also called **absolute addressing**

- Example of instruction with direct addressing:
  - With the instruction LOAD we want, for example to transfer the content from the memory location with address 20512_D into the register R2,
  - in assembly eg. LOAD R2, 20512



- Intel x86 („CISC“)
  - has several types of direct memory addressing.

- ARM9 („RISC“)ž
  - does not have direct memory addressing.

  - direct addressing is in ARM9 used only for register operands.

```
add r5, r0, r1     @ R5 ← R0 + R1
mov R2, R4         @ R2 ← R4
```


- **Indirect addressing** (also deferred addressing)
  - It's used to address memory operands
  - The address in the instruction is given indirectly via some other value or intermediary
  - This other value (or intermediary) is in:
    - the memory - **Memory indirect addressing**

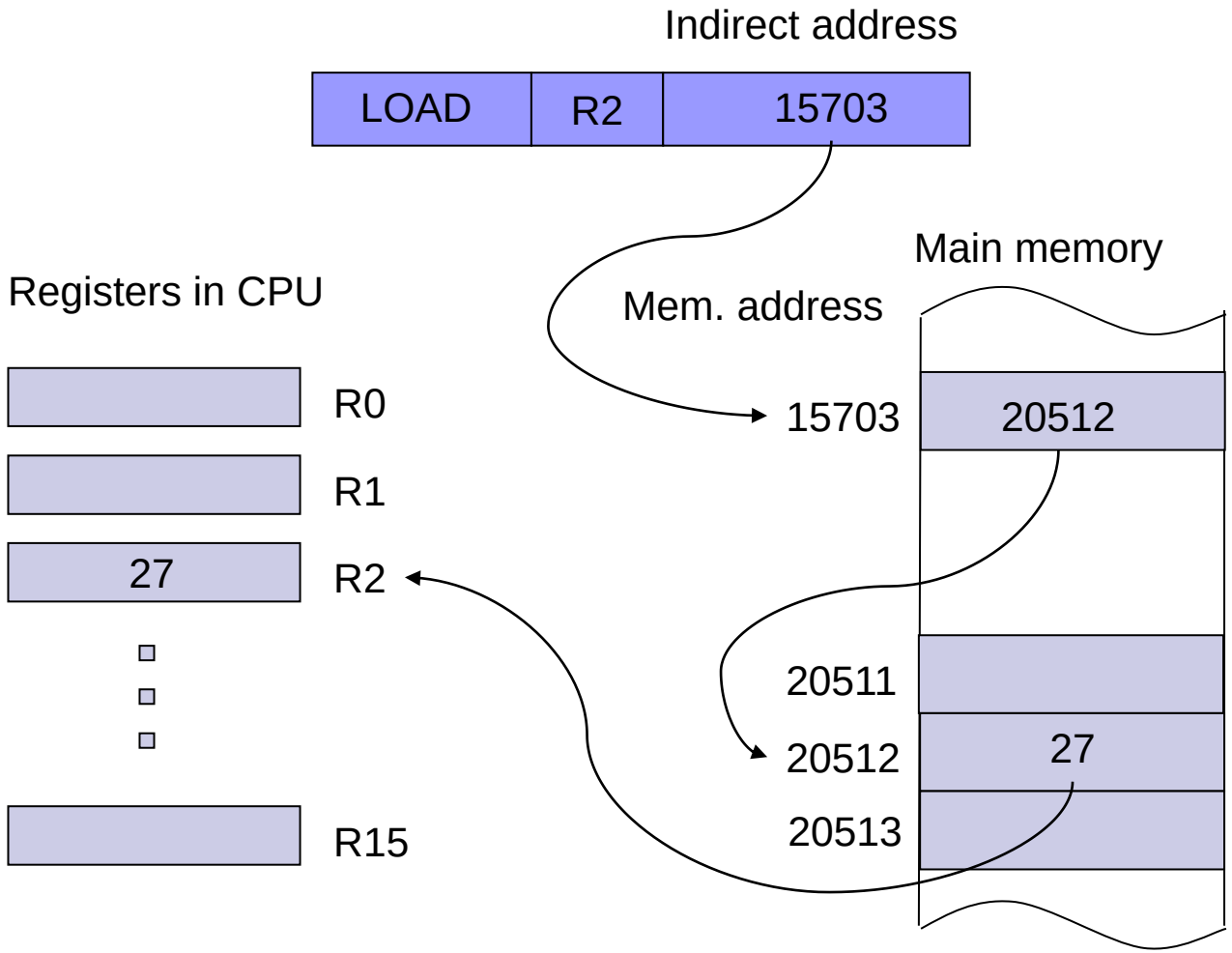
      LOAD R2, @ (15703); R2← M [M (15703_D)]
    - The CPU register - **Register indirect addressing**

      LOAD R2, 12 (R0); R2← M [12+ (R0)]

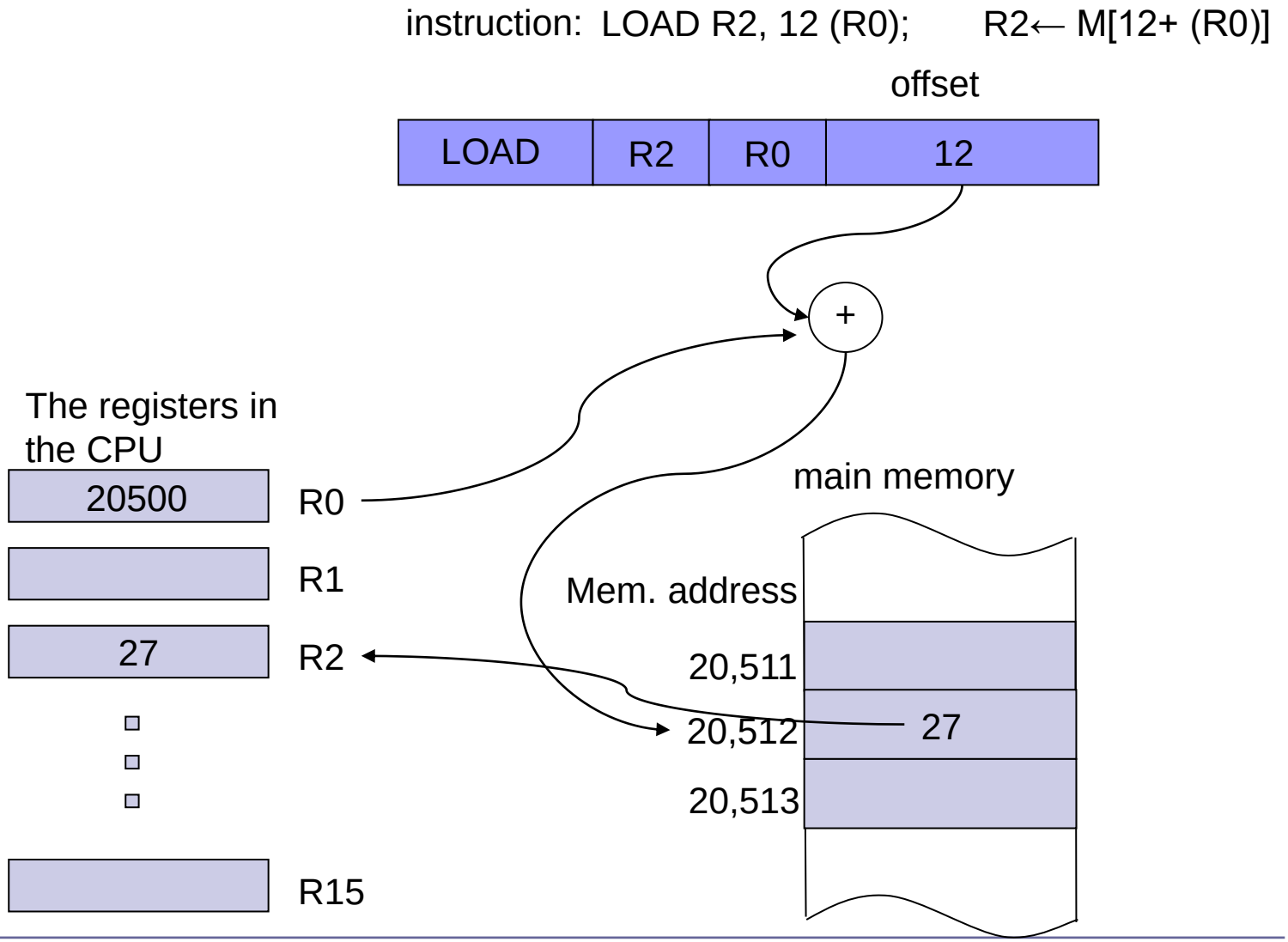
- For memory indirect addressing, the memory address where the operand address is stored in the memory, is given in the instruction.
- For register indirect addressing the register address and offset (displacement) is given in the instruction.
- The memory address of the operand is calculated from the content of the register and the offset.
- Indirect addressing allows you to arbitrarily change the operand address, thus eliminating the weakness of direct addressing.

- An example of a instruction with memory indirect addressing:
  - With the LOAD instruction we want for example to transfer the value of the memory word with the address 20512_D to the register R2
  - The memory address 20512_D (operand address) is saved on the memory address 15703_D (indirect address)
  - In assembly LOAD R2, @(15703)
  - @(….) is often used to denote indirect addressing

Instruction: LOAD R2, @(15703) ; R2 ← M[M(15703_D)]



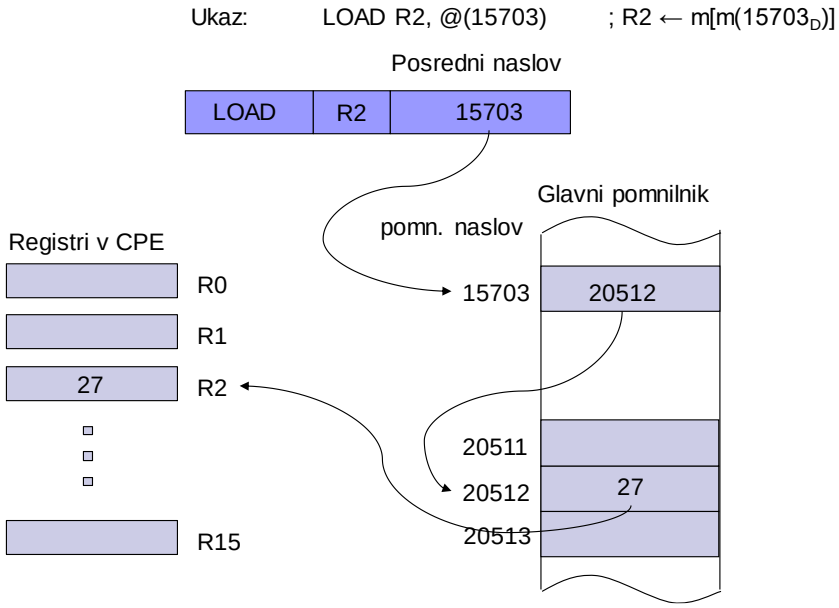
- An example of a instruction with register indirect addressing:
  - With the LOAD instruction we want, for example to transfer the value of the memory word with the address 20512_D to the register R2
  - In the register R0 we stored the address 20500_D (indirect addressing)
  - In assembly e.g. LOAD R2, 12(R0)
  - In the instruction 12_D is the offset which is added to the address in the register R0. The result is the operand address ($20500_D + 12_D = 20512_D$)



Comparison of indirect addressing modes

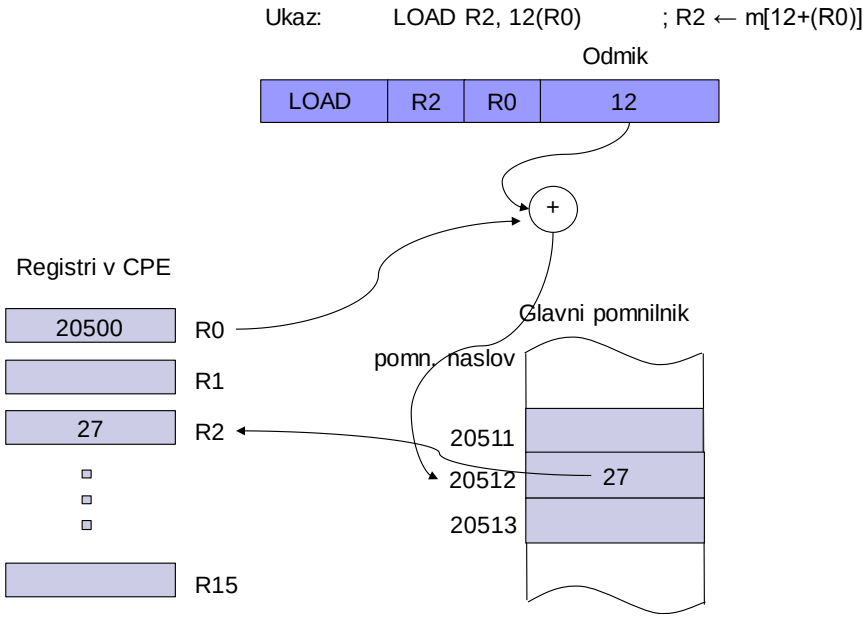
memory indirect addressing

ARM9: not supported



register indirect addressing

ARM9: LDR R2[R0, # 12]



- Register indirect addressing has many variants.
- In majority of computers, the register indirect addressing is the most common way of accessing memory operands.
- Operand address is always determined by the content of at least one register, another name for this type of addressing is therefore **relative addressing.**

- Variants of register indirect addressing:
  - Base Addressing
  - Indexed Addressing
  - Auto-indexing Addressing
  - PC-relative addressing

- **Base addressing**
  - The most common type of register indirect addressing.
  - The address of the register R_b and offset D is given to the instruction.
  - Memory address (eg. A) of an operand is obtained by summing up the contents of the register R_b and the offset D:

    $$A = R_b + D$$

  - The length of the register R_b is usually equal to or longer than the length of memory address
  - R_b is called the **base register** and A the **effective address**

- According to the size of the offset D there are many variants of base addressing
- The extreme examples are that there is no offset (D = 0) or that the offset size in bits is equal to the size of the memory address
- In the second example base addressing changes into **indexed addressing**

Indexed addressing

- The length in bits of the offset is equal to the length of the memory address, which means that we can, just by changing the offset, address the whole address space
- The second option to change base addressing into indexed addressing is by using additional register
- The actual address A is calculated by summing the content of the index register and the offset D1, the offset D1 is the sum of the content of the base register and the offset D:

$$A = Rx + D1 = Rx + \underbrace{Rb + D}$$

Offset by the size of the memory addressing

Rx – index register
Rb – base register

- ARM9:
  - Indirect addressing with immediate offset (= base addressing with offset):

    LDR R5, [R3, #12] @ R5 ← M32[R3 + 12]

    [] is used to mark indirect addressing in ARM9 assembly

  - Indirect addressing with offset in the register (= indexed addressing):

    LDR R5, [r3, r1] @ R5 ← M32[R3 + R1]

    32-bit offset is in the register R1

    Because registers are 32-bit, the offset can be 32-bit and can address any memory word

- **Autoindexing**
  - Pre-decrement addressing
  - Post-increment addressing
  - Size indexed addressing

ARM9 example:
Auto pre-indexing with immediate offset:

```
ldr r0,[r1,#4]! @ r1<-r1+4; r0<-M32[r1]
```

Auto post-indexing with register offset:

```
ldr r0,[r1],r2  @ r0<-M32[r1]; r1<-r1+r2
```

- In branch instructions many computers use **PC–relative addressing**
  - The program counter (PC) is used as the base register
  - The offset is a signed number (presented in two's complement)
  - The actual address is calculated relative to the PC value

ARM9 example:
PC-relativ addressing:

```
ldr r0,stev1  (not a valid instruction, replaced by)
ldr r0,[pc,odm]          @ r0<-M32[pc+odm]
```


ARM9 – Examples of register indirect addressing:

(on the lab exercises we only use the first one)

- Indirect addressing with immediate offset (= base addressing with offset):

```
ldr  r5, [r3, #12]              @ R5 ← M32[R3 + 12]
```

- Indirect addressing with offset in a register (= indexed addressing):

```
ldr  r5, [r3, r1]       @ R5 ← M32[R3 + R1]
```

- Auto pre-indexing with immediate offset

```
ldr r0,[r1,#4]!                 @ r1<-r1+4; r0<-M32[r1]
```

- Auto post-indexing with register offset:

```
ldr r0,[r1],r2                  @ r0<-M32[r1]; r1<-r1+r2
```

- PC-relative addressing:

```
~~ldr r0,stev1~~ (not a valid instruction, replaced by)
ldr r0,[pc,odm]                 @ r0<-M32[pc+odm]
```


4.5 Operations (types of instructions)

- Computers differ a lot according to the number and types of operations.
- The computer has to have enough operations to guarantee equivalence with the Turing machine (it has to be possible to calculate with it all that is calculable).
- For this just a small number of primitive operations is enough or in the extreme case even only one that is powerful enough.

- There are two starting points to determine the type and number of operations on computers:
  - **The instruction set should be strong.** For commonly used functions one or a small number of operations should be used.
  - **Operations should be similar to already established types of operation.** Most manufactures use the same or very similar operations which simplifies programming, creation of compilers and digital electronics in the CPU
- The type of the operation can be determined by its instruction name
- The instruction is denoted with a **mnemonic** with which the instruction is defined in the assembly language

- All computers have the same basic groups of operations:
  - Arithmetical and logical operations
  - Data transfer
  - Control operations
  - Floating point operations
  - System operations
  - Input/Output operations

4.5.1 Arithmetic and Logic operations (ALU)

- These operations are executed in the ALU, instructions which execute those operations are called **ALU instructions**

Arithmetic operations : in this group are operations on fixed point operands (integers)

- Typical arithmetic operations are:
  - Two-operand - summation, subtraction, multiplication and division
  - One-operand - negation, absolute value, increment in decrement

Example ARM9: `add Rd, Rn, shift_op  @ Rd = Rn + shift_op`

`add, sub, ALU instr.  (similar in operands)`

add{*cond*}{**s**} *Rd*, *Rn*, *shift_op*

- shift_op = label for the second operand, that can be:

RA

- Immediate operand:

```
add r1, r2, #5   @ R1 = R2 + 5
```

- Register:

```
add r1, r2, r7   @ R1 = R2 + R7
```

OR

- Shifted register

```
add r1, r2, r7, LSL #2   @ R1 = R2 + R7 * 4
```

LSL #2 … two shifts left = multipl. by 4

```
add r1, r2, r7, LSL r3   @ R1 = R2 + R7*2^r3
```

Data Processing Mode: *shifter_op*

Operation	Syntax	Comments
Immediate value	*#imm8r*	
Register	R*m*	
Logical shift left immediate	R*m*, **lsl** *#imm5*	Allowed 0-31 only
Logical shift left by register	R*m*, **lsl** R*s*	

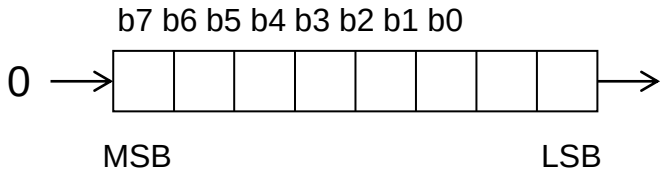
- Logical operations are in addition to Boolean also Shift operations

Boolean operations:

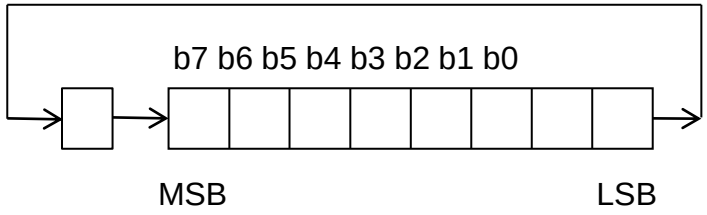
- Although all logical operations can be implemented only with NAND or NOR operation, the majority of computers implement for easer use four Boolean operations
  - two-operand - conjunction (AND), disjunction (OR) and the exclusive disjunction (XOR)
  - single-operand - logical negation (NOT)

Shifts:

- Conventional shift
- Circular shift – rotation
- Both support left or right shift



Right conventional shift



Right circular shift

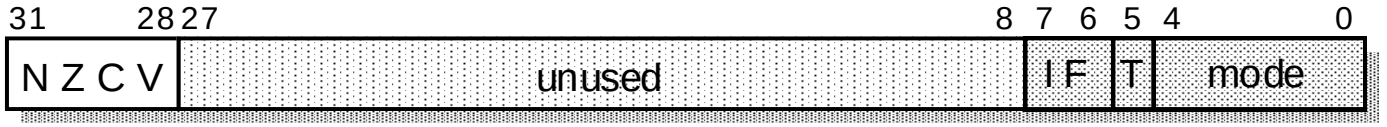
- For right conventional shifts we distinguish:
  - Logical shift – zeros are inserted into the empty positions
  - Arithmetic shift - left most bit (sign) remains unchanged and is expanded into the empty positions
- In binary arithmetic, a right arithmetic shift is equal to the signed division by 2
- For left conventional shift, the arithmetic shift is equal to the logical (multiplication by 2)

- Logical operations are not only used in the calculation of the logic function but more often for example for :
  - Elimination of the individual bits of the word (AND operation or shifts)
  - To insert the individual bits in a word (OR operation or shifts)
  - Generally, when a single word contains bits with different, specific meanings
    - eg. programming of I/O devices

- Example: CPSR register (Current Program Status Register) for ARM processors

- Bit N, Z, C and V – flag bits, status flags

- Flag bits are set to 1 or 0 after arithmetic or logical operation according to the operation result.



- **oVerflow** (bit 28) V = 1: there is an overflow;
  V = 0: no overflow
- **Carry** (bit 29) C = 1: there is a carry;
  C = 0, no carry
- **Zero** (bit 30) Z = 1: the result is 0;
  Z = 0: the result is not 0
- **Negative** (bit 31) N = 0, bit 31 of the result is 0;
  N = 1: bit 31 of the result is 1

4.5.1 Data transfer

- The operation of transferring information from one part of the computer to another is the most elementary operation
- With any transfer, we have the **source** of information and **sink** of informations
- After the execution of the transfer operation is the information at both places, the drain and source. Therefore, it would be more precise to name it as duplication or copying of operands
- Instructions for transfer are for different length of the operands (eg. 8, 16, 32 or 64 bits)

- Most computers have several types of instructions for this operation
- The reason for this is that the operands can be in the CPU registers, in the instruction or in the main memory
- Typically, we use the following mnemonics for data transfer operations:
  - LOAD when transferring from memory to register (ARM9: **`ldr`**)
  - STORE when transferring from register to memory (ARM9: **`str`**)
  - MOVE when transferring from register to register or from memory to memory (ARM9: **`mov`** – transfers between registers or imm. operand to register)
  - PUSH when transferring onto stack (ARM9: **`push,stm`** – transfer from registers to memory)
  - POP (PULL) when transferring from stack (ARM9: **`pop,ldm`** – transfer from memory to registers)

4.5.3 Control operations

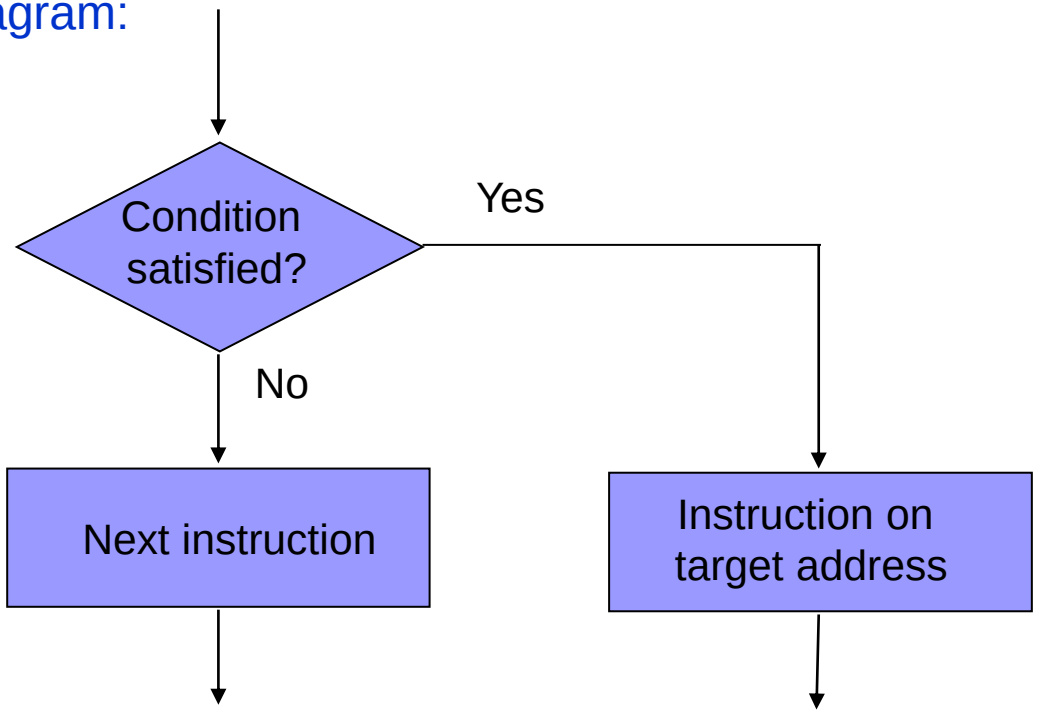
- Usually order of instruction execution is determined by PC ←PC + 1.
- Control operations (instructions) alter the normal order of instruction execution.
- Control instructions include the address of the instruction at which to continue the program execution,

  ⇒ **target address**
- Information on the target address is part of the instruction and is
  - usually given by PC - relative addressing, where offset is given by the instruction and is added to the current value of PC (usually denoted as branch instructions)
  - Cab expressed as absolute address (usually denoted as „jump“ instructions

Control operations are divided into three types:

- **conditional jumps**. Jump to target address is executed only if the condition in the instruction is satisfied, if the condition is not satisfied the next instruction is executed (conventional order of execution)

  - Presentation of the conditional jump instruction in the flow diagram:



- **Unconditional jump.** The jump to the target address is always executed.
  - Typical manufacturers‘ terminology – usually unconditional jumps have the name (mnemonic) **jump** and conditional are named **branch.** Sometimes **jump** is also related to absolute and **branch** to relative address.

- **Call and return from procedures or subprograms.** Procedure or subprogram is a sequence of instructions which execute some specific work and can be called from any part of the program.
  - After the procedure finishes its work, the next instruction after the procedure call has to be executed.

- That's why the call instruction has to save the address into which to return - it is called **return address.**
- Typical mnemonics from procedure call are CALL or JSR (Jump to Subroutine).
- The return address can be saved to some registers in the CPU or on the stack in the main memory.
- The saved return address is used by the return instruction which is the last instruction in the procedure.
- Mnemonics for return instructions are typically RET or RTS.

- ARM9:

  - Unconditional jump

```
        b    address1    @ jump to address1
        ...              @ executes always
        ...
address1 ...
```

  - Conditional jum

```
        beq address2   @ jump to address2
        ...            @ is executed if the
        ...            @ result from the
address2 ...            @ previous operation
                        @ is equal to 0
```

  - Call subroutine

```
        bl   subrout    @ jump to subroutine
        ...             @ lr=r14 return address
subrout ...             @
        ...
        ...
        mov pc,lr     @ vrnitev iz podprograma
```


4.5.4 Floating point operations

- Those operations are typically considered separately although they belong to arithmetic-logical operations.
- The reason is they are typically executed in a separate unit inside the CPU called **floating point unit** (FPU)
- This unit is not a part of the ALU and can typically work in parallel to ALU.
- Intel processors with the Core architecture have 3 ALU and 2 FPU.

- Floating point operations contain:
  - The basic four arithmetic operations
  - and often also dditional arithmetic:
    - Square root
    - Logarithm
    - Exponential functions
    - Trigonometric functions

STM32H7

11.2 **VADD**

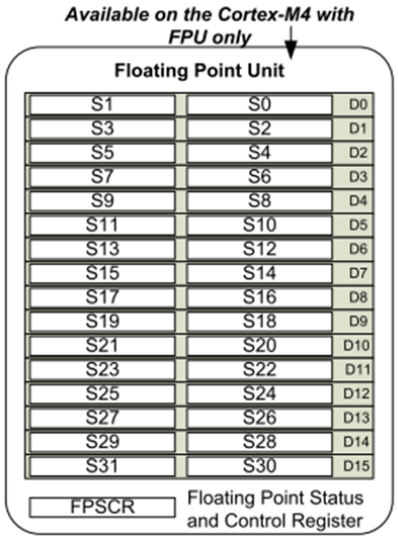
Floating-point Add.

Syntax

```
VADD{cond}.F<32|64> {<Sd|Dd>,} <Sn|Dn>, <Sm|Dm>
VADD{cond}.F64 {Dd,} Dn, Dm
```

Examples

```
VADD.F32 S4, S6, S7
```



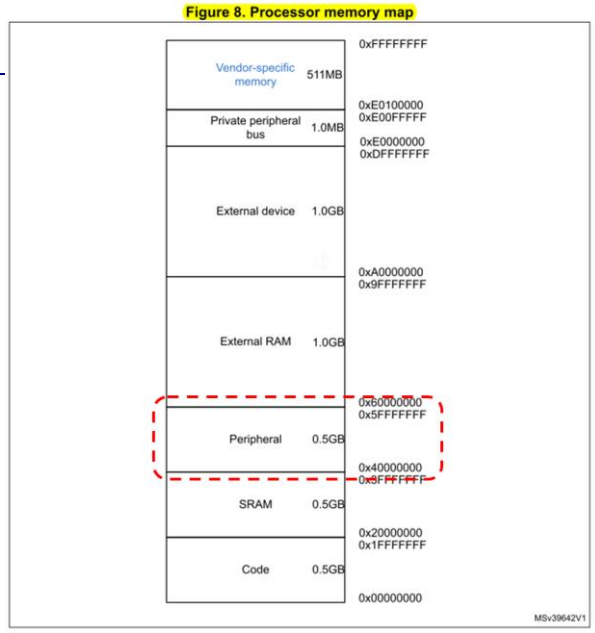
4.5.5 System operations

- Operations with which we change the parameters of the computer operation and monitor its behavior (privileged operations)
- With those operations we can influence on:
  - Interrupts and traps (eg. SWI, …)
  - Operation of the cache
  - Operation of the virtual memory
  - Privilege-level: user/supervisor mode (eg. MRS, MSR (ARM), …)
  - Stop of the operation (eg. HALT, STOP, …)

- On most computers, system operations are treated as privileged instructions.
- Most of the system operations are used only by the operating system and are forbidden for the normal users.
- Example of setting privileged mode on ARM9:

```
mrs r0, cpsr
bic r0, r0, #0x1F  /*clear mode flags */
orr r0, r0, #0xDF /* set supervisor mode
     (0b11111) + DISABLE IRQ, FIQ */
msr cpsr, r0
```


4.5.6 Input / Output operations



- Instructions for input/output operations transfer or trigger a transfer of information between:
  - Main memory and input/output device
  - CPU and input/output device
- Computers that use
  - memory mapped access to input/output devices don't have special instructions for I/O operations. They use normal memory data transfer instructions (i.e. ARM: LDR, STR) to certain addresses.
  - separate I/O address space, have special instructions for I/O operations (e.g. LOADIO, STOREIO are accessing memory of I/O devices
- I/O instructions are typically privileged and are only used by the system programs, e.g. operating system.

4.5.7 Scalar, vector and SIMD instructions

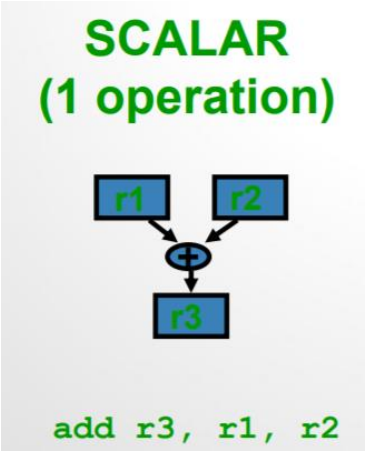
- Arithmetic, logical operations and floating point operations can also be executed by **scalar**, **vector and SIMD operations.**

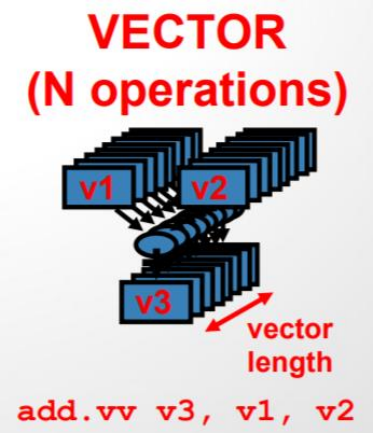
- Typical computers have scalar instructions, operations are executed on operands which are given by the instruction.
  - Vector instruction executes an operation on a sequence of *N* operands.
  - For vector instructions it's not necessary for *N* operations to execute in parallel, but anyway in this case, only one instruction is necessary to be read.

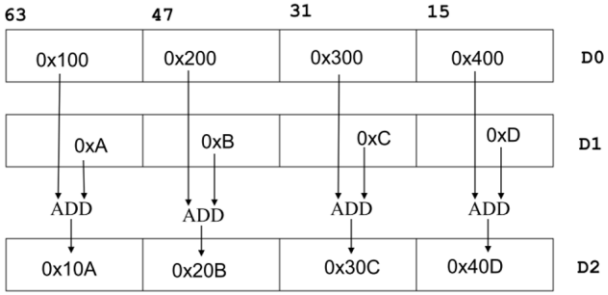
- SIMD commands mean the parallel execution of N operations simultaneously on N pairs of fixed-width operands

Arithmetic, logical operations and floating point operations can also be executed by **scalar**, **vector and SIMD operations.**

- **Scalar instr.**
  - ADD R3,R1,R2
    - 1 instr.: 1 operation, 1 result
    - for N operations, N instructions
- **Vector instr.**
  - ADDV V3,V1,V2
    - 1 instr.: N operations, N results
    - Pipelined/parallel execution
    - „older supercomputers– e.g. Cray“
    - N - variable length of vectors
- **SIMD instr.**
  - VADD.U16 D2,D1,D0
    - ARM NEON SIMD unit
    - 1 instr.: N operations, N results
    - SIMD, parallel,
    - fixed width (N=2,4,8,…)







STM32H7

```
SADD16 R1, R0      ; Adds the halfwords in R0 to the corresponding
                   ; halfwords of R1 and writes to corresponding halfword
                   ; of R1.
SADD8  R4, R0, R5  ; Adds bytes of R0 to the corresponding byte in R5 and
                   ; writes to the corresponding byte in R4.
```


4.6 Type and size of operands

- For operands we wish to handle all types of operands that are also present in higher level programing languages

- The most common types of operands are:
  - Bit
  - Character
  - Integer
  - Real number
  - Decimal number

- **Bit.** One bit operands are not present in most higher level programing languages, but they are useful for system and I/O functions.

- **Character.** 8 bits in size (also 16 bits), represented in the ASCII, EBCDIC or Unicode table. Used also in strings with various lengths.

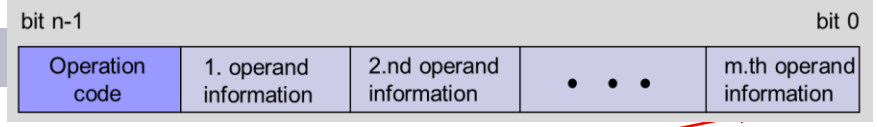
- **Integer.** 8, 16, 32 or 64 bits in size. These operands are typically represented as signed numbers with fixed point (in two's complement).

- **Real number.** Floating point numbers (standard IEEE 754), with 32, 64 or 128 bits in size.

- **Decimal number.** String of 8-bits characters in:

  - Unpacked form (one number in ASCII or EBCDIC in 8-bits character)

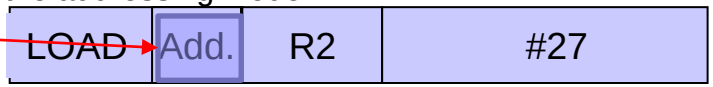
  - Packed form (two BCD numbers in 8-bits characters). Because the highest four bits in ASCII are the same for numbers from 0 to 9 (0011XXXX) we can omit them in BCD format

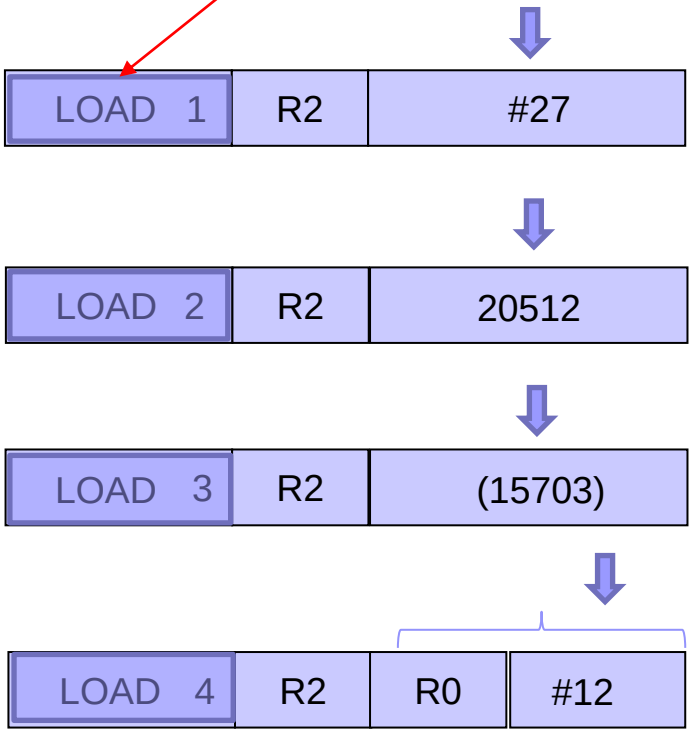


Operand location and addressing modes

How to determine how the operand is given in this field ?

- different operation codes for the LOAD instruction
- or same operation code and special bit that determines the addressing mode





27_D = immediate operand

20512_D = direct memory address

15703_D = indirect memory address

R0 = base register
12_D = offset

R2 = direct register address

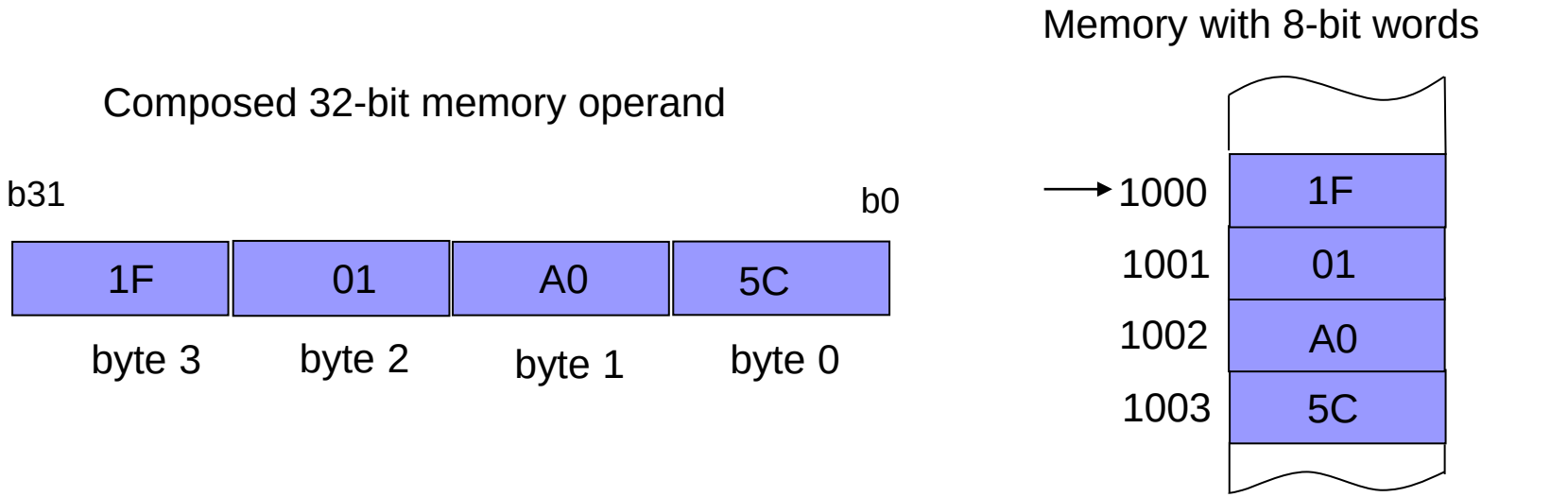
Composed memory operands

- The length of the operands in bits is a multiple of 8 because nowadays the memory word is typically 8 bit long.
- Operands that are longer then 8 bits occupy more memory words, they are called **composed memory operands.**
- We can have 32-bits operands on a computer with 8-bits memory word and they occupy four sequential memory words.

- Word must be sequential so we can address the operand with one address (address of the first one)
- It’s necessary to agree on which of the four words points the address and what is the order of saving them
- For saving composed operands in the memory we nowadays use two common rules:
  - Big Endian Rule
  - Little Endian Rule

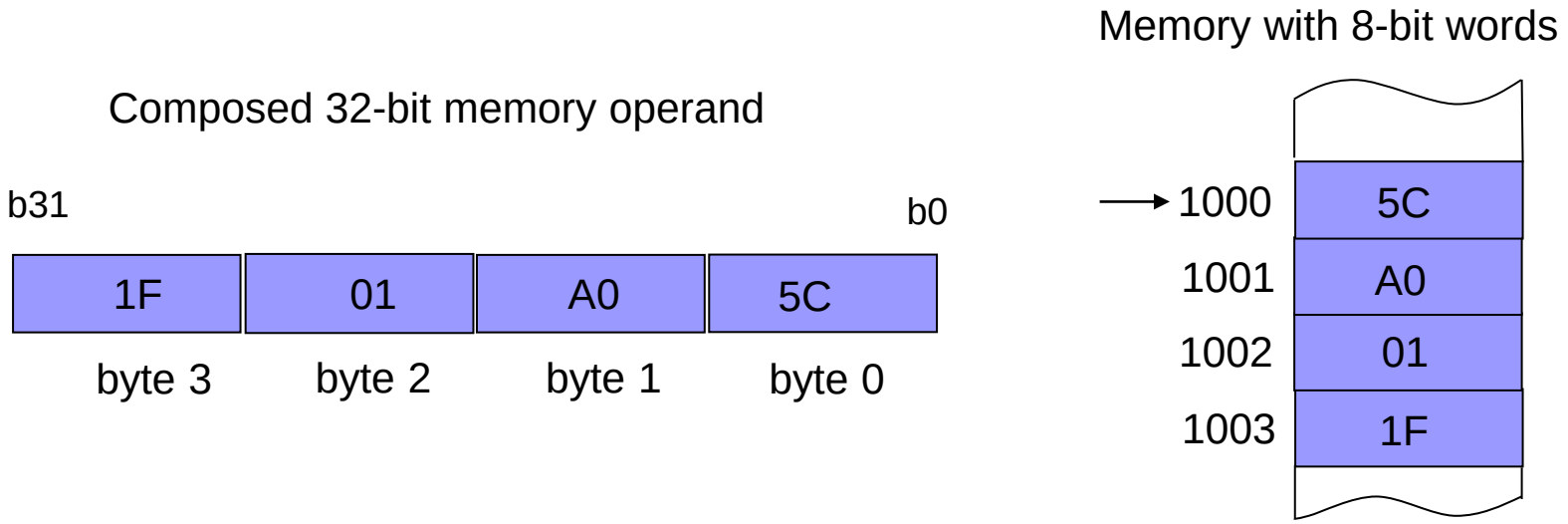
Big Endian Rule

- Address of the composed operand is the address of the word containing the most significant byte of the operand
- Example: 32-bit composed memory operand *0x1F01A05C(hex)*, saved in the big endian rule on the address *1000(dec)*:



Little Endian Rule

- Address of the composed operand is the address of the word containing the least significant byte of the operand
- Example: 32-bit composed memory operand *0x1F01A05C(hex),* saved in the little endian rule on the address *1000(dec)*:

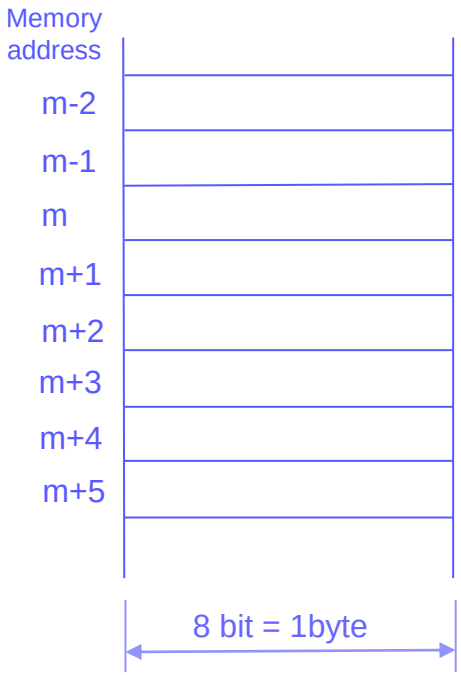


Example of an 32-bit composed memory operand

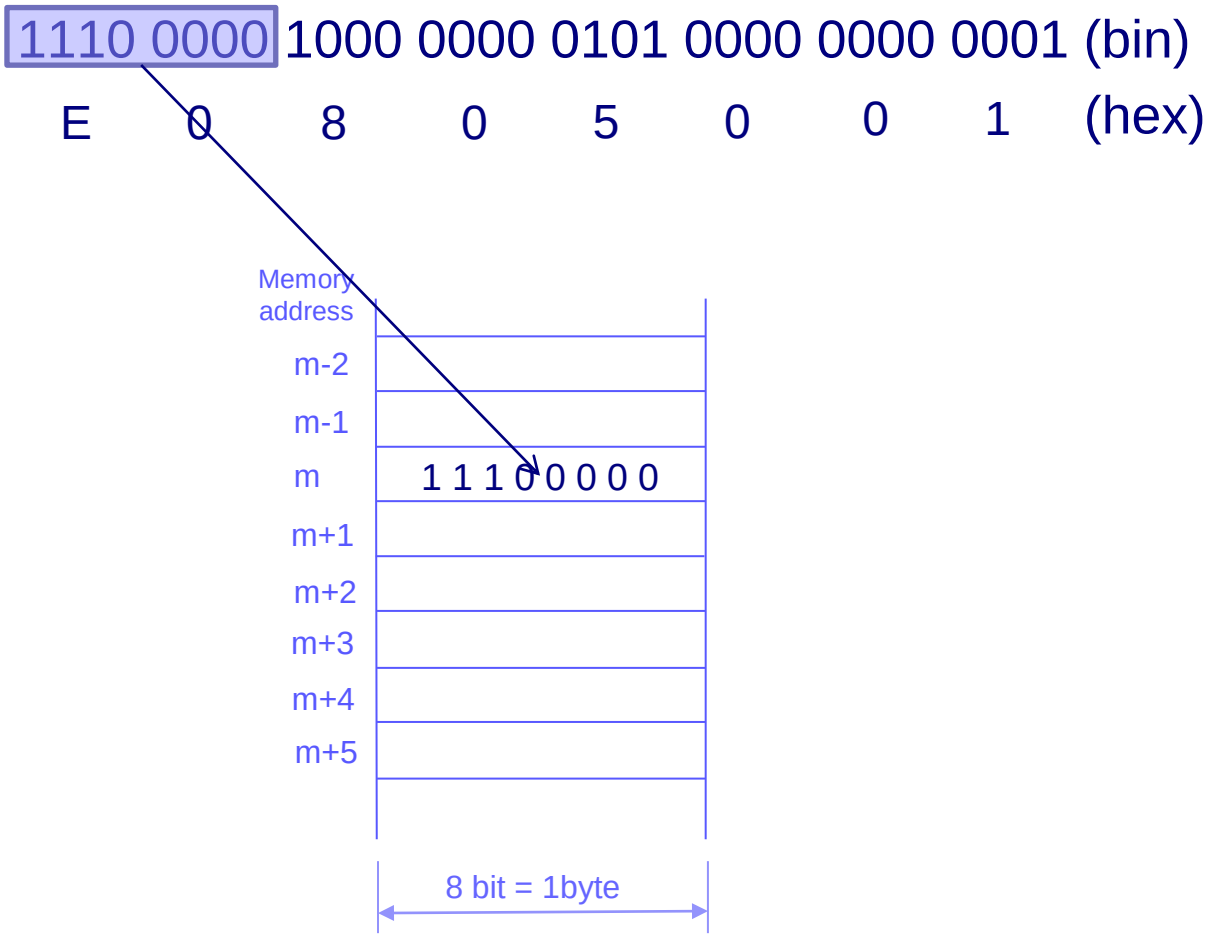
32-bit combination:

1110 0000 1000 0000 0101 0000 0000 0001 (bin)

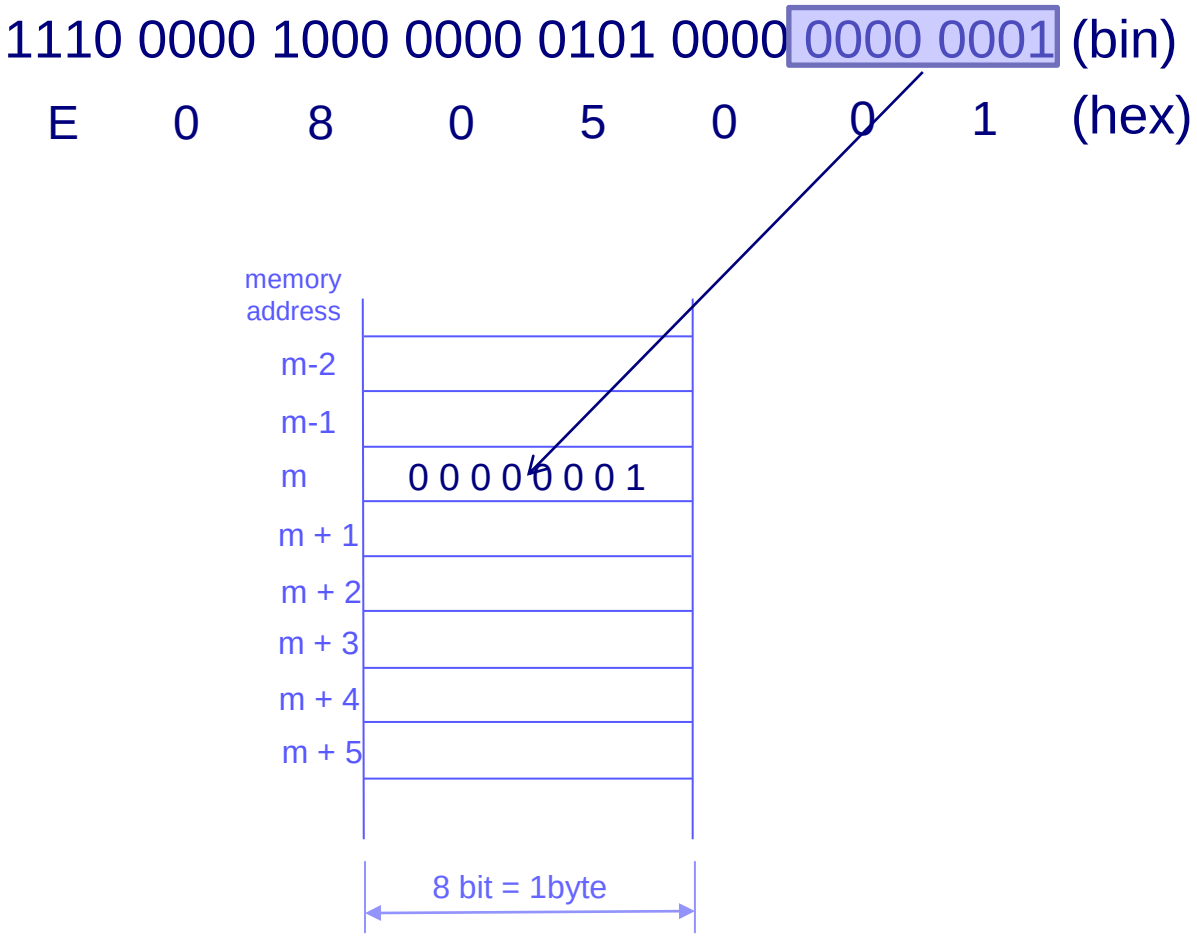
E 0 8 0 5 0 0 1 (hex)



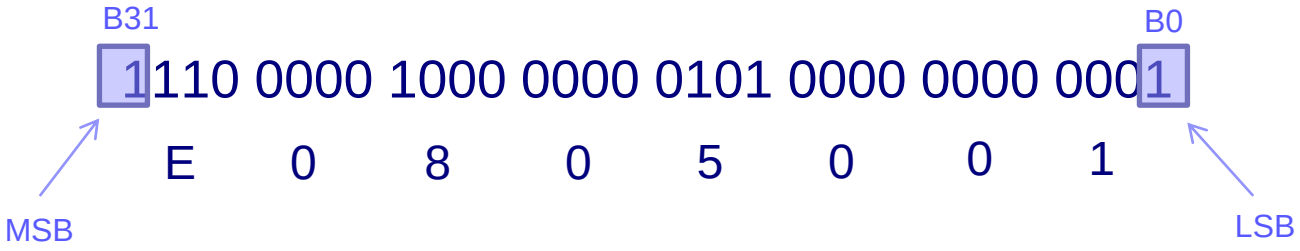
Example of an 32-bit composed operand

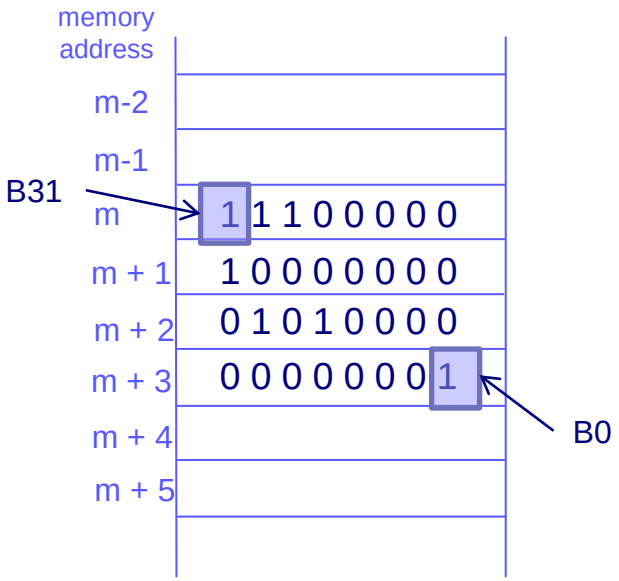


Example of an 32-bit composed operand

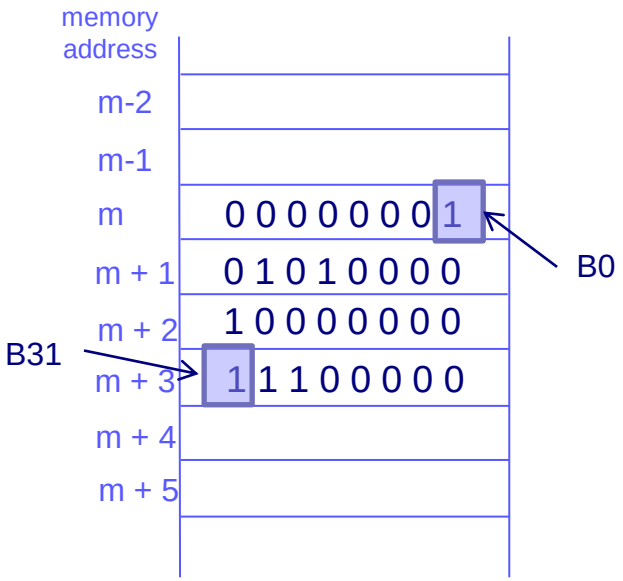


Example of an 32-bit composed operand





Big Endian Rule



Little Endian Rule

- **The problem of alignment** of composite memory operands
  - Operand in the memory is an **aligned operand** when the following holds:

    $$A \bmod s = 0$$

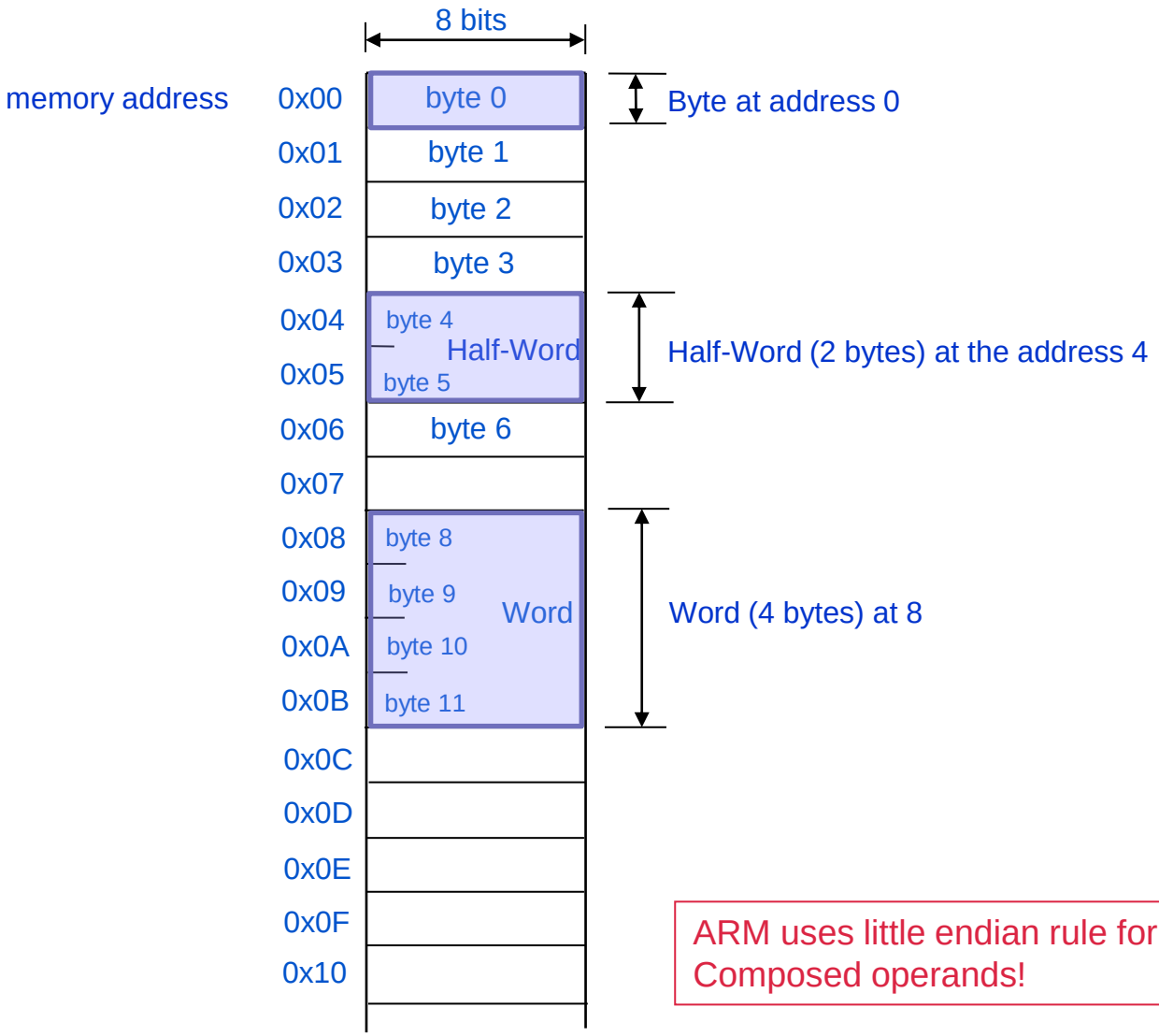
    A - address of the composed memory operand

    s - number of words of the composite memory operand

  - If the above equation holds, then A is a **natural address**

  - ARM Processor:
    - see pseudo instruction `.align`
    - saves composite memory operands (more than 8 bits), according to the little endian rule. Composed memory operands have to be on aligned addresses.

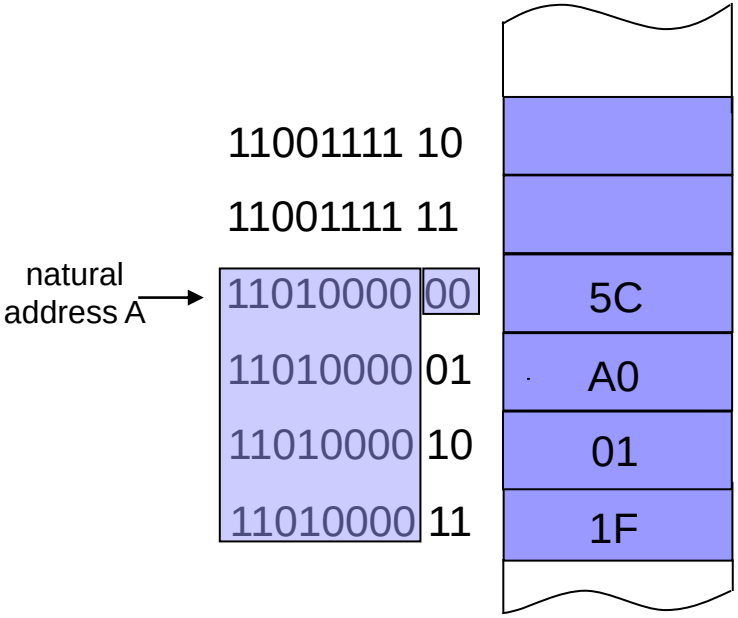
Organization of main memory in ARM processor

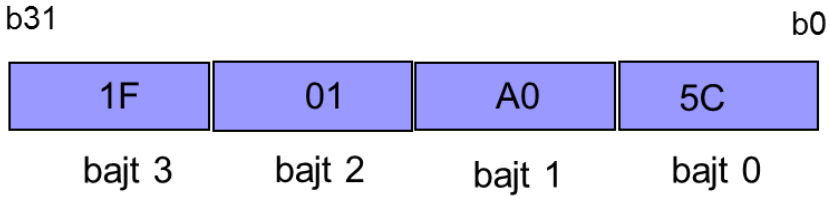


ARM uses little endian rule for Composed operands!

- For our use case (ARM9), the address of our 32-bit composed memory operand must be divisible by four without reminder for the operand to be aligned (eg.$1000_D \ mod \ 4 = 0$)
  - The memory, which allows access to 4 memory words at the same time can be implemented as 4 parallel working memories.
  - If the 32-bit operand is aligned, the lower two bits of a *n*-bit memory address determine in which of the 4 memories the individual bytes of the composite operand are located, the remaining *n-2* bits of the memory address are for all 4 bytes the same.

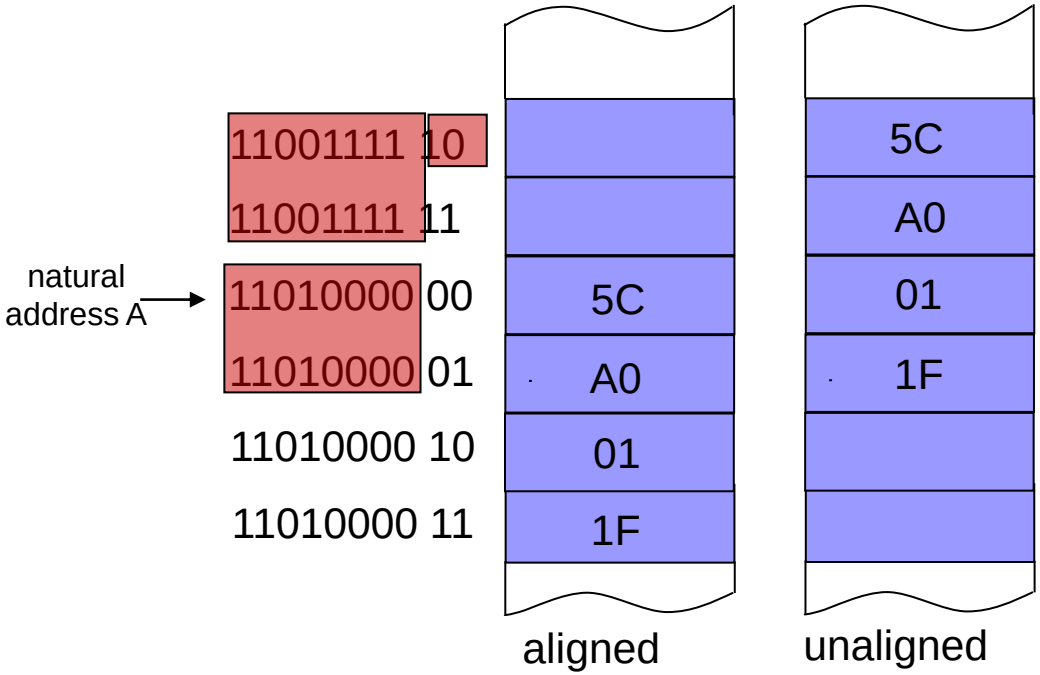
- That's how it's possible to access all four memories (bytes) in parallel
- However, if the 32-bit operand is not aligned, the remaining *n-2* bits of the memory address are not the same for all 4 bytes and the simultaneous access to each of the four bytes is not possible.





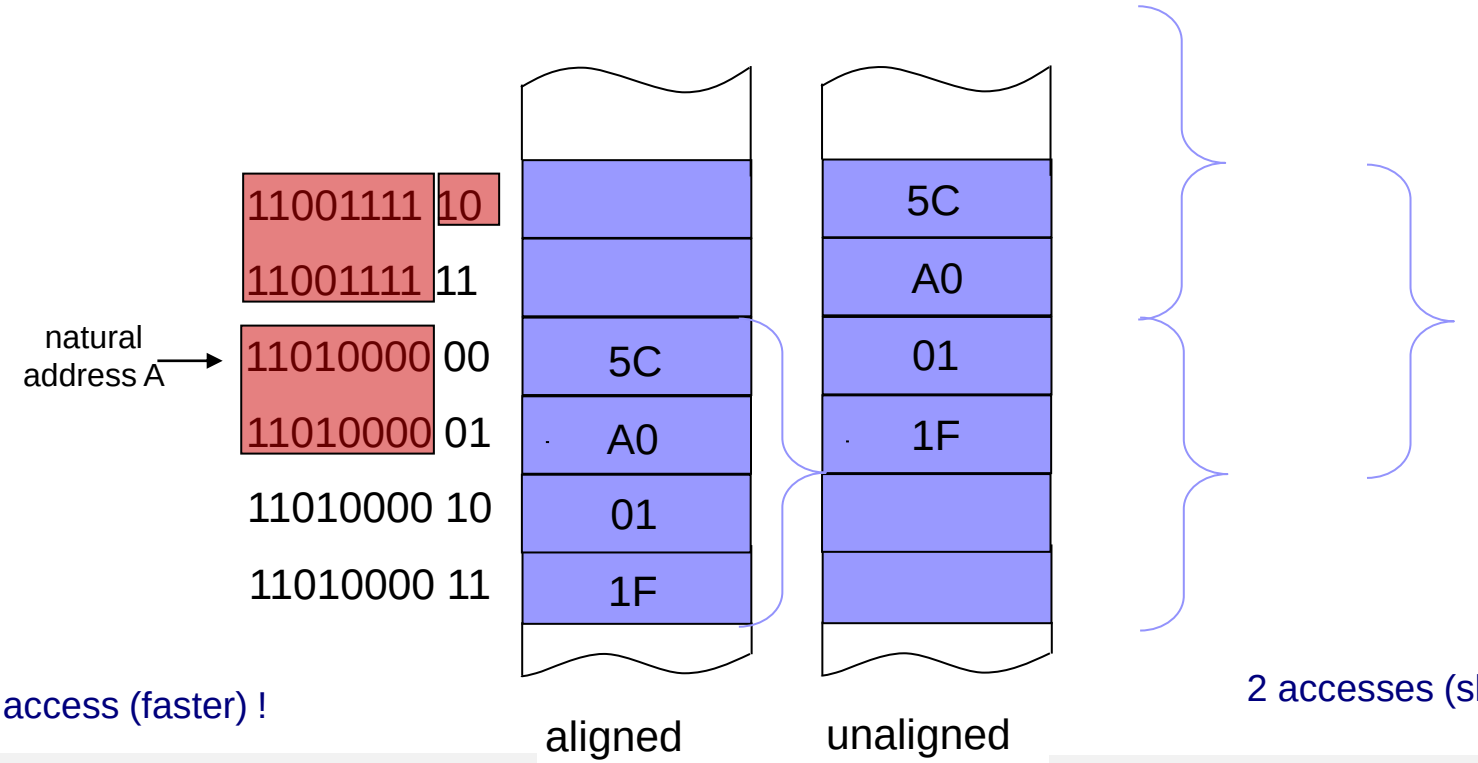
Aligned composed memory operand saved by the little endian rule

- That's how it's possible to access all four memories (bytes) at the same time.
- However, if the 32-bit operand is not aligned, the remaining *n-2* bits of the memory address are not the same for all 4 bytes and the simultaneous access to each of the four bytes is not possible.



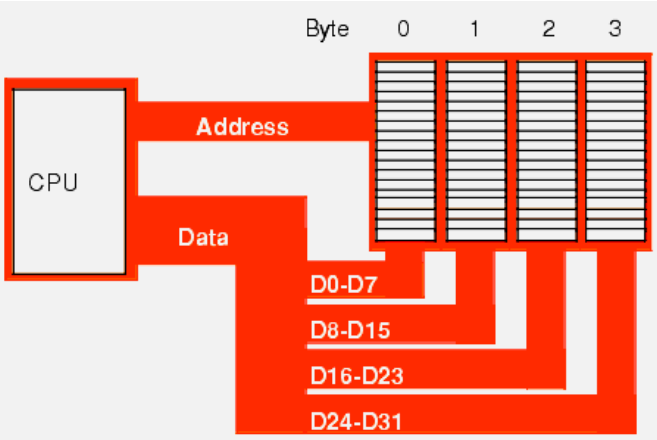
In natural addresses are the lower two bits equal to 0, therefore it is divisible by 4 without a remainder

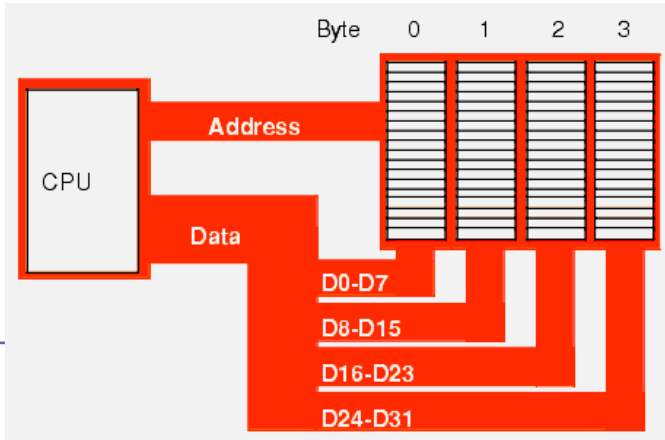
Type and size of operands - composed memory operands



1 access (faster) !

2 accesses (slower) !





Cases of different computer memory organization

Unified memory

(32Kx32b)

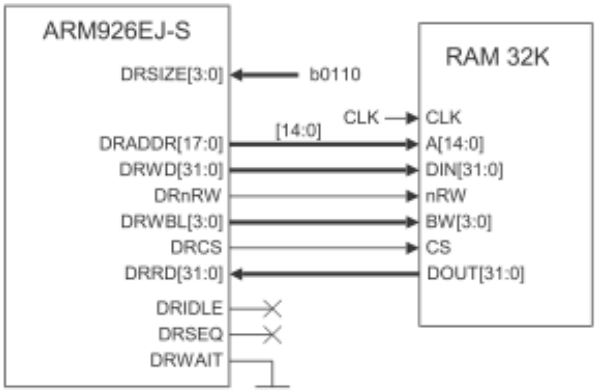


Figure 5-12 Zero wait state RAM example

4 modules per 32Kx8b

4*(32Kx8b)

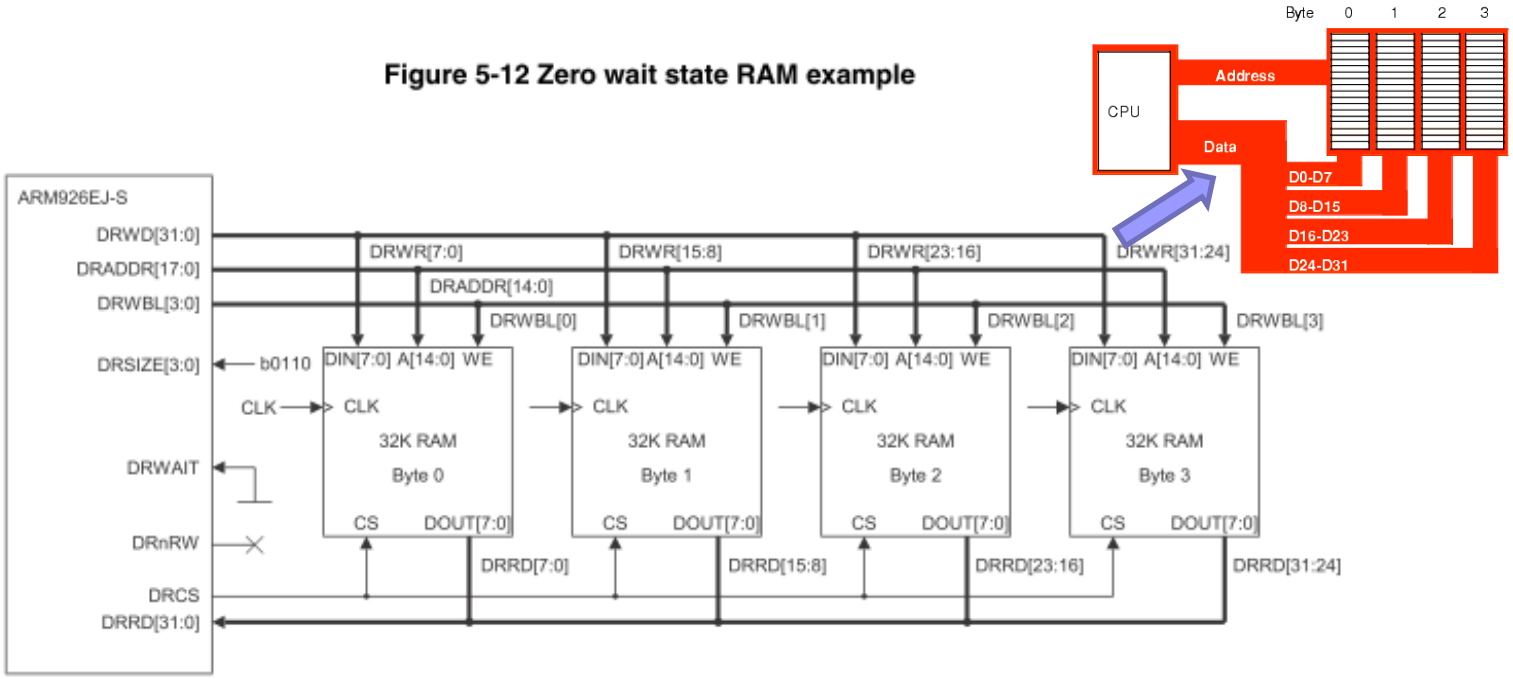


Figure 5-13 Byte-banks of RAM example

Memory address of 8-bit memory words on which the composed memory operands are aligned

Size of the composed operand in bits	Address A on which the composed operand is aligned
8 (not composed)	XXX…XXXXXXXX
16	XXX…XXXXXXX0
32	XXX…XXXXXX00
64	XXX…XXXXX000
128	XXX…XXXX0000

X = 0 or 1

4.7 Instruction format (structure)

- The properties which were selected by developers, have to be build into the instruction
- The selected properties determine the instruction format to some extent, but it's still possible to construct an instruction in different ways
- The most important factors that determine the instruction structure are:
  - **The size of the memory word**. The size of the instruction should be a multiple of the memory word size.

- **Number and type of explicit operands in instruction**. For each explicit operand we must specify where and how it is placed in the instruction

- **The type and number of registers in the CPU**. Registers can be equivalent and equal in size but can also have different size for different purposes. The number of bits in the instruction for the register addressing is dependent on the number of registers

- **Size of the memory address.** In direct memory addressing, the instruction includes the memory address; that's why the direct memory addressing is not often used. An exception are processors with short memory addresses (eg. 68HC11 - 16-bit address)

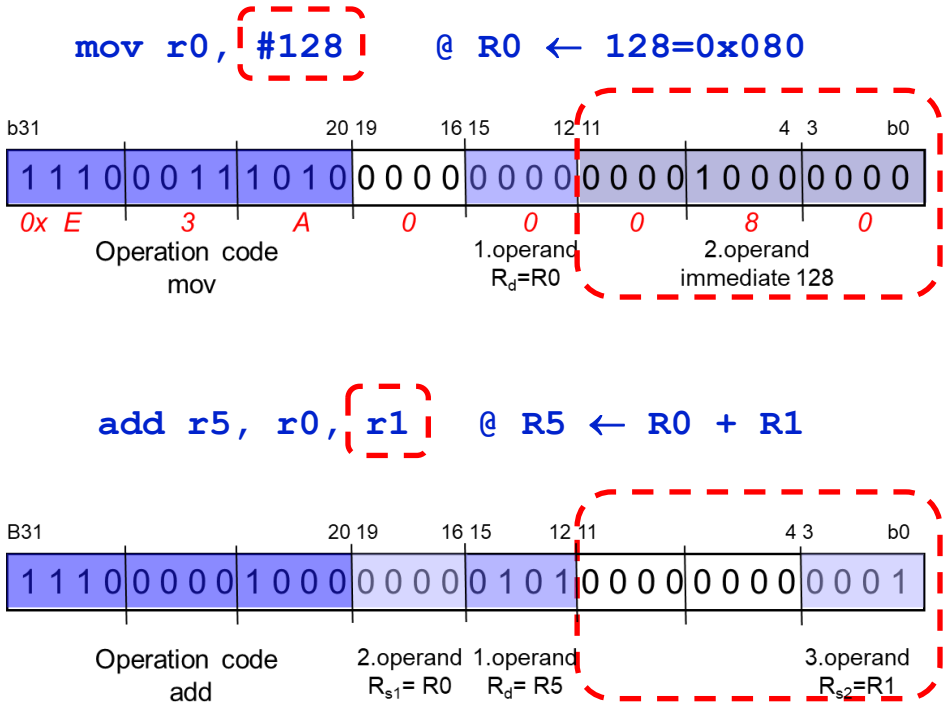
- The most commonly used instructions usually have the shorter instruction formats
- The order of the information on operations and operands can be arranged in different ways
- Example of non-systematic instruction structure are Intel microprocessors, where additional instructions needed to be integrated into the existing ones.
- The result is a large number of unusual and hard to understand instruction formats

- Typical modes of building instructions in today's computers:
  - Variable Length instruction
    - Number of explicit operands in the instructions is changing, many different formats of instructions (Intel, AMD - x86: 1 to 15 bytes)
  - Fixed length instructions
    - Number of explicit operands in instructions is always the same, few instruction formats (PowerPC, SPARC, ARM)
  - Hybrid mode
    - A few different fixed length instructions (IBM 370, ARM Thumb2)
      - ARM Thumb2 (16 or 32 bits instructions) – e.g. STM32H7

- **Orthogonality of instructions** – instruction structure to which applies:
  - Information about the operation is independent of the information on the operands
  - Information about each operand in the instruction is independent of the information about the rest of the operands

  With orthogonal instructions we can used different types of addressing for each operand and all sizes of operands.

- ARM9: Case of limited orthogonality on one of the operands:



4.8 Number of instructions in RISC - CISC computers

Debates on the number of instructions in computers started after the year 1980

- **CISC computers** (Complex Instruction Set Computer) – computers with a greater number of (even complex) instructions

- **RISC computers** (Reduced Instruction Set Computer) – computers with a lower number of simple instructions

- The computer development showed that the number of instructions was constantly increasing

- But measurements on the frequency of instruction execution on CISC computers shows that the majority of instructions are used very rarely

- And the most often used instructions were the simple ones with simple addressing

- Reasons to increase the number of instructions (up to 1980):
  - **Semantic gap** – the difference between how programmer in a higher-level programming language sees the computer in comparison to a programmer in machine language.
  - **Microprogramming** – allows simple addition of new instructions
  - **The ratio between the speed of the main memory and the CPU** – in the years 1960 to 1980 was the speed of access to information in the CPU (to microinstructions) more the 10-times higher then access to main memory.

- Reasons to decrease the number of instructions (after 1980):
  - **Difficulties of using complex instructions in compilers** – its better that the architecture enables simple elements for solving problems than solutions that are complex and often useless
  - **Change in the ratio between the speed of the main memory and the CPU** – microprogram solutions have become slower compared to fixed wired solutions, and complex instructions difficult to realize in a fixed wired logic; caches have bridged the speed gap between CPU and main memory.
  - **The introduction of parallelism in the CPU** – the realization of the pipelines is easier for simple instructions than for complex

The idea of RISC computers (Milestones)

- The first RISC computer was the IBM 801 from 1975
  - On determinaton of the instruction set, two criteria were used:
    1. The instruction must e simple enough to be executed in one clock cycle.
    2. The instruction executes such operation that is not possible to realize faster with some sequence of instruction that are generated by a compiler, that has a higher level of understanding the program
- 1980: Berkeley (RISC I,II), Stanford (MIPS)
- 1985: ARM1
- 2011: RISC-V

- **Definition of RISC architecture.** A computer has a RISC architecture, if it meets the following six criteria:

1. Most of the instructions are executed in one clock cycle
2. Register-register design (load/store computer)
3. Instructions are realized with fixed wired logic and not microprogramed
4. Small number of instructions and addressing modes
5. All instructions have the same length
6. Good compilers (take into account the structure of the CPU)

- The RISC architecture concept means more than just the small number of instructions. In fact this criteria is nowadays the least taken into account.

- Basically majority of all after 1990 developed computers are RISC

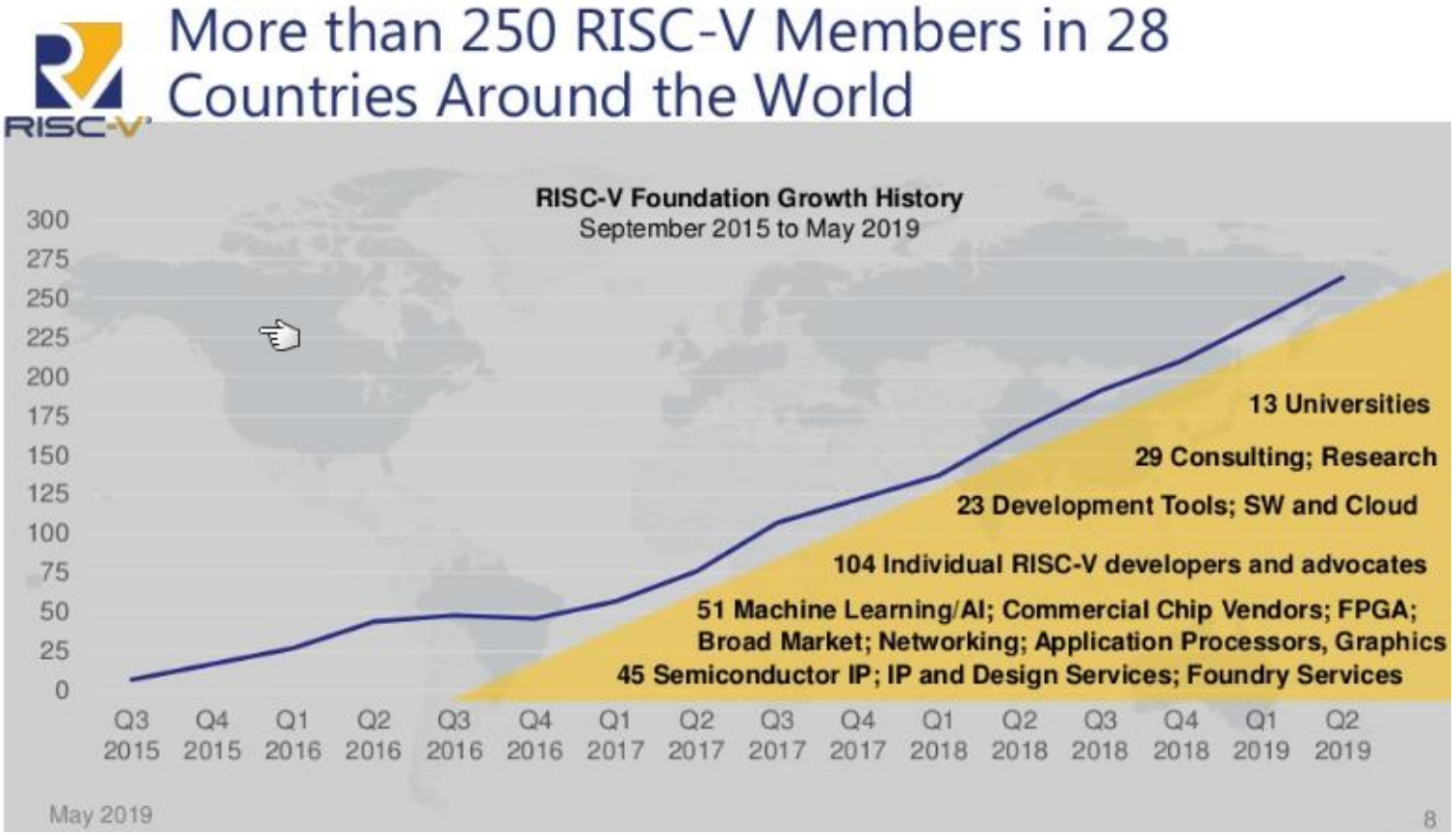
- But that doesn’t mean there are no more CISC computers
  - Intel 80x86, AMD

- Intel has succeeded to integrate ideas from RISC into his CISC based architecture of Pentium processors
- From the 80486 processor onwards, Intel processors use a „RISC like core“ which executes simple (and most often used) instructions
- CPU translates CISC instructions in simpler (RISC like) instructions – „micro operations“.
- On the other hand, more complex and powerful instructions are executed according to special sequence of micro-ops, defined on a microprogram level

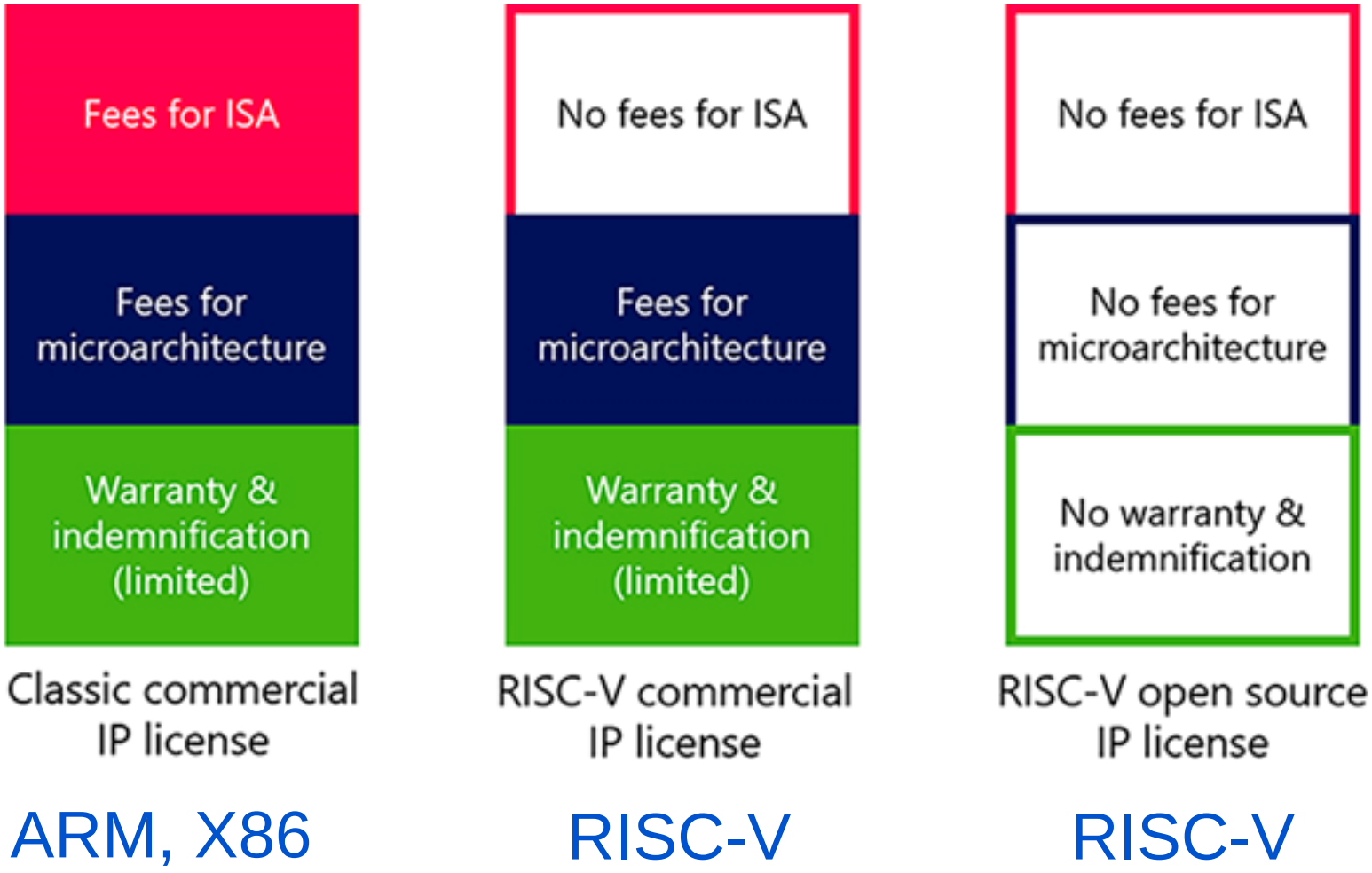
RISC-V (https://riscv.org/)

RISC-V: The Free and Open RISC Instruction Set Architecture

RISC-V is a free and open ISA enabling a new era of processor innovation through open standard collaboration. Born in academia and research, RISC-V ISA delivers a new level of free, extensible software and hardware freedom on architecture, paving the way for the next 50 years of computing design and innovation.



Commercial (license) models comparison



https://riscv.org/wp-content/uploads/2021/01/Codasip_Open-Source-Vs-Commercial-RISC-V-Licensing-Models-fig1.png