Development of Complex Curricula for Molecular Bionics and Infobionics Programs within a consortial* framework**

Consortium leader

PETER PAZMANY CATHOLIC UNIVERSITY

Consortium members

SEMMELWEIS UNIVERSITY, DIALOG CAMPUS PUBLISHER

The Project has been realised with the support of the European Union and has been co-financed by the European Social Fund ***

**Molekuláris bionika és Infobionika Szakok tananyagának komplex fejlesztése konzorciumi keretben

***A projekt az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

ELECTRICAL MEASUREMENTS

(Elektronikai alapmérések)

Complex logic problems, basics of microcontrollers

(Összetett logikai feladatok, mikrokontrollerek működésének alapjai)

Dr. Cserey György

Basics of microcontrollers

- What is a microcontroller? A microcontroller is a computer optimized usually for control tasks, integrated on one single chip.
- It is a chip that contains
  - Central processing unit (CPU), this is responsible for the logical operations
  - Input and output interfaces (I/O)
  - Program and data memory
  - Other extras… from which some can be switched off and on even while running a program, to avoid unwanted consumption

Microprocessor vs. microcontroller

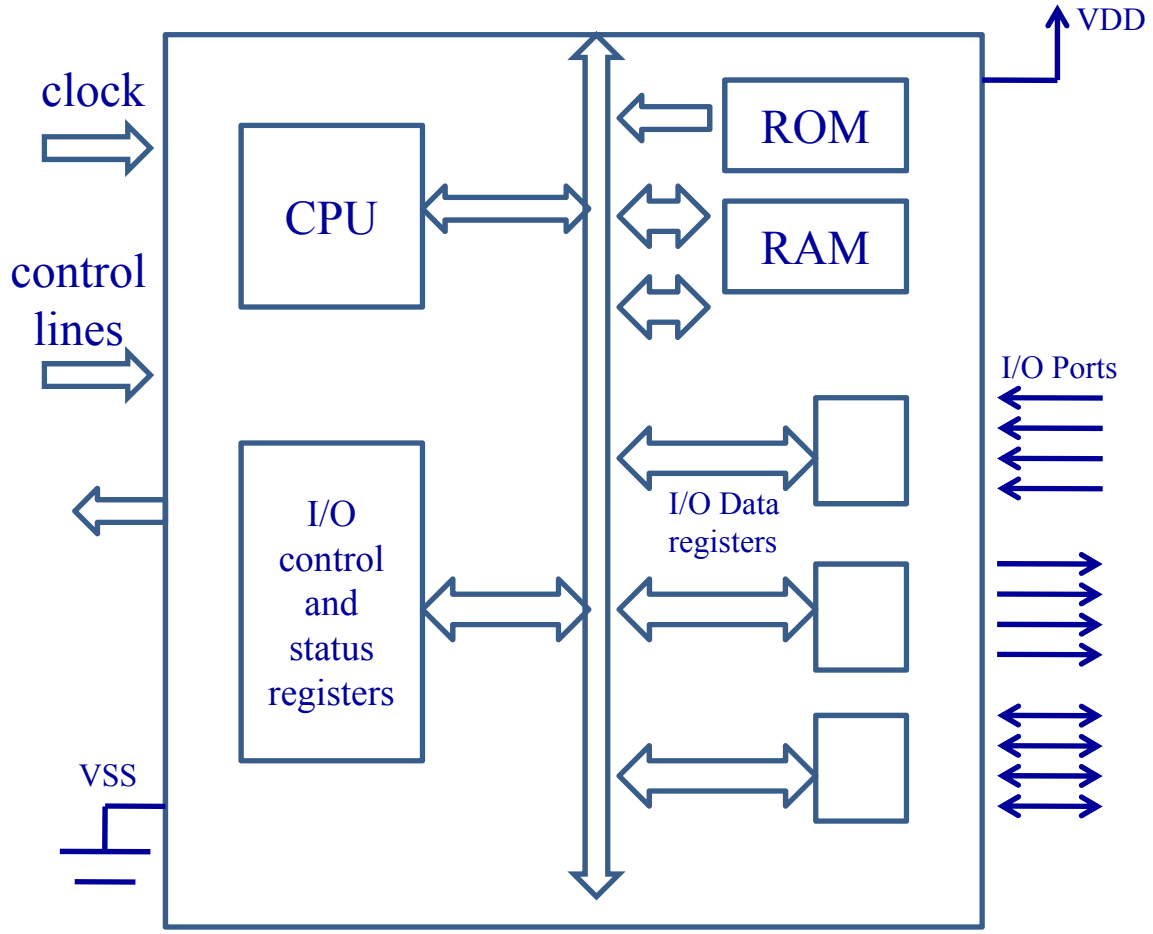
Microprocessor

- CPU is separated from RAM, ROM and I/O
- It is up to the developer to decide how many ROM, RAM and I/O ports he wants to use.
- Widespread
- Versatile, used for general purposes

Microcontroller

- CPU, RAM, ROM, I/O on one chip
- The number of ROM, RAM, I/O ports is defined
- Applied for tasks that are critical in cost, energy and space
- Optimized for tasks

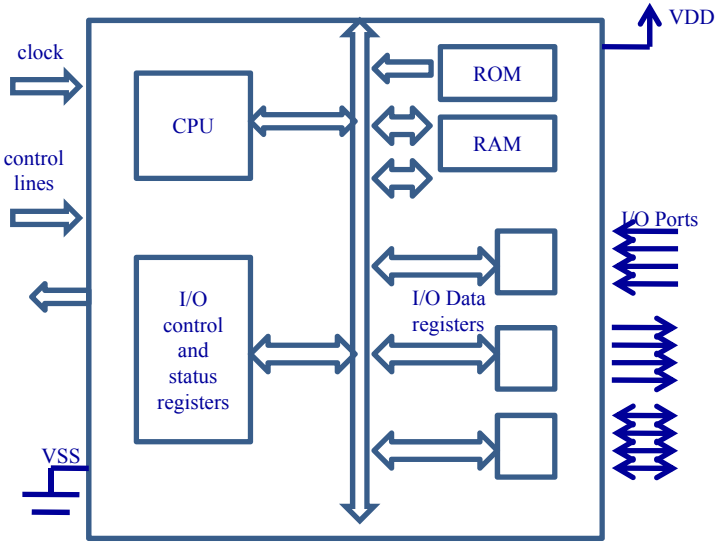
Block diagram



Central Processing Unit

A CPU contains:

- Control Sequencer
- Program Counter
- Instruction Decoder
- Arithmetic and Logic Unit (ALU)
- Registers



Instructions

- NOP ;there is no operation, the microcontroller waits, it can be used for timing
- MOV Rs,Rd ;data transfer from Rs ➔ Rd, where Rs and Rd can be a specific binary or hexadecimal data, memory register
  - MOV A,R06
  - MOV #2,R07
  - MOV #44H,R08
- .equ ;text replacement (#define)
  - C .equ R02

Operations I.

- Addition – between source and destination registers
- ADD Rs,Rd ; Rs + Rd ➔ Rd
  - Handling of addition and transmission (overflow) with the help of the carry bit
- ADC Rs,Rd ;Rs + Rd + Cy ➔ Rd
- Subtraction – between source and destination registers
- SUB Rs,Rd ;Rd - Rs ➔ Rd
- CMP Rs,Rd ; there is no result, just a comparison: which is bigger, which is smaller
  - Handling of transmission with the help of the carry bit
- SBB Rs,Rd ;Rd – Rs - 1 + Cy ➔ Rd

Operations II.

- Addition – fast increase with 1
- INC Rd ; Rd + 1 ➔ Rd
- INCW #iop8,Rp ; Rp +#iop8 ➔ Rp
  - Handling of addition and transmission (overflow) is possible with the help of the carry bit
- Subtraction – fast, reduction, subtracting 1
- DEC Rd ;Rd - 1 ➔ Rd
  - Handling of subtraction and transmission (underflow) with the help of the carry bit
- It sets zero
- CLR Rd ; 0 ➔ Rd

Operations III.

- Multiplication
- MPY Rs,A ; Rs * A ➔ (A:B)
- Division (of integers)
- DIV Rs,A ;(A:B) / Rs ➔ A; B
  - A result, B remains

Logical operations

- AND Rs,Rd ; Rs & Rs ➔ Rd
- OR Rs,Rd ; Rs | Rs ➔ Rd
- XOR Rs,Rd ; Rs ^ Rs ➔ Rd
- INV Rd ; ~Rd ➔ Rd
  - Does not make a Cy flag

Jump I.

- JMPL label16 ; unconditional jump
- JMP label8 ; unconditional jump

- Conditional jumps – a few examples
- JNZ label8 ; jump if no zero
- JZ label8 ; jump if zero
- JNC label8 ; jump if no carry
- JC label8 ; jump if carry
- DJNZ Rn,label8 ; decrement and jump no zero
- JEQ label8 ; jump if equal
- JG label8 ; jump if greater
- JNV label8 ; jump if no overflow

Jump II.

- Subroutine
  - CALL label ; unconditional call
  - CALLR címke ; unconditional call
    - Return address will be saved in the stack
- Return
  - RTS ;return from subroutine
  - RTI ;return from interrupt
- Stack handling
  - PUSH d ;d ➔ (SP) The value of d will be put in a place appointed by the stack pointer
  - POP d ;(SP) ➔ d Where d will have the value that the place appointed by the stack pointer has.

Handling bits

Bit (data bits; flag bits)

- Data bit of any register, the programmer is setting these
  - zero . dbit 0,R00
  - first . dbit 1,R00
  - second . dbit 2,R00
  - …

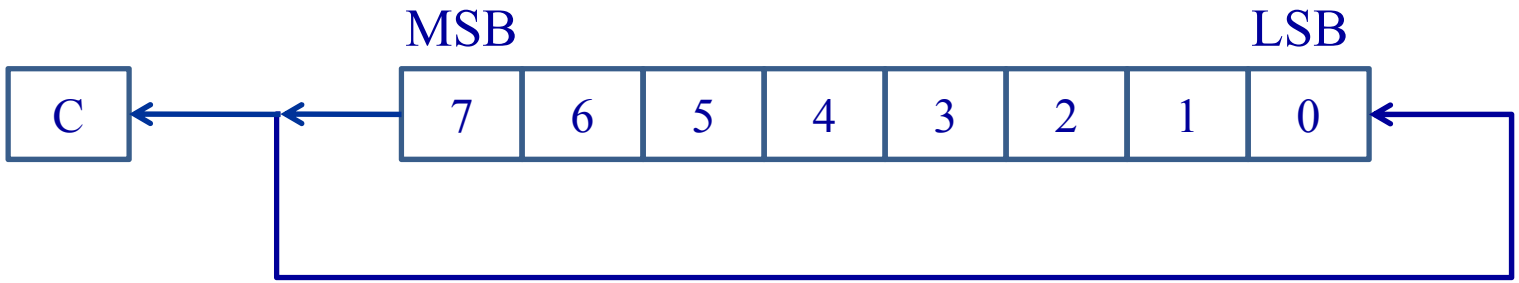
```
SBIT0 name   ; 0 → name
SBIT1 name   ; 1 → name

JBIT0  name,label8
JBIT1  name,label8
BTJZ   s1,s2,label8 ;s1 & s2
```

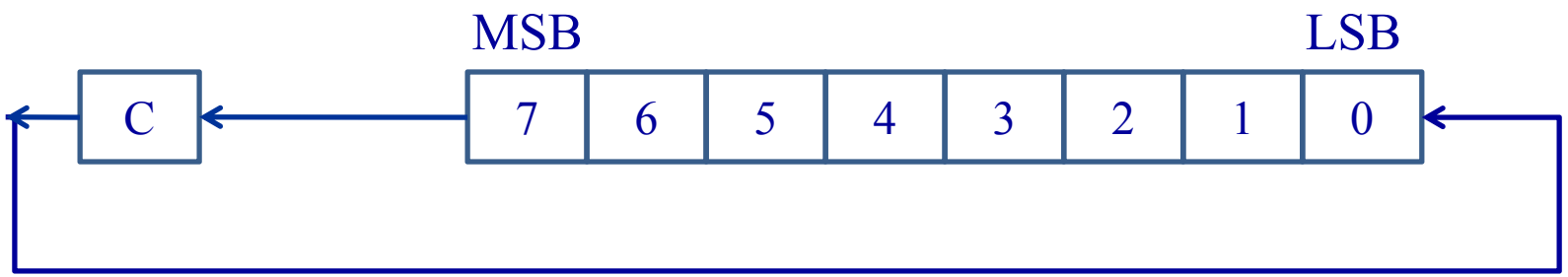
- Carry, Zero, Negative, Overflow, Interrupt Enable (1,2)
  - Usually the setting of these is done by the operations
  - SETC ; 1 → Cy
  - CLRC ; 0 → Cy

Rotate

RL Rd ;rotate left

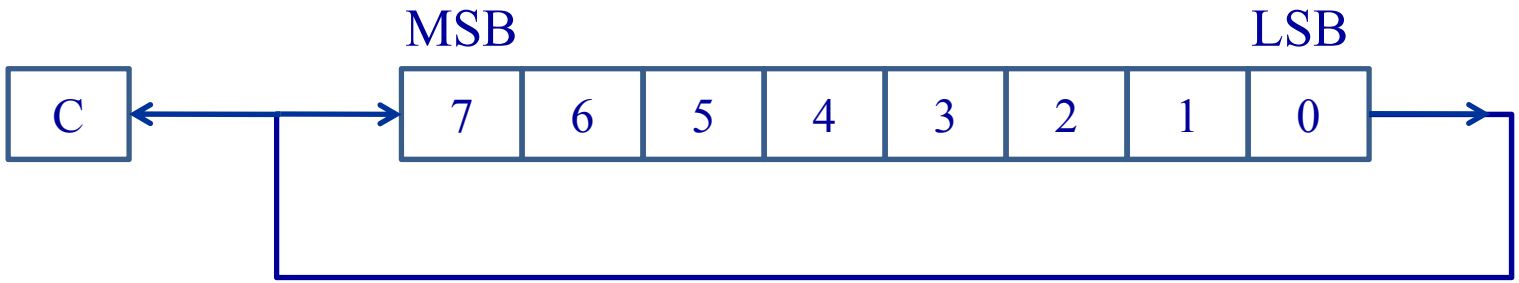


RLC Rd ; Rotate Left Through Carry

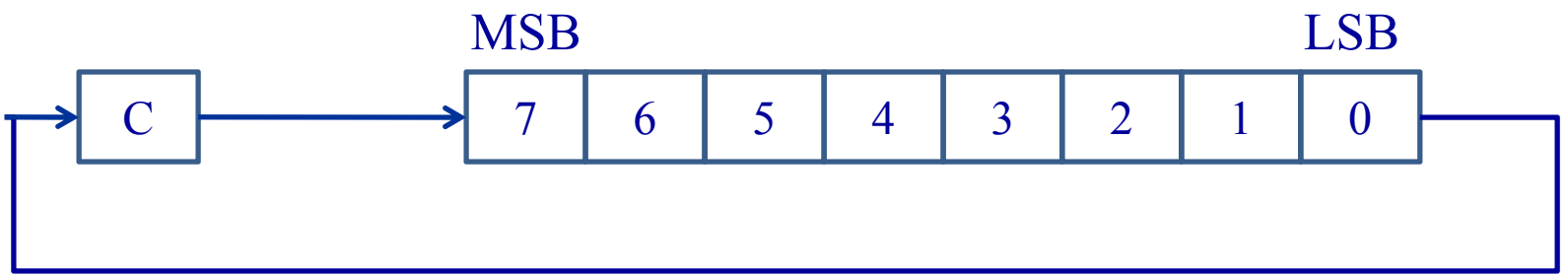


Rotate

RR Rd ;rotate right



RRC Rd ; Rotate Right Through Carry



Flag bits

Bit 7 C. Carry.

- This bit is set by arithmetic instructions as a carry bit or as a no-borrow bit. It is also affected by the rotate instructions.

Bit 6 N. Negative.

- This bit is set to the value of the most significant bit (sign bit) of the result of the previous operation.

Bit 5 Z. Zero.

- This bit is set by the CPU if the result of the previous operation was zero; cleared otherwise.

Bit 4 V. Overflow.

- This bit is set by the CPU if a signed arithmetic overflow condition is detected during the previous instruction. The value of this flag is significant at the completion of the following instructions: ADD, ADC, INCW, INC, DEC, CMP, SBB, SUB, and DIV.

Special operations

- SWAP Rd ; exchage Rd(7..4) Rd(3..0)
  - Changes the content of upper and lower four bits of the Rd register and stores the new values in the Rd register
- TRAP #n ; trap to subroutine
  - Equals a software interrupt call, at its completion, the subroutine is called
- XCHB Rd ; exchange with B
  - Changes the content of the Rd register with the content of the B register
- LDSP ;Load stack pointer
- STSP ;Store stack pointer

Addition, subtraction, comparison

```
signed int i;
signed int j;
j = j + i;
or
j += i;
```

```
signed int i;
signed int j;
j = j - i;
or
j -= i;
```

- signed int i;
- signed int j;
- if (j == i) …;

```
signed long i;
signed long j;
j = j + i;
or
j += i;
```

```
signed long i;
signed long j;
j = j - i;
or
j -= i;
```

- signed long i;
- signed long j;
- if (j == i) …;

Examples

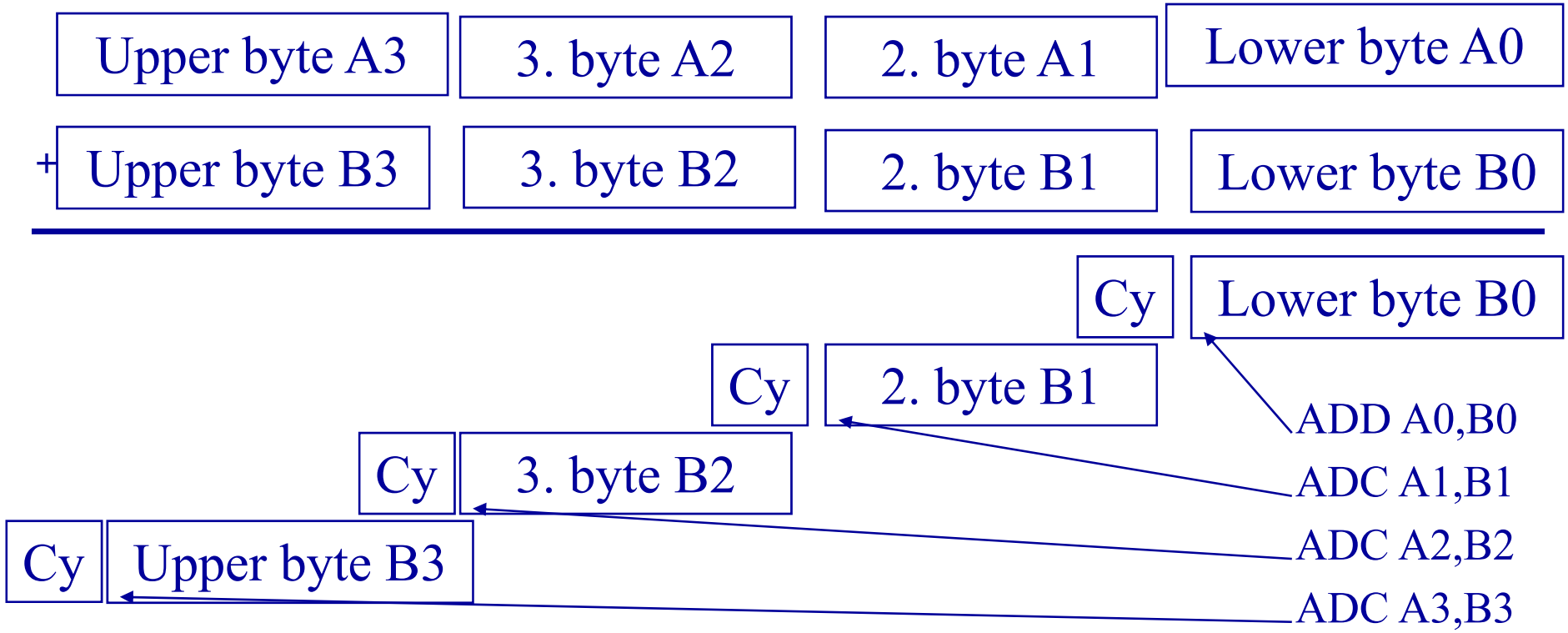
2 byte addition

	Upper byte A1	Lower byte A0	Operand one
+	Upper byte B1	Lower byte B0	Operand two

ADD A0,B0	Cy	Lower byte B0	
ADC A1,B1	Upper byte B1		
Cy	Upper byte B1	Lower byte B0	Result

Examples

4 byte addition



Examples

2 byte subtraction

	Upper byte A1	Lower byte A0	Operand one
-	Upper byte B1	Lower byte B0	Operand two

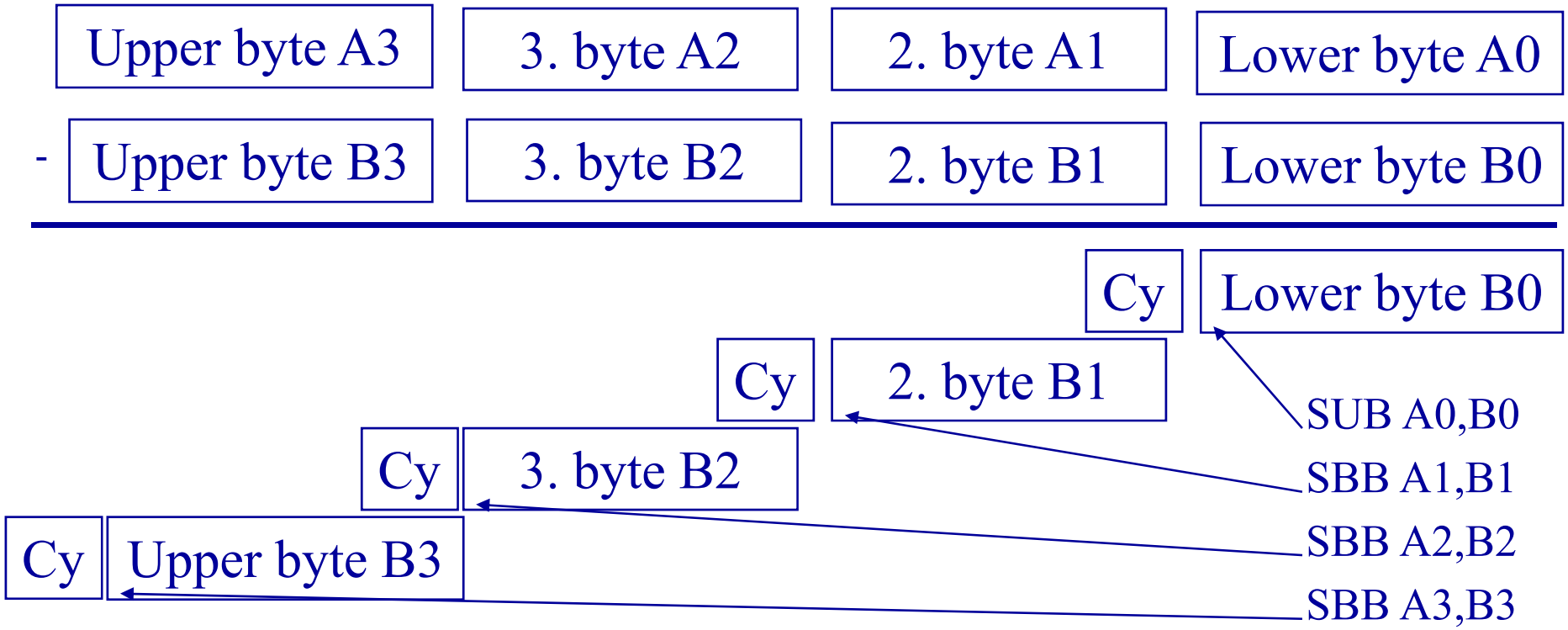
SUB A0,B0 → Cy, Lower byte B0

SBB A1,B1 → Upper byte B1

Cy	Upper byte B1	Lower byte B0	Result

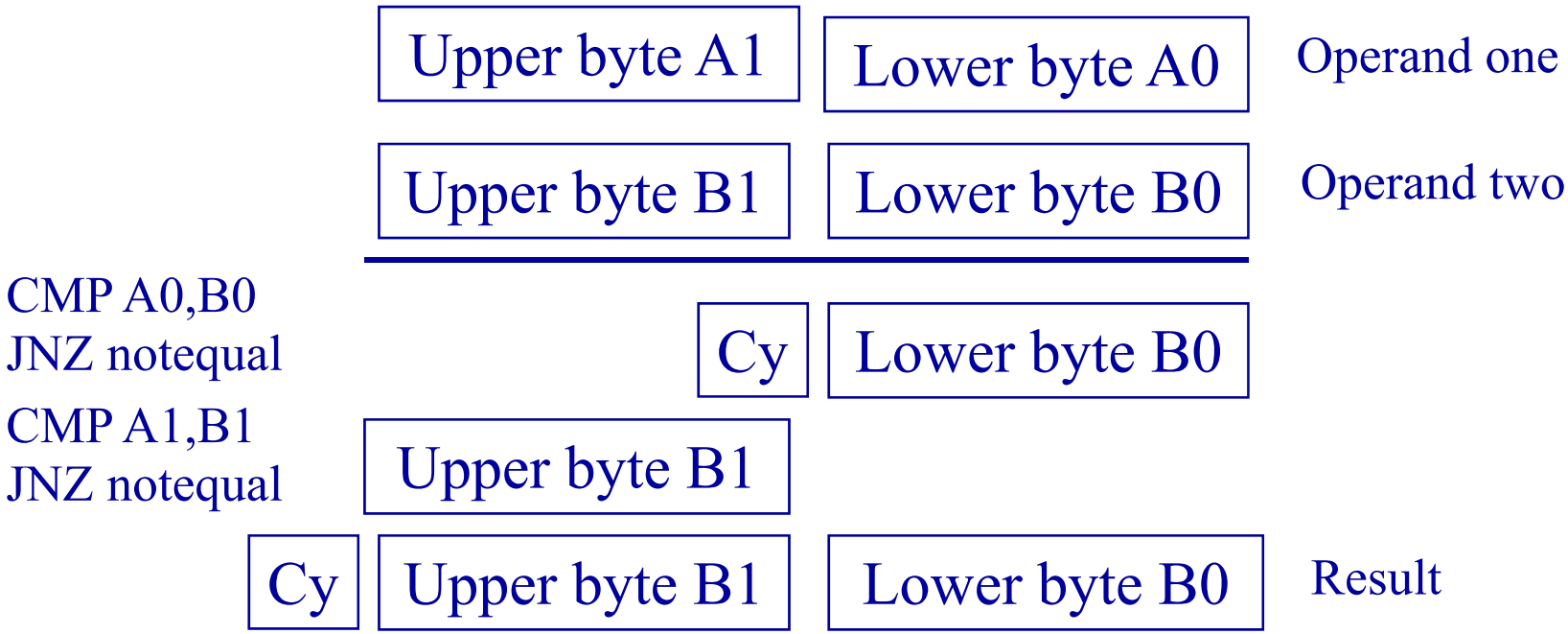
Examples

4 byte subtraction



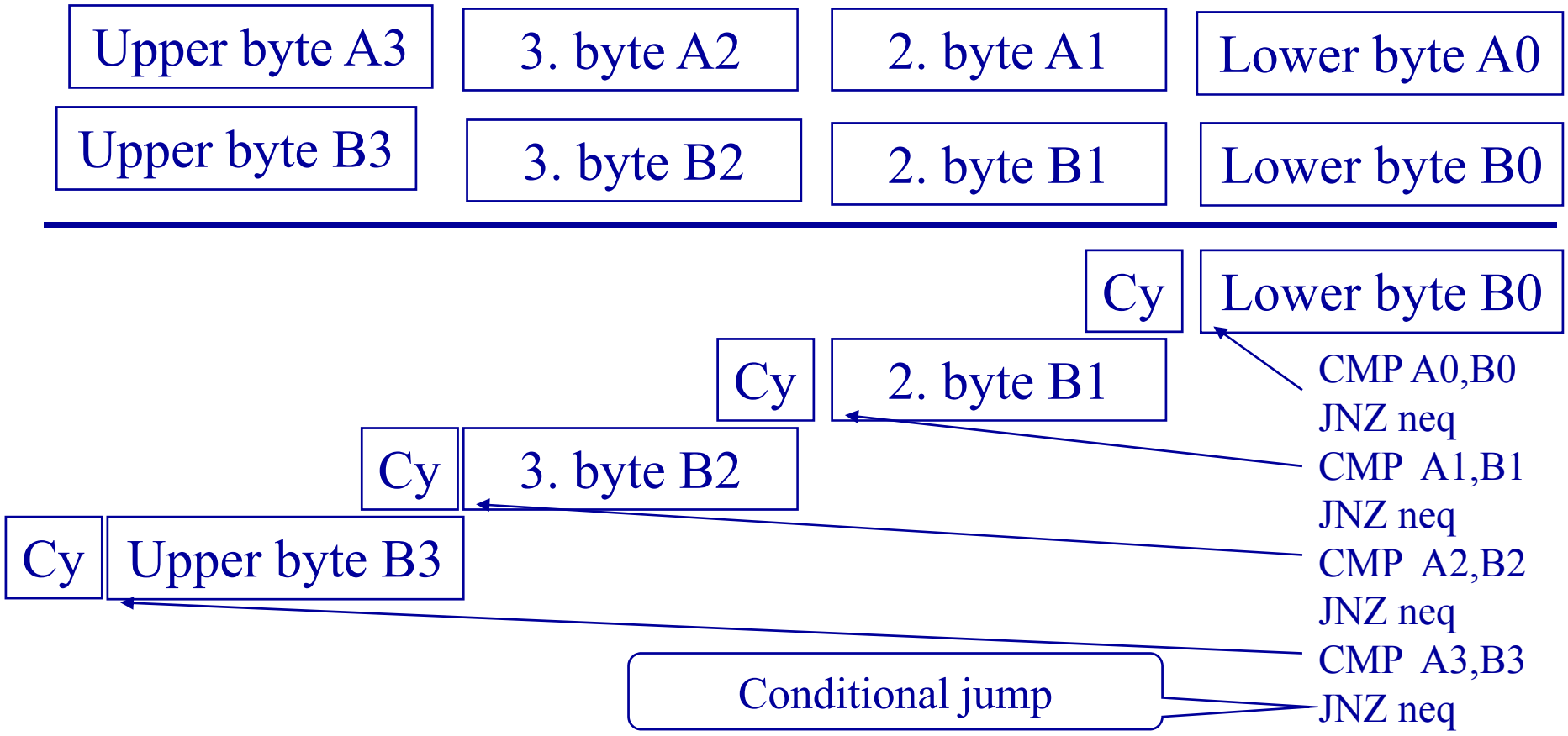
Examples

2 byte comparison



Examples

4 byte comparison



Conditions

Instruction	Mnemonic	C	N	Z	V	Operation
Jump if Carry	JC	1	x	x	x	
Jump if No Carry	JNC	0	x	x	x	
Jump if Equal	JEQ	x	x	1	x	
Jump if Not Equal	JNE	x	x	0	x	
Jump if Nonzero	JNZ	x	x	0	x	
Jump if Zero	JZ	x	x	1	x	
Jump if Lower	JLO	0	x	0	x	
Jump if Higher or Same	JHS	-	x	-	x	(C=1) OR (Z=1)
						Signed Operation
Jump if Greater	JG	x	-	-	-	Z OR (N XOR V) = 0
Jump if Greater of Equal	JGE	x	-	x	-	N XOR V = 0
Jump if Less	JL	x	-	x	-	N XOR V = 1
Jump if Less of Equal	JLE	x	-	-	-	Z OR (N XOR V) = 1
Jump if Negative	JN	x	1	x	x	
Jump if Positive	JP	x	0	0	x	
Jump if Positive or Zero	JPZ	x	0	x	x	
Jump if No Overflow	JNV	x	x	x	0	
Jump if Overflow	JV	x	x	x	1	

Program counter

- The program counter, also called the instruction pointer is the part of the instruction sequencer in some computers. It is a register in the processor of the computer, which shows where the computer is in the instruction sequence. Depending on the characteristics of the computer, it may contain the address of the executed instruction or that of the next instruction to be executed. The program counter automatically increases with each and every instruction cycle, therefore it reads the instructions from memory sequentially. E.g. 2 byte unsigned number
  - Maximum labelled area 64K
  - That is 65536 bytes

Example

PC = 1000H

- 1000 52 60 mov #60h,A

PC = 1002H

- 1002 fd ldsp

PC = 1003H

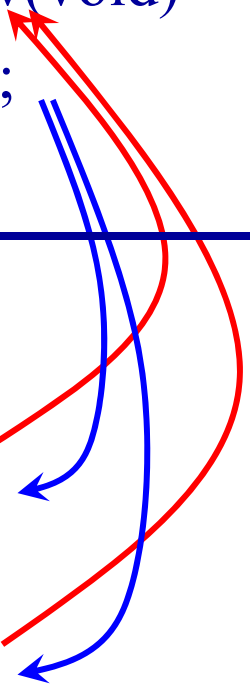
- 1003 ‘8c 10 69 br init ; this can also be jmp

Feltétel nélküli vezérlés átadás

PC = **1069H**

- 1006 … ← never will be executed
- PLACE OF A LABEL!!!!!!

The structure of function call

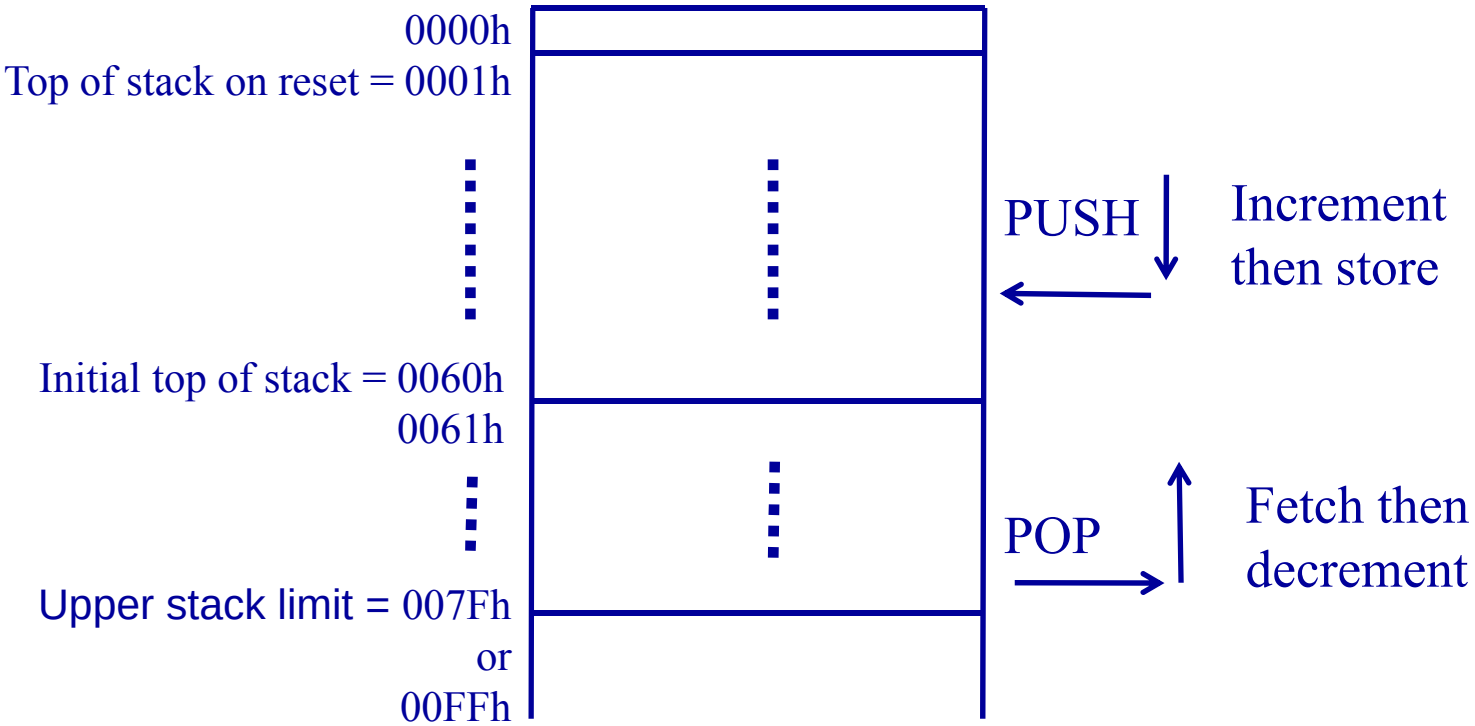


```
void fv(void)          fv: …          label:
{ … };                    rts             rts; return from subroutine

{
…
  fv();                call fv          call label; unconditional call
…                                                  ;it may be several times
  fv();                call fv          call label; unconditional call
}                                                   ; this is another place
```

Stack

FILO ("first in, last out") realization



Example

PC = 1000H SP = ????

- 1000 52 60 mov #60h,A

PC = 1002H SP = ????

- 1002 fd ldsp

PC = 1003H SP = 0060H

- 1003 ‘8e 10 69 call init

Stack 0061 ← 10H 0062 ← 06H - we save the value of return

PC = **1069H** SP = 0062H

- 1006 … ← control returns here

Addressing modes I.

- Absolute addressing mode: in the address part of the instruction, the real and exact address of the operand can be found. The address might be of the memory or one of the registers of the processor. In the case of register addressing, we need a smaller address part than in the case of memory addressing.
- Relative addressing mode: The addressing part of the instruction contains an address related to the base address of the operand. This can be the starting address of the datapage, the starting address of the memory segment, the address of the starting of the program, or the address of the instruction itself. This is the base address. The real address is given by the processor by adding together the base address and the relative address.

Addressing modes II.

- Indirect addressing: In the address part of the instruction, we do not find the address of the operand, but the address of the storage where the processor finds the address of the operand. In the case of some processors, this kind of addressing can be multilevel. For the indirect addressing, the processor can use the storage in the memory, or one of its registers.
- Direct addressing: it means a type of addressing when in the operand part of the instruction we do not find the address of the operand, but the operand itself. This dictates the biggest value of the operand (because of the restrictedness of the address part of the instruction), therefore it can be applied primarily with operations with small value constants.

Addressing modes III.

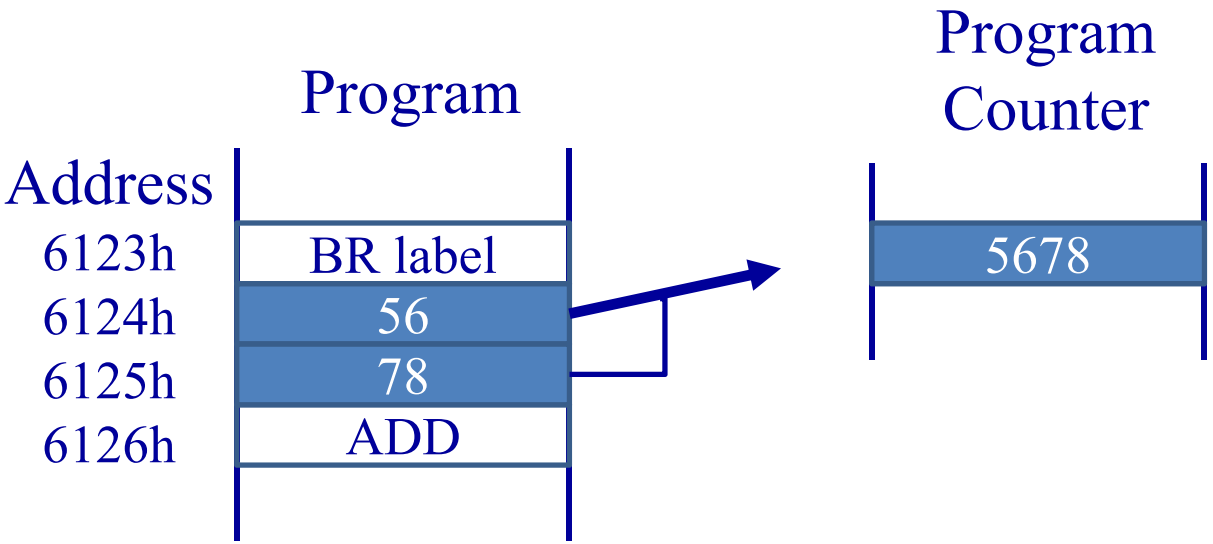
- Indexed addressing should be applied in the case of operations on data sequences. Here, in the address part of the instruction, we find the address of the first element of the data sequence, and in the address register we can find the diversion from this, giving the information which element of the sequence should be used from the data sequence to do the given operation. In the case of some processors, there are other solutions, where the index register, after finding the data, automatically increases or decreases, and this way, the operation on the data sequence becomes easier. This is called autoindexing. The indexed addressing method is similar to the relative addressing, the real address is made by adding the address part of the instruction and the index register.

Direct addressing

Absolute

BR 5678h

[label ⟶ (PC)]

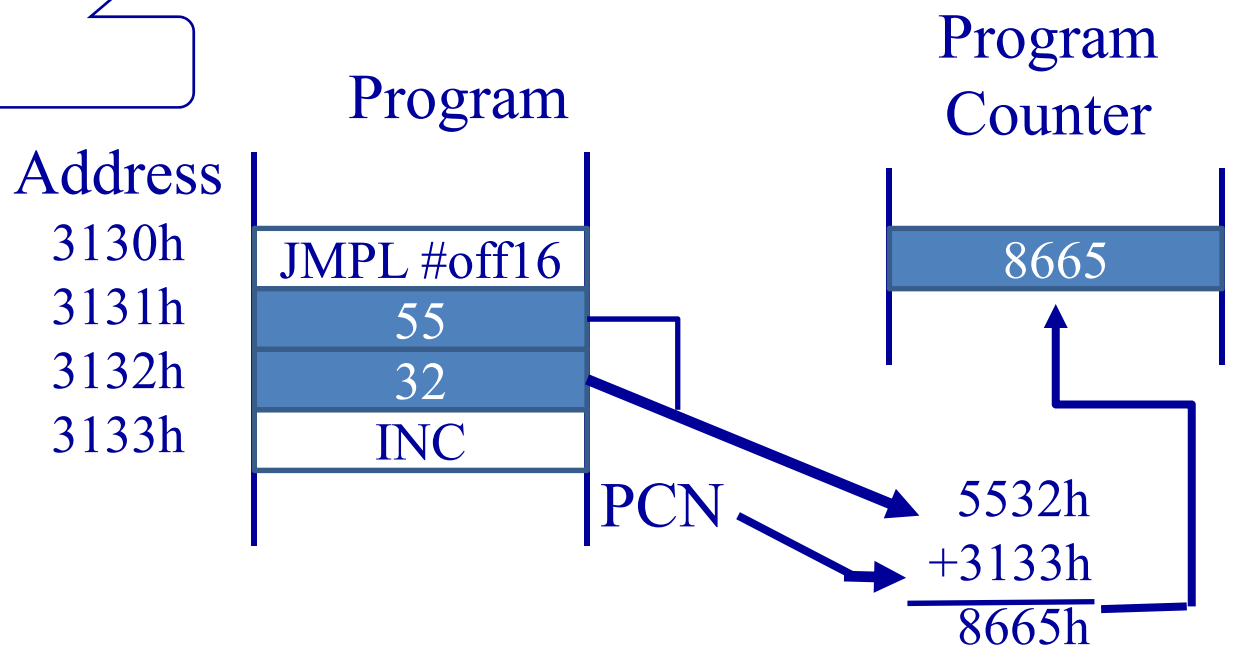


Direct addressing

JMPL #5532h

[PCN + off16 ⟶ (PC)]

Relative

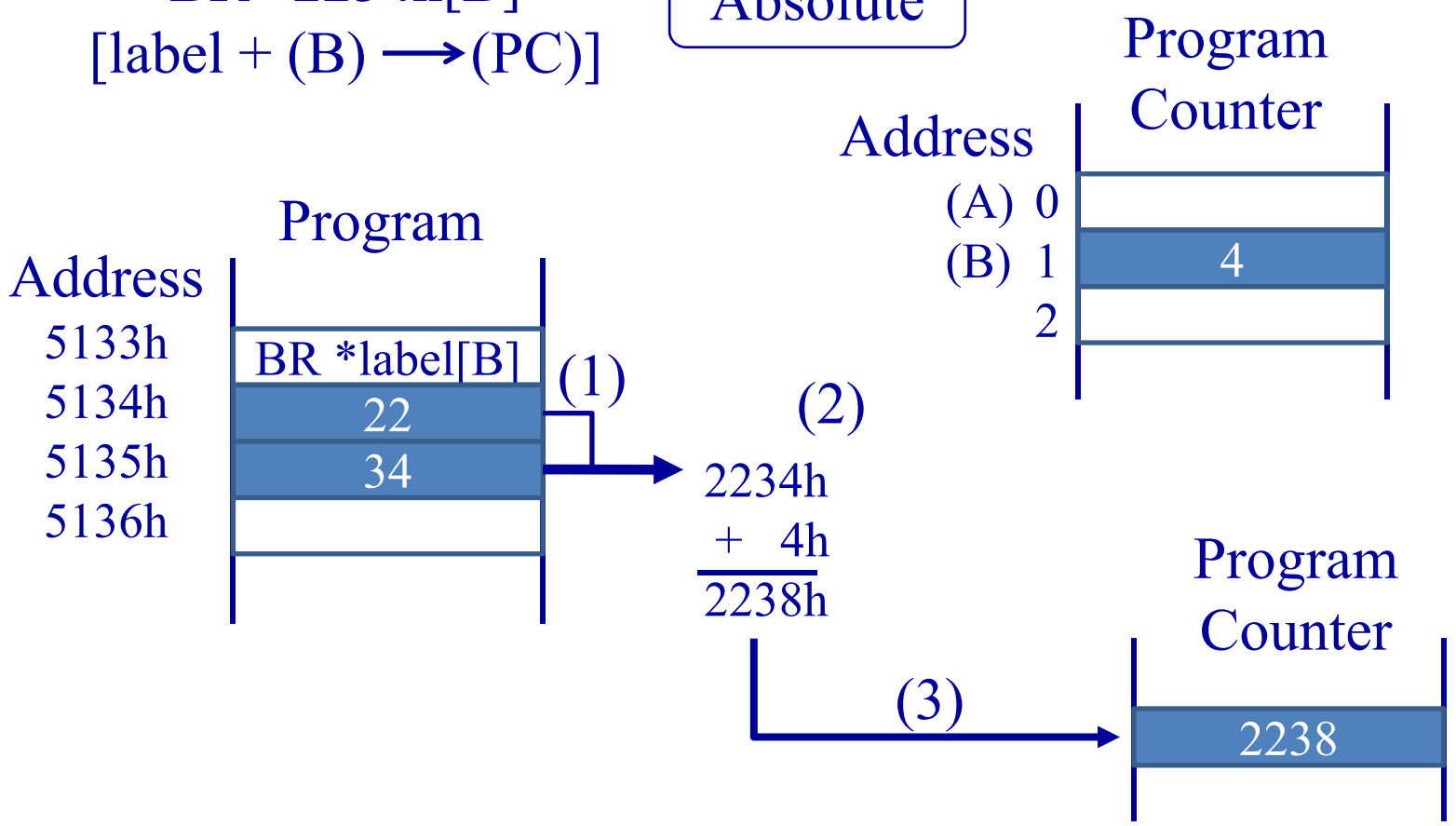


Indexed addressing mode

BR *2234h[B]

[label + (B) ⟶ (PC)]

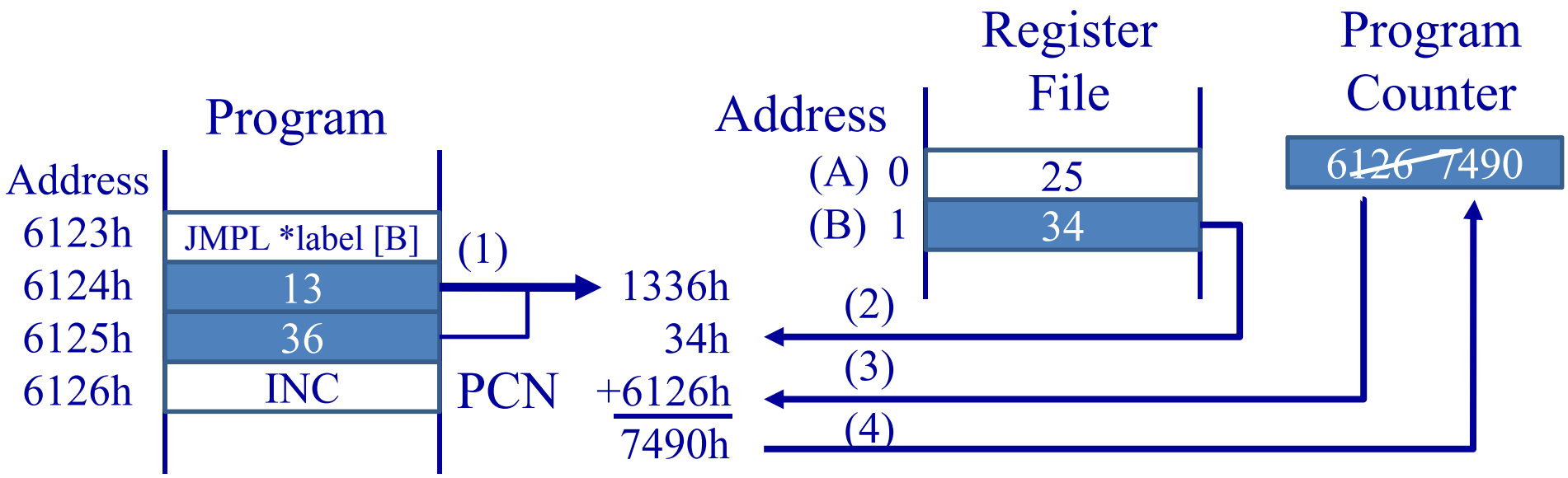
Absolute



Indexed addressing mode

JMPL *1336h[B] — Relative

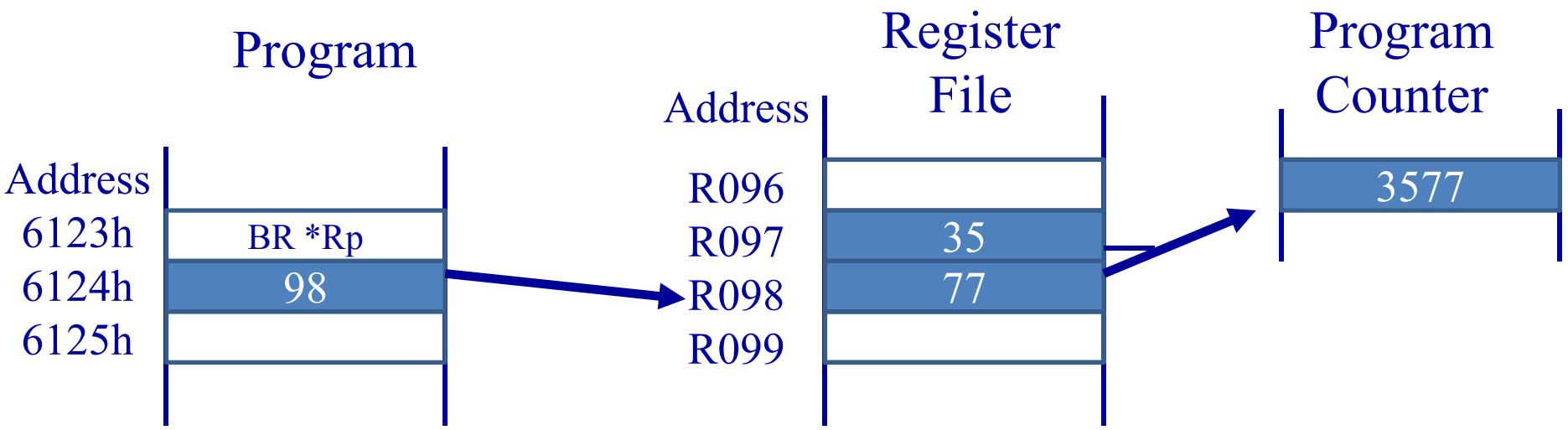
[label + (B) +PCN ⟶ (PC)]



Indirect addressing

BR *R098 — Absolute

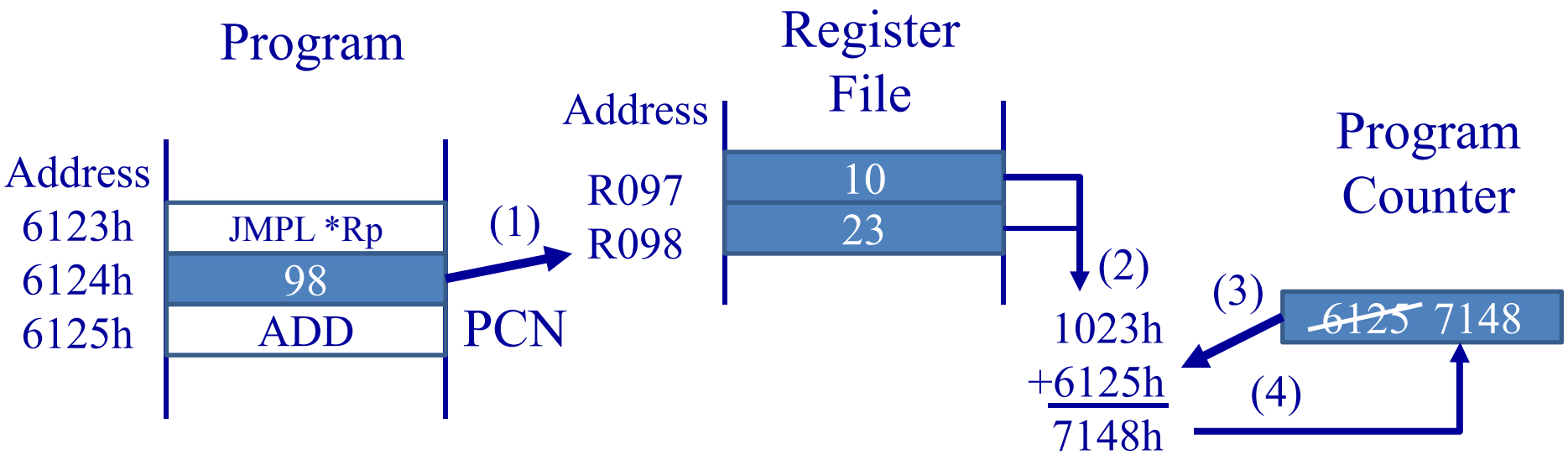
[(Rp – 1: Rp ⟶ (PC)]



Indirect addressing

JMPL *R098 — Relative

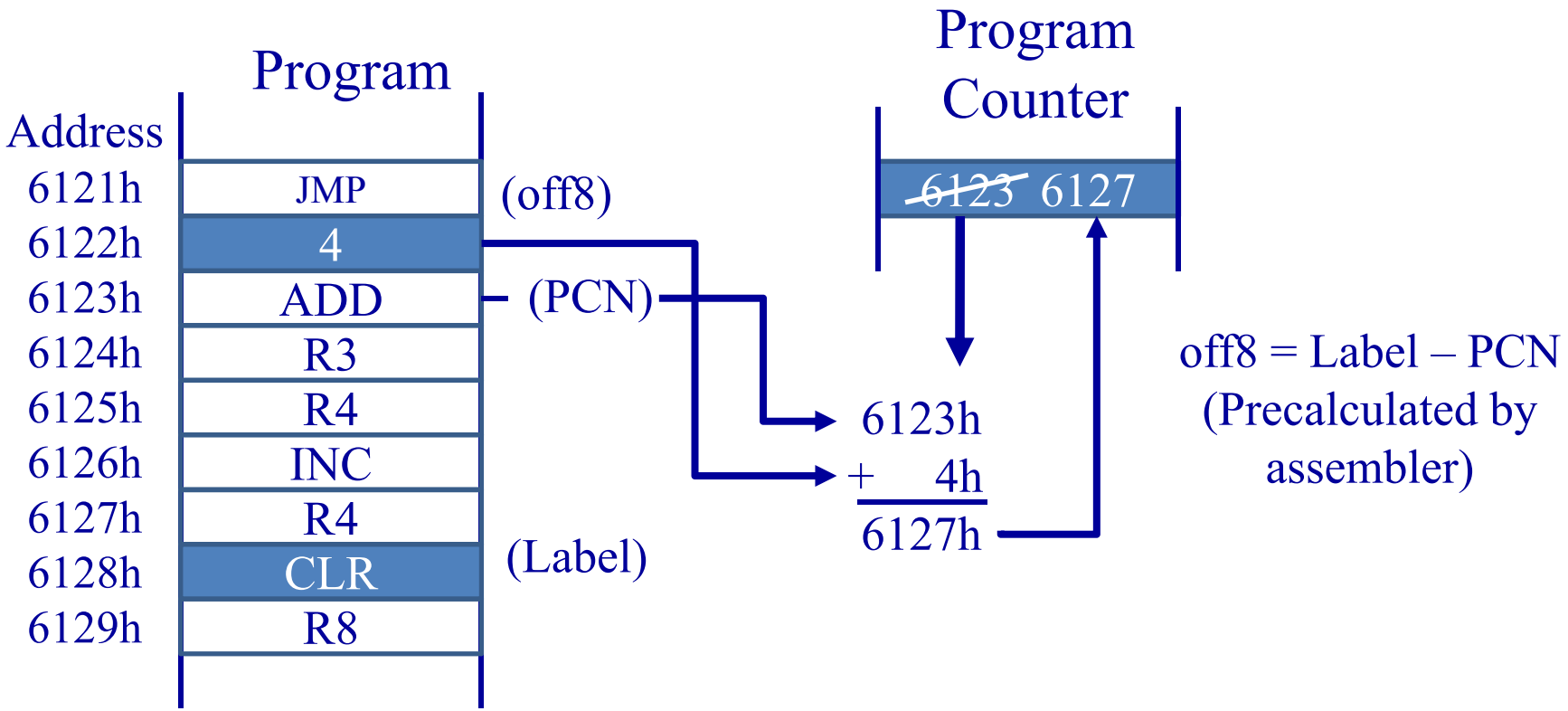
[(Rp – 1:Rp)+PCN ⟶ (PC)]



PC relative

JMPL #label

[PCN + off8 ⟶ (PC)]



Handling scalars, arrays and structures

```
unsigned int i;   Calculation of address:
                          - &i
  … = i;                  - *(&i)
```

```
unsigned int i[10];
  … = i[index];
```

Calculation of address:

- sizeof(unsigned int) x index
- sizeof(unsigned int) x index + &i
- *(sizeof(unsigned int) x index + &i)

```
Struct test {
  unsigned int i;
  signed char c;
  }t[10];

 … = t[index].c;
```

Calculation of address:

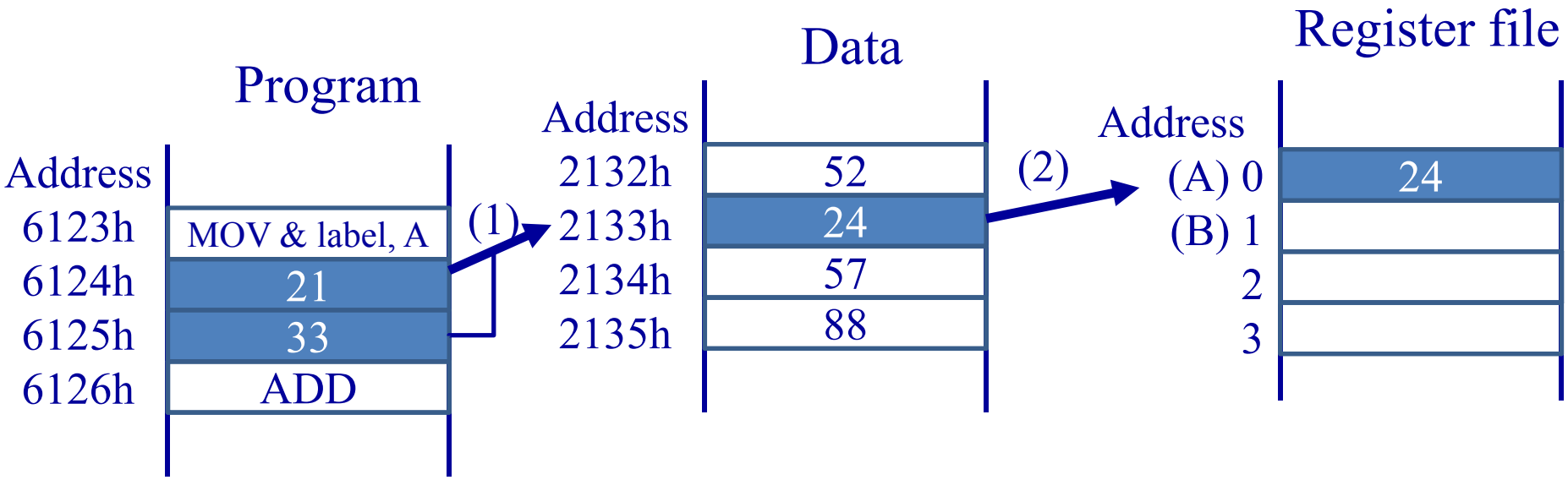
- sizeof(struct test) x index
- sizeof(struct test) x index + &t
- *(sizeof(struct test) x index + &t + offset(c))

Direct addressing

Variables handled directly

MOV &2133h, A

[(label) ⟶ (A)]

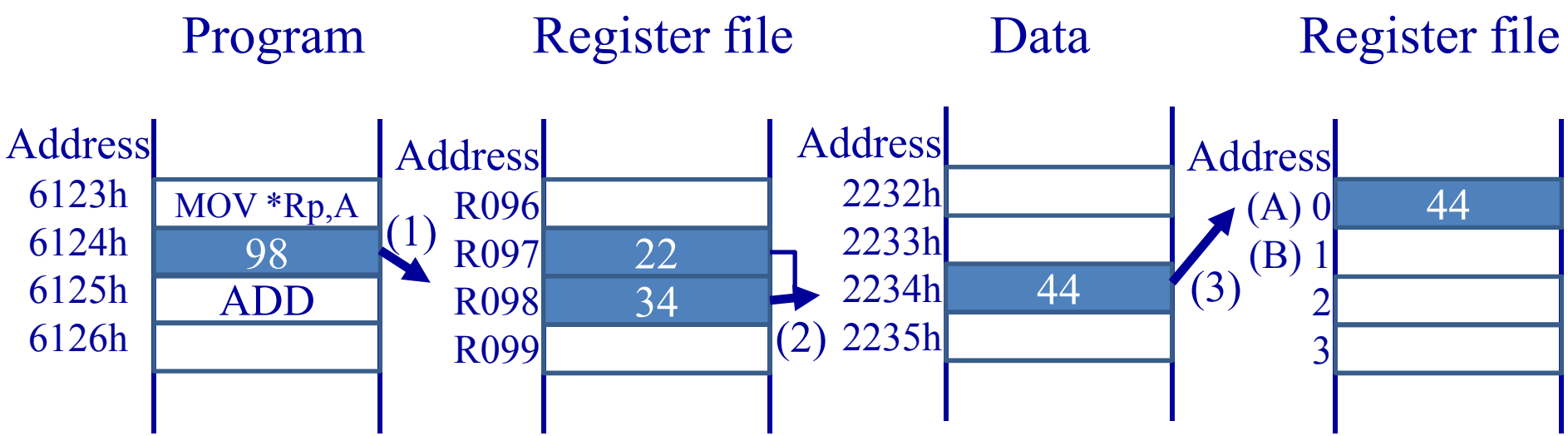


Indirect addressing

The address of the data is listed ina register

mov *R098,A

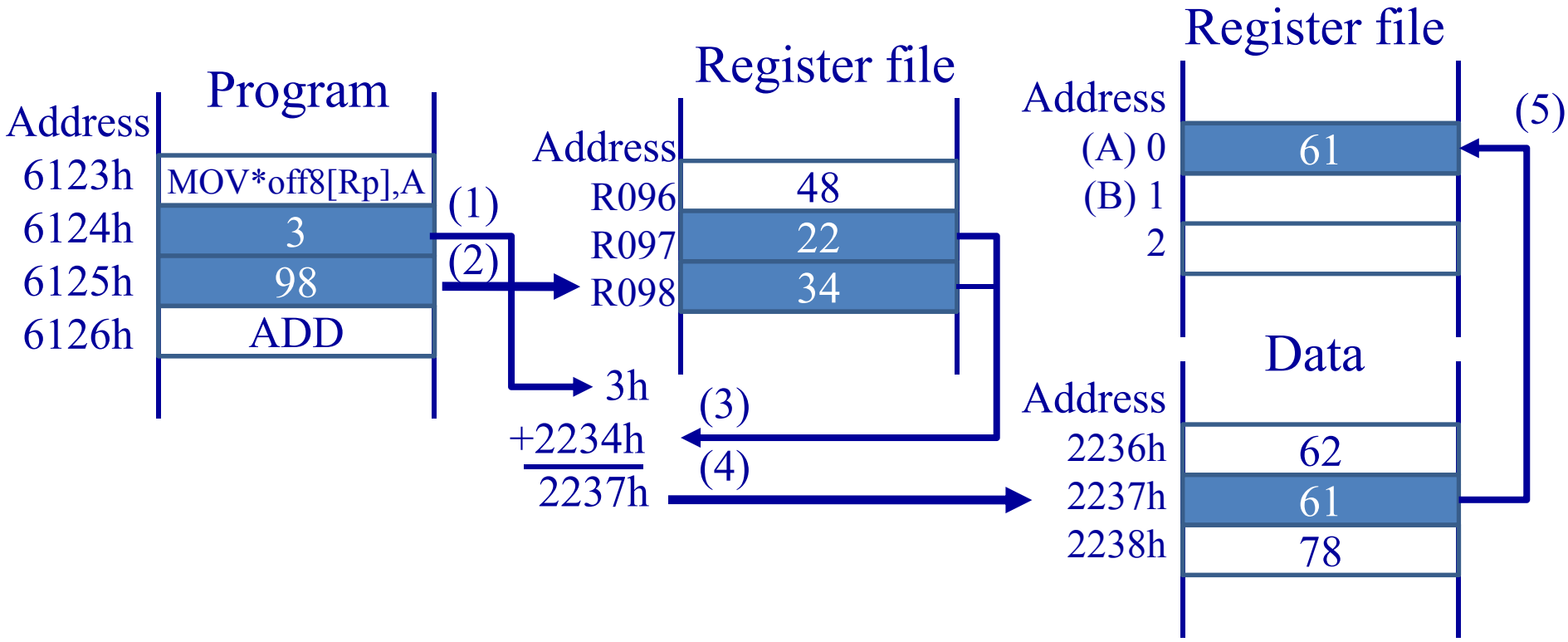
[((Rp-1:Rp)) ⟶(A)]



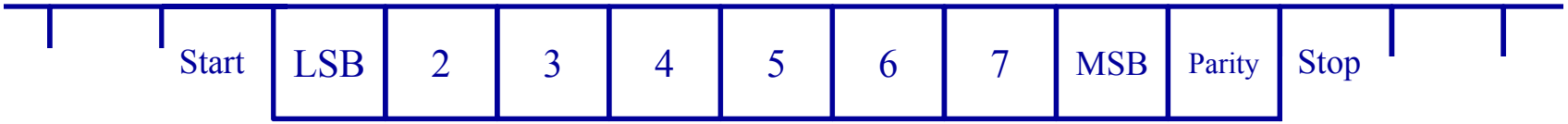
Offset indirect addressing

Mov *2h[R098],A

[(off8 + (Rp-1):Rp)) ⟶ (A)]



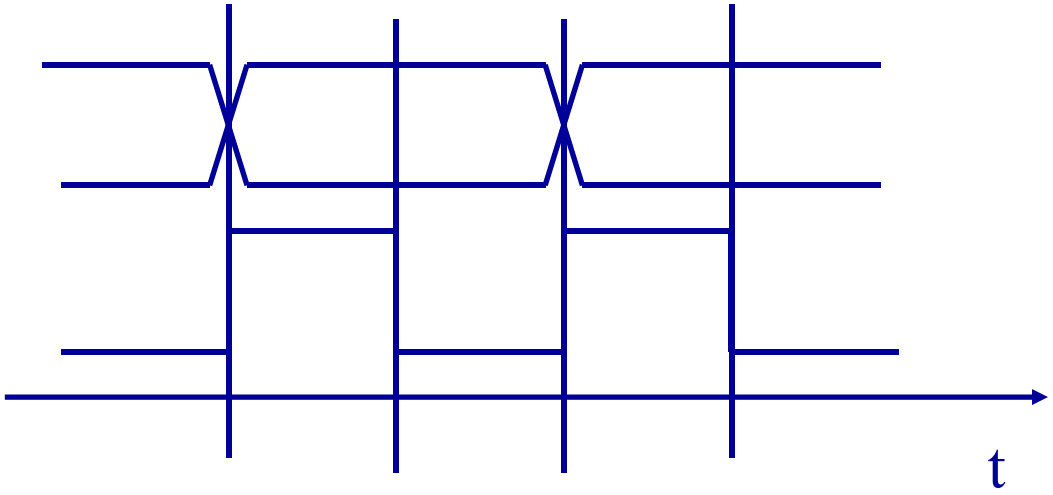
Asyncron serial communication



- Start bit is ‘1’
- Data in a low-high order 5;6;7;8;
- Parity N;M;S;O;E;
- Stop bit 1;1,5;2;
- Rated speed
  - 50, 100, 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400, 460800, …

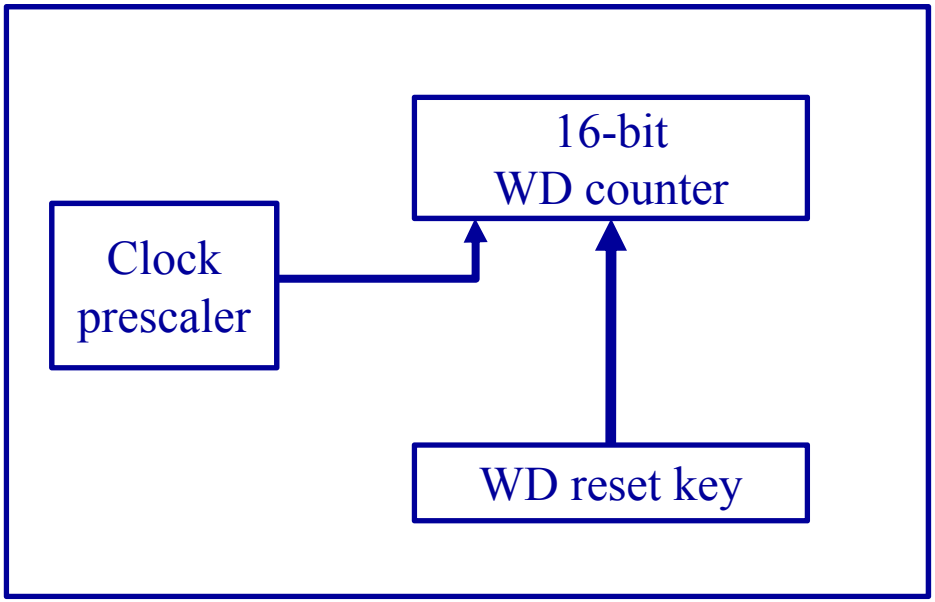
Syncron serial communication

- SPI
  - 1 data input
  - 1 data output
  - 1 clock sign
  - 1 select sign
- I2C
  - 1 data sign two-way
  - 1 clock sign



Watchdog

- Resettable monostable multivibrator
- For controlling the running of the program
- Safety solutions

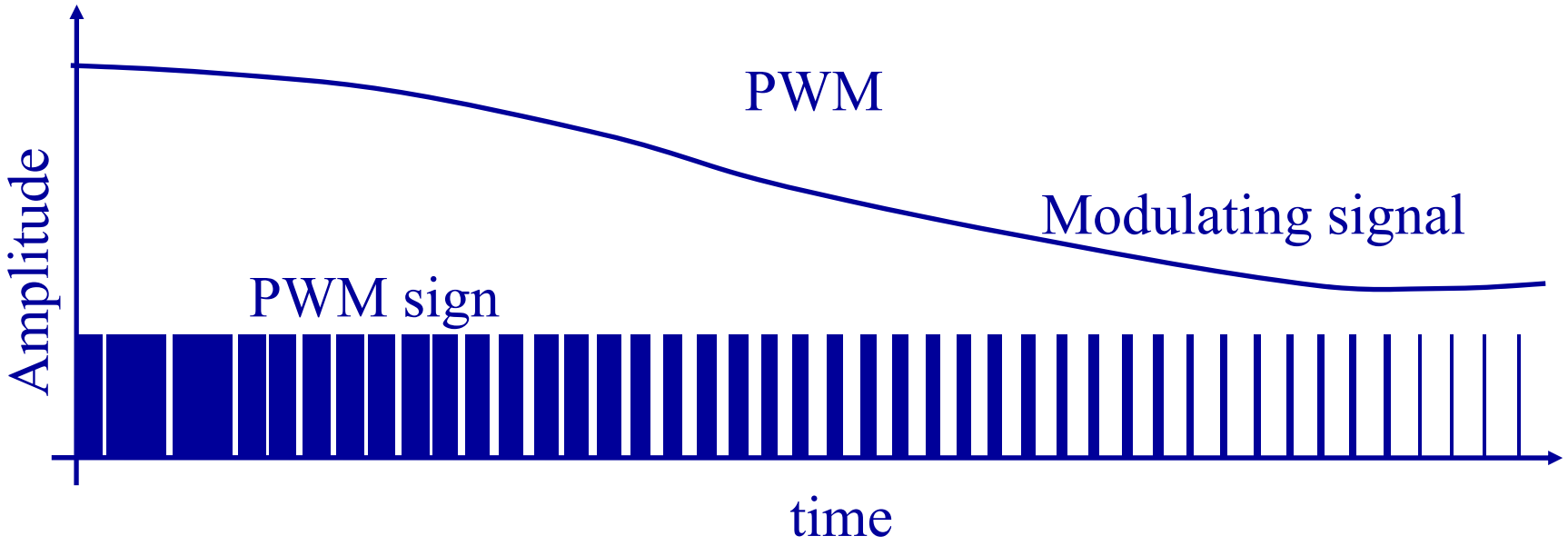


Interrupt

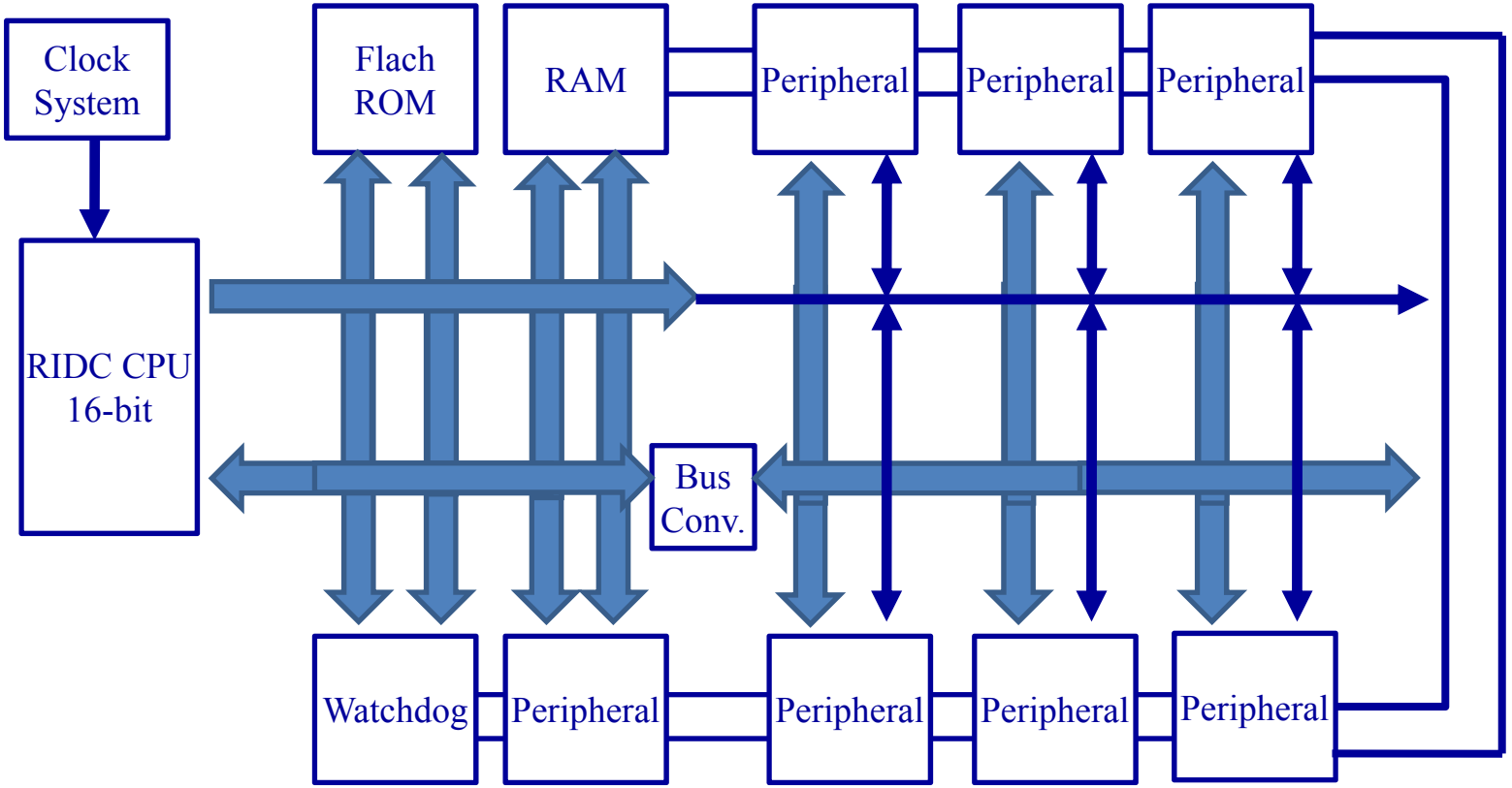
- Program interruption
  - From outer source
  - From inner source
  - Software
- Necessary operations
  - Save, IT prohibited, subprogram call (processor)
  - Save (user)
  - IT code
  - IT resource removal
  - Restoration (user)
  - Return, IT permission

Pulse Width Modulation

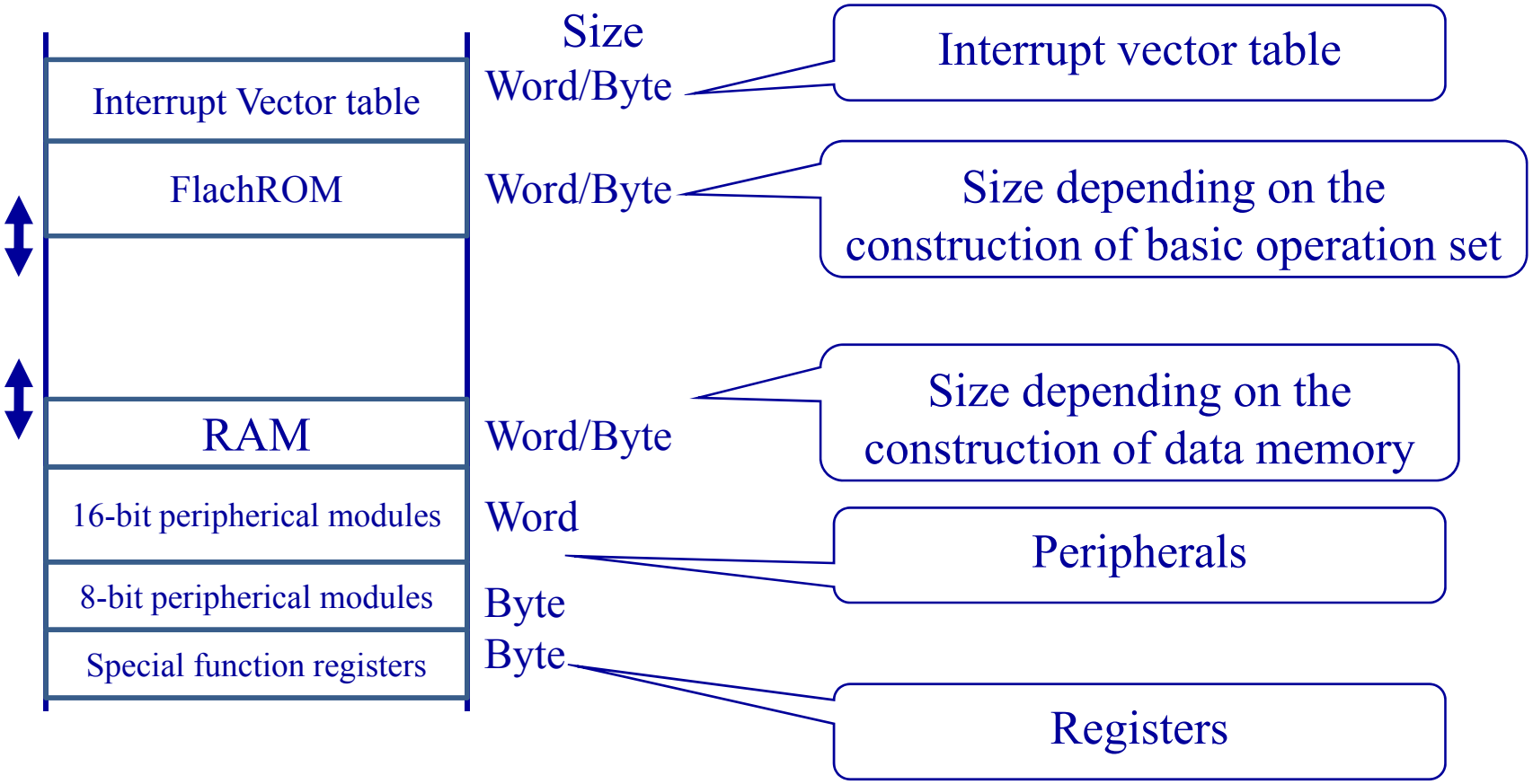
- simple DAC solution



MSP430 µC structure



Memory allocation



MSP430 initialization (hardware & software init)

Hardware init:	Software init:
Power on • Power on Reset (POR) • Brownout Reset (BOR) RST/NMI reset I/O config reset SR reset WatchDog reset PC loading form reset vector	Stack Pointer to the top of RAM Setting WatchDog timing Setting peripherals Interrupt permission Setting memory Program starts