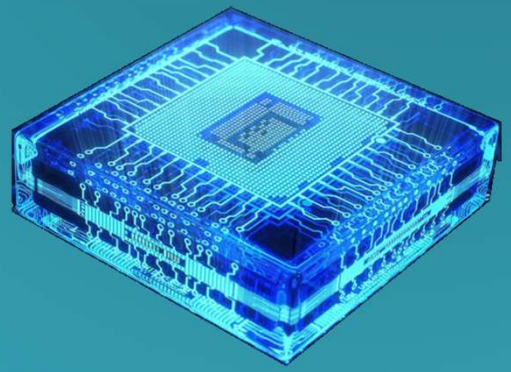
Microprocessors and Assembly language

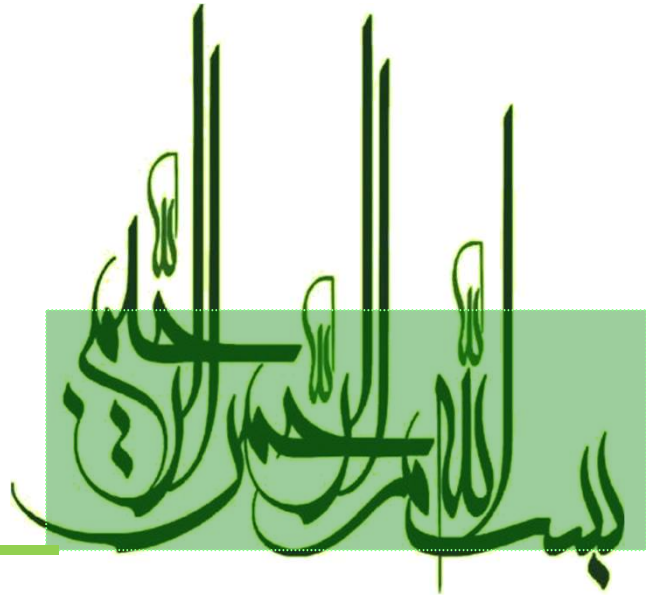
Isfahan University of Technology (IUT)



AVR Architecture and Assembly Language Programming

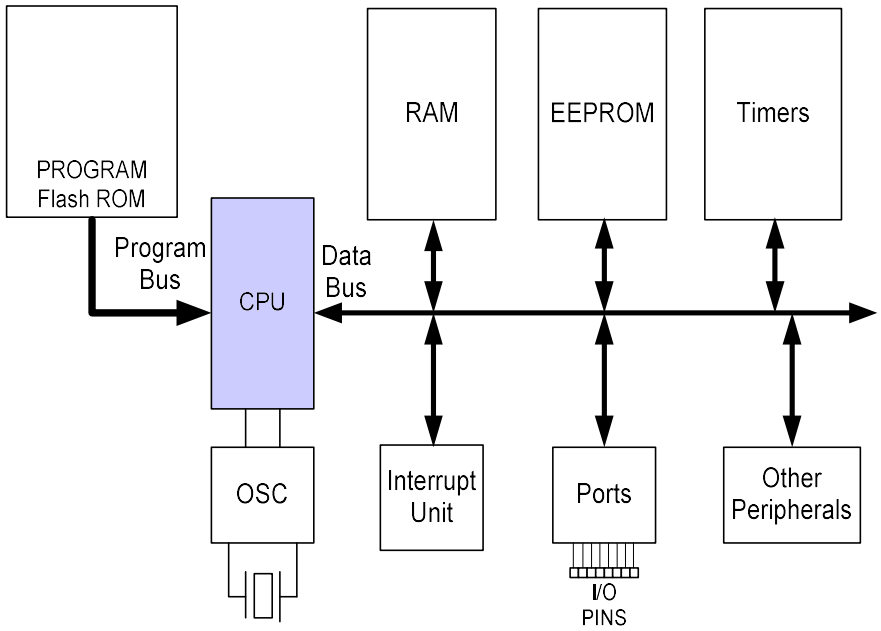
Dr. Hamidreza Hakim

hamid.hakim.u@gmail.com


بسم الله الرحمن الرحيم

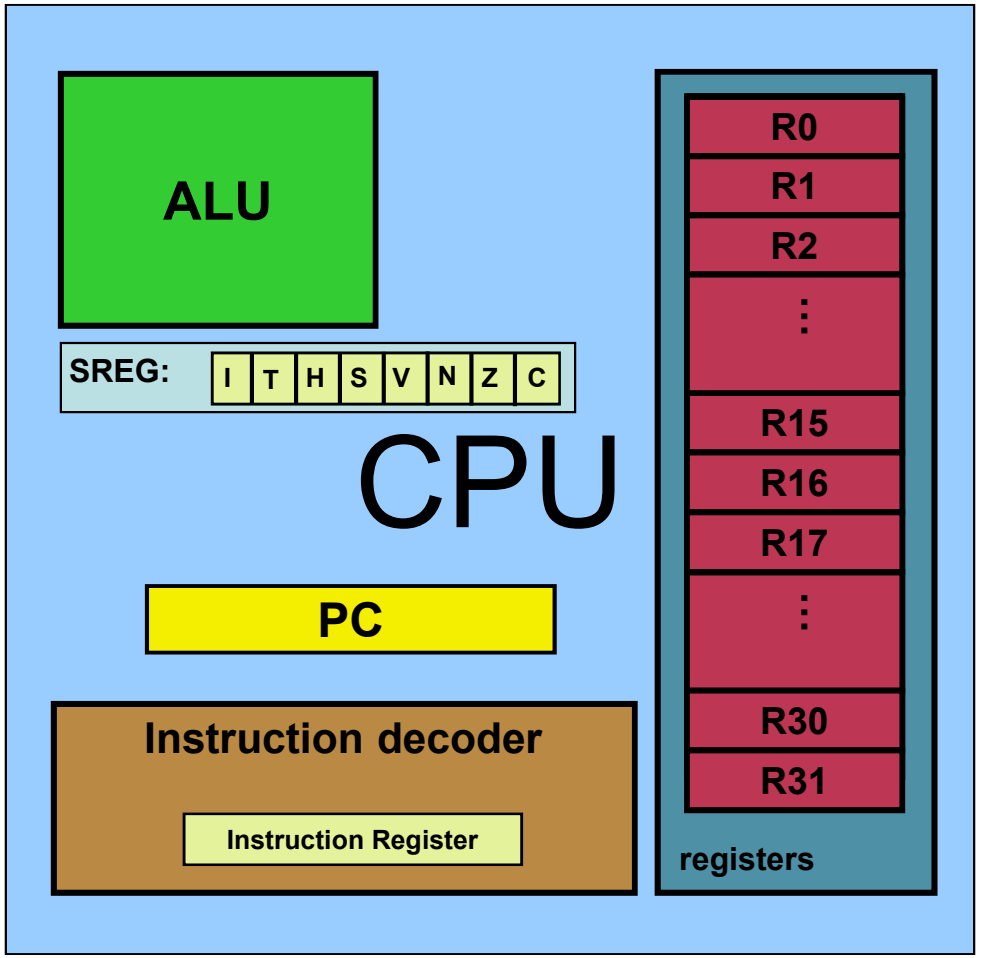
Topics

- AVR's CPU
  - Its architecture
  - Some simple programs
- Data Memory access
- Program memory
- RISC architecture



AVR's CPU

- AVR's CPU
  - ALU
  - 32 General Purpose registers (R0 to R31)
    - 8 bit
    - store information temporarily
  - PC register
  - Instruction decoder

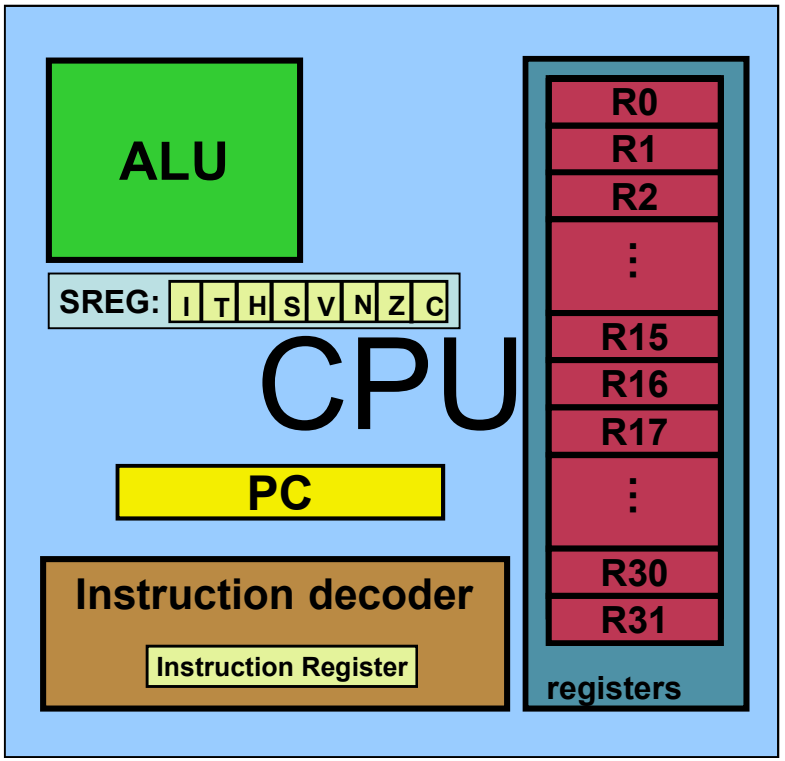


Some simple instructions
1. Loading values into the general purpose registers

LDI (Load Immediate)

- LDI Rd, k
  - Its equivalent in high level languages:Rd = k

    Note: d = [16-32]
- Example (hex-decimal-bin):
  - LDI R16,53
    - R16 = 53
  - LDI R19,$27
  - LDI R23,0x27
    - R23 = 0x27
  - LDI R23,0b11101100
  - Note: $ X 0b



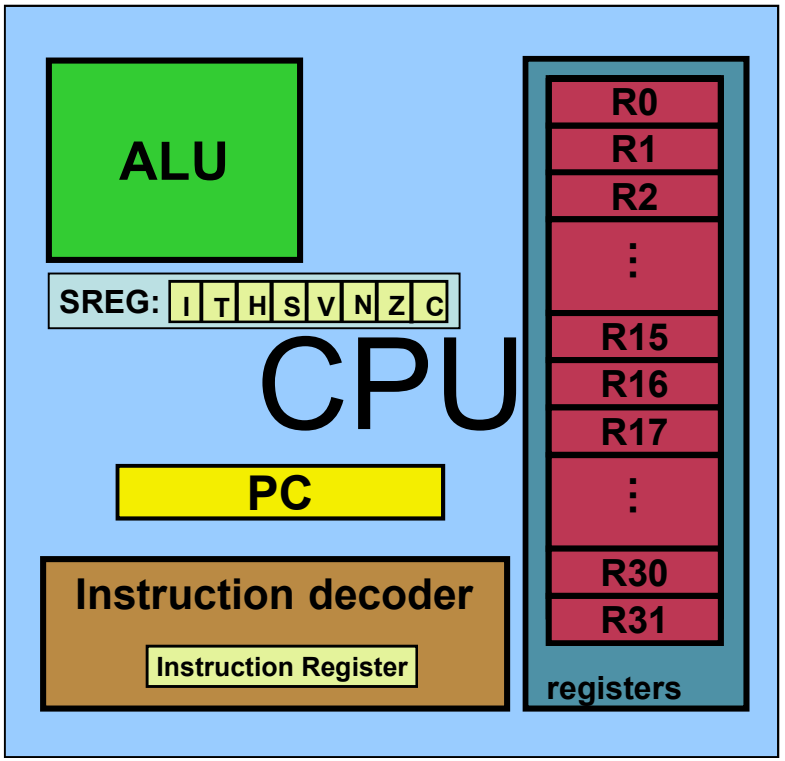
Some simple instructions

2. Arithmetic calculation

- There are some instructions for doing Arithmetic and logic operations; such as:

  ADD, SUB, MUL, AND, etc.

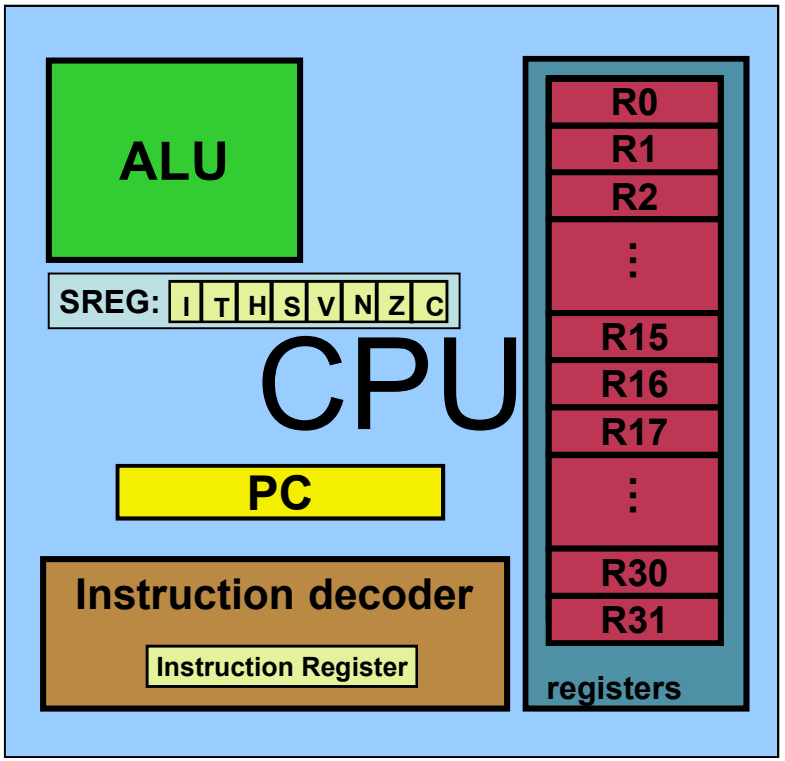
- ADD Rd, Rs
  - Rd = Rd + Rs
  - Example:
  - ADD R25, R9
    - R25 = R25 + R9
  - ADD R17,R30
    - R17 = R17 + R30



A simple program

- Write a program that calculates 19 + 95

```
LDI R16, 19     ;R16 = 19
LDI R20, 95     ;R20 = 95
ADD R16, R20    ;R16 = R16 + R20
```



A simple program

- Write a program that calculates 19 + 95 + 5

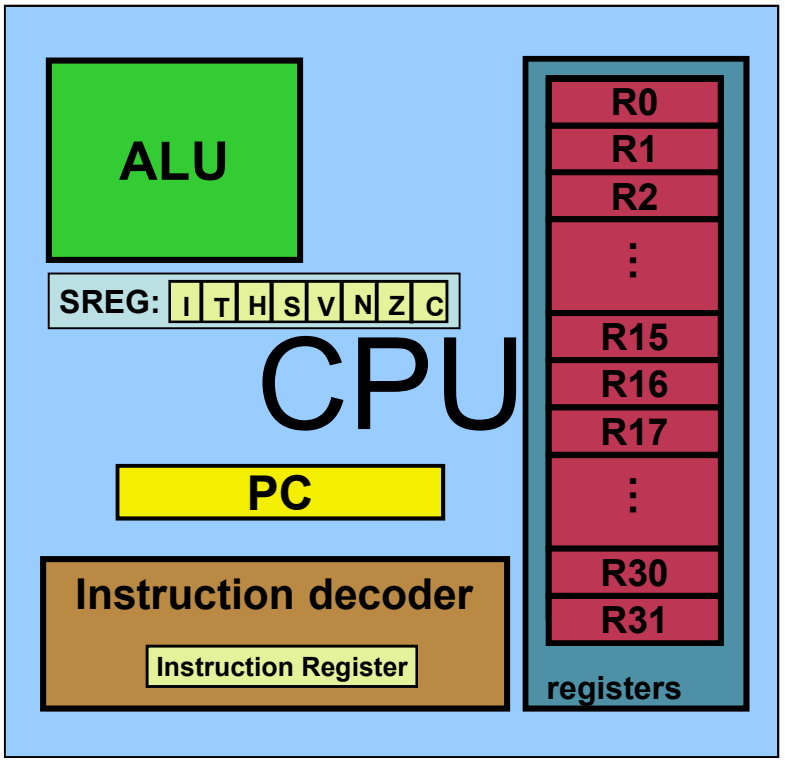
```
LDI     R16, 19         ;R16 = 19
LDI     R20, 95         ;R20 = 95
LDI     R21, 5          ;R21 = 5
ADD     R16, R20        ;R16 = R16 + R20
ADD     R16, R21        ;R16 = R16 + R21
```

```
LDI     R16, 19         ;R16 = 19
LDI     R20, 95         ;R20 = 95
ADD     R16, R20        ;R16 = R16 + R20
LDI     R20, 5          ;R20 = 5
ADD     R16, R20        ;R16 = R16 + R20
```

Some simple instructions

2. Arithmetic calculation

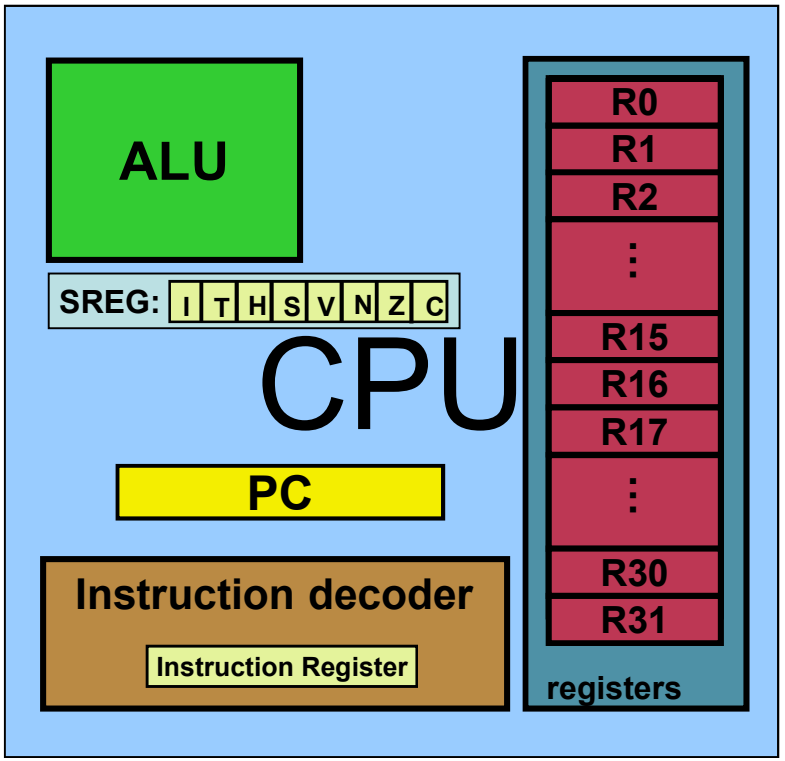
- SUB Rd,Rs
  - Rd = Rd - Rs
- Example:
  - SUB R25, R9
    - R25 = R25 - R9
  - SUB R17,R30
    - R17 = R17 - R30



Some simple instructions

2. Arithmetic calculation

- INC Rd
  - Rd = Rd + 1
- Example:
  - INC R25
    - R25 = R25 + 1
- DEC Rd
  - Rd = Rd - 1
- Example:
  - DEC R23
    - R23 = R23 - 1



AVR General Purpose Registers and ALU

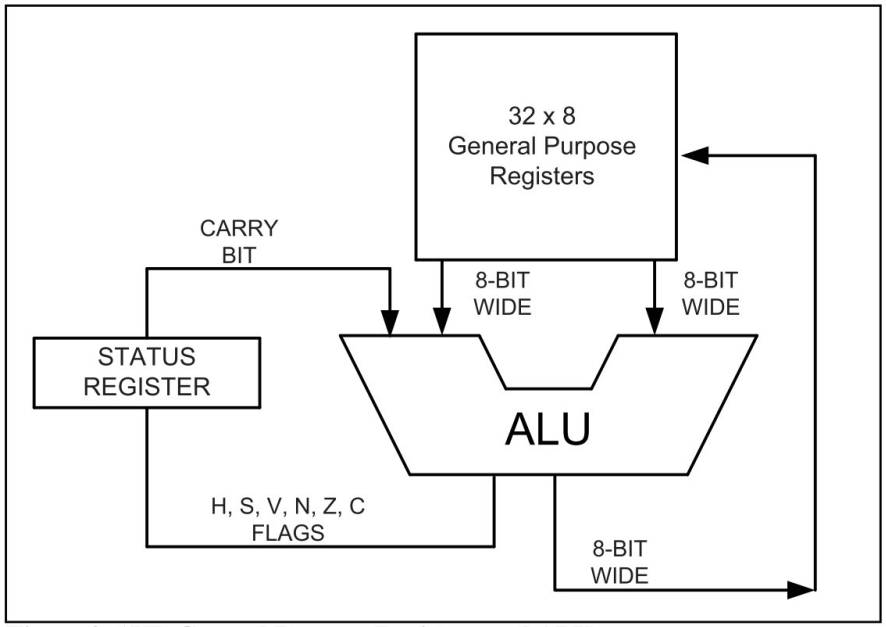
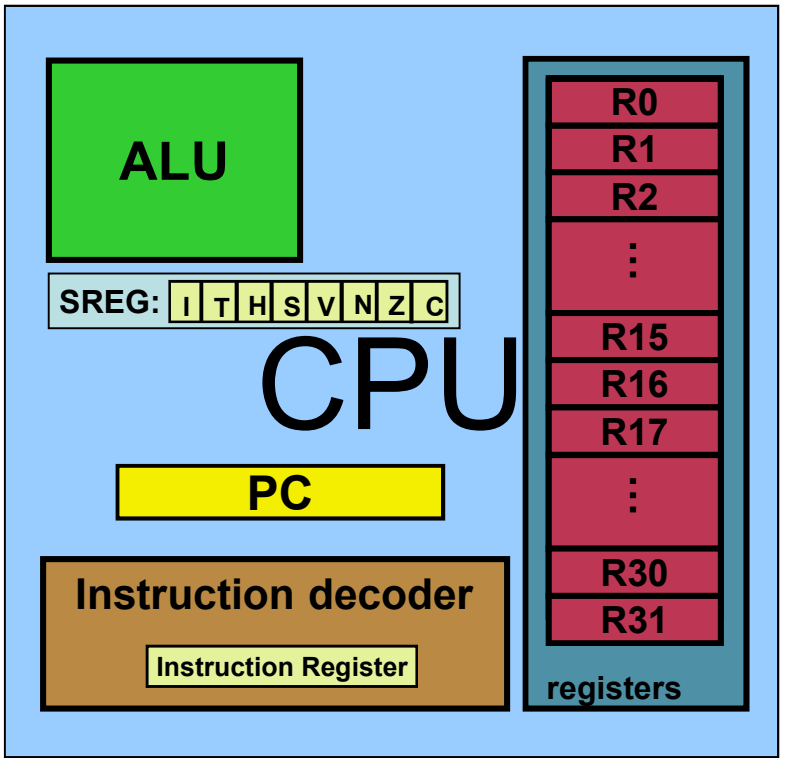


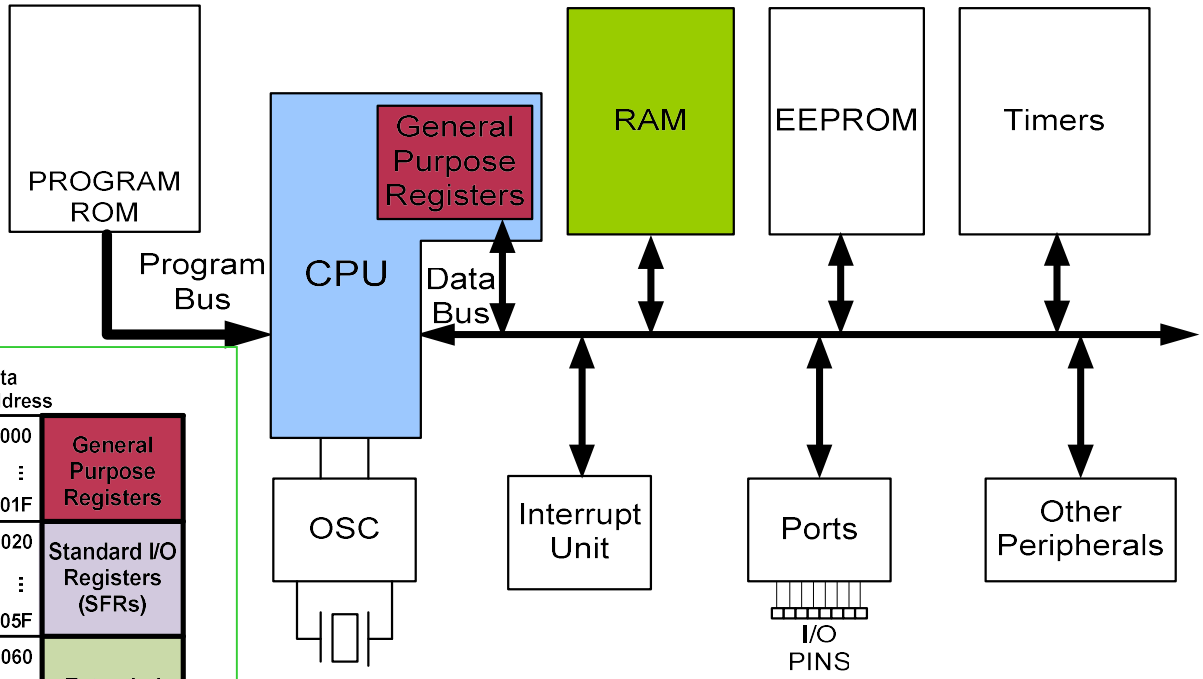
Figure 2. AVR General Purpose Registers and ALU

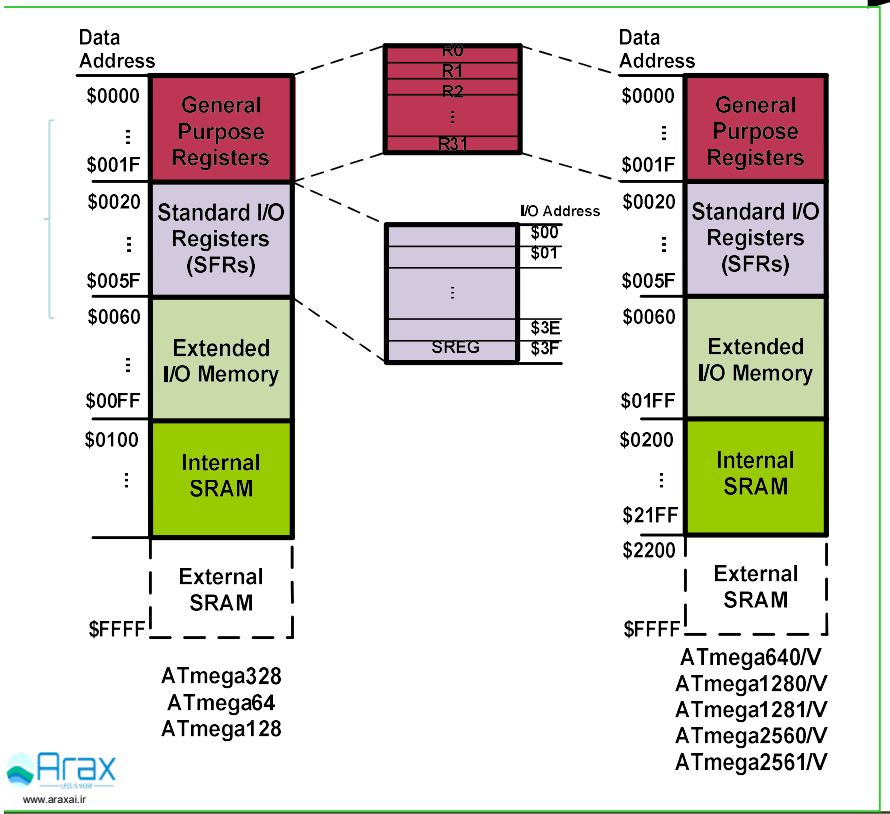


Why two input?

DATA MEMORY ACCESS

Data Address Space





GPRs(32 byte)

General Purpose(GP)

Standard(I/O)Rs (64 byte)

Special Function(SF)

Fix size in all AVRs

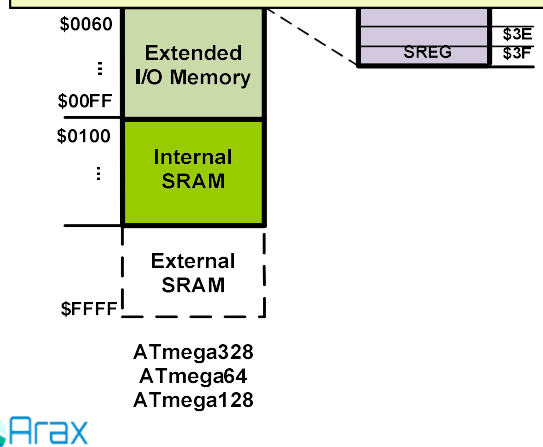
` GPRs and SFRs

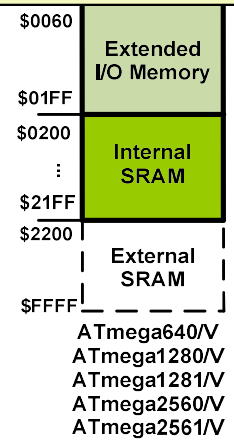
Data Address Space

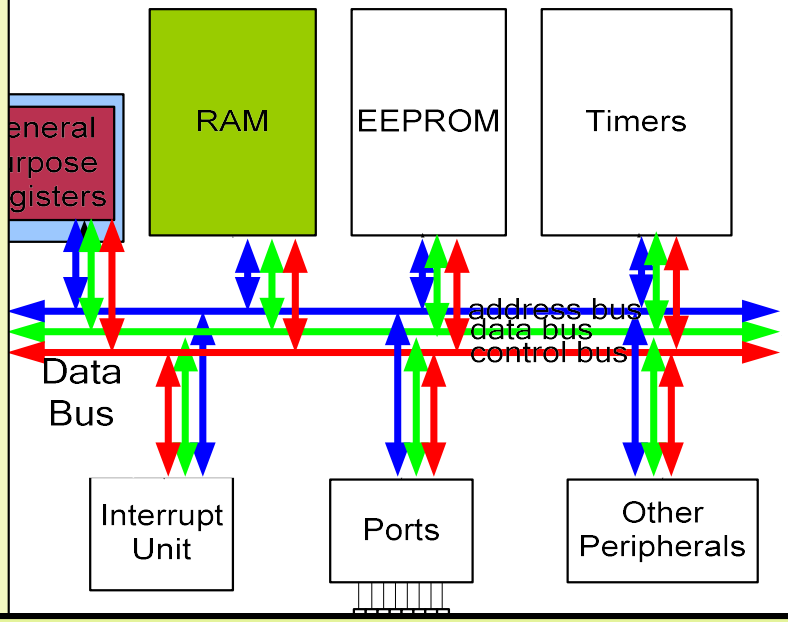
Address		Name
Mem.	I/O	
$20	$00	-
$21	$01	-
$22	$02	-
$23	$03	PINB
$24	$04	DDRB
$25	$05	PORTB
$26	$06	PINC
$27	$07	DDRC
$28	$08	PORTC
$29	$09	PIND
$2A	$0A	DDRD
$2B	$0B	PORTD
$2C	$0C	-
$2D	$0D	-
$2E	$0E	-
$2F	$0F	-
$30	$10	-
$31	$11	-
$32	$12	-
$33	$13	-
$34	$14	-
$35	$15	TIFR0

Address		Name
Mem.	I/O	
$36	$16	TIFR1
$37	$17	TIFR2
$38	$18	-
$39	$19	-
$3A	$1A	-
$3B	$1B	PCIFR
$3C	$1C	EIFR
$3D	$1D	EIMSK
$3E	$1E	GPIOR0
$3F	$1F	EECR
$40	$20	EEDR
$41	$21	EEARL
$42	$22	EEARH
$43	$23	GTCCR
$44	$24	TCCR0A
$45	$25	TCCR0B
$46	$26	TCNT0
$47	$27	OCR0A
$48	$28	OCR0B
$49	$29	-
$4A	$2A	GPIOR1
$4A	$2A	GPIOR2

Address		Name
Mem.	I/O	
$4C	$2C	SPCR0
$4D	$2D	SPSR0
$4E	$2E	SPDR0
$4F	$2F	-
$50	$30	ACSR
$51	$31	DWDR
$52	$32	-
$53	$33	SMCR
$54	$34	MCUSR
$55	$35	MCUCR
$56	$36	-
$57	$37	SPMCSR
$58	$38	-
$59	$39	-
$5A	$3A	-
$5B	$3B	-
$5C	$3C	-
$5D	$3D	SPL
$5E	$3E	SPH
$5F	$3F	SREG







Example: Store 0x53 into the SPH register. The address of SPH is 0x5E

Solution:

```
LDI    R20, 0x53          ;R20 = 0x53
STS    0x5E, R20          ;SPH = R20
```

Data Address Space and the I/O address

Address		Name
Mem.	I/O	
$20	$00	-
$21	$01	-
$22	$02	-
$23	$03	PINB
$24	$04	DDRB
$25	$05	PORTB
$26	$06	PINC
$27	$07	DDRC
$28	$08	PORTC
$29	$09	PIND
$2A	$0A	DDRD
$2B	$0B	PORTD
$2C	$0C	-
$2D	$0D	-
$2E	$0E	-
$2F	$0F	-
$30	$10	-
$31	$11	-
$32	$12	-
$33	$13	-
$34	$14	-
$35	$15	TIFR0

Address		Name
Mem.	I/O	
$36	$16	TIFR1
$37	$17	TIFR2
$38	$18	-
$39	$19	-
$3A	$1A	-
$3B	$1B	PCIFR
$3C	$1C	EIFR
$3D	$1D	EIMSK
$3E	$1E	GPIOR0
$3F	$1F	EECR
$40	$20	EEDR
$41	$21	EEARL
$42	$22	EEARH
$43	$23	GTCCR
$44	$24	TCCR0A
$45	$25	TCCR0B
$46	$26	TCNT0
$47	$27	OCR0A
$48	$28	OCR0B
$49	$29	-
$4A	$2A	GPIOR1
$4A	$2A	GPIOR2

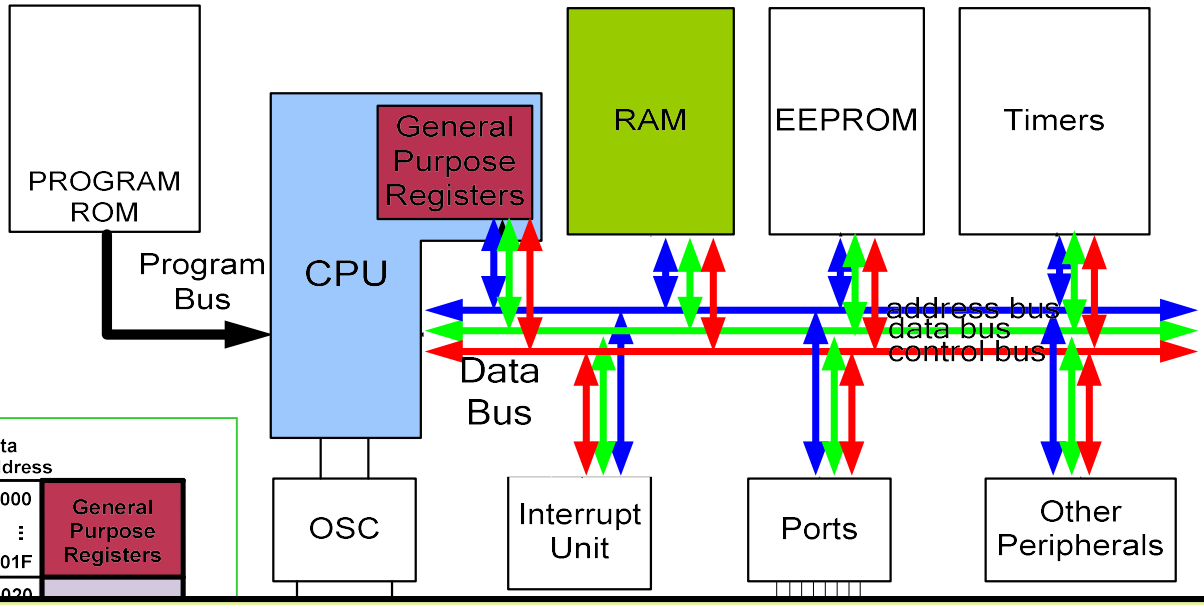
Address		Name
Mem.	I/O	
$4C	$2C	SPCR0
$4D	$2D	SPSR0
$4E	$2E	SPDR0
$4F	$2F	-
$50	$30	ACSR
$51	$31	DWDR
$52	$32	-
$53	$33	SMCR
$54	$34	MCUSR
$55	$35	MCUCR
$56	$36	-
$57	$37	SPMCSR
$58	$38	-
$59	$39	-
$5A	$3A	-
$5B	$3B	-
$5C	$3C	-
$5D	$3D	SPL
$5E	$3E	SPH
$5F	$3F	SREG

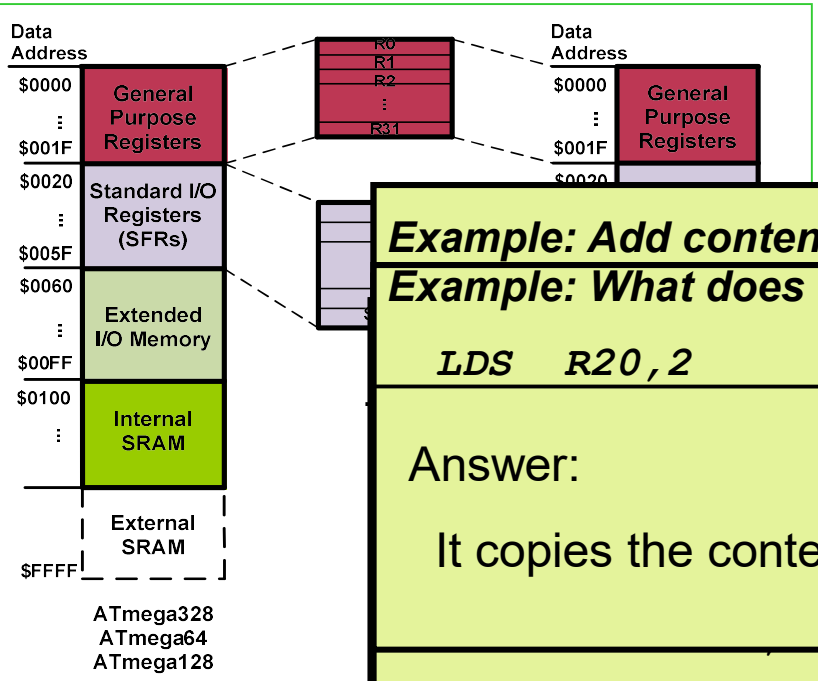
Each location in the data memory
has a unique address
called the data memory address.

Each I/O register
has a relative address
(in comparison to the beginning of the I/O memory;)
this address is called the I/O address

Data Address Space

you cannot copy (store) an immediate value directly into the SRAM location





Example: Add contents of location 0x90 to contents of location 0x95

Example: What does the following instruction do?

```
LDS   R20,2
```

Solution:

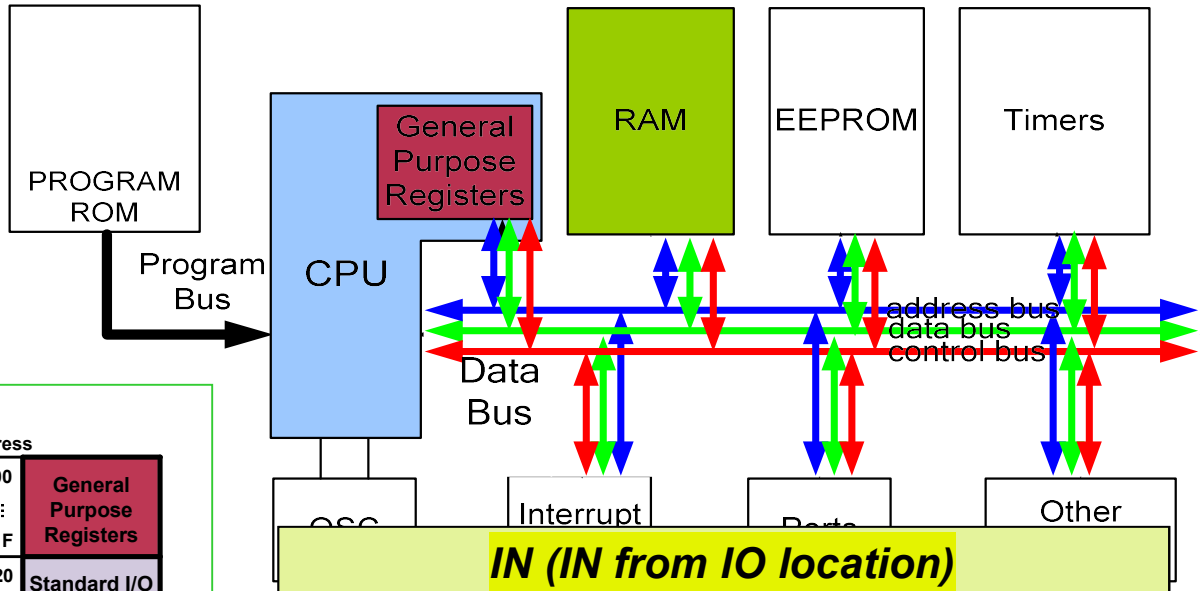
Answer:

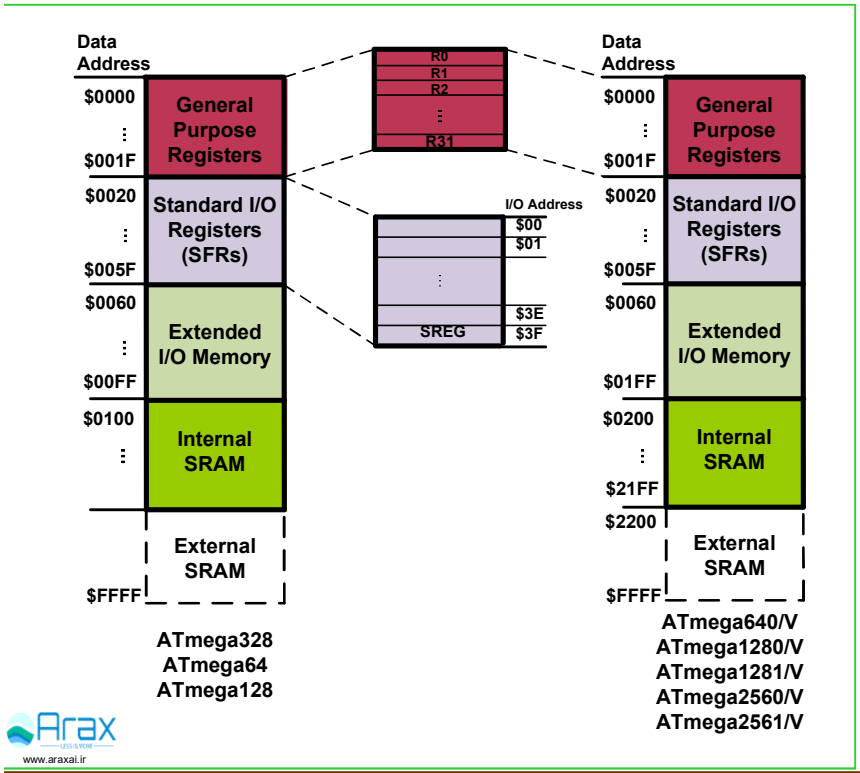
```
LDI    R20, 0x53          ;R20 = 0x53
STS    0x5E, R20          ;SPH = R20
```

It copies the contents of R2 into R20; as 2 is the address of R2.

Data Address Space

IN instruction is 2 byte
LDS is 4 byte





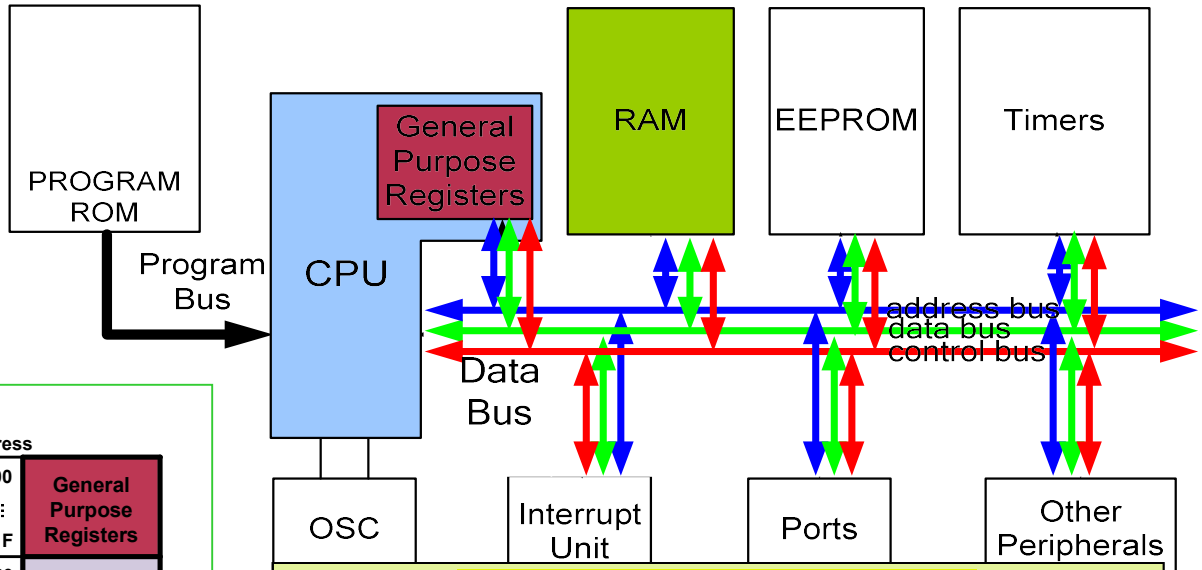
IN (IN from IO location)

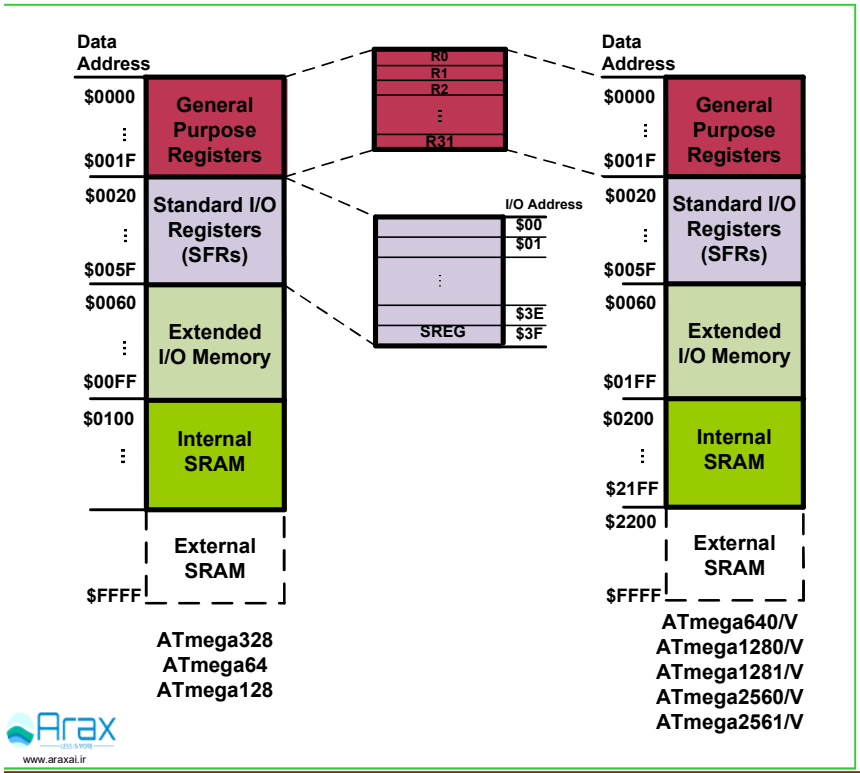
IN Rd,IOaddress ;Rd = [addr]

Example:

```
IN  R1, 0x3F    ;R1 = SREG
IN  R17,0x3E    ;R17 = SPH
```

Data Address Space





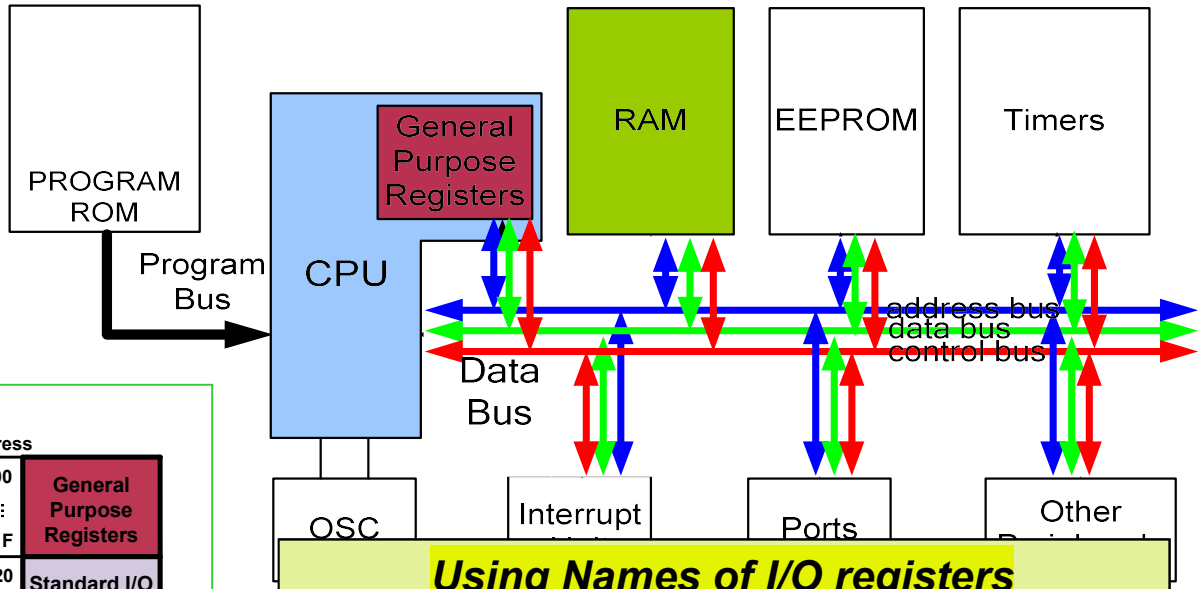
OUT (OUT to I/O location)

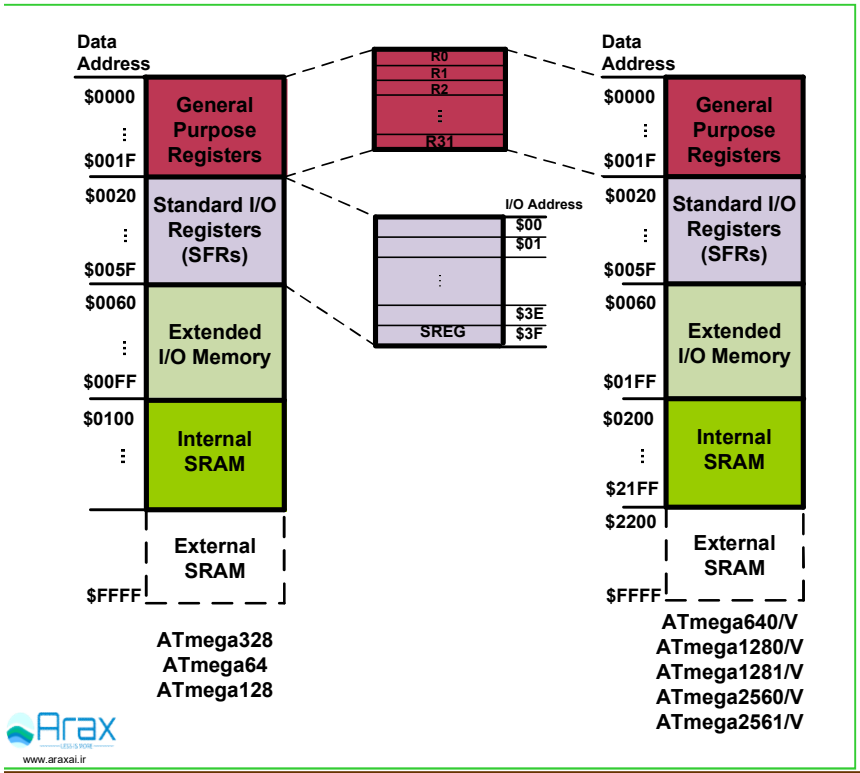
OUT IOAddr,Rd ;[addr]=Rd

Example:

```
OUT  0x3F,R12    ;SREG = R12
OUT  0x3E,R15    ;SPH = R15
```

Data Address Space



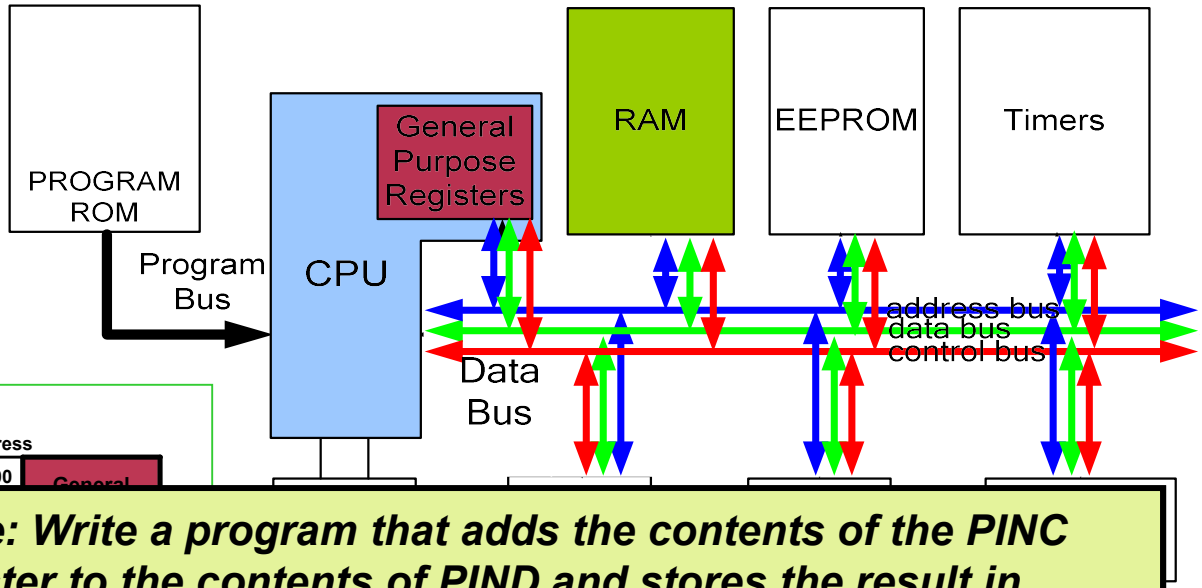


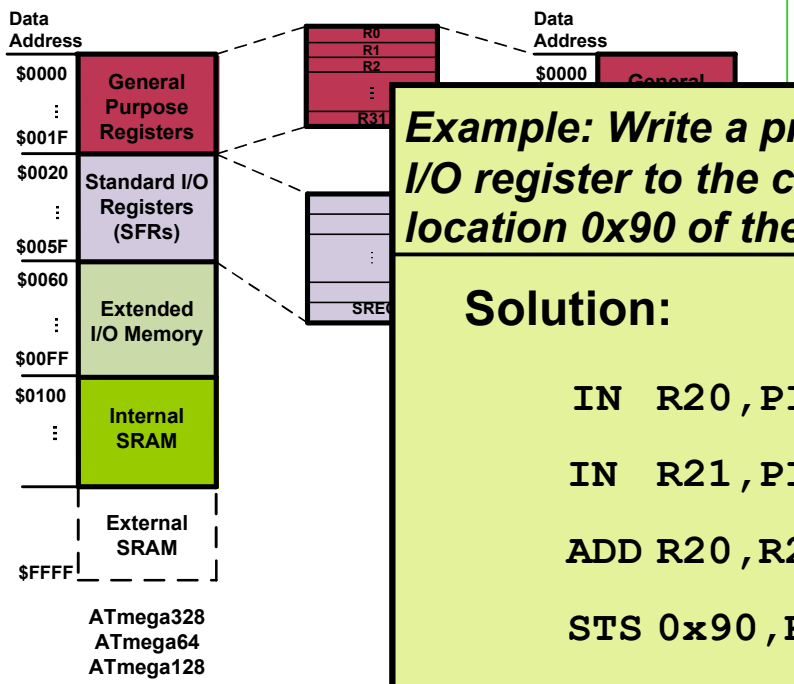
Using Names of I/O registers

Example:

```
OUT   SPH,R12      ;OUT   0x3E,R12
IN    R15,SREG     ;IN    R15,0x3F
```

Data Address Space





Example: Write a program that adds the contents of the PINC I/O register to the contents of PIND and stores the result in location 0x90 of the SRAM

Solution:

```
IN  R20,PINC   ;R20 = PINC
IN  R21,PIND   ;R21 = PIND
ADD R20,R21    ;R20 = R20 + R21
STS 0x90,R20   ;[0x90] = R20
```

Machine Language

- ADD R0,R1

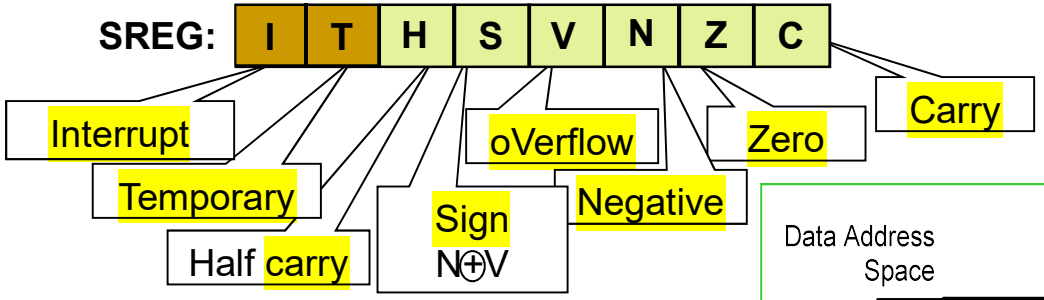
000011 00 0000 0001
opcode operand

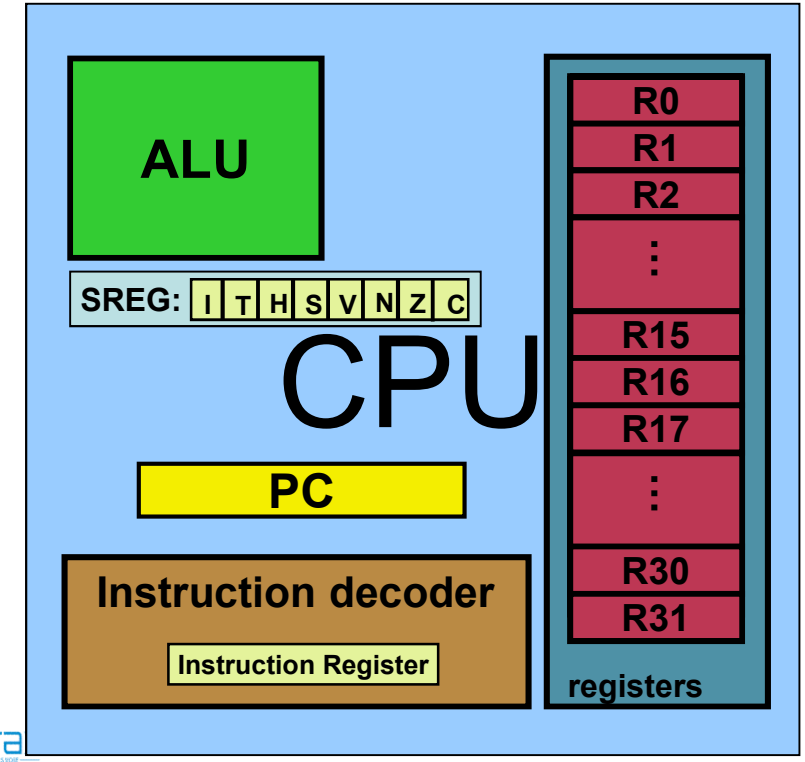
- LDI R16, 2
  LDI R17, 3
  ADD R16, R17

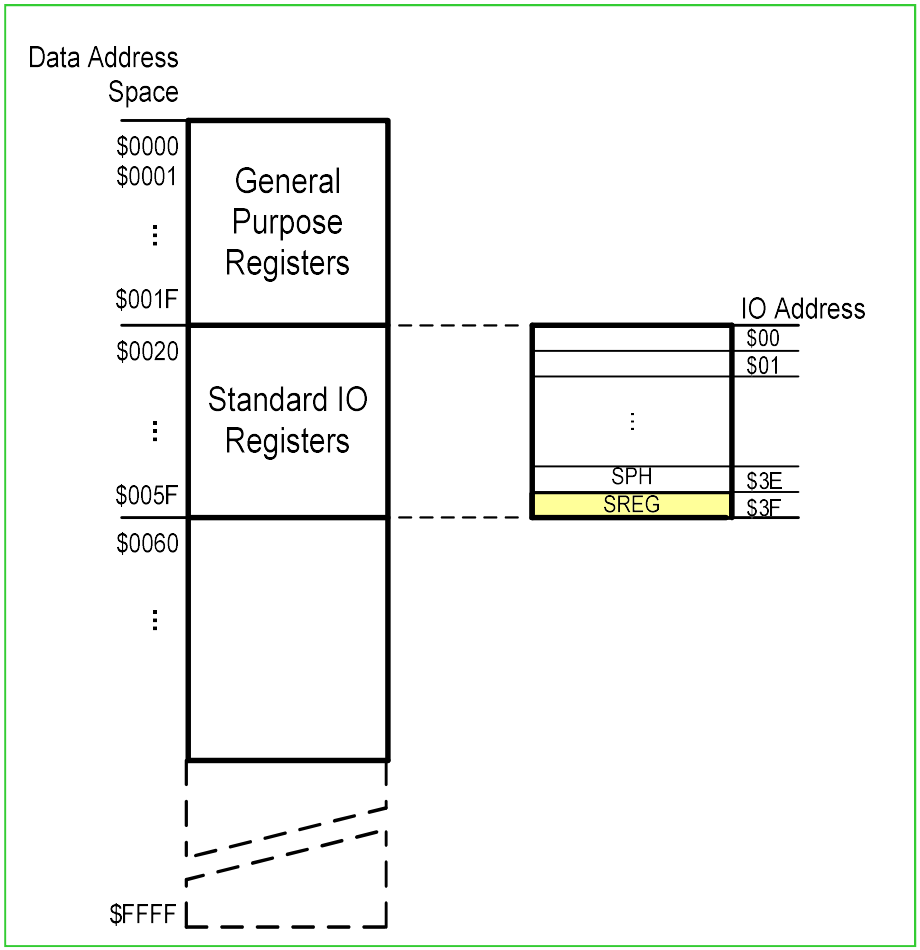
1110 0000 0000 0010
1110 0000 0001 0011
0000 1111 0000 0001

IN instruction is 2 byte
LDS is 4 byte

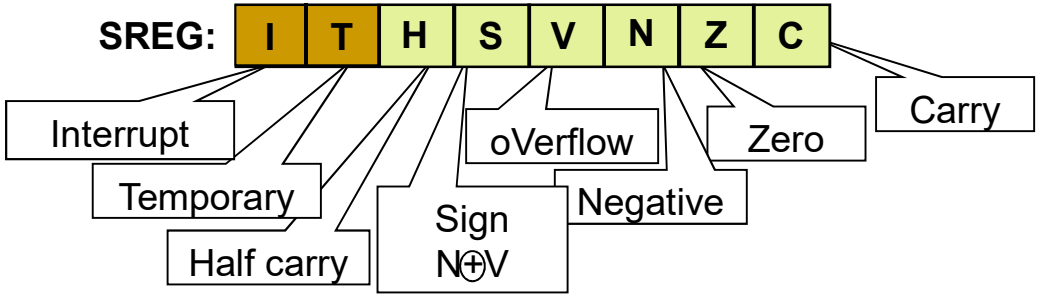
Status Register (SREG)







Status Register (SREG)



- C, the carry flag (D7->D8)

  a carry out from the D7 bit. This flag bit(after an 8-bit addition or subtraction)

- Z, the zero flag

  The zero result (after arithmetic or logic operation)

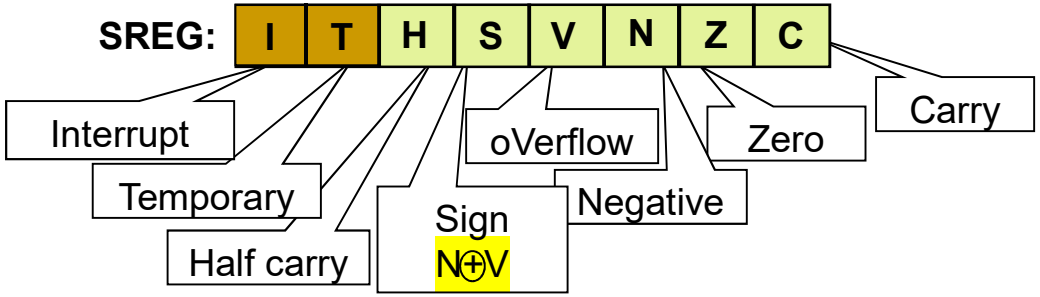
- N, the negative flag (D7)

  Binary representation of signed numbers uses D7 (the result of an arithmetic)

- V, the overflow flag

  causing the high-order bit to overflow into the sign bit.

Status Register (SREG)



•S, the Sign bit

This flag is the result of Exclusive-ORing of N and V flags.

•H, Half carry flag

If there is a carry from D3 to D4 during an ADD or SUB operation

Status Register (SREG)

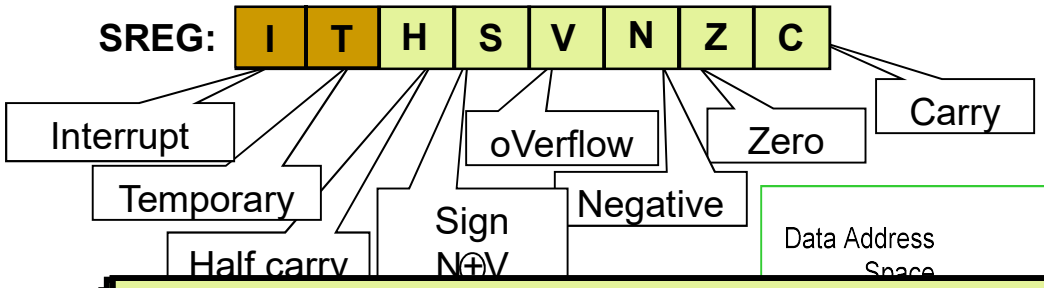


Table 2-5: AVR Branch (Jump) Instructions Using Flag Bits	
Instruction	**Action**
BRLO	Branch if C = 1
BRSH	Branch if C = 0
BREQ	Branch if Z = 1
BRNE	Branch if Z = 0
BRMI	Branch if N = 1
BRPL	Branch if N = 0
BRVS	Branch if V = 1
BRVC	Branch if V = 0

Example: Show the status of the C, H, and Z flags aft
subtraction of 0x9C from 0x9C in the following instru

```
LDI     R20, 0x9C
LDI     R21, 0x9C
SUB     R20, R21        ;subtract R21 from R20
```

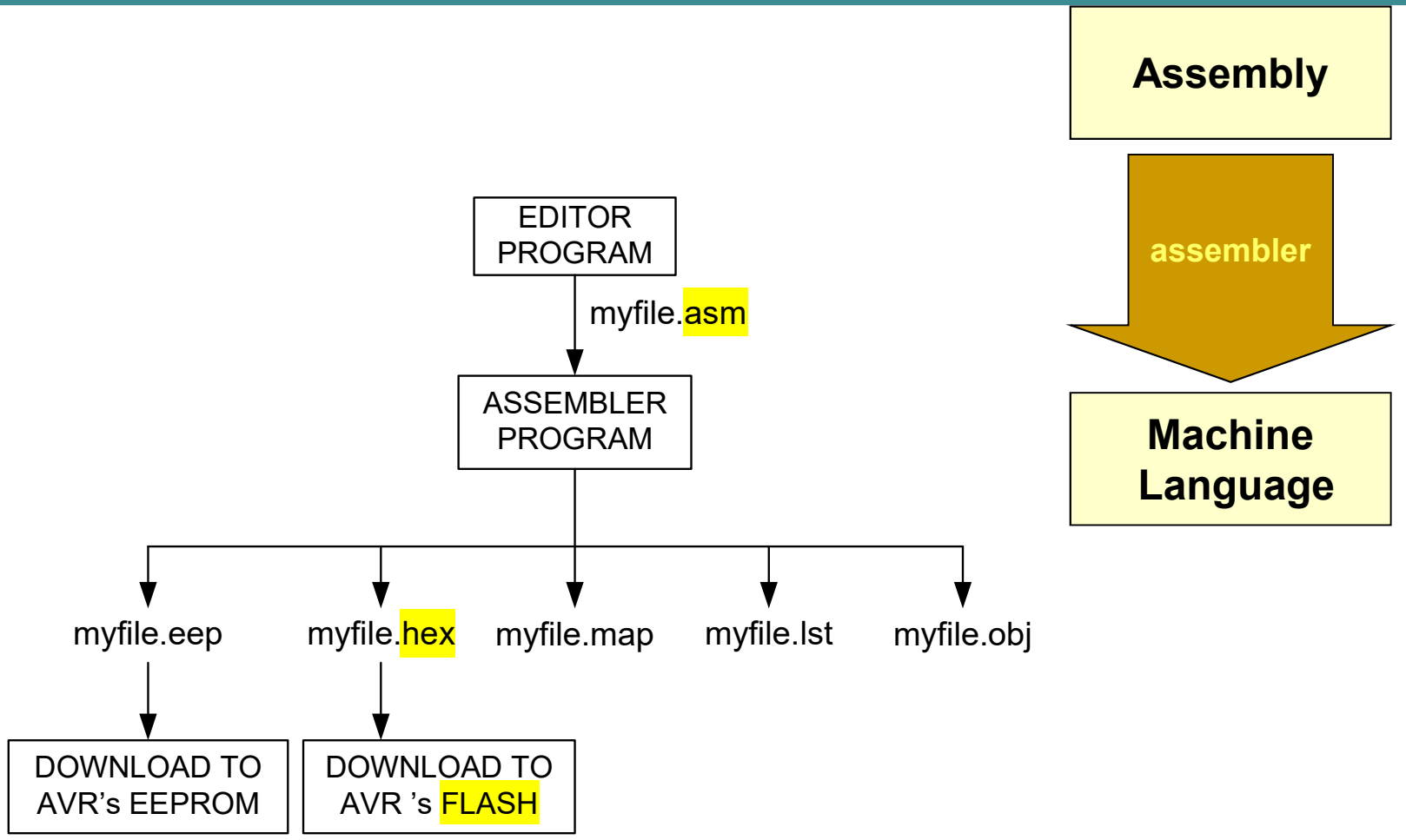
Solution:

	$9C	1001 1100	
-	$9C	1001 1100	
	$00	0000 0000	R20 = $00

C = 0 because R21 is not bigger than R20 and there is no borrow from D8 bit.
Z = 1 because the R20 is zero after the subtraction.
H = 0 because there is no borrow from D4 to D3.

AVR ASSEMBLY PROGRAMMING

Assembler


Assembly
assembler
Machine Language
EDITOR PROGRAM
myfile.asm
ASSEMBLER PROGRAM
myfile.eep
myfile.hex
myfile.map
myfile.lst
myfile.obj
DOWNLOAD TO AVR's EEPROM
DOWNLOAD TO AVR 's FLASH

Assembler_Map

```
AVRASM ver. 2.1.2   F:\AVR\Sample\Sample.asm Sun Apr 06 23:39:32 2008


EQU   SUM             00000300
CSEG  HERE            00000009
```

Figure 11. Map File of Program 1

the labels defined

Assembler_“lst”

```
AVRASM ver. 2.1.2  F:\AVR\Sample\Sample.asm Tue Mar 11 11:28:34 2008

                ;store SUM in SRAM location 0x300.
                   .DEVICE ATMega32
                   .EQU  SUM   = 0x300     ;SRAM loc $300 for SUM

                   .ORG 00                 ;start at address 0
000000 e205        LDI R16, 0x25           ;R16 = 0x25
000001 e314        LDI R17, $34            ;R17 = 0x34
000002 e321        LDI R18, 0b00110001     ;R18 = 0x31
000003 0f01        ADD R16, R17            ;add R17 to R16
000004 0f02        ADD R16, R18            ;add R18 to R16
000005 e01b        LDI R17, 11             ;R17 = 0x0B
000006 0f01        ADD R16, R17            ;add R17 to R16
000007 9300 0300   STS SUM, R16            ;save the SUM in loc $300
000009 940c 0009 HERE: JMP HERE            ;stay here forever

RESOURCE USE INFORMATION
------------------------
...
Memory use summary [bytes]:
Segment    Begin     End       Code    Data    Used     Size    Use%
---------------------------------------------------------------------
[.cseg] 0x000000 0x000016      22       0      22 unknown       -
[.dseg] 0x000060 0x000060       0       0       0 unknown       -
[.eseg] 0x000000 0x000000       0       0       0 unknown       -

Assembly complete, 0 errors, 0 warnings
```

Figure 12. List File of Program 1

AVR-ROM

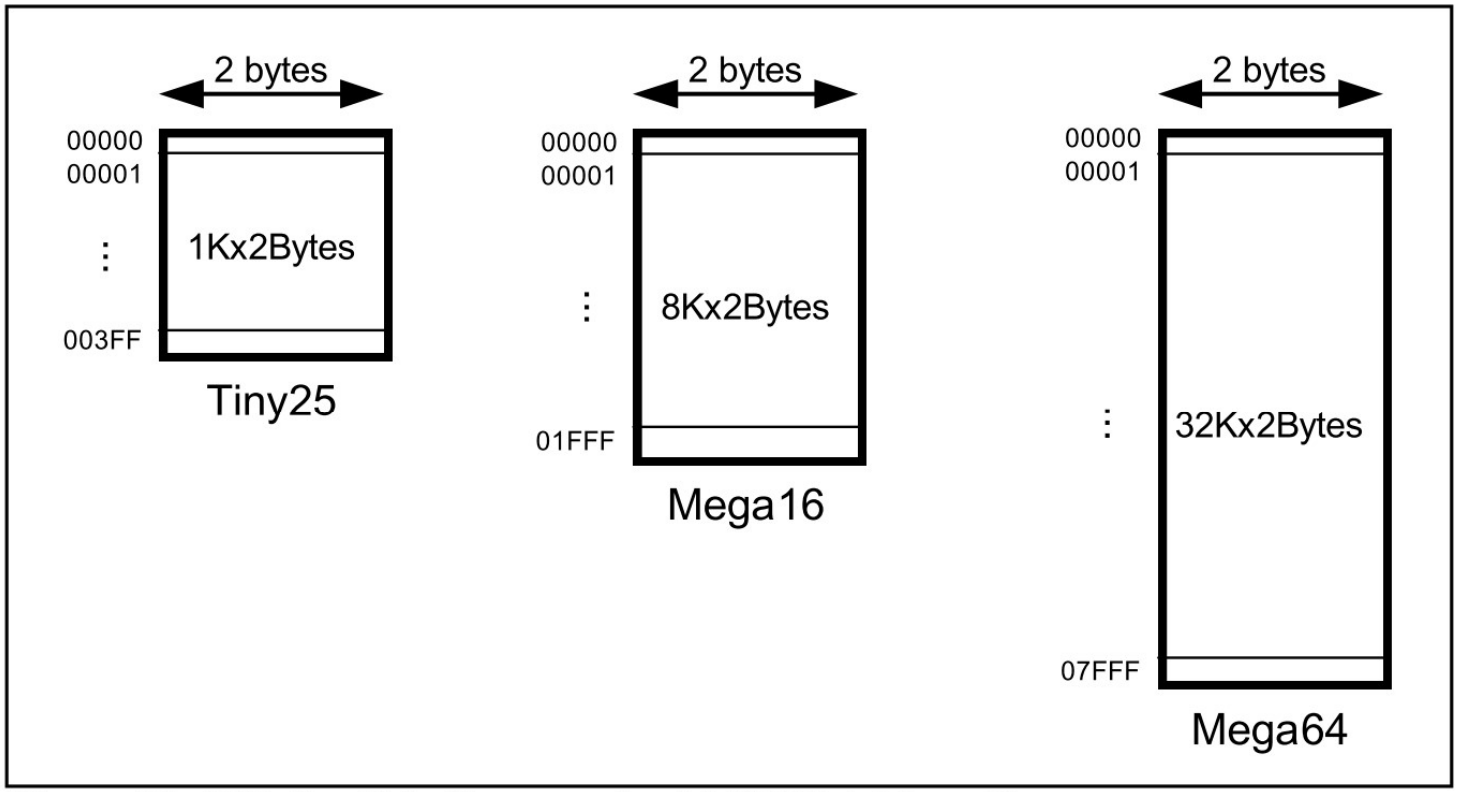


Figure 13. AVR On-Chip Program (code) ROM Address Range

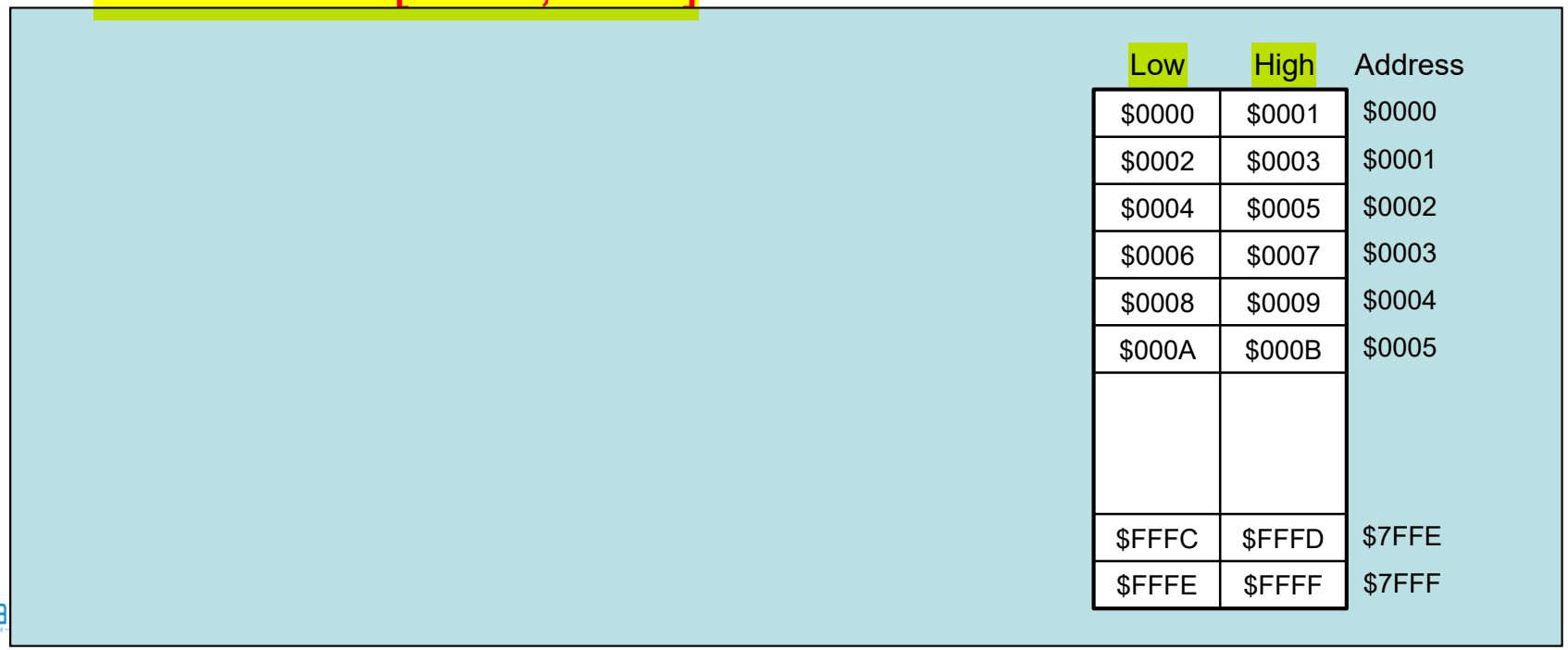
AVR-ROM

Table 7: AVR On-chip ROM Size and Address Space

	On-chip Code ROM (Bytes)	Code Address Range (Hex)	ROM Organization
ATtiny25	2K	00000–003FF	1K × 2 bytes
ATmega8	8K	00000–00FFF	4K × 2 bytes
ATmega32	32K	00000–03FFF	16K × 2 bytes
ATmega64	64K	00000–07FFF	32K × 2 bytes
ATmega128	128K	00000–0FFFF	64K × 2 bytes
ATmega256	256K	00000–1FFFF	128K × 2 bytes

Little-Endian Big-Endian

- AVR is Little-endian
- The low byte goes to the low memory location, and the high byte goes to the high memory address
- Memory Locations [AD_Low,AD_High]
- 0x1234 -> [0x34,0x12]



Assembler Directives

.EQU and .SET

- .EQU *name = value*
  - *Example:*

```
.EQU    COUNT = 0x25
LDI     R21, COUNT              ;R21 = 0x25
LDI     R22, COUNT + 3          ;R22 = 0x28
```

- .SET *name = value*
  - *Example:*

```
.SET    COUNT = 0x25
 LDI    R21, COUNT              ;R21 = 0x25
 LDI    R22, COUNT + 3          ;R22 = 0x28
.SET    COUNT = 0x19
 LDI    R21, COUNT              ;R21 = 0x19
```

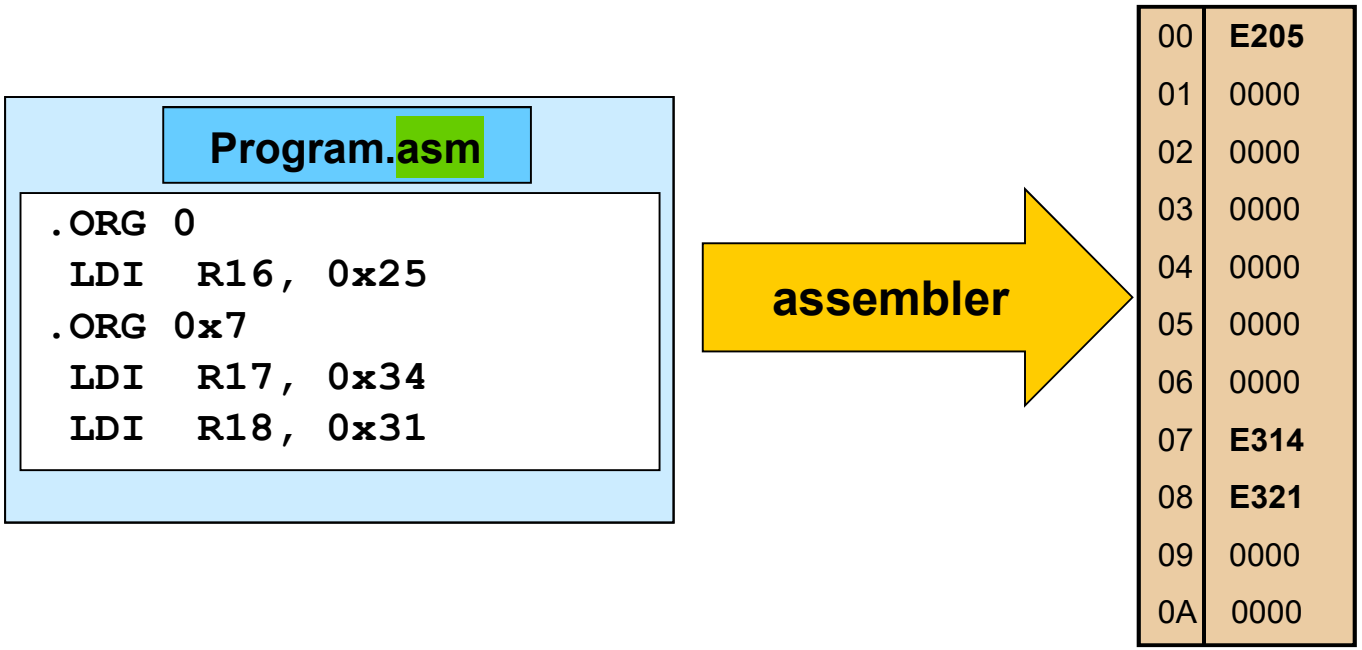
Advantage?
Difference?

Several Use possible

Assembler Directives

.ORG

- .ORG *address*



Assembler Directives

.INCLUDE

- .INCLUDE "*filename.ext*"

Table 2-6: Some of the Common AVRs and Their Include Files

ATM	
ATme	.inc
ATme	.inc
ATme	f.inc
ATme	f.inc
ATme	ef.inc
ATme	

M328def.inc

```
.equ    SREG    = 0x3f
.equ    SPL     = 0x3d
.equ    SPH     = 0x3e
....
```

Program.asm

```
    LDI   R20, 10
    OUT   SPL, R20
```

Assembler Directives

.Label

```
AVRASM ver. 2.1.2  F:\AVR\Sample\Sample.asm Tue Mar 11 11:28:34 2008

                ;store SUM in SRAM location 0x300.
                   .DEVICE ATMega32
                   .EQU  SUM   = 0x300     ;SRAM loc $300 for SUM

                  .ORG 00                  ;start at address 0
000000 e205       LDI R16, 0x25            ;R16 = 0x25
000001 e314       LDI R17, $34             ;R17 = 0x34
000002 e321       LDI R18, 0b00110001      ;R18 = 0x31
000003 0f01       ADD R16, R17             ;add R17 to R16
000004 0f02       ADD R16, R18             ;add R18 to R16
000005 e01b       LDI R17, 11              ;R17 = 0x0B
000006 0f01       ADD R16, R17             ;add R17 to R16
000007 9300 0300  STS SUM, R16             ;save the SUM in loc $300
000009 940c 0009 HERE: JMP HERE            ;stay here forever

RESOURCE USE INFORMATION
------------------------
...
Memory use summary [bytes]:
Segment   Begin     End       Code    Data    Used    Size    Use%
----------------------------------------------------------------------
[.cseg] 0x000000 0x000016      22       0      22 unknown       -
[.dseg] 0x000060 0x000060       0       0       0 unknown       -
[.eseg] 0x000000 0x000000       0       0       0 unknown       -

Assembly complete, 0 errors, 0 warnings
```

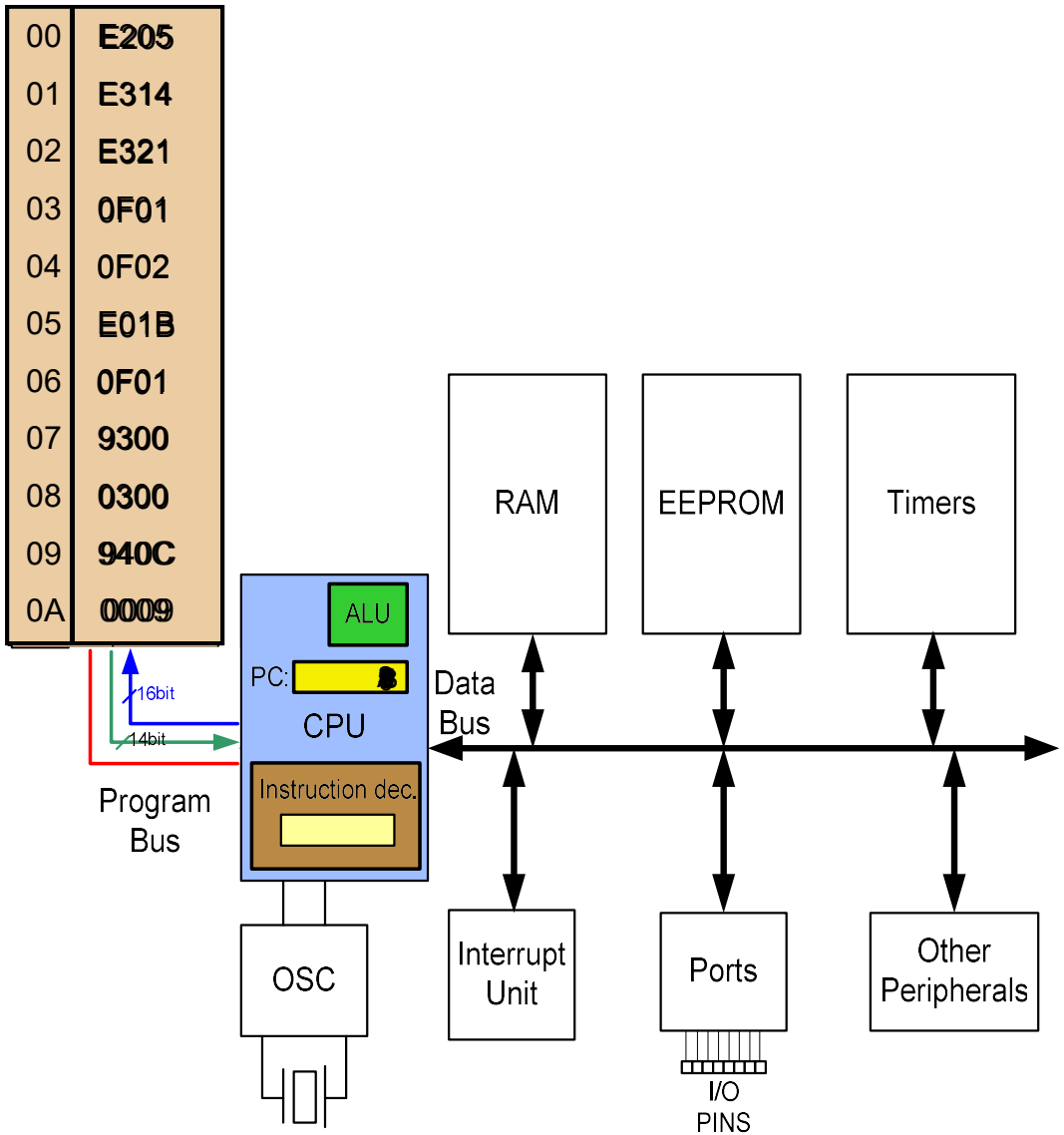
Figure 12. List File of Program 1

Flash memory and PC register

```
      LDI   R16, 0x25
      LDI   R17, $34
      LDI   R18, 0x31
      ADD   R16, R17
      ADD   R16, R18
      LDI   R17, 11
      ADD   R16, R17
      STS   SUM, R16
HERE: JMP  HERE
```

Code?

- The program counter is used by the CPU to point to the address of the next instruction to be executed
- Harvard



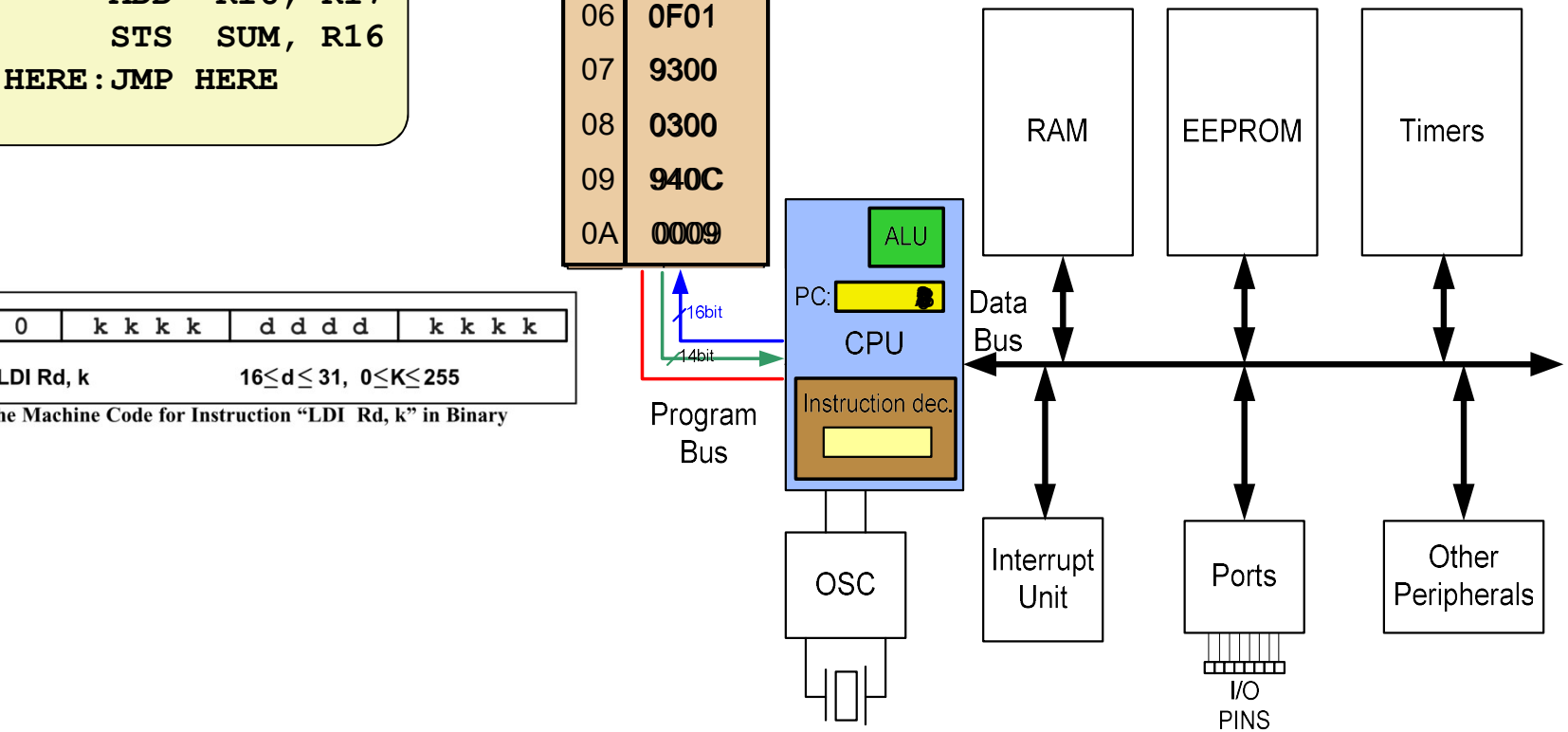
Flash memory and PC register

```
      LDI   R16, 0x25
      LDI   R17, $34
      LDI   R18, 0x31
      ADD   R16, R17
      ADD   R16, R18
      LDI   R17, 11
      ADD   R16, R17
      STS   SUM, R16
HERE: JMP  HERE
```

00	E205
01	E314
02	E321
03	0F01
04	0F02
05	E01B
06	0F01
07	9300
08	0300
09	940C
0A	0009

1 1 1 0	k k k k	d d d d	k k k k
LDI Rd, k		16≤d≤31, 0≤K≤255	

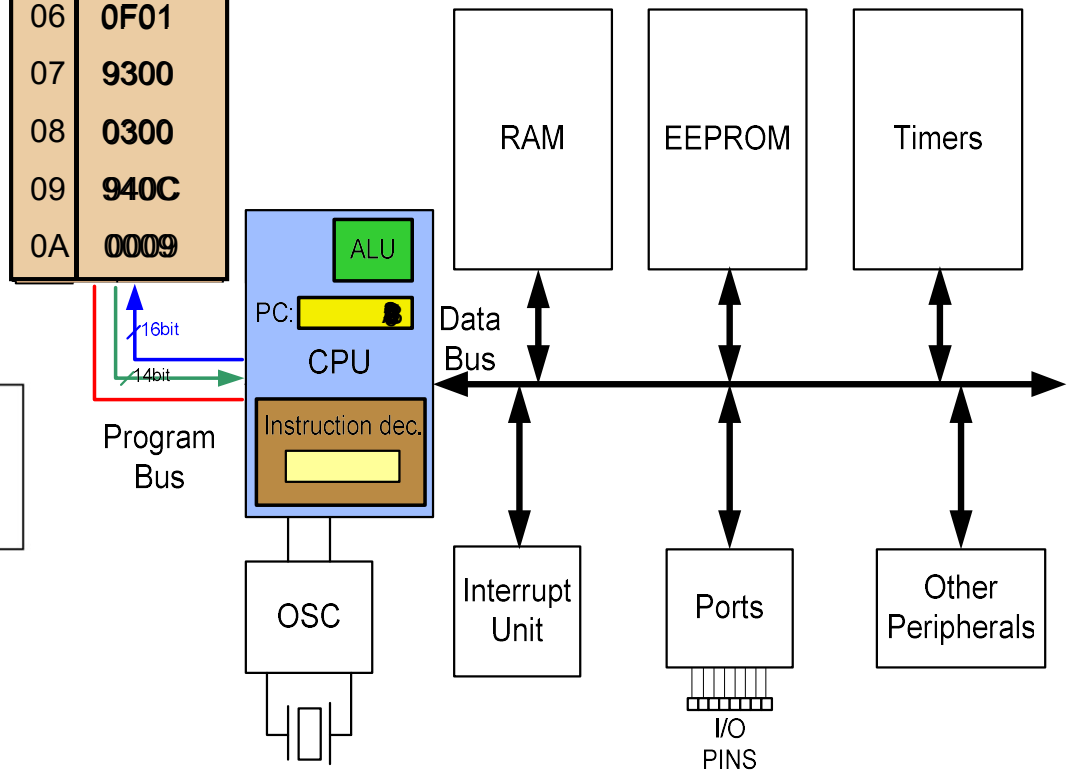
Figure 14. The Machine Code for Instruction "LDI Rd, k" in Binary



Flash memory and PC register

```
        LDI   R16, 0x25
        LDI   R17, $34
        LDI   R18, 0x31
        ADD   R16, R17
        ADD   R16, R18
        LDI   R17, 11
        ADD   R16, R17
        STS   SUM, R16
HERE:JMP  HERE
```

00	E205
01	E314
02	E321
03	0F01
04	0F02
05	E01B
06	0F01
07	9300
08	0300
09	940C
0A	0009



STS k, Rr ;Store register *Rr* in memory location *k*

1001	001r	rrrr	0000
kkkk	kkkk	kkkk	kkkk

$0 \le r \le 31, 0 \le k \le 65535$

Why 4 Byte?

Instruction size of the AVR

```
JMP k    ;Jump to address k
```

1001	010k	kkkk	110k
kkkk	kkkk	kkkk	kkkk

$0 \le k \le 4M$

```
OUT A, Rr   ;Store register Rr in I/O memory location A
```

1011	1AAr	rrrr	AAAA

$0 \le d \le 31, 0 \le A \le 63$

```
IN Rd, A   ;load from Address A of I/O memory into register Rd
```

1011	0AAd	dddd	AAAA

$0 \le d \le 31, 0 \le A \le 63$

```
LDI Rd, K    ;load register Rd with value K
```

1110	KKKK	dddd	KKKK

$16 \le d \le 31, 0 \le K \le 255$

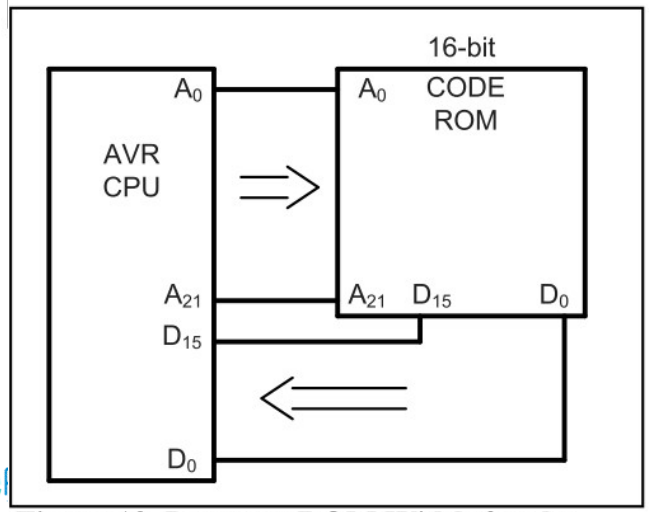
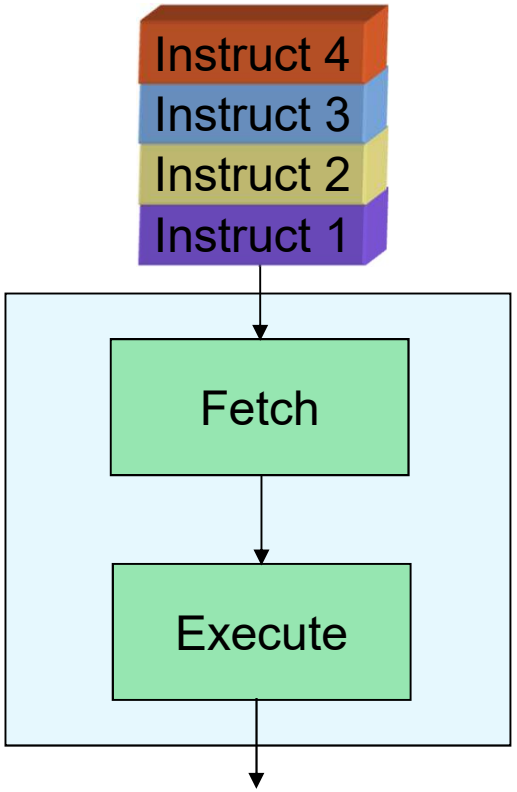
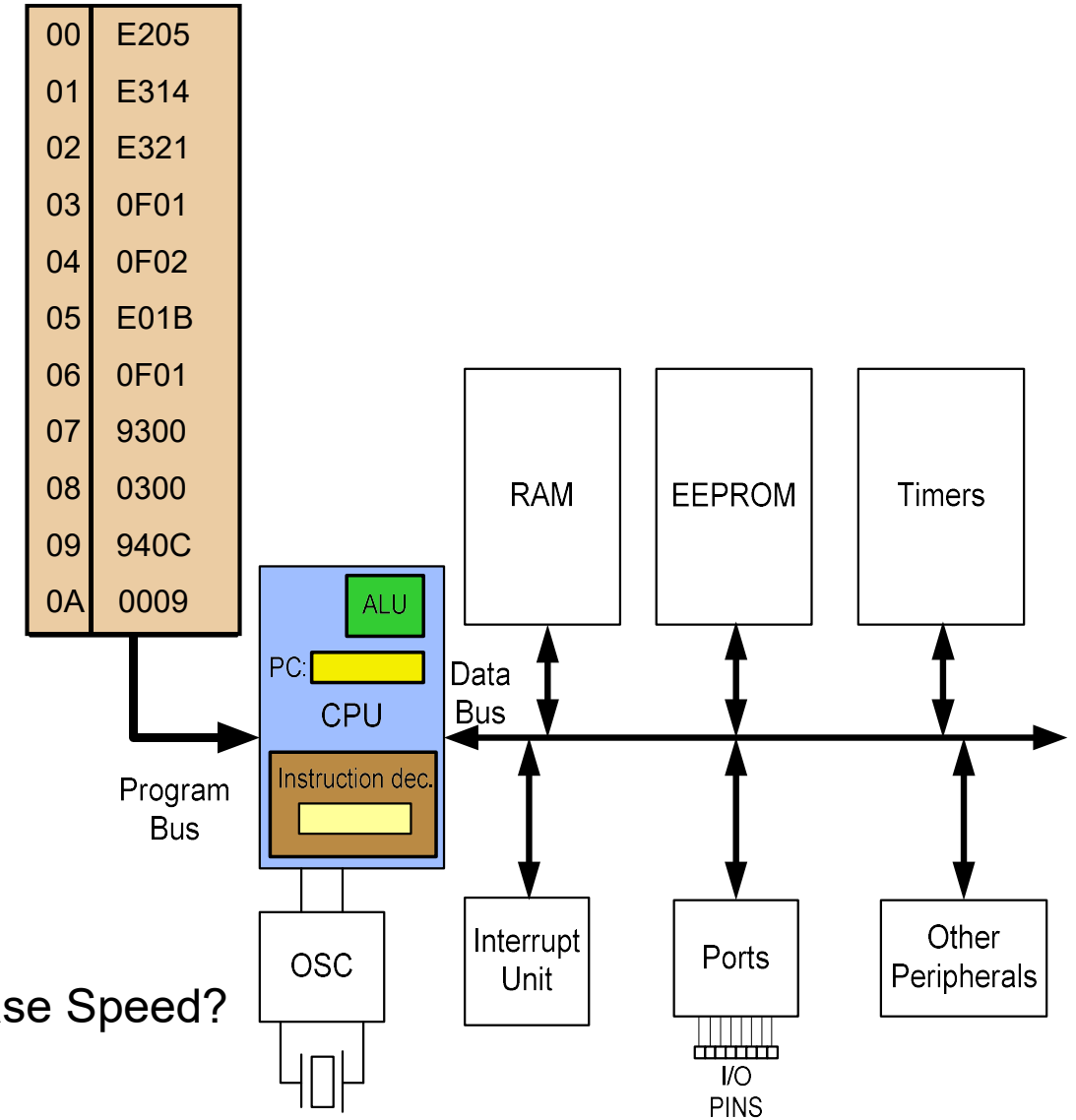


Figure 18. Program ROM Width for the AVR

Fetch and execute

- Old Architectures

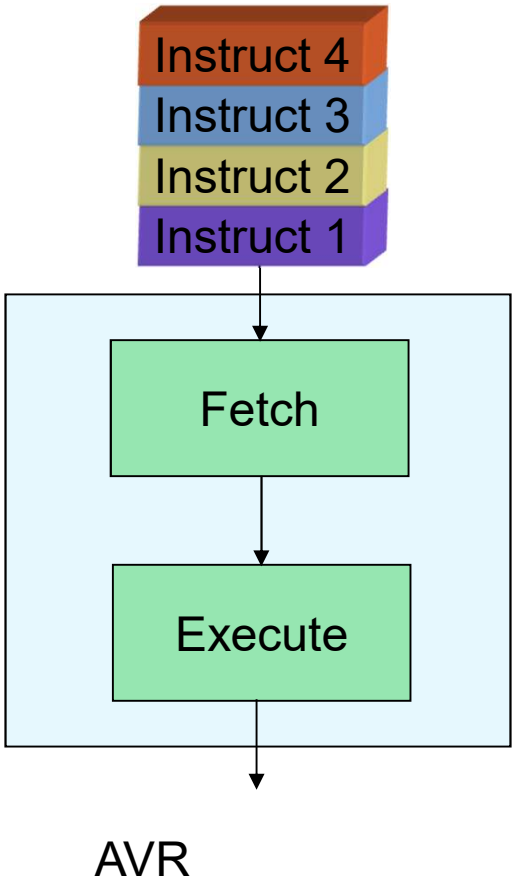




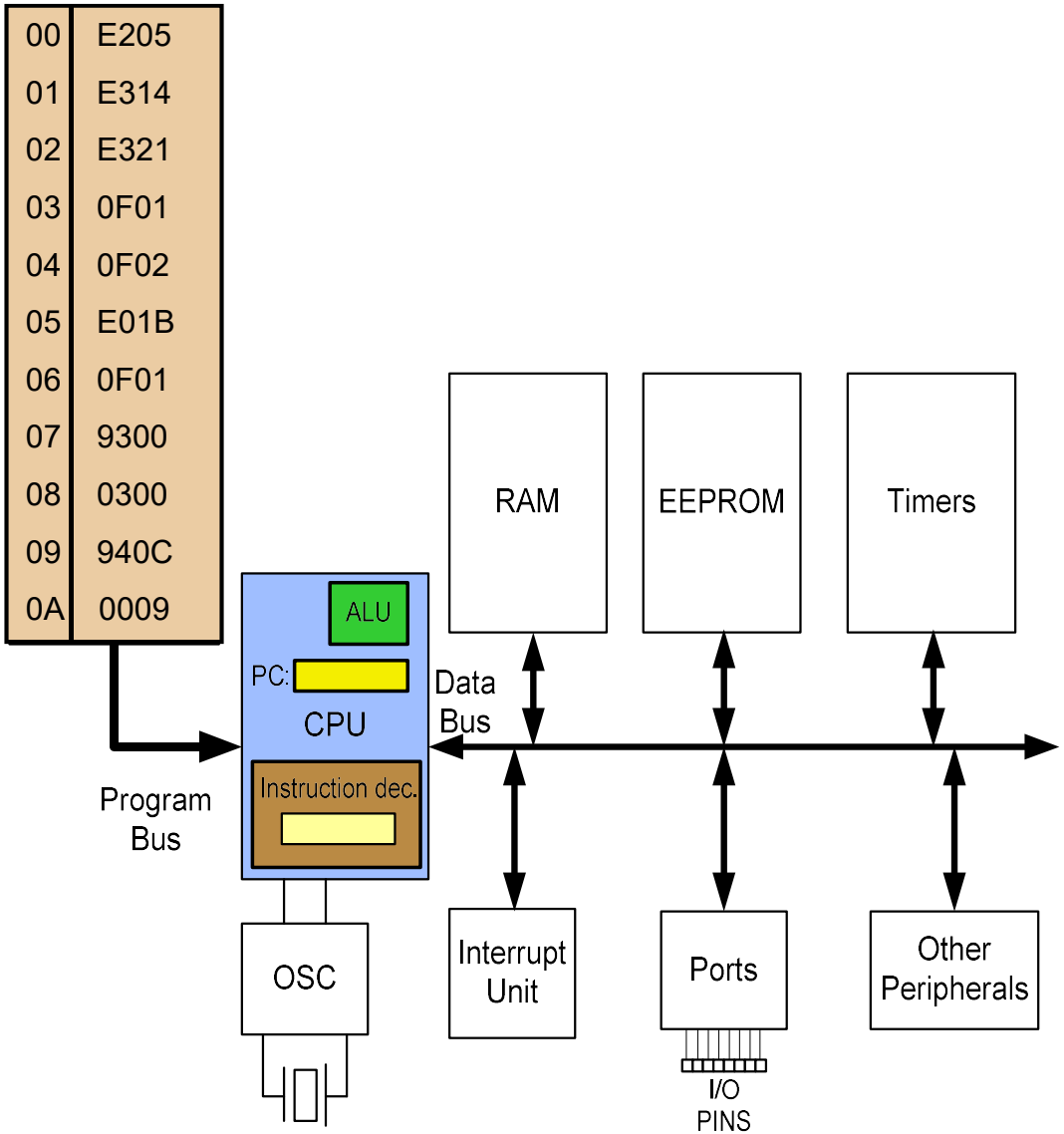
8085
Increase Speed?

Pipelining

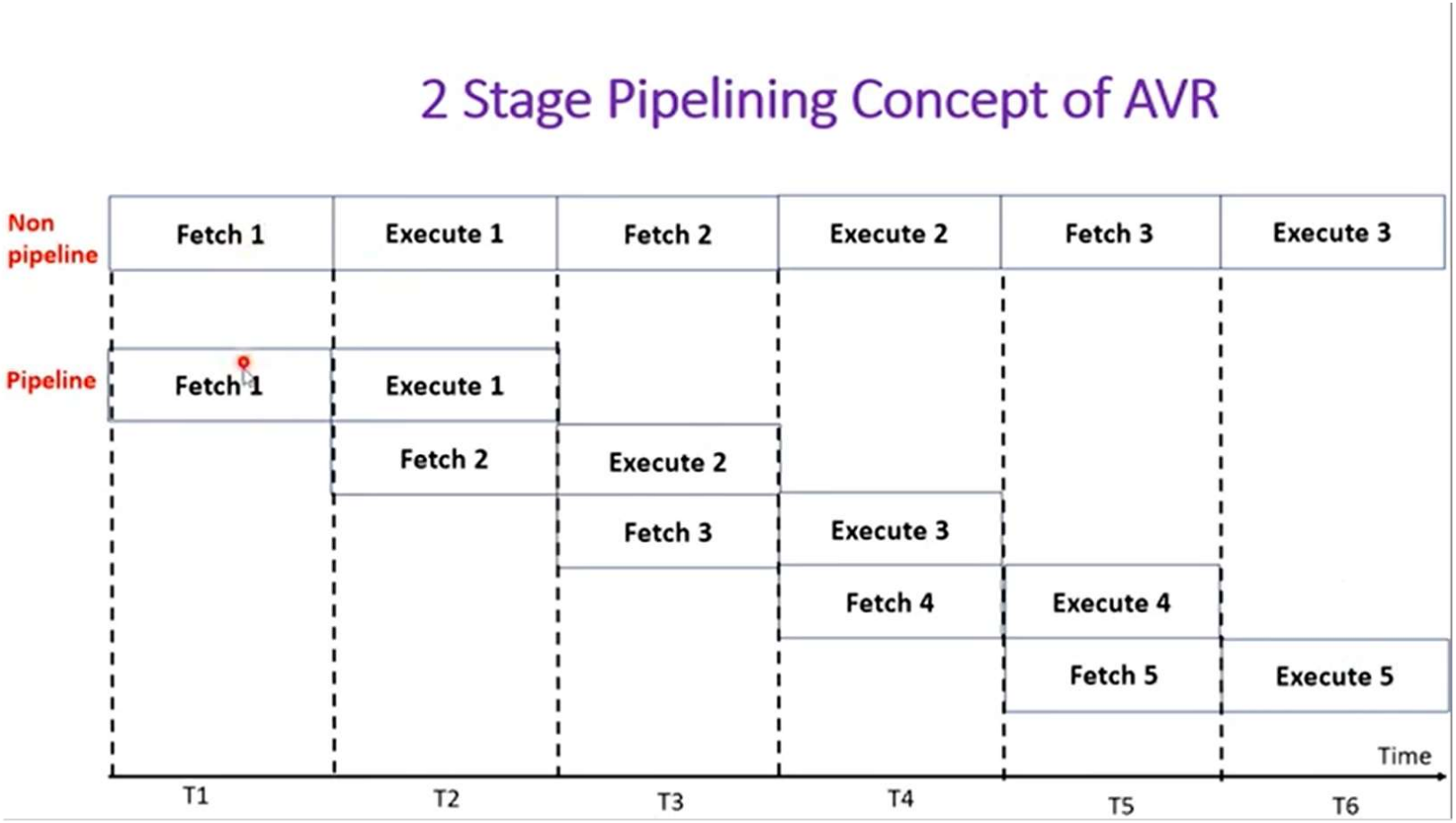
- Pipelining



AVR



2 Stage Pipelining



2 Stage Pipelining-Execution

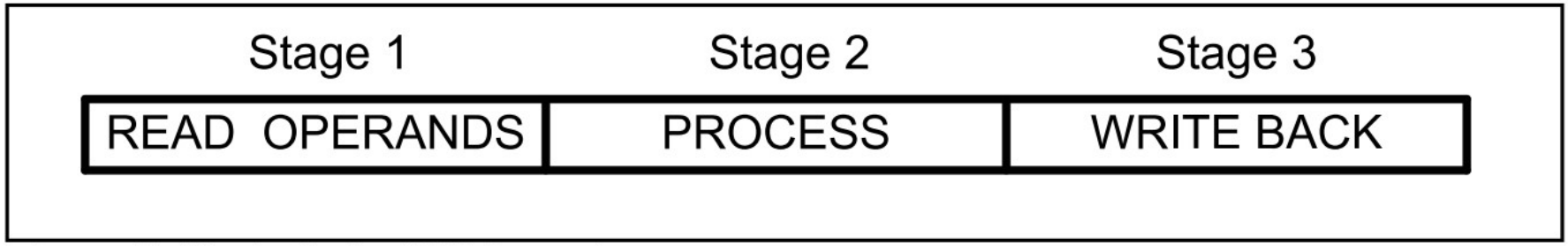


Figure 13. Single Cycle ALU Operation

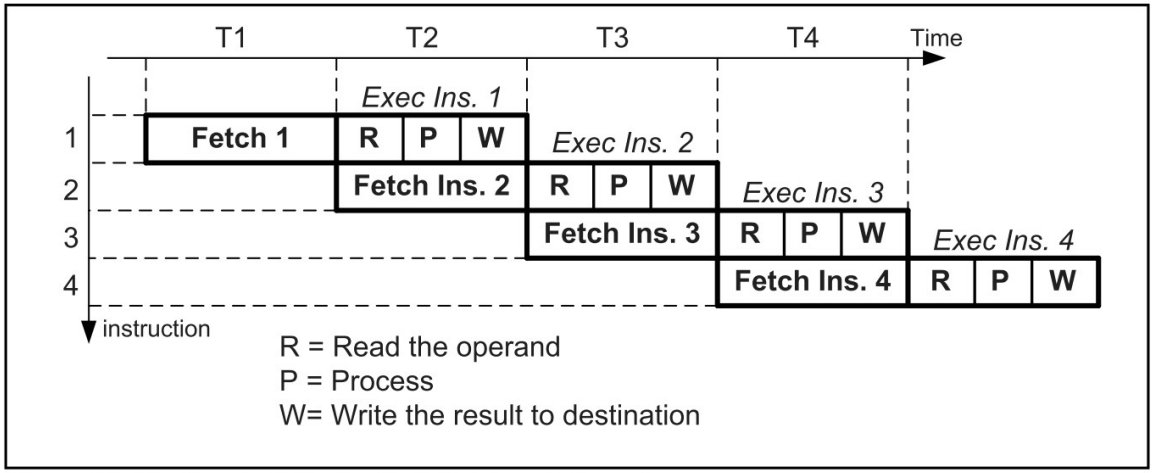
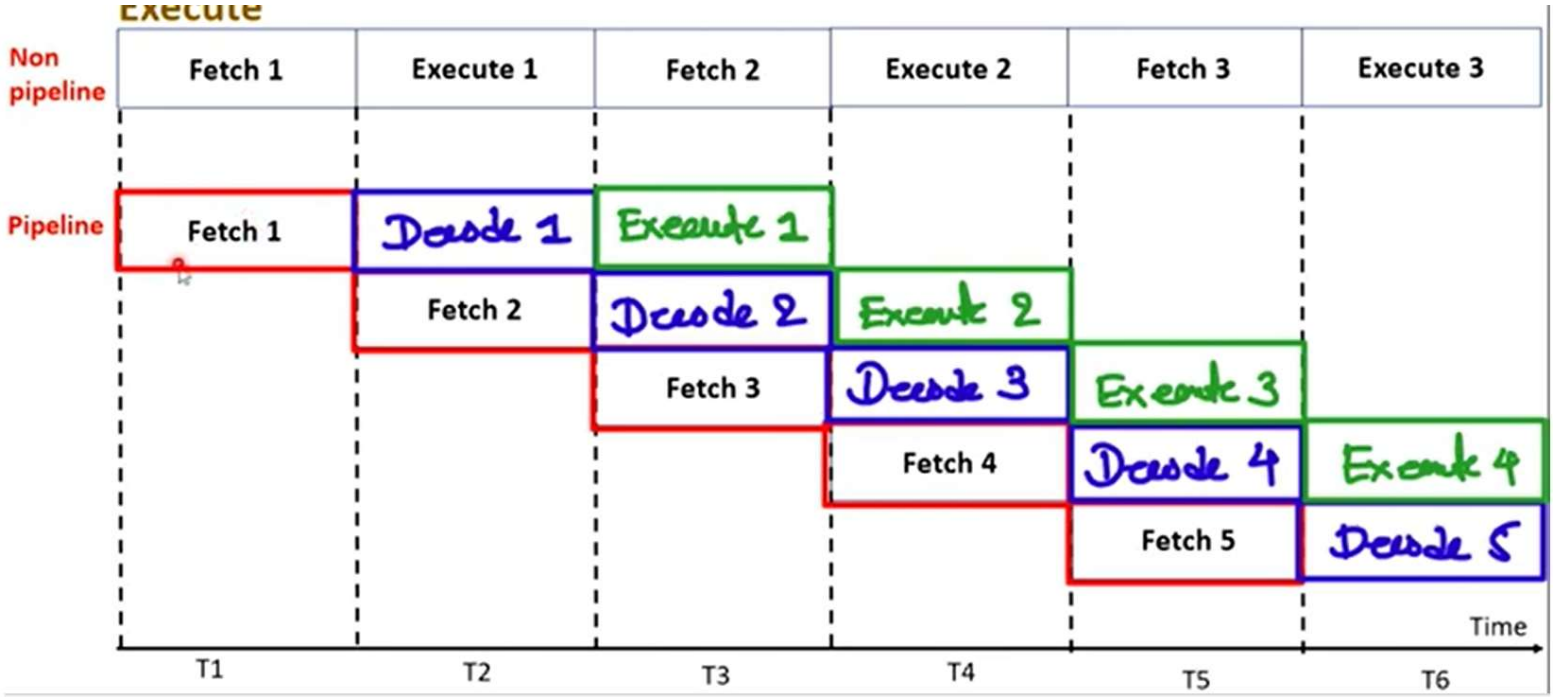


Figure 14. Pipeline Activity for Both Fetch and Execute

3 Stage Pipelining



Emu 8086

- Mov Example
- BuferOverwrite

RISC ARCHITECTURE

How to speed up the CPU

1. Increase the clock frequency
   - More frequency ➔
     - More power consumption & more heat
     - Limitations
2. Change the architecture to Harvard(Pipelining)
3. Change internal architecture of CPU: RISC

Changing the architecture
RISC vs. CISC

- CISC (Complex Instruction Set Computer)
  - Put as many instruction as you can into the CPU
  - greater complexity on the hardware side.
- RISC (Reduced Instruction Set Computer)
  - Reduce the number of instructions, and use your facilities in a more proper way.

RISC architecture

- Feature 1
  - RISC processors have a fixed instruction size. It makes the task of instruction decoder easier.
    - In AVR the instructions are 2 or 4 bytes.
  - In CISC processors instructions have different lengths
    - E.g. in 8051
      - CLR C ; a 1-byte instruction
      - ADD A, #20H ; a 2-byte instruction
      - LJMP HERE ; a 3-byte instruction

RISC architecture

- Feature 2: reduce the number of instructions
  - Pros: Reduces the number of used transistors
  - Cons:
    - Can make the assembly programming more difficult
    - Can lead to using more memory

RISC architecture

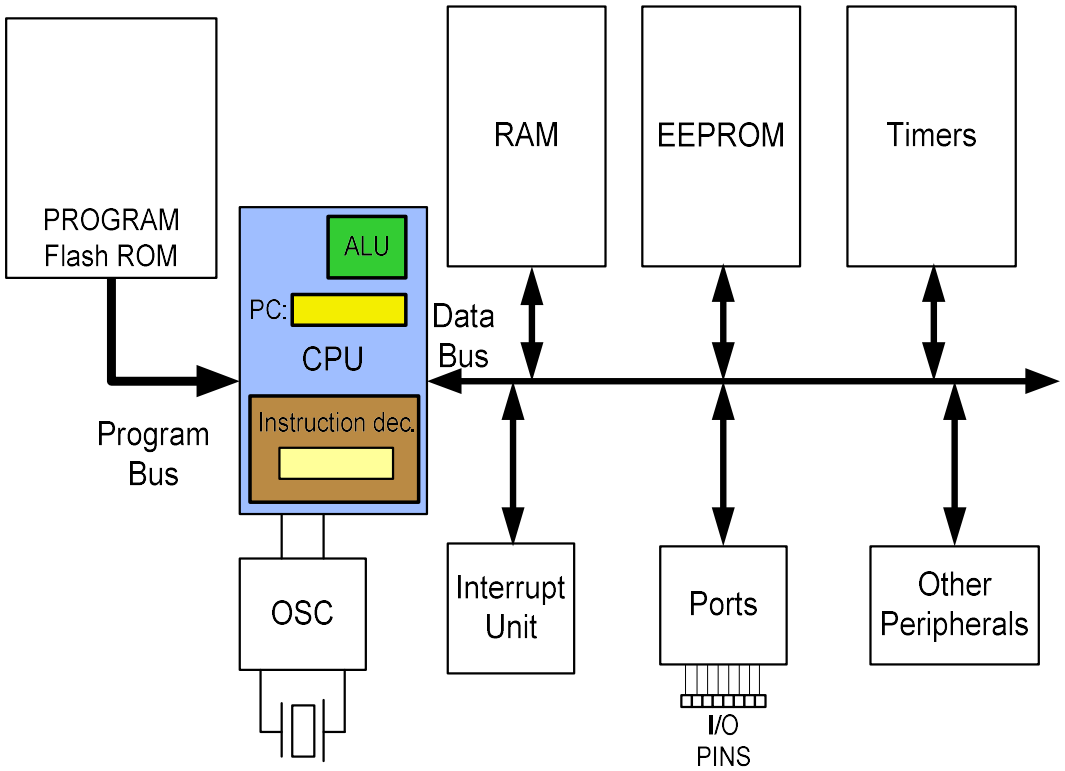
- Feature 3: limit the addressing mode
  - Advantage
    - hardwiring
  - Disadvantage
    - Can make the assembly programming more difficult

RISC architecture

- Feature 4: Load/Store

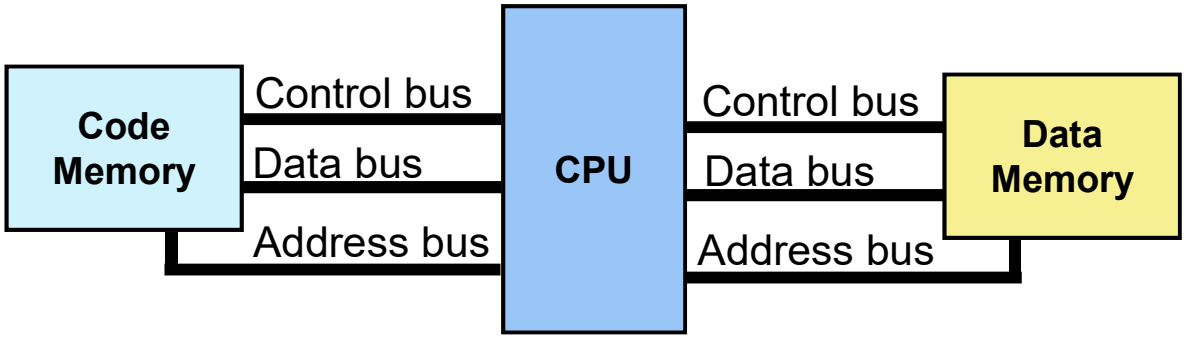
```
LDS R20, 0x200
LDS R21, 0x220
ADD R20, R21
STS 0x230, R20
```

- ~~ADD R21,memory(not risc)~~

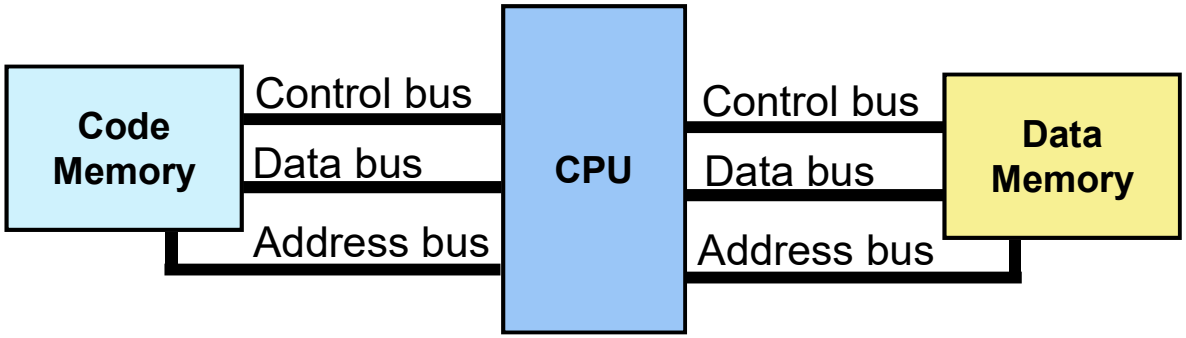


RISC architecture

- Feature 5 (Harvard architecture): separate buses for opcodes and operands
  - Advantage: opcodes and operands can go in and out of the CPU together.
  - Disadvantage: leads to more cost in general purpose computers.

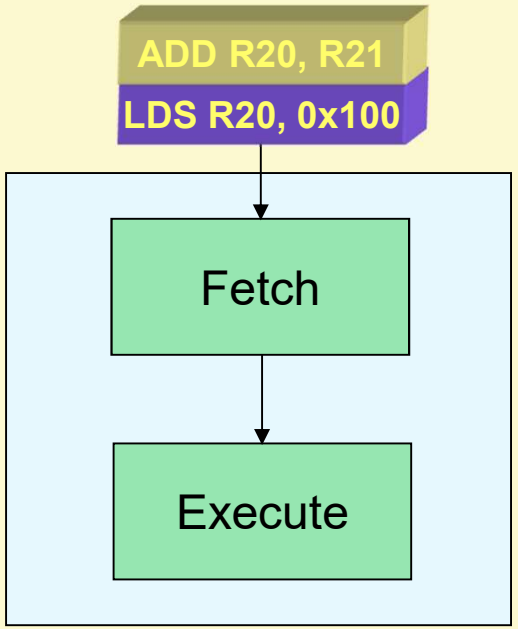


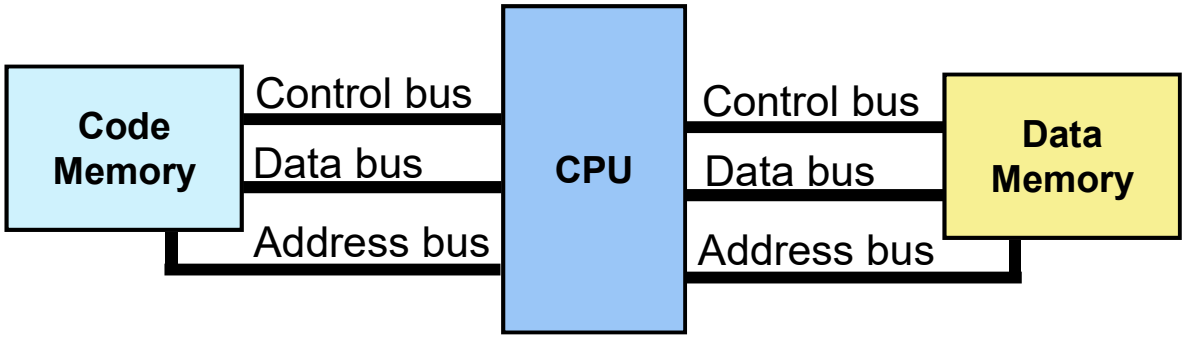
RISC architecture



RISC architecture

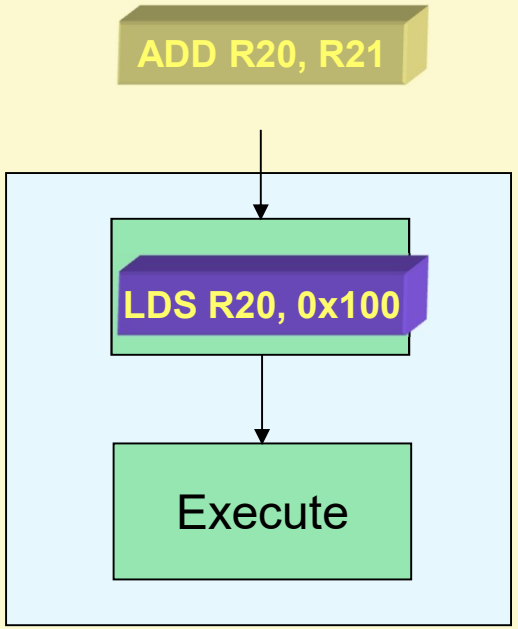
```
LDS R20, 0x100 ; R20 = [0x100]
ADD R20,R21    ; R20 = R20 + R21
```

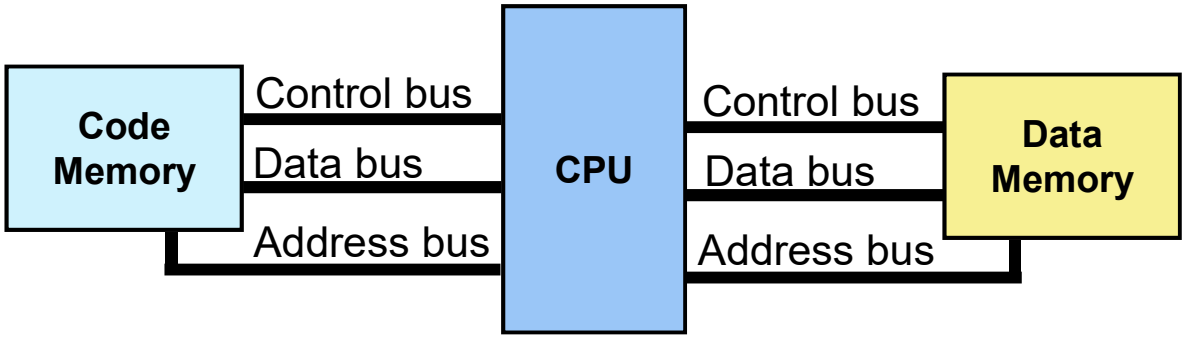




RISC architecture

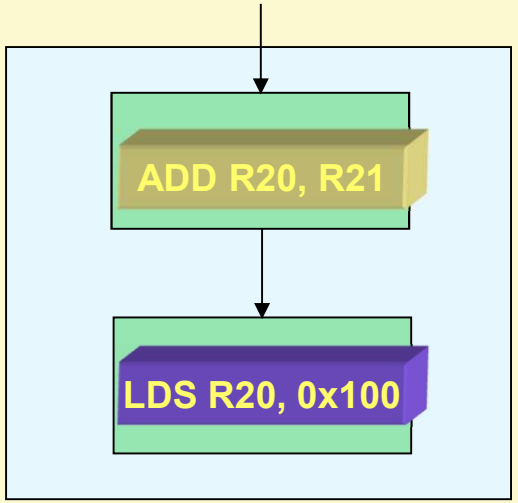
```
LDS R20, 0x100 ; R20 = [0x100]
ADD R20,R21    ; R20 = R20 + R21
```

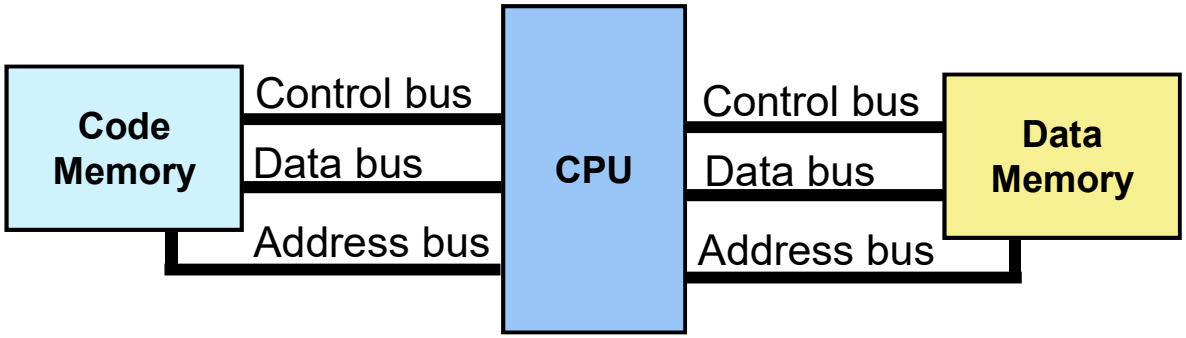




RISC architecture

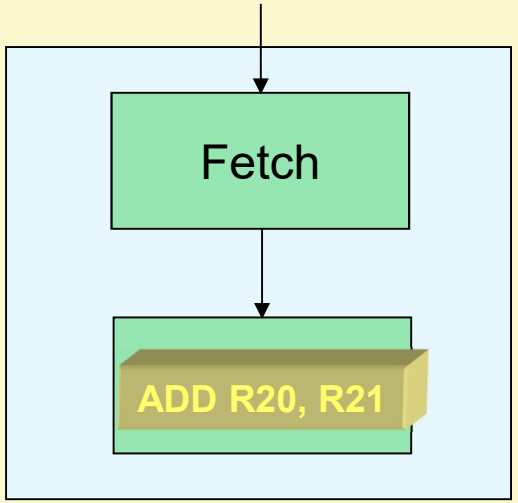
```
LDS R20, 0x100 ; R20 = [0x100]
ADD R20,R21    ; R20 = R20 + R21
```

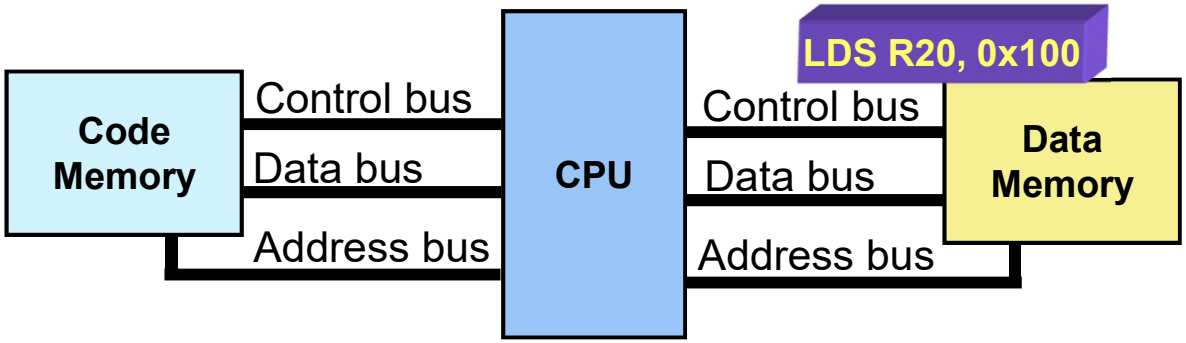




RISC architecture

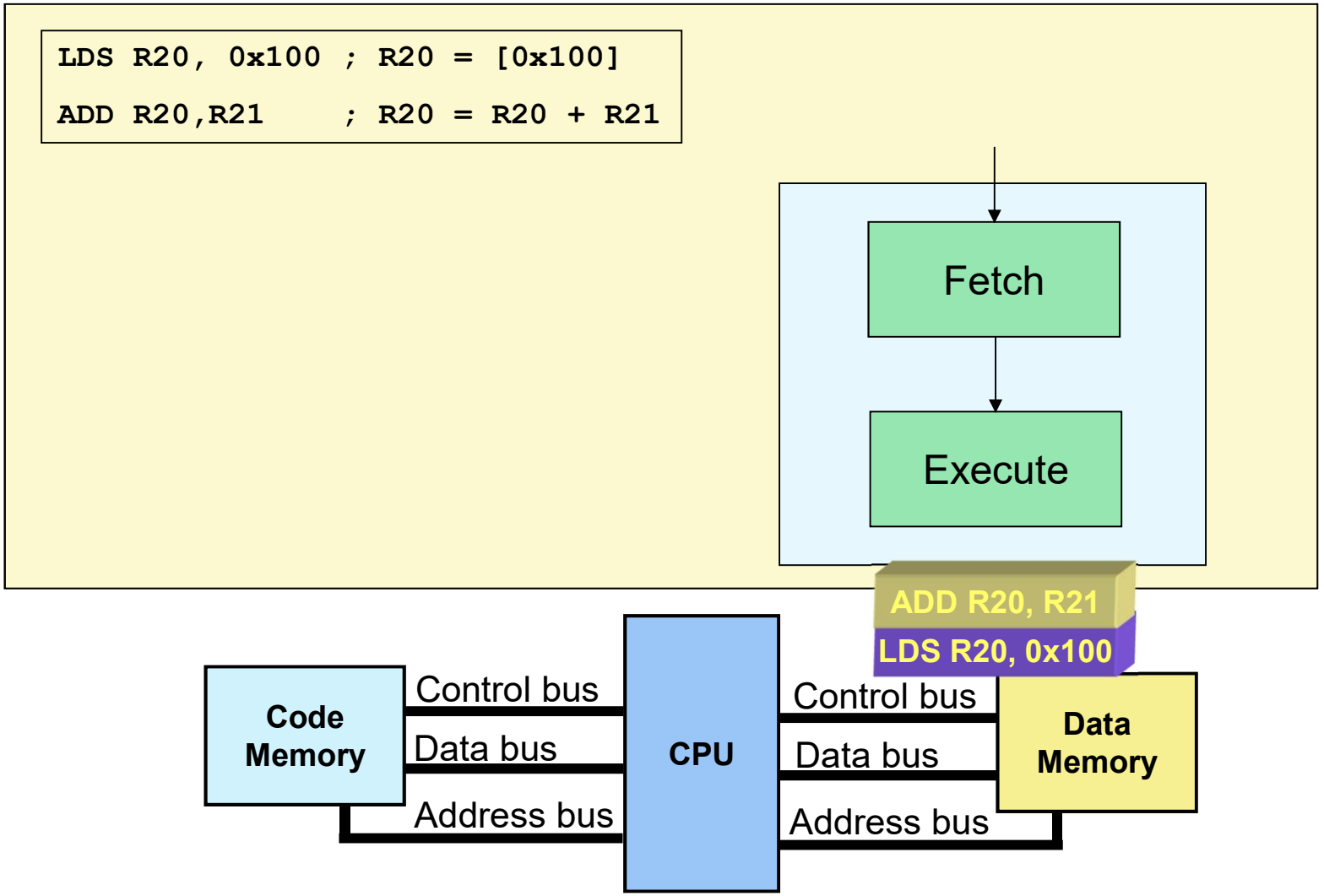
```
LDS R20, 0x100 ; R20 = [0x100]
ADD R20,R21    ; R20 = R20 + R21
```





RISC architecture

```
LDS R20, 0x100 ; R20 = [0x100]
ADD R20,R21    ; R20 = R20 + R21
```



RISC architecture

- Feature 6: more than 95% of instructions are executed in 1 machine cycle

RISC architecture

- Feature 7
  - RISC processors have at least 32 registers. Decreases the need for stack and memory usages.
    - In AVR there are 32 general purpose registers (R0 to R31)

Collaboration Learning

- How the assembly codes organize in the simple windows program?