Imperative/Procedural programming - Assembler

”first **do this**, and next, **do that**”

Overview of the imperative paradigm

- Characteristics/Discipline and idea
  - Hardware technology and the ideas of Von Neumann
  - Incremental *change of the program state* as a function of *time*.
  - Execution of computational steps in an order governed by *control structures*
- Similar to descriptions of everyday routines, such as food recipes and car repair
- Typical commands offered by imperative languages:
  - Assignment, I/O, procedure calls
- Language representatives:
  - Assembler, C, Fortran, Algol, Pascal, Basic
- The natural abstraction is the procedure
  - A procedure (or function) abstracts one or more actions to a procedure, which can be activated as a single action.
  - Also called: "Procedural programming”
- The word *statement* is often used with the special computer science meaning 'a elementary instruction in a source language'. The word *instruction* is another possibility; Devote this word the computational steps performed at the machine level. Use the word 'command' for the imperatives in a high level imperative programming language.

Computer Architecture and Programing languages

- From the beginning computers were programmed in machine code (writing the program instruction for instruction).
- One way to look at PPL (Procedural Programming Language) is that they are just an abstraction of what a program in machine code looks like
- But you need to know what machine code looks like to understand this

Memory content

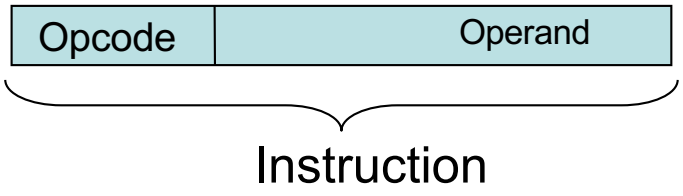
In order to understand Assembly language, it is necessary to have a good understanding of memory content:

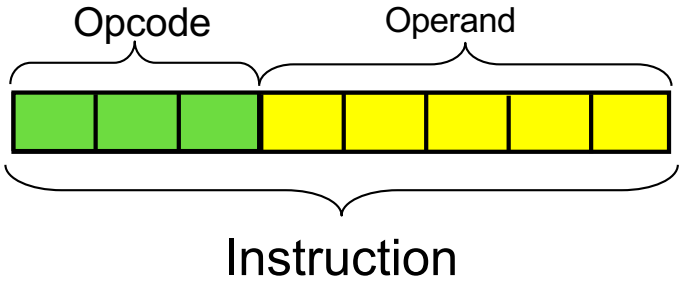
- Program memory:
  - Program (of course!)
  - Static data (e.g. constant tables)
- RAM:
  - Dynamic data (lost when power is lost)
  - Memory references (“Pointers”)
- EEPROM:
  - Data that should survive power loss

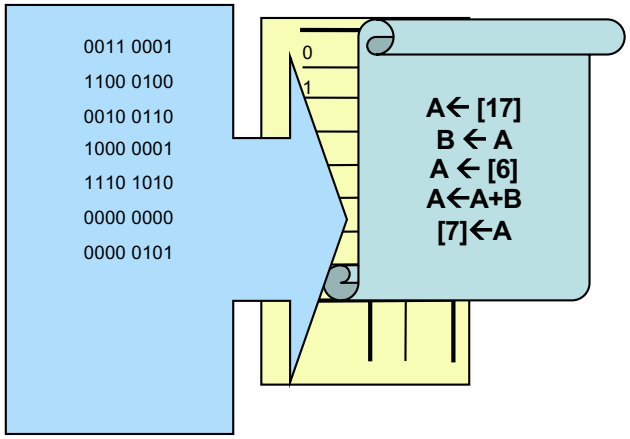
Assembler – an analogy…

- Consider how your life would be if you only had one arm!
- If you need to cut a slice of bread, you may do the following:
  - Take out the bread and place it on the counter, fixing it somehow.
  - Take out the knife and cut a slice of bread.
  - Put the slice on e.g. a plate
  - Put the knife back in the knife rack (or drawer)
  - Put the bread back…
- In assembler, we need to do in a similar fashion, almost always using Registers. E.g to do ”variable = variable + 5” , we need to do the following:
  - Copy the variable into a register
  - Put the literal 5 in another register
  - Add the two registers (placing the result in one of the registers)
  - Copy the result of the addition back into the variable.

How Instruction decoder works





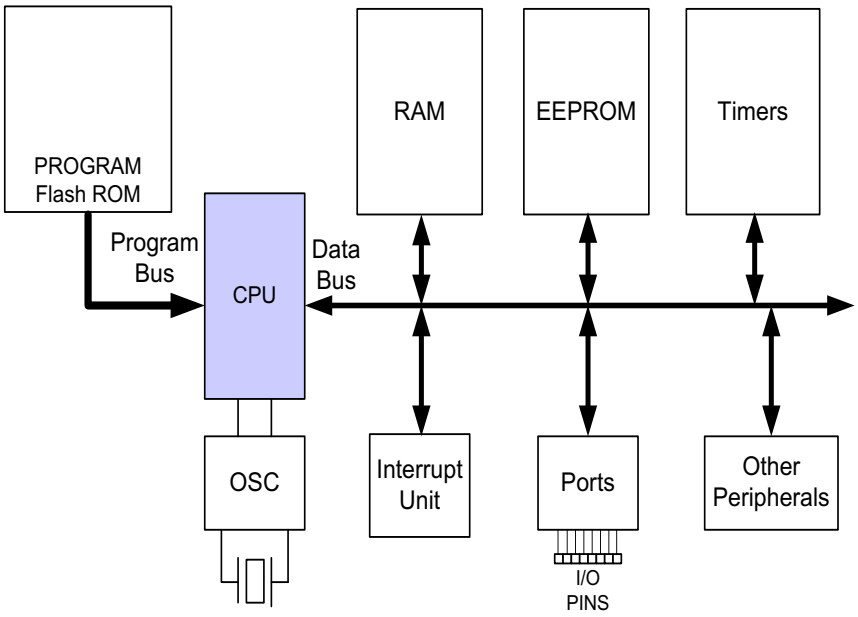


Operation Code	Meaning
000	A ← x
001	A ← [x]
010	A ← A – register (x)
011	A ← A + x
100	A ← A + register (x)
101	A ← A – x
110	Register (x_H) ← Register (x_L)
111	[x] ← A

Operand is the “parameters” to the operation. Some ops have one parameter, e.g. A or x, some ops have several parameters

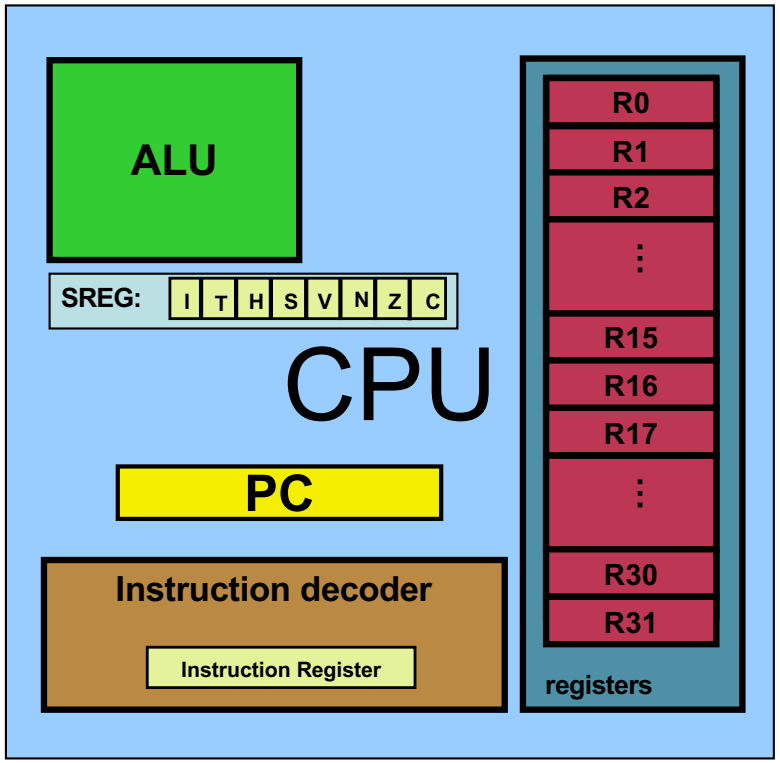
Topics

- 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)
  - PC register
  - Instruction decoder



Introduction to assembler instructions

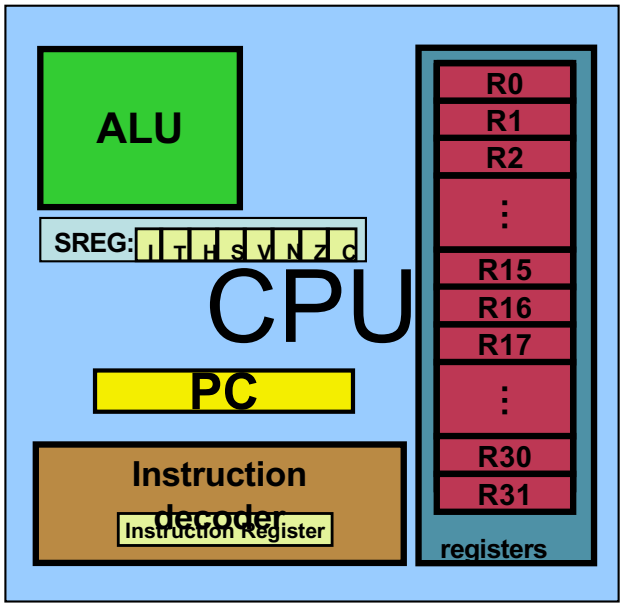
- Instructions using One register:
  - LDI
  - INC
  - DEC
- Instructions using Two registers:
  - ADD
  - SUB

Some simple instructions

1. Loading values into the general purpose registers

LDI (**Load Immediate**)

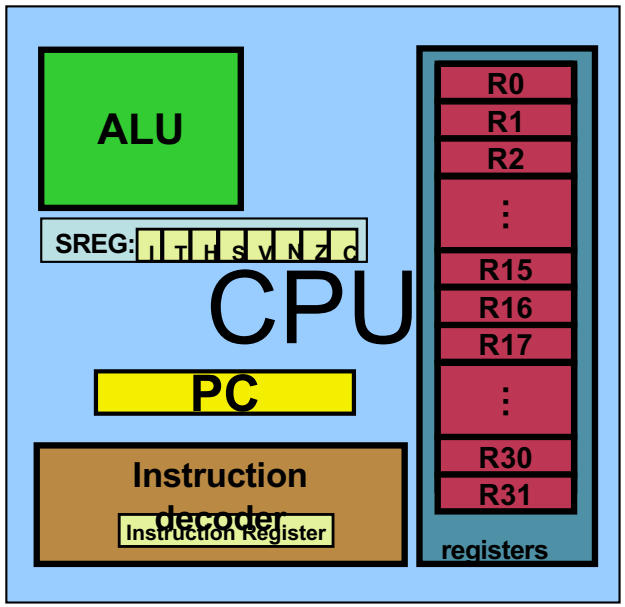
- LDI Rd, k ; Rd is destination register
  ; k is literal (value)
  - Its equivalent in high level languages: Rd = k
- Example:
  - LDI R16,53
    - R16 = 53
  - LDI R19,132
  - LDI R23,0x27
    - R23 = 0x27



Some simple instructions

2. Arithmetic calculation

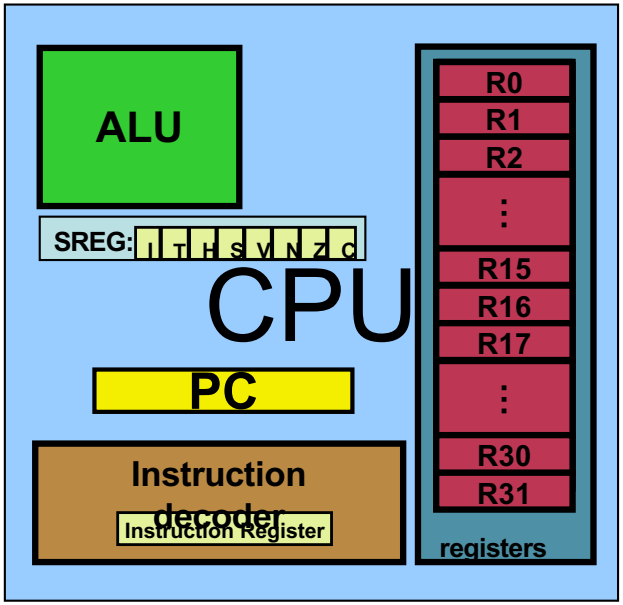
- There are some instructions for doing Arithmetic and logic operations; such as: ADD, SUB, AND, OR, etc.
- ADD Rd,Rs ; Rd is destination Register
  - Rd = Rd + Rs ; Rs is source Register
- Example:
  - ADD R25, R9
    - R25 = R25 + R9
  - ADD R17,R30
    - R17 = R17 + R30



A simple program

- Write a program that calculates 19 + 95

```
LDI R16, 19     ;R16 = 19 (decimal)
LDI R20, 95     ;R20 = 95
ADD R16, R20    ;R16 = R16 + R20
```



A simple program

- Write a program that calculates 19 + 95 + 5

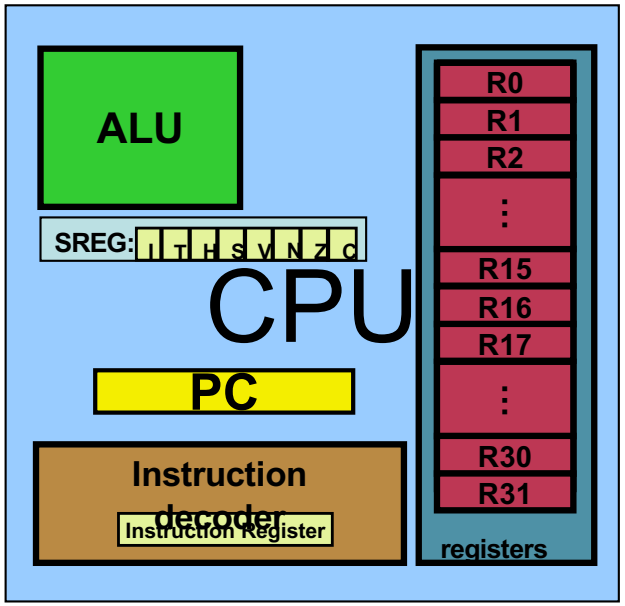
```
LDI   R16, 19          ;R16 = 19
LDI   R20, 95          ;R20 = 95
LDI   R21, 5           ;R21 = 5
ADD   R16, R20         ;R16 = R16 + R20
ADD   R16, R21         ;R16 = R16 + R21
```

```
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

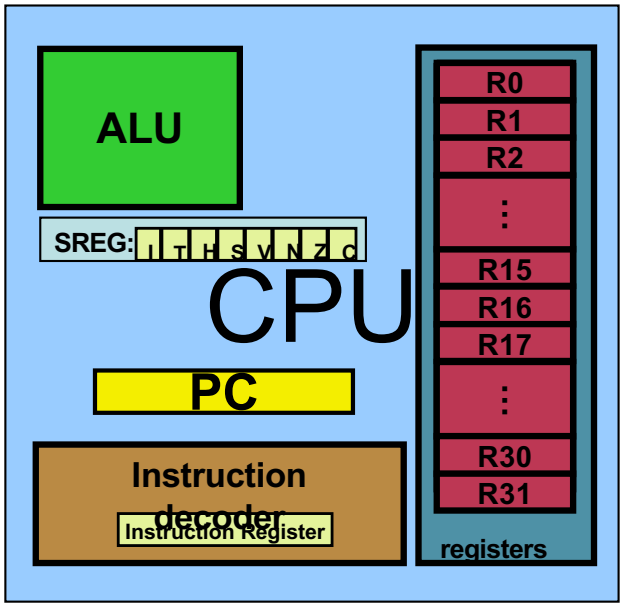


Some simple instructions

2. ADD/SUB immediate…

- It would be useful to have an instruction:
  ADDI Rd, k ; Add k to content of Rd
- Unfortunately, there IS NO such instruction!
- BUT, there IS: SUBI, that can be used….
- Example:
  - LDI R16, 10 ; R16 = 10
  - SUBI R16, 5 ; R16 = 5
  - SUBI R16, -5 ; R16 = 10

  - So, Rd – (-k) = Rd+k



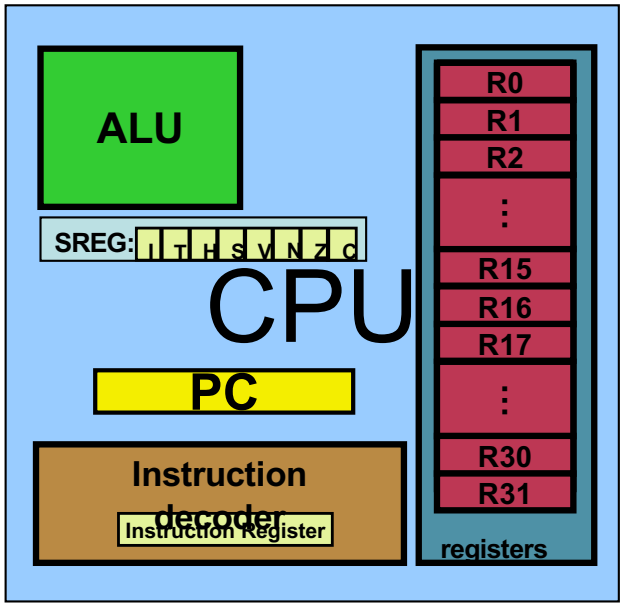
- ANDI, ORI, etc… exist (we come back to these)

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



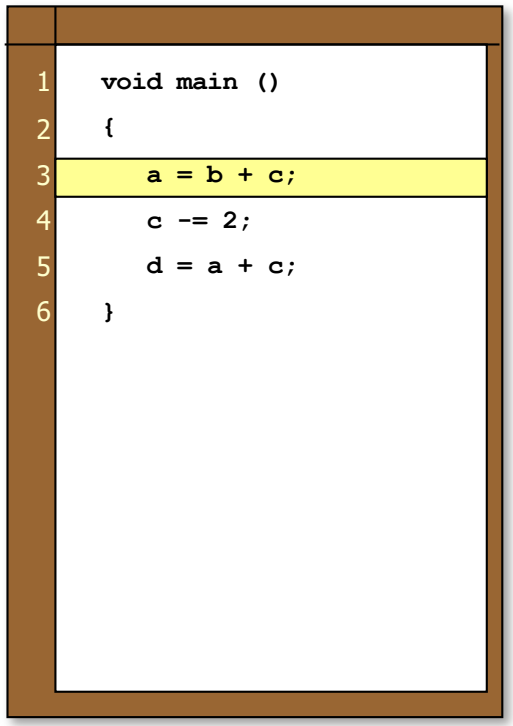
Introduction to jump and call

- Jump
  - One-way transition! (no automatic way back…)
- Call
  - Two-way transition, you can come back…
  - Stack is introduced
  - Calling a subroutine/function
  - Returning from a subroutine

Jump and Call

- CPU executes instructions one after another.

  For example in the following C program, CPU first executes the instruction of line 3 (adds b and c), then executes the instruction of line 4.



Animation: slide 7

Jump and Call (Continued)

- But sometimes we need the CPU to execute an instruction other than the next instruction. For example:
  - When we use a conditional instruction (if-statement) We will come back to this case…
  - When we make a loop
  - When we call a function

Jump and Call (Continued)

- Example 1: Not executing the next instruction, because of condition.

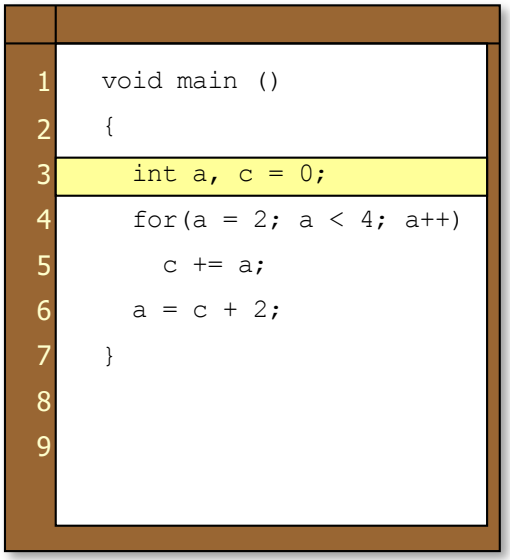
  In the following example, the instruction of line 6 is not executed.

```
1  void main ()
2  {
3      int a = 2;
4      int c = 3;
5      if (a == 8)
6          c = 6;
7      else
8          c = 7;
9      c = a + 3;
   }
```

- Animation: slide 8

Jump and Call (Continued)

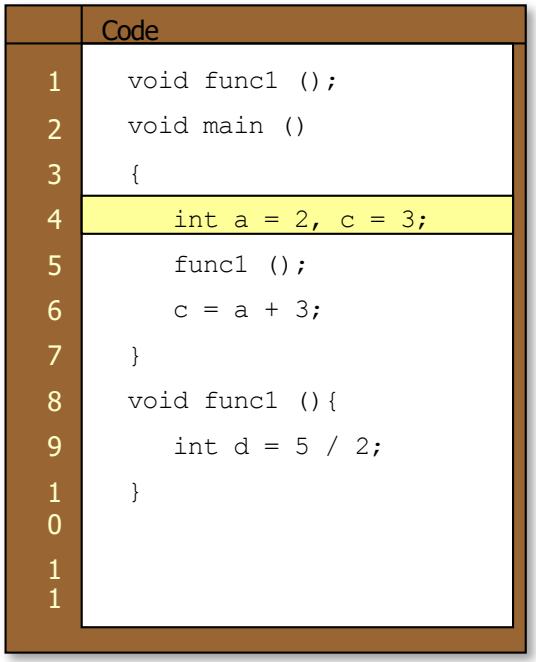
- Example 2: In this example the next instruction will not be executed because of a loop.
  - In the following example, the order of execution is as follows:
    - Line 4
    - Line 5
    - Again, line 4
    - Again line 5
    - Line 6



Animation: slide 9

Jump and Call (Continued)

- Example 3: Not executing the next instruction, because of calling a function.
  - In the following example, the instruction of line 6 is not executed after line 5.



	Code
1	`void func1 ();`
2	`void main ()`
3	`{`
4	`int a = 2, c = 3;`
5	`func1 ();`
6	`c = a + 3;`
7	`}`
8	`void func1 (){`
9	`int d = 5 / 2;`
10	`}`
11	

- Animation: slide 10

Jump and Call (Continued)

- In the assembly language, there are 2 groups of instructions that make the CPU execute an instruction other than the next instruction. The instructions are:
  - Jump: used for making loop and condition
  - Call: used for making function calls

Jump

- A Jump changes the Program Counter (PC) and causes the CPU to execute an instruction other than the next instruction.

- Conceptually, a jump can be thought of as:
  - LDI PC, k ; k is the destination address
    ; of the jump

Jump

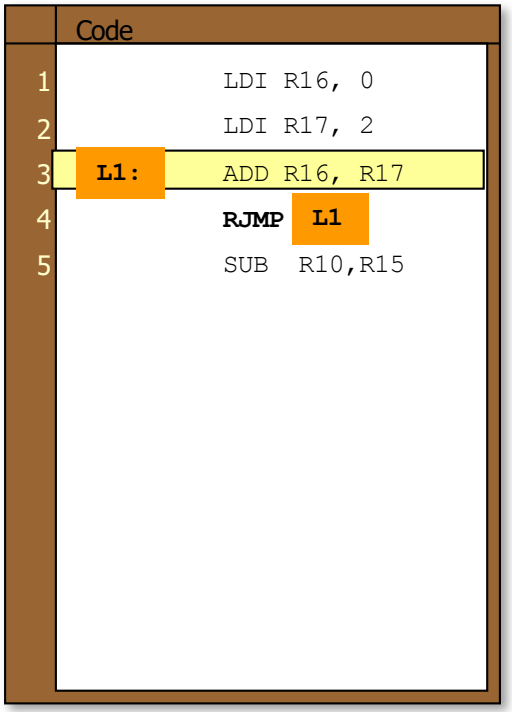
There are 2 kinds of Jump:

- Unconditional Jump:
  - When CPU executes an unconditional jump, it jumps unconditionally (without checking any condition) to the target location.
  - Example: RJMP and JMP instructions

- Conditional Jump:
  - When CPU executes a conditional jump, it checks a condition, if the condition is true then it jumps to the target location; otherwise, it executes the next instruction.
  - Example: BRxx (Branch…) and Sxxx (Skip…) instructions

Unconditional Jump in AVR

- There are 3 unconditional jump instructions in AVR: **RJMP**, **JMP**, and **IJMP**
- We label the location where we want to jump, using a unique name, followed by ':'
- Then, in front of the jump instruction we mention the name of the label.
- This causes the CPU to jump to the location we have labeled, instead of executing the next instruction.
- Animation: slide 11



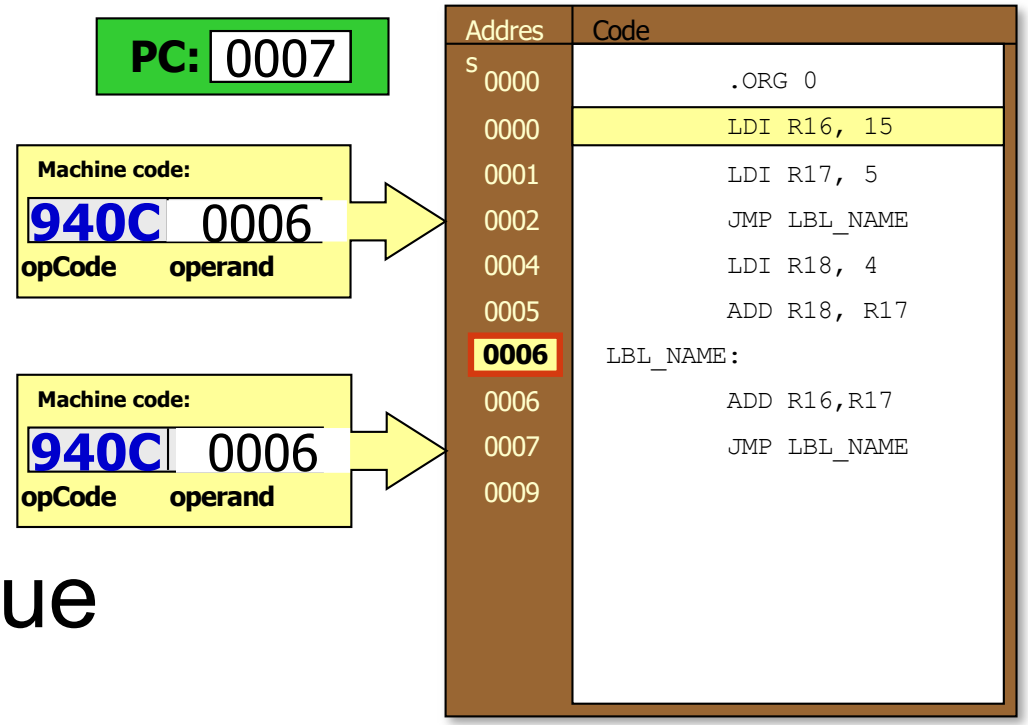
Ways of specifying the jump target

- There are 2 (3) ways to provide the jump address:
  1. PC = operand (“absolute”)
  2. PC = PC + operand (“relative”)

  3. (PC = Z register) MK: seldom used…

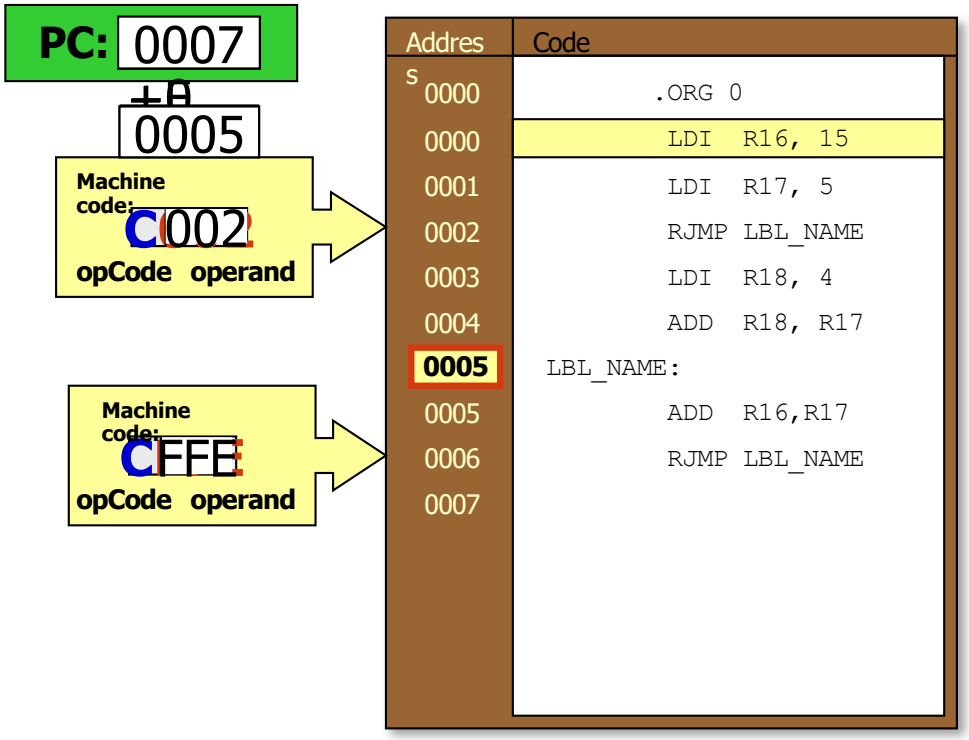
JMP

- In JMP, the operand, contains the address of the destination
- When an JMP is executed:
  - PC is loaded with the operand value
  - Animation: slide 12



RJMP (Relative Jump)

- When RJMP is executed:
  - The operand will be added to the current value of PC



  - Animation: slide 13

Call Topics:

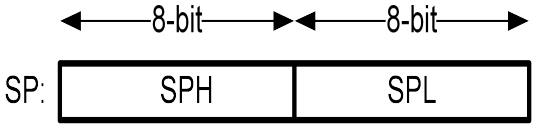
- The concept of the Stack
- Instructions:
  - PUSH and
  - POP
- Calling a function (CALL, RCALL)
- Returning from a function (RET)

Stack

- PUSH *Rr*

```
[SP] = Rr
SP = SP - 1
```

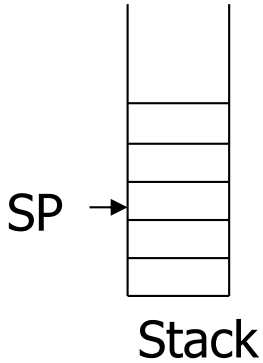
The content of the memory cell referenced by the SP is given the value of Rr



- POP *Rd*

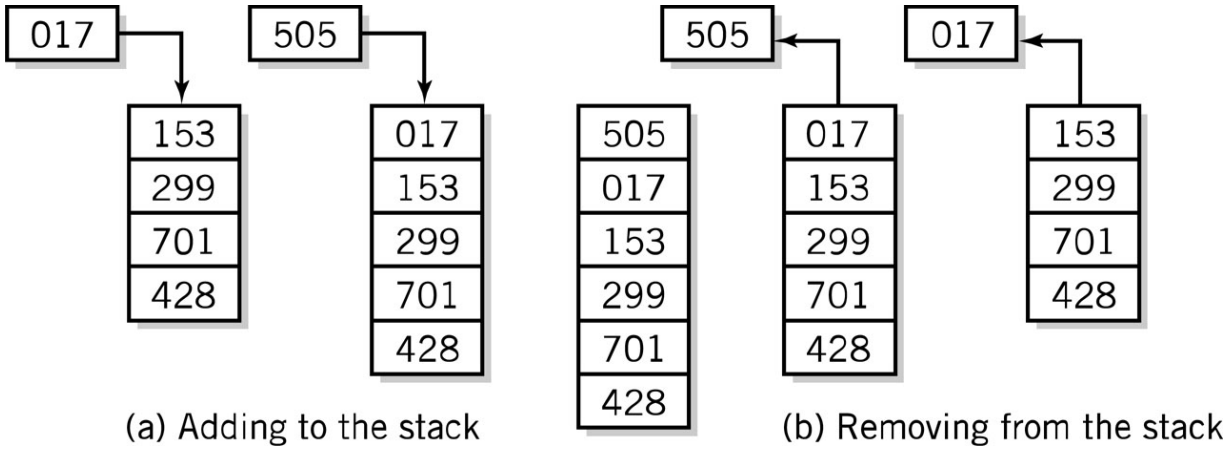
```
SP = SP + 1
Rd = [SP]
```

Rd is given the value of the content of the memory cell referenced by SP

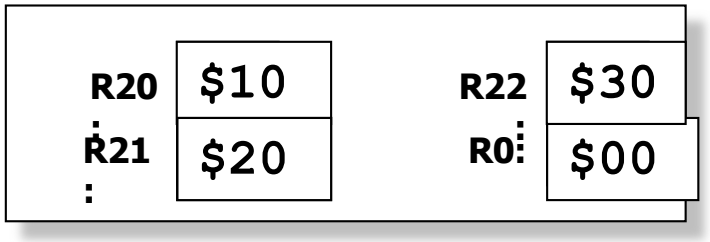


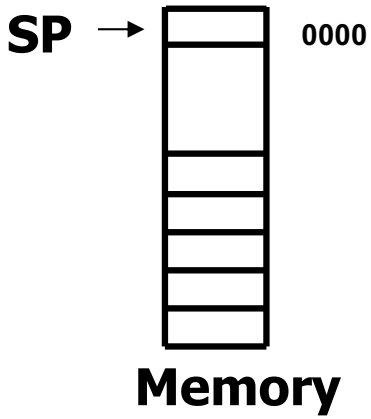
Stack Instructions

- Stack instructions
  - LIFO method for organizing information
  - Items removed in the reverse order from that in which they are added



Stack



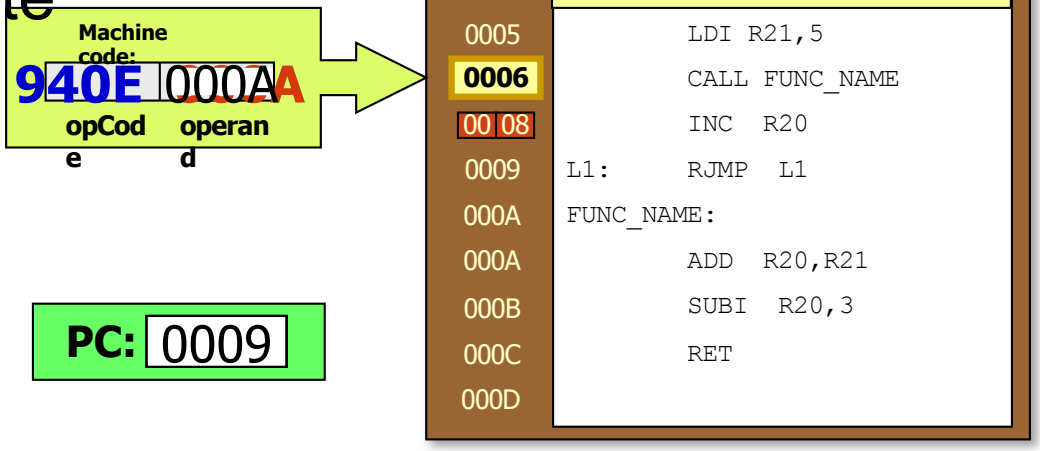


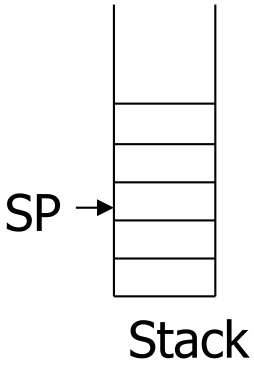
Address	Code
	ORG 0
0000	LDI R16,HIGH(RAMEND)
0001	OUT SPH,R16
0002	LDI R16,LOW(RAMEND)
0003	OUT SPL,R16
0004	LDI R20,0x10
0005	LDI R21, 0x20
0006	LDI R22,0x30
0007	PUSH $10
0008	PUSH $20
0009	PUSH $30
000A	POP R21
000B	POP R0
000C	POP R20
000D	L1: RJMP L1

Animation: slide 16

Calling a Function

- To execute a call:
  - Address of the next instruction is saved
  - PC is loaded with the appropriate value

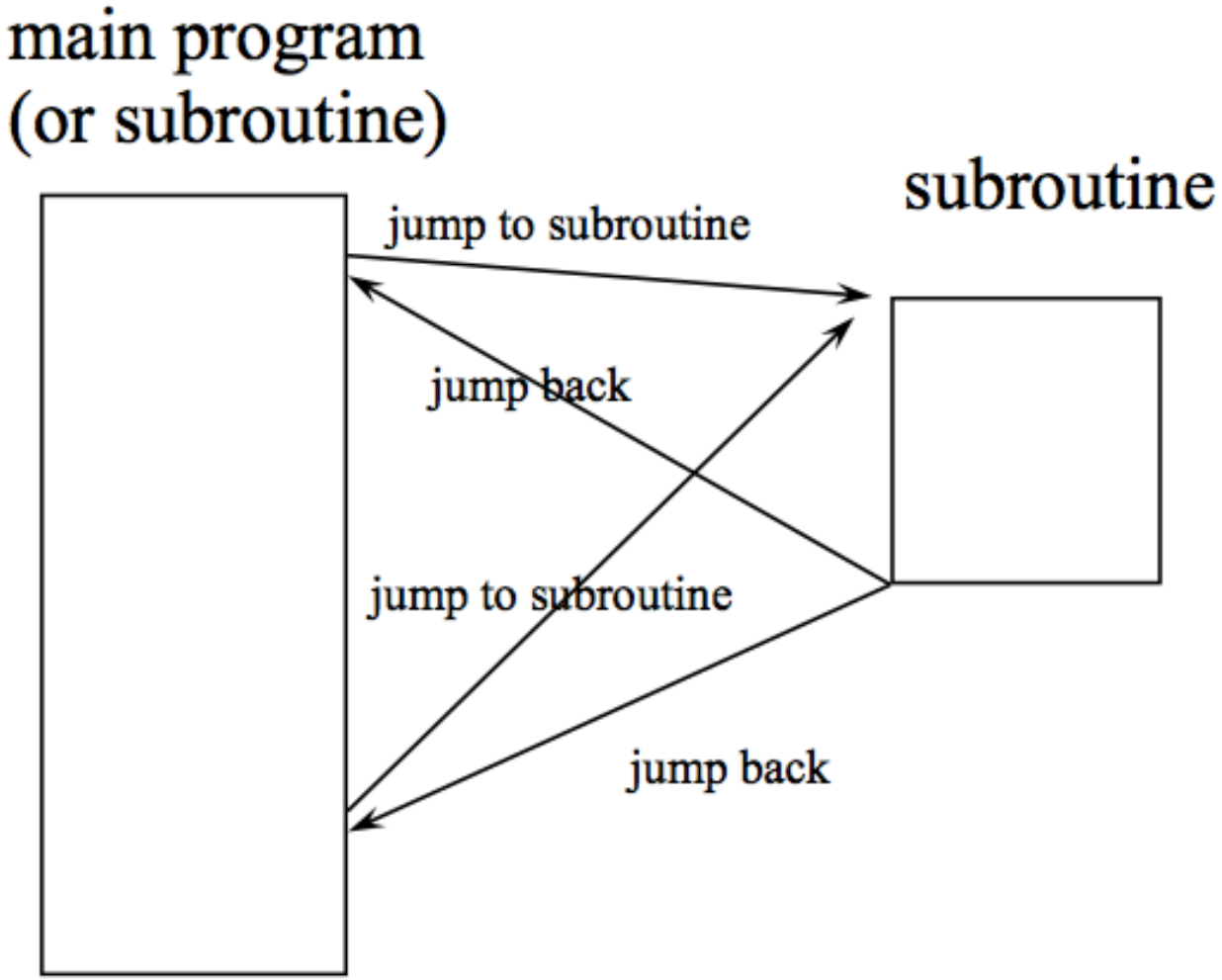




PC: 0009

Animation: slide 17

Subroutine Call – may call from several places



JMP vs. RJMP and CALL vs. RCALL

- When using JMP, the operand is a 16 bit address of the destination
- When using CALL, the same applies

Now…

- RJMP and RCALL used a displacement, a relative number, from current position, instead of a fixed destination.
- This takes less space (operation is 1 word instead of 2), but there are limitations…. (se manual)
- Recommendation: Use Rxxx until Assembler tells you it does not work!

Registers..

- Working registers (R0 – R31, 8 bits)
- Program Counter PC (16 bit)
  - Increased as the program is executed
  - Can be set by the program = Jump!
- Stack Pointer SP (16 bit)
  - Stack is used by CALL/RCALL to save PC (next position in program) when jumping to sub-routine
  - Stack can be used in program (PUSH, POP)
    - Note: The stack pointer counts downwards, when something is added to stack
    - We will come back to how the SP is set

Note: R0 thru R15, the MOV-instruction

Only registers in the range R16 to R31 can be loaded immediate. We cannot load a constant into the registers R0 to R15 directly. It would have to be loaded into a valid register first then copied. To load the value of 10 into register zero (R0):

```
LDI R16,10      ; Set R16 to value of 10
MOV R0, R16     ; Copy contents of R16 to R0
```

Instructions covered so far….

- LDI
- ADD/SUB/SUBI
- INC/DEC
- JMP/RJMP
- PUSH/POP
- CALL/RCALL/RET
- MOV

Assembler directives

```
.BYTE   ; reservere space for one Byte in RAM
.CSEG   ; the next instructions relates to the program memory
.DB     ; define a BYTE constant
.DEF    ; define a symbolic name for eg a register
.DSEG   ; the next instructions relates to the RAM
.DW     ; define a WORD (16 bits) constant
.ENDM   ; end of a MACRO
.EQU    ; declare a symbolic constant or expression (cannot be redef.)
.MACRO; defines the start of a MACRO
.ORG    ; defines the address of the next instructions
.SET    ; defines a symbolic definition (can be redefined later)
```

All directives starts with a ”.”

Assembler Directives: .EQU and .DEF

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

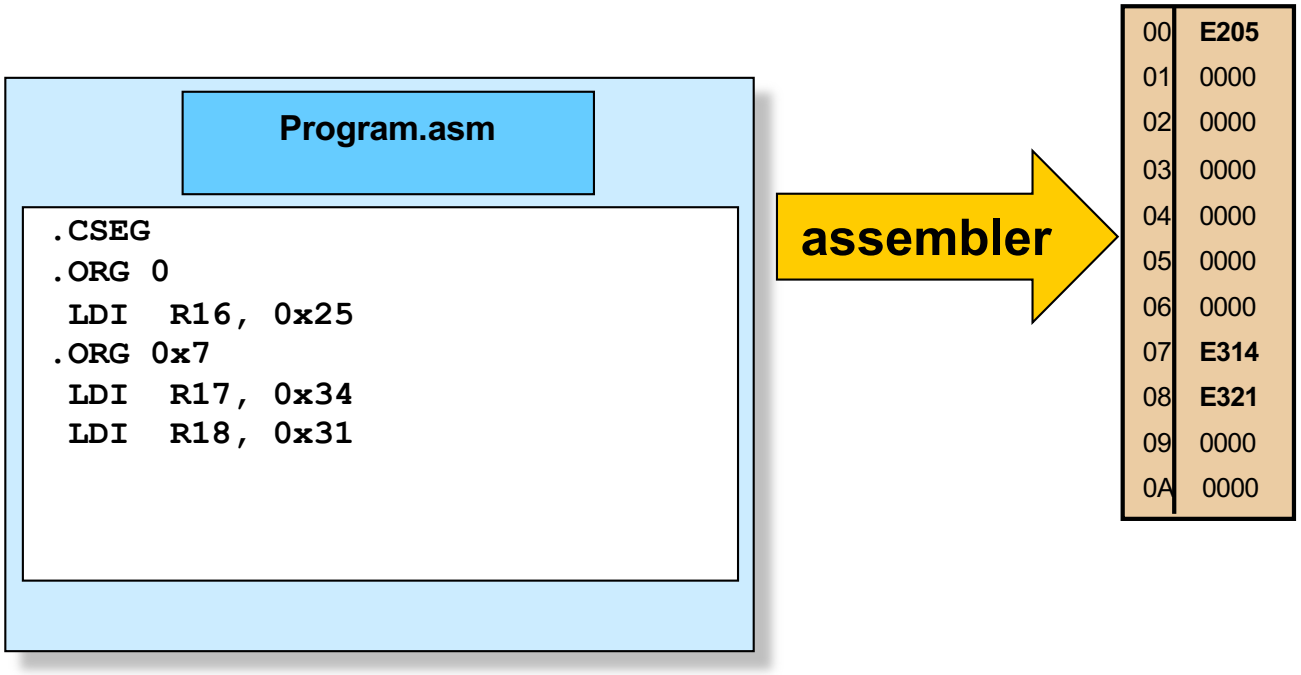
```
.EQU   COUNT = 0x25
LDI    R21, COUNT                ;R21 = 0x25
LDI    R22, COUNT + 3            ;R22 = 0x28
```

- .DEF name = Register
  - *Example:*

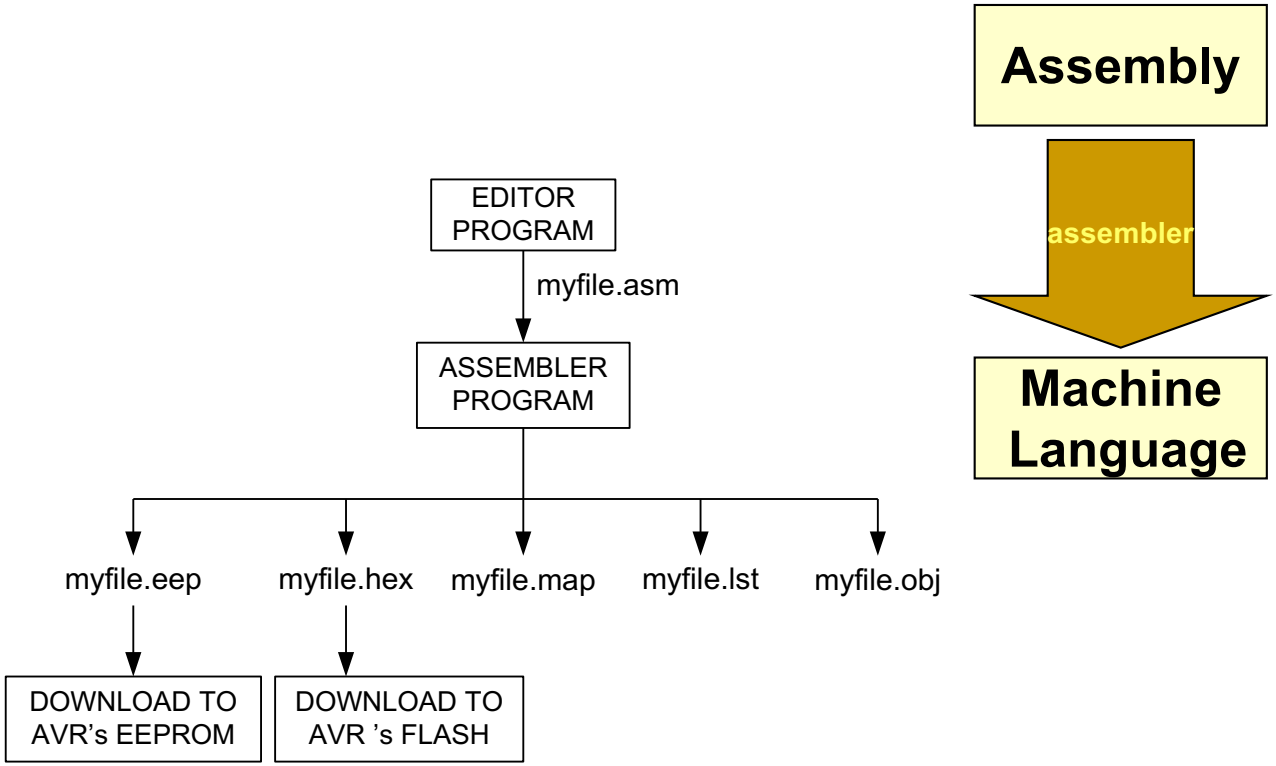
```
.DEF   COUNTER = R16
LDI    COUNTER, COUNT            ;R16 = 0x25
```

Assembler Directives: .ORG

- .ORG *address*
  - *Examples:*

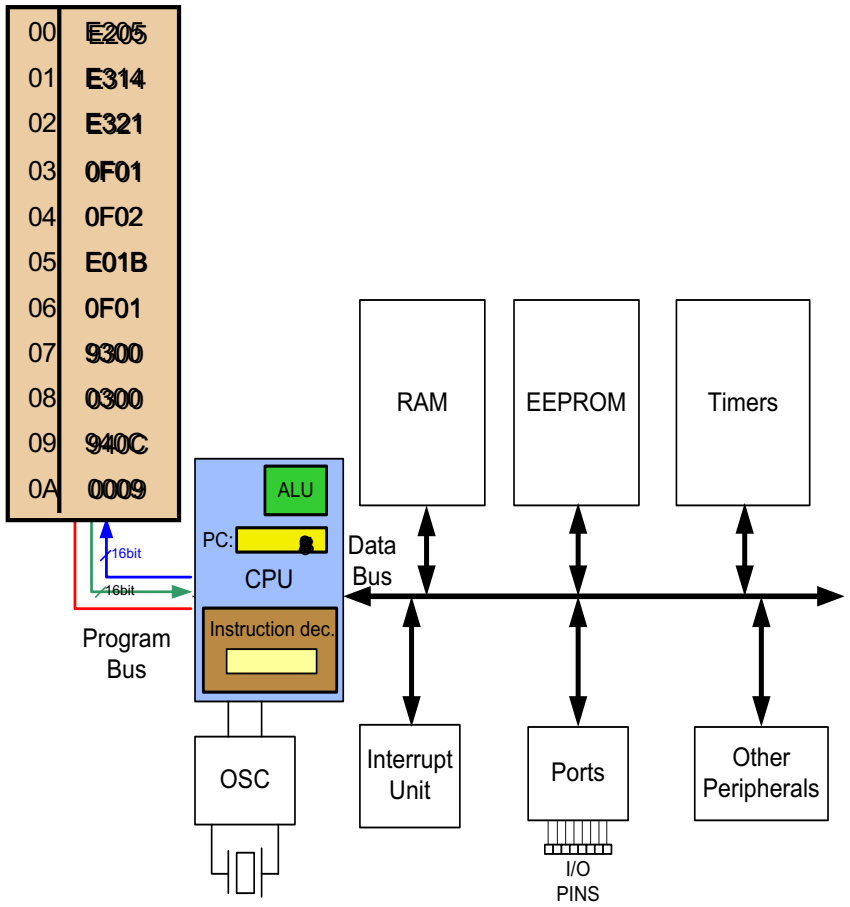


The “Assembler” program



Flash memory and Machine Code

```
      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
```



Most Assembly instructions correspond to One machine instruction (if separate address – two machine instructions)

Instruction execution

- We can think of a microprocessor's execution of instructions as consisting of several basic stages:
  - Fetch instruction: the task of reading the next instruction from memory into the instruction register
  - Decode instruction: the task of determining what operation the instruction in the instruction register represents (e.g., add, move, etc.)
  - Fetch operands: the task of moving the instruction's operand data into appropriate registers
  - Execute operation: the task of feeding the appropriate registers through the ALU and back into an appropriate register
  - Store results: the task of writing a register into memory. If each stage takes one clock cycle, then we can see that a single instruction may take several cycles to complete.

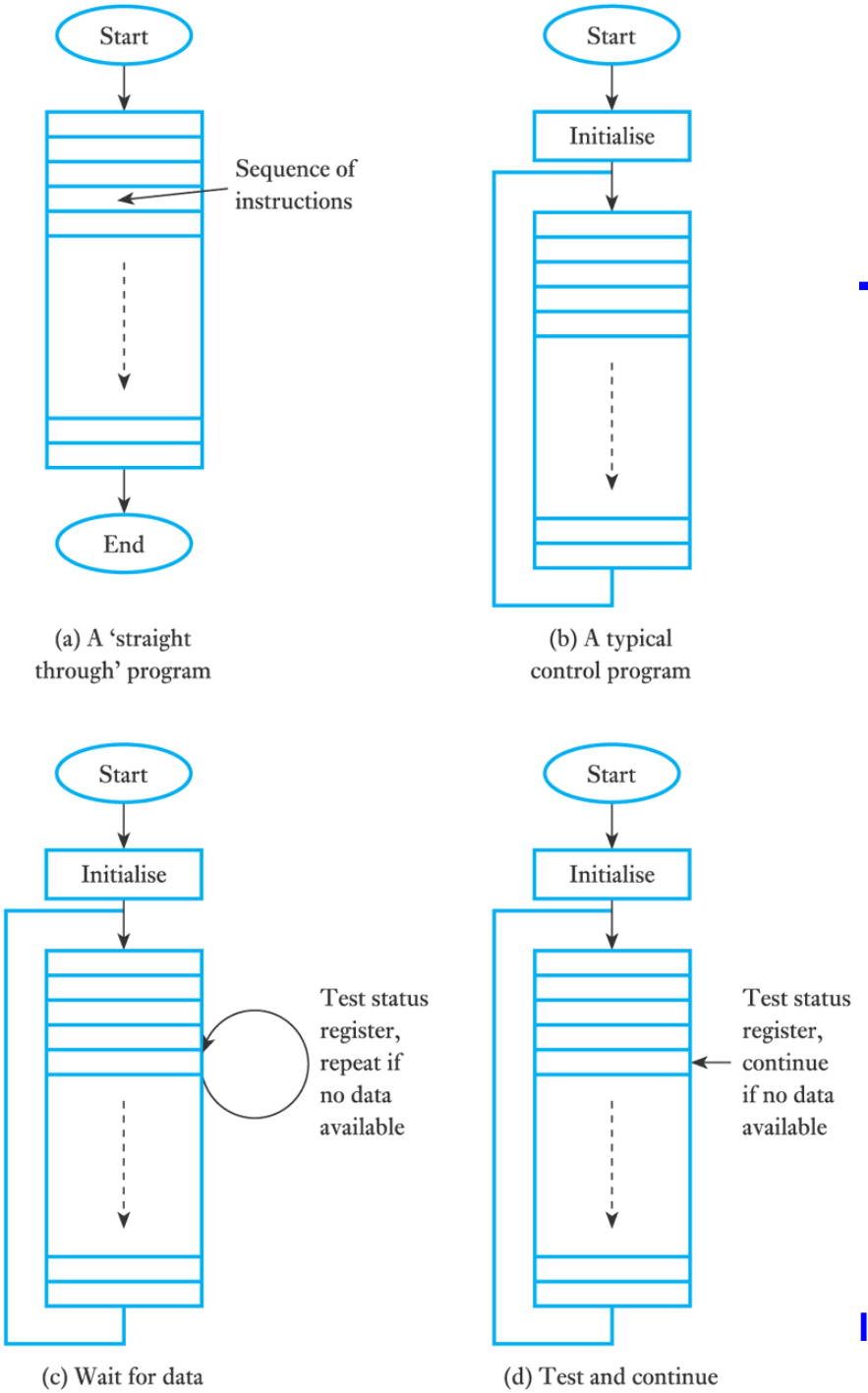
Animation: slide 5-6 (pipelining)

Start of program in Arduino Leonardo…

```
;=============================================================
; Definitions of registers, etc. ("constants")
;=============================================================
            .EQU RESET      = 0x0000
            .EQU PM_START   = 0x0056
;=============================================================
; Start of program
;=============================================================
            .CSEG           ; the next instruction relates to program memory
            .ORG RESET      ; define the address of the text instruction
            RJMP init       ; instruction at address ”RESET”
            .ORG PM_START   ; define the address of the text instruction
;=============================================================
; Program start …
;=============================================================
init:       ….with some instruction… which is stored at address 0x0056
```

Programmed controlled input/output

- polling of I/O devices



Data in RAM

Data is often stored in Registers, but if much data is handled, there are not enough Registers (32 in AVR…). We can also store data in RAM!

– Example:

```
        .DSEG                   ; RAM
        .ORG        0x100       ; first free cells…
Variable_A:     .BYTE 1         ; uninitialised space
Variable_B:     .BYTE 1         ;
        .CSEG                   ; Flash
        <instr…>
```

Writing Immediate Value to RAM

- You cannot copy immediate value directly into RAM location.
- This must be done via Registers
  - Example: The following program writes 0x55 to locations in RAM (from last slide…)

```
LDI     R20, 0x55           ;
STS     Variable_A, R20     ; STS - Store Direct to data Space
LDS     R20, Variable_A     ; LDS – Load Direct from Data Space
STS     Variable_B, R20     ; store
```