LUND
UNIVERSITY

Lab3: Machine Language and Assembly Programming

Goal

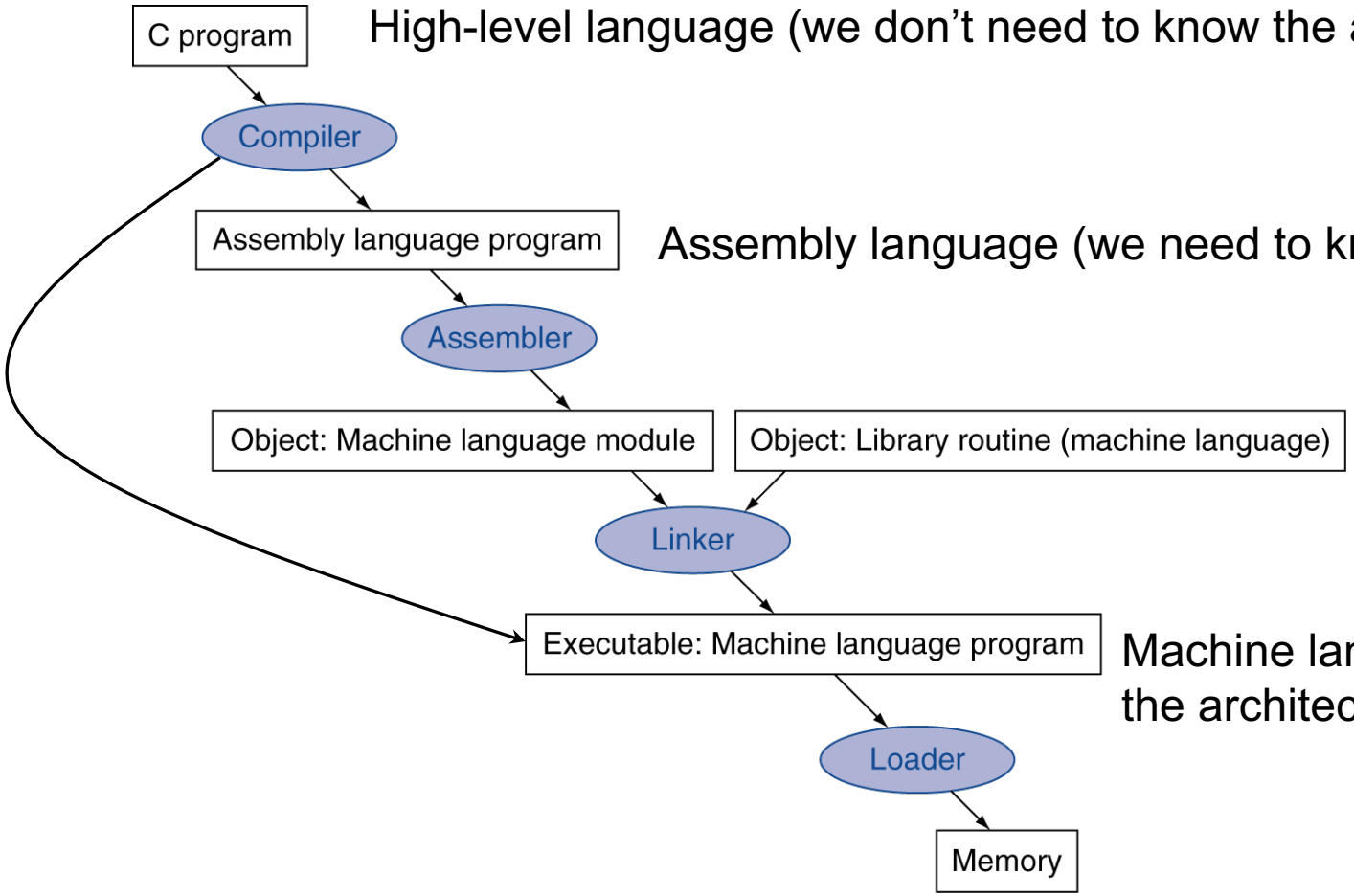
- Learn how instructions are executed
- Learn how registers are used
- Write subroutines in assembly language
- Learn how to pass and return arguments from subroutines
- Learn how the stack is used

Programmers vs. computers

- Programmers can write programs in a high-level programming language, or assembly language

- Computers can only execute programs written in their own native language (machine code)

Programming



High-level language (we don't need to know the architecture)

Assembly language (we need to know the architecture)

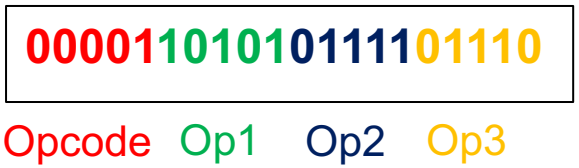
Machine language (native to the architecture)

Example

- `a=b+c;`
  1. Load variable `b` from memory into `register1`
  2. Load variable `c` from memory into `register2`
  3. Perform the addition `register1+register2` and store the result in `register3`
  4. Store `register3` to the memory address of variable `a`
- Each step translates into one machine instruction

Machine Language

- Processor can only execute machine instructions
- The instructions reside in memory along with data
- Machine instruction is a sequence of bits



- There is a set of machine instructions that are supported by a given computer architecture (Instruction Set)

Maskininstruktioner

- Definitioner:
  - Vad ska göras (operationskod)?
  - Vem är inblandad (source operander)?
  - Vart ska resultatet (destination operand)?
  - Hur fortsätta efter instruktionen?

Maskininstruktioner

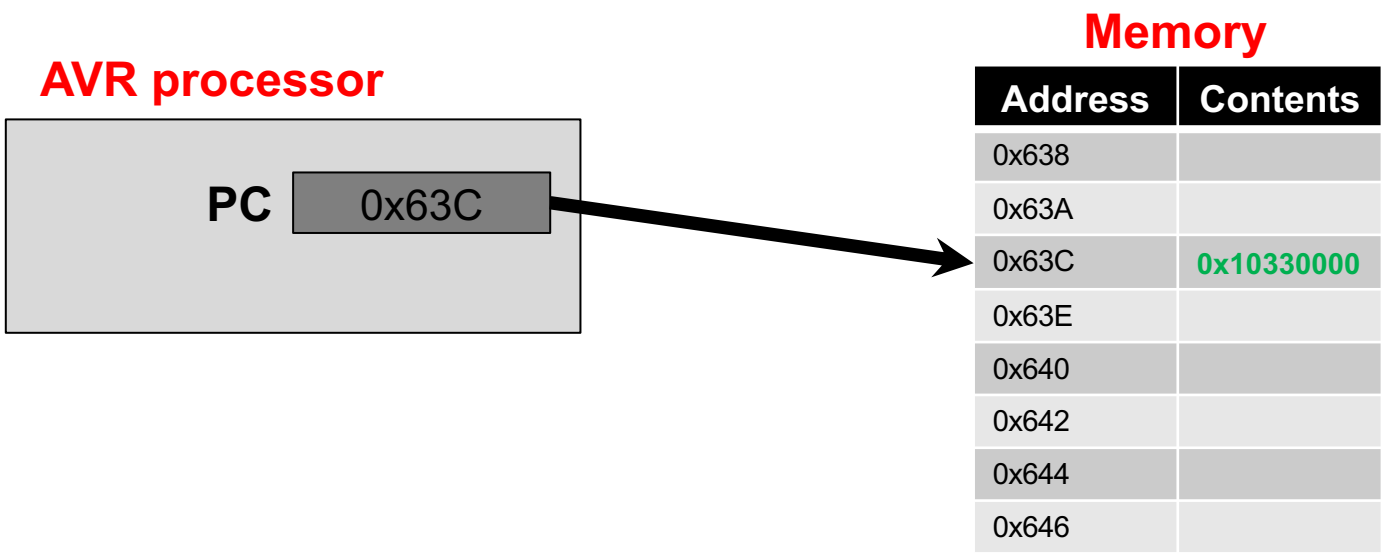
- Att bestämma:
  - Typ av operander och operationer
  - Antal adresser och adresseringsformat
  - Registeraccess
  - Instruktionsformat
    - Fixed eller flexibelt

Inside the AVR processor

- Thirty two 8-bit general purpose registers, `r0-r31`
- A set of special purpose registers
  - **PC,** Program Counter
    - keeps the address of the instruction being executed
  - **SP,** Stack Pointer
    - points to current top of stack
  - **SREG,** 8-bit Status Register
    - contains control and status bits for the processor
      - Carry
      - Overflow
      - Interrupt
      - …

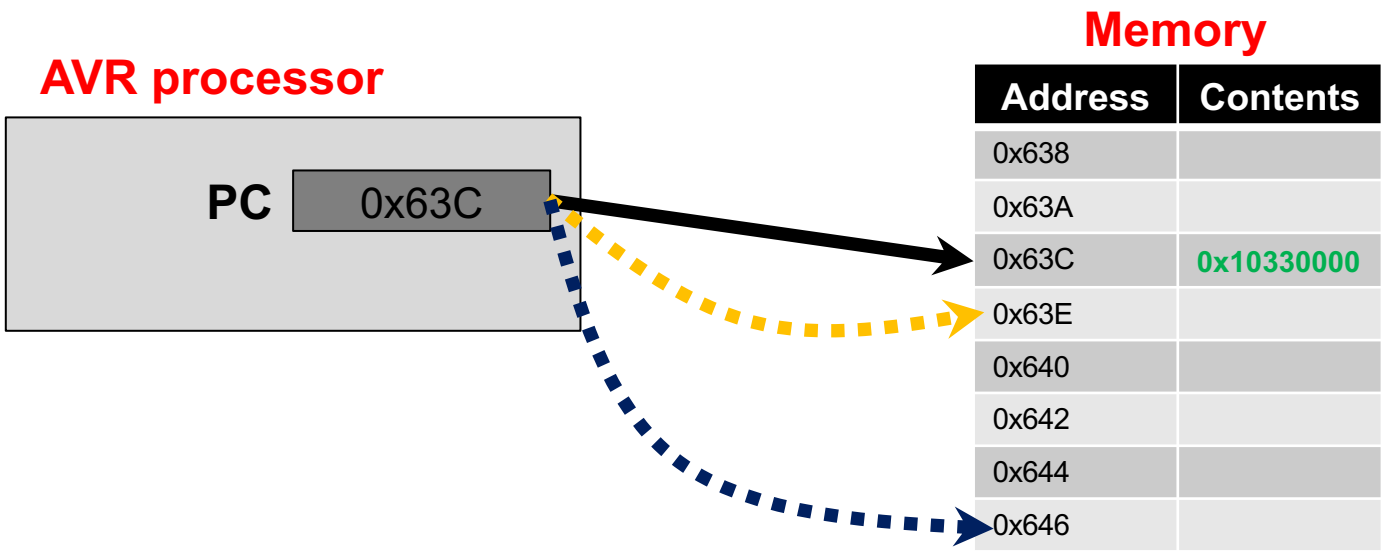
Program Counter (PC)

- Contains the memory address of the instruction that is to be fetched and executed by the processor
- After the execution of the current instruction, this register is updated to point to the memory address of the next instruction that should be fetched and executed

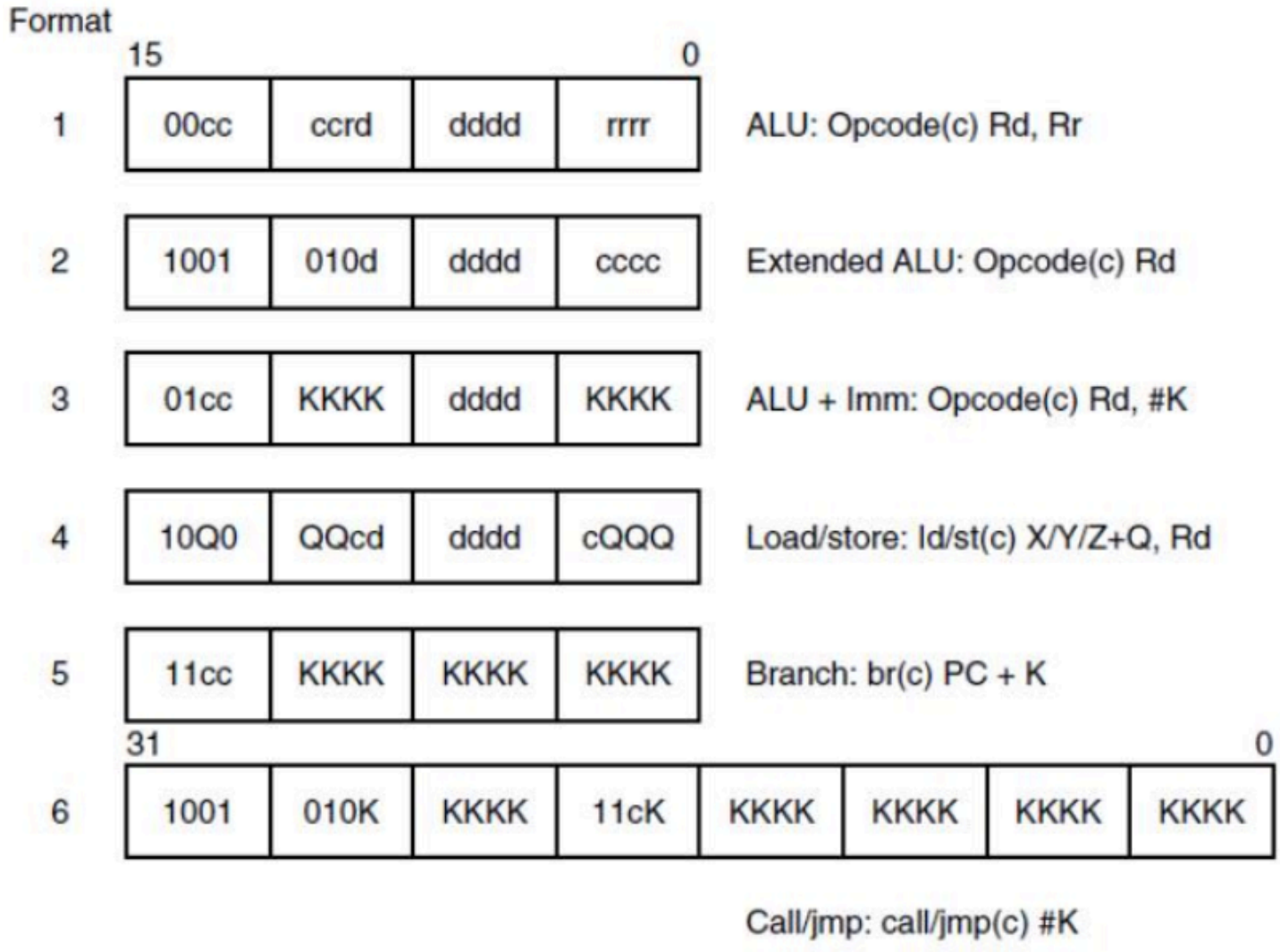


Program Counter (PC)

- If the current instruction does not explicitly modify **PC,** after execution, the **PC** is updated to point to the successive memory address (the yellow arrow)
- If the current instruction explicitly modifies **PC,** after execution, the **PC** points to the new value assigned to it (blue arrow)



AVR machine instructions



ALU instruction- Example

- Logical AND
  - Syntax:

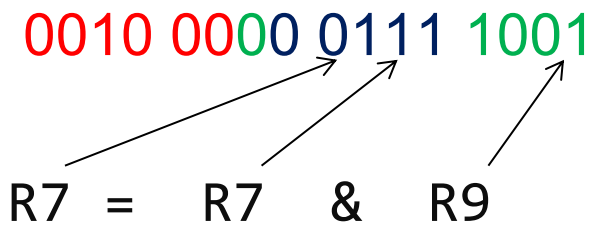
```
AND  Rd, Rr
```

  - Description:

```
Rd = Rd & Rr
```

  - Machine code

  - Machine code example:



ALU + Imm instruction- Example

- Logical AND with Immediate
  - Syntax:

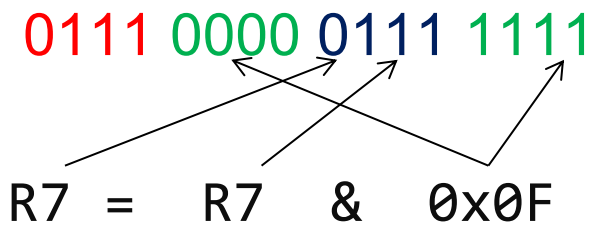
```
ANDI  Rd, K
```

  - Description:

```
Rd = Rd & K
```

  - Machine code

  - Machine code example:



AVR Instruction Set

- Arithmetic Instructions
- Logic Instructions
- Branch Instructions
- Memory Access Instructions
- Other

Arithmetic instructions

ADD Rd, Rr *Add without carry*	Rd=Rd+Rr, Carry flag not affected
ADC Rd, Rr *add with carry*	Rd=Rd+Rr, Carry flag affected
SUB Rd, Rr *Subtract without carry*	Rd=Rd-Rr, Carry flag not affected
SUBI Rd, K Subract Immediate	Rd = Rd – K, 0<=K<=255

Logic instructions

OR Rd, Rr *logical or*	Rd=Rd | Rr
AND Rd, Rr *logical add*	Rd=Rd & Rr
XOR Rd, Rr *logical xor*	Rd=Rd ^ Rr
ANDI Rd, K *logical and with immediate*	Rd=Rd & K, 0<=K<=255

Branch Instructions- Unconditional

Modify the Program Counter (PC) register

RJMP k *Relative Jump*	PC=PC+k+1 Jump distance PC-2048+1 to PC+2048+1
RCALL k *Relative Call to subroutine*	PC=PC+k+1 STACK=PC+1 PC-2048+1 to PC+2048+1
CALL k *Call subroutine*	PC=k Covers entire memory space
RET *return from subroutine*	PC=STACK

- k field is a 16 bit

Branch Instructions- Unconditional

Modify the Program Counter (PC) register

RJMP k *Relative Jump*	PC=PC+k+1 Jump distance PC-2048+1 to PC+2048+1
RCALL k *Relative Call to subroutine*	PC=PC+k+1 STACK=PC+1 PC-2048+1 to PC+2048+1
CALL k *Call subroutine*	PC=k Covers entire memory space
RET *return from subroutine*	PC=STACK

Call a function

Branch Instructions- Unconditional

Modify the Program Counter (PC) register

RJMP k *Relative Jump*	PC=PC+k+1 48+1
RCALL k *Relative Call to subroutine*	PC+1 -2048+1 to PC+2048+1
CALL k *Call subroutine*	PC=k Covers entire memory space
RET *return from subroutine*	PC=STACK

Return from a function

Branch Instructions- Conditional (1)

Modify the Program Counter (PC) register if a condition is satisfied

BREQ label *branch if equal*	if Z==1 PC=label, jump distance to label
BRNE label *branch if not equal*	if Z==0 PC=label, jump distance to label

- Use cp (compare), cpi (compare immediate), sub (subtract) or subi (subract immediate) before to set Z flag

Branch Instructions- Conditional (2)

Modify the Program Counter (PC) register if a condition is satisfied

BRLT label *branch if less than*	if S==1 PC=label, jump distance to label
BRCC label *branch if carry is cleared*	if C==0 PC=label, jump distance to label
BRCS label *branch if carry is set*	if C==1 PC=label, jump distance to label
BRGE label *branch if greater equal than*	if S==0 PC=label, jump distance to label

- Use cp (compare), cpi (compare immediate), sub (subtract) or subi (subract immediate) before to set Z flag

Branch Instructions- Example

Example:

```
cp r1, r0 ; Compare registers r1 and r0
breq equal ; Branch if registers equal
...
equal: nop ; Branch destination (do nothing)
```

Think about the delay slot!

Memory Access Instructions

Processor loads value into registers

LDS Rd, k *Load direct from data space*	Address=k Rd=*Address
LDI Rd, k *Load immediate*	Rd = k

- k is 8-bit immediate value
- Can only access register 16 to 31

Memory Access Instructions

Processor writes content of registers to memory

STS Rd, k *Store direct to data space*	Address=k *Address = Rd

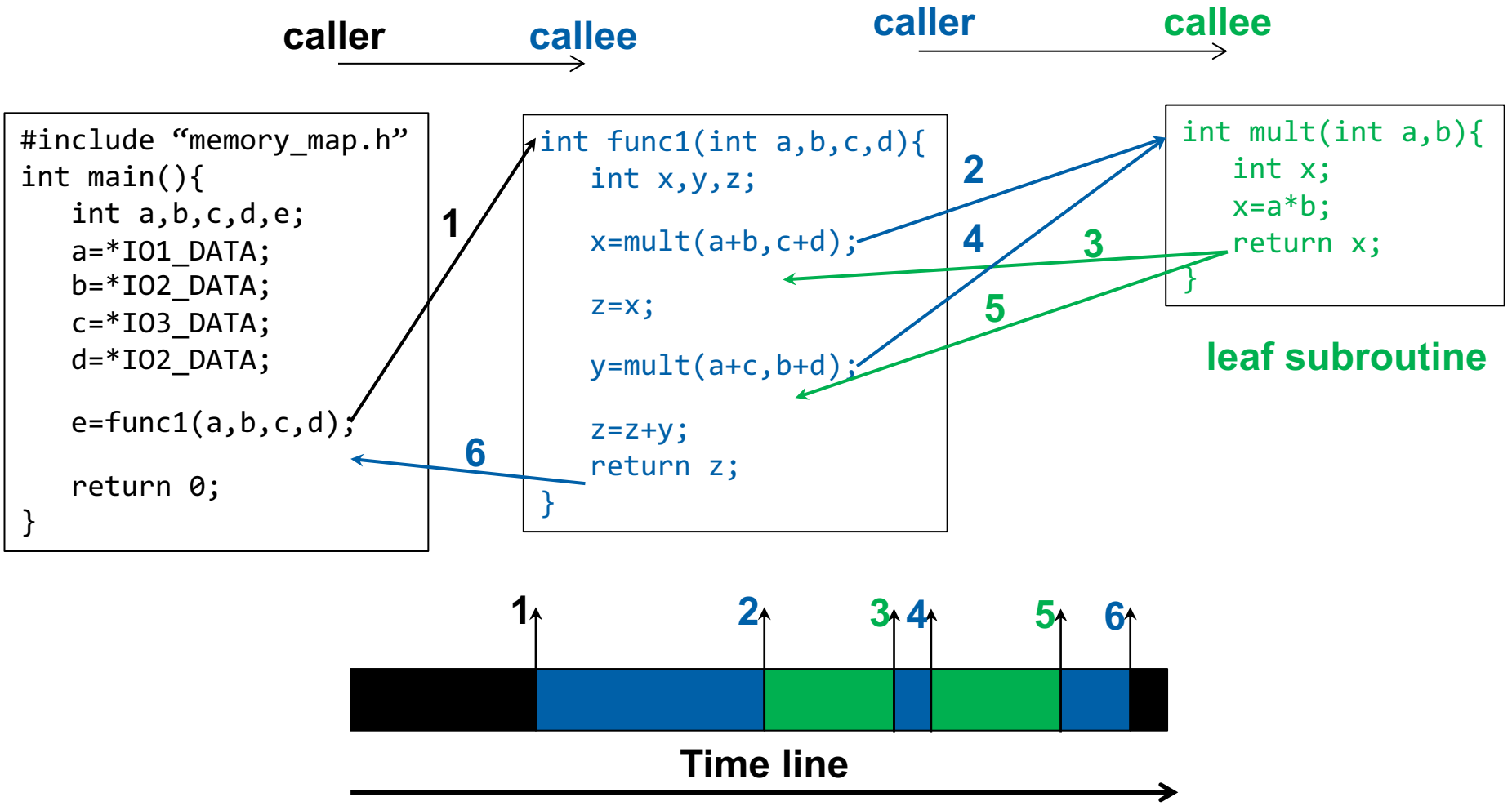
- k is 8-bit immediate value
- Can only access register 16 to 31

Other Instructions

- More instructions are available
- Check the website and online manual

https://www.microchip.com/webdoc/avrassembler/avrassembler.wb_instruction_list.html

Functions (subroutines)



Functions (subroutines)

- Caller
  - Prepare input arguments and pass them to the callee
  - Provide a return address to the callee
- Callee
  - Provide return values (outputs)
  - Ensure that the caller can seamlessly proceed, once the callee returns to the caller

Functions (subroutines) - problems

- How to pass arguments to functions?
- How to return values from functions?
  - **FOLLOW A REGISTER USAGE CONVENTION**
- How to ensure that registers retain values across function calls?
- Where to return after a function has been executed?
- Where to store temporary local variables of a function?
  - **USE THE STACK**

Register Usage Convention

- Dedicated
  - dedicated usage
- Volatile
  - Do not retain values across function calls
  - Store temporary results
  - Passing parameters/ Return values
- Non-volatile
  - Must be saved across function calls
  - Saved by callee

Register Usage Convention

Register	Description	Assembly code called from C	Assembly code that calls C code
r0	Temporary	Save and restore if using	Save and restore if using
r1	Always zero	Must clear before returning	Must clear before calling
r2-r17	"call-saved"	Save and restore if using	Can freely use
r28			
r29			
r18-r27	"call-used"	Can freely use	Save and restore if using
r0			
r31			

More Infos on Calling Conventions (Click Me)

Stack

- Memory segment
- Grows towards lower memory addresses
- Access the stack through a stack pointer
- Stack pointer points to the top of the stack
- Two operations
  - PUSH an item on top of the stack
  - POP the top item from the stack

Memory	
0x00	
0x01	
0x02	
0x03	
…	
0xFFC	
0xFFD	
0xFFE	
0xFFF	

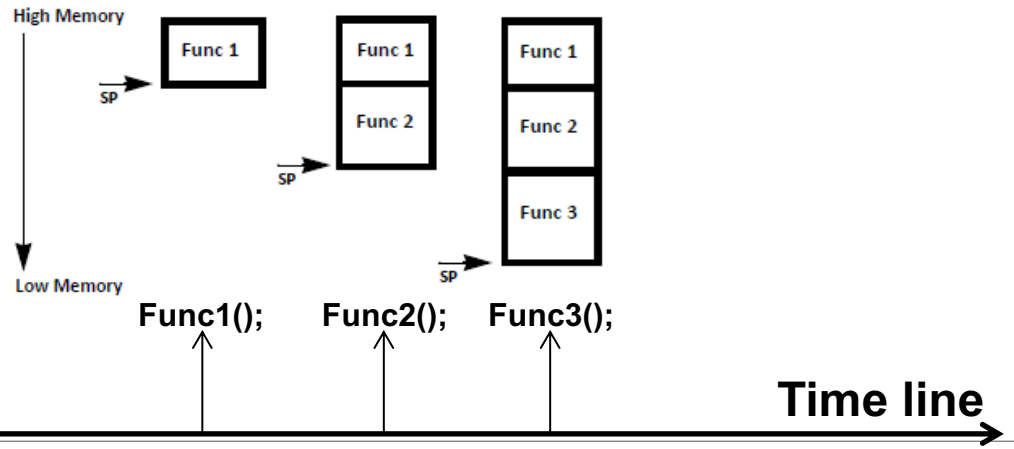
Stack pointer ⟶

Stack frame

- Temporal storage for the function to do its own book-keeping
- Items inside a stack frame include:
  - Return address
  - Local variables used by the function
  - Save registers that the function may modify, but the caller function does not want changed
  - Input arguments to callee functions

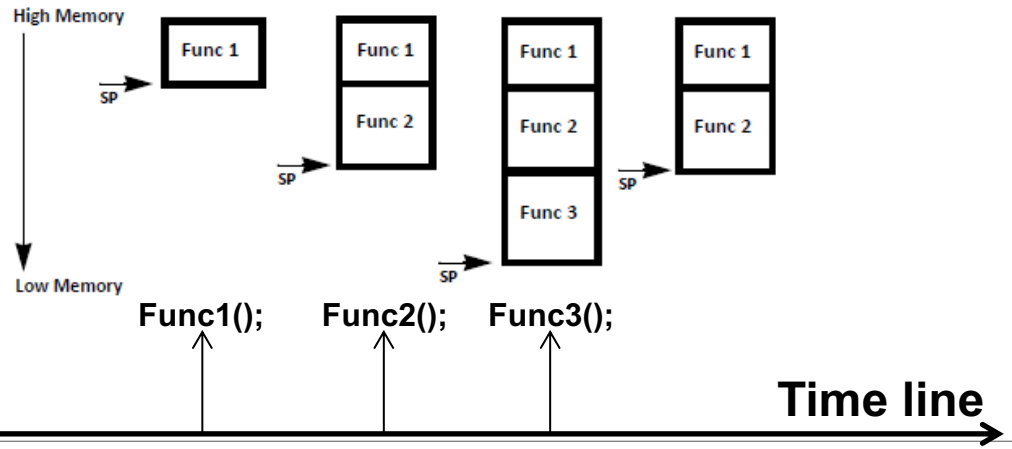
When a function is called…

- Reserve space on the stack for the stack frame
  - Decrement the stack pointer
- Store necessary information in the stack frame
  - Return address
  - Non-volatile registers
- Store input arguments provided through registers, in the caller's stack frame



When a function returns…

- Load necessary information from the stack frame and restore registers
  - Return address
  - Non-volatile registers
- Pop the stack frame from the stack
  - Increment the stack pointer
- Return to the caller



Assembly program

```
.global number_of_ones
.text
.ent number_of_ones
number_of_ones:   add r3,r0,r0
while:            beqid r5, result
                  nop
                  andi r4,r5,1
                  add r3,r3,r4
                  sra r5,r5
                  brid while
                  nop
result:           rtsd r15, 8
                  nop
.end number_of_ones
```

Assembly directives
Assembly instructions
Symbols (labels)

use labels for branch instructions

Assembly program

```
.global number_of_ones
.text
.ent number_of_ones
number_of_ones:     add r3,r0,r0
while:              beqid r5, result
                    nop
                    andi r4,r5,1
                    add r3,r3,r4
                    sra r5,r5
                    brid while
                    nop
result:             rtsd r15, 8
                    nop
.end number_of_ones
```

```
unsigned int number_of_ones(unsigned int x){
unsigned int temp=0;// temp is stored in r3
  while (x!=0){
          temp=temp+x&1;
          x>>=1;
  }
  return temp;
}
```

Disassembled program

0x6C0	add r3,r0,r0
0x6C4	beqid r5, 28
0x6C8	or r0,r0,r0
0x6CC	andi r4,r5,1
0x6D0	add r3,r3,r4
0x6D4	sra r5,r5
0x6D8	brid -20
0x6DC	or r0,r0,r0
0x6E0	rtsd r15, 8
0x6E4	or r0,r0,r0

```
.global number_of_ones
.text
.ent number_of_ones
number_of_ones:  add r3,r0,r0
while:           beqid r5, result
                 nop
                 andi r4,r5,1
                 add r3,r3,r4
                 sra r5,r5
                 brid while
                 nop
result:          rtsd r15, 8
                 nop
.end number_of_ones
```

Tips and tricks

- Initialize a register with a known value
  - Example load register `r16` with 150

    ```
    ldi r16,150
    ```
- Shift to left
  - Example register `r16` to be shifted one position to left

    ```
    add r16,r16 // r16=r16*2==r16<<1
    ```
  - How about shifting multiple positions to the left?

Tips and tricks

- IF statement

Assume x is stored in r5

```
if (x>0){
    block_true
    …
}else{
    block_false
    …
}
y=…
```

```
        cp r5, 0
        breq false
        block_true
        …
        rjmp end_if
false:  block_false
        …
end_if: y=…
```

Note the condition is inverted

Tips and tricks

- WHILE loop

```
while (x>0){
    block
    …
}
y=…
```

```
condition:   cp r5, 0
             breq while_end
             block
             …
             rjmp condition
while_end:   y=…
```

Assume x is stored in r5

Note the condition is inverted

Tips and tricks

- Multiplication
  - Example `r3` stores the product `r5*r6`

```
        add r3,0
again:  cp r6, 0
        breq r6, done
        add r3, r5
        addi r6, -1
        rjmp again
done:   nop
```

LUNDS
UNIVERSITET