B38DF
A case study: AVR microcontrollers

Alexander Belyaev

Heriot-Watt University
School of Engineering & Physical Sciences
Electrical, Electronic and Computer Engineering

E-mail: a.belyaev@hw.ac.uk
Office: EM2.29

Based on "Some assembly Required, Assembly Language Programming with the AVR Microcontroller" of Timothy S. Margush

Atmel AVR Microcontrollers

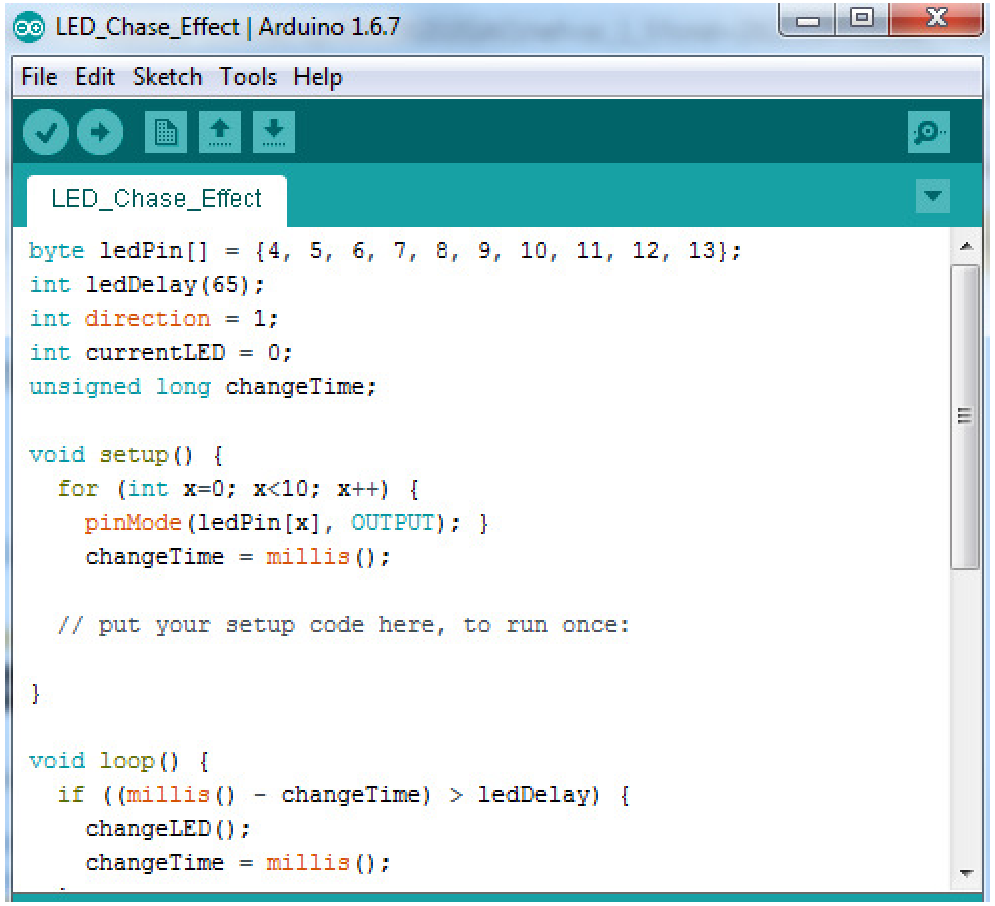
AVR is a family of micricontrollers developed since 1996 by Atmel, acquired by Microchip Technology in 2016. These are Harvard architecture 8-bit enhanced RISC microcontrollers. AVR was one of the first microcontroller families to use on-chip flash memory for program storage, as opposed to ROM, EPROM, or EEPROM used by other microcontrollers at the time. AVR microcontrollers find many applications as embedded systems. They are especially common in hobbyist and educational embedded applications, popularized by their inclusion in many of the **Arduino** line of hardware development boards. (from Wikipedia)

The **8 bit** part typically means that the **microcontroller** uses a word size of **8 bits**, so assembly instructions are **8 bits** each, internal registers are **8 bits** wide, and memory accesses read **8 bits** at a time.

The **AVR architecture** was developed by two graduate students of Norwegian Institute of Technology, Alf-Egil Bogen and Vegard Wollan.

AVR may stand for **A**lf-Egil Bogen and **V**egard Wollan **R**ISC microcontroller, as well as for **A**dvanced **V**irtual **R**ISC.

Arduino



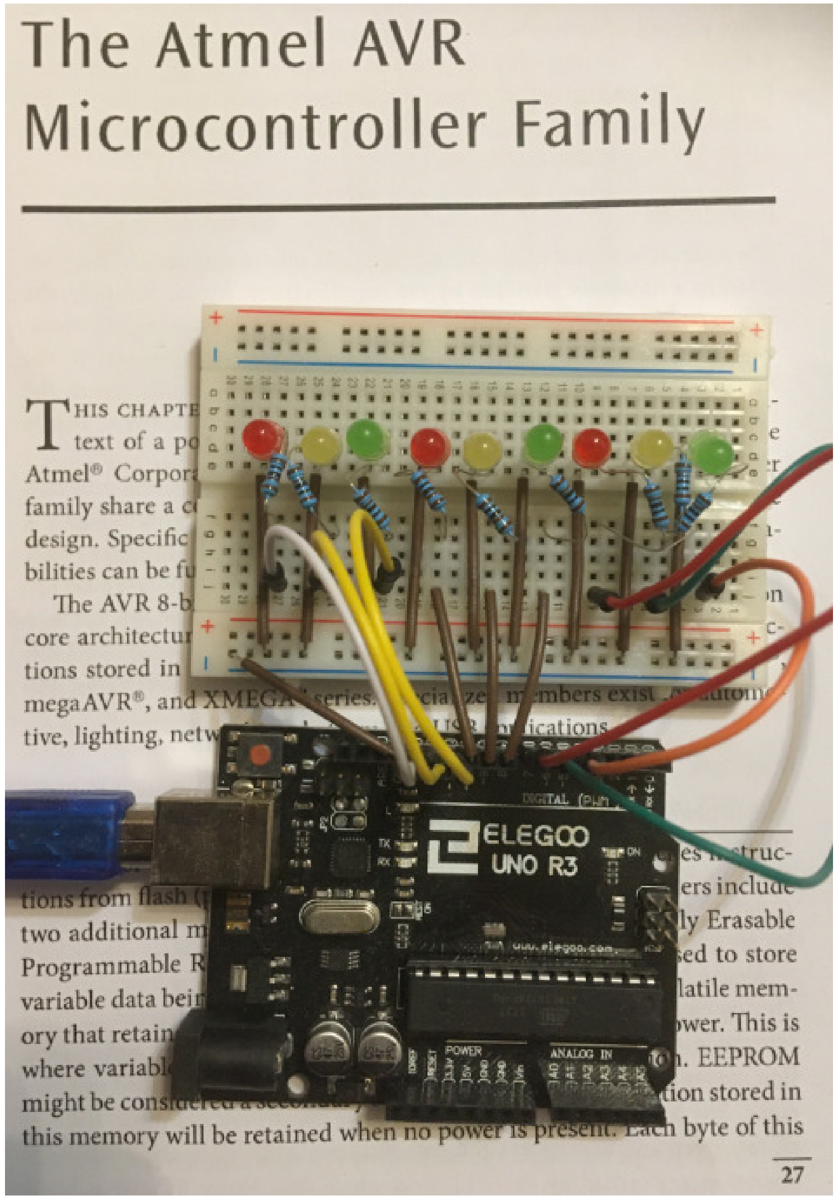
```
byte ledPin[] = {4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
int ledDelay(65);
int direction = 1;
int currentLED = 0;
unsigned long changeTime;

void setup() {
  for (int x=0; x<10; x++) {
    pinMode(ledPin[x], OUTPUT); }
    changeTime = millis();

  // put your setup code here, to run once:

}

void loop() {
  if ((millis() - changeTime) > ledDelay) {
    changeLED();
    changeTime = millis();
```



Atmel AVR Microcontrollers

The AVR 8-bit RISC microcontrollers are designed around a common ore architecture: a Harvard architecture with program instructions stored in ROM.

The core of of the AVR microcontrollers included a Control Unit (CU) that fetches instructions from flash (program) memory.

Many of the microcontrollers include two additional memories, static RAM (SRAM) and Electrically Erasable Programmable Read Only Memory (EEPROM) which are used to store variable data being manipulated by the program.

The microcontroller’s flash memory is also a form of EEPROM, but it is usually must be erased and rewritten in large sections, called blocks. The flash memory is usually treated as read-only when application is running.

Atmel AVR Microcontrollers: Instructions

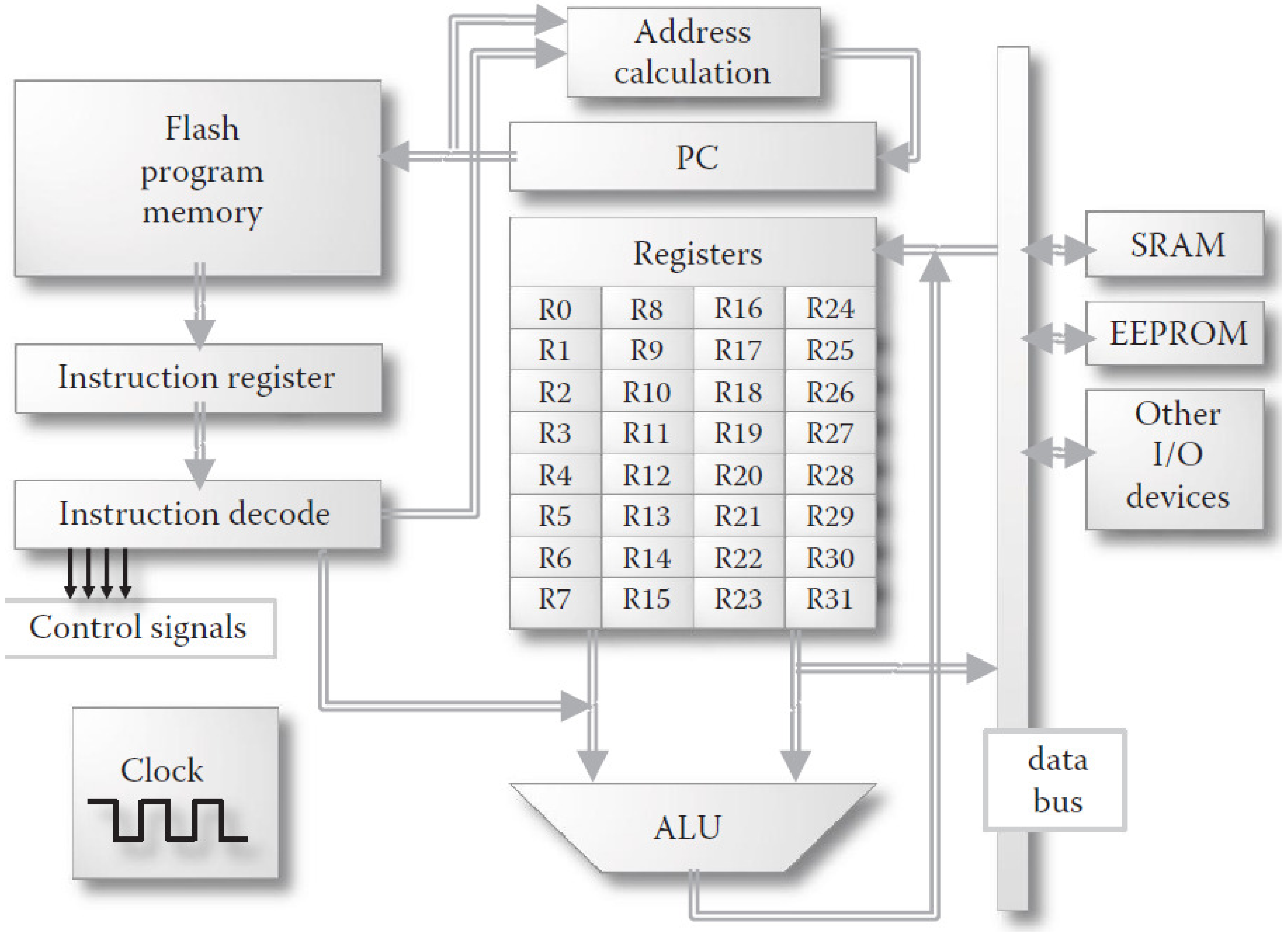
Roughly, the instructions can be divided into four categories:

- arithmetic/logical
- memory access
- branch
- I/O

Most instructions are 16-bits in length (some are 32-bits), and program memory is organized as a collection of 16-bit words. Program memory is word addressable: each program memory address refers to a pair of bytes. The other AVR memories, SRAM and EEPROM, are byte addressable.

Atmel AVR Microcontrollers: Registers

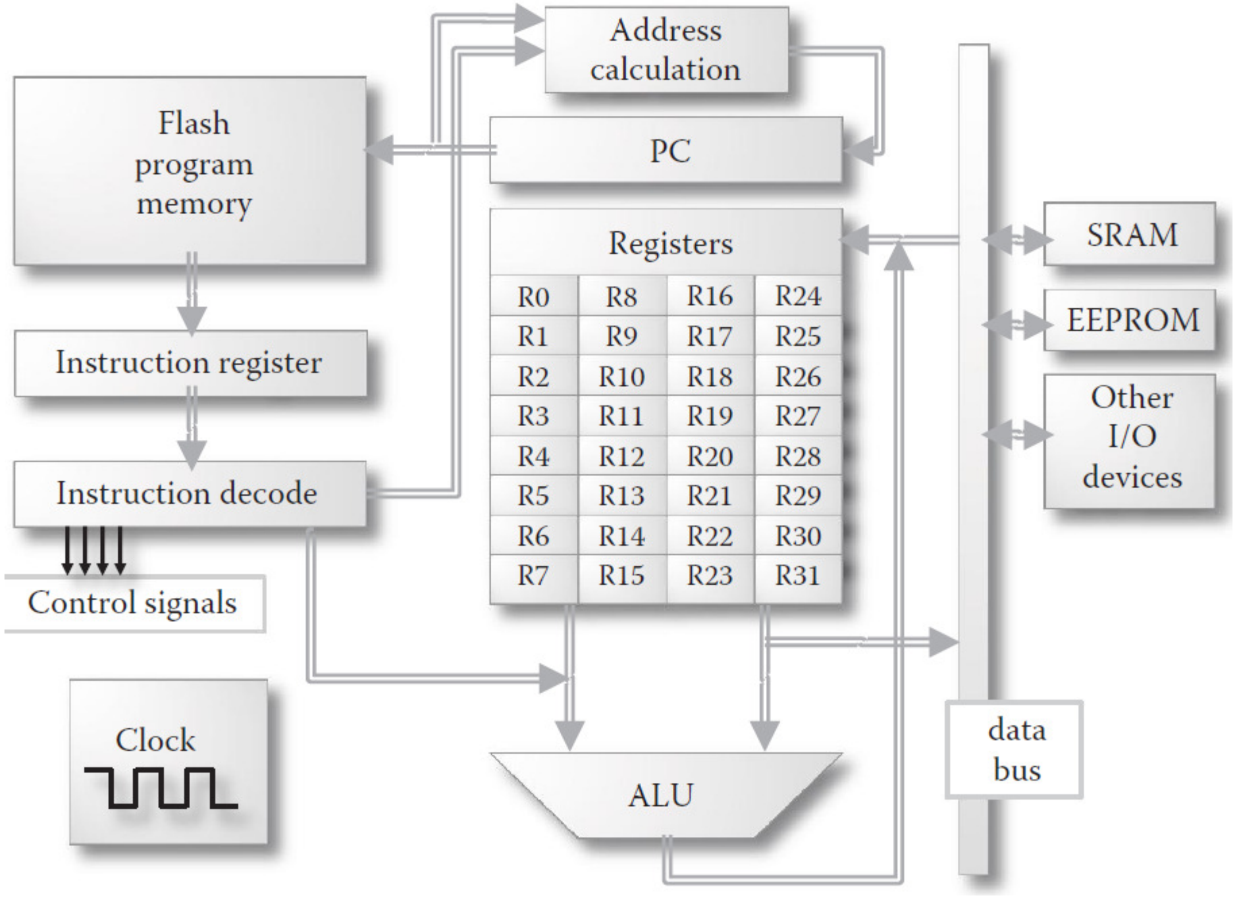
The AVR processor has 32 general purpose 8-bit registers: R0,…,R31.



Atmel AVR Microcontrollers: PC register

The PC always holds the address of the next instruction to be fetched. The PC register is usually a 16-bit register and holds a word address (as memory sizes vary, some models do not include all of the bits and some need more bits to accommodate large flash memories).

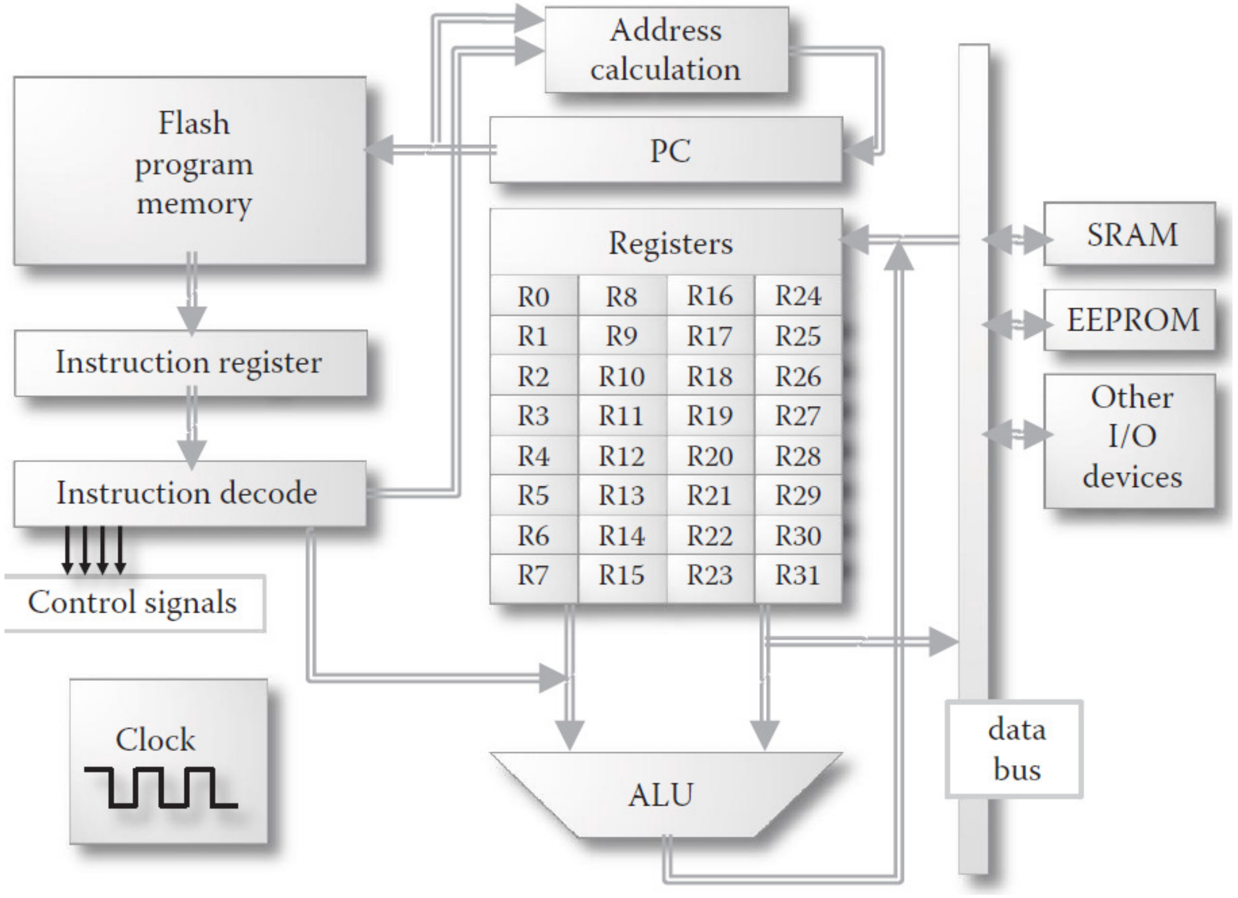
A 16-bit PC register allows 65,536 distinct addresses; each address corresponds to a different word in program memory. Thus, the maximum addressable program space of most of the AVR processors is 64 kwords or 128 kbytes.



Atmel AVR Microcontrollers: Pipelining

The instruction and instruction decode registers allow one instruction to be fetched while the previous is being decoded and executed. This creates a very simple, two-stage pipeline. With a single instruction register, one clock cycle would be required to fetch an instruction; it would be executed during the next clock cycle.

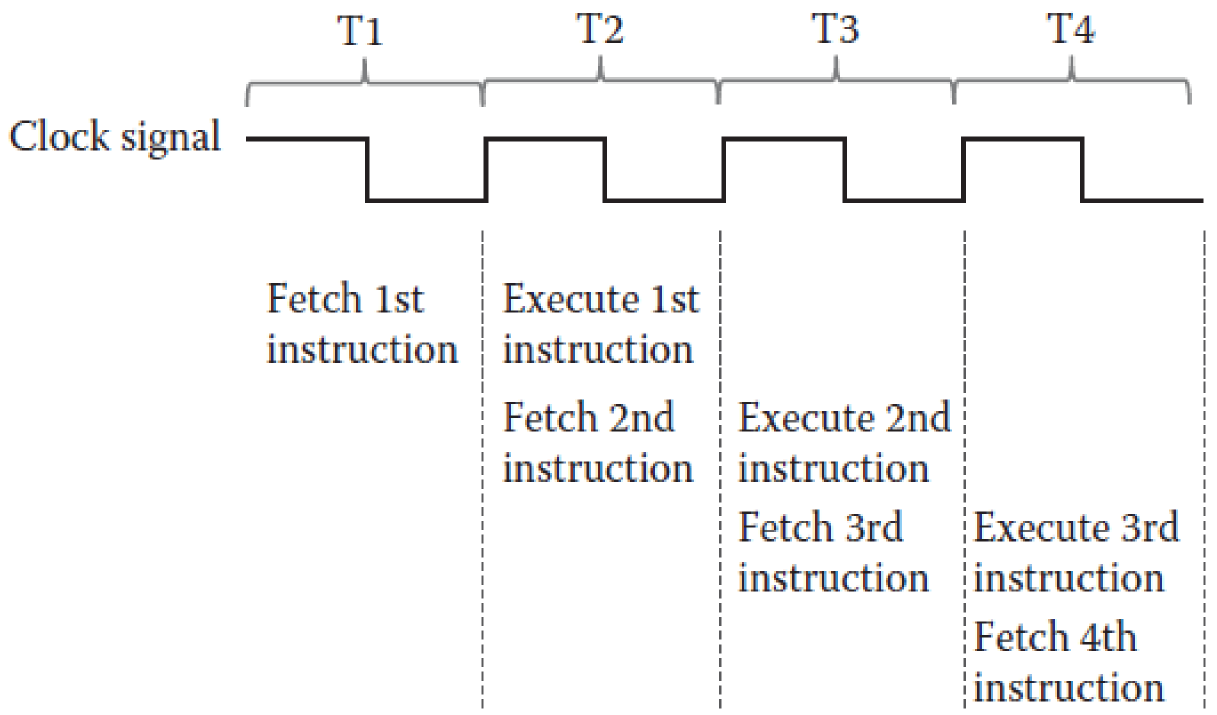
The instruction pipeline makes it possible to fetch and execute in parallel. This, together with the fact that most instructions can be fetched and executed in a single clock cycle, allows the processor to achieve a throughput of close to one instruction per clock cycle.



Atmel AVR Microcontrollers: Clock

The clock is an oscillating (zero-one) signal that drives the processor’s fetch–execute cycle. The AVR processor actually uses several clocks for different subsystems. The clock referred to here is called the CPU clock. The clock signal driving the processor can come from an external clock source (off chip), or it can be an internally generated signal.

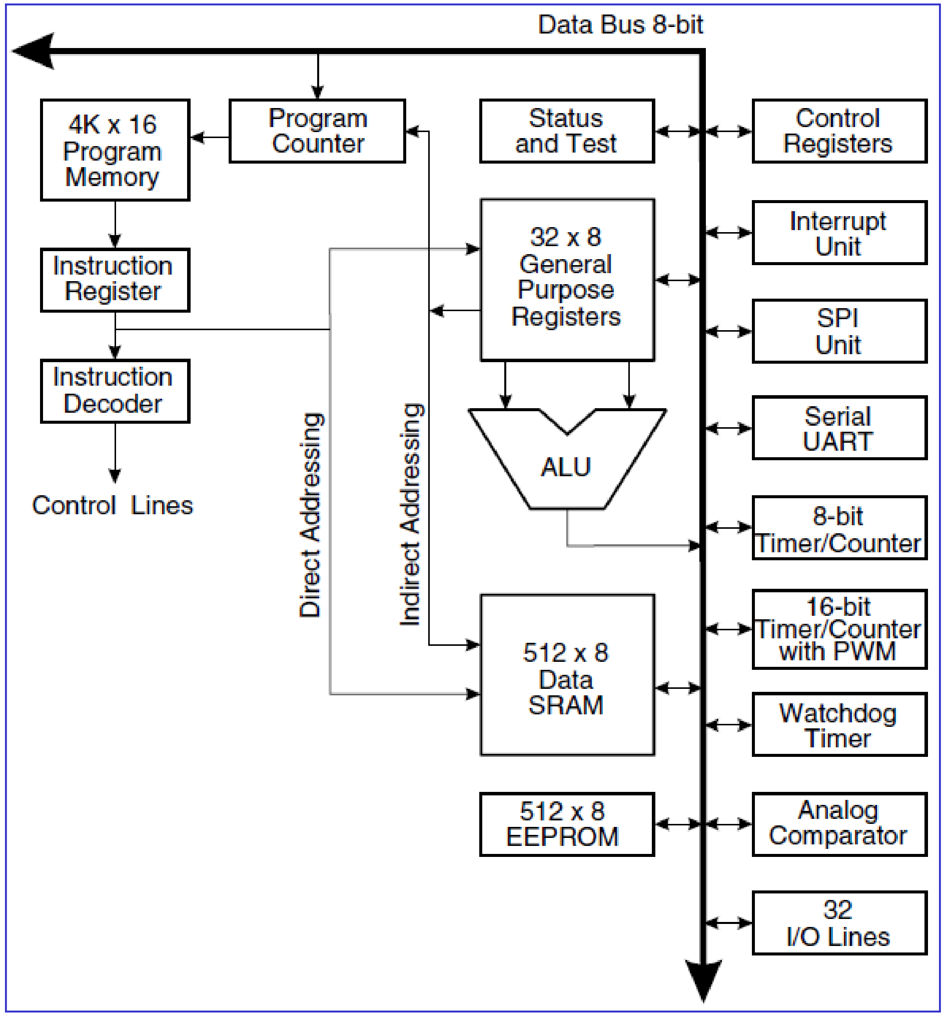
External clocks are used when very precise timings are required. For many purposes, the internal clock is sufficient. The 8-bit AVR processors are designed to run at a variety of clock speeds, from very slow to 32MHz.

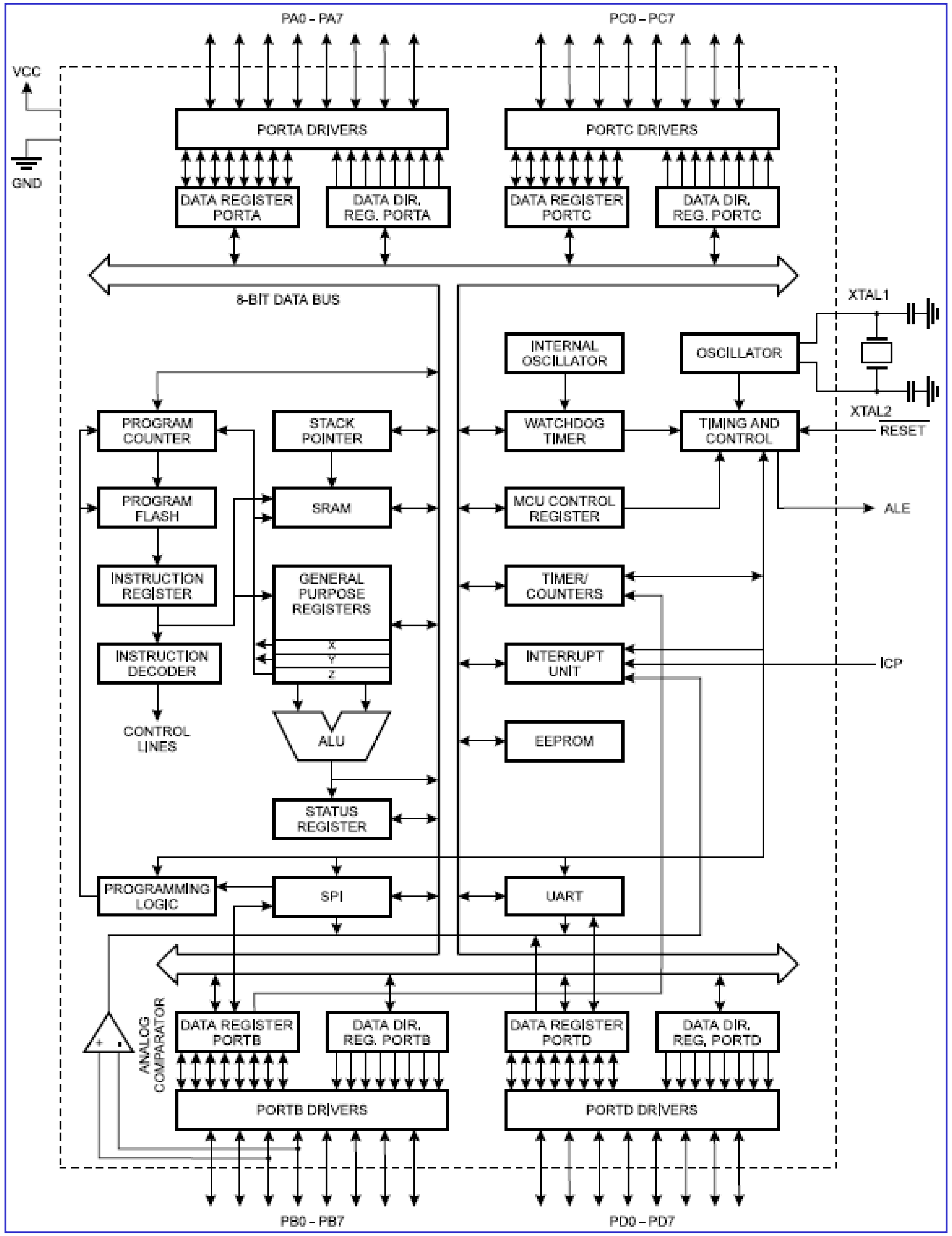


Atmel AVR Microcontrollers

AT90S8515 block diagram

AT90S8515 AVR RISC Architecture





Atmel AVR Microcontrollers: Instruction Set Summary

Assemblers allow programming at a level just above the machine level. Each assembly language instruction is translated by an assembler to a single machine instruction. Higher level languages, such as C, are translated by a compiler to assembly language, or directly to machine language. One high-level statement can take tens or hundreds of machine instructions to accomplish the intended result.

Mnemonic	Operands	Description	Operation	Flags	# Clocks
ARITHMETIC AND LOGIC INSTRUCTIONS					
ADD	Rd, Rr	Add Two Registers	Rd ← Rd + Rr	Z,C,N,V,H	1
ADC	Rd, Rr	Add with Carry Two Registers	Rd ← Rd + Rr + C	Z,C,N,V,H	1
ADIW	Rdl, K	Add Immediate to Word	Rdh:Rdl ← Rdh:Rdl + K	Z,C,N,V,S	2
SUB	Rd, Rr	Subtract Two Registers	Rd ← Rd - Rr	Z,C,N,V,H	1
SUBI	Rd, K	Subtract Constant from Register	Rd ← Rd - K	Z,C,N,V,H	1
SBC	Rd, Rr	Subtract with Carry Two Registers	Rd ← Rd - Rr - C	Z,C,N,V,H	1
SBCI	Rd, K	Subtract with Carry Constant from Reg.	Rd ← Rd - K - C	Z,C,N,V,H	1
SBIW	Rdl, K	Subtract Immediate from Word	Rdh:Rdl ← Rdh:Rdl - K	Z,C,N,V,S	2
AND	Rd, Rr	Logical AND Registers	Rd ← Rd • Rr	Z,N,V	1
ANDI	Rd, K	Logical AND Register and Constant	Rd ← Rd • K	Z,N,V	1
OR	Rd, Rr	Logical OR Registers	Rd ← Rd v Rr	Z,N,V	1
ORI	Rd, K	Logical OR Register and Constant	Rd ← Rd v K	Z,N,V	1
EOR	Rd, Rr	Exclusive OR Registers	Rd ← Rd ⊕ Rr	Z,N,V	1
COM	Rd	One's Complement	Rd ← $FF - Rd	Z,C,N,V	1
NEG	Rd	Two's Complement	Rd ← $00 - Rd	Z,C,N,V,H	1
SBR	Rd, K	Set Bit(s) in Register	Rd ← Rd v K	Z,N,V	1
CBR	Rd, K	Clear Bit(s) in Register	Rd ← Rd • ($FF - K)	Z,N,V	1
INC	Rd	Increment	Rd ← Rd + 1	Z,N,V	1
DEC	Rd	Decrement	Rd ← Rd - 1	Z,N,V	1
TST	Rd	Test for Zero or Minus	Rd ← Rd • Rd	Z,N,V	1
CLR	Rd	Clear Register	Rd ← Rd ⊕ Rd	Z,N,V	1
SER	Rd	Set Register	Rd ← $FF	None	1

Atmel AVR Microcontrollers: Instruction Set Summary

Mnemonic	Operands	Description	Operation	Flags	# Clocks
BRANCH INSTRUCTIONS					
RJMP	k	Relative Jump	PC ← PC + k + 1	None	2
IJMP		Indirect Jump to (Z)	PC ← Z	None	2
RCALL	k	Relative Subroutine Call	PC ← PC + k + 1	None	3
ICALL		Indirect Call to (Z)	PC ← Z	None	3
RET		Subroutine Return	PC ← STACK	None	4
RETI		Interrupt Return	PC ← STACK	I	4
CPSE	Rd, Rr	Compare, Skip if Equal	if (Rd = Rr) PC ← PC + 2 or 3	None	1/2/3
CP	Rd, Rr	Compare	Rd - Rr	Z,N,V,C,H	1
CPC	Rd, Rr	Compare with Carry	Rd - Rr - C	Z,N,V,C,H	1
CPI	Rd, K	Compare Register with Immediate	Rd - K	Z,N,V,C,H	1
SBRC	Rr, b	Skip if Bit in Register Cleared	if (Rr(b) = 0) PC ← PC + 2 or 3	None	1/2/3
SBRS	Rr, b	Skip if Bit in Register is Set	if (Rr(b) = 1) PC ← PC + 2 or 3	None	1/2/3
SBIC	P, b	Skip if Bit in I/O Register Cleared	if (P(b) = 0) PC ← PC + 2 or 3	None	1/2/3
SBIS	P, b	Skip if Bit in I/O Register is Set	if (P(b) = 1) PC ← PC + 2 or 3	None	1/2/3
BRBS	s, k	Branch if Status Flag Set	if (SREG(s) = 1) then PC ← PC + k + 1	None	1/2
BRBC	s, k	Branch if Status Flag Cleared	if (SREG(s) = 0) then PC ← PC + k + 1	None	1/2
BREQ	k	Branch if Equal	if (Z = 1) then PC ← PC + k + 1	None	1/2
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None	1/2
BRCS	k	Branch if Carry Set	if (C = 1) then PC ← PC + k + 1	None	1/2
BRCC	k	Branch if Carry Cleared	if (C = 0) then PC ← PC + k + 1	None	1/2
BRSH	k	Branch if Same or Higher	if (C = 0) then PC ← PC + k + 1	None	1/2
BRLO	k	Branch if Lower	if (C = 1) then PC ← PC + k + 1	None	1/2
BRMI	k	Branch if Minus	if (N = 1) then PC ← PC + k + 1	None	1/2
BRPL	k	Branch if Plus	if (N = 0) then PC ← PC + k + 1	None	1/2
BRGE	k	Branch if Greater or Equal, Signed	if (N ⊕ V = 0) then PC ← PC + k + 1	None	1/2
BRLT	k	Branch if Less Than Zero, Signed	if (N ⊕ V = 1) then PC ← PC + k + 1	None	1/2
BRHS	k	Branch if Half-carry Flag Set	if (H = 1) then PC ← PC + k + 1	None	1/2
BRHC	k	Branch if Half-carry Flag Cleared	if (H = 0) then PC ← PC + k + 1	None	1/2
BRTS	k	Branch if T-flag Set	if (T = 1) then PC ← PC + k + 1	None	1/2
BRTC	k	Branch if T-flag Cleared	if (T = 0) then PC ← PC + k + 1	None	1/2
BRVS	k	Branch if Overflow Flag is Set	if (V = 1) then PC ← PC + k + 1	None	1/2
BRVC	k	Branch if Overflow Flag is Cleared	if (V = 0) then PC ← PC + k + 1	None	1/2
BRIE	k	Branch if Interrupt Enabled	if (I = 1) then PC ← PC + k + 1	None	1/2
BRID	k	Branch if Interrupt Disabled	if (I = 0) then PC ← PC + k + 1	None	1/2

Atmel AVR Microcontrollers: Instruction Set Summary

Mnemonic	Operands	Description	Operation	Flags	# Clocks
DATA TRANSFER INSTRUCTIONS					
MOV	Rd, Rr	Move between Registers	Rd ← Rr	None	1
LDI	Rd, K	Load Immediate	Rd ← K	None	1
LD	Rd, X	Load Indirect	Rd ← (X)	None	2
LD	Rd, X+	Load Indirect and Post-inc.	Rd ← (X), X ← X + 1	None	2
LD	Rd, -X	Load Indirect and Pre-dec.	X ← X - 1, Rd ← (X)	None	2
LD	Rd, Y	Load Indirect	Rd ← (Y)	None	2
LD	Rd, Y+	Load Indirect and Post-inc.	Rd ← (Y), Y ← Y + 1	None	2
LD	Rd, -Y	Load Indirect and Pre-dec.	Y ← Y - 1, Rd ← (Y)	None	2
LDD	Rd, Y+q	Load Indirect with Displacement	Rd ← (Y + q)	None	2
LD	Rd, Z	Load Indirect	Rd ← (Z)	None	2
LD	Rd, Z+	Load Indirect and Post-inc.	Rd ← (Z), Z ← Z + 1	None	2
LD	Rd, -Z	Load Indirect and Pre-dec.	Z ← Z - 1, Rd ← (Z)	None	2
LDD	Rd, Z+q	Load Indirect with Displacement	Rd ← (Z + q)	None	2
LDS	Rd, k	Load Direct from SRAM	Rd ← (k)	None	2
ST	X, Rr	Store Indirect	(X) ← Rr	None	2
ST	X+, Rr	Store Indirect and Post-inc.	(X) ← Rr, X ← X + 1	None	2
ST	-X, Rr	Store Indirect and Pre-dec.	X ← X - 1, (X) ← Rr	None	2
ST	Y, Rr	Store Indirect	(Y) ← Rr	None	2
ST	Y+, Rr	Store Indirect and Post-inc.	(Y) ← Rr, Y ← Y + 1	None	2
ST	-Y, Rr	Store Indirect and Pre-dec.	Y ← Y - 1, (Y) ← Rr	None	2
STD	Y+q, Rr	Store Indirect with Displacement	(Y + q) ← Rr	None	2
ST	Z, Rr	Store Indirect	(Z) ← Rr	None	2
ST	Z+, Rr	Store Indirect and Post-inc.	(Z) ← Rr, Z ← Z + 1	None	2
ST	-Z, Rr	Store Indirect and Pre-dec.	Z ← Z - 1, (Z) ← Rr	None	2
STD	Z+q, Rr	Store Indirect with Displacement	(Z + q) ← Rr	None	2
STS	k, Rr	Store Direct to SRAM	(k) ← Rr	None	2
LPM		Load Program Memory	R0 ← (Z)	None	3
IN	Rd, P	In Port	Rd ← P	None	1
OUT	P, Rr	Out Port	P ← Rr	None	1
PUSH	Rr	Push Register on Stack	STACK ← Rr	None	2
POP	Rd	Pop Register from Stack	Rd ← STACK	None	2

Atmel AVR Microcontrollers: Instruction Set Summary

Mnemonic	Operands	Description	Operation	Flags	# Clocks
BIT AND BIT-TEST INSTRUCTIONS					
SBI	P, b	Set Bit in I/O Register	I/O(P,b) ← 1	None	2
CBI	P, b	Clear Bit in I/O Register	I/O(P,b) ← 0	None	2
LSL	Rd	Logical Shift Left	Rd(n+1) ← Rd(n), Rd(0) ← 0	Z,C,N,V	1
LSR	Rd	Logical Shift Right	Rd(n) ← Rd(n+1), Rd(7) ← 0	Z,C,N,V	1
ROL	Rd	Rotate Left through Carry	Rd(0) ← C, Rd(n+1) ← Rd(n), C ← Rd(7)	Z,C,N,V	1
ROR	Rd	Rotate Right through Carry	Rd(7) ← C, Rd(n) ← Rd(n+1), C ← Rd(0)	Z,C,N,V	1
ASR	Rd	Arithmetic Shift Right	Rd(n) ← Rd(n+1), n = 0..6	Z,C,N,V	1
SWAP	Rd	Swap Nibbles	Rd(3..0) ← Rd(7..4), Rd(7..4) ← Rd(3..0)	None	1
BSET	s	Flag Set	SREG(s) ← 1	SREG(s)	1
BCLR	s	Flag Clear	SREG(s) ← 0	SREG(s)	1
BST	Rr, b	Bit Store from Register to T	T ← Rr(b)	T	1
BLD	Rd, b	Bit Load from T to Register	Rd(b) ← T	None	1
SEC		Set Carry	C ← 1	C	1
CLC		Clear Carry	C ← 0	C	1
SEN		Set Negative Flag	N ← 1	N	1
CLN		Clear Negative Flag	N ← 0	N	1
SEZ		Set Zero Flag	Z ← 1	Z	1
CLZ		Clear Zero Flag	Z ← 0	Z	1
SEI		Global Interrupt Enable	I ← 1	I	1
CLI		Global Interrupt Disable	I ← 0	I	1
SES		Set Signed Test Flag	S ← 1	S	1
CLS		Clear Signed Test Flag	S ← 0	S	1
SEV		Set Two's Complement Overflow	V ← 1	V	1
CLV		Clear Two's Complement Overflow	V ← 0	V	1
SET		Set T in SREG	T ← 1	T	1
CLT		Clear T in SREG	T ← 0	T	1
SEH		Set Half-carry Flag in SREG	H ← 1	H	1
CLH		Clear Half-carry Flag in SREG	H ← 0	H	1
NOP		No Operation		None	1
SLEEP		Sleep	(see specific descr. for Sleep function)	None	1
WDR		Watchdog Reset	(see specific descr. for WDR/timer)	None	1

Let us analyze some of the AVR microcontroller instructions

Load Immediate: 1110 xxxx xxxx xxxx

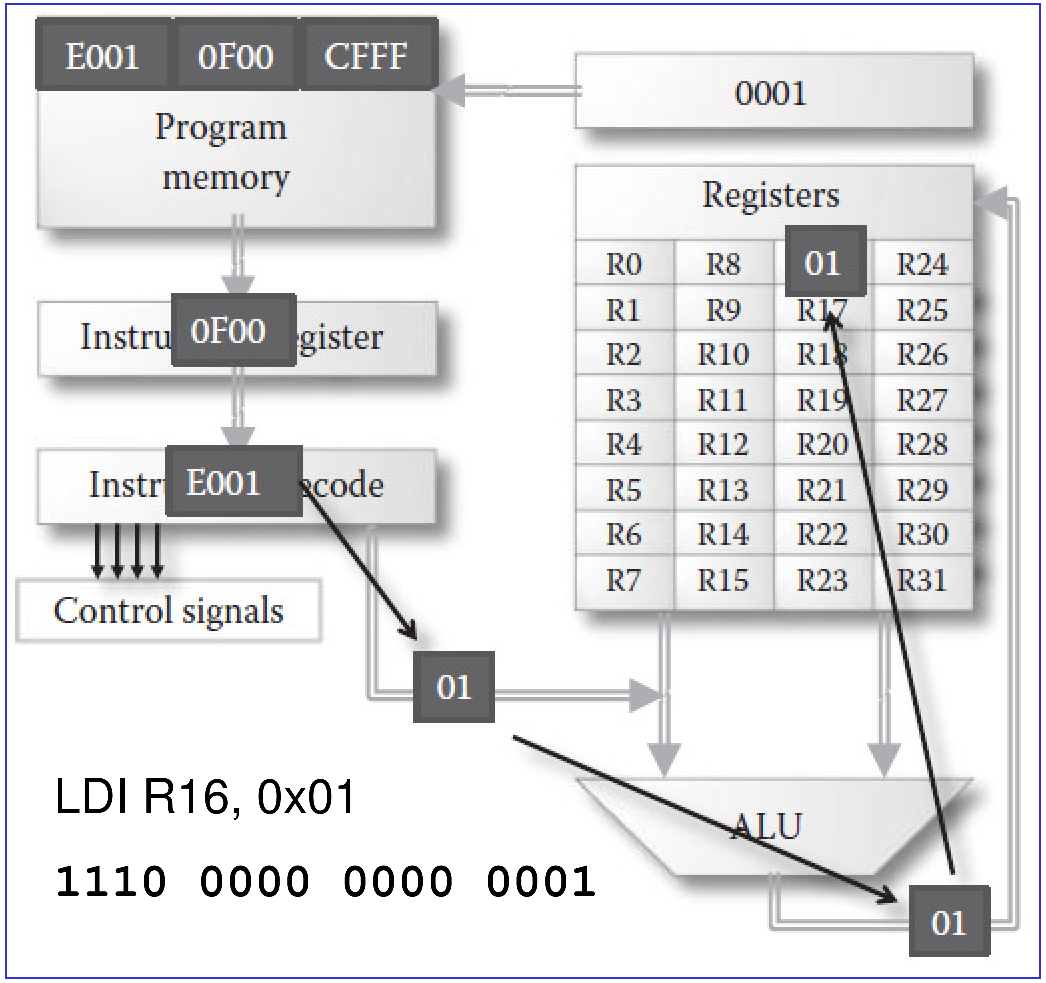
LDI Rd, K

The instruction to load the constant 0x3C into register R25 would be

```
1110 0011 1001 1100
```

E39C (0x9 is the code for R25 since 25 = 0x19). The general form for this instruction in assembly language is

```
1110 bbbb rrrr bbbb
```



LDI R16, 0x01

```
1110 0000 0000 0001
```

Let us analyze some of the AVR microcontroller instructions

Add: 0000 11xx xxxx xxxx

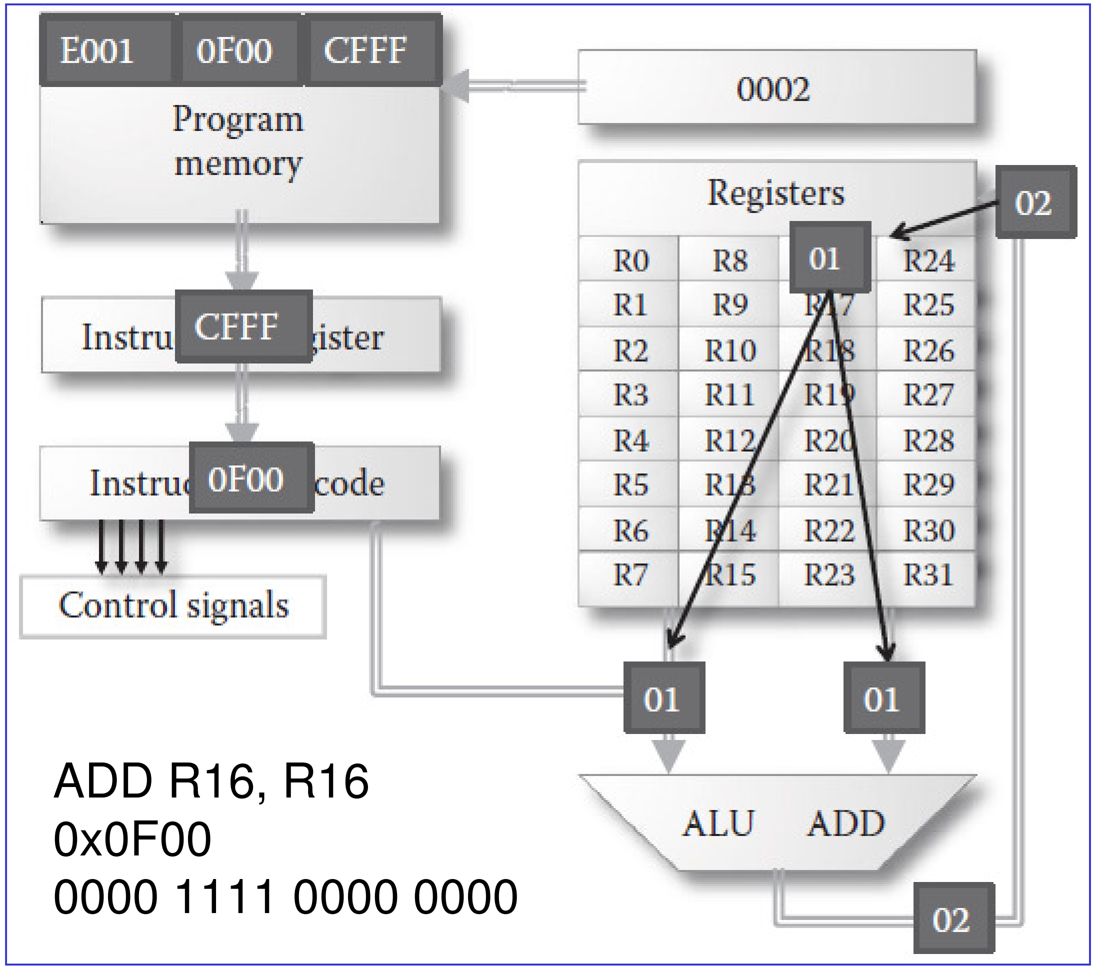
Notice that instruction opcodes can be of varying lengths.

ADD Rd, Rr   (Rd=Rd+Rr)

`0000 11rd dddd rrrr`

For example, if we want to ADD R4, R30, then the instruction would be

`0000 1110 0100 1110`
or 0x0E4E



ADD R16, R16
0x0F00
0000 1111 0000 0000

Atmel AVR Microcontrollers

Many existing RISC architectures require larger code size to perform a given task with the traditional CISC (Complex Instruction Set Computer) architectures. RISC microcontrollers (MicroController Units, MCUs) are often chosen where a high speed is needed. The reduced instruction set will be fast, but reduced in complexity.

The AVR is designed to be a RISC MCU with a larger number of instructions to reduce the code size and to increase the speed further. CISC-like instructions are introduced without letting the RISC performance and low power consumption features suffer. This first major enhancement was made after thorough analysis of several architectures and large amounts of application code. Still, the regular AVR RISC architecture enables cost effective implementations.

Programming for Atmel AVR microcontroller

Instruction	Description
LDS Rp, k	Load contents of data at location k into Rp
ADD Rp, Rq	Adds the contents of register q with register p stores the result into register p
SUB Rp, Rq	Subtracts the contents of register q from register p and stores the result into register p
INC Rp	Increments the contents of register p
DEC Rp	Decrements the contents of register p
BRBS Z, k	Jump to location k if alu operation results in a zero flag being set
BRBS N, k	Jump to location k if alu operation results in a negative flag being set
RJMP k	Jump to location k

```
if (A < B)
{
sum = sum - 2;
}
else
{
sum = sum + 1;
}
```

```
    LD R0, k1 // k1 data address holds A
    LD R1, k2 // k2 data address holds B
    SUB R0, R1 // A-B
    BRBS N, L2 // if A-B < 0
L1: INC R2 // R2 holds the sum
    RJMP L3
L2: DEC R2
    DEC R2
L3: Rest of code
```

Some AVR Microcontroller Instructions

Instruction	Description
LDS Rp, k	Load contents of data at location k into Rp
LDI Rp, K	Load integer value K to register Rp
ADD Rp, Rq	Adds the contents of Rq with Rp stores the result into Rp
SUB Rp, Rq	Subtracts the contents of Rq from Rp and stores the result into Rp
INC Rp	Increments the contents of register Rp
DEC Rp	Decrements the contents of register Rp
BRBS Z, k BRBS N, k BREQ k BRNE k BRGE k BRLT k	Jump to location k if ALU operation results in a zero flag being set Jump to location k if ALU operation results in a negative flag being set Jump to location k if ALU operation results in a zero flag being set Jump in the case of inequality Greater Equal Less Than
CPI Rp,K CP Rp,Rq	Compare the content of Rp with value K Compare the contents of Rp and Rq
RJMP k	Jump to location k

Implementing basic programming structures: `do-while`

```
x=1;
do
{
 // instructions
 // to be repeated
 x++;
}
while (x<=15);
```

```
   LDI R18,1
L1:
   ;instructions
   ;to be repeated
   INC R18
   CPI R18,16
   BRLT L1 ;if R18<16
```

```
x=15;
do
{
 // instructions
 // to be repeated
 x--;
}
while (x>0);
```

```
   LDI R18,15
L1:
   ;instructions
   ;to be repeated
   DEC R18
   BRNE L1 ;if R18!=0
```

Implementing basic programming structures: `while-do`

```
x=1;
while (x<16) do
{
 // instructions
 // to be repeated
 x++;
}
```

```
   LDI R18,1
L1:
   CPI R18,16
   BRLT L2
   RJMP L3
L2:
   ;instructions
   ;to be repeated
   INC R18
   RJMP L1
L3:
```

Implementing basic programming structures: `while-do`

```
x=1;
while (x<16) do
{
 // instructions
 // to be repeated
 x++;
}
```

```
   LDI R18,1
L1:
   CPI R18,16
   BRGE L3
   ;instructions
   ;to be repeated
   INC R18
   RJMP L1
L3:
```

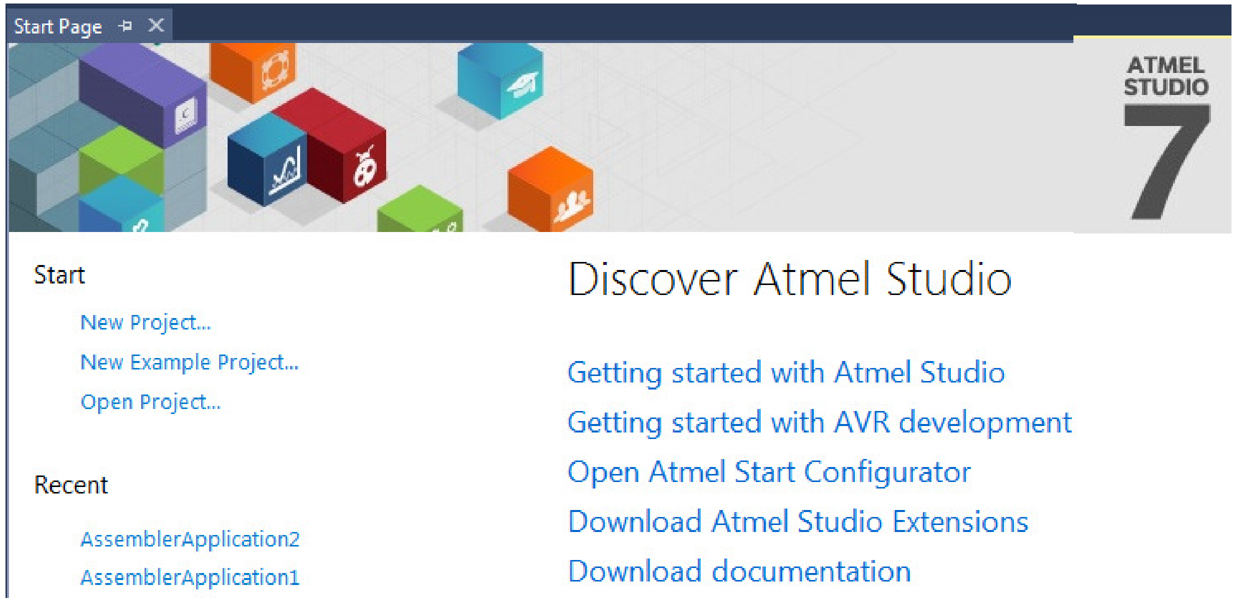
Implementing basic programming structures: nested loops

```
x=1;
while (x<16) do
{
 // instructions
 // to be repeated
 x++;
}
```

```
   LDI R18,1
L1:
   CPI R18,16
   BRGE L3
   ;instructions
   ;to be repeated
   INC R18
   RJMP L1
L3:
```

Integrated Development Environment (IDE) for AVR Microcontrollers

Old



Start

New Project...

New Example Project...

Open Project...

Recent

AssemblerApplication2

AssemblerApplication1

Discover Atmel Studio

Getting started with Atmel Studio

Getting started with AVR development

Open Atmel Start Configurator

Download Atmel Studio Extensions

Download documentation

New

`https://www.microchip.com/mplab/avr-support/atmel-studio-7`

Example: sum of odd integers

Write a program to calculate
1+3+5+…+19

```
sum=0,term=1;
for(i=1; i<=10; i++)
{
  sum += term;
  term += 2;
}
```

Example: sum of odd integers

Write a program to calculate
1+3+5+…+19

```
sum=0,term=1;
for(i=1; i<=10; i++)
{
  sum += term;
  term += 2;
}
```

```
.def sum = r16
.def term = r17
.def i = r18

      ldi sum,0
      ldi term,1
      ldi i,1
again:
      cpi i,11
      brge exit
      add sum,term
      inc term
      inc term
      rjmp again
exit:
      nop
```

Where is an error?

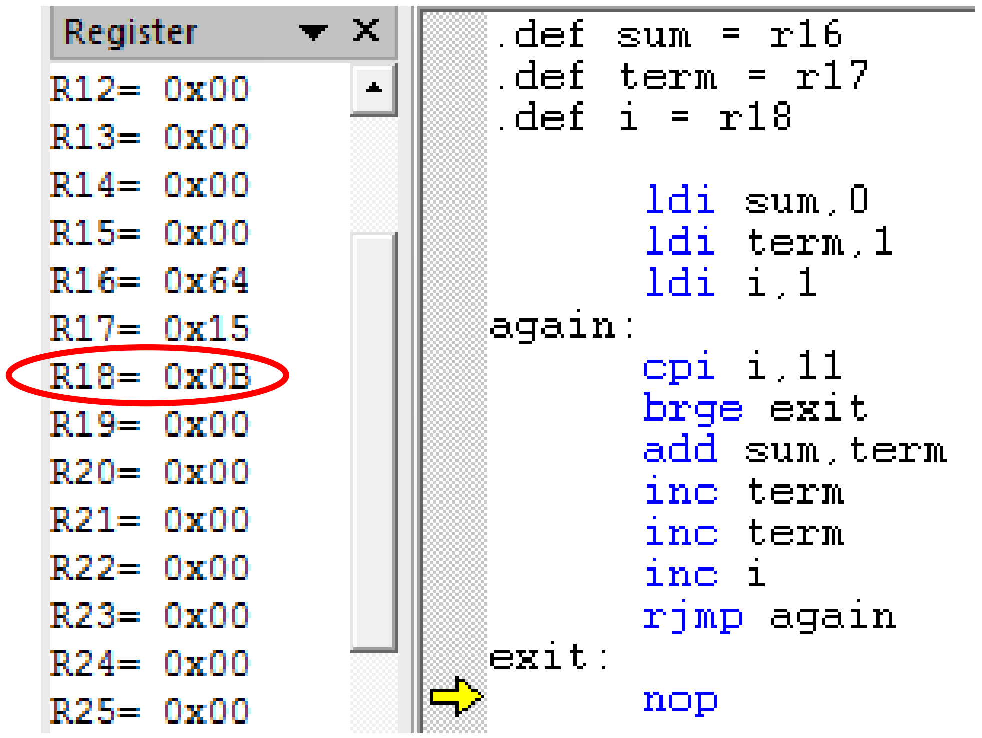
Example: sum of odd integers

Write a program to calculate 1+3+5+…+19

```
sum=0,term=1;
for(i=1; i<=10; i++)
{
  sum += term;
  term += 2;
}
```

```
.def sum = r16
.def term = r17
.def i = r18

      ldi sum,0
      ldi term,1
      ldi i,1
again:
      cpi i,11
      brge exit
      add sum,term
      inc term
      inc term
      inc i
      rjmp again
exit:
      nop
```


Register

```
R12= 0x00
R13= 0x00
R14= 0x00
R15= 0x00
R16= 0x64
R17= 0x15
R18= 0x0B
R19= 0x00
R20= 0x00
R21= 0x00
R22= 0x00
R23= 0x00
R24= 0x00
R25= 0x00
```

```
.def sum = r16
.def term = r17
.def i = r18

      ldi sum,0
      ldi term,1
      ldi i,1
again:
      cpi i,11
      brge exit
      add sum,term
      inc term
      inc term
      inc i
      rjmp again
exit:
      nop
```

Example: sum of alternating integers

Calculate -1+2-3+4-5+…-19+20

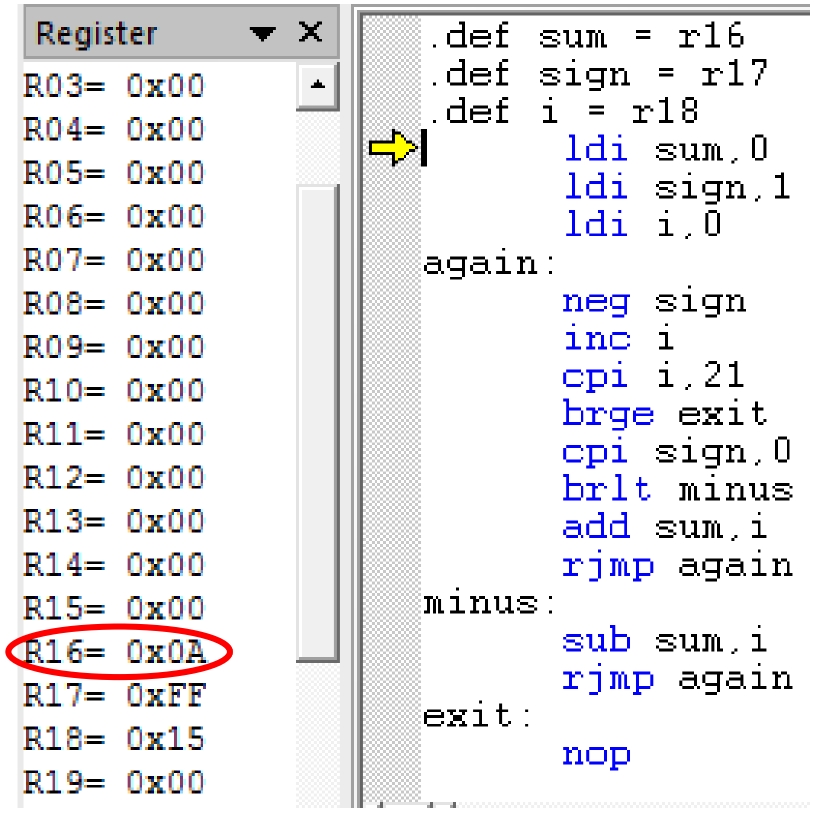
```
sum=0,sign=1;
for(i=1; i<=20; i++)
{
  sign = -sign;
  sum += sign*i;
}
```

Example: sum of alternating integers

Calculate -1+2-3+4-5+…-19+20

```
sum=0,sign=1;
for(i=1; i<=20; i++)
{
  sign = -sign;
  sum += sign*i;
}
```

```
.def sum = r16
.def sign = r17
.def i = r18
      ldi sum,0
      ldi sign,1
      ldi i,0
again:
      neg sign
      inc i
      cpi i,21
      brge exit
      cpi sign,0
      brlt ifminus
      add sum,i
      rjmp again
ifminus:
      sub sum,i
      rjmp again
exit:
      nop
```



```
R03= 0x00
R04= 0x00
R05= 0x00
R06= 0x00
R07= 0x00
R08= 0x00
R09= 0x00
R10= 0x00
R11= 0x00
R12= 0x00
R13= 0x00
R14= 0x00
R15= 0x00
R16= 0x0A
R17= 0xFF
R18= 0x15
R19= 0x00
```

```
.def sum = r16
.def sign = r17
.def i = r18
        ldi sum,0
        ldi sign,1
        ldi i,0
again:
        neg sign
        inc i
        cpi i,21
        brge exit
        cpi sign,0
        brlt minus
        add sum,i
        rjmp again
minus:
        sub sum,i
        rjmp again
exit:
        nop
```

Example: Fibonacci numbers

```
1,1,2,3,5,8,13,21,35,55,…
Fn = Fc + Fp
```

Leonardo Fibonacci
(born c. 1170, Pisa? - died after 1240)

Example: Fibonacci numbers

1,1,2,3,5,8,13,21,35,55,…

Fn = Fc + Fp

```
Fp=0, Fc=1, tmp=0;
for(i=1; i<10; i++)
{
  tmp = Fc;
  Fc += Fp;
  Fp = tmp;
}
```

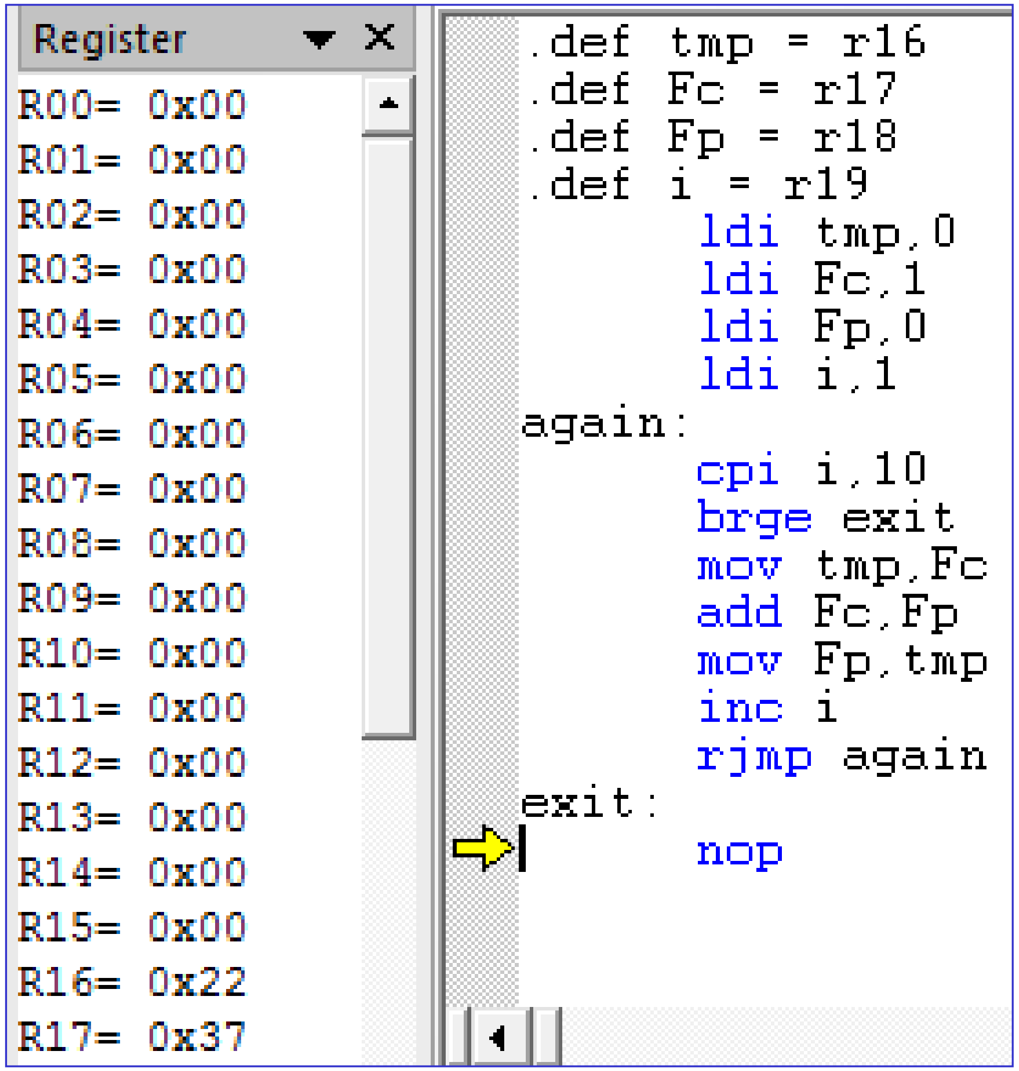
Leonardo Fibonacci
(born c. 1170, Pisa? - died after 1240)

Example: Fibonacci numbers

1,1,2,3,5,8,13,21,35,55,…

Fn = Fc + Fp

```
Fp=0, Fc=1, tmp=0;
for(i=1; i<10; i++)
{
   tmp = Fc;
   Fc += Fp;
   Fp = tmp;
}
```



Register

```
R00= 0x00
R01= 0x00
R02= 0x00
R03= 0x00
R04= 0x00
R05= 0x00
R06= 0x00
R07= 0x00
R08= 0x00
R09= 0x00
R10= 0x00
R11= 0x00
R12= 0x00
R13= 0x00
R14= 0x00
R15= 0x00
R16= 0x22
R17= 0x37
```

```
.def tmp = r16
.def Fc = r17
.def Fp = r18
.def i = r19
        ldi tmp,0
        ldi Fc,1
        ldi Fp,0
        ldi i,1
again:
        cpi i,10
        brge exit
        mov tmp,Fc
        add Fc,Fp
        mov Fp,tmp
        inc i
        rjmp again
exit:
        nop
```

Example: Tribonacci numbers

`0,1,1,2,4,7,13,24,44,81,149,…`

$F_{n+1} = F_n + F_{n-1} + F_{n-2}$

Example: Tribonacci numbers

0,1,1,2,4,7,13,24,44,81,149,…

$F_{n+1} = F_n + F_{n-1} + F_{n-2}$

```
Fn=1,Fnm1=1,Fnm2=0,N = 10;
for(i = 1; i < N-2; i++)
{
  tmp = Fn;
  Fn += (Fnm1+Fnm2);
  Fnm2 = Fnm1;
  Fnm1 = tmp;
}
```

Example: Tribonacci numbers

0,1,1,2,4,7,13,24,44,81,149,…

$F_{n+1} = F_n + F_{n-1} + F_{n-2}$

```
Fn=1,Fnm1=1,Fnm2=0,N = 10;
for(i = 1; i < N-2; i++)
{
  tmp = Fn;
  Fn += (Fnm1+Fnm2);
  Fnm2 = Fnm1;
  Fnm1 = tmp;
}
```

$\Rightarrow$

```
.def Fn = r16
.def Fnm1 = r17
.def Fnm2 = r18
.def i = r19
.def tmp = r20
      ldi Fn,1
      ldi Fnm1,1
      ldi Fnm2,0
      ldi i,1
again:
      cpi i,8
      brge exit
      mov tmp,Fn
      add Fn,Fnm1
      add Fn,Fnm2
      mov Fnm2,Fnm1
      mov Fnm1,tmp
      inc i
      rjmp again
exit:
      rjmp exit
```

Macroinstructions

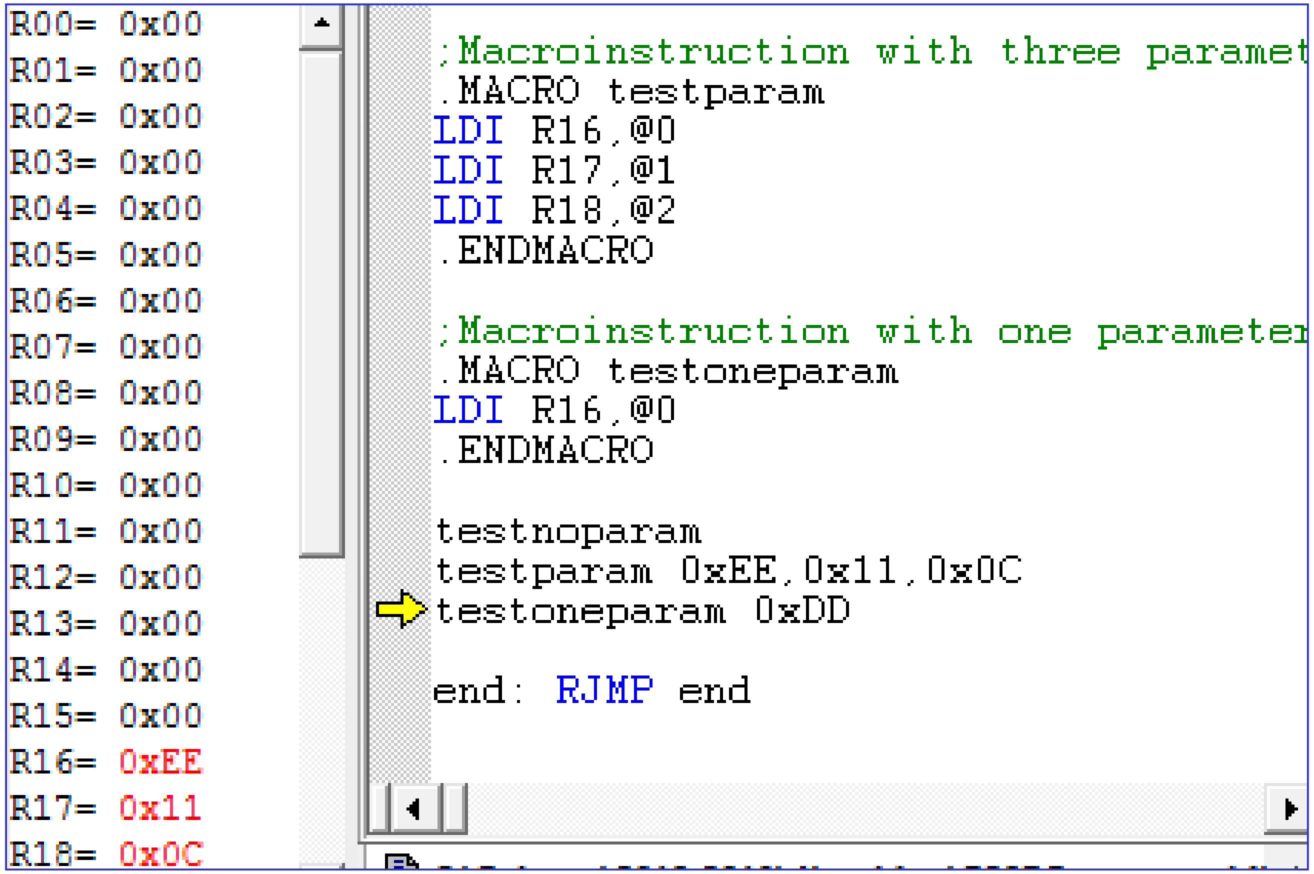
```
;without parameters
.MACRO testnoparam
LDI R16,0xFF
LDI R17,0x01
LDI R18,0x0D
.ENDMACRO

;with three parameters
.MACRO testparam
LDI R16,@0
LDI R17,@1
LDI R18,@2
.ENDMACRO

;with one parameter
.MACRO testoneparam
LDI R16,@0
.ENDMACRO

testnoparam
testparam 0xEE,0x11,0x0C
testoneparam 0xDD

end: RJMP end
```



```
R00= 0x00
R01= 0x00
R02= 0x00
R03= 0x00
R04= 0x00
R05= 0x00
R06= 0x00
R07= 0x00
R08= 0x00
R09= 0x00
R10= 0x00
R11= 0x00
R12= 0x00
R13= 0x00
R14= 0x00
R15= 0x00
R16= 0xEE
R17= 0x11
R18= 0x0C
```

```
;Macroinstruction with three paramet
.MACRO testparam
LDI R16,@0
LDI R17,@1
LDI R18,@2
.ENDMACRO

;Macroinstruction with one parameter
.MACRO testoneparam
LDI R16,@0
.ENDMACRO

testnoparam
testparam 0xEE,0x11,0x0C
testoneparam 0xDD

end: RJMP end
```

Assembly vs. C

- Programming in assembly is tedious

- Programming in C
  - is much faster
  - is easier to modify
  - is portable

- However the assembly language produces binary programs that
  - are much smaller than their C counterparts (if written by a skilful assembly programmer)
  - can run faster (if written by a skilful assembly programmer)

- Today C is a standard in Embedded System programming