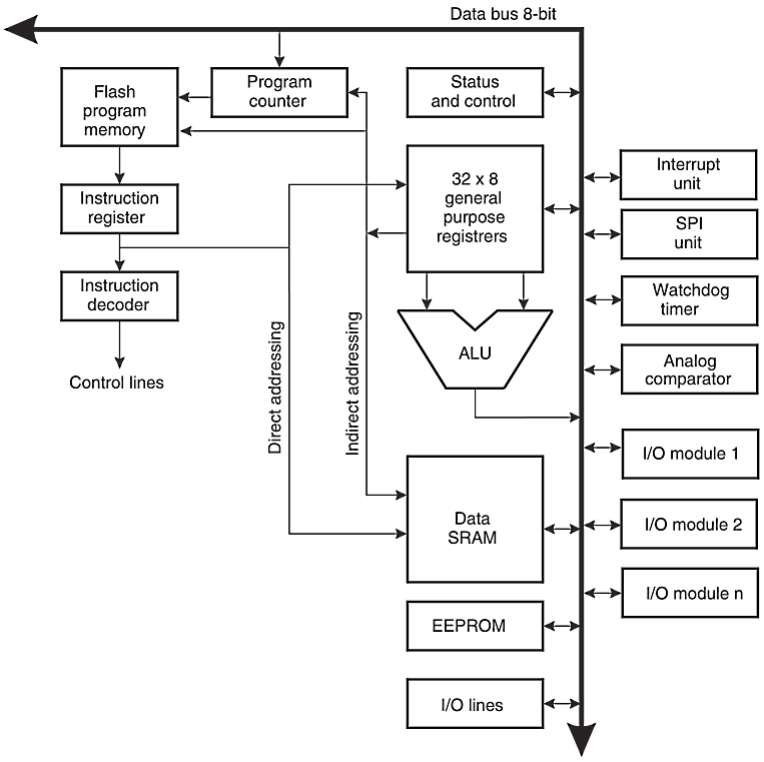
Mikroprosessorsystemer

Assembly.

C-kode er et høynivåspråk, mens assembly er lav-nivå. Når vi kompilerer C så blir det assemblykode, som oversettes direkte til maskinkoden en mikroprosessor forstår.

For å forstå assembly er det nyttig å vite hvordan kjernen til mikroprosessoren ser ut:



Atmel AVR er en såkalt harvardarkitektur siden den har separat data- og instruksjonsbuss. De 32-generelle registrene har aksess til ALU (Aritmetisk Logisk Enhet).

Minne:
Programminne – Flash. Ikke volatil – dvs. beholder innholdet ved strømbrudd. Inneholder operasjonskoder (programmet) og konstanter. Må skrives sidevis og ikke enkeltbytes. 10 000 slette/skrive-sykler
EEPORM – Ikke volatil. Inneholder parametere og statusdata. Kan skrives en og en byte via I/O-registre. Tregt. 100 000 slette/skrive-sykler
SRAM – Volatil. Inneholder 32 generelle registre som brukes ved aksess til ALU (Aritmetisk Logisk Enhet), I/O-registre (standard og extended), samt SRAM (variabler i C og stakken helt sist).

Dataoverføringsintruksjoner (Data transfer Instructions).
Assembly består av en god del flytting av data dit de skal brukes.
Addresseringstyper:

```
        ldi r16,199 //Immediate. last verdien 199 inn i R16
        mov r0,r16  //Register Direct. kopier innholdet i R16 til R0
        in r16,SREG  //I/O Direct. kopier innholdet i Statusregisteret til R16.
        lds r1, SRAM_START // Data Direct. SRAM_START = 0x0100. verdien i første SRAM lokasjon
-> R1.
        ld r16,Z  //Data Dindirect. last verdien i minneposisjonen Z peker på inn i R16
        st Z+,r16  //Data Indirect. lagre verdien i R16 på adressen til Z-pekeren,
postinkrement av Z.
```

Stakken (Stack).
Sett opp stackpekeren til å peke på siste verdi i SRAM. Ikke nødvendig på nyere AVR.

```
        ldi r16, LOW(RAMEND)
        out SPL, r16
        ldi r16, HIGH(RAMEND)
        out SPH, r16
```

Stakken brukes ofte til å lagre unna statusregisteret og andre registre som blir brukt. Eksempel som initialiserer R16 til 0xAA, lagrer unna R16 og SREG, sletter R16 og så gjenoppretter både SREG og R16

```
        ldi  r16,0xAA
        push r16
        in   r16,SREG
        push r16
        clr  r16
        out  SREG,r16
        pop  r16
        out  SREG,r16
        pop  r16
```

Aritmetiske og logiske instruksjoner
Addisjon, subtraksjon, eller, osv. Eksempel: legg sammen r7 og r16:

```
        add r7,r16
```

Branches
Branches er hopp som bestemmer programflyten. De kan tilføre programmet intelligens ved at forskjellig kode gjøres avhengig av resultater/målinger/input.

Ubetinget hopp. Eksempel: relative Program Addressing.

```
        endless_loop:
        //
              rjmp endless_loop
```

Betinget hopp. I AVR instruksjonssettet finns to typer:
Hopper over neste instruksjon hvis betingelse er gyldig, eksempel:

```
        sbis PINB,PINB0 // hopper hvis PB0 er høy
```

Hopper hvis betingelse er gyldig.

```
        loop:
              dec r16
              brne loop:  // hopp hvis ikke r16 ble null
        // ferdig med løkka
```

Brukes til if/for/while.

Bit og bit-test
Roter venstre/høyre, skift venstre/høyre, samt setting/sletting av bit i SREG og noen I/O registre, med mer.

MCU kontroll instruksjoner
nop – gjøre ingen ting, bare venter en klokkesyklus.

Instruksjonskoder
(Tall)verdien som blir lagret i programminnet og tolket som instruksjon av CPU. Se AVR instruction set for detaljer. Eksempler:

LDI R16, 0x31
`R0 <- 0x31`

1110	KKKK	dddd	KKKK
1110	0011	0000	0001
E	3	0	1

ST X+, R16
`(X) <- R7, X <- X+1`

1001	001r	rrrr	1101
1001	0011	0000	1101
9	3	0	D

Inline assembly
Hvis ikke C-koden resulterer i god nok assemblerkode så kan man skrive inline assembly som beskrevet i AVR Libc manualen:
http://www.nongnu.org/avr-libc/user-manual/inline_asm.html
Dette gjelder spesielt ved tidskritiske applikasjoner.