Indføring til assembler programmering:

Med udgangspunkt I et lille C-program, så lader vi compileren oversætte det til Assembler.

#define F\_CPU 16000000UL

#include <avr/io.h>

#include <avr/delay.h>

int main(void)

{

DDRB = (1<<DDB5); // Set PortB bit5 som udgang.

PORTB = 0; // Set porten til 0

while(1) // loop for altid

{

PORTB |= (1<<PORTB5); // Set bit5 på portB til 1 => 2^5 = 0x20=32

\_delay\_ms(100); // vent en tid

PORTB &= ~(1<<PORTB5); // 0-stil bit 5 på portB. And med 10111111

\_delay\_ms(10); // vent en anden tid

//TODO:: Please write your application code

}

}

**Programmet vises her oversat til en AtMega168 processor.**

**Her startes med at lave jump-vector tabellen, både for reset og alle de andre interrupt muligheder. Interrupt kan afbryde det kørende program eks. While(1) i main.**

**00000000 <\_\_vectors>:**

0: 0c 94 34 00 jmp 0x68 ; 0x68 <\_\_ctors\_end> 🡨- Hop til adr. 0x68

4: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

8: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

c: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

10: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

14: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

18: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

1c: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

20: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

24: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

28: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

2c: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

30: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

34: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

38: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

3c: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

40: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

44: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

48: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

4c: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

50: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

54: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

58: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

5c: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

60: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

64: 0c 94 3e 00 jmp 0x7c ; 0x7c <\_\_bad\_interrupt>

Her laver kompileren en basisk opsætning af porte, ikke relevant i forhold til vores program, så det accepteres bare, as is… Området kan initialisere data regioner, eks. Hvis man skriver:

int Data1 = 0;

main { …

så skal Data1 0-stilles før main udføres.

00000068 <\_\_ctors\_end>:

68: 11 24 eor r1, r1 🡨 Her 0-stilles Register R1, ved brug af xor.

6a: 1f be out 0x3f, r1 ; 63

6c: cf ef ldi r28, 0xFF ; 255

6e: d4 e0 ldi r29, 0x04 ; 4

70: de bf out 0x3e, r29 ; 62

72: cd bf out 0x3d, r28 ; 61

74: 0e 94 40 00 call 0x80 ; 0x80 <main> 🡨 Kald main.

78: 0c 94 55 00 jmp 0xaa ; 0xaa <\_exit>

Hvis uinitiliserede interrupts skulle blive udført, så reset CPU.

0000007c <\_\_bad\_interrupt>:

7c: 0c 94 00 00 jmp 0 ; 0x0 <\_\_vectors>

Her begynder vores main-program

00000080 <main>:

#include <avr/io.h>

#include <avr/delay.h>

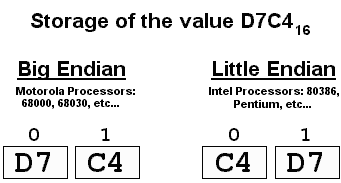
int main(void)

{

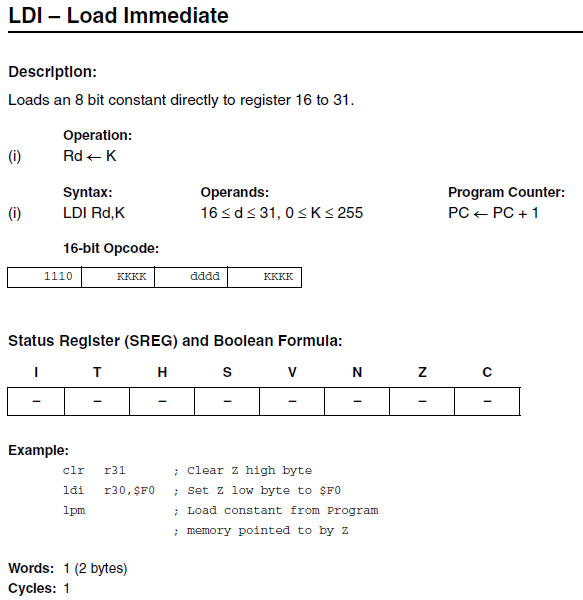
DDRB = (1<<DDB5); 🡨--------- Kompileren viser hvad den oversætter fra c-coden

80: 80 e2 ldi r24, 0x20 ; 32 🡨---- Her omsat til assembler men også adresse + opcode. Adressen er 0x80 mens opcode er 0xe280.  
Opcoden er 2 bytes stor, så den næste assembler kode vil findes på adresse 0x80.  
Opcoden e280, skal læses med de 2 bytes byttet rundt, dette hedder little endian.

Her vist som et eksempel:



Opcode e280 findes i dokumentet AVR-8bit-instruktionsset: <http://www.atmel.com/images/doc0856.pdf>



Her kan vi se start koden e 2 8 0

Omsat til binært tal 1110 0010 1000 0000  
Her af ses: kkkk dddd kkkk =>

dddd=8, hvilket er registeret ofsset med 16 giver dette Register 24, (8+16).  
kkkkkkkk = 00100000 hvilket er 0x20 = 32.

Dvs. load register 24 med værdien 32, sæt ingen bit i Status registeret.  
Programcounter står på 0x81 og forøges til 0x82 som vil være start på næste instruktion.

82: 84 b9 out 0x04, r24 ; 4 🡨 put indeholdet af R24 til IO-reg 0x04  
IO-reg 0x04 er Directional Registeret for PortB, dvs. DDRB.

PORTB = 0;

84: 15 b8 out 0x05, r1 ; 5 🡨Husk R1 er sat til 0

while(1)

{

PORTB |= (1<<PORTB5);

86: 2d 9a sbi 0x05, 5 ; 5 🡨 Set bit 5 i IO-reg 5(PortB), smart!!!

#else

//round up by default

\_\_ticks\_dc = (uint32\_t)(ceil(fabs(\_\_tmp)));

#endif

\_\_builtin\_avr\_delay\_cycles(\_\_ticks\_dc);

88: 2f ef ldi r18, 0xFF ; 255

8a: 81 ee ldi r24, 0xE1 ; 225

8c: 94 e0 ldi r25, 0x04 ; 4

8e: 21 50 subi r18, 0x01 ; 1

90: 80 40 sbci r24, 0x00 ; 0

92: 90 40 sbci r25, 0x00 ; 0

94: e1 f7 brne .-8 ; 0x8e <main+0xe>

96: 00 c0 rjmp .+0 ; 0x98 <main+0x18>

98: 00 00 nop

\_delay\_ms(100);

PORTB &= ~(1<<PORTB5);

9a: 2d 98 cbi 0x05, 5 ; 5

9c: 8f e3 ldi r24, 0x3F ; 63

9e: 9c e9 ldi r25, 0x9C ; 156

a0: 01 97 sbiw r24, 0x01 ; 1

a2: f1 f7 brne .-4 ; 0xa0 <main+0x20>

a4: 00 c0 rjmp .+0 ; 0xa6 <main+0x26>

a6: 00 00 nop

a8: ee cf rjmp .-36 ; 0x86 <main+0x6>

000000aa <\_exit>:

aa: f8 94 cli

000000ac <\_\_stop\_program>:

ac: ff cf rjmp .-2 ; 0xac <\_\_stop\_program>

Delay – hvordan ved kompileren at den skal ramme 100ms og 10ms, vi laver nogle loops i hinanden.  
Hver tæller fra et tal ned til 0, hvorefter de laves ”underflow” og sætter Carry-flaget for at have lånt et bit.  
Den mellemste tæller, tæller altså 225..0, 255(-1) .. 0, 255(-1)… osv.  
Derfor udføres loopen 1\*226\*256+  
 + 4\*256\*256 gange = 320.000 gange

Loopen består af de 3 subtraktioner og en branch-test og looper så igen:  
Subi => 1 clock cycle  
sbci => 1 clock cycle  
sbci => 1 clock cycle  
Brne => 2 clock cycles indtil konditionen er mødt, så koster det kun 1 instruktion.  
Dvs. hvert gennemløb tager 5 clock-cycles.

Der udføres 1 instruktion pr. klok-cyclus, dvs. 16.000.000 gange i sekundet, derfor tager vores loop:

5\*320000/16000000 = 0.1s  
Men, så er der ikke taget højde for 3\*ldi-instruktioner til at loade data til tællerne med, samt at der i slutningen af loopen jo er en enkelt instruktion når der ikke loopes + rjmp på 2 clock samt en nop på en instruktion, så dette giver en fejl, ikke ret stor, men alligevel en fejl på 7 clock-pulser pr. delay.

Lav samme øvelse, men skriv nu 2 bit I port B, og vurder forskellen.