AVR Assembler

Özgun Mirtchev

February 25, 2016

Laborationsrapport för kursen Datorteknik 2016 VT

Universitetsadjunkt Kjell Mårdensjö

Örebro Universitet

Contents

1 Laboration 1: Lysdioder & Strömställare

I kommande koduppvisning kommer läsaren att märka främst två typer av funktioner, init_func och loop_func, som är generella namn på två funktioner som används i assembly-koden. Notera att dessa namn får och kan ändras till ens eget tycke men i denna laboration var funktionsnamnen oförändrade. C-koden ser ut som på nedanstående sätt och står för grunden av assembly-programmeringen. Istället för att skriva funktionerna i vanlig C ska dessa skrivas i assembly-kod som står med i slutet av respektive uppgift.

```
#include <avr/io.h>

int main(void)
{
    init_func();

    while (1
    {
        loop_func();
        wait_milliseconds(300);
    }
}
```

I varje laboration kommer dessutom att fördefinierade värden för portarna och ingångarna att finnas:

```
;;;--- I/O-adresses for Port D ---
#define PIND    0x10
#define DDRD    0x11
#define PORTD   0x12

;;;--- I/O-adresses for Port C ---
#define PINC    0x13
#define DDRC    0x14
#define PORTC   0x15

;;;--- I/O-adresses for Port B ---
#define PINB    0x16
#define DDRB    0x17
#define PORTB   0x18

;;;--- I/O-adresses for Port A ---
#define PINA    0x19
#define DDRA    0x1A
#define PORTA   0x1B
```

1.1 Programmet Hello World

Denna uppgift var främst för att prova lite lätt assembly och därför finns ingen redovisning.

1.2 Assemblerprogram för att tända och släcka en lysdiod med en strömbrytare

Denna uppgift gick ut på att kunna tända en diod genom att trycka in respektive diod-knapp.

Algorithm 1 Tända/släcka en lysdiod med strömbrytare

```
1         .data
2
3  lamps:  .byte 0 ;; unsigned char lamps = 0;
4
5         .text
6         .global init_func
7
8  init_func:
9
10        ;; DDRA = 0x00
11        LDI     R20, 0x00
12        OUT     DDRA, R20
13
14        ;; DDRB = 0xFF;
15        LDI     R20, 0xFF
16        OUT     DDRB, R20
17
18        RET
19
20
21        .text
22        .global loop_func
23
24 loop_func:
25
26        ;; PORTB = PINA
27        IN      R20, PINA
28        OUT     PORTB, R20
29
30        RET
```

1.3 Assemblerprogram för att tända och släcka en lysdiod med en switch

Denna uppgift är har samma förutgrunder som föregående däremot är skillnaden att en switch används för att tända en lysdiod på ATMega32. När tredje switchen aktiveras lyser den första biten i registret som styr lysdioderna.

Algorithm 2 Tända/släcka en lysdiod med switch

```
1  lamps:  .byte 0      ;;   unsigned char lamps = 0;
2
3          .text
4          .global init_func
5
6  init_func:
7          ;; DDRA = 0x00
8          LDI     R20, 0x00
9          OUT     DDRA, R20
10
11         ;; DDRB = 0xFF;
12         LDI     R20, 0xFF
13         OUT     DDRB, R20
14
15         ;; PORTB = 0xFF;
16         OUT     PORTB, R20
17
18         RET
19
20         .text
21         .global loop_func
22
23 loop_func:
24         ;; R20 = PINA & 0x08
25         IN R20, PINA
26         COM R20
27         ANDI R20, 0b00001000
28
29         ;; R20 = R20 >> 3
30         LSR R20
31         LSR R20
32         LSR R20
33
34         ;; R21 = PORTB & 0xFE
35         IN R21, PORTB
36         COM R21
37         ANDI R21, 0xFE
38         OR R20, R21      ; R20 = R20 | R21
39
40         COM R20
41         OUT PORTB, R20   ; PORTB = R20
42
43         RET
```

1.4 Ringräknare

Denna uppgift gick ut på att implementera en ringräknare. Processen för att skapa programmet följdes genom en figur från kurskompendiet som visar stegen för en lösning.

Som det syns i ovanstående figur finns det olika steg man kan följa. Notera att det finns alternativa lösningar till denna uppgift men denna valdes för att lättare kunna designa koden för ringräknaren. I denna lösning används processorns register medan de andra respresenterar lösningar som involverar en mönstertabell eller en if-else-sats. För enkelhetens skull gjordes denna uppgift med en lösning som använder processorns register genom T-bit utilisering.

Stegen som följdes demonstreras i nedanstående steg med tillhörande assembly-

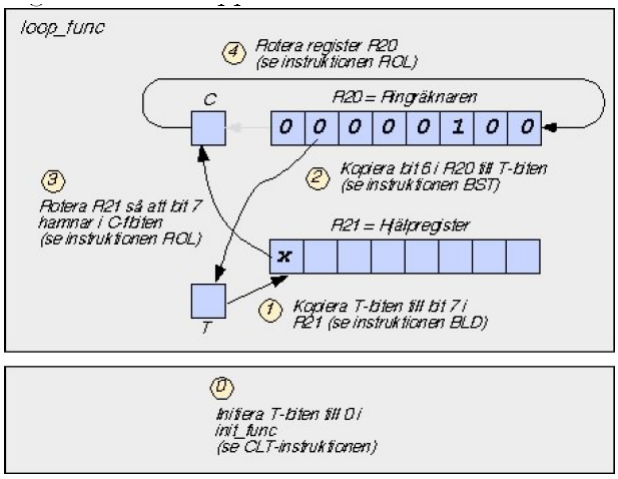

Figure 1: Ringräknare lösning genom register-användning

kod i algoritm-figuren.

1. I första steget initierades T-biten till 0 genom att använda CLT-instruktionen.
2. Andra steget inkluderade att kopiera T-biten till bit 7 i processorns R21-register som är användes som ett hjälpregister genom att använda instruktionen BLD.
3. Samma sak utfördes på registret R20, men i detta fall kopierades bit 6 till T-biten. R20 är ringräknaren, alltså registret som aktiverar lysdioderna.
4. Eftersom R21 fick bit 7 så roterades den så att den hamnar i C-biten som i nästa steg roterades in i ringräknaren-registret R20, båda använde instruktionen ROL.

Algorithm 3 Ringräknare genom registeranvändning

```
1         .data
2
3         .text
4         .global init_func
5
6   init_func:
7
8         LDI R20, 0xFF
9         OUT DDRB, R20
10        CLT
11
12        RET
13
14        .text
15        .global loop_func
16
17  loop_func:
18
19        BLD R21, 7
20        BST R20, 6
21        ROL R21
22        ROL R20
23        OUT PORTB, R20
24
25        RET
```

1.5 Johnsonräknare

1.5.1 Kommentar

Koden som beskrivs nedan är i princip identisk mot ringräknar koden. Den enda skillnaden är att ett register inverteras i johnsonräknaren så att det fortsätter lysa även efter dioden har flyttat sig. I ringräknaren behövdes inte det eftersom det bara var en diod som skulle förflytta sig hela tiden.

1.5.2 Kod

Algorithm 4 Johnsonräknare genom registeranvändning

```
1         .data
2
3         .text
4         .global init_func
5
6   init_func:
7
8         LDI R20, 0xFF
9         OUT DDRB, R20
10        CLT
11
12        RET
13
14        .text
15        .global loop_func
16
17  loop_func:
18
19        BLD     R21, 7
20        BST     R20, 6
21        COM     R21 ;; Detta steg gör ringräknaren till en ↩
              johnsonräknare
22        ROL     R21
23        ROL     R20
24        OUT PORTB, R20
25
26        RET
```

Algorithm 5 Johnsonräknare genom if-else-användning

```
#define vjohn R20
#define one R21
#define vjohn_and R22
vjohn_temp: .byte 0

        .text
        .global init_func

init_func:
        LDI vjohn, 0xFF
        OUT DDRB, vjohn
        LDI vjohn, vjohn_temp    ;; R20 = vjohn <=> R20 = 0
        LDI one, 1   ;; R21 = 1
        RET

        .text
        .global loop_func

loop_func:
        LDS vjohn,vjohn_temp     ;;R20=vjohn
        MOV vjohn_and, vjohn     ;; R22 = R20
        ANDI    vjohn_and, 0x80 ;; R22 & 0x80
        BREQ    if_equal
        RJMP    else_f

if_equal:
        LSL vjohn    ;; R20 << 1
        ADD vjohn, one   ;; R20 += R21
        RJMP    end_loop

else_f:
        LSL vjohn    ;; R20 << 1
        RJMP    end_loop

end_loop:
        STS     vjohn_temp, vjohn
        COM     vjohn    ;; Inverterar för lysdioderna
        OUT     PORTB, vjohn
        RET
```

1.6 Lysdiodsmönster

I denna uppgift skulle ett mönster skapas. Mönstret i fråga ser ut som en johnsonräknare som i cykler räknar inåt och sedan utåt. En representation kan fås genom nedanstående figur 2:

```
00000000
10000001
11000011
11100111
11111111
11100111
11000011
10000001
```

Figure 2: Lysdiodsmönster

För uppgiften användes en lista för att hålla reda på vilka lysdioder som ska lysa. Loop-funktionen gick igenom en for-loop liknande sats som aktiverade respektive position i listan.

Algorithm 6 Lysdiodsmönster genom tabell

```
1         .data
2         .global LEDS
3
4  LEDS: .byte 0x00, 0x81, 0xC3, 0xE7, 0xFF, 0xE7, 0xC3, 0x81
5
6         .text
7         .global init_func
8
9  init_func:
10        ;; i = 0
11        LDI R28, 0x00
12        LDI R29, 0x00
13
14        ;; DDRB = 0xFF
15        LDI R24, 0xFF
16        OUT DDRB, R24
17
18        RET
19
20
21        .text
22        .global loop_func
23
24 loop_func:
25        ;; PORTB = LEDS[i]
26        LDI R30, lo8(LEDS)
27        LDI R31, hi8(LEDS)
28
29        ;; LEDS + i - R31:R30 + R29:R28
30        ADD R30, R28
31        ADC R31, R29
32
33        LD  R24, Z       ;; Z = R31:R30
34        /*COM R24*/ ;;ignore
35        OUT PORTB, R24
36        ADIW R28, 0x01   ;; i += 1
37
38        ;; i = i & 0x07
39        ANDI R28, 0x07
40        ANDI R29, 0x00
41
42        RET
```

2 Laboration 2: Subrutiner och aritmetik

2.1 Villkorliga programsatser i assembler

Algorithm 7 Villkorliga programsatser main-loop

```
    .data

#define vjohn R20
#define vjohn_and R21
#define vring R22
vjohn_temp: .byte 0

    .text
    .global init_func
init_func:

    LDI R18, 0xFF
    OUT DDRB, R18
    LDI R17, 0x00
    OUT DDRA, R17
    LDI vjohn, vjohn_temp
    RET

    .text
    .global loop_func
loop_func:

    IN vjohn, PINA
    MOV R19, vjohn
    ANDI R19, 0x01
    BREQ ring_f
    RJMP john_f
```

Algorithm 8 Villkorliga programsatser (Ring och johnräknare-funktioner)

```
ring_f:

    LSL vring        ;vring
    CPI vring, 0x00
    BREQ ring_eq
    RJMP ring_end

ring_eq:

    LDI vring, 0x01
    RJMP ring_end

ring_end:

    COM vring
    OUT PORTB, vring
    COM vring
    RET

john_f:

    LDS vjohn, vjohn_temp
    MOV vjohn_and, vjohn
    ANDI vjohn_and, 0x80
    BREQ john_if
    RJMP john_else

john_if:

    LSL vjohn
    INC vjohn
    RJMP john_end

john_else:

    LSL vjohn
    RJMP john_end

john_end:

    STS vjohn_temp, vjohn
    COM vjohn
    OUT PORTB, vjohn
    RET
```

2.2 Elektronisk tärning

Algorithm 9 Elektronisk tärning

```
    .data

pattern:
    .byte 0x10, 0x82, 0x92, 0xC6, 0xD6, 0xEE

    .text
    .global init_func
init_func:

    LDI R20, 0xFF
    OUT DDRB, R20
    LDI R21, 0x00
    OUT DDRA, R21
    CLR R20
    RET

    .text
    .global loop_func
loop_func:

    IN R21, PINA
    ANDI R21, 0x01
    BREQ dice_update
    RJMP dice_end

dice_update:

    INC R20 ; counter++
    CPI R20, 0x06
    BREQ dice_if
    RJMP dice_end

dice_if:

    LDI R20, 0x00
    RJMP dice_end

dice_end:

    LDI R30, lo8(pattern)
    LDI R31, hi8(pattern)
    ADD R30, R20
    LD R24, Z
    COM R24
    OUT PORTB, R24
    RET
```

2.3 Förändringsräknare

Algorithm 10 Förändringsräknare

```
    .data

#define oldValue R22
#define newValue R23
counter: .byte 0

    .text
    .global init_func
init_func:

    LDI R20, 0xFF
    OUT DDRB, R20
    LDI R21, 0x00
    OUT DDRA, R21
    RET

    .text
    .global loop_func
loop_func:

    MOV oldValue, newValue
    /*IN R19, PINA
    ANDI R19, 0x01*/
    EOR R19, oldValue    ;För automatisk räkning utan intryckning
    MOV newValue, R19
    CP oldValue, newValue
    BRNE loop_if
    RJMP loop_end

loop_if:

    LDS R18, counter
    INC R18
    STS counter, R18
    RJMP loop_end

loop_end:

    COM R18
    OUT PORTB, R18
    CLR R18
    RET
```

2.4 Simulerad aritmetisk enhet (AU)

Algorithm 11 Simulerings-kod

```
        .data

#define valueX R20
#define valueY R21

        .text
        .global init_func
init_func:

        LDI R23, 0xFF
        OUT DDRB, R23
        LDI R23, 0x00
        OUT DDRA, R23
        LDI R23, 0x00
        OUT DDRD, R23
        RET

        .text
        .global loop_func
loop_func:

        IN R22, PINA
        COM  R22
        ANDI R22, 0xF0
        MOV valueX, R22
        IN R22, PINA
        COM R22
        LSL R22
        LSL R22
        LSL R22
        LSL R22
        MOV valueY, R22
        IN R23, PIND
        ANDI  R23, 0x01
        CPI R23, 0x01
        BRNE Subtract_func
        RJMP Add_func

Subtract_func:

        SUB valueX, valueY
        IN  R22, SREG
        COM valueX
        OUT PORTB, valueX
        RET

Add_func:

        ADD valueX, valueY
        ANDI valueX , 0xF0
        IN  R22, SREG
        ANDI R22, 0x0F
        OR valueX,R22
        COM valueX
        OUT PORTB, valueX
        RET
```

2.5 Några loopar

2.5.1 7-räknare

Tanken med denna uppgift var att räkning skulle ske genom varje flank-detektering. Vid nedaktivering av en knapp för negativ flank och uppaktivering av en knapp för positiv flank. I nedastående kod räknar det däremot automatisk med hjälp av instruktionen EOR som har samma funktion som en XOR-operation.

Algorithm 12 7-räknare

```
    .data

varX: .byte 0

    .text
    .global init_func

init_func:
    LDI R20, 0xFF
    OUT DDRB, R20
    RET

    .text
    .global loop_func

loop_func:
    CALL seven
    RET

seven:
    LDS R21, varX
    INC R21
    STS varX, R21

    CPI R21, 0x08
    BREQ reset

    LDI R24, 0
    LDI R25, 1
    CALL wait_milliseconds

    COM R21
    OUT PORTB, R21
    COM R21

    RJMP seven

reset:
    LDI R21, 0x00
    STS varX, R21
    CLR R21
    RET
```

2.5.2 7- och 30-räknare

Denna kod har lånat seven-funktionen som finns från förra deluppgiften och dessutom har en thirty-funktion lagts till som räknar ned från 30 efter att seven har körts tre gånger.

Algorithm 13 7 och 30-räknare

```
    .data
varX: .byte 0
varY: .byte 0
    .text
    .global init_func
init_func:
    LDI R20, 0xFF
    OUT DDRB, R20
    RET
    .text
    .global loop_func
loop_func:
    CALL seven
    CALL seven
    CALL seven
    CALL thirty
    RET
seven:
    LDS R21, varX
    INC R21
    STS varX, R21
    ; varX = 7 ? reset : seven
    CPI R21, 0x08
    BREQ reset
    CALL display
    RJMP seven
thirty:
    LDS R21, varY
    DEC R21
    DEC R21
    STS varY, R21
    ; varY = 0 ? reset : thirty
    CPI R21, 0x00
    BREQ reset
    CALL display
    RJMP thirty
reset:
    LDI R21, 0x00
    STS varX, R21
    LDI R21, 0x1E
    STS varY, R21
    CLR R21
    RET
display:
    ; wait_milliseconds ~0.25s
    LDI R24, 0
    LDI R25, 1
    CALL wait_milliseconds
    COM R21
    OUT PORTB, R21
    COM R21
    RET
```

3 Laboration 3: Hårdvarunära programmering i C

I denna laboration kommer koden att främst skrivas i C. Många av laborationerna är nästan direkta översättningar från assembler-uppgifterna till C från förra laborationer.

3.1 C-program för att tända och släcka en lysdiod

Algorithm 14 Tända/släcka en lysdiod C

```
int main(void)
{
    DDRA = 0x00;
    DDRB = 0xFF;

    while (1)
    {
        PORTB = PINA;
    }
}
```

3.2 Ringräknare i C

I uppgiften ska en ringräknare konstrueras som stannar vid knapptryckning (sw3) och fortsätter när man trycker igen.

Algorithm 15 Ringräknare C

```c
#include <avr/io.h>

void wait_millseconds(int milliseconds);

int main(void)
{
    DDRA = 0x00;
    DDRB = 0xFF;

    char leds[8] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80};
    unsigned char new_sw3, old_sw3, run;
    int i = 0;

    new_sw3 = old_sw3 = run = 1;

    while (1)
    {
        // switch
        old_sw3 = new_sw3;
        new_sw3 = (PINA & 0x08) == 0;
        if (old_sw3 && !new_sw3)
        {
            run = !run;
        }

        if(run)
        {
                PORTB = ~leds[i];
                i++;
                i = i & 0x07;
        }
        else
        {
            PORTB = 0xFF; //Släcka diod vid !run
        }
        wait_milliseconds(100);
    }
}
```

3.3 Spegling av en switchs värde på en LCD-display

Denna uppgift gick ut på att visa värden av den intryckta switchen på en LCD-display (exempel figur 3) . Koden blev uppdelade i funktioner för att göra det tydligare vad varje sektion av kod gör.

Figure 3: LCD-skärm för ATmega32

Algorithm 16 Main-funktion LCD

```
lcd4 theDisplay;

void dec_to_bin(int sw_value, char * sw_bin);

int main(void)
{
    DDRA = 0x00;
    lcd4_init(&theDisplay, &PORTB, &DDRB, 4000, 100);

    int sw_value = 0;
    char s[20];

    while (1)
    {
        sw_value = PINA;
        print_to_lcd(s, sw_value);
    }
    return 0;
}
```

Algorithm 17 Funktioner för LCD

```
void print_to_lcd(char s[], int sw_value)
{
    // PRINT HEX
    sprintf(s, "HEX = %02x", 0xFF - sw_value);
    lcd4_cup_row1(&theDisplay);
    lcd4_write_string(&theDisplay, s);

    // CALCULATE BINARY
    char sw_bin[9];
    dec_to_bin(sw_value, sw_bin);

    // PRINT BINARY
    sprintf(s,"BIN = %s", sw_bin);
    lcd4_cup_row2(&theDisplay);
    lcd4_write_string(&theDisplay, s);

    for (int i = 0; i <= 8; i++)
    {
        sw_bin[i] = 48;
    }
}

void dec_to_bin(int sw_value, char * sw_bin)
{
    int dec, i = 0;
    dec = 255 - sw_value;
    while(i < 8)
    {
        sw_bin[7-i] = dec % 2 + '0';
        dec /= 2;
        i++;
    }
    sw_bin[8] = '\0';
}
```

3.4 Elektronisk tärning med presentation på en LCD-display

Algorithm 18 Elektronisk tärning på LCD

```
#include <avr/io.h>
#include <stdio.h>
#include <util/delay.h>
#include "lcd4.h"

lcd4 theDisplay;

const int DICE_MAX = 6;

int main(void)
{
    DDRA = 0x00;
    lcd4_init(&theDisplay, &PORTB, &DDRB, 4000, 100);

    int sw_button = 0;

    while (1)
    {
        sw_button = (PINA & 0x01) == 0;
        if(sw_button)
            print_to_lcd();
    }
    return 0;
}

void print_to_lcd()
{
    char s[20];
    int diceValue = rand() % DICE_MAX + 1;

    sprintf(s, "Dice Dec = %d", diceValue);
    lcd4_cup_row1(&theDisplay);
    lcd4_write_string(&theDisplay, s);
}
```

3.5 Bitfältsstruktur för att visa två tärningars värde på lysdioder

Algorithm 19 Bitfältsstruktur - main.c

```
#include <avr/io.h>
#include <stdlib.h>
#include "Dice.h"

void wait_milliseconds(int);

int main(void)
{
    DDRA = 0x00;
    DDRB = 0xFF;

    bDicePORT.unusedOne = 1;
    bDicePORT.unusedTwo = 1;

    while (1)
    {
        int diceOneRandomValue = rand () % 6 + 1;
        int diceTwoRandomValue = rand () % 6 + 1;

        if(!(PINA & 0x08))
        {
            bDicePORT.diceOne = diceOneRandomValue;
            bDicePORT.diceTwo = diceTwoRandomValue;
        }
        wait_milliseconds(300);
    }
}
```

Algorithm 20 Bitfältsstruktur - Dice.h

```
#ifndef DICE_H_
#define DICE_H_

typedef struct
{
    unsigned char diceTwo : 3;      //Bit 0-2
    unsigned char unusedOne : 1;    //Bit 3 används ej
    unsigned char diceOne : 3;      //Bit 4-6
    unsigned char unusedTwo : 1;    //Bit 7 används ej.
} diceRegister;

#define bDicePORT (*(volatile diceRegister *) &PORTB)
#define bDiceDDR (*(volatile) diceRegister *) &DDRB)
#define bDicePIN (*(volatile) diceRegister *) &PINB)

#endif
```

3.6 Mätning av exekveringstider med simulatorn

Följande kod analyserades för denna uppgift:

```
int main(void)
{
    gXi = 20;
    gYi = 30;
    gRi = gXi * gYi;
}
```

3.6.1 Tid för att utföra multiplikation av två heltal med AVR-processor.

Den beräknade tiden från simulationen framgav resultatet 22 mikrosekunder.

3.6.2 Utan och med volatile

Algorithm 21 Assemblykod med volatile

```
{
    gXi = 20;
0000003E  LDI R24,0x14       Load immediate
0000003F  LDI R25,0x00       Load immediate
00000040  STS 0x0065,R25         Store direct to data space
00000042  STS 0x0064,R24         Store direct to data space
    gYi = 30;
00000044  LDI R24,0x1E       Load immediate
00000045  LDI R25,0x00       Load immediate
00000046  STS 0x0061,R25         Store direct to data space
00000048  STS 0x0060,R24         Store direct to data space
    gRi = gXi * gYi;
0000004A  LDS R20,0x0064         Load direct from data space
0000004C  LDS R21,0x0065         Load direct from data space
0000004E  LDS R18,0x0060         Load direct from data space
00000050  LDS R19,0x0061         Load direct from data space
00000052  MUL R20,R18        Multiply unsigned
00000053  MOVW R24,R0        Copy register pair
00000054  MUL R20,R19        Multiply unsigned
00000055  ADD R25,R0         Add without carry
00000056  MUL R21,R18        Multiply unsigned
00000057  ADD R25,R0         Add without carry
00000058  CLR R1          Clear Register
00000059  STS 0x0063,R25         Store direct to data space
0000005B  STS 0x0062,R24         Store direct to data space
}
```

Algorithm 22 Assemblykod utan volatile

```
{
    gXi = 20;
0000003E  LDI R24,0x14      Load immediate
0000003F  LDI R25,0x00      Load immediate
00000040  STS 0x0065,R25        Store direct to data space
00000042  STS 0x0064,R24        Store direct to data space
    gYi = 30;
00000044  LDI R24,0x1E      Load immediate
00000045  LDI R25,0x00      Load immediate
00000046  STS 0x0061,R25        Store direct to data space
00000048  STS 0x0060,R24        Store direct to data space
    gRi = gXi * gYi;
0000004A  LDI R24,0x58      Load immediate
0000004B  LDI R25,0x02      Load immediate
0000004C  STS 0x0063,R25        Store direct to data space
0000004E  STS 0x0062,R24        Store direct to data space
}
```

Eftersom volatile tvingar programmet att köra utan kodoptimering sker beräkningen i assembly normalt **inte** till. Utan volatile så optimeras den delen bort i assembly och resultatetet läggs direkt in i registret.

3.6.3 Exekveringstider med en 8Mhz överklockad processor

Tidsmätningar utfördes på följande datatyper: int, long och float. Tiderna står till höger om varje beräkning i nedanstående kodram.

Algorithm 23 Tidsmätningar på int, long och float

```
int main(void)
{
    IntTest();       // 39,25 ms
    LongTest();      // 100,88 ms
    FloatTest();     // 103,88 ms
}

void IntTest()
{
    volatile int gXi = 0x0040, gYi = 0x0020, gRi = 0;
    gRi = gXi + gYi;     //1,75 ms
    gRi = gXi * gYi;     //2,75 ms
    gRi = gXi / gYi;     //28,5 ms
}

void LongTest()
{
    volatile long gXi = 0x0040L, gYi = 0x0020L, gRi = 0;
    gRi = gXi + gYi;     // 3,50 ms
    gRi = gXi * gYi;     // 12,13 ms
    gRi = gXi / gYi;     // 77,25 ms
}

void FloatTest()
{
    volatile float gXi = 20.0, gYi = 30.0, gRi = 0;
    gRi = gXi + gYi;     // 15,38 ms
    gRi = gXi * gYi;     // 18,50 ms
    gRi = gXi / gYi;     // 62,00 ms
}
```

3.6.4 Beräkningar på float med blandade datatyper

Algorithm 24 Tidsmätningar på float

```
void FloatMixTest()
{
    volatile float gFi = 1.5;
    volatile float gRi = 0;

    // float = float * int;
    volatile int gXi = 0x0040;
    gRi = gFi * gXi;     // 26,25 ms

    // float = float * long;
    volatile long gLi = 0x0020;
    gRi = gFi * gLi;     // 25,50 ms

    // float = float * float;
    gRi = gFi * gFi;     // 20,88 ms
}
```

3.6.5 Resultat av tidsmätningar

Denna simulering visar att datatypen int är snabbast att beräkna medan long och float kommer efter i beräkningstid. Addition är snabbare att beräkna, multiplikation är ungefär lika snabb som addition medan division är långsammast. Simuleringen i förra uppgiften visade också att det inte är så stora skillnader att utföra aritmetiska beräkningar mellan olika datatyper då det bara skiljde några mikrosekunder. Generellt sett verkar det som om float är den datatypen som tar längst tid att utföra beräkningar med. Detta kan bero på att flera instruktioner måste utföras i assembly för att göra samma beräkning, jämfört med till exempel int. Long har 64 bitar, medan float har 32-bitar, likadant som int, men skillnaden är att float har "decimalbitar" som kan försvåra beräkningarna någorlunda.

3.7 Uppgift 7 - Strukturen sedd på maskinnivån som värdeparameter respektive retur-typ

1. Struktens medlemmar läggs till i registren i minnesstacken i en sekventiell följd. Början av strukten lagras på en adress i minnet där också första medlemmen av strukten räknas, därefter kommer resterande meddlemmar ordnade efter varandra i respektive address beroende på storleken av datamedlemmarna. Alltså om struktens början lagras på adress 0x40, lagras också den första medlemmen också på 0x40. Om andra medlemmen är en int så lagras denna på 0x42 då en int är två bytes.

2. En strukt returneras av en funktion genom att först skicka adressen av funktionens början och därefter struktens adress i jämförelse med funktionen.

3.8 Falsk elektronisk tärning

Algorithm 25 Dice - main.c

```
#include <avr/io.h>
#include <stdlib.h>
#include "..\..\help_files\Dice.h"

void wait_milliseconds(int);
void ThrowDice(char, int *, char, int *);

enum { Normal, False } Dice;

int main(void)
{
    DDRA = 0x00;
    DDRB = 0xFF;

    int nFalseThrow = 0;
    int diceValue = 6;  // Inverted value for diode display

    bDicePORT.unusedOne = 1;
    bDicePORT.unusedTwo = 1;
    bDicePORT.diceTwo = diceValue;
    bDicePORT.diceOne = diceValue;

    Dice = Normal;

    while (1)
    {
        char NormalThrow = !(PINA & 0x01);
        char FalseThrow = !(PINA & 0x10);

        ThrowDice(NormalThrow, &diceValue, FalseThrow, &nFalseThrow
    }
}
```

Algorithm 26 ThrowDice - function

```
void ThrowDice(char NormalThrow, int *diceValue,
char FalseThrow, int *nFalseThrow)
{
    switch(Dice)
    {
        case Normal:
        {
            if(NormalThrow)
            {
                if(*diceValue >= 1)
                {
                    bDicePORT.diceTwo = *diceValue;
                    bDicePORT.diceOne = *diceValue;
                    *diceValue--;
                    wait_milliseconds(200);
                }
                else
                {
                    *diceValue = 6;
                }
            }
            else if(FalseThrow)
            {
                Dice = False;
                *nFalseThrow = 0;
            }
        } break;     // Normal
        case False:
        {
            if(NormalThrow)
            {
                *diceValue = 1;
                if(*nFalseThrow < 2)
                {
                    bDicePORT.diceTwo = *diceValue;
                    bDicePORT.diceOne = *diceValue;
                    *nFalseThrow++;
                    wait_milliseconds(200);
                }
                else
                {
                    *diceValue = 6;
                    bDicePORT.diceTwo = *diceValue;
                    bDicePORT.diceOne = *diceValue;
                    Dice = Normal;
                    *nFalseThrow = 0;
                }
            }
        } break;     //False
        default:
        break;
    }   // Switch
}
```

3.9 Kodlås

Algorithm 27 Kodlås - main

```
#include <avr/io.h>

enum
{
    Closed,
    Open,
    PushKey,
    ReleaseKey
} Locker;

void TryUnlock(char, int *, int *, char, char, int *, const int);
void CheckIfWrongCombination(int*, char, int*, char, char, int*, const↩
     int);
int CheckTrueCombination(int, char, int*, char, char );

int main(void)
{
    DDRA = 0x00;
    DDRB = 0xFF;

    const int timeout = 20;
    int time_counter = 0;
    int iCode = 0;
    int code[6] = {1,2,3,1,1,2};
    Locker = Closed;

    while (1)
    {
        char eventKey1 = !(PINA & 0x80);
        char eventKey2 = !(PINA & 0x40);
        char eventKey3 = !(PINA & 0x20);
        PORTB = PINA;

        TryUnlock(eventKey1, code, &iCode, eventKey2, eventKey3, &↩
            time_counter, timeout);

        wait_milliseconds(50);
    }
}
```

Algorithm 28 TryUnlock-function

```
void TryUnlock(char eventKey1, int * code, int *iCode,
char eventKey2, char eventKey3, int *time_counter, const int timeout)
{
    switch(Locker)
    {
        case Closed:
        {
            if((eventKey1 && (code[*iCode] == 1))
            || (eventKey2 && (code[*iCode] == 2))
            || (eventKey3 && (code[*iCode] == 3)))
            {
                *iCode++;
                Locker = PushKey;
            }
            PORTB = 254;
        } break;    // Closed
        case Open:
        {
            if(*time_counter > timeout)
            {
                Locker = Closed;
                *time_counter = 0;
                *iCode = 0;
            }

            *time_counter++;
            PORTB = 255 - *time_counter;    //20 sec
            wait_milliseconds(500);
        } break;    // Open
        case PushKey:
        {
            if(!eventKey1
            && !eventKey2
            && !eventKey3)
            {
                Locker = ReleaseKey;
                *time_counter++;
            }
            if(*iCode >= 6)
            {
                Locker = Open;
                *time_counter = 0;
            }
        } break;    // PushKey
        case ReleaseKey:
        {
            CheckIfWrongCombination(
            &*iCode, eventKey1, code,
            eventKey2, eventKey3,
            &*time_counter, timeout
            );

            *iCode = CheckTrueCombination(
            *iCode, eventKey1, code,
            eventKey2, eventKey3
            );

            PORTB = 254;
        } break;    // ReleaseKey
        default:
        break;
    }   // Switch
}
```

Algorithm 29 CodeCheck-functions

```
void CheckIfWrongCombination(
    int *iCode, char eventKey1, int * code,
    char eventKey2, char eventKey3,
    int *time_counter, const int timeout)
{
    if (*iCode < 3)
    {
        if((eventKey1 && (code[*iCode] != 1))
        || (eventKey2 && (code[*iCode] != 2))
        || (eventKey3 && (code[*iCode] != 3))
        || *time_counter > timeout)
        {
            Locker = Closed;
            *time_counter = 0;
            *iCode = 0;
        }
        else
        {
            if ((eventKey1 && (code[*iCode] != 1))
            || (eventKey2 && (code[*iCode] != 2)))
            {
                Locker = Closed;
                *time_counter = 0;
                *iCode = 0;
            }
        }
    }
}

int CheckTrueCombination(
    int iCode, char eventKey1, int * code,
    char eventKey2, char eventKey3)
{
    if (iCode < 3)
    {
        if((eventKey1 && (code[iCode] == 1))
        || (eventKey2 && (code[iCode] == 2))
        || (eventKey3 && (code[iCode] == 3)))
        {
            iCode++;
            Locker = PushKey;
        }
    }
    else
    {
        if ((eventKey1 && (code[iCode] == 1))
        || (eventKey2 && (code[iCode] == 2)))
        {
            iCode++;
            Locker = PushKey;
        }
    }               return iCode;
}
```

3.10 Avstudsning av en tryckknapp

3.10.1 Tillståndsdiagram

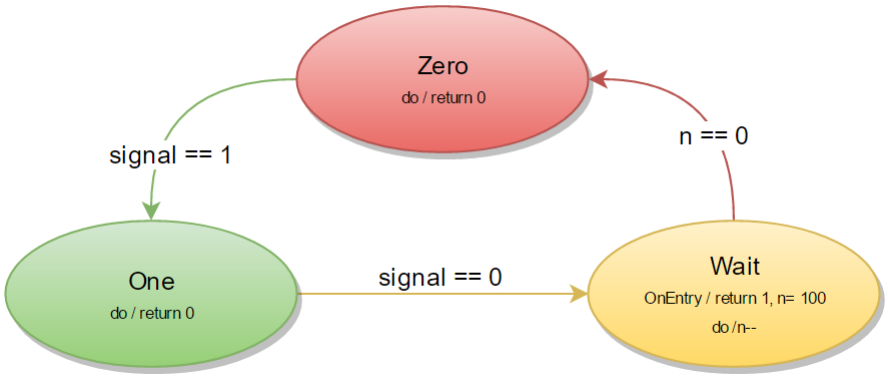

Figure 4: Tillståndsdiagram för Avstudsning av en tryckknapp

3.10.2 C-kod

Algorithm 30 Avstudsning - main

```
#include <avr/io.h>

static int n = 0;
static enum {Zero, Wait, One} Signal;

int main(void)
{
    DDRB = 0xFF;
    DDRA = 0x00;
    Signal = Zero;

    while (1)
    {
        char input = !(PINA & 0x01);
        PORTB = 255 - bounce(input);
        wait_milliseconds(100);
    }
}

int bounce(int signal)
{
    switch (Signal)
    {
        case Zero:
        {
            if(signal == 1)
            {
                Signal = One;
            }
            return 0;
        } break;
        case One:
        {
            if(signal == 0)
            {
                Signal = Wait;
                n = 10;
                return 1;
            }
            return 0;
        } break;
        case Wait:
        {
            n--;
            if(n == 0)
            {
                Signal = Zero;
                return 0;
            }
        } break;
        default:
        break;
    }
}
```

3.11 Trafikljusstyrning

3.11.1 Introduktion

I denna uppgift skulle trafikljus styras genom olika tillstånd. Lösningen genomfördes i C-kod men med objektorienterad stil, genom användning av structs. Följande figurer visar hur kretsens kopplingar såg ut och vilka lampor som representerade vilken riktning.

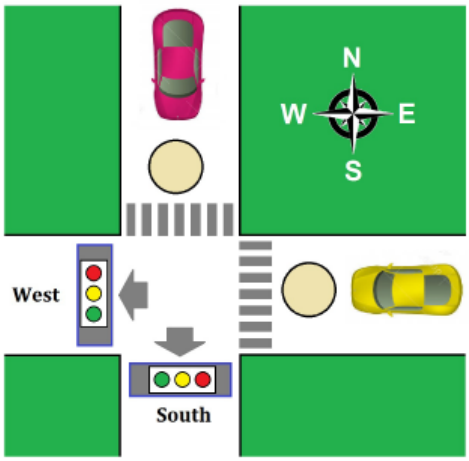

Figure 5: Trafikljus-korsning

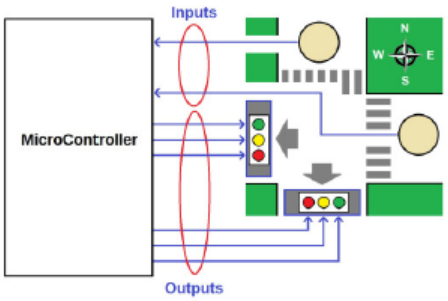

Figure 6: Systemdiagram

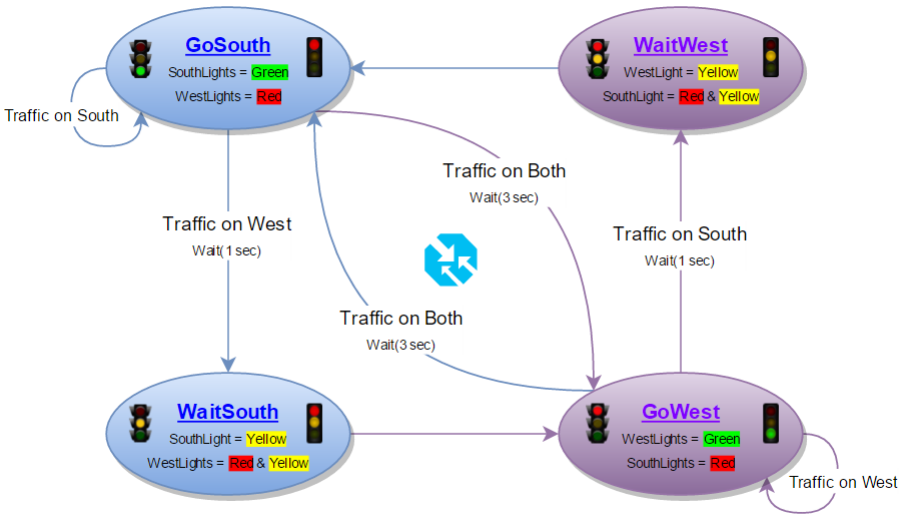

Figure 7: Tillståndsdiagram Trafikljus

3.11.2 Tillståndsdiagram

I figur 7 ser man den uppritade tillståndsdiagrammet för denna uppgift. Trafik detektering sker via sensorer som man trycker på mikroprocessorns. Starttillstånd kommer att bli GoSouth, vilket innebär att det är trafik i södergående riktning, vilket då lyser upp den gröna lampan och låter den västgående riktningen vara röd. Mellanväxlingen mellan riktningarna sker på det viset att när trafik detekteras så åÍbörjas en nedräkning. Beroende på om det är en riktning som är trafikerad eller om båda riktningarna är trafikerade. När en riktning är trafikerad dras timern igång 2 sekunder snabbare än om båda riktningar skulle vara trafikerade. Detta för att vid lite trafik, inte låta trafikanterna vänta på ljusomslag. och vid mycket trafik, låta alla riktningarna få lika mycket tid att passera. Enligt tillståndsdiagrammet visar de mittersta pilarna att omslaget sker direkt från grön till röd och vice versa, men detta är bara en ritning som indikerar en annan timer än de andra pilarna och för att förtydliga hur det går till. Transitionerna passerar alltid väntetillstånden. I nästkommande sidor visas koden för denna uppgift.

3.11.3 C-Kod

Algorithm 31 TrafficLights - main.c

```
#include <avr/io.h>
#include "TrafficLights.h"

int main(void)
{
    DDRA = 0x00;
    DDRB = 0xFF;
    Traffic = GoSouth;

    while (1)
    {
        char TrafficOnSouth = !(PINA & 0x01);
        char TrafficOnWest = !(PINA & 0x02);
        switch (Traffic)
        {
            case GoSouth:
            {
                ActivateSouthLane(TrafficOnWest, TrafficOnSouth);

            } break;
            case WaitSouth:
            {
                TransitionSouthToWest();

            } break;
            case GoWest:
            {
                ActivateWestLane(TrafficOnSouth, TrafficOnWest);

            } break;
            case WaitWest:
            {
                TransitionWestToSouth();

            } break;
            default:
            break;
        }
    }
}
```

Algorithm 32 Trafficlight.h Del I

```
#ifndef TRAFFICLIGHTS_H_
#define TRAFFICLIGHTS_H_

enum {
    GoSouth,
    WaitSouth,
    GoWest,
    WaitWest
} Traffic;

struct TrafficLights {
    char Red;
    char Yellow;
    char Green;
};

struct TrafficLights SouthTrafficLight = {0xFF - 0x20, 0xFF - 0x40, 0↩
    xFF - 0x80};
struct TrafficLights WestTrafficLight = {0xFF - 0x10, 0xFF - 0x08, 0↩
    xFF - 0x04};

// How to make this work?
//#define SouthLight (*(SouthTrafficLight)* &PORTB)
//#define WestLight (*(WestTrafficLight)* &PORTB)

int oneSec = 1000, threeSec = 3000; //Timer tid
```

Algorithm 33 Trafficlight.h - Del II (Metoder)

```
void ActivateSouthLane(char TrafficOnWest, char TrafficOnSouth)
{
    PORTB = SouthTrafficLight.Green;
    PORTB += WestTrafficLight.Red;
    if(TrafficOnWest && !TrafficOnSouth)
    {
        wait_milliseconds(oneSec);
        Traffic = WaitSouth;
    }
    if(TrafficOnWest && TrafficOnSouth)
    {
        wait_milliseconds(threeSec);
        Traffic = WaitSouth;
    }
}

void TransitionSouthToWest()
{
    PORTB = WestTrafficLight.Red;
    PORTB += WestTrafficLight.Yellow;
    PORTB += SouthTrafficLight.Yellow + 1;
    wait_milliseconds(oneSec);
    Traffic = GoWest;
}

void ActivateWestLane(char TrafficOnSouth, char TrafficOnWest)
{
    PORTB = WestTrafficLight.Green;
    PORTB += SouthTrafficLight.Red - 1;
    if(TrafficOnSouth && !TrafficOnWest)
    {
        wait_milliseconds(oneSec);
        Traffic = WaitWest;
    }
    if(TrafficOnSouth && TrafficOnWest)
    {
        wait_milliseconds(threeSec);
        Traffic = WaitWest;
    }
}

void TransitionWestToSouth()
{
    PORTB = WestTrafficLight.Yellow;
    PORTB += SouthTrafficLight.Red;
    PORTB += SouthTrafficLight.Yellow;
    wait_milliseconds(oneSec);
    Traffic = GoSouth;
}

#endif /* TRAFFICLIGHTS_H_ */
```