Datorteknik
Övningsuppgifter

Michael Josefsson (2005–),
Stefan Gustafsson (–2004)

ver 0.6 2019-04-19

OBS! Det är en bra idé att provköra uppgifterna i AtmelStudio. Det är samma miljö som används vid laborationerna så du måste kunna den. Vid simulering nollställs samtliga register och minnen, i verkligheten är inte detta alltid säkert. Använd exempelvis instruktionerna `ldi`, `sts` för detta.

1.1. Assemblerprogrammering

Operationer mellan register och minne är grundläggande.

1.1.1. Minne

1. Flytta innehållet i minnescell `$110` till minnescell `$112`.
2. Addera innehållet i minnescellerna `$110` och `$111`. Placera summan i minnescell `$112`.
3. Skifta innehållet i minnescell `$110` en bitposition åt vänster och placera resultatet i minnescell `$111`. Den minst signifikanta biten (*Least Siginificant Bit, LSB*, bit 0) skall vara nollställd i resultatet.
4. Flytta de fyra mest signifikanta bitarna (*Most Significant Bit, MSB*) i minnescell `$110` till de fyra minst signifikanta bitarna i minnescell `$111`. De fyra mest signifikanta bitarna i minnescell `$111` skall vara nollställda efter instruktionen.
5. Dela innehållet i minnescell `$110` i två fyrabitarsdelar (s.k. *nibbles* eller *nybbles*). Lagra de mest signifikanta bitarna i bit 3–0 i minnescell `$111` och de minsta signifikanta bitarna i bit 3–0 i minnescell `$112`.
6. Minnescellerna `$110` och `$111` innehåller två heltal i binär representation utan tecken. Placera det största av talen i minnescell `$112`.
7. Beräkna $10 \cdot X$ på ett fyrabitars tal X i binär representation utan tecken. Talet finns lagrat i minnescellen `$110`. Placera resultatet i minnescell `$112`.
8. Använd instruktionerna `sts/lds`, `st/ld` respektive `std/ldd` för att komma åt variabler i SRAM.

   Variant: Använd dem för att speciellt komma åt dataregistren `r0` osv. Dessa register är *minnesmappade* till minnets låga adresser.

1.1.2. Strängar

9. Översätt textsträngen ”MIKROdator” till ASCII-kod. Hur kan man veta att strängen är slut?

10. Beräkna kvadraten på ett tal i r16:s lägre halva med hjälp av en tabell utplacerad i FLASH-minnet:

Adress	Innehåll	Kommentar
BAS+0	$00	$0^2 = 0$
BAS+1	$01	$1^2 = 1$
BAS+2	$04	$2^2 = 4$
BAS+3	$09	$3^2 = 9$
BAS+4	$10	$4^2 = 16$
BAS+5	$19	$5^2 = 25$
BAS+6	$24	$6^2 = 36$
BAS+7	$31	$7^2 = 49$
BAS+8	$40	$8^2 = 64$
BAS+9	$51	$9^2 = 81$
BAS+A	$64	$10^2 = 100$
BAS+B	$79	$11^2 = 121$
BAS+C	$90	$12^2 = 144$
BAS+D	$A9	$13^2 = 169$
BAS+E	$C4	$14^2 = 196$
BAS+F	$E1	$15^2 = 225$

11. Bestäm kvadraten på talet X i föregående uppgift med hjälp av instruktionen `MUL` utan att använda någon tabell. Vilka för- och nackdelar finns med de olika metoderna?

12. Konvertera ett siffra X ($X \in [0, 9]$) till sin ASCII-kod.

    Lös uppgiften både med och utan tabellslagning. Du får skapa eget tabellinnehåll.

13. Skriv ett program som väljer mellan tre olika programvägar beroende på innehållet i minnescell `$101`. Om minnescellen innehåller `$01` skall hopp ske till rutinen `ETT`, för `$02` skall hopp ske till rutinen `TVA` och för `$03` skall hopp ske till `TRE`. Använd instruktionen `breq` för hoppen. Vad händer i ditt program om minnescellen innehåller ett annat tal?

14. Skiftinstruktioner förekommer ofta.

    a) Vad är skillnaden mellan aritmetiskt och logiskt högerskift?

    b) Vad är skillnaden mellan aritmetiskt och logiskt vänsterskift?

15. En assemblerinstruktion består av två delar. Den ena är operationskoden, som anger vilken slags operation som skall utföras. Vilken är den andra delen? Måste den alltid finnas?

16. Vad är skillnaden mellan absolut och relativ adressering? Vilka fördelar har man av att använda relativ adressering där det är möjligt?

17. Vid ett visst tillfälle innehåller de interna registren följande:

Register	Innehåll
r20	$61
r21	$F5
X	$100

SRAM innehåller samtidigt följande:

Cell	Innehåll
$100	$19

Avgör vilka register och minnesceller som påverkas av följande instruktioner, och ange det nya innehållet efter det att instruktionen utförts. Instruktionerna antas utföras med de angivna värden var och en för sig från samma utgångstillstånd.

a) `mov r16,r20` b) `lds r20,$100` c) `clr r21`
d) `inc r21` e) `sts $100,r21` f) `ldi r20,$F5`
g) `ld r17,X` h) `ld r17,X+` i) `mul r20,r21`

18. a) Hur sker anrop av en subrutin?
    b) Hur sker återhopp till huvudprogrammet?
    c) Varför kan man inte använda en vanlig hoppinstruktion för återhoppet?
    d) Kan en subrutin anropa en subrutin?
    e) Kan en subrutin anropa sig själv?
    f) Finns det någon begränsning för antalet möjliga nivåer av subrutinanrop?
    g) Vad bör man tänka på när man använder interna register i en subrutin?

19. Parametrarna till en omfattande subrutin får sällan plats i de interna registren. I sådana fall lägger man oftast parametrarna på stacken innan subrutinen anropas.
    a) Antag att en 16-bitars adress lagts på stacken varefter en subrutin anropas. Hur når man argumentet? Hämta talet till `r25:r24`.
    b) Vad måste man tänka på efter återhoppet från subrutinen? Vad händer annars?

20. Använd logiska instruktioner (`andi`, `ori`, `eor` och `com`) för att göra följande:
    a) Nollställ register `r16`.
    b) Ettställ de fyra mest signifikanta bitarna i `r16`.

c) Nollställ de tre minst signifikanta bitarna i `r16`.

d) Utför bitvis `NOR` på register `r16` och `r17`. Lägg resultatet i `r18`.

e) Invertera bitarna 2 till 6 i `r16`.

f) Flytta de fyra minst signifikanta bitarna i `r16` till de fyra minst signifikanta bitarna i `r17`. Övriga bitar i `r17` skall inte ändras. Innehållet i `r16` får förstöras.

21. Skriv en subrutin som med ett argument `n` plockar ut den `n`:te byten ur en lista i FLASH-minnet (programminnet). Listan är högst 64 bytes lång. Använd assemblerinstruktionen `lpm`.

    Variant:

    1) Hindra uppslagning utanför listan.
    2) Se till att inga register onödigtvis påverkas.

22. Det finns några intressanta bit-manipulerande knep som man kanske inte genomskådar omedelbart. Här visas några att analysera. Frågeställningen är ”vad gör koden”? Ansätt några indata och ta reda på det!

a)
```
eor     r16,r17
ret
```

b)
```
mov     r17,r16
andi    r17,$AA
lsr     r17
andi    r16,$55
lsl     r16
or      r16,r17
mov     r17,r16
andi    r17,$CC
lsr     r17
lsr     r17
andi    r16,$33
lsl     r16
lsl     r16
or      r16,r17
mov     r17,r16
andi    r17,$F0
lsr     r17
lsr     r17
lsr     r17
lsr     r17
andi    r16,$0F
lsr     r16
lsr     r16
lsr     r16
lsr     r16
or      r16,r17
ret
```

.

c)
```
mov     r17,r16
andi    r17,$AA
lsr     r17
andi    r16,$55
add     r16,r17
mov     r17,r16
andi    r17,$CC
lsr     r17
lsr     r17
andi    r16,$33
add     r16,r17
mov     r17,r16
swap    r17
add     r16,r17
andi    r16,$0F
ret
```

.

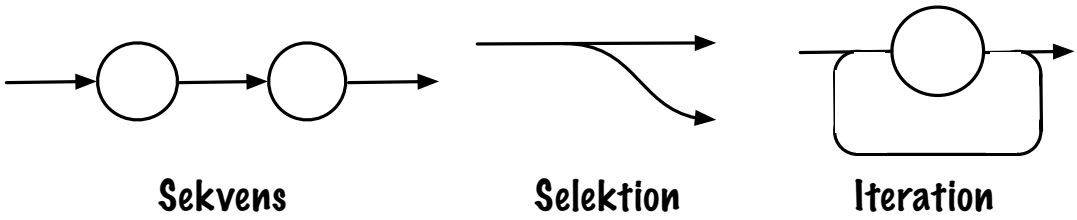
1.2. Strukturerad programmering

För att öva på konstruktion av JSP-diagram kommer några *syntaxdiagram* att tolka strukturerat. Syntaxdiagram kanske kan vara bra att ha sett någon gång men här finns de med enbart som underlag till JSP-övningarna.

1.2.1. Introduktion till syntaxdiagram

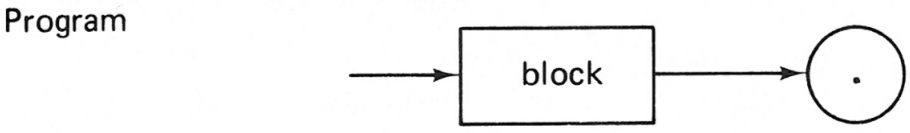
Ett syntaxdiagram är en beskrivning av vilka komponenter som bygger upp ett programmeringsspråk. Diagrammet kan vara ett steg i processen att skriva en kompilator för språket. Här ska vi använda syntaxdiagrammen för att konstruera motsvarande strukturdiagram.

Man läser ett syntaxdiagram från vänster till höger och strukturkomponenterna *sekvens*, *iteration* och *selektion* återfinns i diagrammet fast inte i JSP-form.



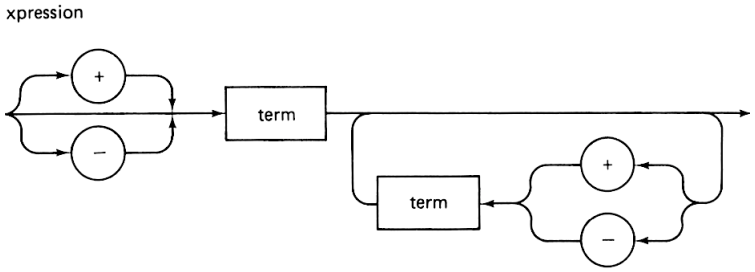
1.2.2. PL/0

Nedan visas några syntaxdiagram att översätta till JSP-notation. Diagrammen definierar exempelspråket PL/0.[1]

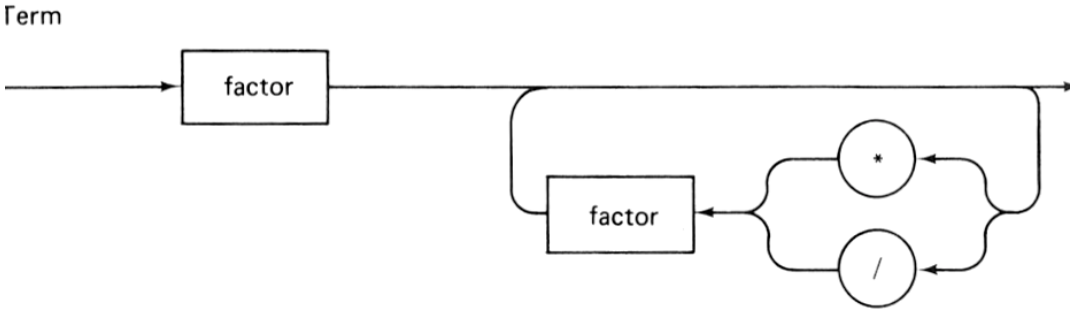


Ett PL/0-program består av ett ***block*** *följt av en punkt.*

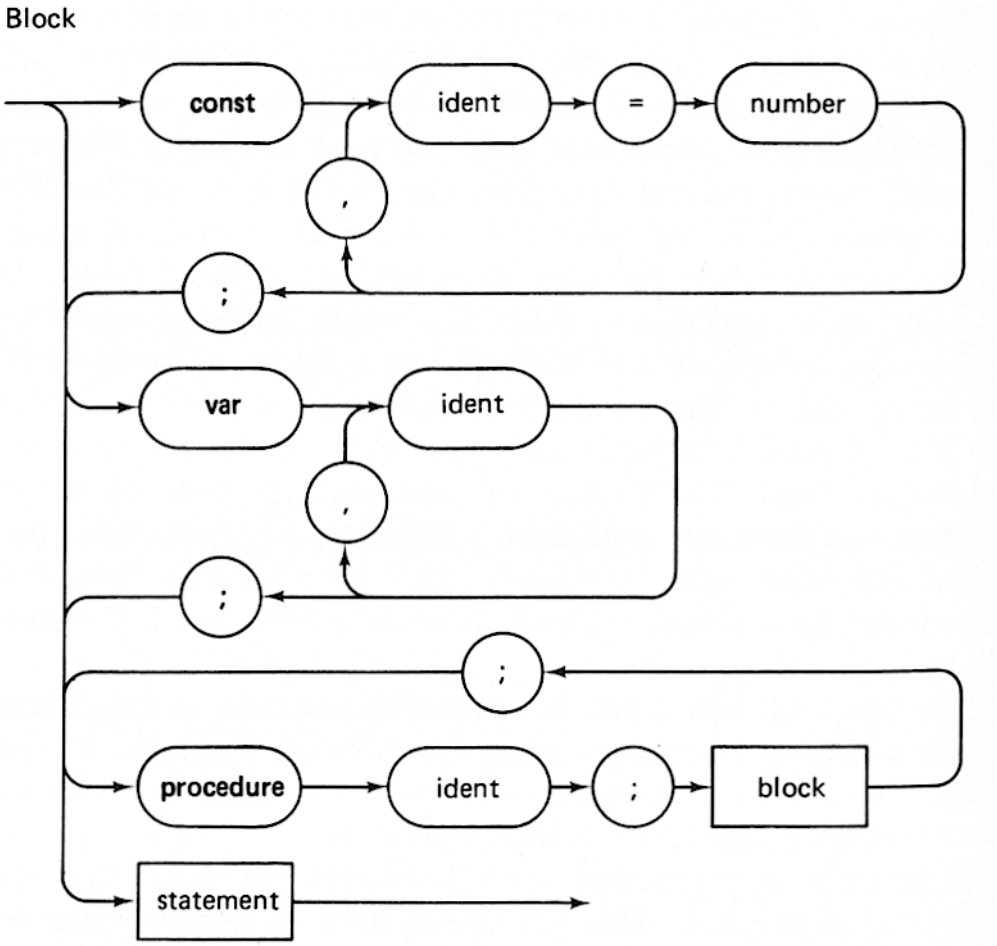
[1]Wirth, *Algorithms + Datastructures = Programs*



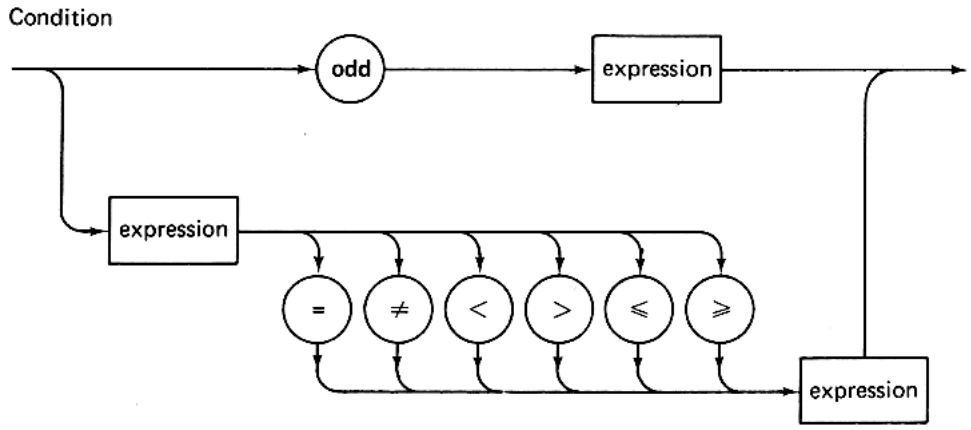
*Ett **expression** inleds med **+**, **-** eller en **term** och kan följas av **+** eller **-** och en term.*



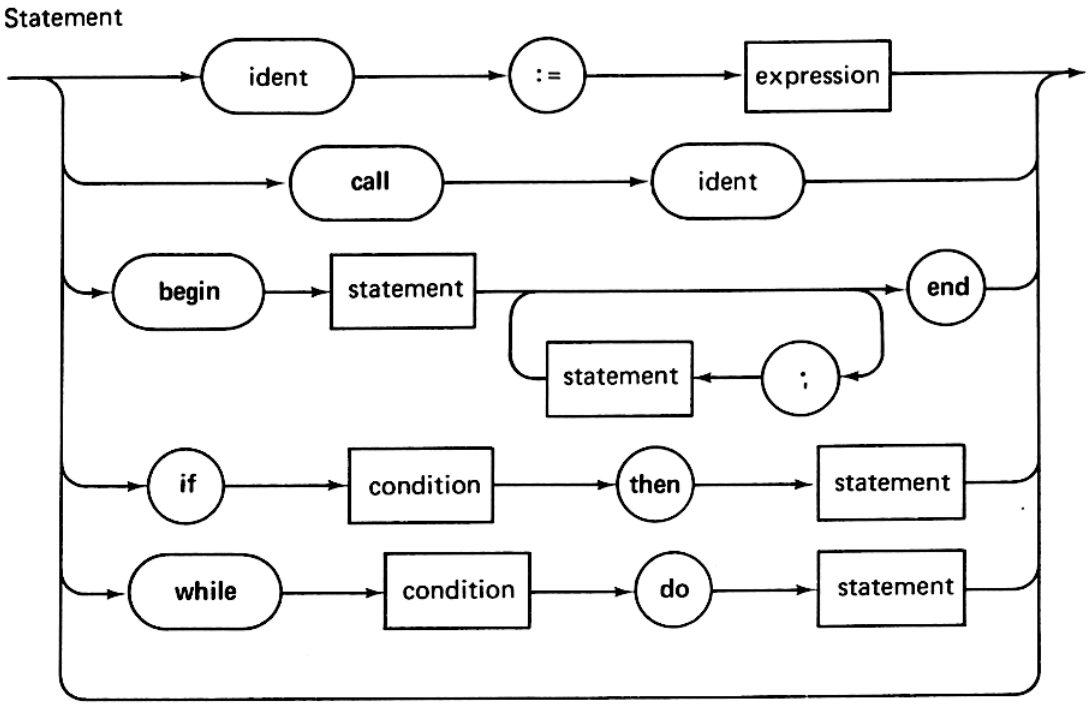
*Syntaxdiagram för en **term**.*



*Ett **block** börjar antingen med en konstantlista efter **const**, variabellista efter **var**, procedur **procedure** eller ett **statement**. Variabellistan utgörs av kommaseparerad lista av identifierare **ident** och raden avslutas med ett semikolon ;. Övriga språkelement konstrueras analogt.*



Syntaxdiagram för ett ***condition****.*



Syntaxdiagram för ett ***statement****. Här känner vi igen språkelement tillhörande ett högnivåspråk.*

1.2.3. Hyfsa kod

23. Rutinen `s2h` på nästa sida översätter en ASCII-sträng till dess hexadecimala mostvarighet. Starta ett C-projekt i AtmelStudio och provkör.

```
void s2h(char *str, char *xstr)
{
    unsigned char tkn,l,i,k;

    l = strlen(str);
    for(i=0, k=0; i<l; i++)
    {
        tkn = str[i];
        if(tkn >= 0 && tkn <= 0x0f)
        {
            xstr[k] = 0x30;
            if(tkn >= 0 && tkn <= 9)
                xstr[k+1] = 0x30 + tkn;
            else
                xstr[k+1] = 0x37 + tkn;
        }
        else  if(tkn >= 0x10 && tkn <= 0x1f)
        {
            xstr[k] = 0x31; tkn -= 0x10;
            if(tkn >= 0 && tkn <= 9)
                xstr[k+1] = 0x30 + tkn;
            else
                xstr[k+1] = 0x37 + tkn;
        }
        else  if(tkn >= 0x20 && tkn <= 0x2f)
        {
            xstr[k] = 0x32; tkn -= 0x20;
            if(tkn >= 0 && tkn <= 9)
                xstr[k+1] = 0x30 + tkn;
            else
                xstr[k+1] = 0x37 + tkn;
        }
        else  if(tkn >= 0x30 && tkn <= 0x3f)
        {
            xstr[k] = 0x33; tkn -= 0x30;
            if(tkn >= 0 && tkn <= 9)
                xstr[k+1] = 0x30 + tkn;
            else
                xstr[k+1] = 0x37 + tkn;
        }
        else  if(tkn >= 0x40 && tkn <= 0x4f)
        {
            xstr[k] = 0x34; tkn -= 0x40;
            if(tkn >= 0 && tkn <= 9)
                xstr[k+1] = 0x30 + tkn;
            else
                xstr[k+1] = 0x37 + tkn;
        }
        else  if(tkn >= 0x50 && tkn <= 0x5f)
        {
            xstr[k] = 0x35; tkn -= 0x50;
            if(tkn >= 0 && tkn <= 9)
                xstr[k+1] = 0x30 + tkn;
            else
                xstr[k+1] = 0x37 + tkn;
        }
        else  if(tkn >= 0x60 && tkn <= 0x6f)
        {
            xstr[k] = 0x36; tkn -= 0x60;
            if(tkn >= 0 && tkn <= 9)
                xstr[k+1] = 0x30 + tkn;
            else
                xstr[k+1] = 0x37 + tkn;
        }
        else  if(tkn >= 0x70 && tkn <= 0x7f)
        {
            xstr[k] = 0x37; tkn -= 0x70;
            if(tkn >= 0 && tkn <= 9)
                xstr[k+1] = 0x30 + tkn;
            else
                xstr[k+1] = 0x37 + tkn;
        }
        k = k + 2;
    }
    xstr[k] = 0x00;
}
```

a) Undersök vad rutinen gör för något. Förslag på testprogram är

```
int main(int argc, const char *argv[]) {
    char *str="AZaz.!9";
    char *hex="               ";

    s2h(str,hex);
    printf("Input: %s \t Output: %s \n", str, hex);
    return 0;
}
```

b) Förenkla C-koden, dvs skriv om den i bättre C.

c) Skriv motsvarande rutin i assembler — som den rimligen skulle gjorts från början.

d) Hur stora blev de olika lösningarna? Jämför med originalet.

e) Hur påverkar C-kompilatorns olika optimeringsflaggor (till exempel `-O1`, `-O2`, `-Os`)?

f) Studera hur anropet sker från C till assembler (via menyalternativet ”Debug/Windows/Disassembly”). Anpassa din egen assemblerrutin och använd denna istället för C-kod.

g) Vilka begränsningar finns i originalrutinen? Kan dessa byggas bort i din version? Drag slutsatsen att man inte kan lita på ”internet-kod” utan att undersöka den mycket noggrant själv!

24. Följande rutin översätter ett BCD-värde i `bcd` till dess representation för att visas på en sjusegmentsdisplay. Argumentet `cc` är 1 om dislayen är av typen *common cathode* och noll annars.

```
char bcdto7seg(char bcd, char cc)
{
    char retval;

    switch(bcd)
    {
      case 0: if(cc==1) retval = 0b00111111;
              else
              if(cc==0) retval = 0b11000000;
              break;
      case 1: if(cc==1) retval = 0b00000110;
              else
              if(cc==0) retval = 0b11111001;
              break;
      case 2: if(cc==1) retval = 0b01011011;
              else
              if(cc==0) retval = 0b10100100;
              break;
      case 3: if(cc==1) retval = 0b01001111;
              else
              if(cc==0) retval = 0b10110000;
              break;
      case 4: if(cc==1) retval = 0b01100110;
              else
              if(cc==0) retval = 0b10011001;
              break;
      case 5: if(cc==1) retval = 0b01101101;
              else
              if(cc==0) retval = 0b10010010;
              break;
      case 6: if(cc==1) retval = 0b01111101;
              else
              if(cc==0) retval = 0b10000010;
              break;
      case 7: if(cc==1) retval = 0b00000111;
              else
              if(cc==0) retval = 0b11111000;
              break;
      case 8: if(cc==1) retval = 0b01111111;
              else
              if(cc==0) retval = 0b10000000;
              break;
      case 9: if(cc==1) retval = 0b01101111;
              else
              if(cc==0) retval = 0b10010000;
    }

    return retval;
}

.
```

Din uppgift är att snygga upp koden och minimera programminnesåtgången. Lämpliga steg *kan* vara att studera koden och sedan

a) identifiera och eliminera onödig kod

b) beskriva i ord vad rutinen *egentligen* gör

c) skriva en bättre (här: kortare) kod i C

d) skriva en bättre kod i assembler.

25. Hyfsa följande rutin.

```
RACKET1_UP:
    lds r16,RACKET1
    lsl r16
    cpi r16,192
    breq RACK1_LIM_UP
    sts RACKET1,r16
    ret
RACK1_LIM_UP:
    ori r16,224
    sts RACKET1,r16
    ret
```

26. Koden nedan kommer ur en projektdokumentation. Kommentarer är borttagna, här är bara programstrukturen viktig. Snygga till den!

```
TWI_INTERRUPT:
    push r22
    in  r22, SREG
    push r22
    in r22, TWSR
    andi r22, $F8
    cpi r22,$60
    breq SET_TWINT
    cpi r22,$80
    breq READ_DATA
    cpi r22,$A2
    breq SEND_DATA
    rjmp SET_TWINT
TWI_FINISHED:
    pop r22
    out SREG, r22
    pop r22
    reti
SET_TWINT:
    ldi r22, (1<<TWINT)|(1<<TWEN)|(1<<TWEA)|(1<<TWIE)
    out TWCR, r22
    rjmp TWI_FINISHED
READ_DATA:
    in r22, TWDR
    mov r17, r22
    andi r17, 0b00000111
    andi r22, 0b00011000
    lsr r22
    lsr r22
    lsr r22
    sts SELECTED_COLUMN, r17
    sts PLAYER, r22
    call ADD_DOT
    rjmp SET_TWINT
SEND_DATA:
    lds r22, TWI_DATA
    out TWDR, r22
    ldi r22, (1 << TWINT) | (1 << TWEN)
    out TWCR, r22
    rjmp SET_TWINT
```

27. Följande kod är ur en projektdokumentation. Skriv den bättre. Hur mycket mindre blir den?

```
// Name: movCOUNT (subroutine)
// Purpose: Store to SRAM
// Uses: r16, r17
movCOUNT:
    cpi     r17,1
    breq    STORE_ONE
    cpi     r17,2
    breq    STORE_TWO
    cpi     r17,3
    breq    STORE_THR
    cpi     r17,4
    breq    STORE_FOU
    cpi     r17,5
    breq    STORE_FIV
    cpi     r17,6
    breq    STORE_SIX
    cpi     r17,7
    breq    STORE_SEV
    cpi     r17,8
    breq    STORE_EIG
STORE_ONE:
    sts     MINNE,r16
    jmp     END_STORE
STORE_TWO:
    sts     MINNE+1,r16
    jmp     END_STORE
STORE_THR:
    sts     MINNE+2,r16
    jmp     END_STORE
STORE_FOU:
    sts     MINNE+3,r16
    jmp     END_STORE
STORE_FIV:
    sts     MINNE+4,r16
    jmp     END_STORE
STORE_SIX:
    sts     MINNE+5,r16
    jmp     END_STORE
STORE_SEV:
    sts     MINNE+6,r16
    jmp     END_STORE
STORE_EIG:
    sts     MINNE+7,r16
END_STORE:
    ret
```

```
// Name: readRAM (subroutine)
// Purpose: Get a byte from SRAM
// Uses: r17, r18
readRAM:
    cpi     r17,0
    breq    FIRST
    cpi     r17,1
    breq    SECOND
    cpi     r17,2
    breq    THIRD
    cpi     r17,3
    breq    FOURTH
    cpi     r17,4
    breq    FIFTH
    cpi     r17,5
    breq    SIXTH
    cpi     r17,6
    breq    SEVENTH
    cpi     r17,7
    breq    EIGHTH
FIRST:
    lds     r18,MINNE
    jmp     ENDZ
SECOND:
    lds     r18,MINNE+1
    jmp     ENDZ
THIRD:
    lds     r18,MINNE+2
    jmp     ENDZ
FOURTH:
    lds     r18,MINNE+3
    jmp     ENDZ
FIFTH:
    lds     r18,MINNE+4
    jmp     ENDZ
SIXTH:
    lds     r18,MINNE+5
    jmp     ENDZ
SEVENTH:
    lds     r18,MINNE+6
    jmp     ENDZ
EIGHTH:
    lds     r18,MINNE+7
ENDZ:
    ret
```

28. Följande kod är tagen ur en projektdokumentation. Den kan skrivas bättre. Gör det! Hur mycket mindre än nuvarande knappt 400 bytes kan du få koden?

Subrutinen `FEL_KNAPP` är belägen någon annanstans. Macrot `TC_MACRO` är definierat som

```
    in      r16,TCNT0
    andi    r16,$07
```

```
LAMPOR_LYS:
   cpi   r16,$00
   breq  NOLL
   cpi   r16,$01
   breq  ETT
   cpi   r16,$02
   breq  TVA
   cpi   r16,$03
   breq  TRE
   cpi   r16,$04
   breq  FYRA
   cpi   r16,$05
   breq  FEM
   cpi   r16,$06
   breq  SEX
   cpi   r16,$07
   breq  SJU
NOLL:
   ldi   r16,$01
   out   PORTB,r16
   jmp   SIGNAL
ETT:
   ldi   r16,$02
   out   PORTB,r16
   jmp   SIGNAL
TVA:
   ldi   r16,$04
   out   PORTB,r16
   jmp   SIGNAL
TRE:
   ldi   r16,$08
   out   PORTB,r16
   jmp   SIGNAL
FYRA:
   ldi   r16,$10
   out   PORTB,r16
   jmp   SIGNAL
FEM:
   ldi   r16,$20
   out   PORTB,r16
   jmp   SIGNAL
SEX:
   ldi   r16,$40
   out   PORTB,r16
   jmp   SIGNAL
SJU:
   ldi   r16,$80
   out   PORTB,r16
   jmp   SIGNAL
SIGNAL:
   sbi   PORTD,1
KNAPP0:
   sbis  PINA,0
   rjmp  KNAPP1
   rjmp  CHECK0
KNAPP1:
   sbis  PINA,1
   rjmp  KNAPP2
   rjmp  CHECK1
KNAPP2:
   sbis  PINA,2
   rjmp  KNAPP3
   rjmp  CHECK2
KNAPP3:
   sbis  PINA,3
   rjmp  KNAPP4
   rjmp  CHECK3
KNAPP4:
   sbis  PINA,4
   rjmp  KNAPP5
   rjmp  CHECK4
KNAPP5:
   sbis  PINA,5
   rjmp  KNAPP6
   rjmp  CHECK5
KNAPP6:
   sbis  PINA,6
   rjmp  KNAPP7
   rjmp  CHECK6
KNAPP7:
   sbis  PINA,7
   rjmp  KNAPP0
   rjmp  CHECK7
CHECK0:
   sbic  PINA,0
   rjmp  CHECK0
   cpi   r16,$01
   brne  FEL
   andi  r16,$FE
   out   PORTB,r16
   call  LJUD
KNAPP_0:
   sbic  PINA,0
   jmp   KNAPP_0
   TC_MACRO
   jmp   LOOP
CHECK1:
   sbic  PINA,1
   rjmp  CHECK1
   cpi   r16,$02
   brne  FEL
   andi  r16,$FD
   out   PORTB,r16
   call  LJUD
KNAPP_1:
   sbic  PINA,1
   jmp   KNAPP_1
   TC_MACRO
   jmp   LOOP
CHECK2:
   sbic  PINA,2
   rjmp  CHECK2
   cpi   r16,$04
   brne  FEL
   andi  r16,$FB
   out   PORTB,r16
   call  LJUD
KNAPP_2:
   sbic  PINA,2
   jmp   KNAPP_2
   TC_MACRO
   jmp   LOOP
CHECK3:
   sbic  PINA,3
   rjmp  CHECK3
   cpi   r16,$08
   brne  FEL
   andi  r16,$F7
   out   PORTB,r16
   call  LJUD
KNAPP_3:
   sbic  PINA,3
   jmp   KNAPP_3
   TC_MACRO
   jmp   LOOP
FEL:
   jmp   FEL_KNAPP
CHECK4:
   sbic  PINA,4
   rjmp  CHECK4
   cpi   r16,$10
   brne  FEL
   andi  r16,$EF
   out   PORTB,r16
   call  LJUD
KNAPP_4:
   sbic  PINA,4
   jmp   KNAPP_4
   TC_MACRO
   jmp   LOOP
CHECK5:
   sbic  PINA,5
   rjmp  CHECK5
   cpi   r16,$20
   brne  FEL
   andi  r16,$DF
   out   PORTB,r16
   call  LJUD
KNAPP_5:
   sbic  PINA,5
   jmp   KNAPP_5
   TC_MACRO
   jmp   LOOP
CHECK6:
   sbic  PINA,6
   rjmp  CHECK6
   cpi   r16,$40
   brne  FEL
   andi  r16,$BF
   out   PORTB,r16
   call  LJUD
KNAPP_6:
   sbic  PINA,6
   jmp   KNAPP_6
   TC_MACRO
   jmp   LOOP
CHECK7:
   sbic  PINA,7
   rjmp  CHECK7
   cpi   r16,$80
   brne  FEL
   andi  r16,$7F
   out   PORTB,r16
   call  LJUD
KNAPP_7:
   sbic  PINA,7
   jmp   KNAPP_0
   TC_MACRO
   jmp   LOOP
LJUD:
   ; code for sound
   ret
LOOP:
   ; some more code
```

29. Även denna kod kommer från en projektdokumentation. Hur bör den utföras? Vilken blir vinsten i kodstorlek?

```
GET_KEY:
   push r18
   in r17, PIND
   andi r17, $0F
   mov r18,r17
   clr r17
   cpi r18, $07
   breq zero
   cpi r18, $08
   breq one
   cpi r18, $09
   breq two
   cpi r18, $0C
   breq three
   cpi r18, $04
   breq four
   cpi r18, $05
   breq five
   cpi r18, $06
   breq six
   cpi r18, $0D
   breq seven
   cpi r18, $01
   breq eight
   cpi r18, $02
   breq nine
   cpi r18, $03
   breq ten
   cpi r18, $0E
   breq eleven
   cpi r18, $0A
   breq twelve
   cpi r18, $00
   breq thirteen
   cpi r18, $0B
   breq fourteen
   cpi r18, $0F
   breq fifteen
zero:
   ldi r17, $00
   rjmp skiprest
one:
   ldi r17, $01
   rjmp skiprest
two:
   ldi r17, $02
   rjmp skiprest
three:
   ldi r17, $03
   rjmp skiprest
four:
   ldi r17, $04
   rjmp skiprest
five:
   ldi r17, $05
   rjmp skiprest
six:
   ldi r17, $06
   rjmp skiprest
seven:
   ldi r17, $07
   rjmp skiprest
eight:
   ldi r17, $08
   rjmp skiprest
nine:
   ldi r17, $09
   rjmp skiprest
ten:
   ldi r17, $0A
   rjmp skiprest
eleven:
   ldi r17, $0B
   rjmp skiprest
twelve:
   ldi r17, $0C
   rjmp skiprest
thirteen:
   ldi r17, $0D
   rjmp skiprest
fourteen:
   ldi r17, $0E
   rjmp skiprest
fifteen:
   ldi r17, $0F
   rjmp skiprest
skiprest:
   pop r18
   ret
```

30. Översätt följande programsnutt i högnivåspråk till assembler. Talen `START` och `SLUT` är heltal som representeras med 8 bitar och ligger lagrade i adresserna `$110` och `$114`. Variabeln `index` lagras lämpligen i ett dataregister:

```
for(index = START; index != SLUT; index++){
...
}
```

31. Uttryck `for`-loopen

```
for(expr1; expr2; expr3){
...
}
```

som en `while`-loop. Går den att skriva som en `do`-loop?

32. Översätt följande programsnutt i högnivåspråk till assembler. Variabeln `x` representeras med 8 bitar i binär kod utan tecken och ligger lagrad i register `r16`.

```
while (x > 10){
    x = x - 5;
}
```

1.3. Programmeringsuppgifter

Här kommer några lite större uppgifter. Tänk på[2]:

> **You don't understand a problem until you can simplify it.**
>
> **— L. Brodie**

33. Beräkna summan av ett antal 8-bitars positiva tal lagrade i minnet med start på en adress som anges av innehållet i `X`. Antalet tal är högst 256. Det sista talet, som skall ingå i summan, är angivet med omvänt tecken. Lagra summan i `r17:r16`, och lagra antalet tal i `r18`.

    Vilka felfall kan inträffa? Vad händer i ditt program om antalet tal är noll? Hur kan man gardera sig mot fel i detta fall?

34. Tillverka en *odometer*, en BCD-räknare med fyra siffror som räknar från `0000` till `9999`. Simulera din kod i Atmelstudio. Använd registren `r20`–`r23` för de olika siffrorna.

    Variant:

    1) Kläm ihop två BCD-siffror i en byte för att spara minnesplats (viktigt i en mikrocontroller).
    2) Lägg siffrorna i SRAM istället. Blev det bättre? Sämre? Lika bra?
    3) Vilka ändringar behövs för att räkna till något annat maxvärde än `9999`? (`2359` verkar lämpligt för ett digitalur till exempel.)

35. Skriv ett assemblerprogram, `4444.asm`, som avgör vilka operationer *op* $\in \{+, -, *\}$ som löser ekvationen

$$A\,op\,B\,op\,C\,op\,D = N$$

[2] Ur *Leo Brodie, Thinking Forth*

Där talen A–D är givna BCD-kodade siffror och N valfritt, till exempel $A = B = C = D = 4$ och $N = 20$.

Utvärderingen ska ske genom uttömmande sökning[3] i den ordning siffrorna kommer dvs prioriteringsregler kan bortses från. Till exempel utvärderas $A = B = C = D = 5$ dvs "$5 + 5 * 5 - 5$" stegvis till $5 + 5 = 10, 10 * 5 = 50, 50 - 5 = 45$ med slutresultatet 45.

Använd instruktionen `mul` för multiplikation.

Programmet måste således uppräkna alla operationer:

`+++`, `++-`, `++*`, `+-+`, `+-`, `+-*`, osv

1) Rita strukturdiagram och pseudokod.
2) Programmera och simulera i assembler
3) Fundera på hur du skulle avgöra om ekvationen alltid har en lösning. Ändras din metod om enbart siffror ≤ 5 får användas?

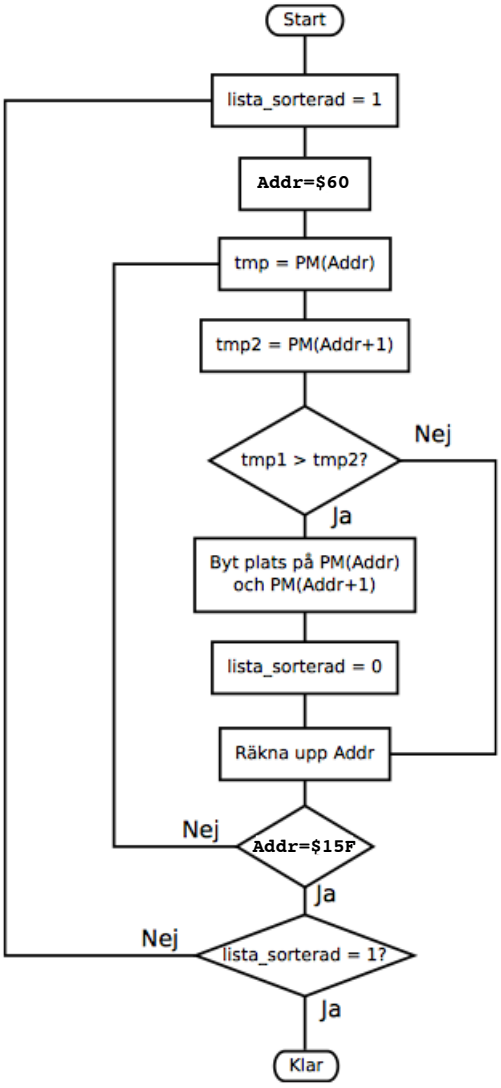
36. Skriv den subrutin som korrekt hämtar sin argumentbyte från stacken om den anropande funktionen placerat argumentet där innan subrutinhoppet. Rita karta över stackinnehållet.

37. Man kan använda pekarna `X`, `Y` och `Z` i assemblerarkitekturen. Men alla kan inte peka på allt. Vilka begränsningar finns?

38. Skriv ett program som sorterar minnesinnehållet i SRAM-adresserna `$60`–`$15F` (255 bytes) i storleksordning. Se till att sorteringen fungerar även om flera värden i minnet är lika.

    Algoritmen *bubbelsort* kan beskrivas i ett flödesschema (PM betyder här *primärminne* dvs vårt SRAM):

[3] Så kallad *brute forcing*



Variant:

1) Översätt flödesschemat till strukturdiagram.
2) Optimera koden så antalet klockcykler blir så få som möjligt
3) Hur ska listan vara beskaffad för att algoritmen ska ta så många cykler som möjligt? Så få som möjligt?

39. Skriv en subrutin som beräknar kvadratroten ur ett sextonbitars teckenlöst tal N som befinner sig i `r25:r24`. `r24` innehåller den lägre halvan av talet och det i denna byte resultatet skall lagras. `r25` skall vara noll vid återhoppet.

    Newton-Raphsons metod är i allmänhet snabb och har kvadratisk konvergens. Tyvärr innehåller den division vilket är mödosamt för vår processor. Processorn har dock en `MUL`-instruktion så man kan skriva en algoritm som provar sig fram:

    Prova bit för bit med början i `$80`, kvadrera och jämför, om kvadraten är större än N kan man fortsätta prova med `$40` osv. Är kvadraten mindre än N fyller man på "provroten" med en bit `$60`, provar igen osv.[4]

40. Här är din uppgift att tillverka en personnummerkollare för svenska personnummer. Som bekant består ett personnummer av födelsedatum och fyra sista siffror. Dessa fyra siffror är utformade så att de två första är ett löpnummer, den därefter följande siffran är jämn för kvinnor och udda för män, och den sista är en kontrollsiffra ($0-9$) som är en funktion av alla tidigare siffror. Här gäller det att beräkna denna kontrollsiffra.

    Exempel: Beräkna kontrollsiffran `x` för nedanstående fiktiva personnummer[5]. Första siffran multipliceras med två och sedan tas tvärsumman av resultatet: $6 \cdot 2 = 12 \rightarrow 1+2 = 3$. Andra siffran multipliceras med ett, tredje med två igen osv. Alla sådana delresultat adderas. `x` är sedan det som behövs läggas till för att summan ska bli nästa tiotal:

    ```
     6 4 0 8 2 3 - 3 2 3 x
     2 1 2 1 2 1   2 1 2 1
     ---------------------
    12 4 0 8 4 3   6 2 6 x
     3+4+0+8+4+3 + 6+2+6=36 + x = 40 ==> x = 4
    ```

    Det korrekta personnumret är alltså `640823-3234`. Övertyga dig om att metoden fungerar och bekanta dig med den genom att prova den på ditt eget personnummer.

    Antag att det personnummer som skall förses med kontrollsiffra ligger `BCD`-kodat i dataminnet på adressen `PNR`. Uppgiften handlar om att komplettera det med sista siffran. Programmet för denna uppgift skall heta `PCHECK`.

    a) Skriv ett `JSP`-diagram över en lösning.

    b) Skriv en subrutin `tvarsumma` som beräknar tvärsumman av en siffra givet ett argument `1` eller `2` som utgör multiplikatorn i algoritmen.

[4] Metoden kan göras överraskande kompakt på AVR-arkitekturen: Under 100 klockcykler för en rotdragning!

[5] Personnumret är ett av skatteverket godkänt nummer för publicering

c) Hur kan programmet ta hänsyn till olika format "`640823-323`", "`640823323`"? Ska det ta hänsyn?

d) Skriv subrutiner för lämpliga deluppdrag i algoritmen.

e) Provkör hela programmet!

f) Ändra programmet så att det börjar med att kontrollera om det tiosiffriga personnumret är korrekt och meddela detta, eller, om det inte är korrekt, meddela detta samt komplettera numret.

g) När rutinen `PCHECK` fungerar: Hur skulle du kunna testa den i labbet? Hur skulle inmatning från ett hexadecimalt tangentbord ske? Hur skulle resultatet presenteras? Skriv motsvarande rutiner och testa i labbet.

41. Antag att komunikationsprotokollet I^{2}C skall implementeras. Vilka åtgärder behöver då tas i funktionerna `I2C_open()`, `I2C_close()`, `I2C_read()`, `I2C_write()` och `I2C_seek()`? Skriv funktionerna i assemblerkod.

42. Antag kommunikationsprotokollet USART. Beskriv vad som kan ingå i `USART_open()`, `USART_close()`, `USART_read()`, `USART_write()` och `USART_seek()`? Skriv funktionerna i assemblerkod.

1.4. In- och utmatning

43. Antag att I/O-registren är laddade med följande data:

Register	Innehåll
DDRA	$F0
DDRB	$0F

Vilka av dataledningarna på portarna `PORTA` och `PORTB` är in- respektive utgångar?

Vilka instruktioner används för att läsa in respektive skriva ut data på dessa portar?

44. Skriv kod för att använda `PORTA` bitar 0, 2, 4 och 6 och `PORTB` bitar 1, 3, 5 och 7 som utgångar och övriga dataledningar som ingångar.

45. I databladen nämns *active pull-up* på vissa I/O-pinnar. Vad betyder det?

1.5. Avbrott

46. Ge exempel på några orsaker som kan resultera i avbrott?

47. Förklara skillanden mellan instruktionerna `ret` och `reti`. Vad händer om man blandar ihop dem?
48. På vilka adresser ligger avbrottsvektorn för A/D-omvandling?
49. Ge exempel på när det kan vara lämpligt att använda externt avbrott. Vad är alternativet?
50. Vad bör man tänka på när man använder interna register i en avbrottsrutin?

1.6. Aritmetik

Uppgifterna XX till YY skall utföras på papper utan processor.

51. I en centralenhet (*Central Processing Unit*, CPU) finns ofta ett statusregister med så kallade flaggor, där man vanligen finner bitarna `C`, `V`, `Z` och `N`. Förklara hur dessa fyra flaggor påverkas av olika instruktioner.
52. Omvandla följande decimala tal till binär representation:

    a) 5 b) 15 c) 25 d) 1000 e) 2047 f) 65 g) 17
53. Omvandla talen i uppgiften ovan till hexadecimal representation.
54. Omvandla följande tal till binär form:

    a) F_{16} b) 67_{16} c) FE_{16} d) $2C_{16}$ e) $6A_{16}$ f) $0A_{16}$ g) $ABCD_{16}$
55. Utför följande binära additioner. Kontrollera resultatet genom att omvandla talen till decimal representation och addera dem i decimal form också.

    a) $10110_2 + 111_2$ b) $100_2 + 110_2$
    c) $111111_2 + 1 + 1 + 1$ d) $1011011_2 + 111001_2 + 1100_2$
    e) $111111_2 + 111111_2 + 111111_2$
56. Beräkna den åttabitars binära tvåkomplementsrepresentationen av följande decimala tal:

    a) -5 b) 25 c) -25 d) 31 e) -100 f) -1 g) -65
57. Hur ser en talbas ut om den uttrycks i den egna basen?
58. Utför följande binära subtraktioner. Betrakta talen som åttabitars tvåkomplements tal. Kontrollera resultatet genom att omvandla talen till decimal representation.

    a) $00010111_2 - 00000011_2$ b) $01011101_2 - 00111110_2$ c) $11010_2 - 1110001_2$
59. På samma sätt som man har tvåkomplement i basen två borde man kunna ha tiokomplement i decimalbasen? Hur skulle man göra då? Vilka tiokomplementkodade tal kan man representera med enbart två siffror?

60. Vilken talbas ges med symbolerna ”0, 1, 2, ..., A, B, ... Z” (dvs 0 till Z ingår)? Vilket decimalt tal är då XYZ?

61. Det binära bitmönstret 11010000 kan tolkas på två sätt. Ange det decimala tal som bitmönstret representerar i följande fall:

    a) Talet representeras som ett tal utan tecken.

    b) Talet representeras på 2-komplementformat.

1.7. Övrigt

Här kommer lite att fundera på om du är extra intresserad och kan C.

1.7.1. Högnivåspråk

På den låga hårdvarunära nivå vi hittills programmerat har det framgått att det är ett mycket tunt lager fernissa som skiljer oss från digitalteknikens brutala verklighet. Man kan förstå att våra processorinstruktioner är ”väl” valda för att motsvara det en programmerare normalt vill kunna utföra utan att se för mycket av den underliggande hårdvaran.

Ett ytterligare steg upp i abstraktionsnivå kan vi få genom att programmera i ett högnivåspråk. I utvecklingsmiljön Atmel Studio kan man även skapa projekt som programmeras i språket C. C-kompilatorn översätter sedan C-programkoden till assembler innan en avslutande kompilering till Dalia-kortet sker.

Skriv om någon laboration i C. Studera den resulterande assemblerkoden.

Frågeställningar:

a) Hur implementeras till exempel `if-`, `for-`, och `switch`-satser i assembler? Kan du göra en bättre implementation? Lek runt med olika variabeltyper (int, char, long, float?) vad händer med assemblerkoden?

b) I ett C-projekt kan graden av kompilatoroptimering göras. -O0 är optimeringsfritt medan -O2 är en vanlig, rätt aggressiv, optimering. Känner du igen din kod efter kompilatorns optimering?

c) Industriellt är i praktiken C allenarådande vid denna typ av lågnivåprogrammering. Varför? Varför tillåts i så fall överhuvudtaget assemblerprogrammering?

A. Lösningsförslag

Förslagen är just förslag, det finns ofta fler än ett sätt att koda en lösning. Ibland anges flera förslag (”3a:”,”3b:”).

```
1:
   lds   r16,$110
   sts   $112,r16

2:
   lds   r16,$110
   lds   r17,$111
   add   r16,r17
   sts   $112,r16

3a:
   lds   r16,$110
   lsl   r16
   lds   r17,$111

3b:
   lds   r16,$110
   clc
   rol   r16
   lds   r17,$111

4:
   lds   r16,$110
   andi  r16,$F0
   swap  r16
   sts   $111,r16

5:
   lds   r16,$110
   mov   r17,r16
   swap  r16
   andi  r16,$0F
   sts   $112,r16
   sts   $111,r17

6: lds   r16,$110
   lds   r17,$111
   cp    r16,r17
   brpl  DONE
   mov   r17,r16
DONE:
   sts   $112,r16

7:
   clr   r18
   lds   r16,$110
   mov   r17,r16
AGAIN:
   add   r18,r16
   dec   r17
   brne  AGAIN

8:
   lds   r16,$110
   ldi   ZL,LOW(BAS*2)
   ldi   ZH,HIGH(BAS*2)
   add   ZL,r16
   lpm   r16,Z
   sts   $112,r16
BAS:
   .db   0, 1, 4,... 196, 225

9:
   lds   r16,$110
   ldi   ZL,LOW(BAS)
   ldi   ZH,HIGH(BAS)
   add   ZL,r16
   ld    r16,Z
   sts   $112,r16

10:
   lds   r16,$110
   mov   r17,r16
   mulu  r16,r17
   sts   $112,r0

11a:
   lds   r16,$110
   ori   r16,$30
   sts   $112,r16

11b:
   lds   r16,$110
   subi  r16,-$30
   sts   $112,r16

12:
   lds   r16,$110
   lds   r17,$114
LOOP:
   ...
   inc   r16
   cp r17,r16
```

```
   brge  LOOP

13:
   expr1;
   while(expr2){
      ...
      expr3;
   {

14:
AGN:
   cpi      r16,10
   brmi     KLAR
   subi     r16,5
   jmp      AGN

15a:
   lds   r16,$101
   subi  r16,1
   breq  ETT
   subi  r16,1
   breq  TVA
   subi  r16,1
   breq  TRE

15b:
   lds   r16,$101
   cpi   r16,1
   breq  ETT
   subi  r16,2
   breq  TVA
   subi  r16,3
   breq  TRE

16--19:
   Se bok

20:
   a) r16=$61
   b) r20=$19
   c) r21=0
   d) r21=$F6
   e) ($100)=$F5
   f) r20=$F5
   g) r17=$19, X=$100
   h) r17=$19, X=$101
   i) r1:r0=FBD5

   .
```

Se kurslitteraturen för uppgifter 21–23.

`24: Tips: Z-pekaren!`

25:

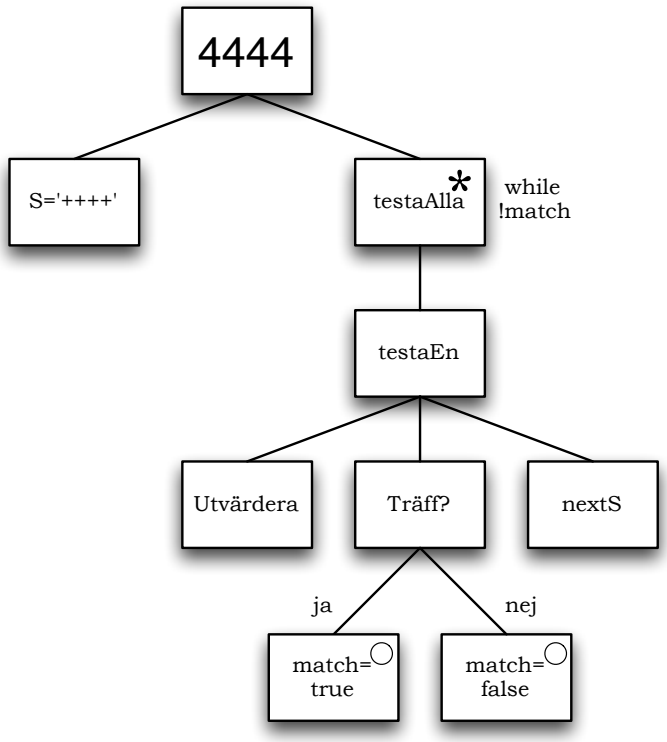
a) Togglar r17:s bitar i r16

b) Bitreverserar byten

c) Summerar ettor i byten

26: Se kurslitteraturen

30: En möjlig JSP-struktur är:



39: Rutinen skall enbart ha en `ret`. Instruktionen `sts` förekommer på två ställen, bara ett behövs. Sammantaget blir den hyfsade rutinen alltså:

```
RACKET1_UP:
   lds   r16,RACKET1
   lsl   r16
   cpi   r16,$C0
   brne  EXIT
```

```
   ori   r16,$E0
EXIT:
   sts   RACKET1,r16
   ret
```

40: Notera att, som koden är skriven, *kan* `r22` bara innehålla `$60`, `$80` eller `$A2`. Uteslutning gör att man i så fall inte behöver jämföra alla tre! Det här verkar suspekt, men så är koden skriven. Enligt JSP skall ett *default*-alternativ alltid finnas.

```
TWI_INTERRUPT:
   push  r22
   in    r22,SREG
   push  r22

   in    r22,TWSR
   andi  r22,$F8

   cpi   r22,$60
   breq  SET_TWINT

   cpi   r22,$80
   brne  SEND_DATA



READ_DATA:
   in    r22,TWDR
   mov   r17,r22
   andi  r17,$07
   andi  r22,$18
   lsr   r22
   lsr   r22
   lsr   r22
   sts   SELECTED_COLUMN,r17
   sts   PLAYER,r22
   call  ADD_DOT
   rjmp  SET_TWINT

SEND_DATA:
   lds   r22,TWI_DATA
   out   TWDR,r22
   ldi   r22,(1<<TWINT)|
             (1<<TWEN)
   out   TWCR,r22

SET_TWINT:
   ldi   r22,(1<<TWINT)|
             (1<<TWEN) |
             (1<<TWEA) |
             (1<<TWIE)
   out   TWCR,r22

TWI_FINISHED:
   pop   r22
   out   SREG,r22
   pop   r22
   reti
```

Notera att den är en avbrottsrutin: Spara statusregistret och skapa ”lokala variabler”.

Läs av `TWSR`, maska, och betrakta innehållet `$60`, `$80` eller `$A2` om det inte var de två förra. Brancha ut till respektive kodstycke:

Om `$60`: Hoppa

Om **inte** `$80` (dvs `$A2`), hoppa till `SEND_DATA` i annat fall fortsätt i `READ_DATA`:

:
Binära konstanter ersatta med hexadecimala.
:
:
:
:

:
Hoppa ur via `SET_TWINT`.
:
:
Gör `SEND_DATA`.
:
:
:
:
Gemensam *exit* genom `SET_TWINT`.

Återställ innan återhopp från avbrottet. Notera en enda *exit*-punkt.
Klar!

Det kan underlätta att rita programflödet med linjer och pilar för att avslöja en alternativ och effektivare programstruktur.

Med koden i detta skick kan det nu vara värt att studera själva funktionen och intruktionerna igen. Det är inte ovanligt att optimeringsmöjligheter öppnat sig när man ser koden med nya ögon efter omstrukturering.