Datorteknik
Programmering av AVR

Michael Josefsson

Version 0.961L 2020-10-31

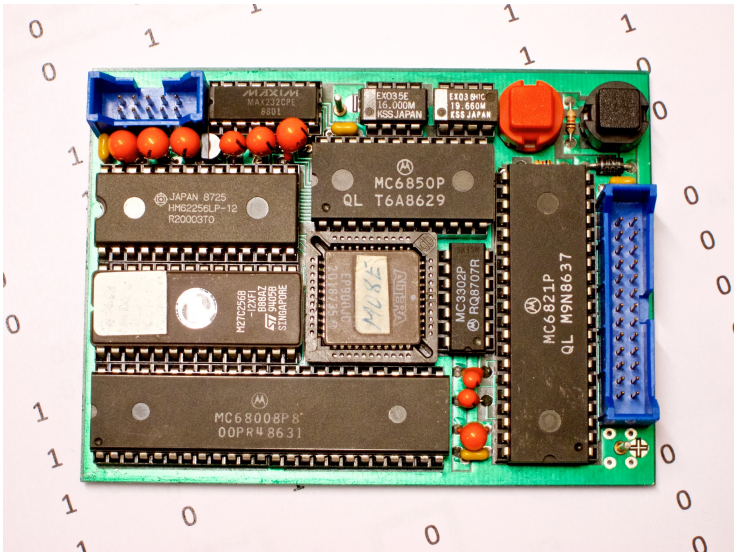
Innehåll

0. Inledning

Ingen processor är exakt den andra lik. Det förekommer skillnader beroende på vad syftet med processorn är, men även bakomliggande filosofier och tillgänglig produktionsmetod inverkar på slutprodukten. När mikroprocessorerna började sitt egentliga intåg på marknaden var de ofta lätt ”egendomliga” i sitt utförande och programmering. Detta berodde bland annat på den dåtida svårigheten att få plats med tillräckligt många transistorer på ett chip för att överhuvudtaget få önskad funktion. Det förekom processorer vars funktioner var fördelade i olika kretsar till exempel.

Utvecklingen gick dock fort och under slutet av 1970- och början av 1980-talen fanns det fullfjädrade och fullt användbara integrerade processorer. För att kunna använda dessa behövdes i princip bara minneskretsar och lite kringelektronik anslutas för en komplett dator.

Här på LiTH användes under många år en sådan processor i de grundläggande datorteknikkurserna för alla studenter på ingenjörsutbildningarna. Processorn var en Motorola M68008 och den befann sig på ett specialtillverkat kretskort komplett med minne, adressavkodningslogik, nödvändiga kommunikations- och klockkretsar som gick under namnet *Tutorkortet*.



Tutorkortet som användes i laborationerna bestod huvudsakligen av processorn MC68008P8, minnena 27C256 och 62256 samt kommunikationskretsarna 6850 och 6821. Allt sammanhållet med den programmerbara logiska kretsen (CPLD) EP900 i mitten.

Processorutvecklingen slutade som bekant inte på 1980-talet och med ökad miniatyrisering blev den så kallade *mikrokontrollern* mer populär. En mikrokontroller är ett chip som innehåller allt det Tutorkortet innehöll och mer därtill.

Typiskt innehåller en mikrokontroller en mer eller mindre avancerad processorkärna (numera ofta av RISC[1]-typ) men också programminne, dataminne och en högst diversifierad uppsättning kringkretsar för många olika syften. Det är till exempel inte ovanligt att mikrokontrollern innehåller egen processorklocka (ställbar i olika frekvenser),

[1] RISC står för *Reduced Instruction Set Computer* vilket innebär att antalet instruktioner, men framförallt adresseringsmoder och påföljande kostsamma avkodning, är reducerad jämfört med till exempel M68008 och andra.

kommunikationsenheter för seriell kommunikation i olika protokoll (USART, I^{2}C/TWI, SPI), så kallade *timers*, analog-digitalomvandlare och så vidare. Det finns normalt dessutom ett stort antal varianter av samma processorkärna med varierande mängder kringenheter!

Vi ska i kursen använda en mikrokontroller[2] ur AVR-serien, ATMEGA16A. Denna är en typisk exponent för vad en modern mikrokontroller kan innehålla då den bland annat innehåller de ovan uppräknade hårdvaruenheterna. I kursen kommer vi främst att använda den för att administrera digitala in- och utsignaler. En stor del av detta häfte beskriver hur den programmeras på sin lägsta nivå i *assembler*.

Assembler är det mest primitiva språket som kan användas för att programmera processorn. *Primitivt* betyder inte i detta fall att det nödvändigtvis är särskilt enkelt, men genom det hårdvarunära assemblerspråket vi kan få *full* tillgång till hela processorns kapacitet. Inget är tillrättalagt utan vi är alltid helt ansvariga för allt som händer.

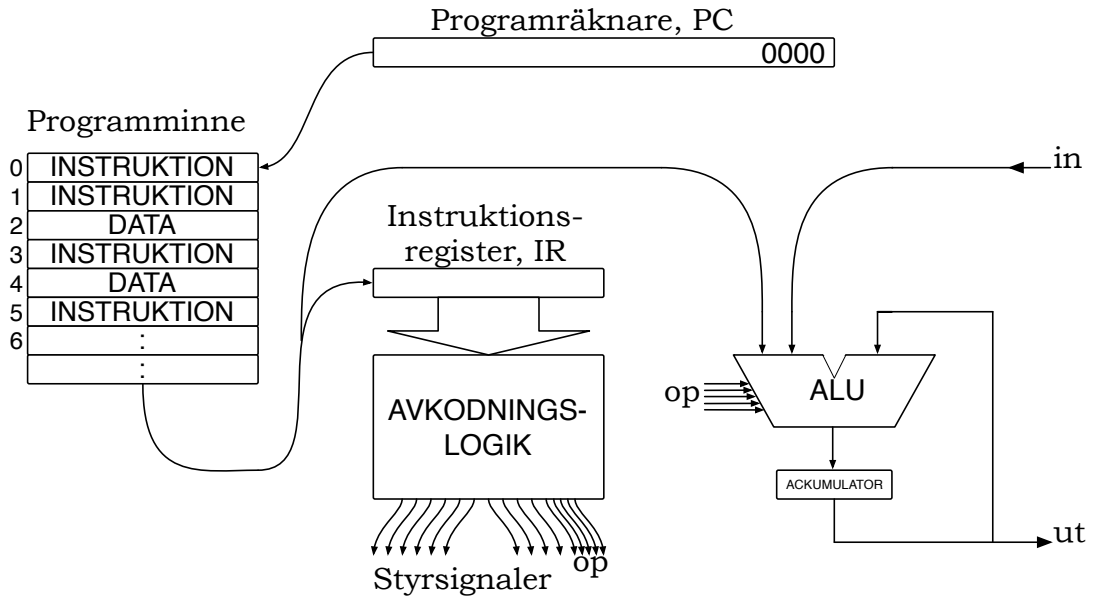
Detta betyder att du som student med nödvändighet bibringas en fundamental förståelse för hur processorer fungerar. Denna kunskap är till glädje inte bara i senare projektkurser utan också i programmeringskurser i högnivåspråk.[3] Även kurser i kompilatorkonstruktion kan ha målet att generera instruktioner för en underliggande processor och då är det återigen assembler det handlar om.

Olika processorer har olika assemblerspråk men likheterna är större än skillnaderna och framförallt är det samma *typ av operationer* som förekommer oavsett processortyp.

Även om processorn i sig är belägen i en mikrokontroller lämpar den sig för undervisningsändamål då den innehåller flera funktioner som återfinns i en modern processor men samtidigt är såpass enkel att man kan förstå den i detalj. Förstår vi hur denna mikrokontroller fungerar är ingen annan processor obegriplig för oss. Även om detaljer kan avvika inte så lite mellan olika processormodeller är det stora program- och dataflödet ungefär lika oavsett det gäller en liten, enkel eller stor, komplicerad processor.

0.1. Datormodell

För att förstå vilka delar en dator måste bestå av ska vi först presentera en enkel datormodell. Det är viktigt att förstå datorns huvuddrag och hur databehandlingen går till för att uppskatta de olika delarna. Alltså, vår datormodell:



[2]Orden mikrokontroller och processor kommer i denna text användas omväxlande och betyda emellanåt samma sak även om det strängt taget är tokigt.

[3]Det visar sig till exempel att begreppet *pekare* av somliga studenter enklast förstås i assemblermiljön.

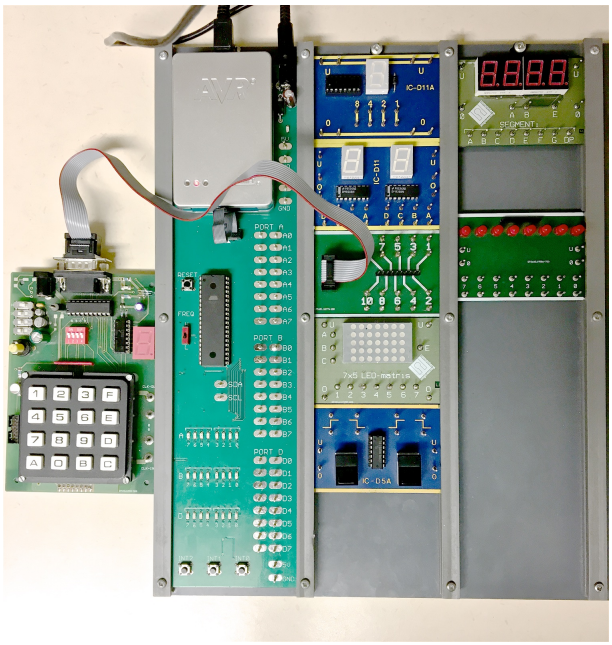
Den här datormodellen tjänar till att presentera det generella förloppet i ett datorsystem. Vi förstår att en del komponenter är nödvändiga. Det finns en *programräknare* som pekar ut adressen till nästa instruktion, ett *instruktionsregister* där instruktionen hamnar efter inläsning och en *ALU/ackumulator* där själva databehandlingen sker. Det finns dessutom *in/ut-enheter* för att kunna kommunicera med omvärlden.

Vi märker snart att det även behövs *minne* för att lagra program och/eller data och det minnet kan befinna sig internt i processorn eller externt i processorns närhet. Redan nu kan vi kanske förstå att det är lyckosamt att ha åtminstone en del minne internt i processorn för att inte behöva prata med ett yttre minne ideligen. Specifikt för en mikrokontroller brukar hela minnet vara beläget i själva kapseln och inte tillgängligt utifrån, ej heller brukar man kunna ansluta ytterligare externt minne till en mikrokontroller. Undantag finns dock.

Uppgiften att konstruera en effektiv och ändamålsenlig uppsättning instruktioner och en lämplig datorhårdvara är svår. Det är därför med stor glädje vi ser att flera tillverkare redan gjort det jobbet åt oss. Det återstår för oss att förstå deras datormodell och instruktioner, något som brukar kallas datorns *arkitektur*.

0.2. Dalia

Alla laborationer i kursen handlar om att programmera en ATMEGA16A på det sk DALIA-kortet med kringkretsar. DALIA innehåller, förutom mikrokontrollern, kontaktstift för enkel anslutning av önskad kringutrustning, programmeringsanslutning via USB till värddator och lysdioder som representerar de olika in- och utsignalernas digitala värde.

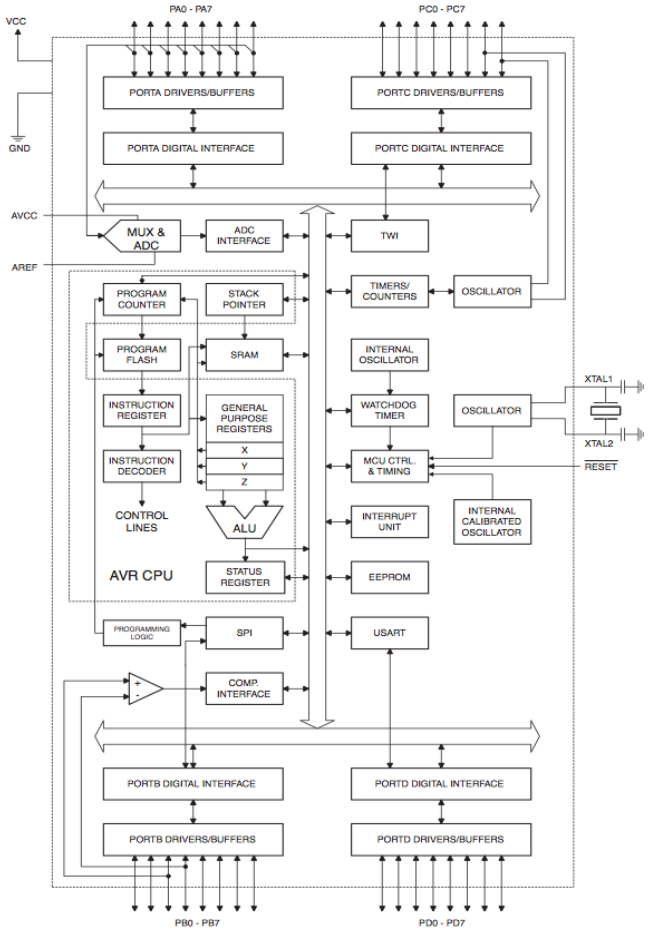


Figur 0.1.: Samtliga laborationer i kursen utförs på det egenutvecklade *DALIA*-kortet (instickskortet till vänster). I bilden återfinns även ett hexadecimalt tangentbord (här anslutet med flatkabel, men den kan också sända information med infrarött ljus). Vidare ser man bland annat sjusegmentsdisplayer, tryckknappar, och en liten spelplan i form av en LED-punktmatris.

1. Programmerarmodell

Alla laborationer i kursen utförs på på mikrokontrollern ATmega16A. Detta är en komplett liten dator med AVR-processorn som centralenhet. Innan laborationerna börjar måste vi förstå hur mikrokontrollern fungerar och speciellt hur man programmerar den att utföra det vi begär av den.

Mikrokontrollern med dess registeruppsättning, programräknare och statusregister brukar kallas processorns *programmerarmodell*. Det är den bild av processorns innanmäte man behöver för att assemblerprogrammera den.



1

Figur 1.1.: *Mikrokontrollern ATmega16A består av en processorkärna "*AVR CPU*" med kringenheter. De mest grundläggande av dessa är de så kallade portarna,* PORTA–D*, som används för att skicka ut respektive läsa in digitala signaler.* (Figuren tagen ur processorns datablad, 2466D–A VR–09/02, Atmel.com)

Processorn är en åtta-bitars processor vilket betyder att alla register och all data är åtta bitar breda. I de fall åtta bitar inte räcker till måste vi använda flera register bredvid varann. Vissa register är av tillverkaren förberedda för detta men i allmänhet är vi ensamma på den fronten.

Processorkärnans huvudsakliga beståndsdelar är:

- **Program Counter** (*Programräknare*, PC). PC används för att peka ut nästa rad i programminnet. Programminnet kan innehålla upp till (16K) 16384 rader programkod och blott ett åttabitars register räcker inte till för att kunna peka ut alla dessa. I detta fall använder tillverkaren således två register som tillsammans utgör det 16-bitar breda registret PC.
- **Program FLASH**. Detta är att betrakta som ett enbart *läs*minne. För att kunna skriva i minnet måste man i allmänhet använda en speciell programmeringsutrustning. Här används minnet för att innehålla ett fast program och det är din uppgift att skriva detta program. Programmet är mikrokontrollerns beteende utåt. Vid spännings*påslag* börjar programmet köras från rad 0 i FLASH-minnet. Vid spännings*frånslag* behålls innehållet i detta minne.
- **Instruction Decoder** Instruktionsavkodaren tolkar instruktionerna och utför dem. Processorn kan bara hämta instruktioner från FLASH-minnet.
- **SRAM** (*Static Random Acccess Memory*) Detta är ett *både* läs- och skrivminne. Här kan processorn både läsa och skriva *data*. Minnet är *flyktigt*, med vilket menas att det tappar sitt innehåll vid spänningsfrånslag. I och med att detta minne ligger inne i själva mikrocontrollern blir läsning och skrivning snabb (en klockcykel).
- **ALU och Status Register** Mikrokontrollerns egentliga databehandling sker i den *AritmetiskLogiska Enheten* (ALU). ALU:n kan addera, subtrahera, utföra logiska operationer och så vidare. Statusregistret innehåller information om senast exekverade instruktionens resultat.
- **General Purpose Registers** Dessa register är arbetsregistren som ALU:n manipulerar med sina aritmetiska och logiska operationer.

Runt om processorkärnan återfinns ett antal *hårdvaruenheter* fördelade längs en gemensam *buss*. För att hantera digitala[2] in- och utsignaler används någon av de tre *portarna* PORTA, PORTB eller PORTD.[3]

Varje port innehåller ett antal in-/utsignaler, dessa signaler återfinns på pinnar längs kapselns utsida. Varje pinne kan konfigureras som antingen ingång eller utgång och utgör processorns sätt att kommunicera med omvärlden.

Vi kommer under kursen att kunna konstatera att mycket av databehandlingen i en processor går åt till att utföra, var för sig tämligen enkla operationer. En traditionell processor ser ungefär ut som den arkitektur vi hittills sett.[4]

1.1. Register, minne och instruktionsformat

I programmerarmodellen ingår en beskrivning av de register och minnen processorn består av. Samtliga är 8 bitar, en *byte*, breda. En mer komplett programmerarmodell innehåller dessutom processorns instruktions- och dataformat samt ingående instruktioner.

Här kommer vi i tur och ordning betrakta minnet, registren och instruktionsformatet.

1.1.1. Minne

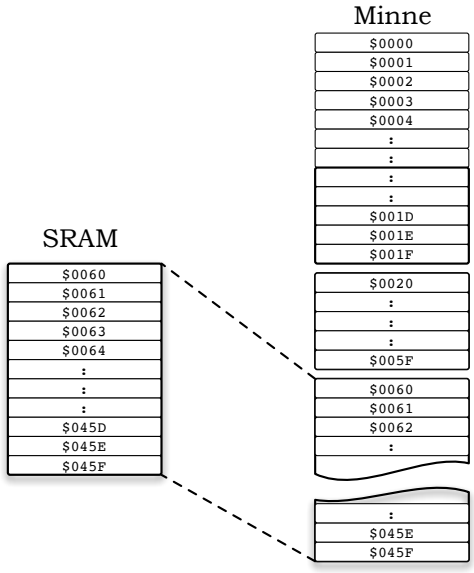
Processorns kan adressera 64 kB minne med en adress om 16 bitar ($2^{16} = 65536$). Enbart adresserna `$0000`–`$045F` är dock bestyckade med hårdvara.

Den del av adressrymden som innehåller det skriv- och läsbart minnet, SRAM:et (*Static Random Access Memory*), är `$0060`–`$045F` och visas i figuren nedan:

[2] Här spänningarna +5 V eller 0 V.

[3] PORTC används för programmering av mikrokontollern och är inte åtkomlig i laborationerna.

[4] Faktum är att 90% eller mer av alla processorer som tillverkas idag ser ut ungefär som den. Det finns variationer i den exakta uppsättningen instruktioner, men den sortens processorer med så kallad *ackumulatorarkitektur* är förhärskande.

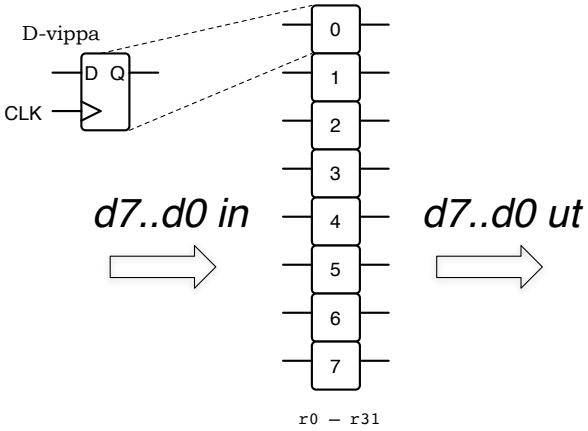


I denna processor utgör SRAM 1024 bytes, det vill säga 1024 adresserbara rader à 8 bitar (en byte).

1.1.2. Generella register

De generella registren (*General Purpose Working Registers* belägna i *Register File*) är de register som allmänt används för att manipulera data. Av bekvämlighetsskäl benämner vi dem bara ”register”. I denna processor finns 32 register av denna typ, `r0–r31`, samtliga är åtta bitar breda.

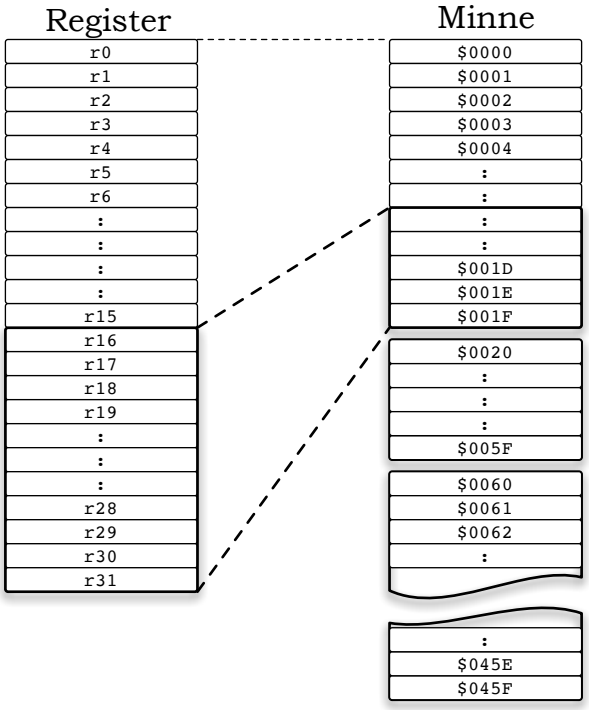
Från digitaltekniken vet vi att en D-vippa kan lagra en bit så varje register kan tänkas bestå av åtta D-vippor:



Trots namnet ”generellt register” är alla register är inte skapade lika. Alla processorns aritmetiska och logiska operationer kan utföras på registren `r16–r31`, medan bara en del av dem[5] kan utföras på `r0–r15`. I kodexemplen kommer vi använda `r16–r31` nästan uteslutande.

Registren återfinns i processorns minne som en del av processorn totala adressrymd med adresser enligt nedan.

[5]Instruktionen `and` kan till exempel utföras mellan vilka två register som helst, men `and` med en konstant kan bara utföras med `r16–r31`.



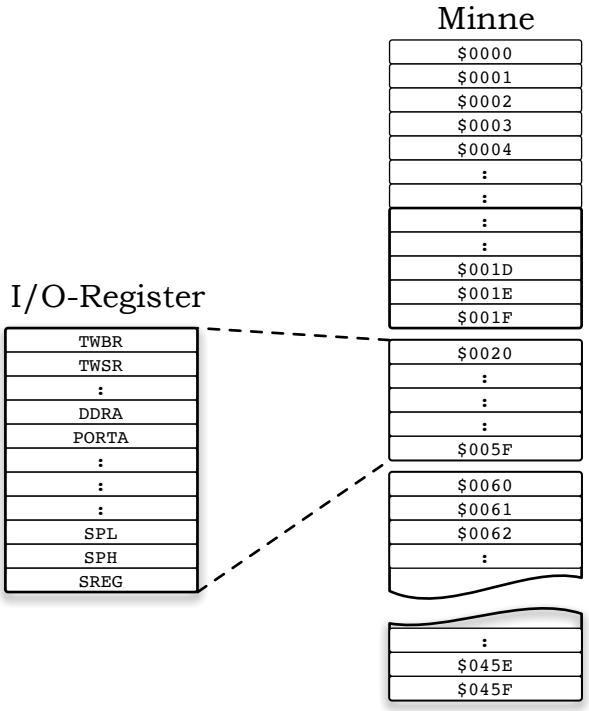
En slutsats av detta minnesupplägg är att register `r0–r31` också kan nås genom minnesadresserna `$0000–$001F`.

Detta kan vi dra nytta av senare när vi tittar på olika adresseringsmoder. Man är alltså inte tvingad att använda ”`rXX`” som argument utan kan också (med rätta instruktioner) använda direkta minnesadresser.

1.1.3. I/O-register

Processorns egentliga databehandling utförs i de generella registren, men det finns andra register som är minst lika viktiga för att processorn skall kunna utföra sitt arbete.

I vår arkitektur är *I/O-registren* vanligen knutna till de olika hårdvaruenheter som är belägna runt själva processorkärnan. Det är genom I/O-registren som all kommunikation till dessa enheter och även omvärlden sker. Även dessa register adresseras bytevis.



I/O-registren, fyller upp minnesadresserna mellan `$0020–$05F`. Vanligen nås dem inte genom dessa adresser utan med sina namn `TWBR`, `TWSR` osv.

Yttervärlden nås genom registren `PORT`*x*.

Registren *statusregistret* (`SREG`) och *stackpekaren* (`SP`) hanteras också som I/O-register i denna processor. Registret `SP` består av två delar `SPH` och `SPL` för dess `H`öga och `L`åga byte. Vi tecknar detta förhållande som `SP=SPH:SPL`.

1.1.4. Programräknaren PC

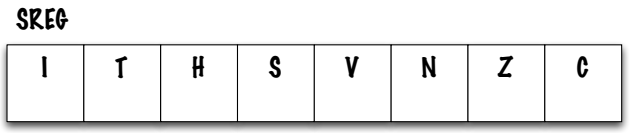
Registret PC innehåller adressen till den instruktion som utförs för tillfället. När programmet körs rad efter rad kommer programräknaren att ökas på allteftersom. Om man gör ett hopp i programmet laddas programräknaren med en ny adress. Programräknaren är 13 bitar

1.1.5. Statusregistret SREG

Detta register innehåller information (så kallade *flaggor*) om den senaste instruktionens resultat. Speciellt intresserade är vi av bitarna Z, N, C och V. Dessa sätts eller nollställs av flera instruktioner. Flaggorna återspeglar resultat från den aritmetisk logiska enheten ALU.

Flagga	Innebörd
Z	1 om resultatet lika med noll
N	1 om resultatet mindre än noll, det vill säga mest signifikant bit satt
C	1 om *Carry*, det vill säga minnesbiten vid beräkningar vid teckenlösa tal satt
V	1 om *oVerflow*, om felaktigt resultat vid beräkningar av *2-komplementkodade* tal

Dessa fyra enskilda flaggors placering till höger i SREG framgår av bilden nedan. Man behöver inte veta flaggans position i registret för att kunna använda den i en instruktion.



Figur 1.2.: *Statusregistret SREG innehåller flaggor som indikerar senast utförda instruktions resultat. Speciellt är vi intresserade flaggorna för oVerflov, Negative, Zero och Carry.*

Informationen i flaggorna är enda sättet för en instruktion att meddela sig till nästa instruktion.

Villkorliga hopp använder till exempel dessa flaggor från den tidigare instruktionen för att avgöra om hopp skall ske eller inte.

För att summera egenskaperna hos registren:

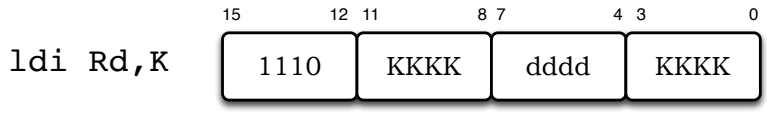
- r0–r15 kan mest användas som lagringsplatser,
- r16–r31 är de egentliga generella registren,
- I/O-registren används för att kommunicera med hårdvaruenheter och stackpekareregistret SP.

Instruktionsformat

Med instruktionsformatet menas hur instruktionerna ser ut i programminnet, här FLASH-minnet, och vad de olika delarna av instruktionen betyder. Vi inte går här inte in på alla detaljer, den intresserade hänvisas till processorns omfattande datablad.

I AVR-arkitekturen är en instruktion oftast två bytes bred. Dessa 16 bitar avgör vilken instruktionen är samt eventuella argument.

Till exempel är instruktionen `ldi Rd,K` uppdelad i tre fyrabitars bitfält där de inledande fyra bitarna `1110` anger att det är en `ldi`-instruktion, destinationsregistret `Rd` anges med de fyra bitarna `dddd` och den åttabitars konstant som instruktionen behöver återfinns i de resterande bitarna `KKKK`.



På liknande sätt kodas alla övriga instruktioner. En del instruktioner är fyra bytes stora då de behöver längre argument än de åtta bitarna ovan.

2. Instruktioner

Innan vi börjar programmera måste vi studera de olika *instruktioner* vi kan använda. Programmet byggs upp av instruktioner som utförs i sekvens. Det finns också instruktioner som bryter sekvensen, så kallade hoppinstruktioner.

I databladet för mikrokontrollern återfinns drygt hundratalet instruktioner. Vid en första anblick kan denna mängd instruktioner te sig lätt skräckinjagande. Vi kan dock snart konstatera att de är grupperade i fem huvudgrupper:

- Grupp 1. Instruktioner som flyttar data (`ldi`, `mov`, ...)
- Grupp 2. Aritmetiska instruktioner (`add`, `addw`, `subi`, ...)
- Grupp 3. Logiska instruktioner (`asl`, `ror`, ...)
- Grupp 4. Hoppinstruktioner (`jmp`, `br`xx, `call`, ...)
- Grupp 5. I/O-instruktioner (`out`, `in`)

Normalt behöver vi inte använda *alla* instruktioner ur alla grupper. De vanligast använda är kanske ett tjugotal av dessa.

2.1. Grupp 1 — Flytta data

Till, och mellan, dataregistren flyttar man data med `ldi` och `mov`.

Exempel: `ldi`

Flytta konstanten `23` till `r16`. Detta brukar uttalas att ”ladda `r16` med `23`”.

```
ldi r16,23        ; Load Immediate
```

■

Vi kan dessutom omedelbart nämna att:

1. `23` är värdet 23 med decimal bas.
2. Det är en smaksak om man skriver `LDI` eller `ldi`, betydelsen är densamma.
3. Ordningen på de två operanderna runt kommatecknet är sådan att resultatet går till vänster. Somliga tycker detta är naturligt ty det liknar uttrycket `r16=23`, fast utan ”=”.[1]
4. Allt efter ”`;`” tolkas som en kommentar.
5. *Immediate* är en *adresseringsmod*, ett sätt att peka ut operanden.

[1] Andra tycker annat!

Exempel: `mov`

Mellan register används instruktionen `mov`. Flytta innehållet i register `r16` till register `r13`.

```
mov   r13,r16        ; Move Direct
```

Lägg märke till hur instruktionen innehåller information om ■

- *varifrån* operanden läses,
- *vilken* operation som skall utföras och slutligen pekar ut
- *vart* resultatet skall.

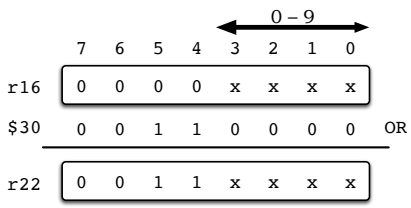
Notera att ordet `mov`, *flytta*, är strängt taget missvisande, eftersom det aldrig flyttas något utan bara kopieras. Innehållet i `r16` påverkas alltså inte av instruktionen. Ingen av flytt-instruktionerna påverkar heller flaggorna i `SREG`.

2.2. 8-bitars register

I denna processor, med ordbredden 8 bitar, används registren normalt som 8-bitarsregister och kan således husera ett binärt tal mellan `0000 0000` och `1111 1111`, motsvarande ett decimalt tal mellan `0` och `255`.

Exempel: Decimalsiffra till ASCII

Registret `r16` innehåller en siffra $0-9$. Översätt denna siffra till dess ASCII-representation[2] och placera resultatet i `r22`.




```
ldi r22,$30
add r22,r16

mov r22,r16
ori r22,$30
```

■

2.3. 16-bitars register

I vissa fall behövs dock bredare register för att kunna ange högre siffervärde. I detta fall använder man två register parallellt och får ett 16-bitarsregister. En sådan registerkombination kan då innehålla ett tal mellan 0 och $2^{16} - 1 = 65\,535$.

Exempel: `adiw`

Öka på `r25:r24` ett steg.

`adiw` fungerar enbart på registerparen `r25:r24`, `r27:r26`, `r29:r28` och `r31:r30`.

```
adiw   r24,1
```

[2] Se Appendix A för en ASCII-tabell.

eller för andra registerpar:

```
        inc     r16     ; affects Z but not C
        brne    DONE
        inc     r17
DONE:
```

■

För att med ett register peka ut en adress i minnesarean måste adressen befinna sig i ett av tre *adress-* eller *pekarregister*. SRAM befinner sig mellan adresserna `$060`–`$45f`. I allmänhet får alltså inte en adress plats i bara en (1) byte utan tar två bytes i anspråk.[3]

I AVR-arkitekturen har man löst detta genom att använda registren `r27:r26`, `r29:r28`, `r31:r30` parvis till tre sextonbitars pekarregister `X`, `Y` och `Z`. Inga andra generella register kan användas på detta sätt. Adressregistren är speciella då de är de enda som kan användas för att peka ut data i SRAM.

Exempel: `st/sts/ld/lds`

Adressen `$102` innehåller ett 8-bitarstal. Invertera alla bitar i talet (*ett-komplementera* det).

Då en rad i SRAM är åtta bitar bred måste talet ligga på en enda adress. Ett-komplementeringen förlitar sig på en logisk operation av något slag. I AVR-processorerna gör instruktionen `com` det vi vill.

För att lagra ett värde i en adress i SRAM används instruktionen `sts` (*Store to SRAM*) och för att läsa innehållet i en adress `lds` (*Load from SRAM*).

```
Utan pekare             Med pekare

lds     r16,$102        ldi   ZH,HIGH($102)
com     r16             ldi   ZL,LOW($102)
sts     $102,r16        ld    r16,Z
                        com   r16
                        st    Z,r16
```

■

Övning: Försök själv att lösa exemplet med andra instruktioner. Vinner man något på det?

2.4. Grupp 2 — Aritmetiska operationer

Bland de aritmetiska instruktionerna har vi träffat på `add`. Naturligtvis finns en motsvarande `sub` för subtraktion och dessutom de ovanligare `mul` och `fmul` för multiplikation respektive fixpunktsmultiplikation.[4]

Exempel: `add`

Addera de två 16-bitarstalen som finns i `r21:r20` och `r17:r16` till `r17:r16`. `r16` och `r20` antas innehålla minst signifikant bit.

```
add r16,r20     ; ingen ingående carry
adc r17,r21     ; med carry
```

■

[3] En enstaka byte kan peka ut adresserna `$00` till `$ff`, med två byte kan man peka ut adresserna `$0000`–`$ffff`.

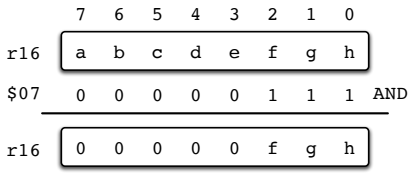
[4] De senare finns dessutom i olika varianter beroende på om operanderna är båda tvåkomplementskodade, båda icke-tvåkomplementskodade eller en av varje.

2.5. Grupp 3a — Logiska operationer

Av logiska operationer har vi redan använt ett-komplementinstruktionen `com` för att invertera alla bitar i ett register. Övriga logiska operationer utför booleska operationer på registerinnehåll bit för bit. Sålunda återfinns `and`/`andi`, `or`/`ori`, `com` och `eor` (för XOR) bland dessa. Sedan digitaltekniken vet vi vad dessa booleska operatorer gör, det som skiljer nu är att de utförs på hela register.

Exempel: Maska bitar

Använd instruktionen `andi` för att skilja ut de tre lägsta bitarna ur byten på adress `$120`.



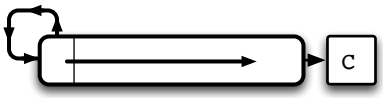

```
lds     r16,$120
andi    r16,$07
```

■

2.6. Grupp 3b — Skiftinstruktioner

I gränslandet mellan de aritmetiska och logiska operationerna befinner sig skiftinstruktionerna. Det finns egentligen fyra skiftinstruktioner men då två av dem sammanfaller behöver vi bara lära oss tre: två högerskift och ett vänsterskift.

`asr` *Arithmetic Shift Right*, Aritmetiskt högerskift.



`lsr` *Logical Shift Right*, Logiskt högerskift.

`lsl` *Logical Shift Left*, Logiskt vänsterskift.

Vill man ha ett aritmetiskt vänsterskift (”`asl`”) kommer detta sammanfalla med det logiska och de båda vänsterskiften ser likadana ut varför enbart `asr`, `lsr` och `lsl` är implementerade i mikrokontrollern. Dessa instruktioner skiftar alla bitar i det angivna registret ett (1) steg.

2.7. Grupp 4 — Hoppinstruktioner

Hittills har vi inte kunnat utföra hopp i vår programkod. Alla program har bara kunnat ”rinna på” uppifrån och ned. Med hopp kan programkörningen avslutas på en programrad och påbörjas någon helt annanstans. Det finns två sorters hopp *villkorliga* och *ovillkorliga*.

Som man förstår av namnet kommer de ovillkorliga hoppen att alltid hoppa medan de villkorliga först måste fatta ett beslut om hoppet ska tas eller inte. För att välja mellan olika alternativ kommer vi se att flaggorna i statusregistret spelar en central roll.

Exempel: Ovillkorligt hopp

Använd ovillkorligt hopp för att hoppa i programmet.

```
A:
       instr
       instr
       jmp B
       :
       :
B:
       instr         ; hit gick hoppet
       :
```

■

Här kan hopp ske med någon av instruktionerna `rjmp` eller `jmp`, dessa hoppar alltid. `A` och `B` är symboliska adresser, *labels*, och är i detta fall destinationen för hoppen. Hoppinstruktionerna påverkar inte flaggor.

Med symboliska adresser behöver man som assemblerprogrammerare inte behöva känna till eller ens bry som om till vilken faktisk adress hoppet kommer ske. Den detaljen tar kompilatorn hand om.

Vi fortsätter med ett programexempel där ett villkorligt hopp måste göras.

Exempel: Villkorligt hopp

I minnesadresserna `$9E` och `$9F` finns två teckenlösa tal, placera det största talet i minnet på adress `$A0`.

```
       lds  r16,$9E
       lds  r17,$9F
       cp   r16,r17     ; (r16-r17), N flag
       brmi R17BIG      ; branch on minus, r17 biggest
       mov  r17,r16     ; r16 was biggest
R17BIG:
       sts  $A0,r17     ; r17 biggest
```

■

Instruktionen `lds` känner vi igen sedan tidigare som den instruktion som läser från en adress i minnet (SRAM), `sts` skriver på motsvarande sätt till minnet.

Ny är däremot instruktionen `cp`, *compare*, som gör en jämförelse mellan två register genom att subtrahera den ena från den andra: `r16-r17` och *kasta bort resultatet*! Inga register påverkas av `cp` men statusflaggorna innehåller information om resultatet.

För *tvåkomplementskodade* kodade tal kan instruktionerna `cp` och `brpl`/`brmi` användas om man noterar:

- Om resultatet är noll är de båda registren *samma* och `Z`-flaggan sätts,
- Om resultatet är negativt är innehållet i `r17` *större* än innehållet `r16` och `N`-flaggan sätts,
- Om resultatet är positivt är innehållet i `r17` *mindre* än innehållet `r16` och `N`-flaggan nollställs,

Villkorsinstruktionen `brpl`, *Branch on Plus*, används sedan för att avgöra om ett hopp skall ske eller inte. För negativa tal används på motsvarande sätt `brmi`, *Branch on Minus*.[5]

I dessa fall kan man alltså dra ut information om jämförelsen genom att studera en flagga var för sig. Genom att kombinera negative-flaggan (`N`) och overflow-flaggan (`V`) får man de inbyggda hoppinstruktionerna `brge`, `brlt` (*greater or equal*, *less than zero*) för `N`$\oplus$`C=0` repsektive `N`$\oplus$`C`=1.

För *teckenlösa* tal kan flaggorna `Z` och `C` användas som i exemplet nedan.

Exempel: Bracket

Skriv kod som sätter `Z` om ett tal i `r16` är $[0..9]$. Teckenlösa tal förutsätts.

```
BRACKET:
    cpi     r16,0           ; r16 - 0
    brcs    BRACKET_DONE    ; C=1 Z=0 om r16<'0'
                            ; C=0 Z=1 om r16='0'
                            ; C=0 Z=0 om r16>'0'
    ; was at least 0
    cpi     r16,9           ; r16 - 9
    brcc    BRACKET_DONE    ; C=1 Z=0 om r16<'9'
                            ; C=0 Z=1 om r16='9'
                            ; C=0 Z=0 om r16>'9'
    ; and at most 9
    sez                     ; set zero flag
BRACKET_DONE:
    ret
```

(I sista jämförelsen (`brcc`): Om `C=1` ramlar vi genom och sätter `Z` innan retur, om `C=0` och `Z=1` är däremot `Z` redan satt så det är OK att lämna på `C=0`.)

Det finns ett drygt dussin olika hoppinstruktioner för ytterligare villkor som *same or higher*, *lower*, *greater than or equal to* med flera. Konsultera mikrokontrollerns datablad för detaljerna. De flaggor som används absolut mest är `Z` och `C`. Dessa måste man kunna utantill.

> *Tips: Det är ofta enklare att jämföra "på träff" istället för mer svepande "större eller mindre än eller lika med". En fördel med heltal överhuvudtaget, och tydligt i vårt fall, är att heltalen i registren är uppräkningsbara så exakt jämförelse är ofta möjlig![6]*

Loopar Ett vanligt förekommande programmeringsfall är att man räknar ner en variabel, eller ett register, och vill hoppa då denna antingen blivit noll eller hoppa så länge operanden är skild från noll.

För detta kommer *decrement*-instruktionen, `dec` , väl till pass. Det är både enklare och snabbare att räkna ner mot noll (`Z`-flaggan förstås!) än att räkna från noll upp till ett givet värde. Motsvarande *increment*-instruktion är `inc`.

[5] Vilken flagga används?

[6] Jämförelse mellan olika flyttal är svårare: När är talen x och $x + \epsilon$ lika? När $\epsilon = 10^{-9}$ eller $\epsilon = 10^{-20}$? Vilket ϵ kan överhuvudtaget representeras? Svårt det där med flyttal.

Exempel: Beräkna summan

$$\sum_{1}^{255} = 1 + 2 + 3 + \ldots + 255$$

och lagra resultatet i `$2D2`.

Under förutsättning att vi förstår att resultatet inte får plats i en byte, men väl i två, beslutar vi att tilldela registren `r20` = varvräknare(index) och `r22:r21` = summan.

```
    clr     r22         ; sum = 0
    clr     r21
    ldi     r20,255     ; index
AGAIN:
    add     r21,r20     ; sum += index
    brcc    NOCARRY
    inc     r22
NOCARRY:
    dec     r20
    brne    AGAIN
    sts     $2D2,r21    ; store sum
    sts     $2D3,r22
```

■

Lägg märke till hur användningen av kommentarer (efter ";") ökar läsbarheten för programmet väsentligt. Övning: Prova att göra om uppgiften men räkna från $0 \rightarrow 255$ istället för att ladda räknaren med 255 från början. Vilken fördel eller nackdel medför detta?

Absoluta och relativa hopp I processorer finns vanligtvis två sorters ovillkorliga hoppinstruktioner: `jmp`, *jump*, och `br`, *branch*.

Skillnaden mellan dem är att i `jmp` anges hoppadressen med sin fulla längd (icke-relativt, *absolut hopp*) medan i `br` anger argumentet hur långt från *där vi står nu* vi skall hoppa, det vill säga hoppet sker *relativt* `PC`.

I och med att programräknaren, `PC`, pekar på nuvarande instruktion handlar det om att addera ett värde till `PC` eller inte. Om hoppet skall tas sker additionen, i övriga fall kommer programräknarens värde att automatiskt peka ut nästa instruktion.

I ATmega16 heter det relativa ovillkorliga hoppet `rjmp`. De villkorliga hoppen är också alltid relativa, med namn som "`brne`" till exempel.

Med absolut adress kan hopp ske över hela programminnesområdet. Relativ adress finns av två typer: `rjmp` kan hoppa $-2048 \rightarrow +2047$ bytes och `br` blott $-64 \rightarrow +63$ bytes (typiskt cirka ± 30 assemblerrader).

Exempel: Absoluta och relativa hopp

Hoppa från `$1000` till `$1200` dels med `jmp` och dels med `rjmp`!

Adress	Absolut hopp	Relativt hopp
	:	:
`$1000 A:`	`jmp B ;'jmp $1200'`	`rjmp B ;'jmp +$200'`
	:	:
	:	:
`$1200 B:`		

Både absoluta och relativa hopp kan hoppa bakåt. För relativa hopp betyder det att argumentet är ett negativt tal ty detta tal adderas till nuvarande värdet hos PC. I och med att $a + (-b) = a - b$ kommer programräknaren minskas.

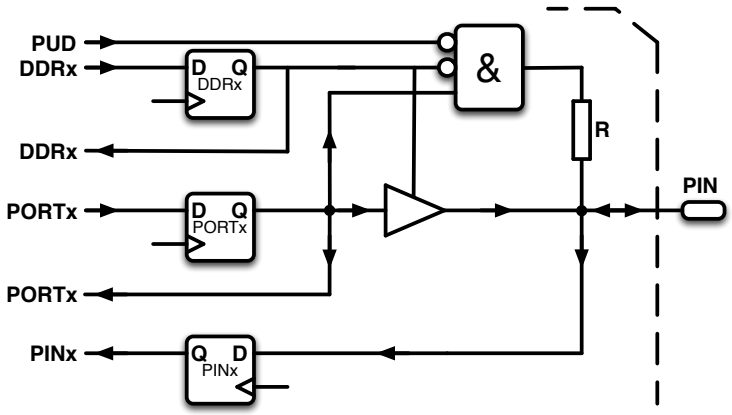
Om vi använder *labels* i programkoden behöver vi inte bry oss om vilken adress eller offset som behövs, kompilatorn sköter de detaljerna åt oss.

2.8. Grupp 5 — I/O-instruktioner

Innan vi presenterar de två I/O-instruktionerna måste vi betrakta den hårdvara de mest används med, de så kallade *portarna*.

2.8.1. I/O-portar

All kommunikation med yttervärlden sker via processorns *portar*. En port är en hårdvaruenhet i processorn som kan skicka respektive ta emot digitala signaler, se fig 2.1.



Figur 2.1.: *Schemat visar en av åtta bitar i en port. Skrivning sker till* PORT*x och läsning från* PIN*x. Biten konfigureras som in- eller utgång med datariktningsregistret* DDR*x*. Triangelsymbolen är tri-state-bufferten.

I vår processor är portarna 8 bitar breda, det vill säga man kan skriva eller läsa en hel byte direkt. Alla bitar i en port är dock individuellt styr- och läsbara så vi har mycket stora friheter att anpassa processorn till den omgivande hårdvaran.

För varje port finns tre I/O-register att ta hänsyn till: DDRx, PORTx och PINx där x ersätts med A, B eller D beroende på vilken port som används.

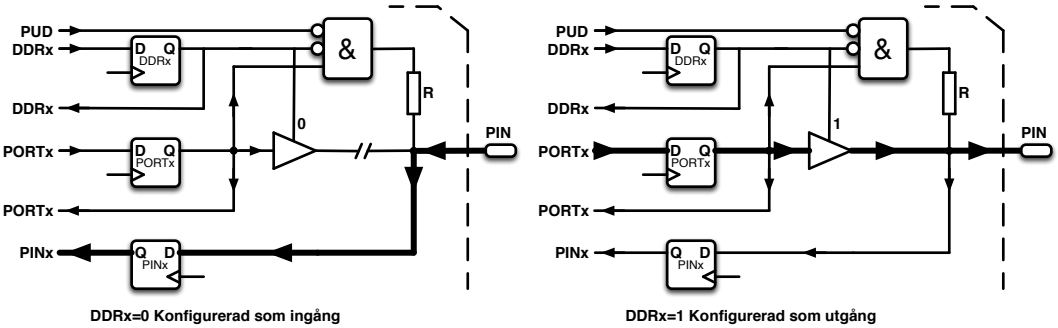
Ur figur 2.1 kan vi se D-vippan PORTx som utsignal om vi aktiverar (öppnar) tri-state-bufferten till pinnen PIN på kapselns utsida. Tri-state-bufferten isolerar D-vippan från pinnen om den är avaktiverad. Bufferten styrs av D-vippan DDRx.

Vi får två fall för varje bit beroende på om DDRx konfigurerar biten som ingång eller utgång, se också fig 2.2.

Ingång I detta fall måste DDRx för biten vara 0-satt vilket avaktiverar och kopplar bort bufferten. Pinnen har nu ingen tydlig digital nivå. Kan man inte tolerera denna otydlighet kan hela porten aktiveras att *svagt* dras mot +5 V genom motståndet R. Detta beteende styrs av signalen PUD, *Pull-Up Disable*, som återfinns i I/O-registret SFIOR. För att aktivera pull-up på en enskild bit sätts PORTx=1 för den biten.

I fallet att porten är konfigurerad som insignal (DDRx-biten=0) kan en yttre pålagd signalnivå avläsas genom att läsa PINx.

Observera att man mycket väl **kan** läsa från PORTx men det är i allmänhet inte det man vill! Man **skriver till PORT** och **läser från PIN**. Detta är ett mycket vanligt programmeringsfel!



Figur 2.2.: *Till vänster visas signalflödet vid läsning från* PIN*x* *då* DDR*x=0. Till höger visas motsvarande signalflöde vid skrivning till* PORT*x* *då* DDR*x=1. En avstängd buffert är elektrisk frånkopplad "—//—".*

Utgång I detta fall måste DDRx för biten vara 1-satt. Detta aktiverar bufferten och leder PORTx-bitens signal ut till pinnen. Pinnen kommer nu, på kapselns utsida, vara antingen digital etta eller nolla (+5 V eller 0 V i vårt fall).

Exempel: Konfigurera portar

Konfigurera PORTB så att lägre halvan är ingång och övre utgång. Läs sedan in de digitala värden som ligger på portens fyra lägre pinnar till r16 och skriv ut samma bitar på portens övre halva.

```
ldi     r16,$F0     ; $F0=11110000
out     DDRB,r16    ;     ooooiiii (o=out, i=in)
in      r16,PINB    ; read from PIN
andi    r16,$0F     ; mask bits
lsl     r16         ; shift 1 step
lsl     r16         ; shift 1 step
lsl     r16         ; shift 1 step
lsl     r16         ; shift 1 step
out     PORTB,r16   ; write to PORT
```

■

Anm 1. Använd instruktionen swap istället.

Anm 2. Observera återigen hur man **skriver till PORT** och **läser från PIN**!

Anm 3. Behövs egentligen andi r16,$0F här?[7]

■

Sammanfattningsvis kan man alltså teckna fyra kombinationer av DDRx och PORTx:

DDRx	PORTx	Betydelse
0	0	input, ingen pull-up
0	1	input, pull-up
1	0	output, 0 V
1	1	output, 5 V

[7] Svar nej! Varför?

2.8.2. `skip`-instruktionen

Med portarna sålunda beskrivna kan det vara lämpligt att avslutningsvis också beskriva hopp-instruktopnen *skip*.[8] Hoppadressen är alltid till näst-nästa instruktion om hoppet tas, annars fortsätter exekveringen med nästföljande instruktion.

I vår processor används skip-instruktionen i samband med att man testar enstaka bitar i ett register eller port (I/O-register). De fyra instruktioner det då handlar om är

`sbrc rX,b`	Hoppa om bit b i register `rX` nollställd (*cleared*)
`sbrs rX,b`	Hoppa om bit b i register `rX` ettställd (*set*)
`sbic A,b`	Hoppa om bit b i I/O-register `A` nollställd (*cleared*)
`sbis A,b`	Hoppa om bit b i I/O-register `A` ettställd (*set*)

Instruktionerna kan användas för registren `r0`–`r31` respektive I/O-registeradresserna 0–31. Notera speciellt att man inte kan nå `SREG`-bitarna med dessa I/O-adresser.

Exempel: `GET_KEY`

En tryckknapp är ansluten till processorns `PORT A`, bit 3. En nedtryckt knapp utgörs av digital etta. Använd `sbic` för att ge ett sant värde om tryckknappen är nedtryckt och ett falskt värde annars.

Det är vanligt att ett returvärde $=0$ betyder *falskt* och ett returvärde $\neq 0$ (ofta `$FF`$=-1$) betyder *sant*.

```
     ;
     ; --- GET_KEY. key != 0 if key pressed
GET_KEY:
     clr     key
     sbic    PINA,3       ; skip over if not pressed
     dec     key          ; key=FF
```

■

Avslutningsvis skall nämnas en variant av `skip`-instruktionen, `cpse`, *Compare and Skip if Equal*, som jämför två register och hoppar om de är lika.

Exempel: Compare and Skip if Equal (addera 1)

Addera 1 till `r16` tills `r16` = `r17` = 200 (decimalt). För addition och subtraktion med ett (1) kan man använda instruktionerna `inc` respektive `dec`. För andra tal använder man `subi`-instruktionen, det vill säga att man utför addition som subtraktion med omvänt tecken!

```
    ldi     r17,200
    clr     r16
LOOP:
    subi    r16,-1      ; r16 = r16 + 1
    cpse    r16,r17
    rjmp    LOOP
      :
      :
```

■

[8] Ett skip är enkelt att implementera hårdvarumässigt.

3. Strukturerad programmering

En uppgift måste alltid delas upp i mindre, mer hanterbara, deluppgifter. En deluppgift är så stor att man lätt kan se att den har en lösning, även om man inte just vid uppdelningen *exakt* vet *hur* den ska kunna lösas. Den stora fördelen med deluppgifter är att man inte ska behöva bry sig om exakt hur de ska lösas, det tar man tag i senare.

I högnivåspråk utför vi deluppgifterna i *funktioner* och *procedurer*. Det finns en teoretisk uppdelning mellan funktioner och procedurer, funktioner returnerar något medan procedurer inte gör det. Båda kan dock ha indata i form av argument.

Vi kan dela upp en komplex hårdvara i mindre delar med var för sig enklare funktion och vi kan dela upp ett större program i deluppgifter typiskt funktioner i C eller subrutiner i assembler.

I assemblerprogrammering använder vi procedurtypen *subrutin* för dessa deluppgifter. Subrutiner i sig har inget stöd varken för argument eller funktionsresultat. Om vi vill returnera ett värde kan vi till exempel göra det i ett register.

När väl programstrukturen är klar på en första högre nivå kan vi fortsätta att dela upp subrutinerna i mindre subrutiner tills varje deluppgift verkar hanterbar.

Nu handlar det förstås inte om att allt blir bättre bara genom att dela upp allt i mindre delar. Det gäller att dela upp i begripliga, logiska delar med tydlig och entydig deluppgift.

En deluppgift ska inte delas upp bara för sakens skull. När dess uppgift är entydig och har enkel kommunikation med omgivningen kan den vidare uppdelningen upphöra.

Exempel Ett dataprogram kan nästan alltid delas upp i tre delar: inmatning, databehandling och utmatning. Var och en av dessa delar kan sedan delas upp i lagom stora deluppgifter efter erfarenhet, tycke och smak. Olika programmerare delar upp en uppgift på olika sätt.

Erfarna programmerare kan göra större uppgifter direkt, medan mindre erfarna programmerare gör sig själv en otjänst om de inte delar upp uppgiften.

Det är bättre att dela upp för mycket än för lite innan man är säker på hur bra programmerare man är. Det finns studier som visar att man inte kan ha överblick över mer än 5–7 rader kod samtidigt, vilket man skulle kunna tolka som att en funktion eller subrutin inte bör vara längre än så.

Exempel Att subrutiner är bra kan vi förstå om vi betraktar succén med hämtpizza. Hämtpizza uppfyller nämligen alla krav på en subrutin:

- det är en tydligt avskiljbar uppgift,
- man vill ha pizzan men bryr sig egentligen inte om detaljerna i produktionen, och
- den levereras när den är klar:

```
:
if ( isHungry )
     beställ_hämtpizza();
while( isHungry )
     ät_hämtpizza();
:
```

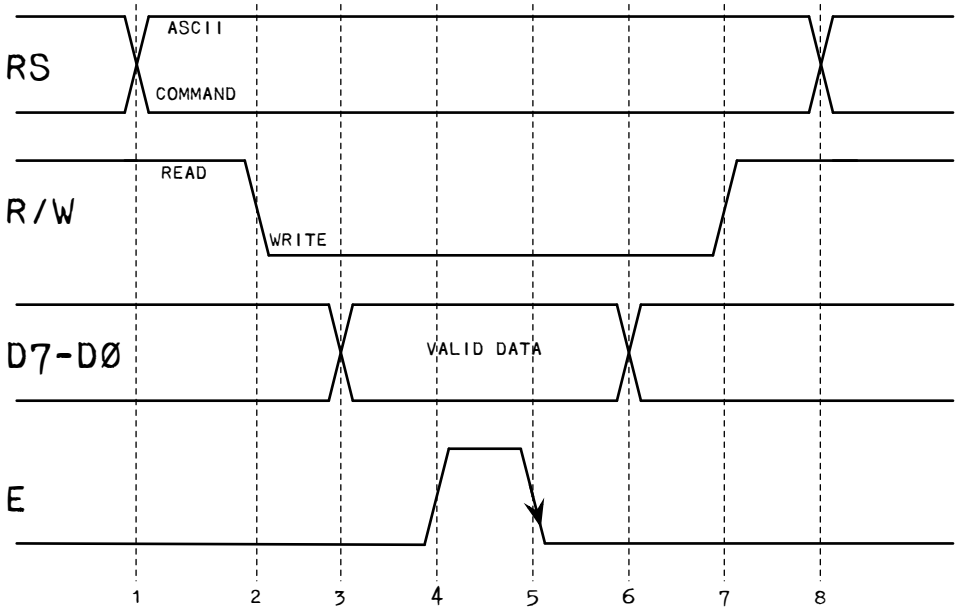
Vid programutveckling underlättar det väsentligt att ha en klar bild för sig av vad man egentligen vill göra! Det låter som en självklarhet — och är det naturligtvis också — men många tycker att programmering är något man gör vid tangentbordet.

Faktum är att programmering är något man gör när man hittat 1) lämpliga datastrukturer och 2) en metod att lösa problemet givet denna datastruktur.

3.1. Exempel på kodförfining: Att programmera en LCD

Till en LCD-display kan man skriva ut åtta-bitars ASCII-tecken på bitarna D7–D0. Förutom ASCII-tecken kan displayen också ta emot kommandon för att rensa displayen, välja markörtyp och liknande. Signalen RS avgör om skrivningen rör ASCII-tecken eller kommandon.

För varje åtta-bitarsvärde måste en komplett skrivsekvens enligt pulsschemat nedan genomföras:



En komplett skrivning innebär att signalerna RS (*register select*), R/W (*read/write*), E (*enable*) och D7–D0 utförs i ordningen ovan. I pulsschemat ser man att någon signal förändras vid tidpunkterna 1 till och med 8.

Med dessa tidpunkter kan skrivningen beskrivas som sekvensen (där RS=0 och RW=1 antages vara normal- och viloläget):

```
   Hitta ASCII-datat som ska skrivas
 1 Sätt RS=1
 2 Sätt RW=0
 3 Lägg ut datat
 4 Sätt E=1
 5 Sätt E=0
 6 Ta bort datat
 7 Sätt RW=1
 8 Sätt RS=0
```

Det blir således sammanlagt nio programrader för varje tecken som skall skrivas. För att till exempel skriva ut ordet `GLASS` på displayen blir sekvensen:

```
Data= 'G'              ; G
Sätt RS=1
Sätt RW=0
Lägg ut datat
Sätt E=1
Sätt E=0
Ta bort datat
Sätt RW=1
Sätt RS=0
Data= 'L'              ; L
Sätt RS=1
Sätt RW=0
Lägg ut datat
Sätt E=1
Sätt E=0
Ta bort datat
Sätt RW=1
Sätt RS=0
Data= 'A'              ; A
Sätt RS=1
Sätt RW=0
Lägg ut datat
Sätt E=1
Sätt E=0
Ta bort datat
Sätt RW=1
Sätt RS=0
Data= 'S'              ; S
Sätt RS=1
Sätt RW=0
Lägg ut datat
Sätt E=1
Sätt E=0
Ta bort datat
Sätt RW=1
Sätt RS=0
Data= 'S'              ; S
Sätt RS=1
Sätt RW=0
Lägg ut datat
Sätt E=1
Sätt E=0
Ta bort datat
Sätt RW=1
Sätt RS=0
```

Här har man gjort det enkelt för sig genom s.k. *copy-paste*-programmering. Metoden har flera nackdelar, bland annat:

- Det blir lång kod. Totalt $5 \cdot 9 = 45$ programrader i exemplet ovan.
- Det blir oöverskådlig kod. En längre text skulle bli flera sidor lång.
- Programmeringen blir lidande eftersom man tvingas tänka på flera nivåer samtidigt, så fort något skall hända måste man ner i varje detalj och gröta: ”Jag vill skriva en bokstav, men måste peta på RS, R/W och E hela tiden.”
- Det drar programminne. Något som, i varje fall när man programmerar mikrocontrollers, är en knapp resurs.
- Det blir svårmodifierad kod. Tänk om man vill förlänga tiden då E=1. Ändringen måste införas *för varje bokstav*, hela fem ställen i sekvensen ovan. Inte bra.

I det givna kodavsnittet är upprepningarna många och resultatet blir svåröverskådlig, svårändrad, svårunderhållen och på alla sätt *dålig* kod.

Faktorisera En kort blick på programmet ger att flera sekvenser återkommer. Genom att identifiera och *faktorisera*[1] ut dessa till funktionsanrop eller subrutiner får vi mycket kortare kod. Sekvensen `Sätt E=1` och `Sätt E=0` är en tydlig kandidat för faktorisering. Vi skriver alltså en rutin, **`do_E`**, för att göra just detta:

```
do_E {
     Sätt E=1
     Sätt E=0
}
```

[1] I assembler: Tänk subrutin!

Nu kan vi skriva koden som

```
Data= 'G'          ; G
Sätt RS=1
Sätt RW=0
Lägg ut datat
do_E
Ta bort datat
Sätt RW=1
Sätt RS=0
Data= 'L'          ; L
Sätt RS=1
Sätt RW=0
Lägg ut datat
do_E
Ta bort datat
Sätt RW=1
Sätt RS=0
Data= 'A'           ; S
Sätt RS=1
Sätt RW=0
Lägg ut datat
do_E
Ta bort datat
Sätt RW=1
Sätt RS=0
Data= 'S'           ; S
Sätt RS=1
Sätt RW=0
Lägg ut datat
do_E
Ta bort datat
Sätt RW=1
Sätt RS=0
Data= 'S'           ; S
Sätt RS=1
Sätt RW=0
Lägg ut datat
do_E
Ta bort datat
Sätt RW=1
Sätt RS=0
```

Fördelen är att vi skiljt ut detaljerna från helheten något. Exakt *vad* som händer i **do_E** är inte intressant för oss här, bara rätt sak händer. Det är säkert intressant *någon* gång, men just nu ska vi skriva ut ett ord och är inte överdrivet intresserade av om E går hög → låg eller låg → hög eller något annat. Bara rätt sak händer.

Notera också att den ovan föreslagna förändringen att förlänga tiden då E=1 nu bara behöver utföras på ett (1) enda ställe. Det är dessutom lätt att hitta vilket detta ställe är, uppenbarligen i **do_E**.

Med denna nya syn på koden kan vi se fler faktoriseringskandidater. En sådan är sekvensen

```
Sätt RS=1
Sätt RW=0
Lägg ut datat
do_E
Ta bort datat
Sätt RW=1
Sätt RS=0
```

som återkommer gång efter annan. Vi kallar denna sekvens för **do_LCD_write**:

```
do_LCD_write {
     Sätt RS=1
     Sätt RW = 0
     Lägg ut datat
     do_E
     Ta bort datat
     Sätt RW=1
     Sätt RS=0
}
```

Nu blir programkoden väsentligt kortare:

```
     Data = 'G'     ; G
     do_LCD_write
     Data = 'L'     ; L
     do_LCD_write
     Data = 'A'     ; A
     do_LCD_write
     Data = 'S'     ; S
     do_LCD_write
     Data = 'S'     ; S
     do_LCD_write
```

Lägg märke till hur kodens själva huvuduppgift nu också framträder mycket tydligare. Ordet *GLASS* är ju faktiskt uttydbart direkt ur koden.

Genom att på detta sätt faktorisera ut undernivåer underlättas läsligheten väsentligt. Med frasen *Är det värt att göra så är det värt att göra det i en subrutin* menar man att urskiljbara ”uppdrag” i en kod gör sig bäst i funktioner/procedurer/metoder.

En framtida vinst kan också visa sig då det oftare är lättare och renare att lägga till ytterligare funktionalitet i en subrutin än att göra det i huvudprogrammet. Det är till exempel inte svårt att hitta var man ska lägga till det utökade kravet ”Vid varje skrivning ska en lysdiod blinka till” om koden är faktoriserad.

För ökad läsbarhet kan man driva detta faktoriskeringstänk ett steg till. Vi vet nu att det egentligen finns två sorters skrivningar: ”ASCII” och ”kommandon”, och det enda som skiljer dem åt är signalen RS. Tag alltså bort den inledande `Sätt RS` i **`do_LCD_write`** och tillverka två ändamålsenliga funktioner med tydliga och bättre namn, `ASCII_write` och `COMMAND_write`:

```
do_LCD_write {
     Sätt RW = 0
     Lägg ut datat
     do_E
     Ta bort datat
     Sätt RW=1
     Sätt RS=0
}
```

```
ASCII_write {
    Sätt RS = 1
    do_lcd_write
}
```

```
COMMAND_write {
    Sätt RS = 0
    do_lcd_write
}
```

Med denna ytterligare faktorisering ser vi ännu mindre av detaljerna, behöver aldrig bry oss om RS-signalen mer och vi får två funktioner med entydig namngivning.

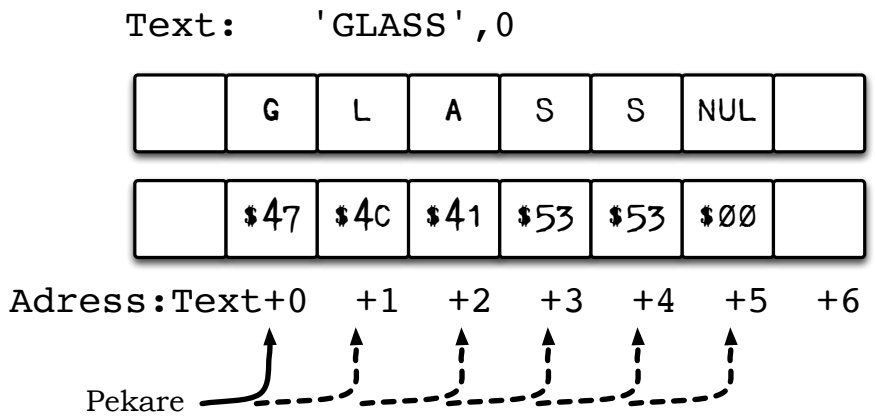
Parametrisering Önskade vi bara skriva ut några få enstaka ord skulle vi kunna vara nöjda nu, men ofta vill man skriva ut mer text än så. Ett menysystem brukar till exempel innehålla ganska många texter och instruktioner, varför även ovan beskrivna sätt att koda visar sig vara onödigt svårhanterligt, även om den är väl faktoriserad.

Det som saknas är *parametrisering*: att förse funktionen eller proceduren med *argument*.

Hela rutinen hittills är ett enda upprepande av ”hämta data och skriv ut den”. Om vi anger data separat i minnet i form av en *sträng* kan vi låta programmet indexera över den strängen tills allt är skrivet.

Med en sträng menas en sammanhängande följd av tecken placerade i minnet med början på en adress och avslutad med ett NUL-tecken, `$00`. Startadressen anges oftast

med en *label*. Nedan visas labeln `Text:` som innehåller strängen och hur tecknen placeras ut i minnet. Därurunder visas det faktiska ASCII-kodade innehållet i respektive minnesscell.



En pekare kan nu användas för att successivt traversera strängen och adressera tecken för tecken:

```
    Text: 'GLASS',0
LCD_ascii_print {
    Peka ut första tecknet
    Hämta tecknet
    Så länge tecken skiljt från NUL {
       ASCII_write
       Peka ut nästa tecken
       Hämta tecknet
    }
}
```

Vi noterar nu att vi gjort oss helt fria från *texten* som skall skrivas, den anges på ett annat ställe. Detta betyder också att rutinen nu kan användas för *alla sorts texter*, inte bara *GLASS*-texten. Vi har vunnit generalitet! Generalitet är bra!

Vi har också vunnit, men kanske inte lika uppenbart, att rutinen kommer användas oftare och därmed också debuggas mer. Ju mer vi använder rutinen i olika sammanhang desto mer och mer säkra blir vi på att den gör rätt oavsett invärden. Denna inbyggda kvalitetstestning av rutinen gör hela programmet mer pålitligt.

Exempel Av någon anledning behöver man vid några tillfällen skriva ut två plustecken efter varandra, ”++”. För att åstadkomma detta kan man förstås skriva rutinen **`Print_twoplus`**:

```
Print_twoplus {
     Data='+'
     ASCII_write
     Data='+'
     ASCII_write
}
```

Det kanske duger — om man säkert inte kommer behöva skriva exempelvis *tre* tecken någon senare gång!

En annan lösning är att använda den färdiga rutinen **`LCD_ascii_print`** ovan med tillhörande strängar ”++” respektive ”+++”. Men hur är det om vi behöver skriva sju sådana tecken? Eller fem eller något annat antal? Det blir lätt en hel radda strängar. Klart opraktiskt.

Är det då inte bättre att skriva rutinen **`Print_plusses`** med ett argument som talar om hur många tecken som ska skrivas ut:

```
Print_plusses (N) {
     För alla N gör {
          Data = '+'
          ASCII_write
     }
}
```

eller till och med

```
Print_plusses (N) {
     Data = '+'
     För alla N gör {
          ASCII_write
     }
}
```

(varför är denna senare snabbare att exekvera?) eller med ytterligare parametrisering, för att tillåta även andra tecken än plustecknet:

```
Print_chars (N,D) {
     Data=D
     För alla N gör {
          ASCII_write
     }
}
```

Den parametriserade varianten kan användas till alla utskriftsönskemål men behöver bara skrivas en (1) gång. Det är en fördel.

3.2. JSP, Jackson Structured Programming

Hittills har vi gått igenom ett kodexempel där vi med lite eftertanke skrev om koden så den blev både kortare, överskådligare och enklare att modifiera om behovet skulle uppstå. Visst vore det bra om vi kunde komma fram till detta slutresultat direkt, utan att behöva gå omvägen att skriva om koden i flera steg? Det finns sådana metoder. Här beskrivs en av dem, JSP.

JSP-metoden för strukturerad programmering utvecklades och populariserades av engelsmannen Michael A. Jackson under 1960- och 1970-talen. I korthet handlar det om att **hitta en *datastruktur* för problemet och sedan konstruera sitt program så att det följer den givna datastrukturen samtidigt som det löser problemet.**

Att programmera tvärs emot datastrukturen blir snart ett elände. Varför simma uppströms och göra det krångligare för sig? Det tar säkert mer tid att lösa uppgiften och man försvårar för sig själv också.

Här ska vi presentera en ”light”-version av JSP med betoning på vikten av en bra programstruktur och hur man kan tänka då man ska lösa en programmeringsuppgift.

Notera att metoden innebär att en större del av programmeringsjobbet utförs redan *innan* man bestämt programspråk, eller ens närmat sig tangentbordet! Detta kan kännas improduktivt[2] men är en förutsättning för att senare kunna koda lösningen.

[2]För visst är det så att man inte känner att något *egentligen* händer innan fingrarna är på tangentbordet och kod börjar skrivas?

JSP-metoden baserar sig på två huvudtankar:

1. Varje program består av tre olika sorters element, nämligen
   - *Sekvens*. Sekvens innebär att saker händer efter varann, tänk ”först händer A, sedan B och sist C”. Programrader som ligger efter varann är exempel på detta.

   - *Iteration*. En iteration beskriver något som sker upprepade gånger eller består av flera saker av samma sort. En iteration *kan* utföras *noll* gånger och motsvaras närmast av `while`-satsen för detta. Asterisken i rutan anger att det är en iteration.

   - *Selektion*. Begreppet står för ett val, en förgrening av programflödet. I ett program betyder selektion ett hoppvillkor (ofta `if`- eller `case`-sats). God programmeringssed i JSP säger att det också skall finnas ett villkorslöst alternativ vid en selektion. Denna är tänkt att fånga upp de fall som villkoren inte täcker upp. Ringen i rutan anger att det är en selektion.

2. Programstrukturen skall anpassas till den datastruktur som problemet innehåller. Ett första steg är alltså att identifiera den datastrukturen. Att finna datastrukturen är samtidigt bland de svåraste momenten. Har man, å andra sidan, funnit en lämplig datastruktur blir programmeringen lättare.

   Det är fullt rimligt att behöva prova flera möjliga datastrukturer innan man finner den som ger enklast program. Tveka inte att experimentera med olika datastrakturer.

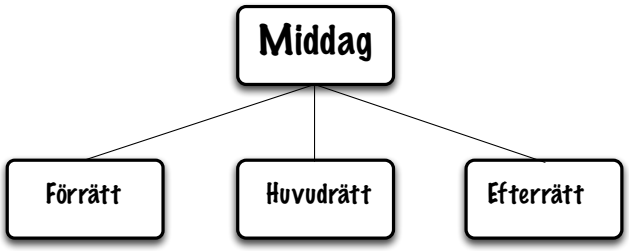
Med en datastruktur som beskriver problemets in- och utdata med de grundläggande elementen *sekvens*, *iteration* och *selektion* kan man sedan beskriva alla förekommande programmeringsuppgifter.

Vi tar några exempel på begreppen *sekvens*, *iteration* och *selektion*.

Exempel: Sekvens

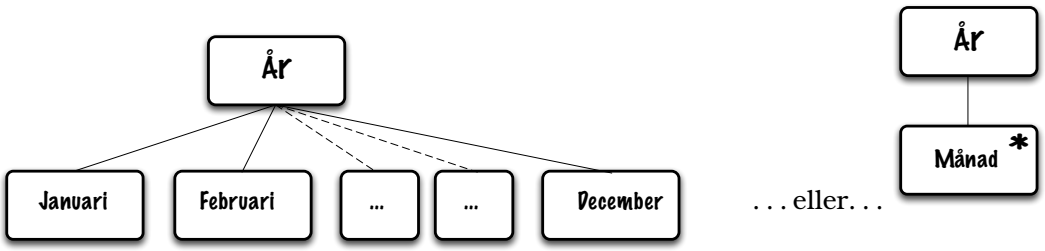
Sekvenser är händelser som sker i en bestämd följd efter varann.

- Datastrukturen över dagens tre huvudmåltider kan beskrivas som en *sekvens* av `Frukost`, `Lunch`, `Middag`.
- En `Middag` kan i sig beskrivas som en *sekvens* av `Förrätt`, `Huvudrätt` och `Efterrätt`.



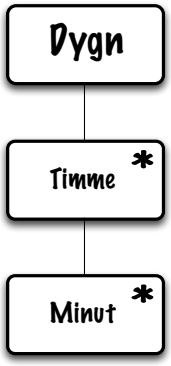
Denna beskrivning betonar *ordningen* mellan de olika delarna: `Förrätt` inleder, därefter `Huvudrätt`, och sist `Efterrätt`. Enligt detta skrivsätt finns inga middagar som utelämnar `Förrätt` eller några andra delar. Vill man kunna utesluta förrätten måste det finnas ett val att göra det, se selektion nedan.

- Ett år består av månader:

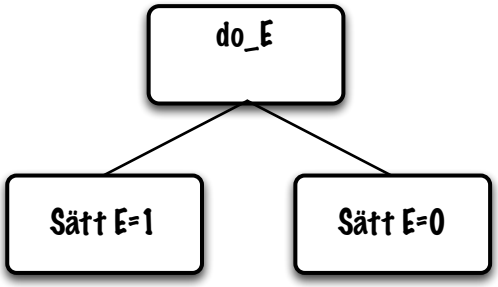


Den första beskrivningen betonar *sekvensen*, den ger ju till och med *ordningen* (januari, februari osv). Det andra beskrivningen anger bara att året består av ett antal månader, en *iteration* av månader. Den senare är användbar för att beräkna årssumma av en hyra, ty då behöver man veta att den består av månatliga räkningar, det är inte nödvändigt att känna till månadernas namn för att lösa uppgiften. ”För alla månadsräkningar, summera!”.

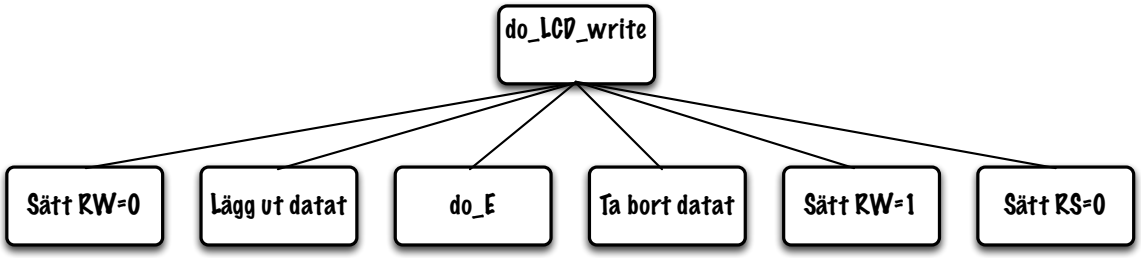
- Ett dygn består av (en iteration av) timmar som i sin tur består av (en iteration av) minuter.



- Den tidigare beskrivna proceduren **`do_E`** (sid. 29) kan skrivas



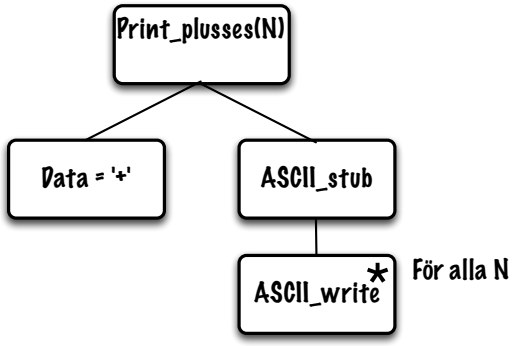
- På liknande sätt kan vi skriva den sista versionen av **`do_LCD_write`**



Exempel: Iteration

Upprepningar är iterationer, en upprepning kan ske noll gånger.

- Ett år kan ses som en *iteration* av (generella) månader.
- En promenad kan ses som en *iteration* av steg.
- Ett telefonnummer kan ses som en *iteration* av siffror.
- Vår tidigare **`Print_plusses`** blir, med **`ASCII_write`**, en *iteration* för alla N:

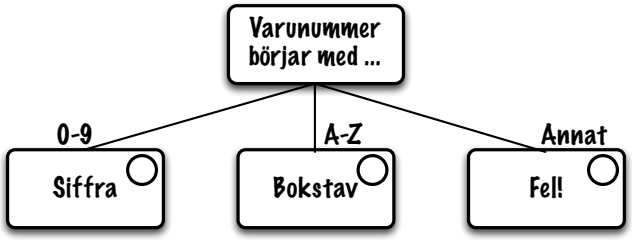


Här kan man också se hur *iterationen och iterationsvillkoret ses som den itererade komponenten*. Med detta tankesätt skjuter man iterationen framför sig när man har fullt upp med att designa sekvensen. Ingen komponent kan vara två saker samtidigt, det vill säga den kan inte vara *både* sekvens och iteration.

Exempel: Selektion

Varje val-situation är en selektion.

- En vecka är en iteration av dagar men dagarna kan vara arbetsdagar eller lediga dagar. Här kan alltså dagarna vara av två typer dvs *selektion*.
- ASCII-tecken kan *antingen* vara bokstäver eller skiljetecken eller ”icke-skrivbara”.
- Ett varunummer kan börja med *antingen* en siffra eller en bokstav, inte båda. Och om det inte är någon av det vi förväntar oss måste programmet signalera `Fel!`



Lägg märke till uttrycken *kan ses som* och liknande på några ställen ovan. Ett år skulle kunna ses som en iteration av dagar också, beroende på vad man har för avsikt att göra med informationen i sitt program.

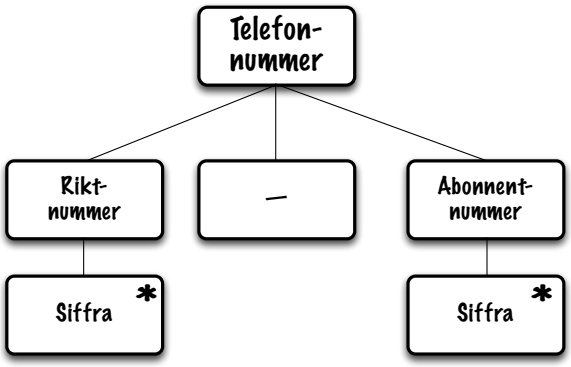
På samma sätt som det är förnuftigt att dela upp sitt program eller projekt i små delar[3] är det förnuftigt att dela upp sin datastruktur i delar *som är av betydelse för uppgiften*.

Om vi ska skriva en lista över årets månader är det uppenbart ointressant att låta månadens dagar ingå i datastrukturen, eller dagarnas namnsdagar eller annan helt överflödig information, även om den finns tillgänglig.

3.3. Ytterligare exempel

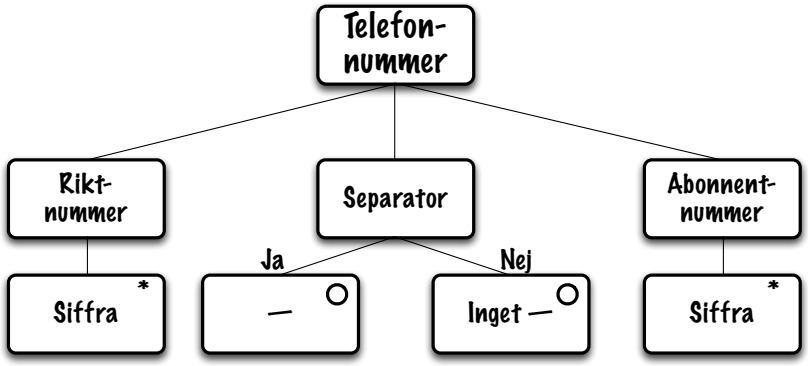
När vi nu studerat strukturdiagrammets komponenter kommer här några fler exempel och avslutningsvis övningar.

Exempel Ett telefonnummer består av ett riktnummer och ett abonnentnummer med ett bindestreck mellan delarna: 013-281200, 0142-85000 osv. Datastrukturen blir för ett sådant nummer blir



Skrivsättet *tvingar* numret att ha ett bindestreck som separator mellan riktnummer och abonnentnummer. Strängt taget passar denna beskrivning för alla nummer med ett bindestreck någonstans i sig. Men det får bara vara ett enda bindestreck. Datastrukturen duger som vi förstår också för *alla* sifferföljder som består av siffror med ett enda bindestreck inom sig men tillåter även nummer som börjar eller slutar med ett bindestreck.[4]

Om telefonnumret däremot inte nödvändigt behöver bindestrecket blir datastrukturen:



Här har man infört möjligheten att separatorn (”—”) mellan riktnummer och abonnentnummer finns eller inte finns. Båda möjligheterna är lika godkända.

[3] men inte *för* små delar!
[4] Iterationen `siffra` kan ju utföras noll gånger.

Vi ser att samma data kan ge olika datastrukturer beroende på vilken uppgift som programmet skall lösa. Häri ligger själva grunden till en bra programstruktur.

Med felaktig datastruktur — eller datastruktur som är motstridig mot uppgiften — kan lätt programmeringen bli en samling ”räddande av specialsituationer”. Är det något man vill undvika är det hantering av speciella fall. Går de att inlemma i en övergripande struktur blir programmeringen enklare.

För att konstruera en bra och användbar datastruktur är en förståelse för uppgiften och dess förutsättningar helt avgörande. Lägg därför ner tid på att hitta en lämplig datastruktur som är så förenklad som möjligt.

Med lite träning hittar man datastrukturer i allt möjligt i omgivningen. Träna på att göra sådana beskrivningar av saker omkring dig!

Övning Hur ser en labsal ut enligt JSP? På en översiktlig nivå kanske som en iteration av labplatser? Men är det viktigt för uppgiften är den ju faktiskt delad i en höger och en vänstersida. Det finns två stolar vid varje labplats, var ska dessa in i JSP-grafen? Vad om det finns tre stolar (ibland)?

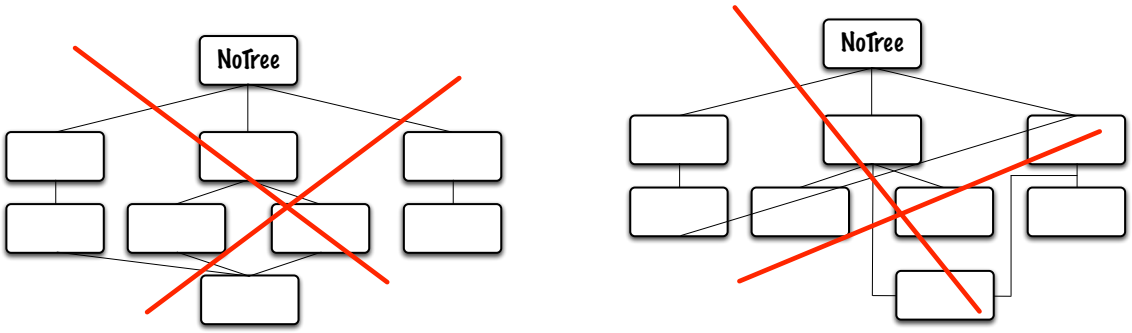
Övning En bok beskrivs enligt JSP som en iteration av sidor. Men det finns ju pärm fram och bak också. Alltså sekvensen ”frampärm, iteration av sidor, bakpärm”. En sida beskrivs som ”marginal, text, marginal”. Var skall sidnumret längst ner till höger fästas i JSP-strukturdiagrammet? Texten består av ord som i sig består av tecken. Tecken är bokstäver och siffror. Bokstäver är versaler och gemener. Versaler är A–Ä, gemener a–ö osv.

Detaljeringsgraden kan ökas *om det behövs för uppgiften*. Skall man skriva ett program som räknar antal böcker är kunskapen om pärmarna ointressant och skall inte ingå i datastrukturen.

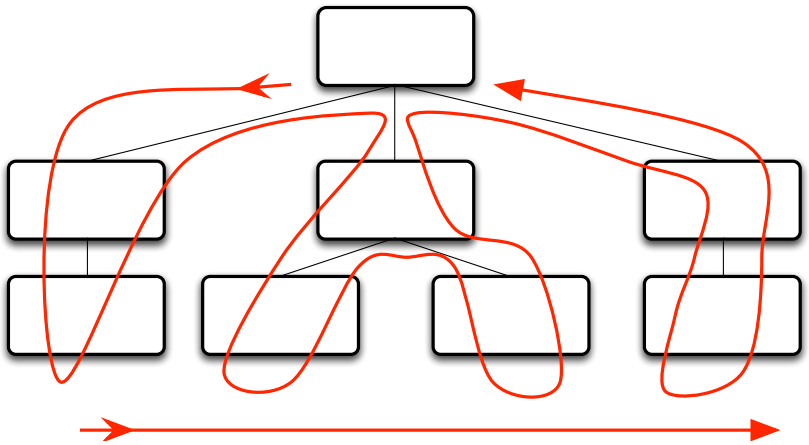
Övning Beskriv din dag från morgon till kväll enligt JSP. Försök öka detaljeringsnivån till finare och finare detaljer. Vilken detaljnivå behövs för att beräkna nedlagd tid på olika aktiviteter?

3.3.1. Strukturdiagram är träd

Ett träd slutar i noder som är löv. Precis som vid vanliga träd går inte löv ihop med varann. Således är dessa figurer *inte* korrekta strukturdiagram:



I strukturdiagram kan löv direkt översättas till procedurer/funktioner eller subrutiner som ju *alltid* återkommer till raden efter sin anropspunkt. En uppdelning av programmet i ”bra” subrutiner (eller procedurer/funktioner) kan avsevärt förenkla programmet och underlättar felsökning.



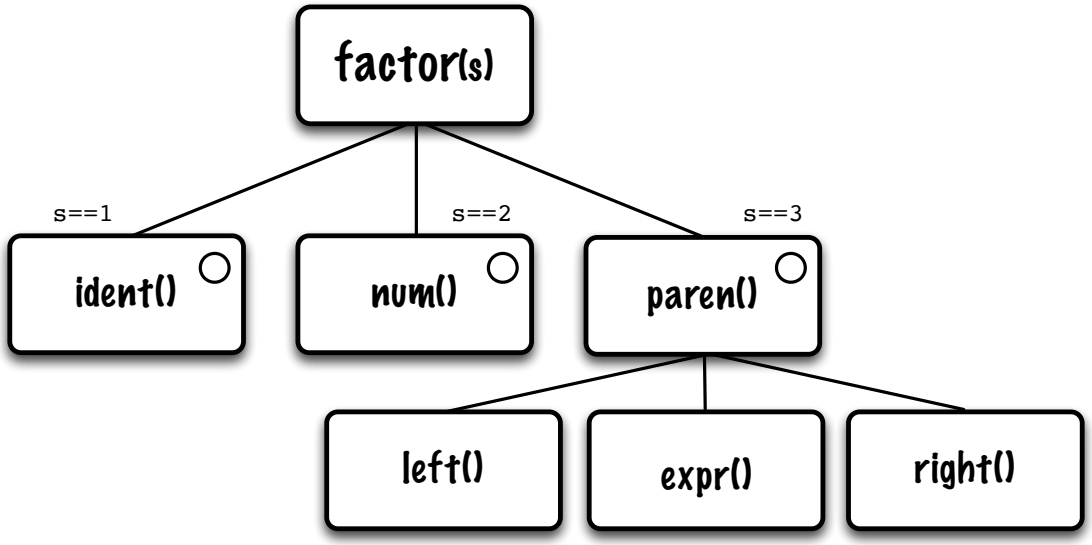
Programflödet går från vänster till höger enligt pilarna och om varje subrutin avslutas med ett återhopp kommer alla element i diagrammet besökas som i figuren ovan.

3.4. Skriva kod

När hela konstruktionen är färdigtänkt och formulerad i JSP-notation återstår att skriva den motsvarande koden i det önskade programmeringsspråket. Med korrekt förarbete är den slutliga översättningen till kod en relativt enkel affär. Det räcker som bekant att kunna skriva kod för de tre elementen *sekvens*, *selektion* och *iteration*!

Selektion och sekvens

Vi ska använda syntaxdiagrammet nedan som exempel för att visa hur *selektion* och *sekvens* kan översättas till ett C-liknande språk och slutligen ända ner till AVR-/assemblerkod.



Vi ser att rutinen `factor(s)` är en selektion av antingen en `ident()`, `number()` eller `paren()` och valet bestäms av värdet på `s`. Valet `paren()` är i sin tur en sekvens av rutinerna `left()`, `expr()` och `right()`

I ett högnivåspråk kan vi översätta detta diagram till kod på några olika sätt. En möjlighet är nästlade `if`-satser, eller sekvensiella om man tycker att nästlade `if`-satser blir för svårlästa eller oöverskådliga om nästlingsdjupet blir stort, en annan möjlighet är en `switch/case`-sats.

```
void factor(s){
   if (s == 1)
      ident();
   else if (s == 2)
            num();
         else if (s == 3){
                  left();
                  expr();
                  right();
               }
}
```

```
void factor(s){
   if (s == 1) ident();
   if (s == 2) num();
   if (s == 3){
      left;
      expr;
      right;
   }
}
```

Det högra alternativet ger enklare kod men ställer krav på att villkorsuttrycket inte får matcha flera `if`-satser.

I vissa språk finns `switch`-satsen som ger ett ännu mer lättläst intryck (lägg märke till användningen av `default` för att fånga upp oväntade värden på `s`):

```
void factor(s){
   switch(s) {
      case 1:
         ident();
         break;
      case 2:
         num();
         break;
      case 3:
         left();
         expr();
         right();
         break;
      default:
         error();
   }
}
```

Utan `switch`-sats, men med mer manuellt arbete, kan exakt samma funktion erhållas med omsorgsfullt använda `if` och `goto`:

```
void factor(s){
   if (s != 1) goto fac_1;
   ident();
   goto fac_4;
fac_1:
   if (s != 2) goto fac_2;
   num();
   goto fac_4;
fac_2:
   if (s != 3) goto fac_3;
   lparen();
   expr();
   rparen();
   goto fac_4;
fac_3:
   error();
fac_4:
}
```

Detta skrivsätt är direkt översättningsbart till AVR-assembler där vi för enkelhets skull antar att `r16` innehåller argumentet `s`:[5]

```
FACTOR:
    cpi    r16,1
    brne   FAC_1
    call   IDENT
    jmp    FAC_4
FAC_1:
    cpi    r16,2
    brne   FAC_2
    call   NUM
    jmp    FAC_4
FAC_2:
    cpi    r16,3
    brne   FAC_3
    call   LEFT
    call   EXPR
    call   RIGHT
    jmp    FAC_4
FAC_3:
    call   ERROR
FAC_4:
    ret
```

I koden ovan har sekvensen `paren()` direkt ersatts av sitt innehåll.

I vissa språk finns `switch`-satsen som ger ett ännu mer lättläst intryck (lägg märke till användningen av `default` för att fånga upp oväntade värden på `s`):

```
void factor(s){
   switch(s) {
      case 1:
         ident();
         break;
      case 2:
         num();
         break;
      case 3:
         paren();
         break;
      default:   error();
   }
}
```

Utan `switch`-sats, men med mer manuellt arbete, kan exakt samma funktion erhållas med omsorgsfullt använda `if` och `goto`:

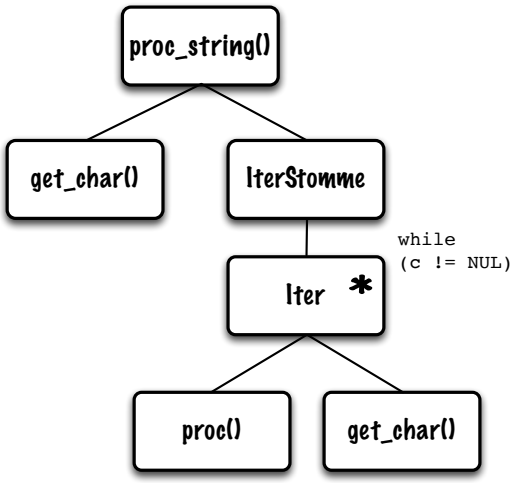
[5] Detta assemblerspråk skiljer inte på versaler och gemener så för att inte konstanten `IDENT` skall kollidera med rutinen `ident` markeras konstanten med inledande `c`.

```
void factor(s){
   if (s != 1) goto fac_1;
   ident();
   goto fac_4;
fac_1:
   if (s != 2) goto fac_2;
   num();
   goto fac_4;
fac_2:
   if (s != 3) goto fac_3;
   paren();
   goto fac4;
fac_3:
   error();
fac_4:
}
```

Iteration

I JSP motsvaras komponenten *iteration* av en `while`-sats. I en `while`-sats utförs det som följer så länge ett villkor är sant men kan också utföras *noll* gånger om villkoret inte skulle vara logiskt sant. Som exempel på iteration tar vi rutinen `proc_string()` som utför en viss databehandling på en sträng.

`get_char()` hämtar nästa tecken `c`, som är `NUL` då strängen är slut. Denna typ av struktur, med `get_char()` både innan iterationen och som sista del av iterationen, är vanlig om man vill traversera en sträng. Den första `get_char()` avgör om iterationen överhuvudtaget skall utföras och den sista ser till att hela strängen behandlas.



I ett högnivåspråk kan diagrammet översättas till

```
void AC(void){
     get_char();
     while (c != NUL){
          iter();
     }
}
```

eller mer direkt:

```
void AC(void){
     get_char();
     while (c != NUL){
          proc();
          get_char();
     }
}
```

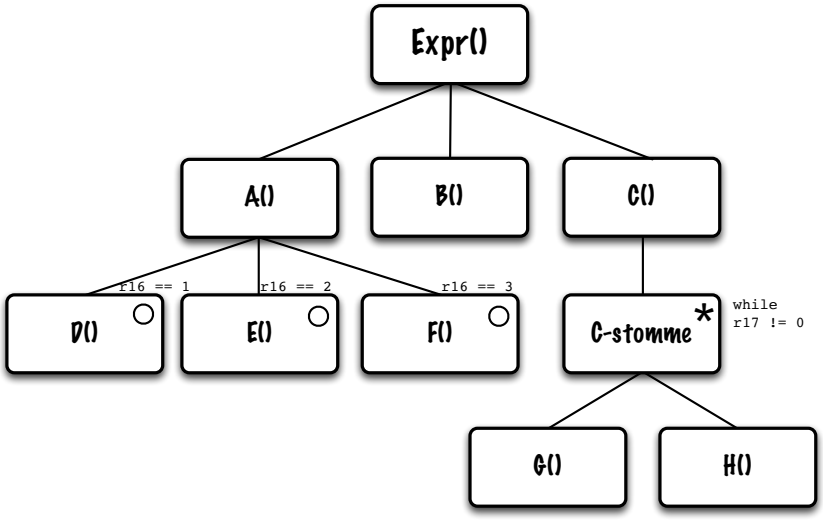
Och om vi går ner på AVR-assemblernivå ser vi att `while`-satsen motsvaras av kombon `cp`[6]`/breq` och ett avslutande `jmp`.

```
proc_string:
    call   get_char
proc_again:
    cpi    r16,NUL
    breq   proc_done
    call   B
    call   get_char
    jmp    proc_again
proc_done:
    ret
```

Vi har nu sett hur JSP-strukturerna *sekvens*, *selektion* och *iteration* kan skrivas på några olika sätt ända ner till assemblernivå. Slutsatsen är att själva kodningen blir närmast ett standardförfarande om bara strukturdiagrammen är korrekta. Vi har samtidigt erhållit fördelen att koden är strukturerad, enklare att modifiera och troligen lättare att följa.

Exempel

Översätt följande diagram till assembler



Vi noterar först att det är korrekt enligt JSP då det inte blandar komponenter av olika typer på samma nivå. För att åstadkomma detta har man behövt införa `C-stomme` som iteration.

[6] `cp` jämför register med register, `cpi` jämför register med en konstant.

För att vara komplett görs övergången till assembler via ”högnivåspråk”:

```
void expr(){
    switch(r16): {
        case 1:
            D();
            break;
        case 2:
            E();
            break;
        case 3:
            F();
            break;
    }
    B();
    while (r17 != 0){
        G();
        H();
    }
}
```

```
EXPR:
    cpi     r16,1
    brne    EXPR_NOT_1
    call    D
    jmp     EXPR_SEL_END
EXPR_NOT_1:
    cpi     r16,2
    brne    EXPR_NOT_2
    call    E
    jmp     EXPR_SEL_END
EXPR_NOT_2:
    cpi     r16,3
    brne    EXPR_NOT_3
    call    F
    jmp     EXPR_SEL_END
EXPR_NOT_3:
EXPR_SEL_END:
    call    B
EXPR_WHILE:
    cpi     r17,0
    breq    EXPR_END
    call    G
    call    H
    jmp     EXPR_WHILE
EXPR_END:
    ret
```

Med lite eftertanke kan avslutningsvis den sista `jmp EXPR_SEL_END` tas bort.

■

4. Binär aritmetik

Från digitaltekniken har vi behandlat binära tal och vet hur de är uppbyggda. I denna kurs ska vi tillämpa dessa kunskaper mer praktiskt. För att hänga med i svängarna har det visat sig att en repetition ofta är nödvändig. Även om du kan det binära talsystemet från digitaltekniken bör du läsa igenom detta kapitel så att inget hamnat mellan stolarna.

4.1. Talbaser

Innan vi går in på de egentliga talrepresentationerna måste vi definiera begreppet *talbas*. Våra vanliga hederliga tal har basen 10 och den mest framträdande egenskapen är kanske att övergången från 9 till 10 innebär att entalssiffran här börjar om från noll samtidigt som talet begåvas med en tiotalssiffra. Det är emellertid inget speciellt med basen 10. Man kan mycket väl tänka sig, exempelvis, basen 5 istället. I det senare fallet byggs alla tal upp av fem symboler — 0, 1, 2, 3 och 4 — och en uppräkning från noll sker enligt

$$0,\ 1,\ 2,\ 3,\ 4,\ 10,\ 11,\ 12,\ 13,\ 14,\ 20,\ 21,\ 22,\ 23,\ 24,\ \ldots$$

Då ”10” ovan är farligt likt det vi normalt kallar ”tio”, fastän det numeriska värdet inte alls motsvarar detta, är det vanligt att ange talbasen i petitstil vid sidan av talet, à la ”10_5”. Våra vanliga decimaltal anges på samma sätt med exempelvis ”17_{10}”. Lägg märke till att basen alltid anges i decimal form![1]

Det är lätt att hitta symboler för alla tal upp till 9, om vi skulle behöva. Vi tar bara våra gamla bekanta 0,. . . , 9. Så för talbaser mindre än 10 uppstår inga problem, vi har redan en uppsättning lämpliga symboler. Men vad händer när vi ska använda en talbas större än 10? Exempelvis talbasen 16? Symbolerna 0 till 9 återanvänder vi som vanligt, och när dom tagit slut fortsätter vi med alfabetets bokstäver, A, B, C, D, E och F, och har på så sätt erhållit 16 symboler. Att räkna med denna talbas kommer bli vardagsmat under kursen och det är väl redan nu ingen större överraskning att man räknar upp talen så här:

$$0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ \mathrm{A},\ \mathrm{B},\ \mathrm{C},\ \mathrm{D},\ \mathrm{E},\ \mathrm{F},$$
$$10,\ 11,\ 12,\ 13,\ 14,\ 15,\ 16,\ 17,\ 18,\ 19,\ 1\mathrm{A},\ 1\mathrm{B},\ 1\mathrm{C},\ \ldots$$

Att kunna hantera och översätta mellan olika talbaser är viktigt och några metoder för detta kommer att presenteras snart. Om detta och andra operationer verkar svårt kan det kanske vara lämpligt att betrakta hur motsvarande operationer utförs under talbasen 10. Alla operationer har naturligtvis sin motsvarighet i den decimala talbasen och de avmystifieras ofta om övergång till decimaltal görs.

Innan vi lämnar dessa preliminärer skall bara nämnas den mest använda talbasen i digitala sammanhang — basen 2. Vi behöver här två symboler och väljer naturligt dessa som 0 och 1.

Med bara två symboler sker uppräkning från 0 av teckenlösa tal som

$$0,\ 1,\ 10,\ 11,\ 100,\ 101,\ 110,\ 111,\ 1000,\ 1001,\ 1010,\ 1011,\ldots$$

[1] Varför då? Försök själv att ange ”10_5” helt i basen 5, det vill säga med basen 5_{10}. . .

För att inte blanda ihop ”1000” ovan med det decimala ”tusen”, är det viktigt att skriva talbasen så fort sammanhanget kan vara osäkert för läsaren, det vill säga talet bör anges som 1000_2. Märk att detta är en tjänst åt *läsaren*. Som vanligt gör man som man vill i egna anteckningar, men resultat och liknande bör förses med gällande talbas.

1000_2 uttalas dessutom aldrig ”tusen” utan ”ett-noll-noll-noll”. Tabellen nedan visar förhållandet mellan baserna 10 (decimal bas), 5 och 2.

Bas 10	bas 5	bas 2
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	10	101
6	11	110
7	12	111
8	13	1000
9	14	1001
10	20	1010
11	21	1011
:	:	:

Lägg speciellt märke till var det inrutade talet ”10” dyker upp i respektive kolumn.

4.2. Addition

Vi har ju ovan redan lärt oss addera — i varje fall med ’ett’ — vid uppräkningen av talen från noll och uppåt. Vi ska emellertid behandla addition lite för att kunna gå vidare och lära oss hantera negativa tal och andra räknesätt.

Addition är enkelt, det finns egentligen bara fyra summor att hålla reda på (talbasen är förstås 2):

0	+	0	=	0
0	+	1	=	1
1	+	0	=	1
1	+	1	=	10

Det enda märkvärdiga med addition är här den fjärde summan, där resultet blir två siffror. Resultatet kan delas upp i en-talsdelen (här 0) och två-talsdelen (här 1).[2] Två-talsdelen kallas som vanligt för minnessiffra eller vanligare i datorsammanhang, *carry* som är det engelska uttrycket.

Begreppet *ordlängd* är viktigt att förstå. Ordlängden anger det antal bitar som används i beräkningen, med undantag för eventuell genererad minnessiffra. I kursens processor är ordlängden åtta.

Exempel: Addition, kortare ordlängd

Addera de båda binära talen `01011` och `10111`.
Antag ordlängden 5_{10}.

```
  01011
+ 10111
--------
 100010
```

[2] Jämför med additionen $9 + 1 = 10$, här får vi en en-talssiffra och en tio-talssiffra.

■

Additionen sker bitvis med början i den minst signifikanta biten. Det viktiga är här att komma ihåg att `1+1+0=10` och `1+1+1=11` som alltså genererar en carry till nästa bit. Vi ser också att denna addition ger ett resultat om sex bitar medan termerna bara var fem siffror långa. Om *ordlängden* är fem bitar kallas den extra, sjätte, biten för *carry*. Om ordlängden däremot är åtta genereras ingen carry generats i detta fall:

Exempel: Addition, längre ordlängd

Addera de båda binära talen `01011` och `10111`!
Antag ordlängden 8_{10}.

```
  00001011
+ 00010111
----------
  00100010
```

Här fick hela summan plats inom åtta bitar och ingen carry genererades.

■

4.3. Talrepresentationer

(*Avsnittet kan hoppas över vid en första genomläsning.*)

Det finns två sorters tal i datorsammanhang *fixtal* och *flyttal*:

- Ett fixtal är av typen $\ldots, -2, -1, 0, 1, 2, \ldots$ det vill säga ett positivt eller negativt tal. Fixtal behöver däremot inte representera enbart heltal. Det går utmärkt att införa en decimalpunkt i talet, men då har denna decimalpunkt ett bestämt läge. Precis som kassaapparater som räknar i ören och har en fast decimaldel som omfattar de två minst signifikanta siffrorna.

  När vi räknar med fixtal behöver vi aldrig bry oss om decimalpunkten. Den finns där på ett fast läge och påverkar inte uträkningarna.

- Flyttalen skiljer sig mot fixtalen genom att de kan innehålla en decimalpunkt var som helst i talet. Praktiskt görs detta genom att talet delas upp en mantissa och en exponent. Talets värde erhålls sedan ur $mantissa \cdot 2^{exponent}$.[3]

  Med decimala tal kan exempelvis talet 42 anges som $42 \cdot 10^0 = 4.2 \cdot 10^1 = 0.42 \cdot 10^2$. Mantissan är här 0.42 och exponenten 2.

  > Addition av flyttal går till på samma sätt som addition av fixtal. För att addera två flyttal måste mantissornas respektive bitar med samma vikt stå under varann. För att åstadkomma detta måste ofta talen *avvnormaliseras* så att detta är fallet.

  #### Exempel: Addition, flyttal

  Addera de båda flyttalen 63 och 398!

  Inget av talen är normaliserade (de är inte ens skrivna på mantissa-exponentform) så vi gör det först (med decimalbas för tydligheten):

[3] Mantissan utförs med en inledande decimalpunkt: $0.0100 = \frac{1}{4}$. Ofta *normaliseras* mantissan, vilket innebär att man genomför upprepade vänsterskiftningar tills högsta biten är ett: $0.0100, 0.100, 1.000$. När man nu vet att ettan är på plats kan man använda denna bit till att ange mantissans tecken.

$63 = 63 \cdot 10^0 = 6.3 \cdot 10^1 = 0.63 \cdot 10^2$ och

$398 = 398 \cdot 10^0 = 39.8 \cdot 10^1 = 3.98 \cdot 10^2 = 0.398 \cdot 10^3$

Nu är talen normaliserade och den egentliga additionen kan inledas. Först gäller det att få siffror med samma vikt i samma position eller, vilket är samma sak, se till att talen har samma exponent. Ett av talen måste alltså *avnormaliseras*. Här väljer vi att avnormalisera det mindre talet, det vill säga

$0.63 \cdot 10^2 = 0.063 \cdot 10^3$.

Med samma exponent kan vi addera mantissorna:

$0.063 + 0.398 = 0.460$

Vi ser att summan är 0.460. Mantissan är normaliserad redan och talet kan slutligen skrivas:

$0.460 \cdot 10^3$ som alltså är summan av 63 och 398.

Ibland kan additionen av mantissorna ge ett tal som är större än ett. I sådana fall måste slutresultatet normaliseras, genom att mantissan dividerats med 10 och exponenten ökas med ett.

■

4.4. Basen 2_{10}

I talbasen 2 representeras talen av dess binära representant som enbart kan utgöras av talbasens två symboler `0` respektive `1`.

Exempel: Binär till decimal

Översätt det binära talet `10011101` till sitt decimala värde:

$$10011101_2 = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 =$$
$$= 128 + 16 + 8 + 4 + 1 = 157_{10}$$

■

Den enskilda binära siffran kallas en *bit* (av engelskans *b*inary dig*it*). En grupp av fyra sådana siffror kallas en *nibble* (ev nybble) medan en åtta-grupp kallas för *oktett* eller numera vanligare *byte*.

Det är viktigt att kunna översätta mellan olika talbaser. Speciellt baserna 2, 10 och 16 är vanliga. Förr användes även den *oktala* talbasen 8 men den är numera sällsynt.

4.5. Höger- och vänsterskift

Man ser snart att man kan multiplicera ett tal med 2 genom att skifta det binära talet ett steg åt vänster. Varje bit blir värd dubbelt så mycket efter ett skift till vänster.[4]

```
010101 1   =   43
   ←
10101 1 0  =  2·43
```

[4] Jämför med hur vi i det decimala talsystemet lätt kan multiplicera ett tal med 10 (tio) genom att skifta talet åt vänster och lägga till en nolla.

Att detta fungerar kan man se av likheten

$$\sum x_i \cdot 2^{n-i} \cdot 2 = \sum x_i \cdot 2^{n-i} \cdot 2^1 = \sum x_i \cdot 2^{(n+1)-i}.$$

På motsvarande sätt kan naturligtvis en divison med 2 utföras genom högerskiftning. För talet ovan är det trivialt att genomföra denna operation, och svaret blir rätt. Om talets minst signifikanta bit däremot är 1 kommer divisionen resultera i en precisionsförlust eftersom denna bit kommer att skiftas ut ur talet. En påföljande multiplikation med 2 kommer inte att resultera i det ursprungliga talet.

Exempel:

$101_2/2_{10} = 10_2$, det vill säga $5_{10}/2_{10} = 2_{10}$ inte 2.5_{10}.
Man ser att detta motsvarar en division med avrundning nedåt.

4.6. Omvandling från decimal till binär form och tvärtom

Med kännedom om positionsvikterna hos ett binärt tal är det lätt att översätta ett sådant tal till dess decimala motsvarighet. Omvändningen är dock inte helt lika enkel. Här skall två metoder presenteras, dels en med hjälptabell över tvåpotenser och dels en genom upprepad division med 2 och inspektion av divisionens rest.

Exempel: Decimal till binär

Översätt 98_{10} till dess binära motsvarighet!

Metod 1. Med hjälptabell

Leta reda på, och subtrahera med, närmast mindre tvåpotens. Om subtraktionen gav ett negativt resultat, notera en nolla, återställ resultatet och börja om. Om subtraktionen gav ett positivt resultat noteras en etta och resultatet används i nästa omgång. Alltså:

$$\begin{array}{rcrcrcr}
98 - 2^6 & = & 98 - 64 & = & 34 & \rightarrow & 1 \\
34 - 2^5 & = & 34 - 32 & = & 2 & \rightarrow & 1 \\
2 - 2^4 & = & 2 - 16 & = & -14 & \rightarrow & 0 \\
2 - 2^3 & = & 2 - 8 & = & -6 & \rightarrow & 0 \\
2 - 2^2 & = & 2 - 4 & = & -2 & \rightarrow & 0 \\
2 - 2^1 & = & 2 - 2 & = & 0 & \rightarrow & 1 \\
0 - 2^0 & = & 0 - 1 & = & -1 & \rightarrow & 0
\end{array}$$

det vill säga $98_{10} = 1100010_2$. Man läser högra kolumnen uppifrån. Och man ser då också varför metoden fungerar, den skiljer precis ut de jämna tvåpotenser som bygger upp talet. Just här använde vi ingen tabell men det är lätt att upprätta en tabell över tvåpotenser för att underlätta översättningen.

Metod 2. Genom upprepad division med 2

Samma tal som ovan ger följande beräkningar. Lägg märke till om rest uppstår vid divisionen.

	rest?		
98/2=49.0	nej	$\rightarrow$	0
49/2=24.5	ja	$\rightarrow$	1
24/2=12.0	nej	$\rightarrow$	0
12/2=6.0	nej	$\rightarrow$	0
6/2=3.0	nej	$\rightarrow$	0
3/2=1.5	ja	$\rightarrow$	1
1/2=0.5	ja	$\rightarrow$	1

... som ger samma svar som förra metoden, 1100010_2. Men man måste läsa nerifrån och upp i det senare fallet.

Division med talbasen två är liktydigt med högerskift av det binära talet. Det som åstadkoms med metoden ovan är alltså att successivt mata ut minst signifikant bit till höger. Positionsvikten för den utskiftade biten är $2^{-1} = 0.5$ och det är dessa halvor (rest) som vi noterar i tur och ordning.

Metoden är generell och fungerar för alla baser. Prova till exempel med decimal bas som då skall delas med tio för att producera kvot och rest.

■

4.7. Hexadecimal representation

Förutom det rent binära talsystemet förekommer även det hexadecimala talsystemet, med basen 16 ofta i fortsättningen. Det är i allmänhet inga svårigheter att omvandla ett binärt tal till dess hexadecimala motsvarighet och tvärtom. Med följande hjälptabell är det särskilt enkelt:

Dec	Bin	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

Dec	Bin	Hex
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

4.7.1. Omvandling binär till hexadecimal form

Omvandling från binär till hexadecimal form är särskilt enkel.[5]

Exempel: Binär till hex

Översätt talet 1110101111_2 till basen 16!

Vi använder tabellen ovan. Översätt varje fyr-grupp till sin hexadecimala motsvarighet och saken är klar:

$$\begin{array}{ccccc} 11 & 1010 & 1111 & = & 1110101111 \\ 3 & A & F & = & 3AF \end{array}$$

Lär dig tabellen och motsvarande binära fyrabitsgrupp utantill!

■

4.7.2. Olika skrivsätt

Förutom att skriva ett litet index som anger talbasen signaleras i allmänhet hexadecimala tal genom att det inleds med ett `$`-tecken, men även andra skrivsätt förekommer. Således är detta sant:

$$\texttt{\$62=0x62=H'62} = 62_{16}.$$

[5] Varför är det så? Varför är inte binär till basen 5 lika enkel till exempel? Vilka baser är särskilt enkla på det här sättet?

4.7.3. Omvandling hexadecimal till decimal form

Översättningen mellan hexadecimala tal och decimaltal görs direkt med kunskap om talpositionen.

Exempel: Hex till decimal

Översätt det hexadecimala talet `$3AF` till decimaltal!

$$3AF = 3_{10} \cdot 16^2 + 10_{10} \cdot 16^1 + 15_{10} \cdot 16^0 = 3 \cdot 256 + 10 \cdot 16 + 15 = 943.$$

■

4.8. 2-komplement

För teckenlösa binära tal har vi sett att talvärdet kan erhållas genom addition av tvåpotenser. Närmare bestämt tvåpotenserna $..., 2^3, 2^2, 2^1, 2^0$ det vill säga talen $..., 8, 4, 2, 1$. Ett fyra-bitars positivt tal, $\mathcal{X}$, som består av bitarna x_3, x_2, x_1, x_0 får alltså det decimala värdet:

$$\mathcal{X} = \mathbf{8} \cdot x_3 + \mathbf{4} \cdot x_2 + \mathbf{2} \cdot x_1 + \mathbf{1} \cdot x_0$$

Vi har även behov av att kunna representera negativa tal. Det finns flera möjliga representationer för dessa.[6] Vi fastnar för en som har trevliga egenskaper och dessutom enkelt kan implementeras i hårdvara, *2-komplement*.

Ett tal, $\mathcal{X} = \{x_3, x_2, x_1, x_0\}$, kan i 2-komplement skrivas:

$$\mathcal{X} = \mathbf{-8} \cdot x_3 + \mathbf{4} \cdot x_2 + \mathbf{2} \cdot x_1 + \mathbf{1} \cdot x_0$$

Vi ser att positionsvikterna 4, 2, och 1 är som vanligt men att mest signifikant bit har störst *och negativ* vikt, –8. Bitarna $\{x_2, x_1, x_0\}$ bygger upp ett positivt tal $0 \leq \mathcal{X} \leq 7$. Om $x_3 = 0$ är detta hela talets värde men om $x_3 = 1$ ”aktiveras” –8 och denna kommer att byta tecken på hela talet. Mest signifikant bit kallas därför för *teckenbit*. För att sammanfatta:

> **Ett tvåkomplementstal med mest signifikant bit *ett*-ställd är alltid ett *negativt* tal. Ett tvåkomplementstal med mest signifikant bit *noll*-ställd är alltid ett *positivt* tal.**

Exempel: Binär till decimal, tvåkomplement

> Översätt det binära talet 1011 till ett decimaltal under förutsättning att vi vet att det är ett tvåkomplementstal! Här underförstås att ordlängden är 4 så den inledande ettan är en teckenbit för talet.

Först bygger vi upp ett positivt tal med bitarna $\{x_2, x_1, x_0\}$ enligt formeln ovan och sedan applicerar vi *teckenbiten*, x_3:

Tydligen betyder $1011_2 = -5_{10}$ vilket verkar rimligt eftersom teckenbiten är satt och talet således är negativt.

■

[6] Till exempel tecken-beloppprepresentation och ett-komplement

En tabell över decimaltal och fyrabitars tvåkomplementstal kan konstrueras på samma sätt som i exemplet:

Decimal utan tecken	Decimal med tecken	Bin
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	–8	1000
9	–7	1001
10	–6	1010
11	–5	1011
12	–4	1100
13	–3	1101
14	–2	1110
15	–1	1111

Vi ser att **tvåkomplementsrepresentationen kan representera *både* positiva och negativa tal**. De negativa binärtalen utmärks av att teckenbiten är ett-ställd. De positiva kännetecknas av att teckenbiten är nollställd. Detta gäller generellt, oavsett vilken ordbredd som används.

Fördelen med tvåkomplementsrepresentationen är att vi kan genomföra additioner och subtraktioner precis som förut. Det behövs ingen speciell additionsmetod bara för att talen är tvåkomplementerade.

En direkt följd av detta är att det inte finns något som överhuvudtaget skiljer tvåkomplementstal från teckenlösa tal. Man kan inte se på dem om de är det ena eller det andra. Det är upp till betraktaren att avgöra om ett tal skall tolkas som ett tvåkomplementstal eller inte.

Exempel: Tvåkomplement fungerar

Visa att addition med tvåkomplementstal ”fungerar” och ger rätt resultat!

Vi kan göra det med tabellen ovan som hjälp.

$$(+2) + (+1) = +3, \text{ ok}$$

$$(+2) + (-1) = +1, \text{ ok}$$

$$(-2) + (-3) = -5, \text{ ok}$$

$$(-2) + (+3) = +1, \text{ ok}$$

■

I och med att vi kan addera vilka tal som helst kan vi också subtrahera dom, ty subtraktion kan ses som addition med omvänt tecken på ena termen. En kunskap vi ju faktiskt redan använde i exemplet ovan.

Tvåkomplementsrepresentationen av ett binärt tal är lätt att beräkna: **Invertera alla bitar och addera 1**, det vill säga

$$-X = \overline{X} + 1$$

Om man lever i en tvåkomplementsvärld sker alltså teckenbyte av ett tal genom att tvåkomplementera det. Teckenbytet gäller förstås både övergång från positiva till negativa tal och tvärtom.

Två viktiga anmärkningar

- Ett vanligt missförstånd är att tvåkomplementstal med nödvändighet är negativa. Det är inte sant. De *kan* vara negativa men det hänger på teckenbiten om de är det. Ett positivt tal kan tvåkomplementeras till ett negativt och ett negativt tal kan tvåkomplementeras till ett positivt.
- Ett tvåkomplementstal byter inte tecken *bara* genom att ändra teckenbiten. Det byter förstås tecken men talet blir samtidigt något annat. Man måste *tvåkomplementera* det för att beloppet ska vara detsamma.

Exempel: Decimal till binär

1) Vad blir –7 uttryckt som ett binärt (tvåkomplements) tal?

$$-7_{10} = \overline{0111} + 1 = 1000 + 1 = 1001_2.$$

2) Vad blir 1101_2 i decimal form?

Talet är negativt då teckenbiten, längst till vänster i talet, är ett-ställd. För att ta reda på vilket negativt tal det är tar vi först fram beloppet, sedan vet vi att detta belopp skall förses med ett negativt tecken.

$$-1101_2 = \overline{1101} + 1 = 0010 + 1 = 0011 = 3_{10}.$$

■

4.9. Addition, subtraktion, skift för tvåkomplementstal

Med några enkla prov kan vi konstatera att följande gäller även för tvåkomplementrepresentationen:

1. Addition fungerar som vi förväntar oss.
2. Subtraktion genomförs som addition med negativt tal, det vill säga

$$a - b = a + (-b)$$

3. Vänsterskift av alla bitar ett steg innebär multiplikation med två.
4. Högerskift av alla bitar kan resultera i samma precisionsförlust som med teckenlösa tal.

5. Högerskift av alla bitar kan vara division med två:

$$\frac{+4_{10}}{2_{10}} = \frac{0100_2}{2_{10}} = 0010_2 = 2_{10}$$

som är korrekt, men

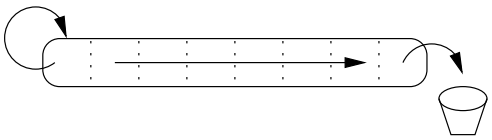
$$\frac{-4_{10}}{2_{10}} = \frac{1100_2}{2_{10}} = 0110_2 = +6$$

som är fel! Det gamla högerskiftet är tydligen bara giltigt om ursprungstalet är positivt.

Problemet löser vi genom att införa ett speciellt högerskift för tvåkomplementskodade tal, det *aritmetiska* skiftet som bevarar teckenbitens värde:

$$\frac{-4_{10}}{2_{10}} = \frac{1100_2}{2_{10}} = 1110_2 = -2.$$

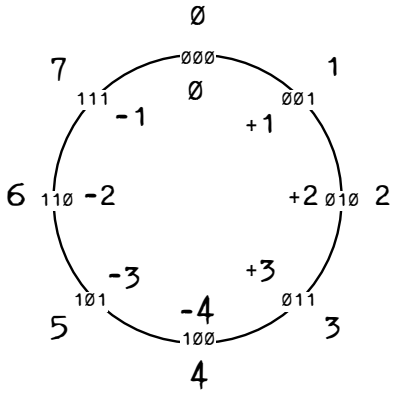
Det aritmetiska högerskiftet kan alltså[7] ritas som



Man ser att teckenbiten längst till vänster bevaras (kopieras till sig själv) medan biten längst till höger går förlorad.

4.10. Spill

Addition och subtraktion kan göras mer åskådliga med en *cirkulärgraf*. Cirkulärgrafen är i princip en tallinje med den skillnaden att den tar hänsyn till att vi rör oss med en begränsad ordlängd, se figur 3.1.



Figur 4.1.: *Cirkulärgrafen åskådliggör binära tal med ordlängden tre bitar. Utanför cirkeln är talen teckenlösa och inuti den syns de tvåkomplementskodade. Utgående från ett tal kan sker addition medurs och subtraktion moturs.*

Som på en tallinje kan addition genomföras genom att gå åt höger, vilket blir medurs här, och omvänt för subtraktion. Det man speciellt ska lägga märke till är vad som händer vid termer med *lika* tecken.

[7] Istället för att kasta bort den minst signifikanta biten lagras den ofta i processorns carryflagga. Det viktiga är här vad som händer vid den mest signifikanta biten — den måste bevaras, den innehåller ju talets tecken.

Exempel: Spill

Addera 3_{10} och 2_{10} med cirkulärgrafen i figur 4.1.

I basen 2 börjar vi vid `011` och stegar sedan fram medurs genom `100` till `101` och summan är tydligen `101` $= -3_{10}$!

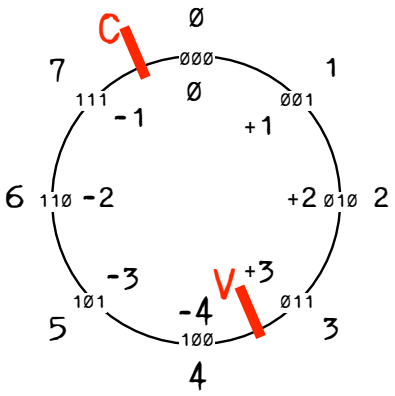
■

Det var ju egendomligt att en addition av *positiva tal* kan ge ett *negativt tal* som resultat, men det är helt konsekvent med vår cirkulärgraf. Orsaken kan naturligtvis härledas till vår begränsade ordlängd. Med en längre ordlängd skulle ovanstående addition fungera — ända tills vi återigen adderar oss över gränsen mellan positiva och negativa tal.

Fenomenet ovan kallas *spill* (*overflow*) och ställer till problem för oss då vi efter varje addition måste kontrollera om spill uppstått. Om spill föreligger är resultatet alltså felaktigt. Lägg märke till att det inte är något fel i själva additionen, bitarna har adderats på rätt sätt, det är vår *tvåkomplementstolkning* som ställer till det. Om vi istället för tvåkomplementstal enbart tänker oss positiva tal hela cirkeln runt så stämmer additionen.

Spill uppkommer om tal med lika tecken ger en summa med annat tecken.

Spill är bara intressant vid 2-komplementstal, annars är ju ”annat tecken” omöjligt! En följd av detta är att om addition av teckenlösa tal ger spill kan man bortse från detta.



Figur 4.2.: *Cirkulärgrafen åskådliggör även carry (C) och spill (V). När talområdet vid teckenlösa tal, utanför cirkeln, överskrids uppstår carry. För tvåkomplementstal är motsvarigheten spill och sker då resultatet har felaktigt tecken.*

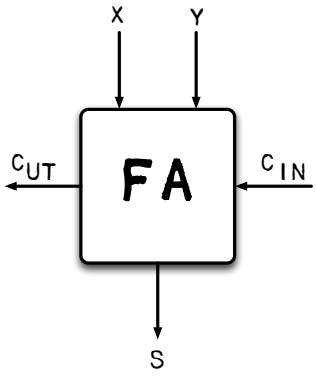
På samma sätt visualiserar cirkulärgrafen när en minnessiffra, *carry* skapas vid övergång mellan `111` och `000`. Lägg märke till att närvaron av en carry inte innebär något problem om talen är tvåkomplementskodade, då är övergången från negativa till positiva tal helt legitim. Carry uppstår även vid övergång från andra hållet det vill säga subtraktion, men motsvarar då en lånesiffra och kallas *borrow*.

Carry uppkommer vid övergång mellan `111..11` och `000..00`. *Borrow* vid omvänd övergång.

Carryn är enbart av intresse om vi tänker oss de ingående talen som teckenlösa. Sker carry i detta fall har talområdet överskridits.

4.11. Hårdvara

Med addition avklarad på papper ska vi se hur den kan utföras i hårdvara. Från digitaltekniken är *heladderaren* bekant och det är heladderaren som är det fundamentala byggblocket.



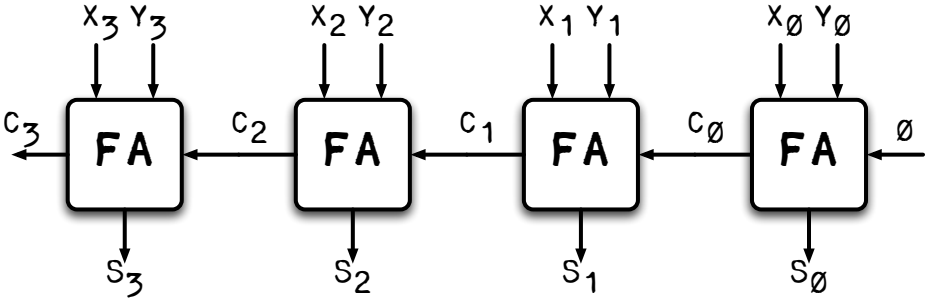
Figur 4.3.: Fulladderaren som vi minns den från digitaltekniken adderar tre inkommande bitar x, y och c_{in} till två bitar $\{c_{ut}, s\}$.

Fulladderaren[8] adderar x, y och en inkommande carry c_{in} och genererar en summa s och en utgående carry, c_{ut}. Med en dylik fulladderare kan man konstruera en godtyckligt bred adderare genom att lägga flera FA bredvid varann och knyta ihop carry-kedjorna.

För att lättare hantera de kommande uttrycken inför vi nu en definition av de *vektorer* av `1`:or och `0`:or som utgör de enskilda talen:

$$\begin{aligned}\mathcal{X} &= \{x_3\, x_2\, x_1\, x_0\}\\ \mathcal{Y} &= \{y_3\ y_2\, y_1\, y_0\,\}\\ \mathcal{S} &= \{s_3\ s_2\, s_1\, s_0\,\}\end{aligned}$$

Med dessa beteckningar kan en fyra-bitars fulladderare tillverkas enligt:



Figur 4.4.: Fulladderare för fyra bitar med fyra bitars resultat och en utgående carry.

Vi får en utgående carry $\mathtt{C}_3$ och en inkommande carry från höger som alltid är noll. Om den var `1` skulle vi utföra $x + y + 1$ och det vill vi ju inte.

Notera att denna hårdvara är identisk för både teckenlösa tal och tvåkomplementskodade tal.

[8]Fulladderaren kallas även *heladderare*. Eftersom förkortningen för heladderare, HA, även kan misstolkas för *halvadderare* undviker jag det namnet.

4.11.1. Spillindikator

Adderaren är hittills tämligen värdelös om den inte kan säga till om talområdet överskridits. För teckenlösa tal räcker det att hålla reda på utgående carryn. För tvåkomplementskodade behövs enligt cirkulärgrafen en speciell spillindikator.

Vi har tidigare sett när spill uppstår i cirkulärgrafen, men när är det *egentligen*? Om vi specialstuderar fulladderaren längst till vänster — det är ju i den ändan teckenbiten sitter — kan vi skapa en tabell:

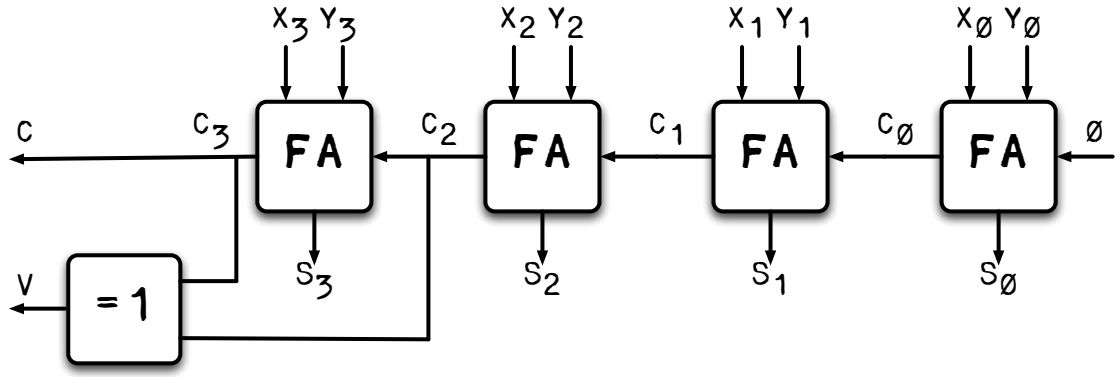
x_3	y_3	c_2	c_3	s_3	spill?	V
0	0	0	0	0	nej	0
0	0	1	0	1	ja	1
0	1	0	0	1	nej	0
0	1	1	1	0	nej	0
1	0	0	0	1	nej	0
1	0	1	1	0	nej	0
1	1	0	1	0	ja	1
1	1	1	1	1	nej	0

Längst till höger har vi tillfogat en kolumn med spill angivit. Resonemanget är vår kunskap om att addition av två positiva tal aldrig kan ge negativt resultat. Om resultatet byter tecken har spill inträffat. Kom ihåg att vi nu betraktar talen som tvåkomplementskodade, det vill säga de har en teckenbit.

Enligt tabellen inträffar spill bara i två av fallen. Kan vi nu bara konstruera logik för att identifiera vilka insignalkombinationer som ger detta kan vi tillverka oss en *spillindikator*. Nu kan man naturligtvis sätta sig att tillverka Karnaugh-diagram för att reda ut logiken, men lite inspektion av tabellen räcker faktiskt. Det är inte ofta naturen är på vår sida, men i just det här fallet blir logiken mycket enkel:

$$\mathrm{V} = c_3 \oplus c_2$$

Det är en enkel sak att komplettera vår adderare med denna indikator och vi har nu plötsligt en fullt fungerande adderare för tvåkomplementstal, den kan addera positiva såväl som negativa tal och i båda fallen meddela att talområdet överskridits.



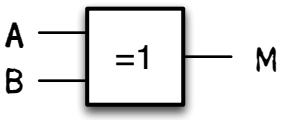
Figur 4.5.: Fulladderaren kompletterad med spill-signal V och utgående carry C tydligare markerad.

4.11.2. Subtraktion

Vi behöver kunna utföra subtraktion också. Visst vore det bra om samma hårdvara kunde utnyttjas? Vi har ovan sett att addition och subtraktion hänger intimt ihop enligt $\mathcal{X} - \mathcal{Y} = \mathcal{X} + (-\mathcal{Y})$, där $-\mathcal{Y} = \overline{\mathcal{Y}} + 1$. Alltså, sammantaget, $\mathcal{X} - \mathcal{Y} = \mathcal{X} + (-\mathcal{Y}) = \underline{\underline{\mathcal{X} + \overline{\mathcal{Y}} + 1}}$.

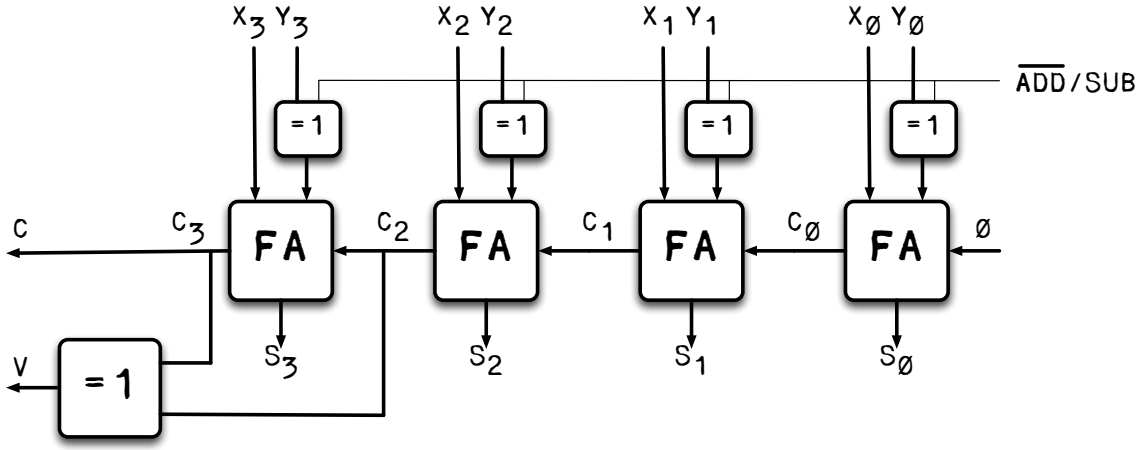
Allt i högerledet — additioner — kan vi redan utföra. Men vi behöver också kunna invertera ena talet för tvåkomplementeringen $\overline{y}$. Således behövs en styrbar inverterare, det vill säga styrbar så att den kan fås att invertera eller inte beroende på en yttre styrsignal som anger addition eller subtraktion.

Det visar sig att det är precis vad en XOR-grind gör:



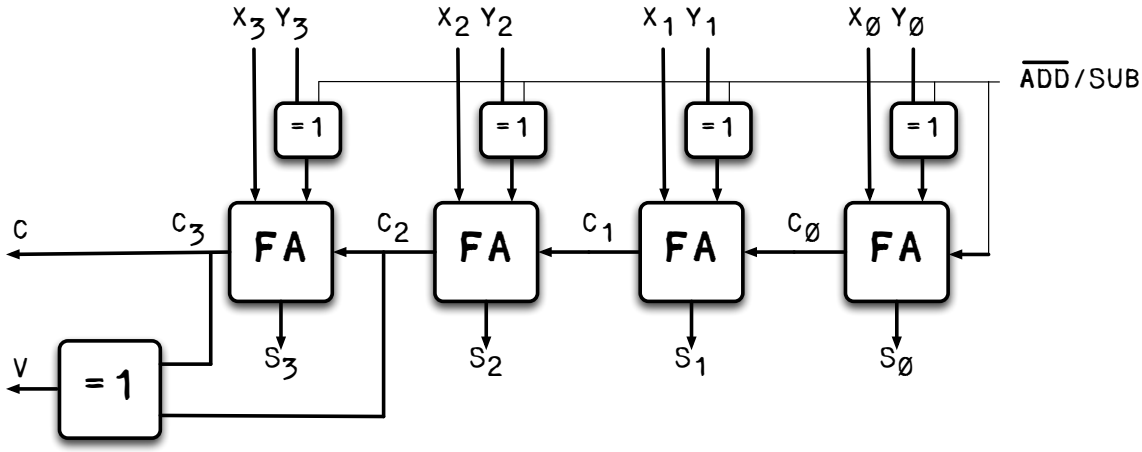
A	B	M
0	0	0
0	1	1
1	0	1
1	1	0

Så nu kan vi införa en styrsignal $\overline{\text{ADD}}/\text{SUB}$ som automatiskt genomför inverteringen av y. Om signalen är låg adderas talen och om den är hög inverteras y.



Figur 4.6.: Fulladderaren utrustad med styrbar invertering för insignalen y.

Återstår avslutningsvis att även addera 1 vid subtraktion. Det kan lösas genom att låta en 1:a anslutas till c_{in} vid minst signifikant bit. Och denna 1:a tar vi naturligtvis från $\overline{\text{ADD}}/\text{SUB}$ som ju är hög vid subtraktion och låg annars.



Figur 4.7.: Fulladderaren avslutningsvis komplett utrustad för addition och subtraktion.

Exempel: Räkna hemma! Spill och carry

Avgör om nedanstående beräkningar ger spill och/eller carry! Vilka har gett korrekt resultat?

	00000110	(+6)		01111111	(+127)		00000100	(+4)		00000010	(+2)
+	00001000	(+8)	+	00000001	(+1)	+	11111110	(–2)	+	11111100	(–4)
=	00001110	(+14)	=	10000000	(–128)	=	00000010	(+2)	=	11111110	(–2)
	C=	V=		C=	V=		C=	V=		C=	V=

	11111110	(–2)		10000001	(–127)			
+	11111100	(–4)	+	11000010	(–62)			
=	11111010	(–6)	=	01000011	(+67)			
	C=	V=		C=	V=			

■

5. Adresseringsmoder

För att kunna använda processorns fulla styrka måste man behärska dess *adresseringsmoder*. Med adresseringsmoder menas de olika sätt man som assemblerprogrammerare kan använda sig av för att lokalisera de data man behöver.

Förr brukade processorer ha en stor mängd adresseringsmoder att välja mellan. Det var praktiskt ur assemblerprogrammerarsynpunkt men kostsamt ur hårdvarusynvinkel. Det visade sig dock, i början av 1980-talet, att en enklare processor med både reducerat antal adresseringsmoder och färre instruktioner, en RISC-processor (*Reduced Instruction Set Computer*)[1], mycket väl kunde hävda sig prestandamässigt mot dåtidens konventionella processorer.

Det kunde till och med vara så att det ibland var snabbare att *syntetisera* vissa instruktioner i en konventionell dator med flera RISC-liknande instruktioner än att köra den ursprungliga instruktionen!

RISC-arkitekturen fick snabbt efterföljare och speciellt i mikrokontrollersegmentet är det numera svårt att hitta någon processor som inte bär åtminstone vissa likheter med denna ursprungliga RISC.

Även vår mikrokontroller hävdar sig vara av RISC-typ även om det avskräckande antalet instruktioner kan tyda på motsatsen. När det gäller antalet adresseringsmoder är dock tydligt en RISC-processor.

De adresseringsmoder kursens mikrokontroller kan hantera är

Mod	Exempelkod	
1. Omedelbar	`subi r20,$12`	
2. Register direkt	`com r16`	`add r20,r21`
3. Data direkt (Absolut)	`lds r20,$A3`	`sts $A3,r20`
4. Data indirekt (Indirekt)	`ld r20,X`	`st Y,r16`
— med förskjutning (offset)	`ldd r20,Y+$05`	`std Z+3,r20`
— med post-inkrement	`ld r20,X+`	`st Y+,r16`
— med pre-dekrement	`ld r20,-X`	`st -Y,r16`

Som tidigare nämnts skiljer man dessutom i allmänhet på att ange adresser som *absoluta* eller *relativa*:

1. Absolut adressering innebär att måladressen anges i instruktionen, i ”klartext”, *addr*=`$2FB`.
2. Relativ adressering innebär att måladressen fås av att addera en angiven förskjutning (*offset*) relativt programräknaren, *addr*=`PC`+*offset*.

Omedelbar och *register direkt*-moderna har vi redan använt i exemplen hittills. Vi koncentrerar oss här på de övriga och börjar med mod 3, Data direkt.

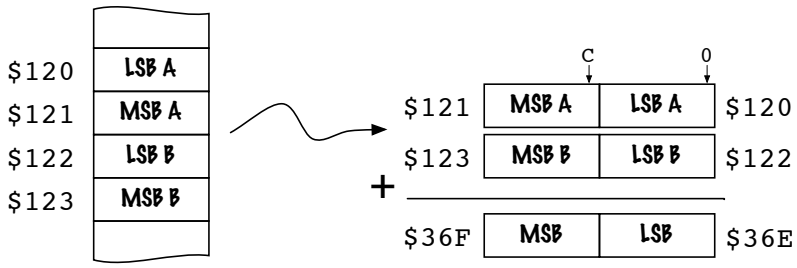
Mod 3 — Data direkt (absolut adress)

Här anges datat genom den minnesadress den ligger på. Vi antar att LSB (*Least Signifikant Byte*), den minst signifikanta byten av talet, ligger på första adressen och MSB (*Most Signifikant Byte*) en adress högre, ”`MSB:LSB`” på ”*addr+1*:*addr*”. Detta överensstämmer med tillverkarens konvention och tidigare exempel.

[1] https://en.wikipedia.org/wiki/Reduced_instruction_set_computing

Exempel: Absolut adress

Operanderna A och B består vardera av 16 bitar och finns lagrade på adresserna `$120` respektive `$122`. Beräkna summan av dem och placera resultatet på adress `$36E`.



Då vi vet adresserna till talen anger vi dem direkt i instruktionerna. Även resultatet anger vi på samma sätt.

En 16-bitars addition utförs som två 8-bitars additioner. Instruktionerna `lds` och `sts` påverkar inte flaggor varför den ur `add` uppkomna carryn `C` kan användas med `adc` senare.

Koden blir slutligen snarlik den vi gjort tidigare för 16-bitarsaddition:

```
lds  r16,$120     ; lds och sts påverkar inte flaggor
lds  r17,$122
add  r16,r17       ; Addition utan ingående carry.
sts  $36E,r16     ; Lagra LSB av resultatet
lds  r16,$121
lds  r17,$123
adc  r16,r17      ; Addition med ingående carry
sts  $36F,r16     ; Lagra MSB av resultatet
```

Instruktioner för att kopiera data (här `lds`/`sts`) påverkar inte flaggorna i `SREG`.

■

Mod 4 — Data indirekt (SRAM/FLASH)

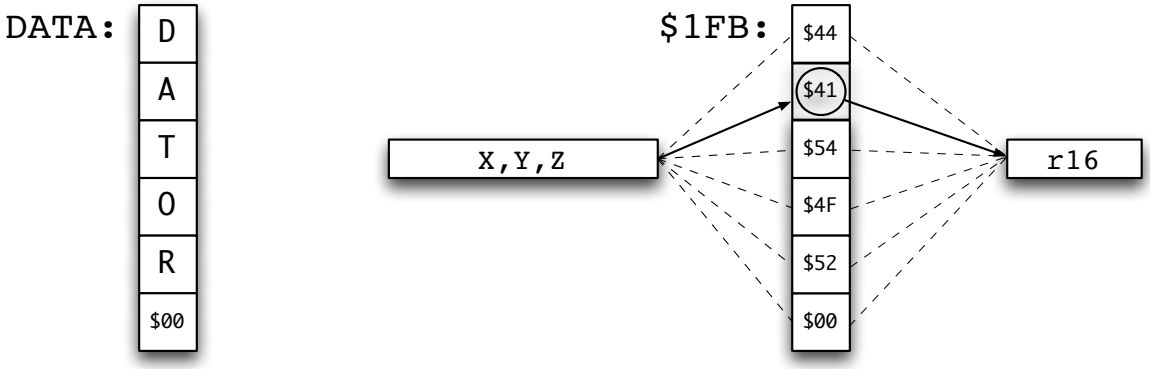
Efter ett tags programmerande märker man att det behövs ”bra” adresseringsmoder för att tillåta komplicerade metoder för att hitta rätt data. Precis som i flera högnivåprospråk är *pekare* lösningen på många datastruktursproblem. Med adresseringsmoden *data indirekt* kan vi även i vår processor använda pekare.

SRAM I processorn kan registren `X`, `Y` eller `Z` användas som pekare mot `SRAM`. När väl datat är utpekat kan det sedan läsas in i ett generellt register.

Exempel: Minnespekare

Traversera en sträng, det vill säga peka ut varje bokstav i tur och ordning!

Med en *sträng* menas en följd av tecken i minnet. Ofta, men inte nödvändigtvis, är en sträng ett stycke läsbar text och ofta markeras strängens slut med värdet `$00`, det så kallade `NUL`-tecknet. Här börjar strängen på den symboliska adressen `DATA:`.



Figur 5.1.: Den NUL-avslutade strängen DATOR återfinns ASCII-kodad på byte-adressen $1FB. Ett pekarregister kan nu successivt peka ut tecken för tecken för vidare befordran till ett register.

Vi bestämmer oss för att använda X-registret som består av XH:XL=r27:r26 så adressen måste delas upp innan den läggs in i X. LSB skall läggas på den lägre adressen som vanligt. För att inte behöva hålla detta faktum i huvudet kan man istället för r26 skriva XL för att beteckna LSB. Motsvarande för r27 är XH.

Det är också praktiskt att använda skrivsätten HIGH() och LOW() för att urskilja de båda 8-bitars talen ur ett 16-bitars tal.

Koden blir slutligen:

```
    ldi     XH,HIGH(DATA)
    ldi     XL,LOW(DATA)
NEXT:
    ld      r16,X
    cpi     r16,0
    breq    KLAR
    call    PROCESS
    adiw    XL,1    ; ev bara 'inc XL' om XH garanterat samma
    :
    jmp     NEXT
KLAR:
```

adiw, *Add Immediate Word*, utför 16-bitars additioner. Argumentet är den lägsta byten av två (XL, YL eller ZL) och talet som adderas kan vara mellan 0 och 63. adiw kan dessutom även användas på registerparet r25:r24.

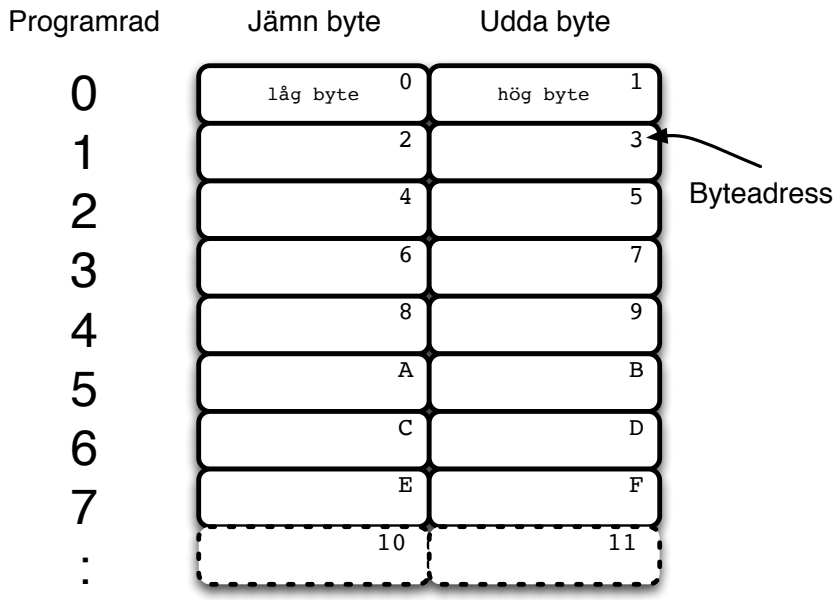
■

FLASH FLASH-minnet (programminnet) består av 16-bitars instruktioner som måste börja på jämn (byte)adress 0, 2, 4 . . . , motsvarande programrader blir då 0, 1, 2, . . . Se figur 4.1.

Z-registret kan peka ut enstaka bytes i FLASH-minnet med instruktionen lpm, *Load from Program Memory*. lpm finns i tre varianter:

```
lpm                 r0 får det Z pekar på (Z underförstås)
lpm   r16,Z         r16 får det Z pekar på.
lpm   r16,Z+        r16 får det Z pekar på och Z ökas på med 1.
```

Nyttan med lpm är främst för tabelluppslagning av konstanter eller permanenta strängar i FLASH-minnet. Instruktionen påverkar likt övriga ”flytta-instruktioner” inte flaggorna.



Figur 5.2.: Man måste skilja på **programrad** och (byte)**adress**. Då det går åt två bytes per programrad kommer programrad 0 innehålla byteadress 0 och byteadress 1 enligt figuren. En pekaradress pekar alltid ut en enstaka byte. Se appendix för en mer detaljerad beskrivning.

Exempel: Tabell i FLASH

Definiera en tabell i FLASH-minnet. Använd sedan `lpm` för att hämta värden ur tabellen.

```
TAB:    .db 0, 1, 2, 3, 4, 5  ; jämnt antal bytes definieras

LOOKUP:                       ; Slå upp i tabellen
       ldi   ZH,HIGH(TAB*2)   ; Beräkna tabellstart
       ldi   ZL,LOW(TAB*2)    ; ---''---
       lpm   r16,Z            ; Hämta 0
       call  PROCESS          ; Gör något med det hämtade värdet
       adiw  ZH:ZL,1          ; Peka ut nästa
       lpm   r16,Z            ; Hämta 1
       call  PROCESS          ; Gör något med det hämtade värdet
         :                    ; och så vidare
```

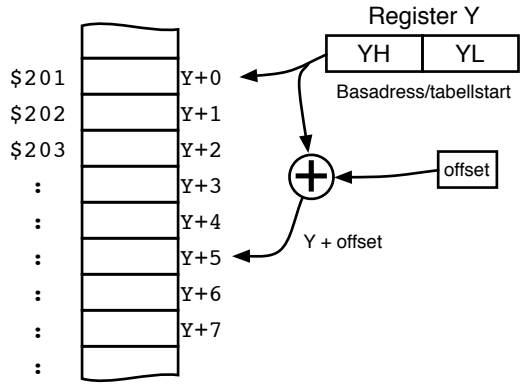
Med `.db` (*Define Byte*) kan vi lägga in enstaka byte i FLASH-minnet och måste då se till att instruktionerna inte börjar på udda byte. I praktiken betyder det att våra tabeller måste innehålla ett jämnt antal bytes.

Mod 4 — Data indirekt med förskjutning

Data indirekt-moden kallas också indexerad med förskjutning då den använder dels ett indexregister/pekarregister, `Y` eller `Z`, som *basadress* dels en *konstant* förskjutning relativt basadressen. Den effektiva adressen fås genom att addera konstanten till indexregistrets innehåll.

Vid en första anblick är denna mod väsentligen samma som föregående (data indirekt) och kan tyckas vara onödig. Det finns dock ett typfall där denna adresseringsmod är motiverad: Tabelluppslagning/indexering i SRAM.

Indexering Om man låter pekarregistret vara adressen till tabellens början kan man peka ett *offset* antal bytes in i tabellen.



Tyvärr tillåts förskjutningen bara vara en siffra 0–63 och fungerar bara med `Y`- och `Z`-pekarna. Förskjutning kan inte användas med `X`-pekaren.

Exempel: Data indirekt med förskjutning

Använd adressering med förskjutning för att i exemplet ovan kopiera en byte från `Y+6` till `Y+1`.

```
ldi  YH,HIGH($201)  ; Använd Y-pekaren
ldi  YL,LOW($201)
ldd  r16,Y+6        ; load with displacement
std  Y+1,r16        ; store with displacement
```

Mod 4 — Data indirekt med post-inkrement

När man använder indexerad adressering enligt ovan — och speciellt i det beskrivna fallet med strängar — är det ofta nödvändigt att indexera sig fram till nästa element.

Det kan göras ”manuellt”, till exempel med instruktionen `adiw`, men också med hjälp av *post-inkrement* direkt i instruktionen.

Exempel: Data indirekt med post-inkrement

Skriv koden för rutinen `SEND` som sänder en `NUL`-terminerad sträng belägen i `FLASH`-minnet. Sändningen i sig behöver bara antydas.

```
STRING:  .db   ”DATORTEKNIK”,0
SEND:    ldi   ZH,HIGH(STRING*2)  ; Använd Z-pekaren
         ldi   ZL,LOW(STRING*2)
SEND1:   lpm   r16,Z+             ; load with post increment
         cpi   r16,0              ; NUL?
         breq  DONE
         :
         :     (kod för sändningen)
         :
         jmp   SEND1
DONE:    :     (klar)
```

■

Mod 4 — Data indirekt med pre-dekrement

I en del fall vill man inte *öka* indexet utan istället *minska* det. För detta finns adresseringsmoden *Data indirekt med pre-dekrement*. Den fungerar på liknande sätt som moden ovan men räknar ner istället. Dessutom sker som namnet anger denna nedräkning *innan* indexeringen sker.

Exempel: Data indirekt med pre-dekrement

Använd instruktionerna `st` och `ld` med pre-dekrement. Notera var minustecknet hamnar.

```
st  -Z,r25

ld  r18,-Z
```

■

Lägg märke till att det — åtminstone i denna processor — bara finns pre-dekrement och post-inkrement *inte* pre-inkrement eller post-dekrement. *Räkna upp efter eller räkna ner före*.

Med de hittills beskrivna adresseringsmoderna kan man programmera godtyckligt komplicerade program. Vi skulle även kunna nöjt oss utan postinkrement- och predekrementmoderna men de är så användbara att det vore synd att inte känna till dem.

6. Subrutiner och stacken

Ofta behöver man utföra en bestämd avgränsad uppgift flera gånger i ett program. Det kan till exempel handla om att läsa av en tangent, tända en lysdiod, skriva en bokstav till en display, göra en speciellt krånglig beräkning, vänta en bestämd tid och så vidare.

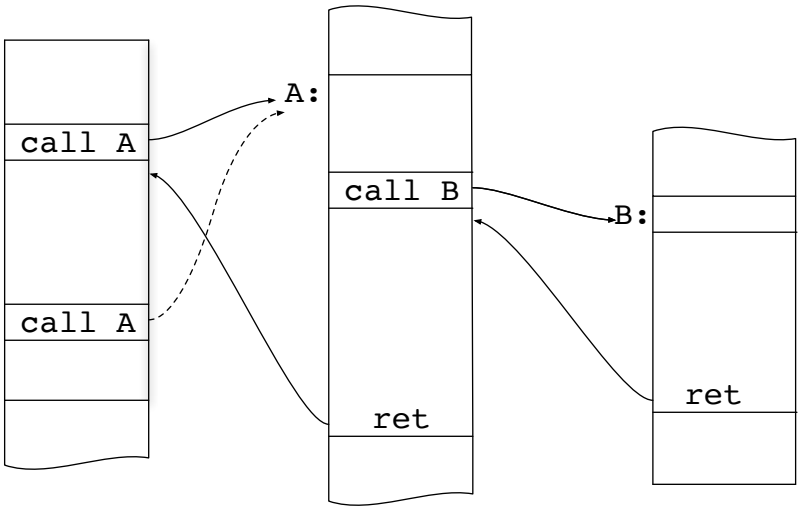
Som vi hittills lärt oss är enda möjligheten att göra dessa uppgifter att programmera dem på de ställen där de behövs i programmet. Det innebär att alla de programrader uppgiften kräver måste klistras in i programmet där de behövs.

Visst vore det praktiskt att bara ha programraderna för uppgiften på *ett* ställe och sedan bara referera till dem varje gång de behöver göras? Det skulle inte bara betyda att vi vet *var* uppgiften utförs utan skulle också hjälpa oss att strukturera hela programmet.

Vi kan redan nu göra detta genom att lägga den önskade programrutinen på någon plats i minnet och sedan hoppa med `jmp` till detta ställe från huvudprogrammet. Det är inga som helst problem att göra detta hopp till rutinen och sedan låta rutinen hoppa tillbaka till huvudprogrammet med ett avslutande `jmp`.

Men det blir omedelbart problem om rutinen ska anropas från flera *olika* ställen i huvudprogrammet. Då måste ju också *återhoppen* ske till olika ställen. Men våra hoppinstruktioner `jmp` hoppar ju bara till ett ställe, adressen är fast.

Vi söker alltså följande beteende:



Här krävs uppenbarligen en ny mekanism. De vanliga hoppen duger inte eftersom de inte har någon aning *varifrån* hoppen skedde. Det är här *subrutiner* kommer väl till pass.

Vi förstår att de instruktioner som orsakar ett subrutinanrop måste uppfylla tre villkor:

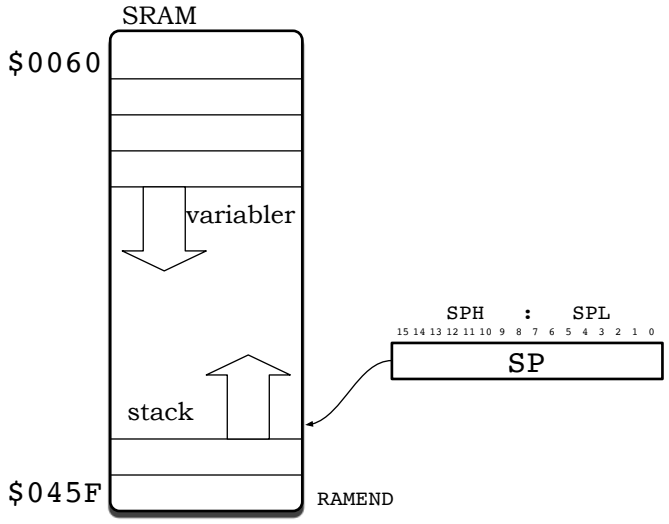
- Programflödet måste styras över till subrutinen.
- En återhoppsadress måste sparas.
- Subrutinen måste, efter förättat värv, kunna hoppa till återhoppsadressen.

Subrutiner är rutiner i ett program som kan anropas från var som helst i programmet och som sedan återhoppar till programraden precis efter där anropet skedde. Subrutinerna måste alltså ha ett ”minne” som gör att de minns vart de skall hoppa när de är klara.

Ursprungligen infördes subrutiner som ett sätt att erhålla minskad total programstorlek men de blev efter ett tag viktiga verktyg för att inte minst även ge programmen överblickbar struktur och minskad komplexitet.

För att använda några programrader som en subrutin måste de anropas med antingen instruktionerna `call/rcall` (*Absolut Call* eller *Relative Call*)[1].

Dessa instruktioner fungerar precis som `jmp` med den skillnaden att de samtidigt lagrar undan adressen till nästföljande instruktion på en *stack*.



Stacken är i verkligheten ett stycke minne (SRAM) dit man pekar med en *stackpekare*. På denna processor pekar stackpekaren, registret `SP`, ut första lediga plats på stacken.[2] Stacken är lite egendomlig i och med att den växer mot mindre adresser. Det är vanligt bland processorer att stacken växer ”nedåt” på detta sätt. Nu är detta inte något egentligt problem för oss eftersom vi i allmänhet inte märker något av det; de instruktioner som använder sig av stacken tar automatiskt hänsyn till det.

För att genomföra återhoppet duger nu inte ett vanligt hopp — våra vanliga hopp har ju ingen aning om vad som finns på stacken, eller ens om att det finns en stack — utan instruktionen `ret` (*return*) måste användas. `ret` genomför återhoppet till den adress som ligger på stacken.

Genom att den senaste subrutinen skall återhoppa till den senast ditlagda adressen på stacken och så vidare, kommer stacken alltid att vara korrekt parad med rätt adress för subrutinåterhopp.

Det är enkelt att använda subrutiner och de förenklar programmeringen i och med att det är lättare att hålla en vettig struktur på sitt program. Det finns fyra saker att tänka på:

- Hopp till subrutin görs med `call` eller `rcall`
- En subrutin måste avslutas med `ret`
- Registret `SP=SPH:SPL` är stackpekare och är ett I/O-register
- Stackpekaren måste initieras med följande rader

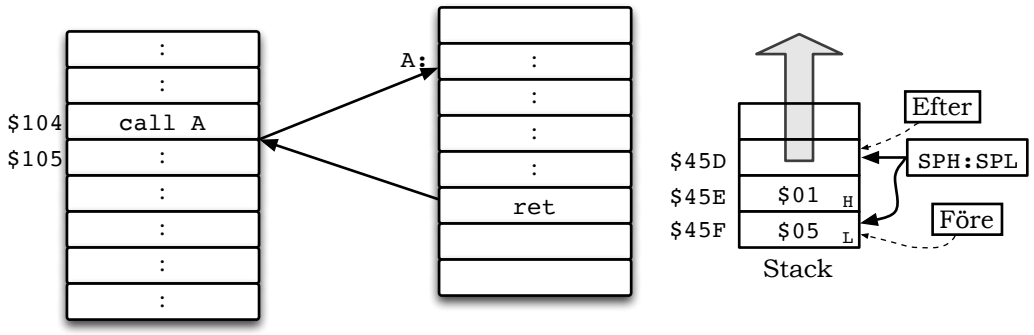
[1] På ATmega16-processorn finns också `icall` (*Indirect Call*).
[2] Det behöver inte vara så. I processorn M68000 pekas till exempel sist ditlagda element ut.

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

Konstanten `RAMEND` är definierad filen *m16def.inc*:

```
.equ    RAMEND = $045F
```

Hela subrutinmekanismen för AVR-processorn blir nu



Det enda som skiljer en allmän rutin från en subrutin är att den senare avslutas med en `ret` som i exemplet nedan (vi kommer använda paritetsrutinen senare också):

Exempel: Udda paritet

Skriv en subrutin som beräknar *udda paritet* av byten i `r17`.

Med udda paritet menas att summan av alla `1`-or i ett binärt tal skall vara udda. Om ett åttabitarstal förses med paritet är denna den mest signifikanta biten `P` i byten, ”`Pxxxxxxx`”.

Subrutinen måste räkna antalet `1`-or i de 7 lägre bitarna och om antalet är jämnt sätta paritetsbiten, `P`, till `1` annars till `0`. Talet ska inte förstöras, och rutinen får använda enbart `r17`. Vi kan förutsätta att $P = 0$ initialt.[3]

```
PARITET:
          ldi   r16,9
          clr   r18
          clc             ; clear carry
LOOP:
          ror   r17
          brcc  NOLLA
          inc   r18       ; sum += index
NOLLA:
          dec   r16
          brne  LOOP      ; for all bits
          lsr   r18       ; LSB -> C
          brcs  UDDA
          ori   r17,$80   ; set parity
UDDA:
          ret
```

■

[3]Eller så kan vi se till att det är så i början av rutinen.

Lägg märke till hur sista raden gjorde att denna rutin är och kan anropas som en subrutin: Instruktionen `ret`. Detta gör att den måste anropas med `call` eller `rcall`.

För subrutiner på Atmels AVR-processorer har konstruktörerna valt följande konvention (SP är stackpekaren, $M(SP)$ är minnet som SP pekar ut):

Instruktion	Innebörd
`call` *addr*	$M(SP) := PC + 1, SP := SP - 1, PC := addr$
`ret`	$SP := SP + 1, PC := M(SP)$

Vi ser att detta är förenligt med adresseringsmoderna *predekrement* och *postinkrement* utan att ytterligare moder behöver införas.

Som programmerare behöver jag inte röra stackpekaren mer än vid initieringen, allt sköts transparent och automatiskt av anrops- och återhoppsinstruktionerna.

Det kan vara intressant att fundera på svaren till dessa frågor:

- Vad händer om en subrutin avslutas med `jmp` istället för `ret`?
- Vad händer om en subrutin avslutas med `call` istället för `ret`?
- Vad händer om `jmp` används för att hoppa till en korrekt avslutad subrutin?

6.1. Lokala variabler

En subrutin skall kunna anropas från flera ställen i den övriga programkoden. För att göra detta så smärtfritt som möjligt får inte subrutinen förstöra innehållet i de register huvudprogrammet, det anropande programmet, använder. Huvudprogrammet kanske använder de övriga registren till något väsentligt och blir nog förvånad om subrutinen har ändrat i dem?

Vi måste alltså se till att subrutinen inte förändrar innehållet i andra register än det huvudprogrammet förväntar sig.

Det enklaste sättet att spara undan innehållet i de register rutinen använder, är att vid början av subrutinen lägga detta innehåll på stacken för att som sista moment i rutinen plocka tillbaka dem i de ursprungliga registren igen.

Det anropande programmet hoppar till subrutinen med ett `call`, som vanligt, och ser till att `r17` förses med rätt värde innan, till exempel:

```
        ⋮       ⋮
        lds     r17,$110
        call    PARITET
        sts     $110,r17
        ⋮       ⋮
```

Subrutinen skapar sedan egna lokala variabler genom att inledas och avslutas med programrader som sparar undan och återställer de register rutinen använder:

```
        ⋮       ⋮
PARITET: push   r16
        push    r18
        ⋮
        ⋮       ; kod som tidigare
        ⋮
        pop     r18
        pop     r16
        ret
        ⋮       ⋮
```

Instruktionerna `push` och `pop` använder stacken men även detta sker transparent för användaren. Rutinen måste dock lämna stacken i opåverkat skick. Vad händer om rutinen ovan gör `ret` utan föregående `pop`?

6.2. Parameteröverföring

Subrutiner hjälper till på ett högst avsevärt sätt att strukturera programflödet. För att ytterligare förenkla subrutinanropen måste vi också hantera parametrar och argument till dem.

Här visas två olika metoder, överföring via register och överföring via stacken. Den senare förekommer ofta vid kompilatorer för högnivåspråk men är i allmänhet rätt püsslig och nämns här och nu mest som exempel på att stacken inte bara kan innehålla returadresser. Rutinen `PARITET` enligt tidigare används i exemplen.

6.2.1. Parameteröverföring via register

Det enklaste sättet, och kanske det lämpligaste vid handassemblering, är att överföra en parameter till en subrutin genom att lägga parametern i ett register och konstruera subrutinen så att den förväntar sig sin parameter i just detta register.

```
⋮       ⋮
lds     r17,$110
call    PARITET
sts     $110,r17
⋮       ⋮
```

En nackdel är att `r17` i och med detta är ”hårdkodat” varje gång subrutinen skall anropas. Kanske använder den anropade rutinen registret `r17` till något annat och måste bolla med registren för att kunna frigöra `r17`? En annan nackdel är naturligtvis att argumentet i detta fall inskränker sig till vad som kan rymmas i ett register.

6.2.2. Parameteröverföring via returstacken

Ett annat vanligt sätt är att låta huvudprogrammet lagra parametern på returstacken innan subrutinanropet sker. Subrutinen vet sedan att dess argument finns att hämta på stacken, det vill säga anropet till rutinen skulle se ut som

```
⋮       ⋮
lds     r25,$110
push    r25
call    PARITET
pop     r25
sts     $110,r25
⋮       ⋮
```

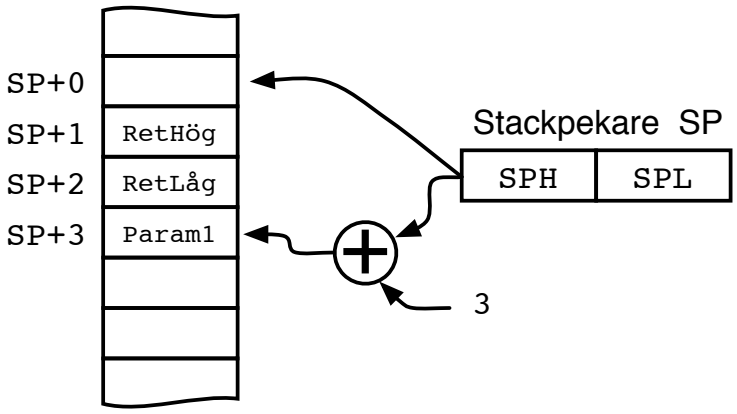
På detta sätt behöver inte `r25` hårdkodas som i det förra fallet. Programmet vet inte, och skall strängt taget inte behöva veta, vilka register `PARITET` använder.

Lägg märke till hur den anropade rutinen inte kan poppa argumentet hur som helst nu: Överst på stacken ligger ju återhoppsadressen (två bytes) och först därunder kommer argumentet!

Lösningen är att använda en adresseringsmod som tillåter adressering relativt stackpekaren, *indirect with displacement*, där en förskjutning (eng *displacement*) anger var argumentet ligger.

Som exempel visas stackinnehållet som subrutinen ser det från början. Stackpekaren pekar på första lediga plats på stacken. Återhoppsadressen kräver två bytes och först därunder återfinns parametern `param1`.

Stackinnehållet när subrutinen körs blir allstå:



Med indirekt adressering med förskjutning, och i detta fall förskjutningen `+3`, kan `param1` hämtas. Notera att stackpekarens innehåll inte ändras medan minnet adresseras.[4]

I vår processor kan denna adresseringsmod inte använda den vanliga stackpekaren för indexeringen, man är hänvisad till `Y`- eller `Z`-pekaren. Vanligen kopierar man `SP` → `Y` (eller `Z`) för att därefter komma åt parametern som ovan.

Rutinen får alltså modifieras à la

```
          ⋮      ⋮
PARITET:  in     ZH,SPH
          in     ZL,SPL
          push   r17
          ldd    r17,Z+3
          push   r16
          push   r18
          ⋮
          ⋮      kod som förut
          ⋮
          pop    r18
          pop    r16
          std    Z+3,r17
          pop    r17
          ret
          ⋮      ⋮
```

Notera hur `Z`-pekaren agerar stackpekare för argumenten medan den vanliga `SP` använts för instruktionerna `push` och `pop`. Ordningen är viktig. Hade `push` utförts innan `in`-instruktionerna hade förskjutningen inte varit samma.[5]

[4] Den beskrivna adresseringsmoden är vanlig i kompilatorer för högnivåspråk som C, C++ med flera. Adressering med förskjutning sker då relativt stackpekaren.

[5] Och vad hade den blivit då? Rita en figur eller simulera!

6.2.3. Argument och returvärde via returstacken

Om subrutinen skall lämna returvärden kan huvudprogrammet i förhand bereda plats för dessa på stacken.

Betrakta följande exempel för anropet "`r20=FUNK(r16,r17)`":

```
⋮     ⋮
push  r0    ; plats för returvärde
push  r16   ; arg
push  r17   ; arg
call  FUNK
pop   r0
pop   r0
pop   r20
ret
⋮     ⋮
```

Rutinen `FUNK` är här konstruerad så att alla argument hämtas från stacken. Den är därmed helt isolerad och kan använda vilka register som helst i sitt interna arbete. Registret `r0` använts bara för att det är iögonenfallande, det är inget speciellt med det registret.

6.2.4. Komplett `PARITET`

Här kommer två kompletta versioner av `PARITET`. Först är den modifierad för att inte påverka andra register och därefter modifierad så argumentet tas från stacken. Det senare gör rutinen lämplig att ingå i biblioteksrutiner.

Den stackhanterande inledningen och avslutningen markeras av `; --` för tydlighets skull.

```
PARITET:
    push    r16
    push    r18
    ; ---
    ldi     r16,9
    clr     r18
    clc
LOOP:
    ror     r17
    brcc    NOLLA
    inc     r18
NOLLA:
    dec     r16
    brne    LOOP
    lsr     r18
    brcs    UDDA
    ori     r17,$80
    ; ---
    pop     r18
    pop     r16
    ret
```

```
PARITET:
    push    r16
    push    r17
    push    r18
    in      ZH,SPH
    in      ZL,SPL
    ldd     r17,Z+6
    ; ---
    ldi     r16,9
    clr     r18
    clc
LOOP:
    ror     r17
    brcc    NOLLA
    inc     r18
NOLLA:
    dec     r16
    brne    LOOP
    lsr     r18
    brcs    UDDA
    ori     r17,$80
    ; ---
    std     Z+6,r17
    pop     r18
    pop     r17
    pop     r16
    ret
```

För den intresserade En mindre variant av ursprungsrutinen är nedanstående kod. Går den att minska ytterligare?

```
PARITET:
    ldi     r16,6
    mov     r18,r17
    lsl     r18
LOOP:
    lsl     r18
    sbrc    r18,7
    subi    r17,-$80
    dec     r16
    brne    LOOP
    ret
```

7. Externa och interna avbrott

För effektivaste användning av en processorns resurser gäller det att få den att göra det man vill, när man vill det.

Hittills har vi sett hur ett program kan få saker utförda i subrutiner som programmet anropar: Med instruktionen `call` kan vi anropa andra programsnuttar för att sedan hoppa tillbaka med `ret` för att fortsätta som om inget hade hänt. Men det är hela tiden programmets flöde som bestämmer vad som sker.

7.1. Pollning

Med *pollning* går programmet under sin normala gång i tur och ordning och frågar (*pollar*) omgivande hårdvara om det finns något för programmet just nu. Finns det något kan programmet antingen styra exekveringen direkt till korrekt rutin, eller sätta en flagga/indikator att något inträffat till någon annan rutin som senare ombesörjer detaljerna. Metoden med flaggor kan användas för att styra programmet att befinna sig i olika *tillstånd*.

En nackdel med pollning är att en yttre enhet inte säkert kan få processorns uppmärksamhet omedelbart. Programmet noterar att hårdvaran behöver betjänas först när programflödet kommit till den rutin som frågar hårdvaran.

7.2. Avbrott

Med ett *avbrott* kan en yttre enhet signalera en *avbrottsbegäran* till processorn att den omedelbart behöver betjänas. Processorn tvingas då hoppa till en särskild subrutin, en *avbrottsrutin*, som utför det önskade.

Det låter kanske förvånande, men vi människor gör så hela tiden. Ett typexempel är telefonen: Vi går inte runt och provlyssnar i luren för att höra om någon råkar ringa just då. Vi är mer praktiska än så och har försett telefonen med en ringsignal (avbrottsbegäran) så den själv påkallar uppmärksamhet när det behövs. Ett typiskt avbrott. När vi pratat klart fortsätter vi med vad vi nu höll på med, med datorspråk gör vi en ”retur från avbrottsrutinen”.

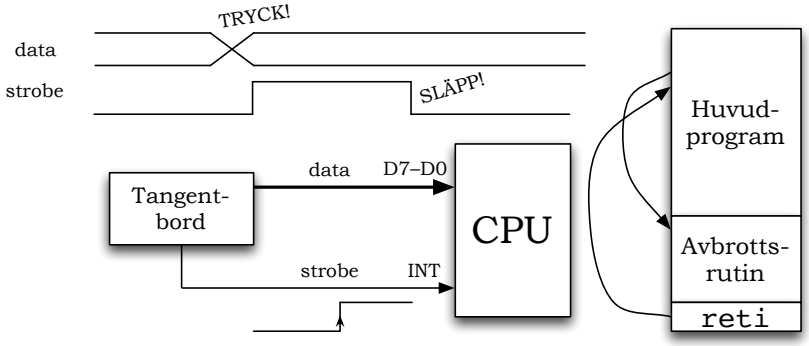
I stort sett på samma sätt går det till i en processor. Vi betraktar först avbrott i allmänhet med ett exempel.

Exempel: Avbrott

> En yttre enhet kan behöva snabb service, till exempel om en tangent på tangentbordet blivit nedtryckt eller om sekundpulsen från en yttre klocka kommer. Detta är en utmärkt situation för en avbrottsrutin.[1]

I figuren nedan läggs den nedtryckta tangentens binära värde ut på dataledningarna och påkallar sedan uppmärksamhet genom att lägga stroben hög. Denna strobe utgör avbrottsinsignal till processorn och avbrottsrutinen kommer köras. När tangenten sedan släpps faller stroben låg men då avbrottet sker enbart på stigande flank är detta ofarligt.

[1] Det kritiska är att den yttre enheten vill bli betjänad snabbt. Om den inte är så tidskänslig kanske man kan avstå från att använda avbrott och istället låta programmet, *polla*, det vill säga då och då titta efter om en tangent blivit nedtryckt.



Processorn är konstruerad så att den, innan den påbörjar varje ny instruktion, kontrollerar om någon ryckt i ”avbrottslinan”. Om så är fallet väljer den bort nästa instruktion för ögonblicket och gör ett *subrutinanrop* till avbrottsrutinen istället. Subrutinanropet sparar en återhoppsadress till följande instruktion, på stacken i vanlig ordning.

Avbrottsrutinen läser i sin tur in tecknet från tangentbordet, lagrar värdet i minnescellen för ”senaste tangent” och hoppar sedan tillbaka till huvudprogrammet.

7.2.1. Återhopp från avbrott

I denna processor stängs möjligheten att acceptera avbrott av under pågående avbrott, den så kallade *avbrottsflaggan* `I` i `SREG` nollställs för att hindra ytterligare avbrott. Ett nyinkommet avbrott kan alltså inte avbryta ett pågående avbrott. För att få processorn att acceptera avbrott igen används den speciella returinstruktionen `reti`, *Return from Interrupt.*[2]

Precis som vid vanliga subrutiner lagras återhoppsadressen på stacken. Men eftersom ett avbrott kommer ”som en blixt från klar himmel” måste avbrottsrutinen se till att alla register den använder återlämnas oförvanskade då avbrottsrutinen är färdig. Speciellt kan man vara tämligen säker på att statusflaggorna i `SREG`-registret ändras av avbrottsrutinen.

7.2.2. Spara inre tillståndet!

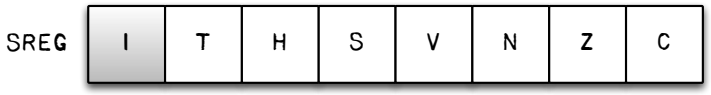
Med processorns *inre tillstånd* menar man processorns samtliga registers innehåll. Att spara hela processorns inre tillstånd är en tidsmässigt dyr affär varför man vill spara så lite som absolut möjligt, men ändå så mycket att avbrottsrutinen kan kliva in precis var som helst i programkörningen.[3]

Avbrottsrutinen måste lämna processorns inre tillstånd oförändrat när rutinen avslutas.

Denna processor sparar inget automatiskt utan det är programmeraren som får ansvara för att processorns inre tillstånd inte förstörs av ett avbrott. Det gäller speciellt att `SREG` återlämnas i korrekt skick men även övriga register som avbrottsrutinen påverkar behöver normalt återställas.

[2] Avbrott, interrupt, är en del av processorns så kallade *undantagshantering*.
[3] På denna processor kan inte en avbrottsrutin avbryta en annan avbrottsrutin eller sig själv.

h



Figur 7.1.: Avbrottsflaggan `I` är belägen i stausregistret. Normalt skall man inte påverka denna flagga manuellt, det sköts automatiskt.

En minimal avbrottsrutin utförs således som nedan till vänster. Om avbrottsrutinen i sig behöver använda `r16` måste även detta register sparas på stacken som i koden till höger.

```
INT:
    push    r16
    in      r16,SREG
     :
     :
     :
    out     SREG,r16
    pop     r16
    reti
```

```
INT:
    push    r16
    in      r16,SREG
    push    r16
     :
    pop     r16
    out     SREG,r16
    pop     r16
    reti
```

Pollning Pollning används som begrepp i två fall: Antingen som alternativ till avbrott eller som tillvägagångssätt för att — efter att avbrott inträffat — avgöra vem som skapade avbrottet.

Det senare fallet är aktuellt om flera enheter kan påkalla avbrott, det vill säga likt på bussar att alla kan trycka på knappen för att stanna bussen. Då måste processorn på något sätt kunna avgöra vem det var som önskade uppmärksamhet. Avbrottsrutinen konstrueras då så att den i tur och ordning frågar de olika enheterna ”Var det du? Var det du? Var det du? . . . ”. Olika delar av avbrottsrutinen kan då användas till att betjäna olika enheter.

7.3. Avbrott på ATMEGA16-processorn

Processorn kan ha åtskilliga avbrottskällor. Förutom det vanliga externa avbrottet vi nämnt hittills kan flera hårdvaruenheter *inuti* kapseln dessutom orsaka avbrott. Till exempel kan komplett skrivning till hårdvaruenheter som USART, TWI eller en färdig AD-omvandling signaleras med avbrott.

Till varje typ av avbrott knyts en adress till motsvarande avbrottsrutin. Dessa adresser kallas *avbrottsvektorer* och ligger i början av FLASH-minnet där de upptar adresserna `$00`–`$28`. Första programrad efter tabellen blir då `$2A`. Normal programstart efter spänningspåslag sker på rad `$0000` i programminnet och är ett hopp till rutinen (adressen) `RESET`. En programbörjan med några ifyllda avbrottsvektorer kan se ut som nedan.

```
               .org $0000          ; För tydlighet, behövs inte
       jmp     RESET               ; Reset Handler
               .org INT0addr
       jmp     INT0                ; INT0 Handler
               .org INT1addr
       jmp     INT1                ; INT1 Handler
               .org OC2addr
       jmp     TIM2_COMP           ; Timer2 Compare Handler
               .org TWIaddr
       jmp     TWI_INT             ; 2-wire Serial Interface Handler
       :
       :                           ; Massor med andra vektorer
       :
       .org    INT_VECTORS_SIZE    ; Hoppat bock över vektorerna
RESET:                             ; (INT_VECTORS_SIZE = $2A)
       ldi     r16,HIGH(RAMEND)    ; Första programraden
       out     SPH,r16
       ldi     r16,LOW(RAMEND)
       out     SPL,r16
       sei                         ; Tillåt avbrott härifrån!
```

Lägg märke till hur man i koden använt fördefinierade adresser till avbrottsvektorernas placering, ”`.org INT1addr`” och så vidare.

Efter kallstartshoppet till `RESET` fylls avbrottsvektortabellen på med övriga avbrott. Ett `INT0`-avbrott kommer köra andra avbrottsvektorn, det vill säga att den raden måste innehålla en instruktion som hoppar till avbrottsrutinen. Observera att det är ett `jmp` och inte ett `call`, avbrottsrutinen skall ändå avslutas med `reti` för att komma tillbaka till där avbrottet skedde.

Allmänt gäller att tre saker skall vara uppfyllda för att ett avbrott skall verkställas:

- Villkoret för det enskilda avbrottet är tillåtet
- Det enskilda avbrottets händelse skall ske.
- Avbrott skall överhuvudtaget vara tillåtna på processorn

I det fall då två avbrott inträffar samtidigt ”vinner” det avbrott som har lägst adress i tabellen. `INT0` har alltså högre *prioritet* än `INT1` och så vidare. Trots detta kan inte `INT0` avbryta ett pågående avbrott.

Konfigurationsbitar för avbrottsinställningar återfinns i de olika hårdvarufunktionernas I/O-register. Dessa bitar är emellanåt utspridda bland flera olika register. Här måste man läsa databladet för att få alla detaljer.

Dessutom måste bit 7, *Global Interrupt Enable* `I` i `SREG` vara aktiverad, enklast med assemblerinstruktionen `sei`.

7.3.1. Avbrottskod

För initiering av hårdvaran att använda de två avbrotten `INT1` och `INT0` måste man således vidtaga följade handgrepp.

```
    .org    $0000
    jmp     COLD

    .org    INT0addr
    jmp     AVBROTT_0

    .org    INT1addr
    jmp     AVBROTT_1

    .org    INT_VECTORS_SIZE

    ; --- Cold Start

COLD:
    ldi     r16,HIGH(RAMEND)
    out     SPH,r16
    ldi     r16,LOW(RAMEND)
    out     SPL,r16

    ; --- Configure

    ; Trig
    ldi     r16,(1<<ISC01) |
                (0<<ISC00) |
                (1<<ISC11) |
                (0<<ISC10)
    out     MCUCR,r16

    ; Activate
    ldi     r16,(1<<INT0) |
                (1<<INT1)
    out     GICR,r16

    ; Enable Interrupts Globally
    sei

WAIT:
    jmp     WAIT
```

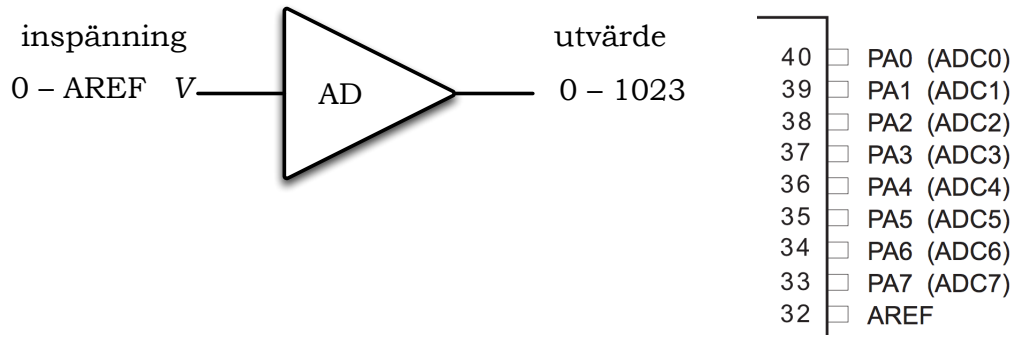
```
    ;
    ; --- ISR Interrupt Service Routines
    ;

AVBROTT_0:
    push    r16
    in      r16,SREG
    push    r16
    :
    :
    pop     r16
    out     SREG,r16
    pop     r16
    reti

AVBROTT_1:
    push    r16
    in      r16,SREG
    push    r16
    :
    :
    pop     r16
    out     SREG,r16
    pop     r16
    reti
```

.

8. AD-omvandling

Mikrokontrollern är bestyckad med en tio-bitars AD-omvandlare. En sådan omvandlar en analog (amplitudkontinuerlig) spänning till ett binärt siffervärde om tio bitar, $d_9 - d_0$, motsvarande $2^{10} = 1024$ nivåer $0 - 1023$.

Den högsta spänning som kan omvandlas är *referensspänningen* AREF som ger värdet 1023 (`$3FF`), den lägsta är 0 V och ger värdet `$000`. Referensspänningen är i DALIAs fall 5 volt dvs processorns matningsspänning AVCC, men kan väljas att tas från en yttre pålagd spänning AREF eller en intern precisionsreferens om 2.56 volt.



Figur 8.1.: AD-omvandlaren omvandlar en analog insignal mellan 0 och AREF volt till ett digitalt värde mellan 0 och 1023. Den analoga insignalen kan hämtas från någon av ingångarna `ADC7`–`ADC0`.

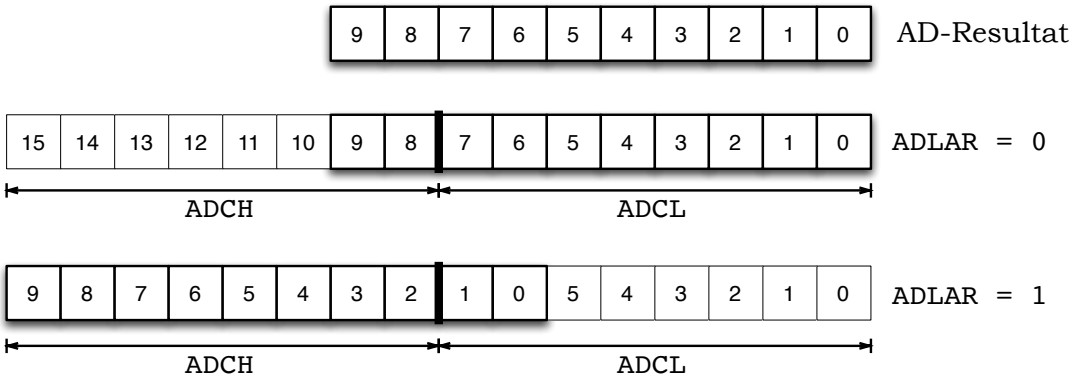
Omvandlaren kan styras att göra omvandling på en av sex ingångar `ADC7`–`ADC0`. AD-omvandlaren kommunicerar med fyra åttabitars I/O-register:

- `ADMUX` Registret väljer ingång, referensspänning och resultatets skift
- `ADCSRA` Kontrollregister för aktivering, omvandlingstakt samt start av omvandling bland annat
- `ADCH:ADCL`-registerparet innehåller slutligen omvandlingens resultat. `ADCH` måste läsas sist.

Vissa konfigurationsbitar i dessa register är viktiga att hålla reda på och beskrivs nedan. AD-omvandlarens funktion och många ytterligare inställningar beskrivs utförligt i mikrokontrollerns datablad.

Konfigurationsbitar

ADLAR Då ett tio-bitars omvandlingsresultat inte kan få plats i enbart ett register måste två register användas. Resultatet kan därmed också placeras på två sätt i ADCH:ADCL beroende på konfigurationsbiten ADLAR, *AD Left Adjust Result*:



Där vi ser att resultatet placeras höger- eller vänsterjusterat i ADCH:ADCL beroende på ADLAR.

REFS Referensspänningen kan väljas med de två bitarna REFS1 och REFS0 i ADMUX enligt tabellen nedan.

REFS1	REFS0	AREF
0	0	AREF
0	1	AVCC
1	0	—
1	1	2.56 V

Vid uppstart är bitarna nollställda vilket i vårt fall motsvarar en referensspänninng om 5 V.[1]

ADPS AD-omvandlaren innehåller delar som inte kan klockas lika fort som resten av processorn. I själva verket måste den förses med en egen klocka, AD Clock, med en frekvens inom 50 – 200 kHz för full omvandlingsprecision.

Denna klocka erhålls genom att dela ner, (eng *prescale*), processorns klockfrekvens med något lämpligt. Något sådant lämplig kan väljas med de tre bitarna ADPS2, ADPS1, ADPS0 i registret ADCSRA enligt

ADPS2	ADPS1	ADPS0	prescaler
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

(Ja, prescalern skall vara ”2” i båda översta raderna.)

[1] Se DALIAs kopplingsschema.

8.1. Omvandling med 10-bitars resultat

För vår del belyses användningen av omvandlaren enklast med två exempel, först en omvandling med 10-bits resultat. Denna omvandling innebär att vi läser av det kompletta tio-bitarsresultatet efter omvandling.

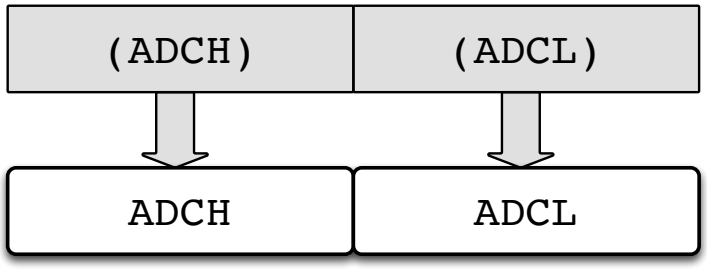
Exempel

Starta en AD-omvandling och hämta ett 10-bitars omvandlingsvärde så fort det finns tillgängligt.

En omvandling startas genom att man ett-ställer ADSC-biten (*AD Start Conversion*) i AD-omvandlarens statusregister ADCSRA. Så länge hårdvaruenheten är upptagen är biten ADSC i ADCSRA ettställd (sk *busy bit*), när den nollställs är omvandlingen klar.[2]

```
ADC10:
     ldi   r16,0             ; kanal 0, 5 V ref
     out   ADMUX,r16
     ldi   r16,(1 << ADEN)   ; AD enable
     out   ADCSRA,r16
CONVERT:
     sbi   ADCSRA,ADSC       ; starta en omvandling
WAIT:
     sbic  ADCSRA,ADSC       ; om nollställd är vi klara
     rjmp  WAIT              ; annars testa busy-biten igen
     in    r16,ADCL          ; observera ordningen, läs låg byte först
     in    r17,ADCH          ; hög byte sedan
```

Resultatregistret är dubbelbuffrat vilket innebär att ett nyare omvandlingsresultat inte flyttas till ADCH:ADCL innan ADCH är läst.



Figur 8.2.: Ett eventuellt nyare omvandlarresultat ligger färdigt att överföras till ADCH:ADCL. Överföringen sker inte förrän ADCH är läst för att inte kunna läsa resultat från två olika omvandlingar.

Läsningen av ADCH ger klartecken till omvandlaren att uppdatera registerparet. Därför är ordningen på läsningarna viktig. Det omvandlade värdet återfinns nu i r17:r16.

■

[2]En bit med denna funktion brukar kallas *busy*-bit och återfinns i både det interna EEPROM:et och yttre komponenter som LCD-displayer med flera.

8.2. Omvandling med 8-bitars resultat

Processorn kan inte göra en åtta-bitars omvandling, den gör alltid 10-bitars omvandling, men vi kan välja att läsa av de åtta övre bitarna av resultatet.

Exempel

Starta en 8-bitars AD-omvandling och hämta ett nytt värde så fort det finns tillgängligt.

Detta innebär samma program som ovan med skillnaden att resultatet vänsterjusteras för att kunna återfinnas i ADCH.

```
ADC8:
     ldi   r16,(1 << ADLAR) ; kanal 0, 5 V ref, left adjust result
     out   ADMUX,r16
     ldi   r16,(1 << ADEN)   ; AD enable
     out   ADCSRA,r16
CONVERT:
     sbi   ADCSRA,ADSC       ; starta en omvandling
WAIT:
     sbic  ADCSRA,ADSC       ; om nollställd är vi klara
     rjmp  WAIT              ; annars testa busy-biten igen
     in    r16,ADCH          ; En läsning av hög byte
```

Det omvandlade värdet återfinns nu i r16. På grund av dubbelbuffringen sker ingen förändring av ADCH:ADCL förrän läsning av ADCH är läst.

■

9. Preprocessor och kompilering

När koden är skriven måste den passera två steg innan den kan överföras till mikrokontrollern. Dessa två steg kallas tillsammans att *bygga* (eng *build*) koden.

Det första är ett *preprocessor*-steg där symboler ersätts med sina värden och enklare (statiska) uttryck kan beräknas. Det andra steget är en översättning av den resulterande preprocessade textmassan till hexadecimal kod för mikrokontrollerns FLASH-minne.

Preprocessor

Som ett inledande steg innan koden kompileras genomgår den en behandling i en *preprocessor*. Preprocessorns uppgift är att färdigställa koden så den går att kompilera, till exempel ersätts alla fördefinierade konstanter `DDRB` och så vidare med sina faktiska sifferadresser. Dessa fördefinierade konstanter är beskrivna i en *include*-fil som i Atmelstudio alltid automatiskt läggs till innan koden. I filen återfinns också storleken på tillgängligt minne med mera. Preprocessorn kan också utföra enklare beräkningar. Preprocessorns resultat går vidare till kompilering.

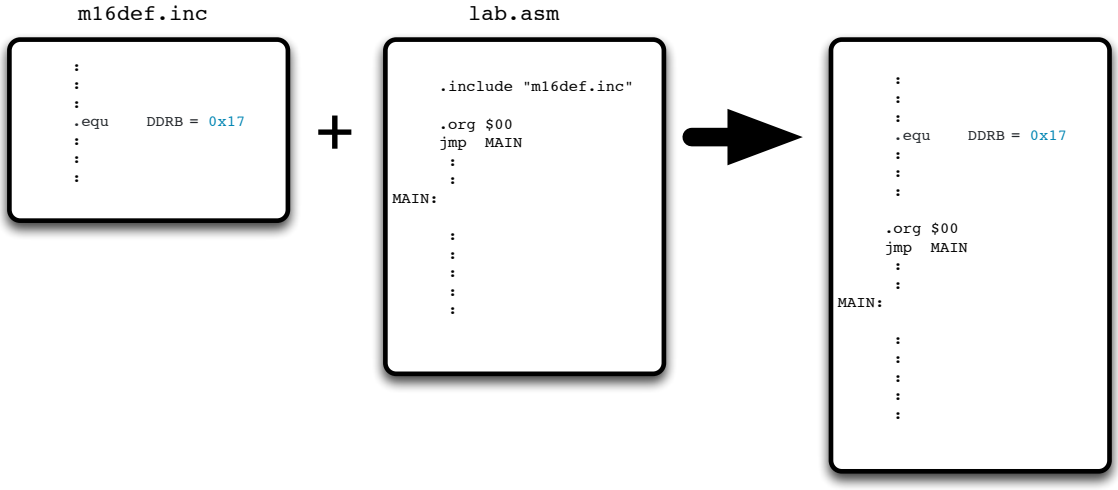
Några vanliga och användbara preprocessordirektiv anges nedan

Direktiv	Namn	Betydelse
`.org`	*origin*	Skriv här (adress)
`.byte`	*byte*	Reservera byte i SRAM
`.dseg`	*data segment*	Följande gäller SRAM
`.cseg`	*code segment*	Följande gäller programminnet
`.eseg`	*extra segment*	Följande gäller EEPROM
`.def`	*define*	Döp register till namn
`.equ`	*equate*	Döp konstant
`.db`	*define byte*	Skriv följande *byte* (8-bit) i minnet
`.dw`	*define word*	Skriv följande *word* (16-bit) i minnet
`.macro`	*macro*	”copy-paste” av följande
`.endmacro`	*endmacro*	...avsluta ett macro
`<<` n	*shift left*	vänsterskift n bitar
`&, |, ^`	*logical AND, OR, XOR*	bitvis OCH, ELLER, XOR
`+, -, *, /`	*arithmetic*	som förväntat
`HIGH, LOW`	*high low*	ger höga resp låga delen av följande uttryck

I den tidigare programmeringsmiljön AVRStudio behövde man själv lägga till `include`-filen i sin kod genom att inleda assemblerfilen med:

```
.include    "m16def.inc"
```

`.include` är ett *direktiv* till preprocessorn att inkludera denna fil på plats som i figuren nedan:



Detta steg sker numera automatiskt och innebär att `include`-filens definitioner alltid går före egna definitioner.

Notera att preprocessordirektiven nedan **inte** är instruktioner.

.org `.org` sätter kompilatorn på en specifik adress i SRAM- eller FLASH-minnet.

.cseg, .db `.cseg` anger explicit att följande kod hör till programminnet. Med `.db` kan sedan tabellvärden definieras i FLASH.

```
        .cseg                       ; default
        .org    $0000
        jmp     START
        ;
        ; avbrottsvektorer
        ;
        .org    INT_VECTORS_SIZE    ; definierad i m16def.inc till 42
TAB:    .db     1, 2, 3, 4          ; Tabellen
START:
        ; Programstart
```

.dseg, .byte För att utföra definitioner i SRAM används först direktivet `.dseg`. Med `.byte` reserveras sedan ett antal bytes där. För att växla tillbaka till FLASH används `.cseg`:

```
        .dseg
        .org    $67 ; adress $67 i SRAM
ARR:                ; ARR=$67, handtag till struct nedan
VAR1:   .byte   7   ; VAR1, adressen till 0-te byten av dessa 7
VAR2:   .byte   2   ; VAR2, adressen till forsta lediga efter VAR1

        .cseg
        ; till programminnet igen
```

Observera att det inte finns någon inbyggd kontroll att de definierade adresserna för `.org` är rimliga. Det är till exempel fullt möjligt att definiera en tabell, `TAB`, med `.db` som skriver över befintlig kod:

```
        .org    $0000
        jmp     START
```

```
        ;
        ; avbrottsvektorer
        ;
        .org    $100
START:
        ; Programstart
        :
        kod
        :
        .org    $100
TAB:    .db     1, 2, 3, 4   ; <--- krasch!!
```

Exempel

Fallgrop. Varför blir även detta fel?

```
TAB:    .org    $200
        .db     1, 2, 3, 4
```

.macro Ett makro definierar ett kodstycke som skall kopieras in i koden. Skilj det alltså från ett subrutinanrop. Följande makro kan användas för att stuva undan `ZH:ZL`-registret:

```
        .macro   PUSHZ
                 push   ZH
                 push   ZL
        .endmacro

        :
        PUSHZ          <-- ersätts med de två raderna
        :
```

Använt som ovan har makrot enbart en kosmetisk effekt på koden. Ett makro kan dock ta argument vilket gör det mer användbart, vi kan tillverka den saknade instruktionen ADDI:

```
        .macro   ADDI    ; macro med argumenten @0 och @1
                 subi    @0,-@1
        .endmacro

        :
        ADDI     r20,3   ; r20 = r20 + 3
                ;@0 @1
        :
```

Övrigt Preprocessorn kan också göra livet enklare för assemblerprogrammeraren med sitt stöd för aritmetiska och logiska operationer. Observera att dessa beräkningar görs som en del av preprocessningen innan kompileringen och långa uttryck resulterar alltså inte i längre kod.

```
ldi     r16,(1<<3)|(1<<ADSC) ; biten ADSC och bit 3 ett-satta i r16
        :
ldi     r16,HIGH(42*31)      ; högsta byten av 42*31=$0516, dvs $05
ldi     r17,LOW(42*31)       ; lägsta byten av samma dvs $16
```

Kompilering

Kompileringssteget är det steg som känner igen assemblerinstruktioner och kan översätta dem till de hexadecimala tal som skall programmeras in i mikrokontrollerns FLASH-minne. Kompileringen sker i två steg, en så kallad *två-pass*-assembler (*two pass assembler*):

1. Först analyseras programkoden med avseende symboliska adresser och konstanter. *Labels*, exempelvis `START:`, ersätts med sina faktiska värden.
2. Med informationen ovan kan det andra steget köras. Detta är det egentliga kodgenererande steget. Här översätts assemblerinstruktioner till hexadecimala tal.

För den extra intresserade belyser vi hela processen i mer detalj med ett exempel.

Programraden

```
START:    ldi   r16,HIGH(RAMEND)
```

omformas under byggsteget till hexadecimala tal:

```
START:    ldi   r16,HIGH($045F) ; Preprocessorn ersätter RAMEND
START:    ldi   r16,$04         ; Preprocessorn använder HIGH på $045f
$0044:    ldi   r16,$04         ; Kompilatorn sätter adresser
$0044:    $E004                 ; Kompilatorn genererar hexadecimala tal
```

Kikar man närmare på det färdiga talet `E004` kan man se att det består av flera olika delar: Alla `ldi`-instruktioner börjar till exempel med `E` och innehåller sedan bitar som motsvarar argumenten, det vill säga det finns bitar som avgör

- vilket register som är inblandat (där `r16` är det nollte) och
- vilken konstant det handlar om (`$04` här).

För att vi inte ska behöva peta i detaljerna sköter kompilatorn om detta åt oss.

Nedan visas ett stycke kod med kompilerad form i en kolumn till höger:

```
        ldi     r16,$04          E004
        out     SPH,r16          BF0E
        out     SPL,r16          BF0D
        out     DDRB,r16         BB07
        clr     XL               27AA
        clr     XH               27BB
        ldi     ZH,HIGH(TEXT*2)  E0F0
        ldi     ZL,LOW(TEXT*2)   E6EE
BLANK:  rcall   TWOBEEP          D016
GETCH:  rcall   TWOBEEP          D015
          :
```

Högerkolumnen placeras sedan vid processorns programmering i FLASH-programminnet:

```
RAD  HEX
0000 E004
0001 BF0E
0002 BF0D
0003 BB07
0004 27AA
0005 27BB
0006 E0F0
0007 E6EE
0018 D016
0019 D015
```

För läslighets skull visas hexkoden skriven som ovan. I det fysiska minnet är hög och låg byte ombytta som vanligt: byte `0000` innehåller `04` och så vidare.

För den extra extra intresserade

Kompilatorns utfil är en så kallad *hexfil* (`.hex`) med ett standardiserat format:

```
:020000020000FC
:1000000004E00EBF0DBF07BBAA27BB27F0E0EEE65A
:1000100016D015D005910030...
```

Raderna som börjar med `:1` innehåller de hexadecimala talen som skall programmeras. Först kommer en adress och sedan 32 bytes med information. Det `5A` som är sist på raden återfinns inte i den hexadecimala koden utan är en checksumma för hela den raden.

—o-Ö-o—

A. ASCII-tabell

Praktiskt taget alla tecken kodas numera i ASCII, *American Standard Code for Information Interchange*, varför en kännedom om denna tabell är nödvändig.

$HL	MSB	0000	0001	0010	0011	0100	0101	0110	0111
LSB	L \ H	0	1	2	3	4	5	6	7
0000	0	NUL	DLE	SP	0	@	P	`	p
0001	1	SOH	DC1	!	1	A	Q	a	q
0010	2	STX	DC2	"	2	B	R	b	r
0011	3	ETX	DC3	#	3	C	S	c	s
0100	4	EOT	DC4	$	4	D	T	d	t
0101	5	ENQ	NAK	%	5	E	U	e	u
0110	6	ACK	SYN	&	6	F	V	f	v
0111	7	BEL	ETB	'	7	G	W	g	w
1000	8	BS	CAN	(	8	H	X	h	x
1001	9	HT	EM	)	9	I	Y	i	y
1010	A	LF	SUB	*	:	J	Z	j	z
1011	B	VT	ESC	+	;	K	[	k	{
1100	C	FF	FS	,	<	L	\	l	|
1101	D	CR	GS	–	=	M	]	m	}
1110	E	SO	RS	.	>	N	^	n	~
1111	F	SI	US	/	?	O	_	o	DEL

ASCII-tabellen är uppbyggd kolumnvis. Kolumnerna 0 och 1 är styrtecken till terminaler, kolumn 2 och 3 är skiljetecken och siffror. Resten av kolumnerna är alfabetet i versaler

och gemener. Ett tecken anges med sin kolumn och rad, tecknet `A` är i kolumn 4 och rad 1 vilket ger värdet `$41`=$0100\,0001_2$ osv.

All text i programmering utförs numera med ASCII-tecken.

En datafil med texten ”American Standard Code for Information Interchange” kan analyseras byte för byte med kommandot `hexdump -C`:

```
Addr     0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F  Text
0000   41 6D 65 72 69 63 61 6D  20 53 74 61 6E 64 61 72  American Standar
0010   64 20 43 6F 64 65 20 66  6D 72 20 49 6E 66 6F 72  d Code for Infor
0020   6D 61 74 69 6F 6E 20 49  6E 74 65 72 63 68 61 6E  mation Interchan
0030   67 65 0A                                         ge.
```

Utskriften ovan inleds med en adresskolumn varefter själva innehållet anges byte för byte. Vi ser att filen börjar `41 6D 65`... vilket motsvarar tecknen `Ame`... och så vidare.

Vanliga styrtecken[1] är

ASCII	betydelse	i C	översättning
NUL	*null*	\0	null
BEL	*bell*	\a	klockpling
BS	*backspace*	\b	radera
HT	*horizontal tab*	\t	tabb-steg
LF	*line feed*	\n	ny rad
FF	*format feed*	\f	ny sida
CR	*carriage return*	\r	vagnretur
SP	*space*		mellanslag

I hexdumpen ovan känner man igen SP, `$20`, mellanslag. En vagnretur i UNIX utförs som LF, `$0A`, men kan också utföras som kombinationen CR+LF, `$0D $0A`

[1] Användning av flera, till exempel `STX` och `ETX` som *Start* respektive *End of Transmission*, är föråldrad.

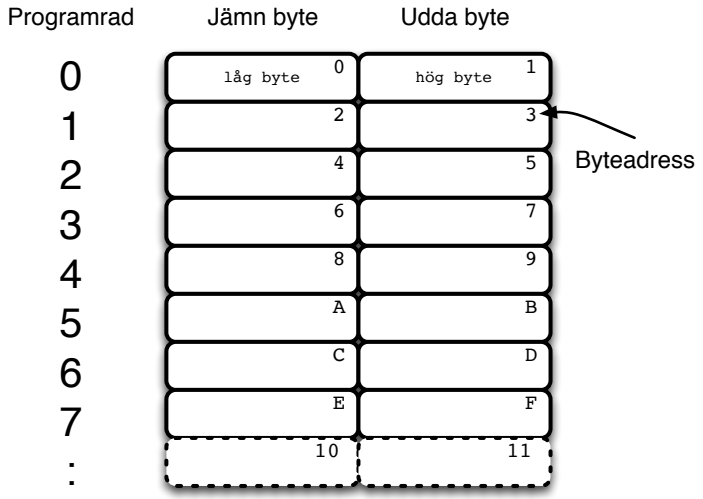
B. Tabeller i FLASH

För att använda tabeller i processorns FLASH-minne måste man skilja på

- programrad, och
- minnes/byte-adress.

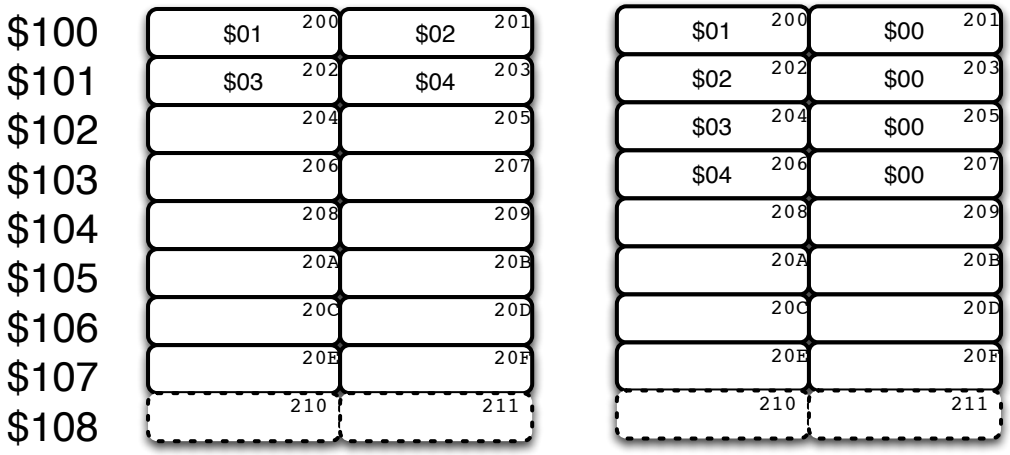
I normalfallet använder vi hela tiden programmets rader när vi skriver program. Det är dessa rader som innehåller programmets instruktioner och det är dessa rader vi hoppar till med `jmp` osv.

En instruktion är 16 bitar bred dvs består av två bytes. Sådana 16-bitars *word* utgör programraderna och en programrad består av en jämn och en udda byte:



Med preprocessordirektiven `.db` och `.dw`, *define byte* respektive *define word*, kan man lägga tabeller i FLASH-minnet (notera att adressen för `TAB` blir `TAB*2`):

```
          .org  $100                  .org  $100
TAB:
          .db $01, $02, $03, $04      .dw $0001, $0002, $0003, $0004
```



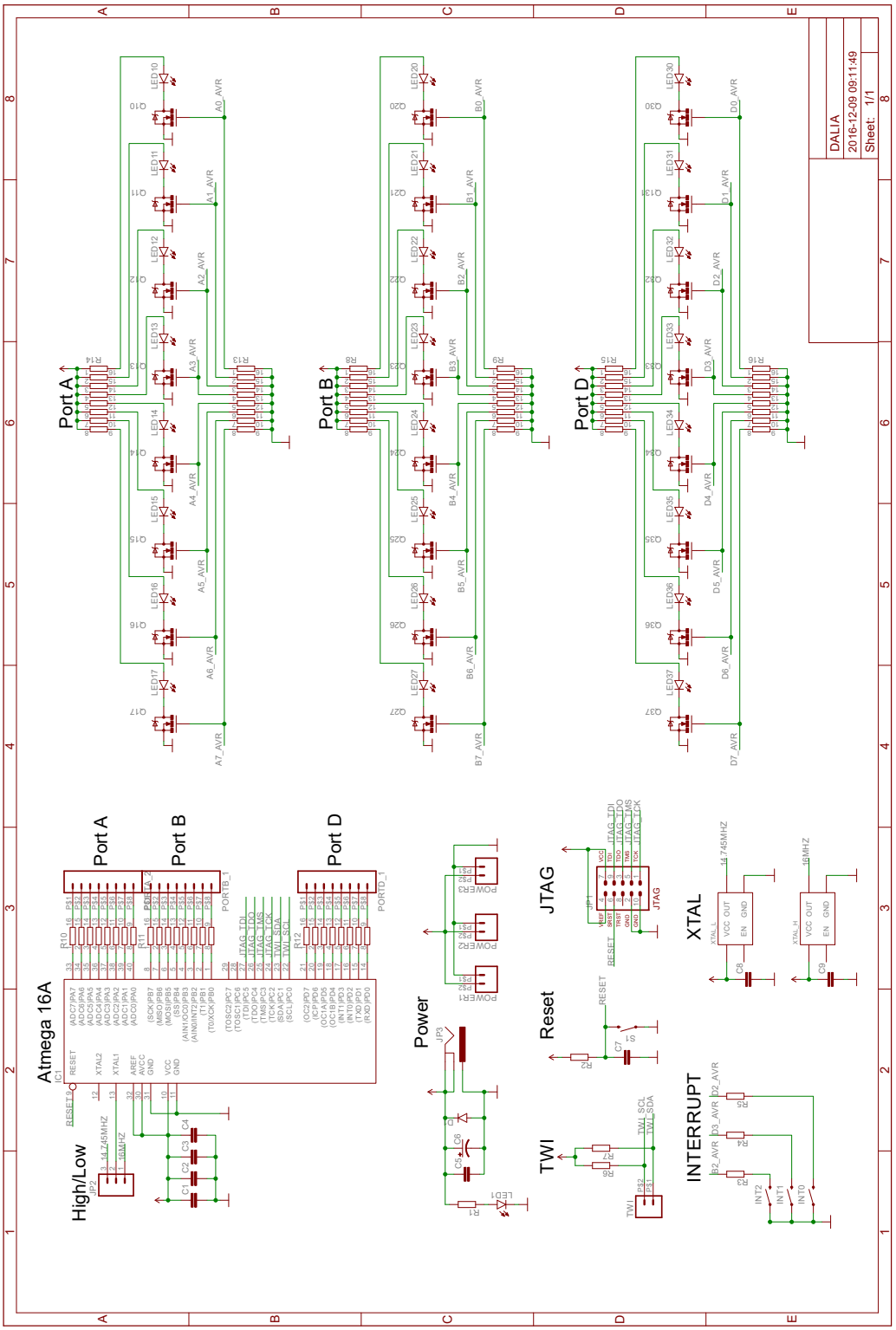
Med `.dw` läggs alltid ett word i minnet och det hamnar då på jämn adress automatiskt. Man ser också att det kan vara oekonomiskt för små tabellvärden, då den högre byten sätts till noll.

Instruktionen `lpm` hämtar den byte som `Z`-registret pekar på[1], dvs inte programraden utan byteadressen. I vänstra figuren ovan kan `lpm` med postinkrement användas för att stega igenom tabellen medan i den högra krävs "`adiw ZL,2`" för att peka ut nästa korrekta word.

Detta förklarar också varför en tabell måste innehålla ett jämnt antal bytes. Ty annars pekar programräknaren inte på hela instruktioner. Kompilatorn ger en varning om detta men lägger nästa instruktion på en jämn adress ändå.

[1] `ldi ZH,HIGH(TAB*2)`
`ldi ZL,LOW(TAB*2)`

C. Kretsschema DALIA-kortet



D. Utdrag ur filen m16def.inc

Här visas ett urval av rader från `m16def.inc` som inkluderas vid varje kompilering.

```
; ***** I/O REGISTER DEFINITIONS ******************************
; NOTE:
; Definitions marked "MEMORY MAPPED"are extended I/O ports
; and cannot be used with IN/OUT instructions
  .equ        SREG   = 0x3f
  .equ        SPH    = 0x3e
  .equ        SPL    = 0x3d
  .equ        OCR0   = 0x3c
  .equ        GICR   = 0x3b
  .equ        GIFR   = 0x3a
  .equ        TIMSK  = 0x39
  .equ        TIFR   = 0x38
  .equ        SPMCSR = 0x37
  .equ        TWCR   = 0x36
  .equ        MCUCR  = 0x35
  .equ        MCUCSR = 0x34
  .equ        TCCR0  = 0x33
  .equ        TCNT0  = 0x32
          :
  .equ        PORTA  = 0x1b
  .equ        DDRA   = 0x1a
  .equ        PINA   = 0x19
     :
  .equ        ACSR   = 0x08
  .equ        ADMUX  = 0x07
  .equ        ADCSRA = 0x06
  .equ        ADCH   = 0x05
  .equ        ADCL   = 0x04

; ***** TIMER_COUNTER_0 **************
; TCCR0 - Timer/Counter Control Register
  .equ        CS00   = 0        ; Clock Select 1
  .equ        CS01   = 1        ; Clock Select 1
  .equ        CS02   = 2        ; Clock Select 2
  .equ        WGM01  = 3        ; Waveform Generation Mode 1
  .equ        CTC0   = WGM01    ; For compatibility
  .equ        COM00  = 4        ; Compare match Output Mode 0
  .equ        COM01  = 5        ; Compare Match Output Mode 1
  .equ        WGM00  = 6        ; Waveform Generation Mode 0
  .equ        PWM0   = WGM00    ; For compatibility
  .equ        FOC0   = 7        ; Force Output Compare

; TIMSK - Timer/Counter Interrupt Mask Register
  .equ        TOIE0  = 0        ; Timer/Counter0 Overflow Interrupt Enable
  .equ        OCIE0  = 1        ; Timer/Counter0 Output Compare Match Interrupt

; TIFR - Timer/Counter Interrupt Flag register
  .equ        TOV0   = 0        ; Timer/Counter0 Overflow Flag
  .equ        OCF0   = 1        ; Output Compare Flag 0

; SFIOR - Special Function IO Register
  .equ        PSR10  = 0        ; Prescaler Timer/Counter1 and Timer/Counter0
```

```
; ***** EXTERNAL_INTERRUPT ***********
; GICR - General Interrupt Control Register
  .equ        GIMSK = GICR      ; For compatibility
  .equ        INT2  = 5         ; External Interrupt Request 2 Enable
  .equ        INT0  = 6         ; External Interrupt Request 0 Enable
  .equ        INT1  = 7         ; External Interrupt Request 1 Enable

; GIFR - General Interrupt Flag Register
  .equ        INTF2  = 5        ; External Interrupt Flag 2
  .equ        INTF0  = 6        ; External Interrupt Flag 0
  .equ        INTF1  = 7        ; External Interrupt Flag 1

; MCUCR - General Interrupt Control Register
  .equ        ISC00  = 0        ; Interrupt Sense Control 0 Bit 0
  .equ        ISC01  = 1        ; Interrupt Sense Control 0 Bit 1
  .equ        ISC10  = 2        ; Interrupt Sense Control 1 Bit 0
  .equ        ISC11  = 3        ; Interrupt Sense Control 1 Bit 1

; MCUCSR - MCU Control And Status Register
  .equ        ISC2   = 6        ; Interrupt Sense Control 2

; ***** AD_CONVERTER *****************
; ADMUX - The ADC multiplexer Selection Register
  .equ        MUX0   = 0        ; Analog Channel and Gain Selection Bits
  .equ        MUX1   = 1        ; Analog Channel and Gain Selection Bits
  .equ        MUX2   = 2        ; Analog Channel and Gain Selection Bits
  .equ        MUX3   = 3        ; Analog Channel and Gain Selection Bits
  .equ        MUX4   = 4        ; Analog Channel and Gain Selection Bits
  .equ        ADLAR  = 5        ; Left Adjust Result
  .equ        REFS0  = 6        ; Reference Selection Bit 0
  .equ        REFS1  = 7        ; Reference Selection Bit 1

; ADCSRA - The ADC Control and Status register
  .equ        ADPS0  = 0        ; ADC  Prescaler Select Bits
  .equ        ADPS1  = 1        ; ADC  Prescaler Select Bits
  .equ        ADPS2  = 2        ; ADC  Prescaler Select Bits
  .equ        ADIE   = 3        ; ADC Interrupt Enable
  .equ        ADIF   = 4        ; ADC Interrupt Flag
  .equ        ADATE  = 5        ;
  .equ        ADFR   = ADATE    ; For compatibility
  .equ        ADSC   = 6        ; ADC Start Conversion
  .equ        ADEN   = 7        ; ADC Enable

; ***** PORTA ***********************
; PORTA - Port A Data Register
  .equ        PORTA0 = 0        ; Port A Data Register bit 0
  .equ        PORTA1 = 1        ; Port A Data Register bit 1
  .equ        PORTA2 = 2        ; Port A Data Register bit 2
  .equ        PORTA3 = 3        ; Port A Data Register bit 3
  .equ        PORTA4 = 4        ; Port A Data Register bit 4
  .equ        PORTA5 = 5        ; Port A Data Register bit 5
  .equ        PORTA6 = 6        ; Port A Data Register bit 6
  .equ        PORTA7 = 7        ; Port A Data Register bit 7

; DDRA - Port  = A Data Direction Register
  .equ        DDA0   = 0        ; Data Direction Register, Port A, bit 0
  .equ        DDA1   = 1        ; Data Direction Register, Port A, bit 1
  .equ        DDA2   = 2        ; Data Direction Register, Port A, bit 2
  .equ        DDA3   = 3        ; Data Direction Register, Port A, bit 3
  .equ        DDA4   = 4        ; Data Direction Register, Port A, bit 4
  .equ        DDA5   = 5        ; Data Direction Register, Port A, bit 5
  .equ        DDA6   = 6        ; Data Direction Register, Port A, bit 6
  .equ        DDA7   = 7        ; Data Direction Register, Port A, bit 7
```

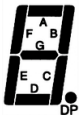
```
; PINA - Port A Input Pins
  .equ        PINA0     = 0        ; Input Pins, Port A bit 0
  .equ        PINA1     = 1        ; Input Pins, Port A bit 1
  .equ        PINA2     = 2        ; Input Pins, Port A bit 2
  .equ        PINA3     = 3        ; Input Pins, Port A bit 3
  .equ        PINA4     = 4        ; Input Pins, Port A bit 4
  .equ        PINA5     = 5        ; Input Pins, Port A bit 5
  .equ        PINA6     = 6        ; Input Pins, Port A bit 6
  .equ        PINA7     = 7        ; Input Pins, Port A bit 7

; ***** CPU REGISTER DEFINITIONS ****************************
  .def        XH        = r27
  .def        XL        = r26
  .def        YH        = r29
  .def        YL        = r28
  .def        ZH        = r31
  .def        ZL        = r30

; ***** DATA MEMORY DECLARATIONS ****************************
  .equ        FLASHEND   = 0x1fff        ; Note: Word address
  .equ        IOEND      = 0x003f
  .equ        SRAM_START = 0x0060
  .equ        SRAM_SIZE  = 1024
  .equ        RAMEND     = 0x045f
  .equ        XRAMEND    = 0x0000
  .equ        E2END      = 0x01ff
  .equ        EEPROMEND  = 0x01ff
  .equ        EEADRBITS  = 9

; ***** INTERRUPT VECTORS ***********************************
  .equ        INT0addr   = 0x0002        ; External Interrupt Request 0
  .equ        INT1addr   = 0x0004        ; External Interrupt Request 1
  .equ        OC2addr    = 0x0006        ; Timer/Counter2 Compare Match
  .equ        OVF2addr   = 0x0008        ; Timer/Counter2 Overflow
  .equ        ICP1addr   = 0x000a        ; Timer/Counter1 Capture Event
  .equ        OC1Aaddr   = 0x000c        ; Timer/Counter1 Compare Match A
  .equ        OC1Baddr   = 0x000e        ; Timer/Counter1 Compare Match B
  .equ        OVF1addr   = 0x0010        ; Timer/Counter1 Overflow
  .equ        OVF0addr   = 0x0012        ; Timer/Counter0 Overflow
  .equ        SPIaddr    = 0x0014        ; Serial Transfer Complete
  .equ        URXCaddr   = 0x0016        ; USART, Rx Complete
  .equ        UDREaddr   = 0x0018        ; USART Data Register Empty
  .equ        UTXCaddr   = 0x001a        ; USART, Tx Complete
  .equ        ADCCaddr   = 0x001c        ; ADC Conversion Complete
  .equ        ERDYaddr   = 0x001e        ; EEPROM Ready
  .equ        ACIaddr    = 0x0020        ; Analog Comparator
  .equ        TWIaddr    = 0x0022        ; 2-wire Serial Interface
  .equ        INT2addr   = 0x0024        ; External Interrupt Request 2
  .equ        OC0addr    = 0x0026        ; Timer/Counter0 Compare Match
  .equ        SPMRaddr   = 0x0028        ; Store Program Memory Ready

  .equ        INT_VECTORS_SIZE = 42      ; size in words
```

E. Miniprojekt

För den som tidigare inte programmerat assembler brukar den första kontakten med programmeringsspråket bli lätt tumultartad, det är många begrepp och tekniker som måste passa ihop på en och samma gång för att lösa uppgiften.

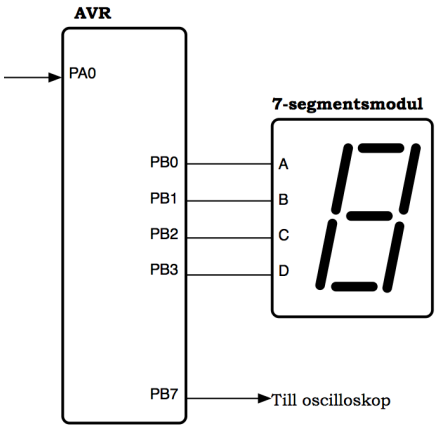
För att underlätta för läsaren kommer här ett *miniprojekt*. Miniprojektet är helt genomarbetat för att visa på en möjlig lösningsgång av uppgiften. Det finns säkert andra lösningsgångar som kan vara lika bra eller bättre.

Uppgift Konstruera ett program som räknar upp siffrorna 0–9 på en sjusegments LED-display så länge en tryckknapp hålls nedtryckt. När knappen släpps upp skall räknaren stanna.



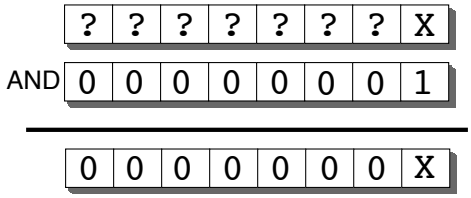
Hårdvara För att kunna genomföra alla detaljer i programmet måste vi ha inblick i hur programmet interagerar med yttervärlden. All kommunikation sker via någon av processorns 8-bitars portar som används för in- respektive utsignaler. Det handlar om läsning respektive skrivning, sett ur programmets synvinkel. Med en läsning får man normal alltid en hel byte information. Vid skrivning sker det omvända men det är fortfarande alltid en hel byte som skrivs. För enkelhets skull använder vi hela port `PINA` för läsning och hela port `PORTB` för skrivning.

Från `PINA` ska vi läsa in tryckknappsvärdet på en enstaka bit. Till `PORTB` ska vi skriva det fyrabitars värde som motsvarar den siffra vi vill se på displayen. För att det ska fungera måste displayen ha en *7-segmentsavkodare* ansluten. Denna komponent översätter ett fyrabitars värde och skickar ut motsvarande 7-bitar som styr varje enskild lysdiod i displayen.



(Figuren ovan är tagen ur lab1 och innehåller dessutom skvallersignalen, från PB7, till oscilloskop. Den är ointressant nu.)

Vid läsning läser man normalt en hel byte. I vårt fall är en tryckknapp ansluten till en av pinnarna på PINA. Vi förutsätter att det är på minst signifikant bit, bit 0. Vid läsning av pinnar läser vi alltså alla bitar även om det bara är en bit som intresserar oss. Vi kan inte lita på värdet hos övriga pinnar! Det är lätt att anta att en icke ansluten pinne kommer att läsas som binär nolla, men det finns inget som garanterar det. Alltså måste vi läsa in hela porten och sedan *själva skilja ut*, maska, den intressanta biten. Och frasen ”skilja ut” gör att vi lystrar och förstår att det måste involvera den logiska instruktionen andi, som påverkar statusflaggorna:




```
TSTBIT0:
     in      r16,PINA  ; ta in hela porten
     andi    r16,$01   ; maska ut bit0
     breq    TSTBIT0   ; omigen
     :                 ; hit om bit == 1
```

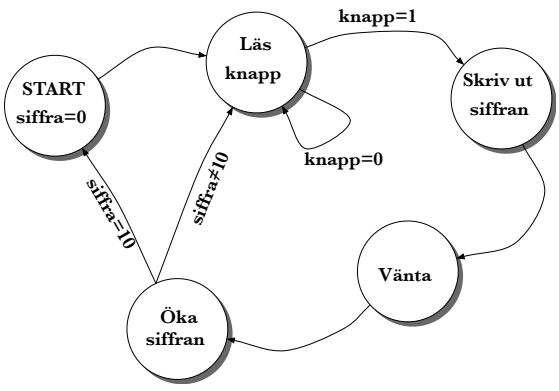
Alternativt kan man göra en avkänning direkt mot pinnen med en skip-instruktion. Det är något kompaktare kodmässigt då man slipper läsa hela byten och maska ut det väsentliga:

```
TSTBIT0:
     sbis    PINA,0    ; testa bit0 enbart
     jmp     TSTBIT0   ; omigen
     :                 ; hit om bit == 1
```

För skrivning till PORTB skall man se till att inte skriva till oönskade bitar (om man inte kan acceptera att skrivning sker dit, till exempel genom att portriktningen gör skrivning ofarlig). Som ovan kan man nollställa bitar med andi-instruktionen om det skulle behövas. Med ori-instruktionen kan man på motsvarande sätt ett-ställa de bitar man önskar.

Handlar det om enstaka bitar i I/O-register som ska ett- eller nollställas kan instruktionerna sbi och cbi användas. I vårt fall är de olämpliga eftersom vi ska ett-ställa alla fyra lägsta bitarna.

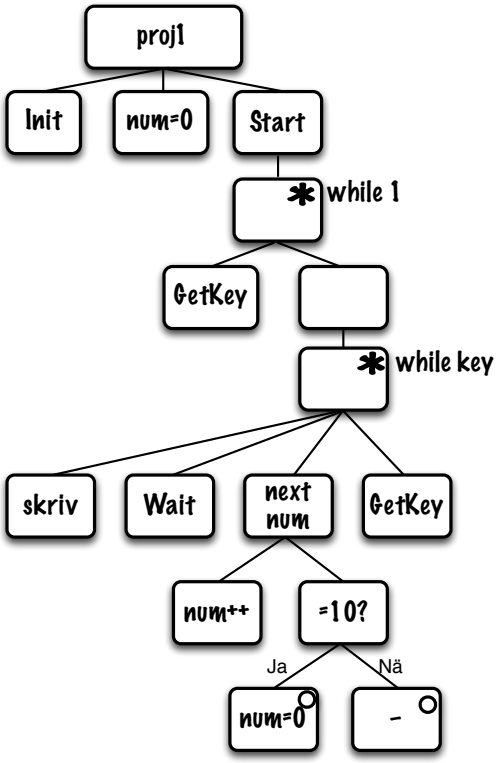
Programmet som tillståndsgraf För den här uppgiften kan man visa programflödet i en tillståndsgraf för att sortera tankarna och ta reda på exakt *vad* som egentligen ska göras. Tillståndsgrafen är en teknik vi lånar från digitaltekniken varför den inte presenteras närmare här utan vi går direkt på en lösning:



Programmet som strukturdiagram Med förberedelserna ovan är vi klara att gripa oss an själva programmeringen. I detta fall är koden inte så omfattande att man med lite erfarenhet skulle kunna följa flödet ovan utan större risk.

För övnings skull föreslås här ett möjligt strukturdiagram. Det är fullt rimligt att diagrammet måste skrivas om några gånger innan man är nöjd med resultatet.

När det gäller strukturdiagrammet måste man göra en avvägning av vilka rutor som skall vara egna subrutiner och vilka man kan skriva ner rakt av. För många subrutiner kan ge plottrig och svårläst kod. Erfarenheten får avgöra från fall till fall. Det är inget självändamål att ha så många subrutiner som möjligt bara för att det är strukturerad programmering. Med strukturen given enligt diagrammet nedan kan man göra uppdelning i subrutiner senare och ändå få ett korrekt program.



Ett förslag på lösning med strukturdiagram. Kan du konstruera alternativa lösningar?

En kod som realiserar strukturdiagrammet kan vara:

```
     ; r16-r19 free to use
     .def     num = r20        ; number 0-9
     .def     key = r21        ; key pressed yes/no

     ldi      r16,HIGH(RAMEND)  ; set stack
     out      SPH,r16           ; for calls
     ldi      r16,LOW(RAMEND)
     out      SPL,r16
     call     INIT
     clr      num
FOREVER:
     call     GET_KEY      ; get keypress in boolean 'key'
LOOP:
     cpi      key,0
     breq     FOREVER      ; until key
     out      PORTB,num    ; print digit
     call     DELAY
     inc      num          ; num++
```

```
     cpi     num,10       ; num==10?
     brne    NOT_10       ; no, so jump
     clr     num          ; was 10
NOT_10:
     call    GET_KEY
     jmp     LOOP
```

Här definieras först registren `r20` och `r21` att heta `num` respektive `key` i hela programmet. `num` innehåller en siffra $0-9$ i sina lägsta bitar och `key` är en boolesk variabel som är $\neq 0$ om knapp nedtryckt och 0 för övrigt.

Programmet anropar rutiner som inte behöver vara färdiga förrän vid ett senare tillfälle. Vi har `INIT` som konfigurerar in- och utgångar, `GET_KEY` som känner av knappen och returnerar `$FF` om nedtryckt och `$00` om inte och `DELAY` som väntar en dryg halvsekund. Finare uppdelning har inte genomförts.

För `GET_KEY` återfinns returvärdet i registret `key`:

```
     ;
     ; --- GET_KEY. Returns key != 0 if key pressed
GET_KEY:
     clr     key
     sbic    PINA,0       ; skip over if not pressed
     dec     key          ; key=$FF
     ret
```

Hårdvaruinitieringen sker med koden nedan.[1]

```
     ;
     ; --- Init. A0 in, B3-B0 out
INIT:
     clr     r16
     out     DDRA,r16
     ldi     r16,$0F
     out     DDRB,r16
     ret
```

Om man skulle vilja aktivera *weak pull-up* på ingångarna kan man dessutom lägga till dessa två rader i `INIT`:

```
     ldi     r16,$FF
     out     PORTA,r16    ; Weak pullup
```

Slutligen behövs en rätt rejäl vänterutin, `DELAY`, som består av tre nästlade vänteloopar. Den inre loopen från `D_2:` till `brne D_1` räknar `r16` $= 0, 255, 254, 253, \ldots 1, 0$ det vill säga 256 varv. Den något yttre loopen med `r17` ser till att detta sker 256 gånger och liknande för `r18`. Totalt tar `DELAY` 592 137 klockcykler enligt simulatorn, vilket motsvarar knappt 0.6 sekunder vid en klockfrekvens på 1 MHz:

```
     ;
     ; --- DELAY. Wait a lot!
DELAY:
     ldi     r18,3
D_3:
     ldi     r17,0
D_2:
     ldi     r16,0
D_1:
     dec     r16
     brne    D_1
     dec     r17
     brne    D_2
     dec     r18
     brne    D_3
     ret
```

Kod utan subrutiner I koden ovanför har subrutiner använts för att få tydlig struktur på hela uppgiften. Det är så man normalt gör. Om vi av någon anledning inte vill använda subrutiner kan så klart uppgiften lösas ändå:

[1] Om du känner till returstacken: Varför kan man du inte initiera stacken i en denna subrutin?

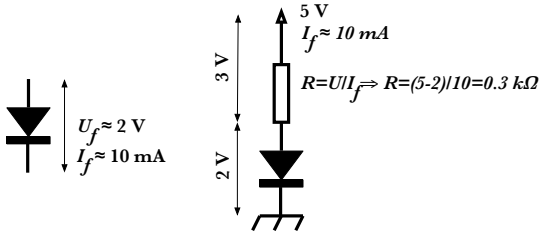
```
     ; r16-r19 free to use
     .def     num = r20   ; number 0-9
     .def     key = r21   ; key pressed yes/no

     clr      r16
     out      DDRA,r16
     ldi      r16,$0F
     out      DDRB,r16
     clr      num
FOREVER:
     clr      key         ; get keypress in boolean 'key'
     sbic     PINA,0      ; skip over if not pressed
     dec      key         ; key=FF
LOOP:
     cpi      key,0
     breq     FOREVER     ; until key
     out      PORTB,num   ; print digit
     ldi      r18,3
D_3:
     ldi      r17,0
D_2:
     ldi      r16,0
D_1:
     dec      r16
     brne     D_1
     dec      r17
     brne     D_2
     dec      r18
     brne     D_3
     inc      num         ; num++
     cpi      num,10      ; num==10?
     brne     NOT_10      ; no, so jump
     clr      num         ; was 10
NOT_10:
     clr      key         ; get keypress in boolean 'key'
     sbic     PINA,0      ; skip over if not pressed
     dec      key         ; key=FF
     jmp      LOOP
```

Man ser att subrutinerna återfinns inklippta i koden utan sina respektive `ret`. Initieringen av stacken har också eliminerats. Med lite träning kan man även göra det omvända: gå från koden ovan till en med väl valda subrutiner!

F. Utanför labbet

I labmiljön behöver du bara koppla sladdar mellan de olika ingående modulbrickorna. I verkligheten, eller i en senare projektkurs till exempel, kan du behöva koppla in lysdioder och tryckknappar själv och måste göra det på ett sådant sätt att komponenterna inte förstörs. Här beskrivs hur det går till.

Hur tänder man en lysdiod? För att en lysdiod ska tända och lysa krävs en ström genom den. Allmänt kan kan anta att en ström om 10 mA är lagom. I själva verket *kan* den tända vid lägre ström. Mer än cirka 20 mA bör man inte driva genom lysdioden om den inte är av speciell högströmstyp. När lysdioden lyser blir spänningen över den cirka 2 V. Eftersom vår logiska etta är 5 V kan vi inte ansluta denna direkt till lysdioden — inte i för många millisekunder i varje fall, ty dioden blir fort varm och brinner upp. För att hindra för hög ström genom dioden måste ett *strömbegränsningsmotstånd* anslutas i serie med dioden. Beräkning av lämpligt motståndsvärde sker enligt:

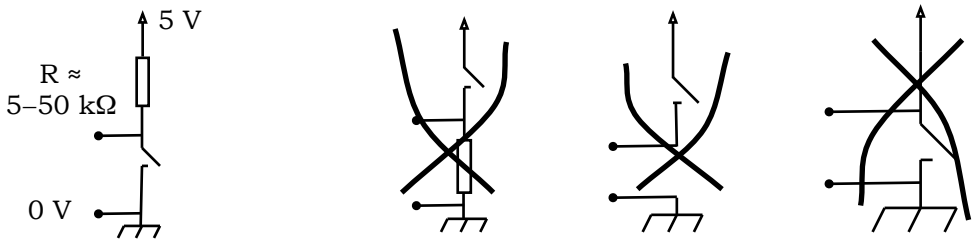


För att vara säker på att strömmen inte överstiger den dimensonerade väljer vi i praktiken närmast högre motståndsvärde hellre än ett lägre. I detta fall alltså standardvärdet $330\,\Omega$. Lysdiodens exakta spänningsfall, V_f, beror på vilken färg den lyser i, 2 V duger bra som riktvärde för röda lysdioder. Moderna lysdioder tänder dessutom vid lägre ström än detta varför även $390\,\Omega$ eller till och med $1k\,\Omega$ antagligen skulle fungerat i detta fall. Samtliga dessa tre är tillgängliga standardvärden.

Hur kopplar man in en tryckknapp? Med tryckknappen vill vi på ett säkert sätt överföra informationen ”nedtryckt” respektive ”icke nedtryckt” till exempelvis `PINA`.

Av flera skäl[1] är det lämpligt att låta signalen vara logisk etta när knappen *inte* är nedtryckt, och logisk noll då knappen är nedtryckt. Detta kan upplevas som omvänt till en början men omhändertas lätt i programkoden.

Inkopplingen blir då enligt enligt den vänstra figuren. De andra är helt förkastliga:



[1] Man måste till exempel ha en ”säker” 0 V, varför man inte kan ansluta motståndet mellan knappen och 0 V.

Motståndet måste vara tillräckligt litet för att försörja ingången med tillräcklig ström men samtidigt stort nog att inte orsaka onödig strömförbrukning då brytaren är i sitt tillslagna läge. Ett motstånd i storleken 5–50 kohm är rimligt — oftast brukar 10 kohm användas.

Många oroas här över frånvaron av en avstudsning, som var så viktig i digitaltekniken. Att den inte behövs här kan man dock lätt inse om man tänker sig att inga pulser uppstår utan tryck på knappen. Då spelar det uppenbart inte någon roll om vi registrerar studsarna eller ”huvudpulsen”. Oavsett vilket lägger programmet märke till att knappen blivit nedtryckt. Till skillnad från i digitaltekniken väljer vi här i programmet själva när avkänning av tryckknappen ska ske.[2]

[2]Om tryckknappen skall påverka avbrottsingångarna måste den dock vara avstudsad. Mer om det senare i kursen.

G. Kodexempel

Med bra namn på variabler och funktioner kan koden bli så lättläst att kommentarer ofta är överflödiga. Med en bra programstruktur underlättas både samarbete kring en programmeringsuppgift och felsökning samtidigt som man kan vara rätt säker på att programmet faktiskt löser den uppgift man vill.

Inte sällan är man så inne i programmeringen att man inte lägger särskilt stor vikt vid variabel- och funktionsnamn. I vissa fall följer koden en matematisk beskrivning och då kan det matematiska språket skina igenom till exempel med korta indexvariabler `i,j` för $x_{i,j}$.

Här kommer ett exempel i C++[1]. En klass för en *sist-in-först-ut*-stack för ASCII-tecken kan skrivas:

```
#include <iostream>
using namespace std;

class Stack {
public:
    void emptyStack() {
        top = EMPTY;
    };
    void pushToStack(char c) {
        s[++top] = c;
    };
    char popFromStack(){
        return s[top--];
    };
    bool isEmptyStack() const {
        return (top == EMPTY);
    };
    bool isFullStack() const {
        return (top == FULL);
    };
private:
    enum {
        MAX_LEN = 100,
        EMPTY   = -1,
        FULL    = MAX_LEN - 1
    };
    char s[MAX_LEN];
    int top;
};
```

Redan klassens namn `Stack` avslöjar att det handlar om en stack (duh!). I detta fall är det en stack avsedd för `char`s så man kan överväga om namnet `char_Stack` skulle vara lämpligare? Här används bara klassen till en sak så det extra förtydligandet känns onödigt.[2]

De ingående metoderna är namngivna för tydlighet som

- `emptyStack`
- `pushToStack`
- `popFromStack`
- `isEmptyStack`
- `isFullStack`

[1] Kod inspirerad från *C++ by Dissection* (Ira Pohl), sid 187.
[2] Tycker *jag* i alla fall.

Metodnamnen beskriver exakt vad de gör och varje metod gör bara en sak. Att metoden `isEmptyStack` returnerar ett logiskt värde som svar på frågan om stacken är tom är också tydligt.

Efter en tid blev det dock tjatigt att alla metoder heter ”någonting`Stack`”, hela klassen handlar ju om en stack så denna namngivning känns pratig och övertydlig. Efter lite övervägande reducerades namnen till

- `empty`
- `push`
- `pop`
- `isEmpty`
- `isFull`

som är kortare och otvetydiga då de är standardmanipulationer för en stack.

Klassen ska användas för att skriva ut en textsträng baklänges. Ett program som löser denna uppgift är:

```
int main(){
    Stack s;
    char string[40] = { "Hej hopp i lingonskogen" };
    int  i = 0;

    cout << string << endl;
    s.empty();
    while (string[i] && !s.isFullStack())
        s.push(string[i++]);
    while (!s.isEmpty())
        cout << s.pop();
    cout << endl;
};
```

En dissektion av programmet ger dess funktion stegvis:

- Skriv ut strängen med en avslutande radbrytning

  ```
  cout << string << endl;
  ```

- Töm stacken

  ```
  s.empty();
  ```

- Pusha strängen på stacken

  ```
  while (string[i] && !s.isFullStack())
          s.push(string[i++]);
  ```

- Skriv ut stackinnehållet tecken för tecken

  ```
  while (!s.isEmpty())
          cout << s.pop();
  ```

- Avsluta med en radbrytning

  ```
      cout << endl;
  ```

En körning av programmet ger alltså strängen följt av samma sträng skriven bakänges:

```
Hej hopp i lingonskogen
negoksnognil i ppoh jeH
```

Så långt allt väl. Programmet fungerar. Men lite strukturerad programmering i ryggen ser man att programmet, som det står, dock är rätt fult: Det blandar saker på olika nivåer. På den övre nivån instansieras `s` (`Stack s`), strängen definieras och skrivs. På nästa nivå initieras sedan stacken och med ett index överförs strängen till stacken så länge strängen inte är slut och stacken inte är full. På denna nivå skrivs sedan strängen ut från stacken. Avslutningsvis är programmet tillbaka på övre nivån — för att skriva ut en radbrytning.

Som vanligt vill vi undvika detaljer[3] varför vi kan formulera om koden. Programmets

[3]Här `string[i] && !s.isFull()` och `s.push(string[i++])` till exempel

uppgift är att skriva ut en sträng baklänges, låt då det framgå på den översta nivån, det vill säga skriv om `main()` till:

```
    :
    cout << string << endl;
    printStringReversed(string);
    cout << endl;
    :
```

Faktoriseringen gav den nya funktionen:

```
void printStringReversed(char string[]){
    Stack s;

    s.empty();
    int  i = 0;
    while (string[i] && !s.isFull())
        s.push(string[i++]);
    while (!s.isEmpty())
        cout << s.pop();
}
```

som har fördelarna

- funktionen gör vad den heter,
- variabeln `i` är nu lokal och
- detaljerna är borta från `main()`.

På liknande sätt kan man faktorisera ut även den första utskriften, kanske som funktionen `printString()`:

```
void printString(char string[]){
    cout << string << endl;
}
```

Återstår till slut den obekväma `cout « endl;`-instruktionen som inte hittar någon naturlig plats i tillvaron.

Slutversion Sammantaget kan nu `main()` skrivas med val av funktionsnamn så att den är självförklarande och befriad från störande klutter:

```
int main(){
    char string[40] = { "Hej hopp i lingonskogen" };

    printString(string);
    printStringReversed(string);
    cout << endl;
}
```

Fortfarande är sista raden `cout « endl;` störande. Man kan använda `printString(newLine)` till denna men då med en ytterligare deklaration `char newLine[2] = "\n";` som ger fler rader i källkoden. Så en ”sista” slutversion blir:

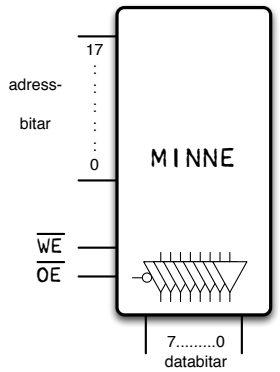
```
int main(){
    char newLine[2] = "\n";
    char string[40] = { "Hej hopp i lingonskogen" };

    printString(string);
    printStringReversed(string);
    printString(newLine);
}
```

H. Minnesadressering

Detta avsnitt kan du använda som en övningsuppgift och skriva assemblerrutiner själv innan du går vidare och kikar på lösningsförslagen. Förutom själva uppgiften finns förslag på moment du bör göra för att sätta dig in i hela problemet. Naturligtvis använder du simulatorn flitigt för att ofta konstatera att koden gör det du tänkt.

I ett system behöver man läsa och skriva till ett byte-brett externt minne. Det externa minnet adresseras med en 24-bitars adress. Tanken är att lägga ut adressen mot minnet och sedan genomföra en läsning eller skrivning (8 bitar). Minnet skall alltid börja genomlöpas på rad `$000000` och eventuellt fortsätta till maximalt `$03ffff`.

Din uppgift är att skriva koden för läs- och skrivrutinerna. För detta måste man först gå till databladen för att se den erforderliga signaleringen till minnet. Minnen har i allmänhet funktion enligt nedan:



Notera att $\overline{OE}$ är aktivt låg, dvs minnet försätts i ut-läge när $\overline{OE}$ är låg. Signalen styr direkt minnets ut-buffrar.

Även *write enable*, $\overline{WE}$ är aktivt låg.

Läsning kan antas gå till så att adressen läggs ut till minnet, varefter signalen *output enable*, $\overline{OE}$ aktiveras och inläsning sker. Vid läsning är $\overline{WE}$ är inaktiv (hög).

Skrivning sker på motsvarande sätt fast med $\overline{OE}$ avaktiverad och $\overline{WE}$ aktiverad. Efter att data lagts ut mot minnet sker inläsning till minnet genom att $\overline{WE}$ aktiveras.

Vid skrivning till minnet får inte minnets ut-buffrar vara aktiverade på grund av den busskollision som då uppträder.

Moment: *1) Rita upp tidsdiagram med alla signaler för läsning och skrivning. 2) Jämför med ett verkligt minne (till exempel SRAM 6264).*

Programmeringen delas som vanligt upp i funktionella subrutiner.

1. Man vill använda register `r25:r24:r23` för adressen. Konstruera en uppräknare `INC_ADDR` för dessa register.
2. Man måste hindra adressräknaren från att gå högre än `$03ffff`. Skriv en subrutin `IS_MAX_ADDR` som testar om så är fallet.
3. Antag att registren `r25:r24:r23` motsvaras av portarna `D:C:B` och att port `D`:s bitar 2 och 3 är $\overline{OE}$ respektive $\overline{WE}$. Skriv subrutinerna `MEM_WR` för skrivning och `MEM_RD` för läsning från minnet.

   Moment: *Rita ett schema över hårdvaran där du anger portar och signaler till/från minnet. Det underlättar att ha en visuell bild av situationen innan vidare programmering.*
4. Vid ett senare tillfälle behövs en samtidig ytterligare adress, `eko`, med en viss konstant förskjutning bakåt från den tidigare. Hantera detta faktum.

5. Genomför de nödvändiga följdförändringarna i `MEM_WR` och `MEM_RD`.
6. Studera din kod. Är allt på rätt plats? Förekommer repetitioner? Var inte nöjd förrän koden är snygg!

1 En 24-bitarsräknare kan utföras som

```
INC_ADDR:
    subi    r23,-1  ; +1
    brcc    XINC
    subi    r24,-1
    brcc    XINC
    subi    r25,-1
XINC:
    ret
```

2 Rutin för att identifiera högsta minnesadressen (`$03ffff`)

```
    ; --- if $03ffff Z=1 at exit
IS_MAX_ADDR:
    cpi     r23,$FF
    brne    XIS_MAX_ADDR
    cpi     r24,$FF
    brne    XIS_MAX_ADDR
    cpi     r25,$03
XIS_MAX_ADDR:
    ret
```

Med rutinerna ovan är en vanlig strategi att **först kontrollera** om uppräkning är tillåten och **sedan utföra** uppräkningen, dvs en rutin som denna passar in:

```
NEXT_ADDR:
    call    IS_MAX_ADDR
    breq    XNEXT
    call    INC_ADDR    ; else ++
XNEXT:
    ret
```

3 Bortsett från portinitiering får vi till exempel (om den satta portriktningen för *data* alltid är *inåt*):

```
    ; --- Adress in r25:r24:r23,
    ; --- data in r16
MEM_RD:
    call    MAX_ADDR    ; max?
    breq    MEM_RD1
    call    INC_ADDR    ; else ++
MEM_RD1:
    cbi     PORTD,2     ; oe low
    call    SET_ADDR    ; adress
    in      r16,PINA
    sbi     PORTD,2     ; oe high
    ret
```

```
    ; --- Adress in r25:r24:r23,
    ; --- data out r16
MEM_WR:
    call    IS_MAX_ADDR
    breq    MEM_WR1
    call    INC_ADDR    ; else ++
MEM_WR1:
    cbi     PORTD,3     ; wr low
    call    SET_ADDR    ; adress
    ser     r17         ; A output
    out     DDRA,r17
    out     PORTA,r16
    sbi     PORTD,3     ; wr high
    clr     r16         ; A input
    out     DDRA,r16
    ret
```

Med koden sida vid sida syns uppenbara koduppprepningar av avsnittet mellan `MEM_RD` och `MEM_RD1` och samma i `MEM_WR`. Rutinen `NEXT_ADDR` används för renare kod:

```
    ; --- Adress in r25:r24:r23,
    ; --- data in r16
MEM_RD:
    call    NEXT_ADDR
    cbi     PORTD,2     ; oe low
    call    SET_ADDR    ; adress
    in      r16,PINA
    sbi     PORTD,2     ; oe high
    ret
```

```
    ; --- Adress in r25:r24:r23,
    ; --- data out r16
MEM_WR:
    call    NEXT_ADDR
    cbi     PORTD,3     ; wr low
    call    SET_ADDR    ; adress
    ser     r17         ; A output
    out     DDRA,r17
    out     PORTA,r16
    sbi     PORTD,3     ; wr high
    clr     r16         ; A input
    out     DDRA,r16
    ret
```

.

Notera hur NEXT_ADDR alltid levererar en giltig nästa adress, antingen max-värdet nåtts eller ej. MEM-rutinerna behöver inte känna till några detaljer utan enbart förlita sig på att nästa adress tas fram. På detta sätt har MEM-rutinerna fått en enklare uppgift och ett mindre ansvarsområde samtidigt som NEXT_ADDR gör det den heter.

4 Det är fel att bara skriva om de tidigare rutinerna att använda andra register, till exempel r22:r21:r20. Ändra istället INC_ADDR och IS_MAX_ADDR så att de använder pekare. Här pekare X-pekaren på minst signifikant byte i 3-bytesadressen.

Moment: *Rita en skiss över minnessituationen. Hur används pekaren?*

```
    ; --- usage
    ; ---   ldi     XL,regnr
    ; ---   clr     XH
    ; ---   call    INC_ADDR

INC_ADDR:
    lds     r16,X
    subi    r16,-1
    sts     X+
    brcc    XINC
    lds     r16,X
    subi    r16,-1
    sts     X+
    brcc    XINC
    lds     r16,X
    subi    r16,-1
    sts     X
XINC:
    subiw   X,2 ; restore pointer
    ret
```

```
    ; --- usage
    ; ---   ldi     XL,regnr
    ; ---   clr     XH
    ; ---   call    IS_MAX_ADDR

IS_MAX_ADDR:
    lds     r16,X+
    cpi     r16,$FF
    brne    XIS_MAX_ADDR
    lds     r16,X+
    cpi     r16,$FF
    brne    XIS_MAX_ADDR
    lds     r16,X
    cpi     r16,$03
XIS_MAX_ADDR:
    subiw   X,2 ; restore pointer
    ret
```

subiw har använts för att återställa pekaren till värdet den hade vid rutinens ingång.

Det blir generellare kod på bekostnad av något fler programrader. Man måste fundera på när det ”kostar mer än det smakar” i form av programrader. Utan eko skulle koden inte behövt modifieras. När börjar det löna sig att göra om rutinerna ovan till loopar?

5 Genomför de nödvändiga ändringarna i MEM_WR och MEM_RD. Koden är i huvudsak som förut med skillnaden att den aktuella adressen anges med en pekare istället för att ligga i r25:r24:r23.

```
    ; --- inc address if possible
NEXT_ADDR:
    call    IS_MAX_ADDR ; max?
    breq    XNEXT
    call    INC_ADDR    ; else ++
XNEXT:
    ret

    ; --- Adress in r25:r24:r23,
    ; --- data in r16
MEM_RD:
    ldi     XL,23
    ldi     XH,0
    call    NEXT_ADDR
    cbi     PORTD,2     ; oe low
    call    SET_ADDR    ; adress
    in      r16,PINA
    sbi     PORTD,2     ; oe high
    ret
```

```
    ; --- Adress in r25:r24:r23,
    ; --- data out r16
MEM_WR:
    ldi     XL,23
    ldi     XH,0
    call    NEXT_ADDR
    cbi     PORTD,3     ; wr low
    call    SET_ADDR    ; adress
    ser     r17         ; A output
    out     DDRA,r17
    out     PORTA,r16
    sbi     PORTD,3     ; wr high
    clr     r16         ; A input
    out     DDRA,r16
    ret
```

```
    ; --- write address to port
SET_ADDR:                                   out     PORTC,X
    ld      r16,X+                          ld      r16,X
    out     PORTB,X                         out     PORTD,X
    ld      r16,X+                          ret
```

6 Det är tämligen aggressivt att låsa adressen till registren `r25:r24:r23` under hela programmets livslängd om det inte finns **väldigt** starka skäl att göra det. Den naturliga placeringen av variabler är naturligtvis minnet (SRAM) varför även dessa adresser flyttas dit.

Reservera således 3 bytes per adress i minnet:

```
    .dseg
ADDR0:
    .byte 3
ADDR1:
    .byte 3
```

och låt `X` vara pekare:

```
    ldi     XL,LOW(ADDR0)
    ldi     XH,HIGH(ADDR0)
    call    MEM_RD
    :
    :
```

I. Spelet *4 i rad*

Spelet *4 i rad* är ett spel för två spelare och går ut på att lägga sin spelbricka på ett sådant sätt att man får fyra egna brickor i rad — horisontellt, vertikalt eller diagonalt. En central funktion i spelet är att utvärdera spelplanen för att hitta om någon spelare åstadkommit fyra brickor i rad. Detta exempel handlar om den programkoden.

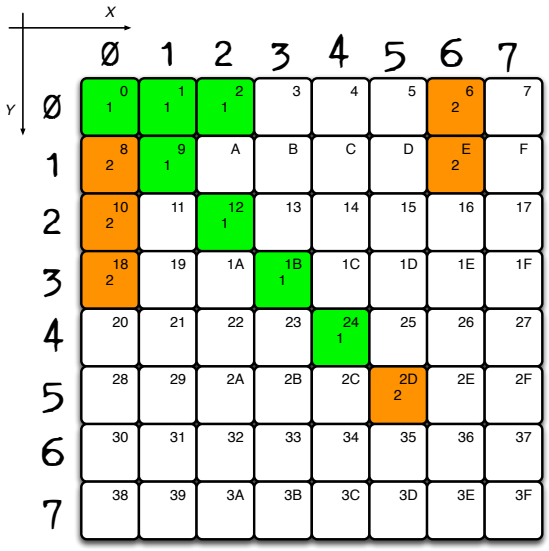
Här används en spelplan som är 8 x 8 punkter och implementationen skall ske i AVR-assembler. Vi antar helt glatt också att hårdvara för detta redan finns så vi behöver inte bry oss om denna detalj.

I minnet måste spelplanen representeras med en lämplig datastruktur. För att börja någonstans antar vi att detta redan är beslutat och här används en array i SRAM med indexen $[0..63]$ där varje element kan innehålla en färg enligt följande kodning:

0 Oanvänd

1 Grön

2 Orange



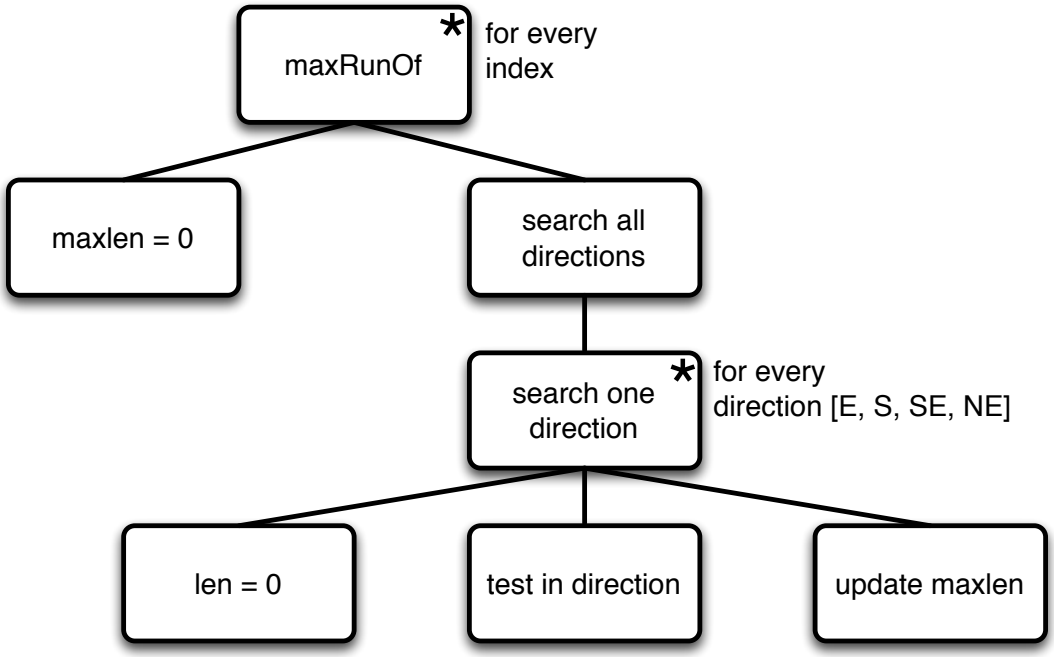
Arrayens index går från $00 till $3F (0..63). Indexet/8 ger både kolumnkoordinat (x-led) och radkoordinat (y-led). Oanvända positioner (här vita) antas ha värdet 0.

Arrayens index $[0, 1, \ldots, \$3E, \$3F] = [000\,000, 000\,001, \ldots, 111\,110, 111\,111]$ kan översättas till koordinater i y- och x-led genom att dela indexet med 8. Rad 0 innehåller då adresserna $[\$00..\$07]$, rad 1 adresserna $[\$08..\$0F]$ och så vidare, som i bilden ovan.

Programmets strategi har valts att vara tämligen direkt:

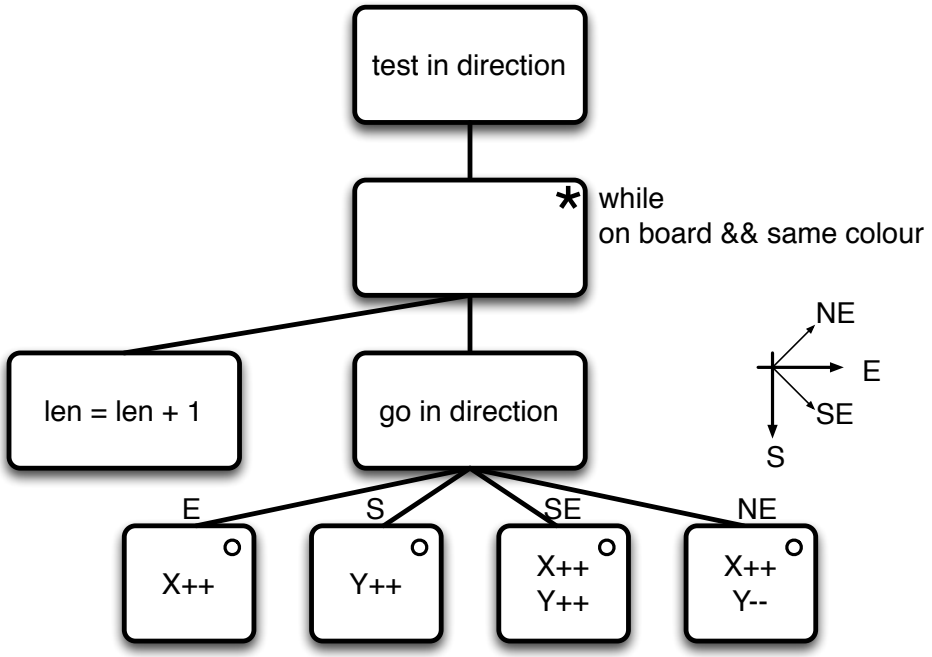
Antag att den initiala maximala längden av en önskad färg är noll. Från en given koordinat och den önskade färgen, analysera sedan omgivningen i de önskade riktningarna horisontellt, vertikalt eller diagonalt och undersök antalet brickor i respektive riktning tills färgen upphör. Om antalet brickor överstiger det tidigare maximala, uppdatera till det nya maximalvärdet. Gör detta för alla spelplanens alla positioner.

Strategin omformas till följande JSP-strukturdiagram. Den första administrerar att alla index används som startpunkter och att alla riktningar analyseras. Variabeln `len` nollställs innan sökningar i en viss riktning påbörjas och, om den erhållna längden var större än den hittills största (i `maxlen`), uppdateras den sistnämnda.



Med en önskad färg som argument letar algoritmen ut den längsta förekomsten av denna färg i riktningarna horisontellt, vertikalt eller diagonalt.

Givet en startpunkt räknas längden `len` upp så länge man inte är utanför spelplanen eller färgen upphört. Man ”provgår” alltså i en riktning steg för steg så länge man kan. Strukturdiagram för detta blir:



Pseudokod Någon form av pseudokod för diagrammen ovan kan skrivas

```
maxRunOf(colour):
    maxlen = 0
    for index = 0 to 63
        X = index mod 8
        Y = index / 8
        for dir in [E, S, SE, NE]:
            len = 0
            testInDirection(X, Y, colour, dir)
            if(len > maxlen) maxlen = len
    return maxlen;
```

respektive

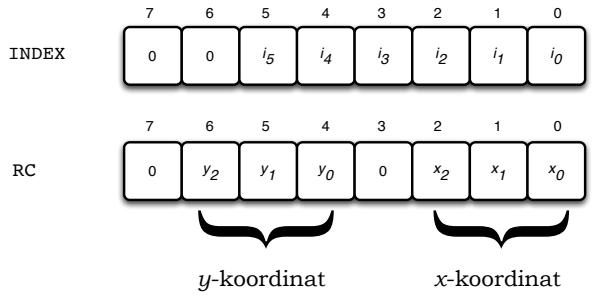
```
// check if within board and same colour
stillOK(X, Y, colour):
    return  (X>=0) &&
            (X<8)  &&
            (Y>=0) &&
            (Y<8)  &&
            (v[X][Y] == colour)


// step in direction as long as allowed
testInDirection(X, Y, colour, dir){
    while(stillOK(X, Y, colour))
        len = len + 1
        if dir == E:    X++
        if dir == S:    Y++
        if dir == SE:   X++, Y++
        if dir == NE:   X++, Y--
    return len
}
```

Här antas att spelplanen inryms i arrayen `v` med koordinater som tidigare:

```
v[8][8]={
    {0,2,1,0,2,1,0,1},
    {0,2,2,0,2,1,0,0},
    {0,0,2,0,0,2,1,0},
    {0,0,0,2,2,0,2,2},
    {0,2,0,1,2,1,1,1},
    {1,0,1,0,1,0,0,0},
    {1,0,0,0,0,1,0,0},
    {0,1,2,2,0,2,2,1}
}
```

Assemblerkoden utförs med ledning av pseudokoden och JSP-diagrammen. I detta skede behöver inga optimeringar göras mer än val av datarepresentationer.

I nedanstående kod används en enda byte för både y- och x-koordinat, övre nibblen är y-koordinat och lägre x-koordinat:



Arrayens index går från $00 till $3F i bitarna $i_5, \ldots, i_0$. Variabeln `RC` *(för* Rectangular Coordinate*) innehåller motsvarande koordinater i två nibbles.*

Givet denna `RC` kan en första version av koden skrivas:

```
    .equ    BLACK   = 0
    .equ    GREEN   = 1
    .equ    ORANGE  = 2
    .equ    BOARDSIZE = 64      ; 8x8

    ; --- variables in SRAM

    .dseg
LEN:    .byte 1 ; length so far
MAXLEN: .byte 1 ; best length
INDEX:  .byte 1 ; index, 0..63
RC:     .byte 1 ; Rectangular Coordinates[y,x]
DIR:    .byte 1 ; search direction 0..3
COLOUR: .byte 1 ; colour to find
VMEM:   .byte BOARDSIZE

    .cseg
    rjmp    MAIN

BOARD:  ;   0 1 2 3 4 5 6 7
        .db 0,0,2,1,0,0,1,1 ; 0
        .db 0,0,1,1,1,1,1,0 ; 1
        .db 2,1,0,0,1,1,1,1 ; 2
        .db 2,1,1,1,1,1,1,0 ; 3
        .db 2,1,1,0,1,0,1,0 ; 4
        .db 2,2,0,0,1,1,1,1 ; 5
        .db 2,0,1,1,1,2,2,0 ; 6
        .db 0,1,1,0,0,0,1,0 ; 7

POPULATE_BOARD:
    ldi     XH,HIGH(VMEM)
    ldi     XL,LOW(VMEM)
    ldi     ZH,HIGH(BOARD*2)
    ldi     ZL,LOW(BOARD*2)
    ldi     r17,BOARDSIZE
POP_LOOP:
    lpm     r16,Z+
    st      X+,r16
    dec     r17
    brne    POP_LOOP
    ret

MAIN:
    ldi     r16,HIGH(RAMEND)
    out     SPH,r16
    ldi     r16,LOW(RAMEND)
    out     SPL,r16
    rcall   POPULATE_BOARD

    ; maxrunof(COLOUR)
    ldi     r16,GREEN
    sts     COLOUR,r16
    rcall   maxrunof
    lds     r16,MAXLEN
AGAIN:
    rjmp    AGAIN

    ; --- Still valid position and colour?
    ; --- Z=1 if ok, Z=0 else
STILLOK:
    push    r17
    ; coords within 0..7?
    lds     ZL,RC
    andi    ZL,$88      ; 10001000
    brne    OUTSIDE ; Z = 0
    ; get INDEX from coordinates
    lds     ZL,RC
    mov     r17,ZL
    andi    r17,$70
    lsr     r17
    andi    ZL,$07
    or      ZL,r17
    ; check if same colour
    ldi     r17,LOW(VMEM)
    add     ZL,r17      ; add VMEM_START to get adress
    ld      r16,Z       ; get colour in array
    lds     r17,COLOUR  ; get search colour
    cp      r16,r17     ; Z = 1 om OK
OUTSIDE:
    pop     r17
    ret

MAXRUN:
    push    r17
MAXRUN1:
    rcall   STILLOK
    brne    MAXRUN_DONE ; not same colour
    lds     r16,LEN     ; len++
    inc     r16
    sts     LEN,r16
    ; current direction
    lds     r16,RC
    lds     r17,DIR
    cpi     r17,0
    breq    EAST
    cpi     r17,1
    breq    SOUTH
    cpi     r17,2
    breq    SEAST
    ; was NORTHEAST
NEAST:
    subi    r16,$10     ; y--
EAST:
    subi    r16,-$01    ; x++
    rjmp    COORDS_OK
SEAST:
    subi    r16,-$01    ; x++
SOUTH:
    subi    r16,-$10    ; y++
COORDS_OK:
    sts     RC,r16      ; update coords
    rjmp    MAXRUN1
MAXRUN_DONE:
    pop     r17
    ret
```

```
    ; --- Find longest run of COLOUR
    ; --- Update MAXLEN
MAXRUNOF:               ; from position INDEX
    clr     r16         ; no MAXLEN yet
    sts     MAXLEN,r16
NEXTP:
    sts     INDEX,r16   ; Save current origin
    ldi     r16,$0      ; start EAST
NEXTDIR:
    sts     DIR,r16     ; save  direction to test
    clr     r16         ; current LEN = 0
    sts     LEN,r16
    ; get RC from INDEX
    lds     r16,INDEX
    mov     r17,r16
    andi    r17,$38
    lsl     r17
    andi    r16,$07
    or      r16,r17
    sts     RC,r16
    rcall   MAXRUN      ; find length from this coord
    lds     r16,MAXLEN  ; if longer, update MAXLEN
    lds     r17,LEN
    cp      r16,r17
    brpl    NOT_LONGER
    sts     MAXLEN,r17  ; new maxlen
NOT_LONGER:
    lds     r16,DIR     ; get new direction
    inc     r16
    cpi     r16,4       ; are directions left?
    brne    NEXTDIR     ; yes
    lds     r16,INDEX   ; are startpoints left?
    inc     r16
    cpi     r16,BOARDSIZE
    brne    NEXTP       ; yes
    lds     r16,MAXLEN  ; return with maxlen
    ret
                        .
```

I detta skick kan koden kompileras och provköras med olika färger (`COLOUR`) och spelplaner (`BOARD`). När den väl konstateras fungera kan några förenklingar och förtydlingar genomföras vilket småningom leder oss till två ytterligare versioner.

Andra versionen Läser man koden sticker koordinatomvandlingarna ut efter en stund. Dessa är en serie bitmanipulationer på en lägsta nivå som stör läsningen av koden. Därför läggs dessa i subrutiner med bra namn så är de ”ur världen” på denna nivå:

```
    ; --- Get INDEX into ZL pointer from RC
INDEX_FROM_RC:
    lds     ZL,RC
    mov     r17,ZL
    andi    r17,$70
    lsr     r17
    andi    ZL,$07
    or      ZL,r17
    ret
```

```
    ; --- Convert INDEX to RC
RC_FROM_INDEX:
    lds     r16,INDEX
    mov     r17,r16
    andi    r17,$38
    lsl     r17
    andi    r16,$07
    or      r16,r17
    sts     RC,r16
    ret
```

Lägg märke till hur `INDEX` alltid innehåller startpunkten och `RC` erhålls ur denna när så behövs: `RC` är en funktion av `INDEX` och aldrig tvärtom! Det är ett enkelriktat beroende.

Nu kan berörda subrutiner skrivas läsligare som

```
    ; --- Still valid position and colour?
    ; --- Z=1 if ok, Z=0 else
STILLOK:
    push    r17
    ; coords within 0..7?
    lds     ZL,RC
    andi    ZL,$88
    brne    OUTSIDE ; Z = 0
    rcall   INDEX_FROM_RC
    ; check if same colour
    ldi     r17,LOW(VMEM)
    add     ZL,r17
    ld      r16,Z
    lds     r17,COLOUR
    cp      r16,r17
OUTSIDE:
    pop     r17
    ret


    ; --- Find longest run of COLOUR
    ; --- Update MAXLEN
MAXRUNOF:
    clr     r16
    sts     MAXLEN,r16
NEXTP:
```

```
    sts     INDEX,r16
    ldi     r16,$0
NEXTDIR:
    sts     DIR,r16
    clr     r16
    sts     LEN,r16
    rcall   RC_FROM_INDEX
    rcall   MAXRUN
    lds     r16,MAXLEN
    lds     r17,LEN
    cp      r16,r17
    brpl    NOT_LONGER
    sts     MAXLEN,r17
NOT_LONGER:
    lds     r16,DIR
    inc     r16
    cpi     r16,4
    brne    NEXTDIR
    lds     r16,INDEX
    inc     r16
    cpi     r16,BOARDSIZE
    brne    NEXTP
    lds     r16,MAXLEN
    ret
```

Man kan nu fortsätta på den inslagna vägen och extrahera kodrader som hänger ihop, namnge dem som subrutiner och förtydliga den övergripande koden. Det är en fråga om *fingerspitzgefühl*, estetik och tillgängligt minnesutrymme, när, och exakt var, man ska sluta denna typ av förenkling.

Tredje versionen I programmering gäller generellt att vi måste anstränga oss veta så lite som möjligt (helst ingenting) om hur en rutin löser sin uppgift, dvs vi ska inte behöva känna till rutinens *interna* göranden. Exakt *hur* en rutin löser sin uppgift är rutinens problem, inte vårt. Med detta resonemang *isolerar* vi rutinerna från varann och minimerar möjligheten för korskopplingar och oväntade sidoeffekter.

I programspråket `C` är detta löst med funktionsprototyper i *header*-filer. Headerfilerna innehåller information om hur en funktion ser ut *utåt* men ingenting om hur den ser ut *inåt*. I standardbiblioteket `stdlib.h` återfinns till exempel prototypen

```
void qsort( void *base,
            size_t nitems,
            size_t size,
            int (*compar)(const void *, const void*))
```

som beskriver funktionens namn och dess argument. Precis det man behöver veta för att använda den. Själva koden för `qsort` återfinns i `qsort.c` och exakta implementationsdetaljer är dolda.

I objektorienterade språk erhålls motsvarande effekt med nyckelord som `private`, `protected` och dylikt.

I vår vardag är detta inget konstigt alls. Vi gör så automatiskt hela tiden. Ett exempel:

> På en resturang kan vi beställa en köttbit (helt utan argument). Med ett argument kan vi bestämma hur hårt stekt den skall vara. Det är så långt vi normalt går. Hur kocken ställer in värmen, om gas- eller elspis används, storlek på stekpannan med mera är kockens egen angelägenhet. Och inget vi ska behöva bry oss om.

Med detta tänk är den bästa rutinen den som inte har några in-argument alls, den näst bästa har ett enda argument och så vidare!

När vi programmerar, det må gälla lågnivå- eller högnivåprogrammering, skall vi agera på samma sätt. Till exempel underkänner vi koden till vänster direkt, medan den till höger är korrekt utförd:

```
SOUND:
    brne    NO_SOUND
    push    r16
    push    r17
    push    r18
    push    r22
    push    r25
    call    DO_SOUND
    pop     r25
    pop     r22
    pop     r18
    pop     r17
    pop     r16
NO_SOUND:
    ret
```

```
SOUND:
    brne    NO_SOUND
    call    DO_SOUND
NO_SOUND:
    ret

.
```

Rutinen `SOUND` skall inte behöva känna till vilka register som behöver sparas för att `DO_SOUND` skall fungera, det är *enbart* `DO_SOUND`:s interna angelägenhet. Dessa kodrader skall ligga i `DO_SOUND`.

På köpet blir också `SOUND` kortare och enklare att läsa och förstå.

Man kan tänka i termer om *ansvarsområden*. Vems ansvar är det att hålla ordning på sina variabler? Rutinens ansvar så klart, och definitivt *inte* alla andras ansvar.

Med denna nyvunna insikt, hur skall programmet *4 i rad* modifieras?

I rutinen `MAXRUNOF` beräknas koordinaterna från `INDEX`. Men `MAXRUNOF` använder inte den informationen. Det är bara en tjänst åt `MAXRUN`. Uppenbarligen är `MAXRUNOF` helt ointresserad av om `MAXRUN` använder koordinater (eller något helt annat?) för att utföra sina beräkningar. Låt alltså `MAXRUN` ta ansvar för detta och modifiera rutinerna:

```
MAXRUN:
    push    r17
MAXRUN1:
    rcall   STILLOK
    brne    MAXRUN_DONE
    lds     r16,LEN
    inc     r16
    sts     LEN,r16
    ; current direction
    call    RC_FROM_INDEX   ; <- ny rad
    lds     r16,RC
    lds     r17,DIR
    :
    :
    :
    :
    :
    rjmp    MAXRUN1
MAXRUN_DONE:
    pop     r17
    ret

MAXRUNOF:
    clr     r16
    sts     MAXLEN,r16
NEXTP:
    sts     INDEX,r16
    ldi     r16,$0
NEXTDIR:
    sts     DIR,r16
    clr     r16
    sts     LEN,r16
                        -> till RC_FROM_INDEX
    rcall   MAXRUN
    lds     r16,MAXLEN
    lds     r17,LEN
    cp      r16,r17
    brpl    NOT_LONGER
    sts     MAXLEN,r17
NOT_LONGER:
    lds     r16,DIR
    inc     r16
    cpi     r16,4
    brne    NEXTDIR
    lds     r16,INDEX
    inc     r16
    cpi     r16,BOARDSIZE
    brne    NEXTP
    lds     r16,MAXLEN
    ret
```

Nu behöver `MAXRUNOF` enbart bry sig om `INDEX`, `DIR`, `LEN` och `MAXLEN`. ”Enbart” förresten? Till och med dessa kan vara för många.

`MAXRUN` känner till `DIR` och `LEN` men tar själv reda på koordinaterna då den behöver det (och inte tidigare).

Ytterligare modifieringar, för att inte tala om helt annorlunda algoritmer, är säkert möjliga. Prova själv!

J. Drivrutiner

I mindre program kan det ofta räcka med en enda rutin som initierar hårdvaran och man gör det på ett enda ställe. Det handlar typiskt om att sätta portriktningar och konfigurera intern hårdvara (A/D-omvandlarkanal, *prescaler*, *baudrate* med mera).

Ett större program struktureras i moduler där kommunikation med hårdvara sker via speciella programmoduler, *drivrutiner*. Fördelarna med drivrutiner är flera, det blir till exempel lättare att testa de enskilda modulernas funktion var för sig, men även programflödet på en högre nivå tjänar på att uttryckas i form av signalflöden mellan moduler. Det handlar hela tiden om att abstrahera bort för ögonblicket ovidkommande, information.

I operativsystemet UNIX används traditionellt fem funktioner för detta, nämligen:

Funktion	Syfte
`open()`	Initiera och aktivera hårdvaran
`seek()`	Adressera hårdvaran
`read()`	Hämta data från hårdvaran
`write()`	Skriv data till hårdvaran
`close()`	Avaktivera hårdvaran

Det är dessa funktioner vi egentligen genomfört hittills i hårdvaruinitieringen.[1]

Alla funktioner stöds inte nödvändigtvis av all hårdvara. För en A/D-omvandlare kan man föreställa sig följande funktioner:[2]

Funktion	Utför
`ADC_open()`	*A/D enable* = 1, *prescaler*, *single ended mode* i `ADCR`
`ADC_seek()`	Välj ADC-kanal dvs sätt `ADMUX`
`ADC_read()`	Läs av `ADCH`
`ADC_write()`	— (Inte tillämpbart)
`ADC_close()`	*A/D enable* = 0 i `ADCR`

Det kan underlätta kodförståelsen om dessa ”standard”-funktioner används. Dessutom underlättas namngivningen något, det är alltid svårt att hitta bra och precisa namn på subrutiner.

En läsning från A/D-omvandlaren kanalen `CHAN_3` kan nu utföras som

```
ldi     r16,CHAN_3      ; select channel
call    ADC_seek        ; set channel
call    ADC_read        ; do the read, result in r16
:
```

eller, om `ADC_read` tar kanalnumret som argument direkt i `r16`:

```
ldi     r16,CHAN_3      ; select channel by #define
call    ADC_read        ; do the read, result in r16
:
```

[1] Vad mer, än dessa fem funktioner, kan man vilja göra med en hårdvaruenhet?

[2] I UNIX används funktionerna enligt ”`value = read(ADC)`” där `ADC` typiskt är en *fil*, vilket förutsätter begreppet filsystem och strömmar, något vi inte är begåvade med i vår enkla assemblermiljö. Oavsett kan abstraktionen med dessa fem funktioner vara nyttig i assembler.

På motsvarande sätt kan man föreställa sig att kommunikation till en ansluten LCD-hårdvara sker genom anrop till följande rutiner:

Funktion	Utför
`LCD_open()`	Sätt I/O-portar, välj antal rader, cursortyp osv
`LCD_seek()`	Sätt minnesadress för läsning eller skrivning
`LCD_read()`	Läs av en minnesadress
`LCD_write()`	Skriv tecken eller kommando
`LCD_close()`	Återställ portar, dvs gör dem till ingångar

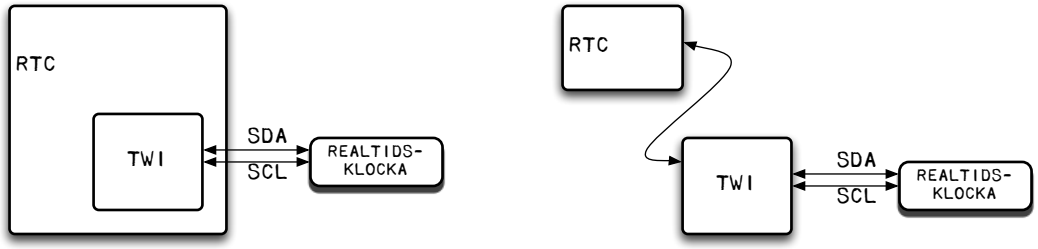
På en övergripande nivå består programmet av en del med själva programlogiken och en del med drivrutiner där drivrutinerna är enda vägen att hantera hårdvara.

> **All hantering av portar och inbyggd hårdvara sker via drivrutinerna och direkt åtkomst är förbjuden.**

På detta sätt är de olika programmodulerna isolerade från varandra och påverkan sinsemellan är minimerad så länge de enskilda modulerna konstrueras för detta.

Moduluppdelning Det kan vara instruktivt att studera hur en drivrutin för en extern realtidsklocka RTC kan designas på modulnivå. I detta fall kommunicerar den yttre hårdvaran via TWI, *Two Wire Interface*, och naturligt används den interna hårdvaran för detta.

Två möjliga lösningar visas i bilden nedan:



I den vänstra figuren har TWI-funktionaliteten bakats in i koden för realtidsklockan RTC. Upplägget blir svårare att felsöka och man har blandat ihop två olika uppgifter i ett kodstycke.

I den högra figuren har TWI-funktionaliteten frikopplats till en egen modul som då a) kan testas och verifieras för sig och b) också kan anropas från andra rutiner än enbart RTC. Man har vunnit testbaret och generalitet. Detta upplägg är att föredra.

Anropsmodell Det är en fördel om ett standardiserat protokoll används för argumentöverföring till och från drivrutinerna. Två huvudmetoder vilka beskrivs här:

1. Metoden med **argument via stacken** förutsätter ingen kunskap den anropade rutinen. I vår assembler `push`:as argumenten innan subrutinsanrop, det är allt vi behöver veta. Om vi dessutom antar allmänt att `r16` är ett temporärregister har metoden dessutom fördelen att inte behöva påverka mer än detta enda register, trots möjligen flera argument:

```
ldi     r16,ARG1        ; push arguments
push    r16
ldi     r16,ARG2
push    r16
call    DRIV_rutin
pop     r0              ; dummy pop
pop     r20             ; get result in r20
```

Här används det sällananvända registret `r0` som markör till programmeraren att det poppade värdet är ointressant. Ett enkelt resultatvärde kan direkt poppas till önskat register. En nackdel är det extra arbete `DRIV_rutin` måste utföra för att komma åt argumenten på stacken.

2. Metoden med **argument via register** är enklare och lämpar sig då argumentet endast är en enda byte:

```
ldi     r16,ARG1        ; load argument
call    DRIV_rutin      ; get result in r16
```

Inte bara anropet blir enklare utan även `DRIV_rutin` blir kortare. Metoden förutsätter dock att kommunikation med drivrutinen sker via registret `r16`, det vill säga åtminstone någon kännedom om rutinens konstruktion, något vi normalt vill undvika.

Privata subrutiner och variabler För att ytterligare isolera programavsnitt, subrutiner och/eller variabler från varandra kan vi implementera *inkapsling* (*encapsulation*), en term som kanske vanligen associeras med objektorienterad programmering men är äldre än så.

Det finns inget särskilt stöd i assembler för inkapsling men det hindrar oss inte att trots detta programmera med inkapsling som mål.

Redan nu använder vi inkapsling genom att `push`:a register på stacken för att skapa lokala variabler. Dessa variabler har då ett *scope* inom rutinen och måste återställas innan återhopp.

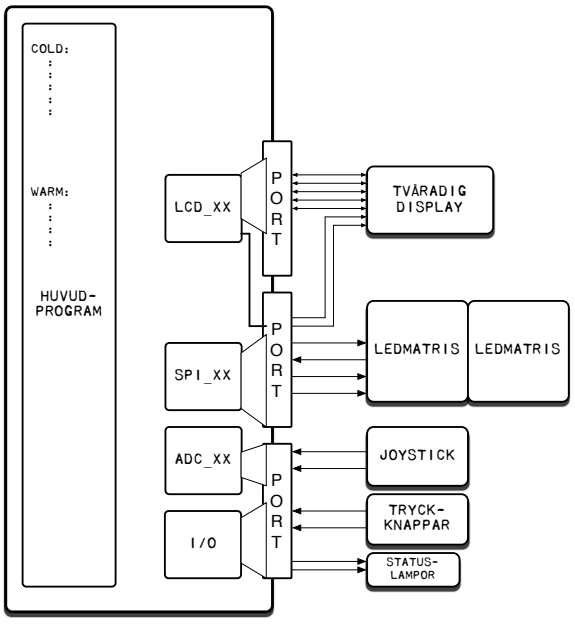
På en programmodulsnivå kan vi också implementera inkapsling genom att inte tillåta yttre åtkomst eller ens kännedom om interna, privata subrutiner.

Sådana privata subrutiner kan konstrueras med en enklare anropsmodell då de aldrig skall exponeras utåt. En hel del optimeringar kan därmed möjliggöras.

Huvudprogrammet med programlogiken består av två tydliga symboliska ingångar `COLD:` och `WARM:`

- `COLD:` innehåller startup-kod för processorn, såsom att sätta stackpekaren, kanske rensa `RAM`-minne och utförs typiskt en gång.
- `WARM:` innehåller om-initiering av variabler med mera, till exempel omstart av ett spel för en ny spelomgång.

Bilden till höger visar huvudprogrammet med sitt beroende av drivrutinerna `LCD_XX`, `ADC_XX` och så vidare för att hantera hårdvara.



Med en sådan initial moduluppbyggnad kan drivrutinerna senare utökas med ytterligare funktionalitet. Man kan föreställa sig kollisionsflaggor som indikerar om hårdvaruresursen är ledig att användas eller inte, kontroll om argumenten har tillåtna värden, felhantering om kommunikation med hårdvaran inte lyckats och så vidare.

K. Erfarenhetslista

Detta del av appendix var från början tänkt att heta ”God programmeringssed” eller något ditåt, men benämningen **erfarenhetslista** är bättre då man erfarenhetsmässigt får bättre program, färre fel och enklare felsökningar om man tänker och skriver kod på detta sätt.

Bra kod skall:

1. vara strukturerad och använda subrutiner
   - skriv en subrutin för varje identifierbar, specifik programdel
   - faktorisera där det är meningsfullt
   - använd *så få argument* som möjligt (helst noll)
   - en subrutin får ha enbart en (1) ingång vid början (”`LABEL:`”)
   - en subrutin får ha enbart en (1) utgång vid slutet (”`ret`”)
   - `jmp`/`rjmp` får enbart användas *inom* en subrutin
2. vara indenterad och lämpligt kommenterad
   - symboliska adresser ska börja i kolumn 0
   - instruktioner ska börja ett TAB-steg in
   - ibland är en subrutin bättre än en kommentar
   - undvik onödiga kommentarer av typen

     ```
     inc ZL      ; increment ZL
     ```

     bättre är

     ```
     inc ZL      ; next element
     ```
3. använda meningsfulla variabel- och subrutinnamn
   - en rutin ska göra det den heter
   - sträva efter självdokumenterande kod
4. använda pekare
   - pekare underlättar avancerade datastrukturer
   - pekararitmetik är kraftfullt i assembler och C/C++.
5. parametriseras
   - använd globala parametrar (`.def` och `.equ` i filens start) och preprocessorn

6. vara relokerbar
   - programmet ska inte ställa krav på att ligga på vissa minnesplatser
7. vara resurssnål
   - använd inte onödigt mycket variabler
   - undvik onödig minnesåtgång, både i kod och data
   - använd loopar där detta är möjligt
   - rulla ihop repetitiva kodstycken
8. inte ha sidoeffekter
   - använd subrutiner för att isolera funktioner
   - undvik globala variabler
   - använd lokala variabler
   - överväg parameteröverföring via stack

L. Citatsamling

Här kommer en liten citatsamling som på olika sätt beskriver praktiskt hårdvarunära programmering:

- Något är fel när vi inte får ett väntat resultat
- Alla fel beror på något
- Det är sällan fel på komponenterna
- Utan felsökning står man still
- Är det värt att göra är det värt att göra i en subrutin
- Lös det ingenjörsmässigt: Få något att fungera först. Optimera sedan.
- ”Make everything as simple as possible, but not simpler.” — Albert Einstein
- Man löser problem **bättre** och man felsöker **bättre** om man vet **mer** om systemet än om man vet precis så mycket som behövs för att lösa uppgiften. — Gammalt Forth-ordspråk
- ”You don’t understand a problem until you can simplify it.” — Gammalt Forth-ordspråk igen
- ”The enemy of a good plan is the dream of a perfect plan.” — Carl von Clausewitz
- ”Never, never, never give up!” — Winston Churchill