Datorteknik
Programmering av AVR

Michael Josefsson

Version 0.8 2018-01-15

Innehåll

0. Inledning **7**
0.1. Datormodell 10

1. Programmerarmodell **11**
1.1. Generella register och minneslayout 13
1.2. I/O-register 14
1.2.1. Programräknaren `PC` 14
1.2.2. Statusregistret `SREG` 14

2. Instruktioner **17**
2.1. Grupp 1 — Flytta data 17
Exempel: `ldi` 17
Exempel: `mov` 18
2.2. 8-bitars register 18
Exempel: ASCII 18
2.3. 16-bitars register 19
Exempel: `adiw` 19
Exempel: `st/sts/ld/lds` 19
2.4. Grupp 2 — Aritmetiska operationer 20
Exempel: `add` 20
2.5. Grupp 3a — Logiska operationer 20
Exempel: Maska bitar 21
2.6. Grupp 3b — Skiftinstruktioner 21
2.7. Grupp 4 — Hoppinstruktioner 22
Exempel: Ovillkorligt hopp 22
Exempel: Villkorligt hopp 22
Exempel: Beräkna summan 24
Exempel: Absoluta och relativa hopp 25
2.8. Grupp 5 — I/O-instruktioner 25
2.8.1. I/O-portar 25
Exempel: Konfigurera portar 27
Exempel: Skip-instruktionen (`GET_KEY`) 28
Exempel: Compare and Skip if Equal (addera 1) 30

3. Binär aritmetik **31**
3.1. Talbaser 31
3.2. Addition 32
Exempel: Addition, kortare ordlängd 33
Exempel: Addition, längre ordlängd 33
3.3. Talrepresentationer 34
Exempel: Addition, flyttal 34

3.4. Basen 2_{10} . . . 35
Exempel: Binär till decimal . . . 35
3.5. Höger- och vänsterskift . . . 35
3.6. Omvandling från decimal till binär form och tvärtom . . . 36
Exempel: Decimal till binär . . . 36
3.7. Hexadecimal representation . . . 37
3.7.1. Omvandling binär till hexadecimal form . . . 38
Exempel: Binär till hex . . . 38
3.7.2. Olika skrivsätt . . . 38
3.7.3. Omvandling hexadecimal till decimal form . . . 38
Exempel: Hex till decimal . . . 38
3.8. 2-komplement . . . 39
Exempel: Binär till decimal, tvåkomplement . . . 39
Exempel: Tvåkomplement fungerar . . . 41
Två viktiga anmärkningar . . . 41
Exempel: Decimal till binär . . . 42
3.9. Addition, subtraktion, skift för tvåkomplementstal . . . 42
3.10 Spill . . . 43
Exempel: Spill . . . 44
3.11 Hårdvara . . . 45
Exempel: Räkna hemma! Spill och carry . . . 47

4. Adresseringsmoder **49**
Exempel: Absolut adress . . . 50
Exempel: Minnespekare . . . 51
Exempel: Tabell i FLASH . . . 53
Exempel: Förskjutning . . . 54
Exempel: Postinkrement . . . 55
Exempel: Predekrement . . . 55

5. Subrutiner och stacken **57**
Exempel: Udda paritet . . . 59
5.1. Lokala variabler . . . 60

6. Externa och interna avbrott **61**
6.1. Inledning . . . 61
Exempel: Avbrott . . . 61
6.1.1. Pollning . . . 63
6.2. Avbrott på AVR . . . 63

7. Parameteröverföring **65**
7.1. Parameteröverföring via register . . . 65
Exempel: Parameteröverföring . . . 65
7.2. Parameteröverföring via returstacken . . . 66
7.3. Returargument via returstacken . . . 67

8. AD-omvandling **69**
Exempel . . . 69

9. Preprocessor och kompilering **71**
Exempel 73

A. Tabeller i FLASH **77**

B. Utdrag ur filen m16def.inc **79**

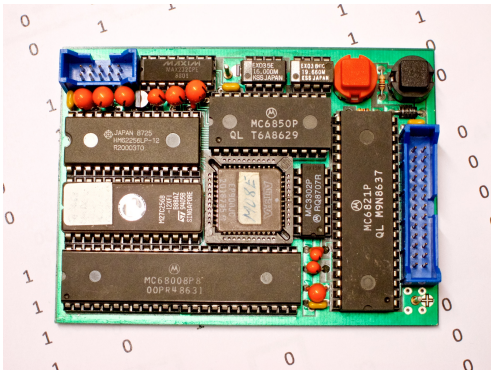
C. Miniprojekt **83**

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, 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), 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, ATMEGA16. 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.

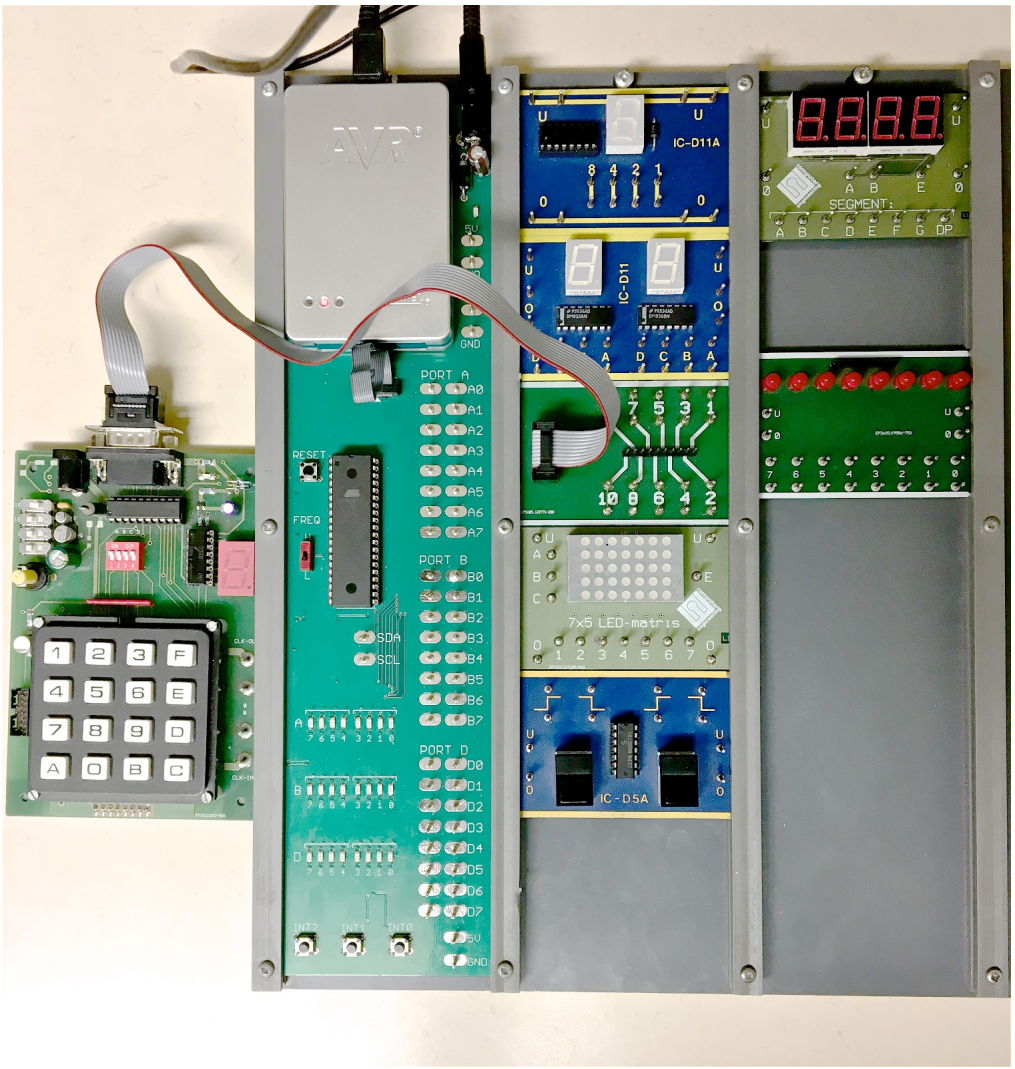
Detta betyder att du som student med nödvändighet bibringas en fundamental förståelse för hur processorer fungerar. Denna kunskap är till glädje inte bara i senare projektkurser utan också i programmeringskurser i högnivåspråk.[3] Även kurser i kompilatorkonstruktion kan ha målet att generera instruktioner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

[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.

[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.



Figur 0.1.: Samtliga laborationer i kursen utförs på det egenutvecklade *dalia*-kortet (instickskortet till vänster). Dalia innehåller processor (ATmega16A), kontaktstift för anslutning av önskad kringutrustning, programmeringsanslutning via USB till värddator och lysdioder som representerar de olika in- och utsignalernas digitala värde. 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.

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:

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*.

1. Programmerarmodell

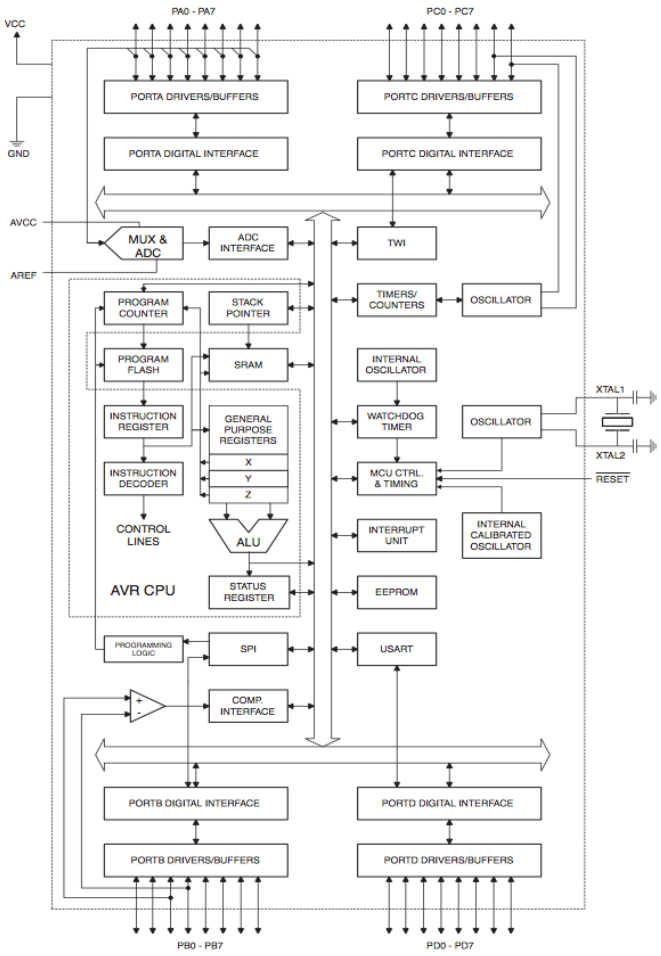
Alla laborationer i kursen utförs på på mikrokontrollern ATmega16. Detta är en komplett liten dator med AVR-processorn som centralenhet. Innan laborationerna börjar måste vi förstå hur mikrokontrollern fungerar och speciellt hur man programmerar den att utföra det vi begär av den.

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

Processorn är en åtta-bitars processor, det 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, PCH och PCL, 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 minnet.
- **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, och därmed vi, 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).
- **General Purpose Registers** SRAM-minnet innehåller, trots att det inte framgår av figuren, även processorkärnans arbetsregister. Vi ser dessa som register även om de i realiteten är implementerade i SRAM.



1

Figur 1.1.: *Mikrokontrollern ATmega16 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.*

- **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.

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 på kapselns utsida. Varje pinne kan konfigureras som antingen ingång eller utgång och utgör processorns sätt att kommunicera med

[2]Här spänningarna +5 V eller 0 V.

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

omvärlden.

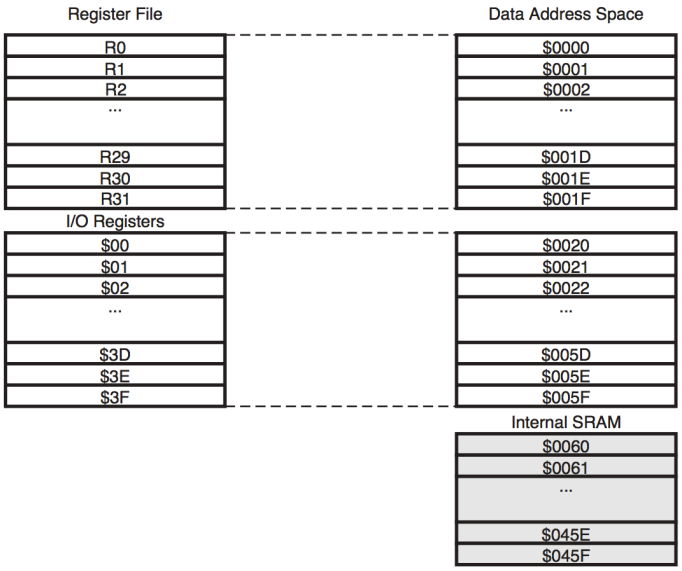
Vi kommer under kursen att kunna konstatera att mycket av databehandlingen i en processor går åt till att utföra, var för sig tämligen enkla operationer. En traditionell processor ser ungefär ut som den arkitektur vi hittills sett.[4]

1.1. Generella register och minneslayout

I programmerarmodellen ingår normalt en beskrivning av de register och minne processorn består av. En mer komplett programmerarmodell innehåller dessutom processorns instruktions- och dataformat samt ingående instruktioner. Här nöjer vi oss med en avslutande beskrivning av register och minne.

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

Registren återfinns i processorns minne som en del av SRAM med adresser enligt figuren:



Figur 1.2.: *SRAM återfinns på adresserna `$000–$45f`. Processorns interna register, `r0–r31`, mappas in i SRAM:ets lägsta adresser. På liknande sätt mappas* I/O*-adresser in på efterföljande adresser:* I/O*-adress `$3d` återfinns alltså på SRAM-adress `$5d`.*

[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.

En slutsats av detta minnesupplägg är att register r0–r31 också kan nås genom minnesadresserna $00–$1f och så vidare. Man är alltså inte tvingad att använda "rXX" som argument utan kan också använda direkta minnes*adresser*. Detta kommer vi dra nytta av senare när vi tittar på olika adresseringsmoder.

Trots namnet "generellt register" är alla register är inte skapade lika. Aritmetiska och logiska operationer kan bara utföras på registren r16–r31, medan r0–r15 bara kan användas för temporär lagring av data.

1.2. I/O-register

Visserligen är det i de generella registren processorns egentliga jobb utförs, 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, via portarna, sker.

Registren *programräknaren* och *statusregistret* hanteras också som I/O-register i denna processor. I andra processorer kan de vara hårdare knutna till processorn medan en del tillverkare anser att de är en del av processorkärnan.

1.2.1. Programräknaren PC

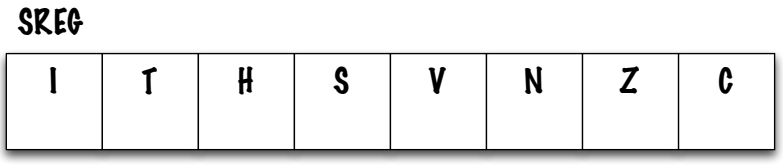
Registret PC innehåller adressen till den instruktion som utförs för tillfället. När programmet körs rad efter rad kommer programräknaren att ökas på allteftersom. Om man gör ett hopp i programmet laddas programräknaren med en ny adress. Programräknaren är 16 bitar och registret PC består alltså av två delar PCH och PCL för programräknarens Höga och Låga byte. Vi tecknar detta förhållande som PC=PCH:PCL.[5]

1.2.2. 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.

[5]Denna notation kan vi dock inte använda när vi skriver kod.

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 1.3.: *Statusregistret SREG innehåller flaggor som indikerar senast körda 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 användas som lagringsplatser,
- r16–r31 är de egentliga generella registren,
- r26–r31 kan användas som adressregister (pekarna X, Y och Z),
- I/O-registren används för att kommunicera med hårdvaruenheter.

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`”.

■

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.

Exempel: `mov`

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

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 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: ASCII

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

■

[1] Andra tycker annat!

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.[2]

■

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*).

[2] `adiw` fungerar enbart på registerparen `r25:r24`, `r27:r26`, `r29:r28` och `r31:r30`.
[3] En enstaka byte kan peka ut adresserna `$00` till `$ff`, med två byte kan man peka ut adresserna `$0000`–`$ffff`.

■

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.

■

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 digitalteknikken vet vi vad dessa booleska operatorer gör, det som skiljer nu är att de utförs på hela register.

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

Exempel: Maska bitar

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

■

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.

`asl` *Arithmetic Shift Left*, Aritmetiskt vänsterskift = Logiskt vänsterskift.

Vi ser att 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 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.

■

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 `$9F` finns två positiva tal, placera det största talet i minnet på adress `$A0`.

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.

Instruktionerna `cp` och `brpl` förutsätter *tvåkomplementskodade* tal. Den exakta innebörden av detta kommer senare. För nu räcker det att känna till att en byte kan innehålla ett sådant tvåkomplementskodat tal mellan –128 och +127.

Följande fall kan inträffa för statusflaggorna:

- 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.[5]

I dessa fall kan man alltså dra ut information om jämförelsen genom att studera en enstaka flagga. På liknande sätt kan man specialstudera carry-flaggan (`C`) och overflow-flaggan (`V`) med hoppinstruktionerna `brcs`, `brcc` (*carry set*, *carry clear*) respektive `brvs`, `brvc` (*overflow set*, *overflow clear*).

Det finns även hoppinstruktioner för villkoren *same or higher*, *lower*, *greater than or equal to* med flera. Konsultera mikrokontrollerns datablad för detaljerna.

> *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]*

[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.

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 `sub`-instruktionen 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.

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 beslutar vi att tilldela registren så att `r20` = varvräknare, `r22:r21` = summan.

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 Vanligtvis finns i processorer två sorter av de ovillkorliga hoppinstruktionerna `jmp`, *jump*, och `br`, *branch*. Skillnaden mellan dem är att i `jmp` anger argumentet hoppadressen i 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 ATmega16 heter det relativa hoppet `rjmp`.

Det visar sig att de villkorliga hoppen alltid är relativa, det vill säga hoppets destination fås genom att addera en *offset* till den nuvarande instruktionens adress. 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.

Både `jmp` och `rjmp` kan hoppa bakåt. För `rjmp` 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.

Exempel: Absoluta och relativa hopp

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

■

I vår processor används namnet *branch* enbart för de villkorliga hoppen (av typen `brne`) och är alltid relativa.

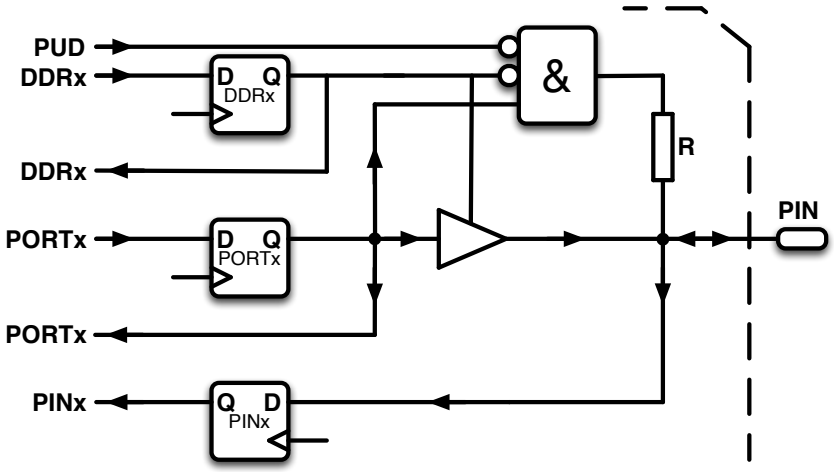
2.8. Grupp 5 — I/O-instruktioner

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:

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.



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.*

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.

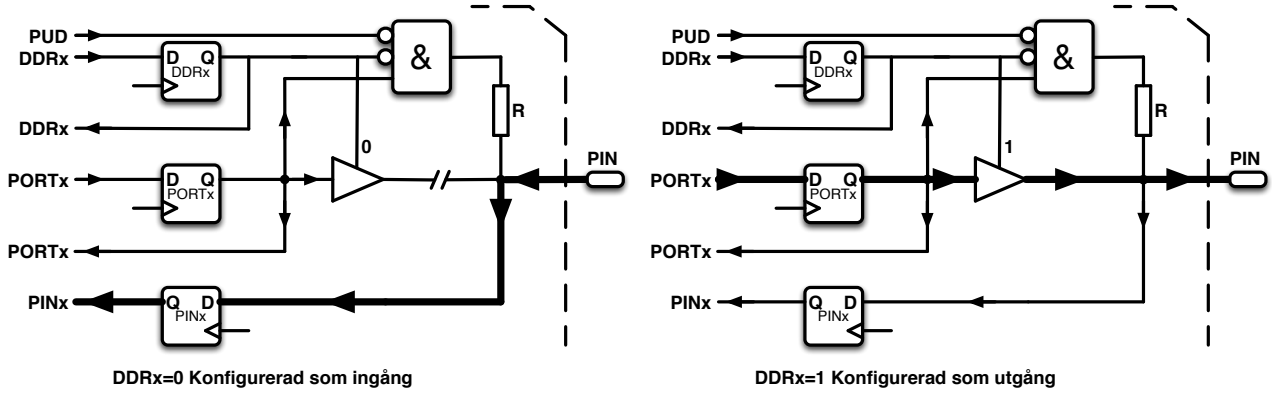
Vi får två fall för varje bit beroende på om DDRx konfigurerar biten som ingång eller utgång, se också fig 2.2:

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!

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).



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 "—//—".*

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.

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]

■

[7] Svar nej! Varför?

Sammanfattningsvis kan man alltså teckna fyra kombinationer av `DDR`x och `PORT`x:

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

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: Skip-instruktionen (`GET_KEY`)

> En tryckknapp är ansluten till processorns PORT A, bit 3. En nedtryckt knapp utgörs av digital etta. Använd `sbic` för att returnera 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` betyder *sant*.

[8] Ett skip är enkelt att implementera hårdvarumässigt.

Avslutningsvis skall nämnas ytterligare en skip-instruktion, `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!

■

3. 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.

3.1. 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, ...

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, A, B, C, D, E, F,
10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, ...

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

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,. . .

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.

3.2. 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] Tvåtalsdelen 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
```

■

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.

■

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

3.3. 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):

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

[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.

$$0.063 + 0.398 = 0.460$$

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 divideras med 10 och exponenten ökas med ett.

■

3.4. 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 åttagrupp 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.

3.5. 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!

3.6. 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:

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.

. . . 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.

■

3.7. 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

3.7.1. 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:

Lär dig tabellen och motsvarande binära fyrabitsgrupp utantill!

■

3.7.2. 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:

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

3.7.3. 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!

■

[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?

3.8. 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:

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:

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.

■

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

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?

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.

Talets belopp är således 3 och med rätt tecken är svaret –3.

■

3.9. Addition, subtraktion, skift för tvåkomplementstal

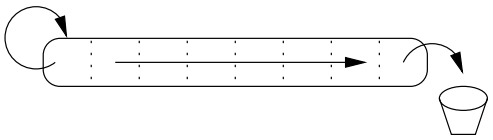
Med några enkla prov kan vi konstatera att följande gäller även för tvåkomplementrepresentationen:

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:

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

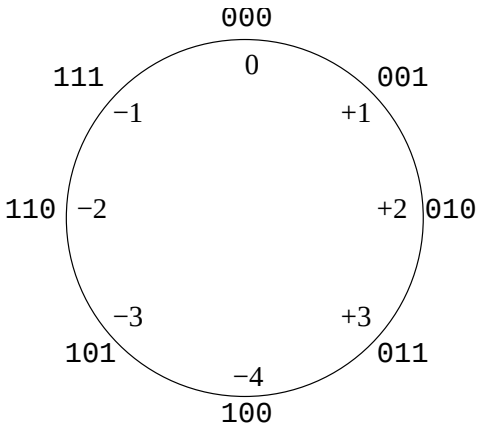
Det aritmetiska högerskiftet kan alltså[7] ritas som



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

3.10. 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 3.1.: *Cirkulärgrafen åskådliggör addition och subtraktion i tal med begränsad ordlängd. Här är den utförd för tvåkomplementstal med ordlängden 3. Addition sker 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 3.1.

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.

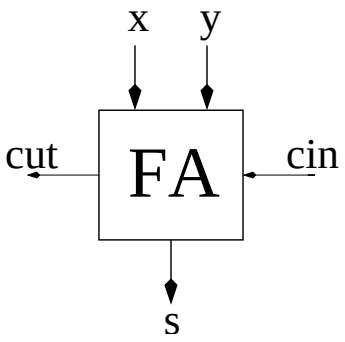
På samma sätt visualiserar cirkulärgrafen när en minnessiffra, *carry* skapas vid övergång mellan `111` och `000`. Lägg märke till att närvaron av en carry inte innebär något problem om talen är tvåkomplementskodade, då är övergången från negativa till positiva tal helt legitim. Carry uppstår även vid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

3.11. 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 3.2.: 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:

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

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

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

Vi får en utgående carry 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.

Men spill då? Adderaren är tämligen värdelös om den inte kan säga till om den räknat fel. 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 liten tabell:

Om vi nu kommer ihåg när spill kan inträffa kan vi tillfoga en kolumn för dessa fall också. 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 betraktar talen som tvåkomplementskodade, det vill säga de har en teckenbit.

Som tur är inträffar spill bara i två fall. 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:

Det är en enkel sak att komplettera vår adderare med denna indikator och har nu plötsligt en fungerande adderare för tvåkomplementstal, den kan addera positiva såväl som negativa tal.

Men 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,

Allt i högerledet — additioner — kan vi redan utföra. Men vi behöver också kunna invertera ena talet för tvåkomplementeringen $\overline{\mathcal{Y}}$. Vid addition ska talet förstås inte inverteras. Således måste vi ordna till en styrbar inverterare, det vill säga styrbar så att den kan fås att invertera eller inte beroende på en yttre styrsignal.

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

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

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 $\overline{\texttt{ADD}}/\texttt{SUB}$ som ju är hög vid subtraktion och låg annars.

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)			
+	00001000	(+8)	+	00000001	(+1)	+	11111110	(–2)			
=	00001110	(+14)	=	10000000	(–128)	=	00000010	(+2)			
		C= V=			C= V=			C= V=			

	00000010	(+2)		11111110	(–2)		10000001	(–127)			
+	11111100	(–4)	+	11111100	(–4)	+	11000010	(–62)			
=	11111110	(–2)	=	11111010	(–6)	=	01000011	(+67)			
		C= V=			C= V=			C= V=			

■

4. 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

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

Man skiljer 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 den i exemplen hittills. Vi koncentrerar oss här på de övriga och börjar mer 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*) mest signifikant byte av talen 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.

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`.

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

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

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

Instruktioner för att flytta data (här `ld`/`lds`/`st`/`sts`) påverkar inte flaggorna.

Övning: Vilka adresseringsmoder används i varje rad i programmen? Spelar det någon roll om talen A och B är tvåkomplementskodade eller inte?

■

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 högnivåprogrammering är *pekare* lösningen på många datastruktursproblem. Med adresseringsmoden *data indirekt* kan vi även i vår processor använda pekare.

SRAM Lösningen är att ha ett register som *pekare* till det statiska minnet. I processorn kan något av `X`-, `Y`- eller `Z`-registren användas för detta. 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.

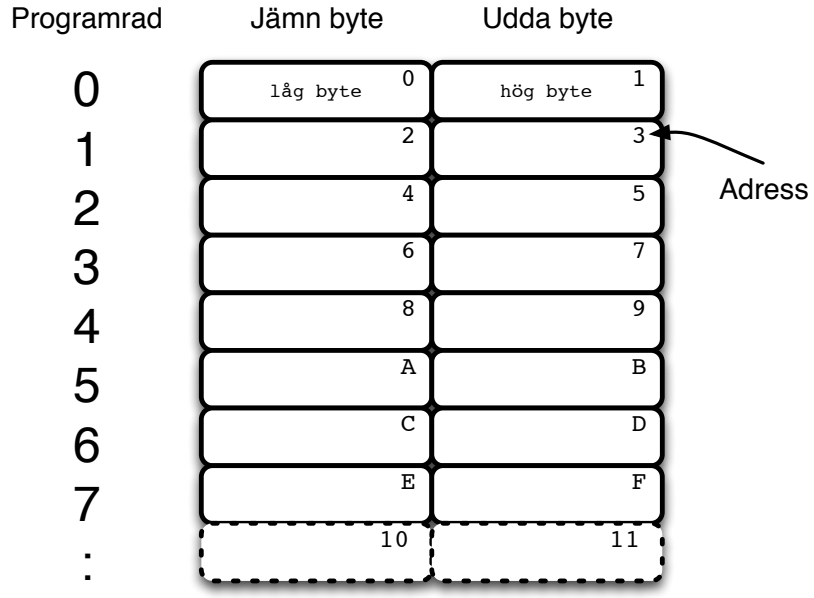
Vi bestämmer oss för att använda X-registret som består av XH:XL=r27:r26 så adressen måste delas upp innan den läggs in i X. LSB skall läggas på den lägre adressen som vanligt. För att inte behöva hålla detta faktum i huvudet kan man istället för r26 skriva XL för att beteckna LSB. Motsvarande för r27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

■

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.



Figur 4.1.: 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.

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

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

Exempel: Tabell i FLASH

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

Med .db (*Define Byte*) kan vi lägga in enstaka byte 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.

adiw är en specialinstruktion för 16-bitars adressberäkningar. Argumentet är den lägsta byten av två (XL-, YL- eller ZL), eller motsvarande uttryck ”XH:XL” och talet som adderas kan vara mellan 0 och 63. adiw kan dessutom användas på register r25:r24.

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.

Exempel: Förskjutning

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

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: Postinkrement

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

■

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: Predekrement

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

■

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.

5. 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.

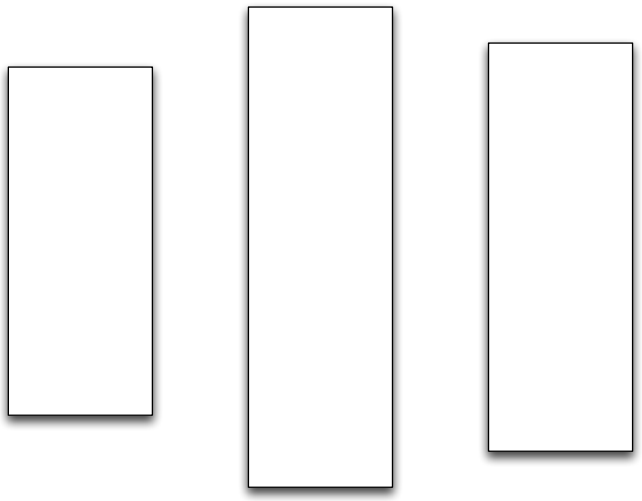
Som vi hittills lärt oss är enda möjligheten att göra dessa uppgifter att programmera dem på de ställen där de behövs i programmet. Det innebär att alla de programrader uppgiften kräver måste klistras in i programmet där de behövs.

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/rjmp` 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/rjmp`.

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/rjmp` 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 `rjmp` 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 rätt 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

[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.

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 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?

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. Använd dem ofta. 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!

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]

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

■

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

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

5.1. Lokala variabler

En subrutin skall vara klar att anropas från flera ställen i den övriga programkoden. För att göra detta så smärtfritt som möjligt vore det bra om subrutinen inte förstör innehållet i de register huvudprogrammet, det anropande programmet, använde. Huvudprogrammet kanske använde 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 dessa 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:

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

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. 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.

Med *avbrott* kan en yttre enhet signalera till processorn att det är dags att hoppa till en särskild subrutin, en *avbrottsrutin*.

6.1. Inledning

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 så kan den själv påkalla 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]

[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.

Tangentbordet lägger ut den nedtryckta tangentens binära värde på sina dataledningar och påkallar sedan uppmärksamhet genom att dra i avbrottssignalen IRQ.

Processorn är konstruerad så att den, innan den påbörjar varje ny instruktion, kontrollerar om någon ryckt i ”avbrottstlinan”. Om så är fallet väljer den bort nästa instruktion för ögonblicket och gör ett subrutinanrop (som sparar återhoppsadressen på stacken i vanlig ordning) till avbrottsrutinen istället.

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

■

På 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.

Med processorns *inre tillstånd* menar man processorns samtliga registers innehåll. Att spara hela processorns inre tillstånd är en tidsmässigt dyr affär varför man bara vill spara så lite som absolut möjligt.

Processorn sparar inget automatiskt utan det är programmeraren som får ombesörja att `SREG` återlämnas i korrekt skick.

En typisk avbrottsrutin konfigureras som nedan beroende på om `r16` riskerar förstöras i rutinen eller inte.

[2]Avbrott, interrupt, är en del av processorns så kallade *undantagshantering*.

6.1.1. 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.

I det första fallet handlar det om att programmet i sin normala gång i tur och ordning frågar 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 ombesörjer detaljerna. Metoden med flaggor kan användas för att styra programmet att befinna sig i olika *tillstånd*.

Om flera enheter kan påkalla avbrott, det vill säga likt på bussar att alla kan trycka på knappen för att stanna bussen, måste processorn på något sätt kunna avgöra vem det var som önskade uppmärksamhet. Då kan *avbrottsrutinen* konstrueras 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.

6.2. Avbrott på AVR

Processorn har åtskilliga avbrottskällor. Förutom det vanliga externa avbrottet kan flera hårdvaruenheter i kapseln orsaka avbrott. Till exempel kan komplett skrivning till USART, TWI eller en färdig A/D-omvandling signaleras med avbrott.

De olika avbrottsvektorerna ligger i början av FLASH-minnet och upptar adresserna `$00`–`$28`. Första programrad efter tabellen blir då `$2A`. En programbörjan kan se ut som nedan. Normal programstart efter spänningspåslag sker på rad 0 i programminnet och är ett hopp till rutinen (adressen) `RESET`.

```
        jmp     RESET             ; Reset Handler
        jmp     INT0              ; IRQ0 Handler
        jmp     INT1              ; IRQ1 Handler
        jmp     TIM2_COMP         ; Timer2 Compare Handler
        jmp     TIM2_OVF          ; Timer2 Overflow Handler
        :
        :                         ; Massor med andra vektorer
        :
        .org    $2A               ; Hoppat bock över vektorerna
RESET:
        ldi     r16,HIGH(RAMEND)  ; Första programraden
        out     SPH,r16
        ldi     r16,LOW(RAMEND)
        out     SPL,r16
        sei                       ; Tillåt avbrott härifrån!
```

Efter kallstartshoppet till `RESET` fylls avbrottsvektortabellen på med övriga avbrott. Ett `IRQ0`-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` inte ett `call`!

Övriga rader i *avbrottsvektorerna* fungerar på motsvarande sätt beroende på olika tillstånd hos de olika hårdvaruenheterna.

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

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

Om två avbrott skulle inträffa samtidigt ”vinner” det avbrott som har lägst adress i tabellen. INT0 har alltså högre *prioritet* än INT1 och så vidare.

Konfigurationsbitar för avbrottsinställningar återfinns i de olika hårdvarufunktionernas I/O-register. Dessutom måste bit 7, *Global Interrupt Enable* `I` i `SREG` vara aktiverad, oftast med assemblerinstruktionen `sei`.

7. 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 kompilering av 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.

7.1. Parameteröverföring via register

Det enklaste sättet, och kanske det lämpligaste vid handassemblering, 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.

Exempel: Parameteröverföring

Använd registret `r17` som parameterregister vid subrutinanrop.

■

En nackdel är att `r17` i och med detta är ”hårdkodat” varje gång subrutinen skall anropas. En annan nackdel är naturligtvis att argumentet i detta fall inskränker sig till vad som kan rymmas i ett register.

7.2. 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

På detta sätt behöver inte `r25` ”hårdkodas” som i det förra fallet.

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. Återhoppsadressen kräver två bytes och först därunder återfinns parametern `param1`.

Stackinnehållet efter anropet, det vill säga när subrutinen körs blir då:

Med indirekt adressering med förskjutning, och i detta fall förskjutningen +3, kan `param1` hämtas. Notera att stackpekarens innehåll inte ändras medan minnet adresseras.[1]

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

I AVR-processorn 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

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.[2]

7.3. Returargument 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)`":

[2]Och vad hade den blivit då? Simulera!

8. AD-omvandling

Mikrokontrollern är bestyckad med en 10-bitars AD-omvandlare. Omvandlaren kan styras att göra omvandling på en av sex ingångar. AD-omvandlaren hanteras med fyra åttabitars register:

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

AD-omvandlarens funktion och många inställningar beskrivs utförligt i mikrokontrollerns datablad. För vår del belyses användningen enklast med ett exempel.

Exempel

> Starta en A/D-omvandling och hämta ett nytt värde så fort det finns tillgängligt.

En omvandling startas genom att man ett-ställer `ADSC`-biten (*AD Start Conversion*) i A/D-omvandlarens statusregister `ADCSR`. Så länge hårdvaruenheten är upptagen är biten `ADSC` i `ADCSR` ettställd, när den nollställs är omvandlingen klar.[1]

[1] 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.

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 steg där symboler ersätts och i vissa fall enklare (statiska) uttryck kan beräknas. Det andra steget är en översättning av den resulterande preprocessade textmassan till hexadecimal kod för mikrokontrollerns FLASH-minne.

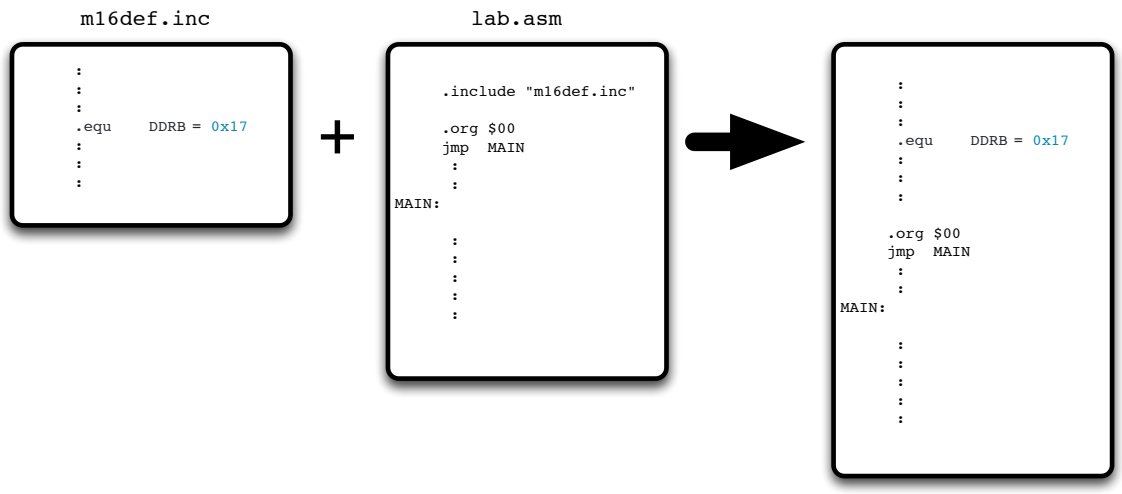
Preprocessor

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.

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 här ett *direktiv* till preprocessorn att inkludera denna fil på plats som i figuren nedan:



Några vanliga och användbara direktiv anges nedan

Direktiv	Namn	Betydelse
`.org`	*origin*	Skriv här (adress)
`.byte`	*byte*	Namnge 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

.org, .cseg sätter kompilatorn på en specifik adress i SRAM- eller FLASH-minnet. `.cseg` anger explicit att följande kod hör till FLASH-minnet. Kod för att hoppa över avbrottsvektorerna vid programstart:

```
        .cseg                           ; default
        .org    $0000
        jmp     START
        ;
        ; avbrottsvektorer
        ;
        .org    INT_VECTORS_SIZE    ; definierad i m16def.inc till 42
START:
        ; Programstart
```

.byte, .dseg, .db definierar (inte reserverar) variabeladresser i SRAM. För att övergå till att SRAM används först direktivet `.dseg` och för att växla till FLASH `.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 värdena 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:

```
        .macro   SUBIW ; macro med argumenten @0, @1 och @2
                 subi  @1,LOW(@0)
                 sbci  @2,HIGH(@0)
        .endmacro

        :
        SUBIW    $1234,r16,r17   ; r17:r16=r17:r16-$1234
                 ; @0  @1  @2
        :
```

Övrigt Preprocessorn kan också göra livet enklare för assemblerprogrammeraren med sitt stöd för aritmetiska och logiska operationer. Observera att dessa beräkningar görs 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 samma dvs $16
```

Kompilering

Kompileringssteget är det steg som känner igen assemblerinstruktioner och kan översätta de till 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 assembler instruktioner 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

- 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—

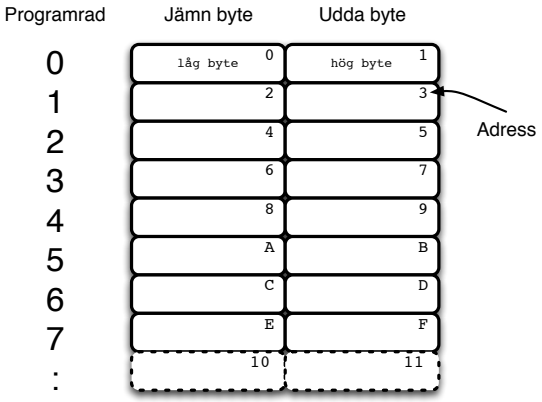
A. Tabeller i FLASH

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

- programrad, och
- minnesadress.

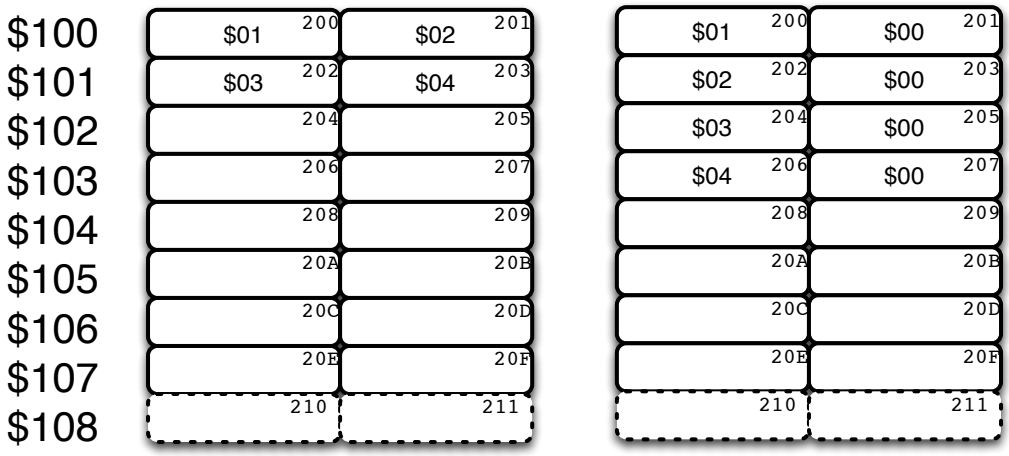
I normalfallet använder vi hela tiden programmets rader när vi skriver program. Det är dessa rader som innehåller programmets instruktioner och det är dessa rader vi hoppar till med `jmp` osv.

En instruktion är 16 bitar bred dvs två bytes. Dessa 16-bitars *word* utgör programraderna och man kan skriva att raden består av en jämn och en udda byte:



Med `.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 jämna 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 adressen. I vänsterfallet ovan kan `lpm` med postinkrement användas för att stega igenom tabellen medan högerfallet kräver "`adiw ZL,2`" för att peka ut nästa korrekta word.

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

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

B. Utdrag ur filen m16def.inc

```
; ***** 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

; ***** BIT DEFINITIONS ***************************************

; ***** 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 IVCE  = 0 ; Interrupt Vector Change Enable
     .equ IVSEL = 1 ; Interrupt Vector Select
     .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
```

C. Miniprojekt

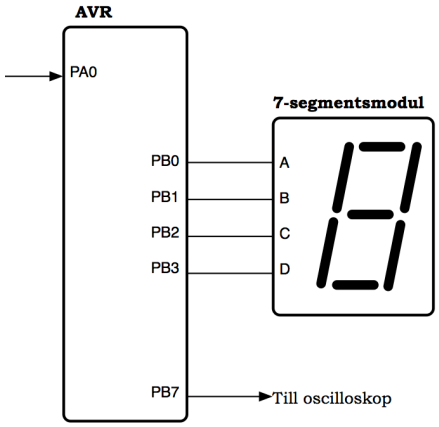
För den som tidigare inte programmerat assembler brukar den första kontakten med programmeringsspråket bli lätt tumultartad, det är många begrepp och tekniker som måste passa ihop på en och samma gång för att lösa uppgiften.

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 inrespektive 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 port `PINA` för läsning och 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 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* den intressanta biten. Och frasen ”skilja ut” gör att vi lystrar och förstår att det måste involvera den logiska instruktionen `andi`:

	?	?	?	?	?	?	?	X
AND	0	0	0	0	0	0	0	1
	0	0	0	0	0	0	0	X

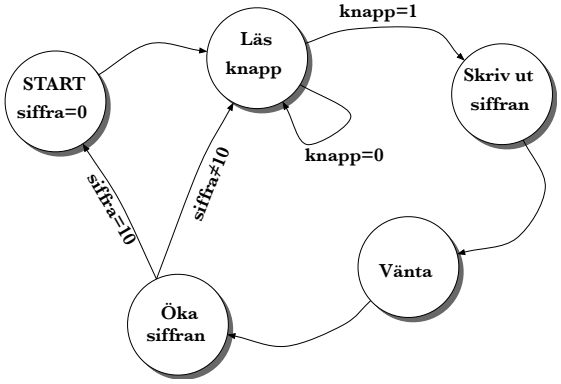
Genom att använda en `skip`-instruktion kan man alternativt göra en avkänning direkt mot pinnen. Det är något kompaktare kodmässigt då man slipper läsa hela byten och maska ut det väsentliga:

```
sbic    PINA,0    ; pinne 0
jmp     A         ; hit om biten==ett
:                 ; hit om biten==noll
```

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.

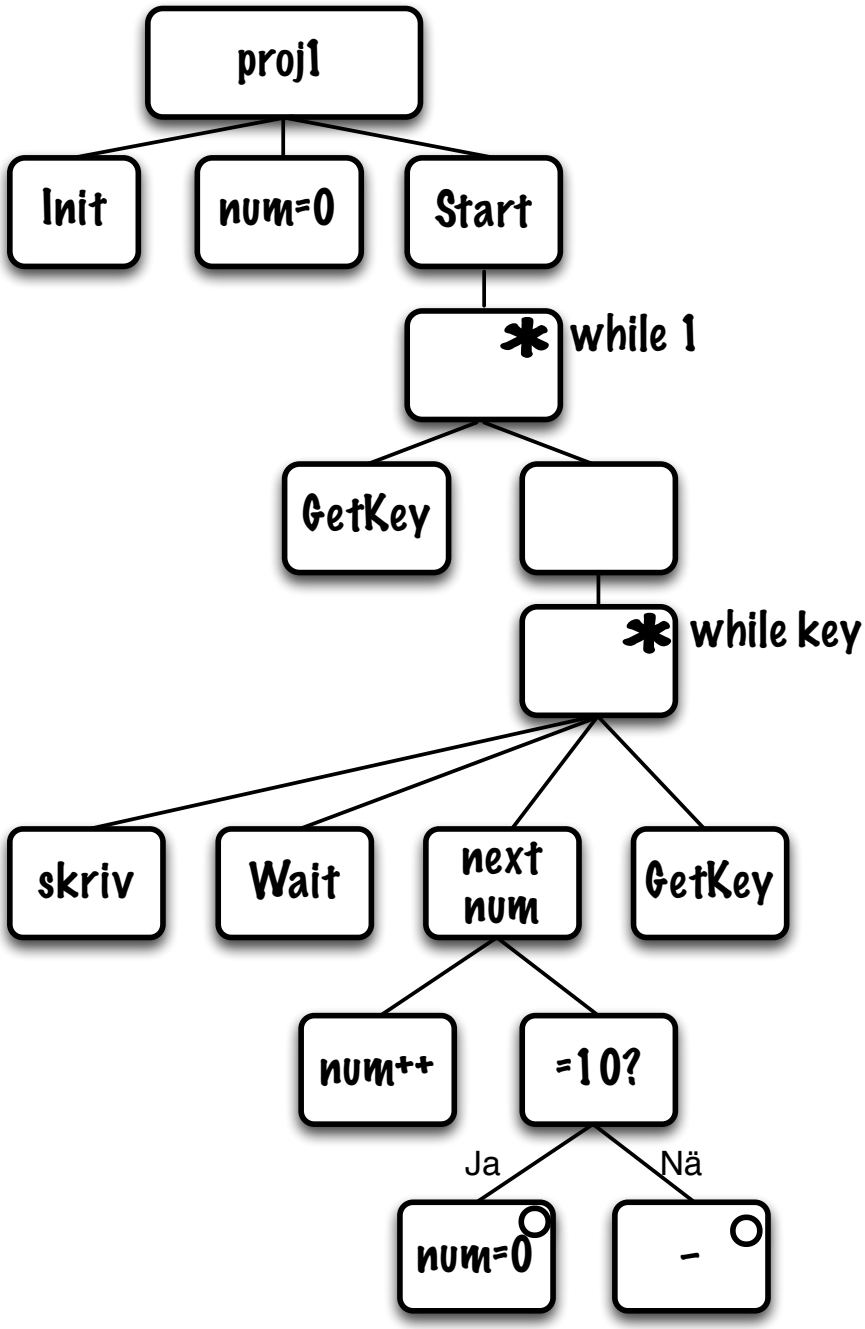
Handlar det om enstaka bitar i I/O-register som ska ett- eller nollställas kan instruktionerna `sbi` och `cbi` användas. I vårt fall är de olämpliga eftersom vi ska ställa alla fyra lägsta bitarna.

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.



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

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 kan man göra uppdelning i subrutiner senare och ändå få ett korrekt program.

Ett förslag på kod som realiserar strukturdiagrammet på en högre nivå kan då vara:

```
     ; r16-r19 free to use
     .def     num = r20      ; numer 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. Observera att det inte går att initiera stacken i en subrutin. Varför?

```
     ;
     ; --- 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`:

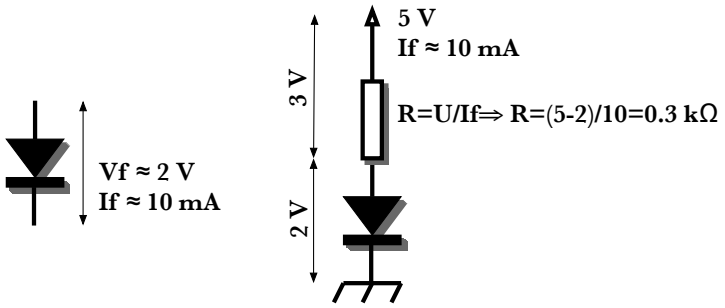
```
     ldi     r16,$FF
     out     PORTA,r16    ; Weak pullup
```

Och slutligen behövs en rätt rejäl vänterutin. Den är ganska tråkig men finns med här som referens:

```
     ;
     ; --- DELAY. Wait a lot!
DELAY:
     ldi     r18,50
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
```

`DELAY` 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 till och liknande för `r18`. Totalt tar `DELAY` 9868954 klockcykler enligt simulatorn, vilket motsvarar drygt 0.6 sekunder vid en klockfrekvens på 16 MHz.

Hur tänder man en lysdiod? 7-segmentsdisplayen består av lysdioder. 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 *förkopplingsmotstånd* anslutas i serie med dioden. Beräkning av lämpligt motståndsvärde sker som i figuren nedan.



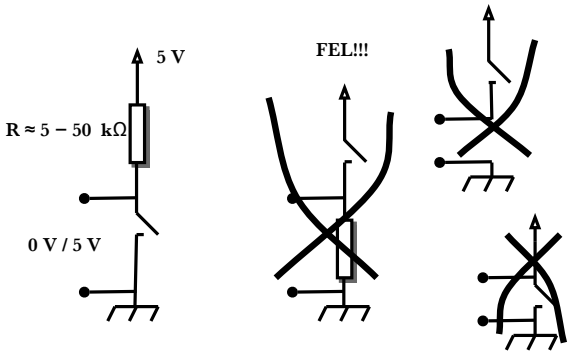
För att vara säker på att strömmen inte överstiger den dimensonerade väljer vi i praktiken närmast högre motståndsvärde hellre än ett lägre. I detta fall alltså standardvärdet $330\,\Omega$. Lysdiodens exakta spänningsfall, V_f, beror på vilken färg den lyser i, 2 V duger bra som riktvärde. Moderna lysdioder tänder dessutom vid lägre ström än detta varför även 390Ω eller till och med $470\,\Omega$ skulle fungerat i detta fall. Samtliga dessa tre är tillgängliga standardvärden.

Hur kopplar man in tryckknappen? Även om laborationen inte använder en tryckknapp behövs det för detta miniprojekt. Med tryckknappen vill vi på ett säkert sätt överföra informationen ”nedtryckt” respektive ”icke nedtryckt” till `PINA`. Som nämnts kan vi i allmänhet inte anta några värden för andra bitar än den vi själva styr med tryckknappen.

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.