

---

---

Inbyggda system med

---

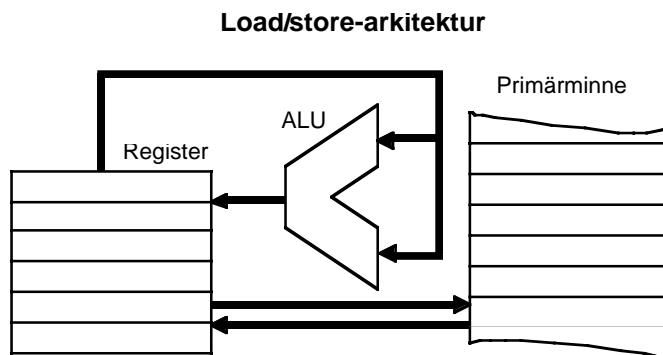
# AVR RISC PROCESSORER

---

*Logiska grindar och vippor*

*Binära tal*

*Binär aritmetik & binära koder*



*Programmerarens modell*

*Assemblerprogrammering*

*I/O-programmering*

*INBYGGDA SYSTEM MED AVR RISC PROCESSORER*

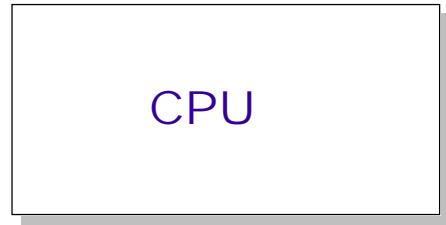
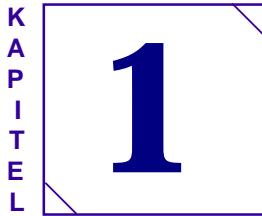
*16 nov 2003 KjM*

# Innehållsförteckning

<b>1 NÅGRA DATORTEKNISKA BEGREPP.....</b>	<b>6</b>
1.1 Var och hur används datorer? .....	6
1.2 En enkel datormodell.....	6
1.3 Information och betydelsen av information.....	8
1.4 Databuss, adressbuss och styrbuss.....	9
1.4.1 Modell av ett minne .....	9
1.4.2 Hur ordnas byten i minnet? .....	10
1.4.3 Bussar .....	11
1.4.3.1 Databussen .....	11
1.4.3.2 Adressbussen .....	11
1.4.3.3 Styrbussen.....	11
1.4.3.4 Inre och yttre bussar .....	11
1.4.3.5 En CPU ansluten till minnesenheter .....	12
1.4.4 Anslutning av yttre enheter till en processor .....	12
1.5 Persondatorn.....	13
1.6 Maskin-, assembler- och högnivåspråk .....	15
1.7 Uppgifter	
1.8 Svar.....	19
<b>2 GRUNDLÄGGANDE DIGITALTEKNIK.....</b>	<b>20</b>
2.1 Logiska tillstånd – Objekt med två tillstånd .....	20
2.2 Boolesk algebra.....	22
2.3 Grindlogik.....	26
2.4 Vippor och latchar – minneselement .....	27
2.5 CMOS, den digitala kretsens byggsten .....	30
2.6 Bussproblematiken.....	32
2.7 Uppgifter	
2.8 Svar	
<b>3 TALSYSTEM, KODER OCH BINÄR ARITMETIK .....</b>	<b>41</b>
3.1 Talsystem.....	41
3.2 Omvandling mellan talsystem.....	42
3.3 Binär representation av negativa tal.....	43
3.3.1 Tecken-belopp-kodning .....	43
3.3.2 1- och 2-komplementkodning.....	44
3.4 Binär aritmetik	
3.4.1 Addition och subtraktion	
3.4.1.1 Addition av binära tal.....	46
3.4.1.2 Grafisk addition och subtraktion av 4 bitars 2-komplementtal.....	47
3.4.1.3 Spill – overflow .....	48
3.5 BCD – Binary Coded Decimal.....	49
3.6 ASCII .....	50
3.6.1 Tabellen.....	50
3.6.2 ASCII-koder och C-programmering.....	51
3.7 Några format för binära filer/data .....	52
3.7.1 BinHex-formatet .....	52
3.7.2 Motorolas S-records.....	54
3.8 Uppgifter	
3.9 Svar.....	61
<b>4 AVR 8 BITARS DATORER.....</b>	<b>67</b>
4.1 Lite om datorarkitektur.....	67
4.1.1 Harvard- eller von-Neumann-arkitektur.....	67
4.1.2 CPU-arkitektur: CISC eller RISC .....	69
4.2 ATMEL ATmega16 .....	70
4.2.1 Portar.....	71
4.2.2 Externa avbrott.....	71
4.2.2.1 Reset-pinnen.....	71
4.2.3 Blockdiagram.....	72
4.2.4 Minnesmapp för ATmega16 .....	73

4.3 Uppgifter	
4.4 Svar	
<b>5 INSTRUKTIONER SOM AVR-PROCESSORN KAN UTFÖRA.....</b>	<b>75</b>
5.1 Principerna för AVR RISC CPU:n .....	75
5.1.1 Registerfil på 32 generella register på en byte vardera.....	76
5.1.2 ALU-enheten.....	76
5.1.3 Statusregistret – SREG .....	77
5.1.4 Programräknaren PC (instruktionspekarregister).....	78
5.1.5 Stackpekarregistret – SP.....	78
5.2 Instruktioner för att hämta och lagra data .....	79
5.2.1 Direkt adressering av ett register.....	79
5.2.2 Direkt dataadressering .....	80
5.2.3 Ladda ett register direkt från dataminnet.....	80
5.2.4 Ladda register 16 till 31 med en 8 bitars konstant .....	81
5.2.5 Spara ett register direkt i dataminnet.....	81
5.2.6 Sammanfattning av instruktioner för dataöverföring.....	82
5.3 Instruktioner för att bearbeta data.....	84
5.3.1 Direkt adressering av 2 register .....	85
5.3.2 Addition utan minnessiffra (carry) .....	85
5.4 Några hoppinstruktioner för ovillkorliga hopp .....	87
5.4.1 JMP – JuMP .....	87
5.4.2 RJMP – Relative JuMP .....	88
5.5 Subrutinanrop utan parameteröverföring.....	89
5.5.1 RCALL – Relative CALL to subroutine .....	89
5.5.2 RET – Relative CALL to subroutine.....	90
5.6 Uppgifter	
5.7 Svar	
<b>6 DIGITALA IN- OCH UT SIGNALER.....</b>	<b>96</b>
6.1 Instruktioner för IO-registermanipulering.....	99
6.1.1 I/O-adressering.....	99
6.1.2 Ladda ett I/O-register till ett register i registerfilen .....	100
6.1.3 Skriv ett register i registerfilen till ett I/O-register .....	100
6.2 Assemblerinstruktioner för bitantering.....	102
6.3 Lysdioder.....	104
6.4 Strömställare .....	105
6.4.1 Kontaktavstudsning i elektromekaniska strömställare .....	105
6.4.1.1 Hårdvarumässig lösning på kontaktstudproblemet .....	106
6.4.1.2 Mjukvarumässig lösning på kontaktstudproblemet .....	106
6.4.2 Många tryckknappar <-> tangentbord .....	109
6.5 Uppgifter	
6.6 Svar	
<b>7 GNU ASSEMBLATOR/LÄNKARE .....</b>	<b>115</b>
7.1 Maskinberoende syntax .....	115
7.2 Sektioner och relokering av kod/data .....	118
7.3 Assemblerdirektiv .....	119
7.3.1 .org new_location , fill .....	119
7.3.2 .equ symbol , uttryck .....	120
7.3.3 .comm symbol , längd.....	120
7.3.4 .byte uttryck_1 , ... , uttryck_n .....	121
7.3.5 .word uttryck_1 , ... , uttryck_n .....	121
7.3.6 .long uttryck_1 , ... , uttryck_n .....	121
7.3.7 .float uttryck_1 , ... , uttryck_n .....	121
7.3.8 .ascii "sträng1", ... , "sträng_n" .....	121
7.3.9 .asciz "sträng1", ... , "sträng_n" .....	121
7.3.10 .data .....	122
7.3.11 .text .....	122
7.3.12 .global symbol.....	122
7.4 Exempel: Lysdiodprogram .....	122

7.4.1	Indirekt dataadressering .....	122
7.4.2	Pseudokodslösning i C .....	123
7.4.3	Assemblerlösning.....	124
7.4.4	Anrop av assemblerprogrammet från main-funktionen .....	125
7.5	GNU Make - Generering av exekverbara filer.....	126
7.5.1	Makefile i våra projektmappar .....	128
7.6	Uppgifter	
7.7	Svar	
<b>8</b>	<b>C OCH ASSEMBLER.....</b>	<b>136</b>
8.1	Högnivåspråket C på maskinnivå .....	136
137		
8.2	Uppgifter	
8.3	Svar	
<b>9</b>	<b>VILLKORLIGA OCH REPETETIVA PROGRAMSATSER I ASSEMBLER ...</b>	<b>138</b>
9.1	Hopp i ett program .....	138
9.1.1	BREQ – Hoppa om lika med (BRanch if EQUAL).....	139
9.1.2	Hopp baserat på jämförelse av två tal .....	141
9.2	If-else-konstruktioner på assembler.....	142
9.3	Repetition där antalet varv i loopen ej är känt .....	146
9.4	Operationer på en vektor.....	147
9.4.1	Indirekt dataadressering .....	148
9.4.2	Summering av talen i en vektor .....	148
9.5	Uppgifter	
9.6	Svar .....	151
<b>10</b>	<b>MODULÄR PROGRAMMERING MED SUBRUTINER .....</b>	<b>155</b>
10.1	Modularisering av programmet med subrutiner.....	155
10.1.1	Stacken .....	155
10.1.2	CALL – CALL to subroutine .....	156
10.1.3	RET – RETurn from subroutine .....	158
10.2	Temporär datalagring på stacken.....	162
10.2.1	PUSH – PUSH register on stack.....	162
10.2.2	POP – POP register from stack.....	163
10.3	Överföring av parametrar till subrutiner.....	163
10.4	Uppgifter	
10.5	Svar .....	165
<b>INDEX</b>	<b>.....</b>	<b>168</b>



# Några datortekniska begrepp

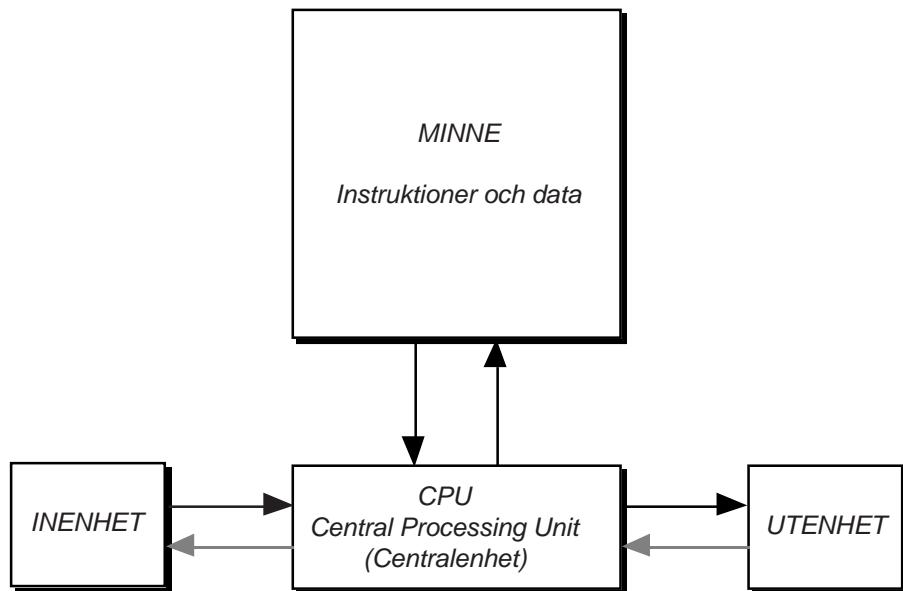
## 1.1 Var och hur används datorer?

Datorn är i grunden en *numerisk beräkningsmaskin* och används i många olika sammanhang. Räknedosan är kanske en av de vanligaste och mest uppenbara av datorer och finns i mer eller mindre alla hushåll. I ett land som Sverige finns det med all sannolikhet många gånger fler datorer än människor! Var kan vi hitta dem? I punktform skall vi lista ett antal tillämpningsområden:

- Bildatorer, i moderna fordon som bilar, lastbilar, bussar, etc. finns det många datorer. Uppgifterna för dessa är allt ifrån att styra bränsleinsprutning till motorn, transmissionsstyrning- och övervakning, läsfria bromsar, fönsterhissar, etc.
- Hemelektronik, i en modern TV, CD-spelare, radio, etc finns det normalt minst en mikrodator.
- Mobiltelefoni.
- Persondatorer.
- Elektronisk handel. I en vanlig snabbsköpsbutik finns det datoriserade kassor med streckkodsläsare , dessa kan stå i kontakt med någon centraldator som administrerar butikens lager. Kreditkortsläsare som via datorkommunikation är kopplade till någon central bankdator.
- Processdatorer för styrning av olika processavsnitt i t.ex. valsverk, pappersmaskin, kemiska processer, etc.
- Flygplanselektronik, ett flygplan som Boeing 767 har över 1000 mikrodatorer distribuerat över hela planet från vingtippar, landningställ, till cockpit för att styra, reglera och övervaka planet. Det distribuerade datorsystemet sparar kabel och därmed vikt.

## 1.2 En enkel datormodell

I alla tekniska sammanhang behöver vi använda modeller av olika slag för att kunna hantera komplexiteten i ett system. Modeller av olika slag använder vi för att skapa en bättre förståelse av det system vi arbetar med. Med hjälp av en modell framhäver vi det som är väsentligt. Och döljer detaljer som ej är relevanta i sammanhanget, som skulle kunna distrahera oss från att se helheten. I figuren som följer visas en enkel modell för en dator, vi kommer längre fram att använda andra modeller beroende vad vi vill fokusera på.



**Figur 1.1:** En enkel datormodell (variant 1)

Kärnan i datormodellen är *CPU:n* vilket är en förkortning av *Central Processing Unit*, på svenska kan detta översättas till *centralenhet* eller centrala beräkningsenheten. En dator kan ses som en maskin för numeriska beräkningar i det binära talsystemet.

Normalt finns det någon form av *inenhet* för att hämta in information från "omvärlden" till datorn. I en persondator har du t.ex. tangentbord och mus. I figuren finns det två pilar mellan inenheten och CPU:n. I en digital termometer har du en temperaturgivare som inenhet till mikrodatorn. Den svarta pilen indikerar att det huvudsakliga dataflödet går från inenheten till CPU:n och den gråa att styrinformation kan gå från CPU:n till inenheten för att programmera dess funktionalitet.

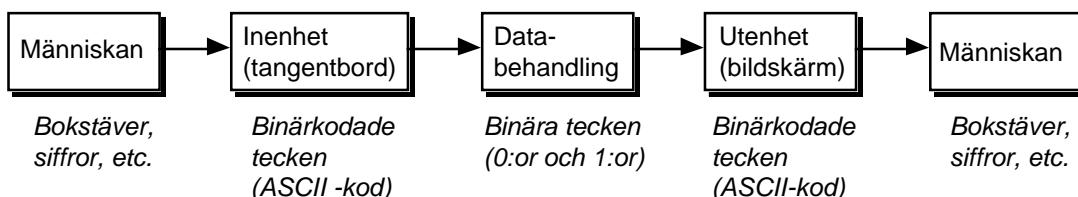
En *utenhet* tar hand om resultatet av beräkningarna. I en persondator är utenheten en bildskärm. I ett positionsservo blir utenheten motorn som styrs. I ett multiprocessorsystem kan utenheten vara ett gemensamt minne, där datat sedan används som indata till en annan processor. Den svarta pilen indikerar att det huvudsakliga dataflödet går från CPU:n till utenheten och den gråa att statusinformation kan gå från utenheten till CPU:n

I ett *minne* finns datorns *program* och *data*. Ett program består i ett antal instruktioner till CPU:n och bestämmer vilka slags beräkningar som skall göras. En CPU kan utföra enkla instruktioner av typen addera, subtrahera, flytta data mellan CPU:n och minnet, etc.

Program och data i en dator lagras internt i minnet som en sekvens av bitar, en *bit – binary digit*, är alltså en binär siffra. En binär siffra kan anta två värden: 0 eller 1. All informationsbehandling, d.v.s. beräkningar i en dator, sker alltså på en sekvens av bitar. I en dator består dessa sekvenser normalt av 8, 16, 32 eller 64 bitar. En grupp om 8 bitar brukar kallas en *byte*. Mikroprocessorn som vi skall använda i denna kurs är av fabrikatet ATMEL och har beteckningen ATmega16, denna processor bearbetar data med 8 bitars bredd och är en så kallad 8-bitarsdator. En vanlig persondator har ofta en mikroprocessor av fabrikatet Intel typ 80486 eller Pentium dessa är av typen 32-bitarsdatorer.

## 1.3 Information och betydelsen av information

I digitaltekniken och datortekniken är basenheten för information en bit, d.v.s. en binär siffra 0 eller 1. All annan information uttrycks som en kombination av bitar. Betydelsen är den tolkning vi ger informationen. Ordet *digital* kommer från latinets *digitus* som betyder finger, siffra.



*Figur 1.2: Information till/från en dator*

### ■ **Exempel 1.1: TOLKNING AV INFORMATION**

Vilken tolkning vi ger beror på sammanhanget informationen finns i. Till exempel kan bitsträngen 0010 0110<sub>2</sub> tolkas som:

- Tolkning 1 som heltalet  $38 = 0 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$ .
- Tolkning 2 som heltalet 26 (BCD-kodat tal) där varje grupp om 4 bitar bildar en decimal siffra.
- Tolkning 3 är som tecknet & tolkat enligt ASCII<sup>1</sup>-tabellen.

Av ovanstående inser vi att tolkningen av det som ligger lagrat i minnet ej är entydigt. Tolkningen av data i minnet måste bestämmas av den som programmerar.

**Slut exempel 1.1 ■**

---

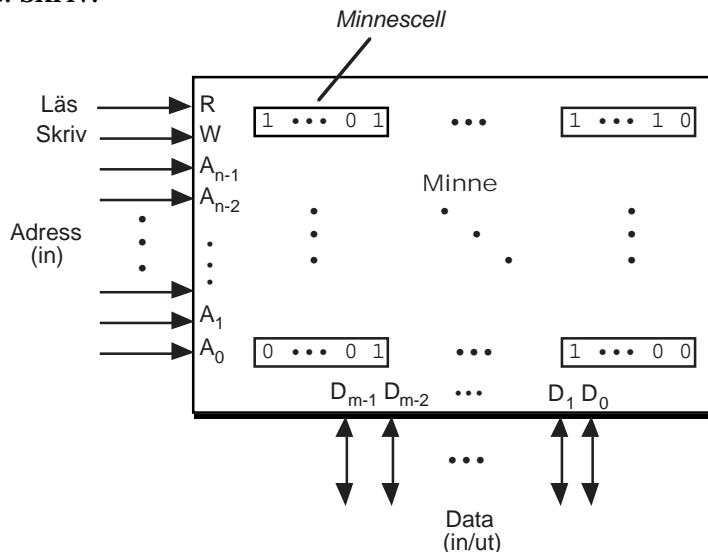
<sup>1</sup>American Standard Code for Information Interchange. Översatt till svenska: Amerikanska standardkoden för informationsutbyte.

## 1.4 Databuss, adressbuss och styrbuss

### 1.4.1 Modell av ett minne

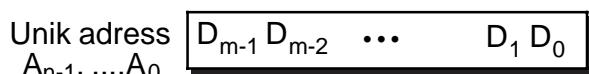
Vi har tidigare sagt att en dator arbetar mot ett *minne* där program och data finns lagrade. Ett datorsystem kan grovt sett sägas bestå av en CPU (processor) och ett minne. Vi behöver en funktionell modell som beskriver minnet sett från processorsidan.

Ett minne är en digital enhet där data kan lagras och från vilket data kan hämtas. Ett minne kan ur modelleringsynpunkt ses som uppbyggt av ett antal minnesceller. I varje minnescell lagras det ett *binärt ord*. *Ordlängden* i den generella minnesmodellen är på  $m$  bitar, vanligtvis är ordlängden 8, 16, 32 eller 64 bitar i ett datorsystem. Ordlängden är ofta definierad relativt det maximala antalet bitar som CPU-enheten kan arbeta med samtidigt. Däremot kan minnessystemet ha en annan ordlängd. Datat i minnet lagras i form av binära ord i minnescellerna. Det binära ord som finns i en minnescell kan avläsas via dataledningarna  $D_{m-1}, \dots, D_1, D_0$ . På analogt sätt kan ett nytt binärt ord skrivas in i en minnescell via dataledningarna  $D_{m-1}, \dots, D_1, D_0$ . Dataledningarna är dubbelriktade; de används som utgångar vid läsning av minnet och som ingångar vid inskrivning av nya binära ord i minnet. För styrning av en läsoperation respektive en skrivoperation finns två logiska styrsignaler:  $R$  som står för read d.v.s. läs och  $W$  som står för write d.v.s. skriv.



**Figur 1.3:** Minnesmodell

Associerat med varje minnescell finns det en unik adress, d.v.s. ingen minnescell har samma adress. För att peka ut (adressera) vilken minnescell som skall kunna nås via dataledningarna  $D_{m-1}, \dots, D_1, D_0$  finns det  $n$  stycken adressledningar  $A_{n-1}, \dots, A_1, A_0$ .



**Figur 1.4:** Minnescell

## 1.4.2 Hur ordnas byten i minnet?

Termerna *big-endian* och *little-endian* beskriver i vilken ordning en sekvens av bytes lagras i minnet.

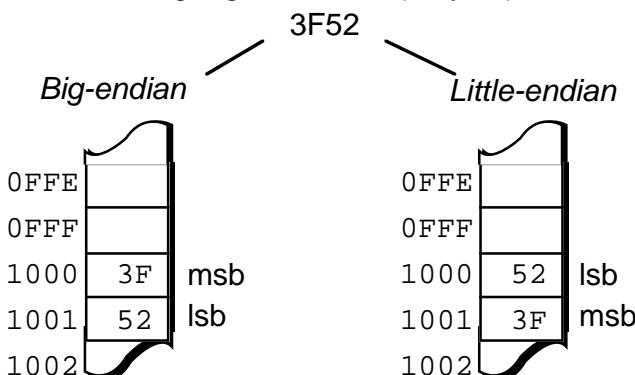
**Big-endian** är den ordning i vilken den mest signifikanta byten i sekvensen sparas först i minnet, med först i minnet menar vi den **lägsta adressen**. Denna princip kan ses som naturligt för oss i västerlandet där vi läser från vänster till höger, en textsträng eller ett tal lagras då tecken för tecken i den ordning som vi läser. Motorolas processorer, de flesta av de RISC-baserade datorerna och IBM:s stordator 370 använder alla big-endian som byte-lagringsprincip.

**Little-endian** är den ordning i vilken den minst signifikanta byten i sekvensen sparas först i minnet. Intels processorer typ 80486 och Pentium använder little-endian som byte-lagringsprincip.

### ■ Exempel 1.2: Lagring av det hexadecimala talet 3F52

Antag att det hexadecimala talet 3F52 som består av två byte, mest signifikanta byten är 3F och minst signifikanta byten är 52, skall lagras från adress 1000 i minnet. Beroende på vilken lagringsmetod som används blir resultaten enligt figuren:

Lagring av heltalet (2 bytes)

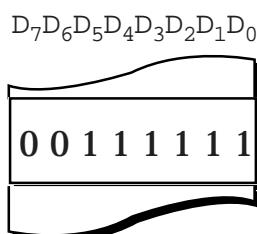


Figur 1.5: Big-endian och little-endian

Slut exempel 1.2 ■

Bitordningen i en byte är dock inte lika med byte-lagringsmetoderna. Bitordningen är normalt av typen big-endian. D.v.s. i varje byte kommer mest signifikanta biten först. I figuren som följer visas hur byten med hexadecimala värdet 3F lagras i minnet:

Bitordningen är big-endian



Figur 1.6: Bitordningen är normalt av typen big-endian

Orden big-endian och little-endian kan härlidas tillbaka till klassikern "Gulliver's resor" av Jonathan Swift. Big-endians var en partigrupp i det Lilliputianska samhället som knäckte sina ägg på den breda änden (primitivt sätt), dessa gjorde uppror mot kungen i Lilliputian som hade publicerat ett edikt (påbud) att alla skulle knäcka sina ägg på den smala änden. Little-endians var den partigrupp som lyste kungen och knäckte sina ägg på den smala änden.

### 1.4.3 Bussar

En *buss* är ett verktyg med vilken man kan överföra information. En buss är ett antal kommunikationslinjer, som grupperats efter funktion. En parallellbuss består av tre delbussar: databussen, adressbussen och styrbussen.

#### 1.4.3.1 Databussen

*Databussen* överför data mellan olika enheter. En databuss har dubbelriktad kommunikation, t.ex. så kan data gå från en enhet till mikroprocessorn och från mikroprocessorn till minnet.

#### 1.4.3.2 Adressbussen

*Adressbussen* används för att ange källan eller destinationen för signaler, som sänds via en annan buss eller signallinje. Adressbussen bär adresser. Normalt anger adressbussen ett register (minnescell/port) i en av systemets enheter, som skall användas som källa vid läsning av data eller destination vid skrivning av data.

Adressbussen börjar i mikroprocessorn och bär adresser till de enheter som är anslutna till databussen. Adressbussen börjar normalt i ett speciellt register i mikroprocessorn, *minnesadresseringsregistret*.

#### 1.4.3.3 Styrbussen

*Styrbussen – control bus* synkronisera systemets operationer. Den innehåller status- och styrinformation både från och till mikroprocessorn. Styrbussen bär synkroniserade signaler mellan processorn och alla de enheter som är anslutna till de olika bussarna. Vanliga signaler på en styrbuss är:

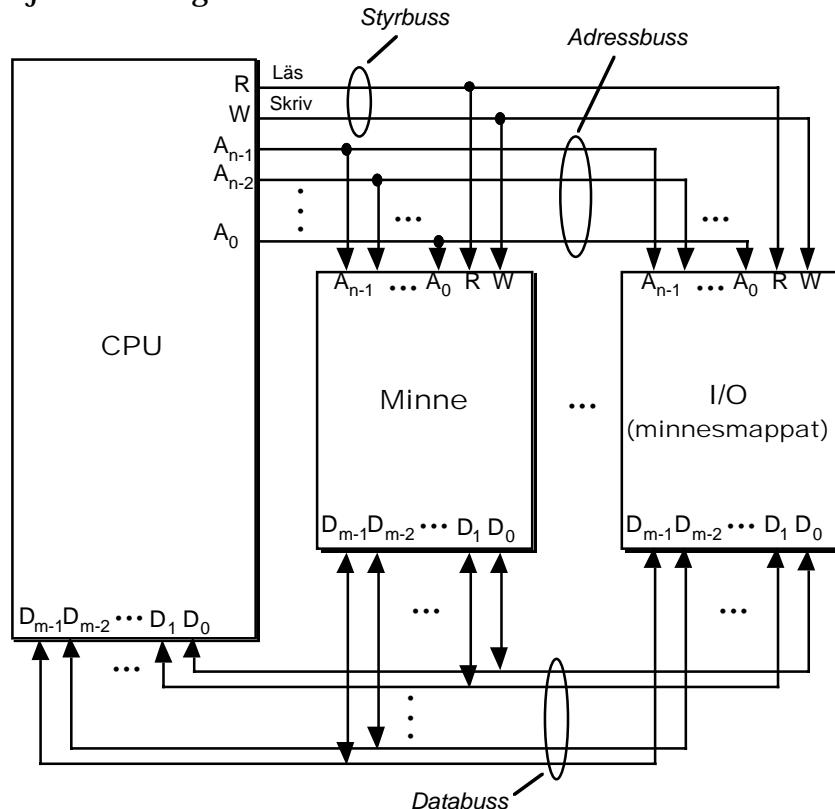
- Läs/skriv-signaler
- Klocksignaler
- Avbrottssignaler
- Återställningssignaler
- Kvitteringssignaler

#### 1.4.3.4 Inre och yttre bussar

En *inre buss* är en buss som finns direkt på chipet (jmfr. mikroprocessorns interna uppbyggnad). En *yttre buss* kan bestå av kablar för anslutning av specialenheter eller bakplan för anslutning av elektronikkort. I en modern persondator är bakplanet den del av datorkortet (moderkortet) som är bestyckat med kontaktdon för anslutning av kretskort.

### 1.4.3.5 En CPU ansluten till minnesenheter

Figuren som följer visar en generell bild av hur en CPU ansluts till minnesenheter.



*Figur 1.7: Anslutning av en CPU till minne*

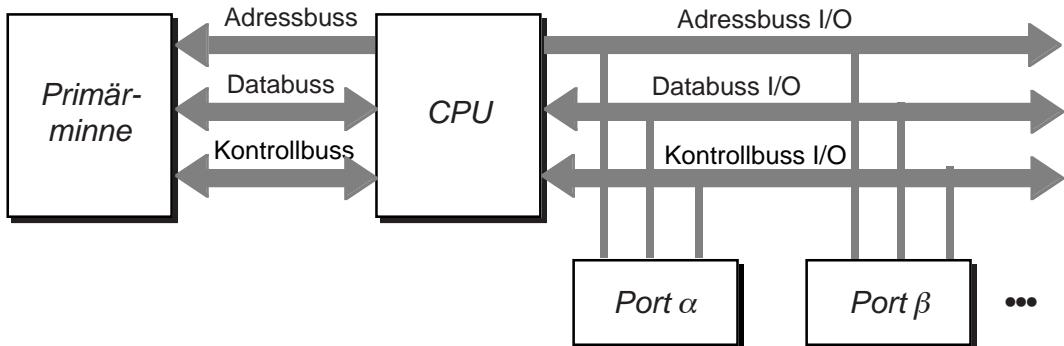
En skillnad görs mellan primärminne och sekundärminne. Med *primärminne* menas minne som är anslutet direkt till processorns adress- och databuss. *Sekundärminne* är minne som nås på annat sätt än att vara anslutet direkt till processorns adress- och databuss. Typiska sekundärminnesenheter är hårddiskar, CD-ROM enheter, floppydiskenheter, etc.

### 1.4.4 Anslutning av yttrre enheter till en processor

Yttrre enheter typ hårddiskar, analoga I/O-kort, digitala I/O-kort, etc. ansluts till en processor genom så kallade portar. En *I/O-port* kan ses som en ingång/utgång till datorsystemet för yttrre enheter som i sin tur står i kontakt med omvärlden, t.ex. en temperaturgivare ansluten till ett analogkort.

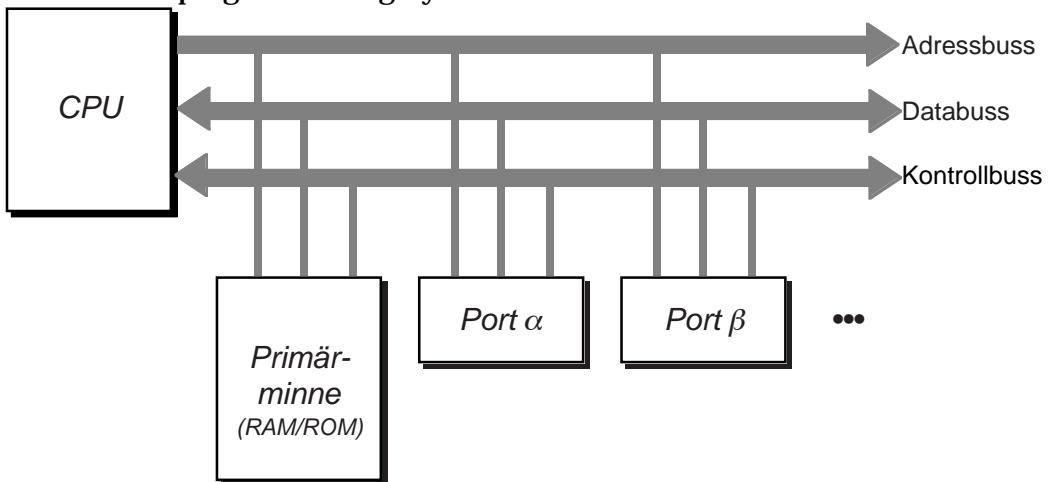
Yttrre enheter kan i princip anslutas på två olika sätt till en dator: Antingen via en speciell I/O-buss med en eget adresseringsområde eller finnas minnesmappad i processorns "vanliga" adressrymd för kod och data.

I figuren nedan visas ett datorsystem med en speciell I/O-buss för att ansluta yttrre enheter, det här faller det en parallel I/O-buss med det finns även seriella I/O-bussar. Intels processorer som sitter i en vanlig persondator har en I/O-buss.



Figur 1.8: Ytterenheter s. k. I/O-portar ansluts till en speciell I/O-buss

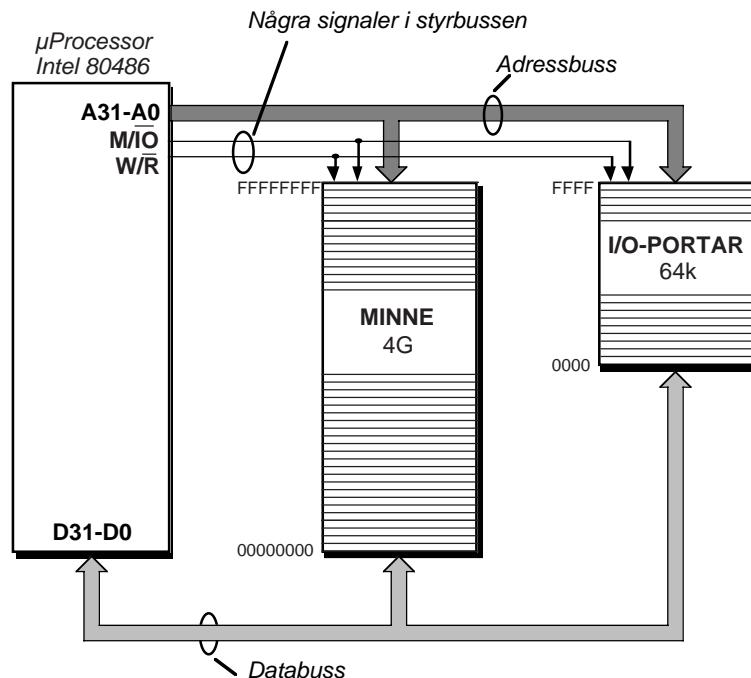
Den andra varianten som används bl. a. av Motorolas processorer låter alla ytterenheter anslutas till den buss som används för att kommunicera med primärminnet. Från processorns sida uppträder då I/O-portarna som vanligt minne, vilket kan ha en del fördelar sett ur programmeringssynvinkel.



Figur 1.9: I/O portarna är minnesmappade

## 1.5 Persondatorn

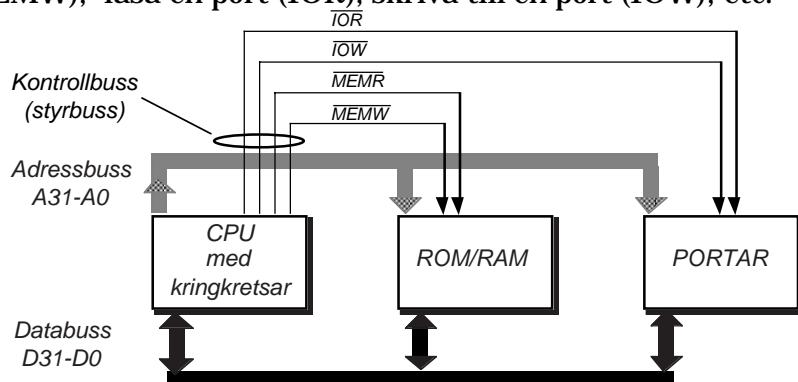
Eftersom detta är en kurs i mikrodatorteknik där vi använder en persondator för att utveckla programvara till vårt målsystem, som är ett datorkort bestyckat med en ATMEL-processor av typen ATmega16, kan det vara av intresse att även få en inblick i hur processorerna av fabrikatet INTEL 80486 och Pentium fungerar. För dessa typer av mikroprocessorer gäller att man gör skillnad på adressering av "vanligt" skriv-/läsminne och minne som tillhör en I/O-enhet. Minne som tillhör en I/O-enhet brukar benämnas *I/O-portar*. Val av adressering av minne respektive I/O-portar styrs av en pinne på mikroprocessorn med namnet M/I/O. Om denna pinne är låg eller hög, styrs av vilken typ av dataförflyttning instruktion som processorn utför.



**Figur 1.10:** Två typer av minne i ett Intelprocessorbaserat mikrodatorsystem

I en modern persondator finns ett antal olika *bussar*. Med en buss menas ett kommunikationssystem mellan processorn och olika digitala enheter såsom: skriv/läminne, I/O-portar, tangentbord, mus, etc. Bussarna är i princip av två typer: *parallelbuss* eller *seriell buss*.

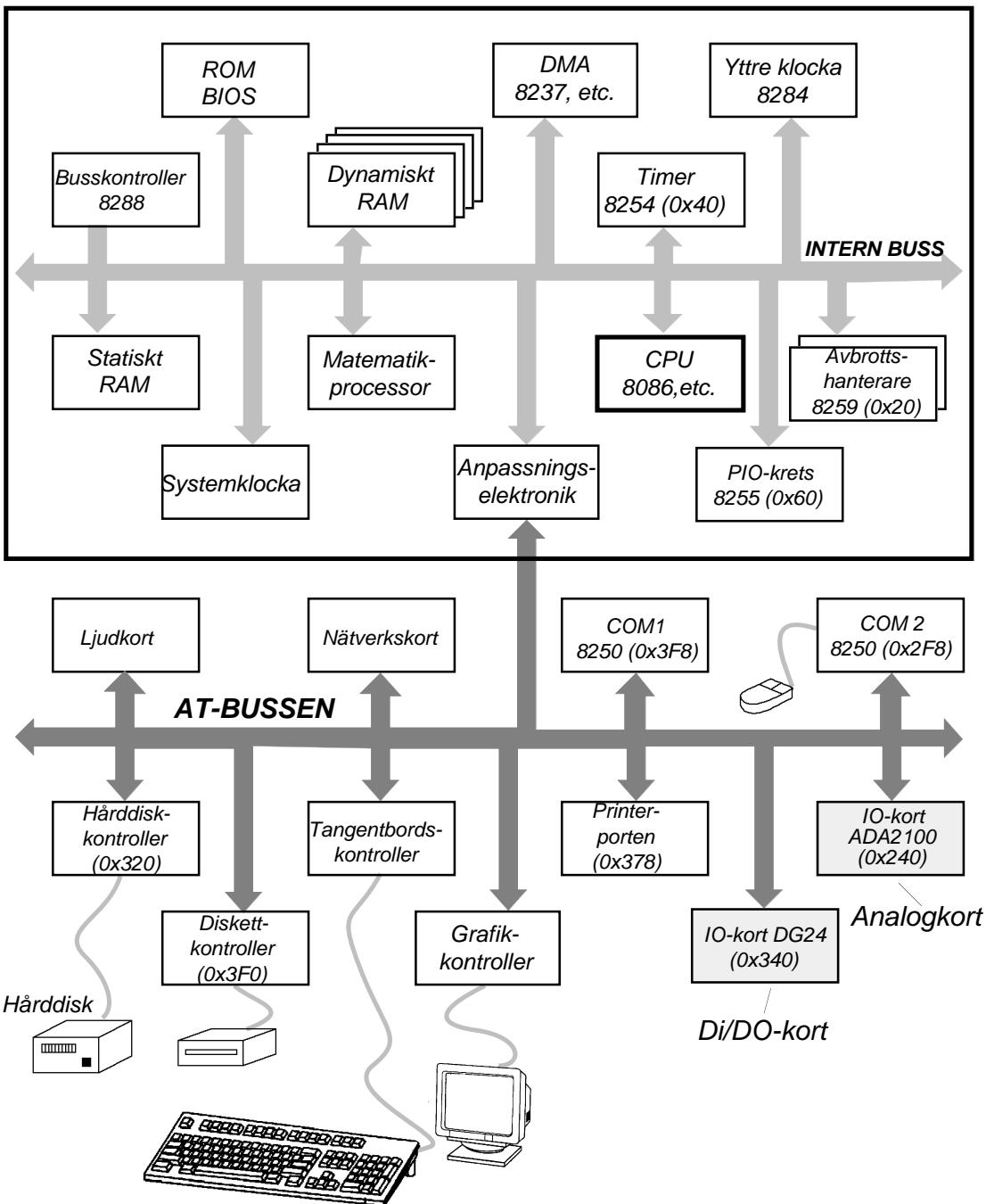
I figuren nedan visas ett blockschema för en parallelbuss i en persondator. Bussen består av tre delar: *Adressbussen* som används för att peka ut en minnescell/port. *Databussen* som används för att läsa in data till processorn för databehandling och sedan skriva resultatet till en minnescell/port. *Styrbussen* som innehåller olika styrsignaler som begär att läsa en minnescell (*MEMR*) eller skriva in data till en minnescell (*MEMW*), läsa en port (*IOR*), skriva till en port (*IOW*), etc.



**Figur 1.11:** Persondatorsystem

### Blockdiagram för en “äldre” persondator

I figuren nedan visas en principskiss för funktionsblock som finns i persondatorer av typen 8086-80486, Pentium, etc. Vilka delar som ligger på moderkortet respektive som instickskort till AT-bussen varierar. I en modern persondator finns samma funktionalitet, men det mesta är integrerat på moderkortet.

**MODERKORT**

Figur 1.12: Principskiss för en persondator typ äldre modell med Intelprocessor

**1.6 Maskin-, assembler- och högnivåspråk**

En *kompilator*<sup>1</sup> är ett program som översätter ett program skrivit i ett högnivåspråk typ C++, Pascal, Ada eller FORTRAN till ett maskinspråk. *Maskinspråk* är datorns "medfödda" (ursprungliga) språk och består av bitsträngar som tolkas av elektroniken i en dator. En typisk maskinspråksinstruktion i en dator av typen IBM persondator är:

<sup>1</sup>Kompilator kommer från det engelska ordet compile som betyder plocka ihop.

00000101

normalt skriver man detta i hexadecimal form vilket då ser ut på följande vis

05

Det första och mest primitiva programmeringsspråket var assembleringsspråk<sup>1</sup> och detta språk tillät programmeraren att skriva

**add**

istället för bitsträngen ovan. Detta sätt att komma ihåg instruktionen på ett enklare sätt brukar kallas *mnemonics* (minneskonst). Konsten att med vissa knep göra det lättare att komma ihåg saker brukar kallas *memoteknik*. Program för datorer uppbyggda på detta sätt med hjälp av mnemonics för instruktionsbitsträngar, brukar kallas *assemblerprogram*.



*Figur 1.13: Från högnivåspråk till maskinkod*

Från SIS<sup>2</sup> har jag hämtat följande ordförklaringar. Ett *assemblerspråk* är ett datorspråk där instruktionerna vanligen entydigt motsvarar maskininstruktioner, kan även vara makroinstruktioner. En *assemblerator* eller *assemblerare* är ett datorprogram som används för att översätta ett assemblerprogram till maskinspråk samt eventuellt länka underprogram. En *maskininstruktion* är en instruktion som kan tolkas av centralenheten (CPU-enheten) i den dator för vilken instruktionen är avsedd.

### ■ Exempel 1.3: I ett program shall operationen $x=y+5$ utföras

Programsatsen i högnivåspråket

$x = y + 5;$

blir översatt till assemblerspråk för en mikrodator av typen Intel, d.v.s. den processor som sitter i en vanlig persondator:

```

    mov    ax,word ptr [bp-6]
    add    ax,5
    mov    word ptr [bp-4],ax
  
```

och sedan översatt till maskinspråk i hexadecimal form:

```

    8B 46 FA
    05 00 05
    89 46 FC
  
```

**Slut exempel 1.3 ■**

<sup>1</sup>Kommer från engelskan och har betydelsen hoppsättning, montering.

<sup>2</sup>Standardiseringskommisionen i Sverige.

■ **Exempel 1.4:** Ett enkelt C++-program och dess maskinkod.

Följande C++-program

```
main()
{
    int x;
    int y = 5;

    x = y + 5;
    cout << x;
}
```

blir översatt till assemblerkod och till maskinkod (hexadecimal form):

Relativ

adress	Maskinkod	Assemblerkod	
0000		_main proc far	Assemblerkod är processorns abstraktionsnivå
0000 8C D0		mov ax,ss	
0002 90		nop	
0003 45		inc bp	
0004 55		push bp	
0005 8B EC		mov bp,sp	C-kod är högnivåspråkets abstraktionsnivå
0007 1E		push ds	
0008 8E D8		mov ds,ax	
000A 83 EC 06		sub sp,6	
	;	int x;	
	;	int y = 5;	
000D C7 46 FA 0005		mov word ptr [bp-6],5	
	;	x = y + 5;	
0012 8B 46 FA		mov ax,word ptr [bp-6]	
0015 05 0005		add ax,5	
0018 89 46 FC		mov word ptr [bp-4],ax	
	;	cout << x;	
001B 8B 46 FC		mov ax,word ptr [bp-4]	
001E 89 46 F8		mov word ptr [bp-8],ax	
0021 8B 46 F8		mov ax,word ptr [bp-8]	
0024 99		cwd	
0025 52		push dx	
0026 50		push ax	
0027 1E		push ds	
0028 68 0000e		push offset DGROUP:_cout	
002B 9A 0000000se		call far ptr @ostream@\$blsh\$ql	
0030 83 C4 08		add sp,8	
	;	}	

```

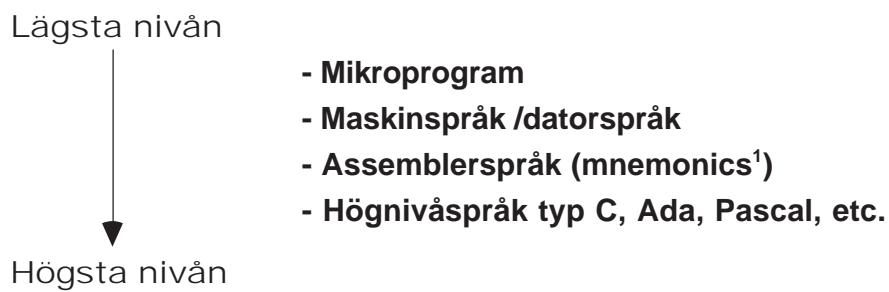
0033 8D 66 FE      lea    sp,word ptr [bp-2]
0036 1F             pop   ds
0037 5D             pop   bp
0038 4D             dec   bp
0039 CB             ret
                           _main  endp

```

Då ovanstående program skall exekveras av mikroprocessorn läses maskinkoden in till primärminnet från ett sekundärminne. Därefter laddas programräknaren i processorn med startadressen till programmet, varpå instruktionsexekveringen startar.

### **Slut exempel 1.4 ■**

På en ännu lägre nivå finns så kallade *mikroprogram*, bakom varje maskinspråksinstruktion finns normalt ett mikroprogram som ger styrsignaler till register och kombinatoriska nät i mikroprocessorn, vi är då nere på grind- och registernivån.



*Figur 1.14: Språkhierarki*

## **1.7 Uppgifter**

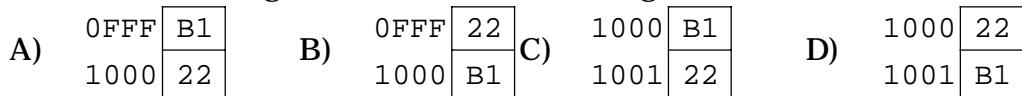
### **U1: Teorifrågor om datortekniska begrepp**

- a) Vad är *bit* en förkortning av?
- b) Ordet *digital*, vilken ursprunglig betydelse har det?
- c) Vad är basenheten för information i en dator? Vilken betydelse kan informationen i följande åtta bitar ges: 00111100 ?
- d) Rita en enkel modell av datorn.
- e) Vad är en buss för någonting i datorsammanhang?
- f) Vilka tre olika bussar finns det normalt i en generell dator?
- g) Vad menas med minneslagringsmetoderna big-endian respektive little-endian?
- h) Hur lagras det binära talet A287 från adressen 2F0E då lagringsmetoden är big-endian? Ordlängden i minnet är 8 bitar.
- i) Rita en enkel modell över ett minne.
- j) Vad menas med en I/O-port?
- k) Redogör för språkhierarkin i en dator.

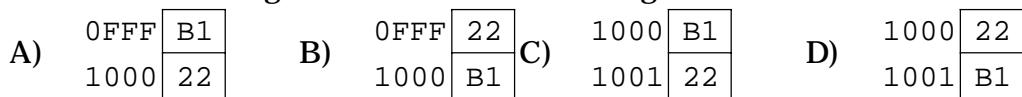
<sup>1</sup>mnemonics: minneskonst, minnesteknik

**U2: Big endian och little endian**

- a) Hur lagras 16-bitars talet 0xB122 i lagringsformatet little-endian i en dator från minnesadress 1000. Utgå ifrån att minnets ordlängd är 8 bitar.



- b) Hur lagras 16-bitars talet 0xB122 i lagringsformatet big-endian i en dator från minnesadress 1000. Utgå ifrån att minnets ordlängd är 8 bitar.

**1.8 Svar****U1: Teorifrågor om datortekniska begrepp**

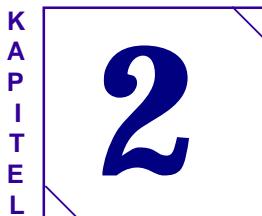
- a) binary digit, binär siffra
- b) Från latinets *digitus* som betyder siffra eller finger.
- c) -
- d) -
- e) -
- f) Adress-, data- och styrbuss.
- g) -
- h) -
- i) -
- j) -
- k) -

**U2: Big endian och little endian**

- a) D) 

1000	22
1001	B1
- b) C) 

1000	B1
1001	22



$$1 + 1 = 1$$

## Grundläggande digitalteknik

Följande kapitel är en kortfattad sammanfattning av de grundläggande begrepp och teorier inom digitaltekniken som vi bör känna till.

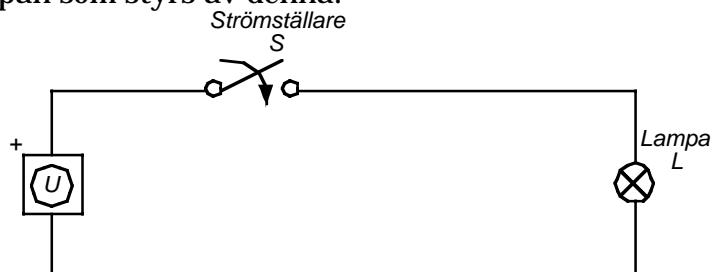
### 2.1 Logiska tillstånd - Objekt med två tillstånd

Många fysiska objekt karakteriseras av att de kan anta två tillstånd. En strömbrytare kan vara ÖPPEN eller SLUTEN; en lampa kan vara TÄND eller SLÄCKT; en ventil kan vara ÖPPEN eller STÄNGD; ett element kan vara PÅ eller AV. På liknande sätt karakteriseras ett påstående av att det kan vara SANT eller FALSKT. Vi har alltså tvåvärda kvantiteter eller med ett vanligare ord i dessa sammanhang *binära kvantiter*. Dessa kan representeras av *binära variabler* som är symboliska namn för kvantiteterna. Några vanligt förekommande binära kvantiter är:

Påstående	Ventil	Element.	Lampa
SANT (TRUE)	ÖPPEN (OPEN)	PÅ (ON)	TÄND (ON)
FALSKT (FALSE)	STÄNGD (CLOSED)	AV (OFF)	SLÄCKT (OFF)

*Figur 2.1: Binära kvantiter*

Följande elektriska krets med en strömställare har två tillstånd, både för strömställaren och lampan som styrs av denna:



S	L
ÖPPEN	SLÄCKT
SLUTEN	TÄND

*Figur 2.2: Enkel elektrisk krets av binär karaktär*

Istället för att använda orden ÖPPEN och STÄNGD om strömställaren  $S$  och SLÄCKT och TÄND om lampan  $L$  skall vi använda de symboliska namnen '0' respektive '1' för att beskriva de binära kvantiteterna. Kodning av ÖPPEN till '0' eller '1' är godtyckligt. Det som gäller är att hålla reda på vad '0' respektive '1' står för i de olika sammanhangen. Följande *sanningstabell – truth table* beskriver relationen mellan de två binära variablerna  $S$  och  $L$ . Vänsterkolumnen i sanningstabellen kan betraktas som en insignal kolumn och resultatet av dessa insignalers verkan på utsignalen visas i högerkolumnen.

$S$	$L$
0	0
1	1

Mellan  $S$  och  $L$  råder ett funktionellt samband:  $L = f(S)$ .

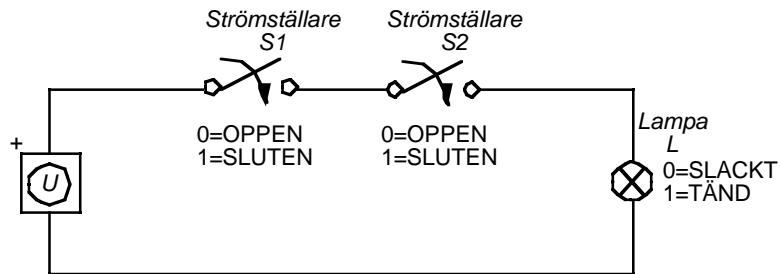
Oftast kodas SANT, PÅ, ÖPPEN ventil, SLUTEN strömbrytare, TÄND lampa till '1' och FALSKT, AV, STÄNGD ventil, ÖPPEN strömbrytare och SLÄCKT lampa till '0'.

Kretsen som visas i figuren som följer visar på två strömställare i serie som styr om en lampa skall vara TÄND eller SLÄCKT. För att lampan skall vara tänd gäller att  $S_1$  skall vara sluten OCH att  $S_2$  skall vara sluten. Detta kan sammanfattas i följande enkla formel:

$$L = S_1 \text{ OCH } S_2$$

eller på engelska som vi skall använda

$$L = S_1 \text{ AND } S_2$$

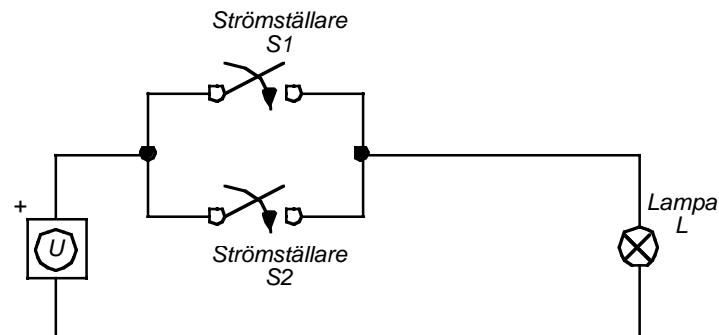


$S_1$	$S_2$	$L$
0	0	0
0	1	0
1	0	0
1	1	1

**Figur 2.3:** Två strömställare i serie

En ELLER-funktion kan åstadkommas med två strömställare parallellkopplade. För att lampan skall vara tänd gäller att  $S_1$  skall vara sluten ELLER att  $S_2$  skall vara sluten. Detta kan sammanfattas i följande enkla formel:

$$L = S_1 \text{ OR } S_2$$

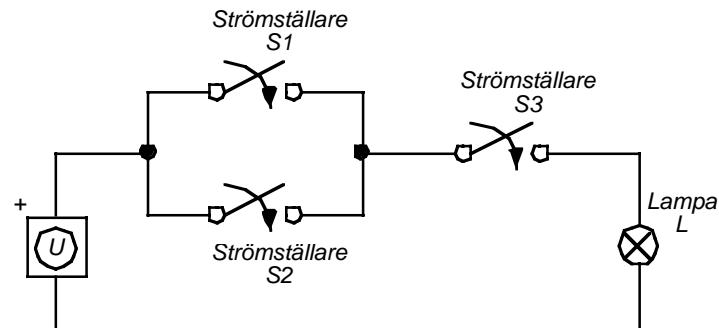


S1	S2	L
0	0	0
0	1	1
1	0	1
1	1	1

**Figur 2.4:** Två strömställare i en parallellkoppling

Kretsen i figuren nedan visar på en parallel och seriekopplad krets, dvs vi har en OR respektive AND-konfigurerings. Formeln för att lampan skall lysa blir:

$$L = (S_1 \text{ OR } S_2) \text{ AND } S_3$$

**Figur 2.5:** En parallel- och seriekonfigurerings

## 2.2 Boolesk algebra

Den Booleska algebran tillhandahåller ett skrivsätt och metoder för att systematiskt behandla system som arbetar med objekt som beskrivs av två binära kvantiter – två tillstånd. Ett binärt system beskrivs i den booleska algebran med hjälp av *konstanter*, *variabler* och *funktioner*. Detta fungerar på analogt sätt som den vanliga heltalsalgebran.

Boolesk algebra och *symbolisk logik* är i mångt och mycket detsamma. Logik är studiet av språkliga eller formelmässiga uttalanden samt reglerna för *slutledning*. En slutledning är inom logiken ett förlopp var vi ur en eller flera premisser härleder en eller flera slutsatser. Ett exempel på detta är: Premisserna "alla mänsklor är dödliga" och "Sokrates är en människa" som leder till slutsatsen att "Sokrates är dödlig".

Den booleska algebran definieras av konstanter, operationer och axiom. Ett axiom i

matematisk teori är en grundläggande utsaga som är sann. Utifrån dessa axiom kan olika räknelagar och bevis härledas. Den booleska algebran bygger på åtta axiom A1-A8, se figuren som följer:

**Tabell 2.1 Boolesk algebra - Definition**

<b>BOOLESK ALGEBRA - DEFINITION</b>		
<b>Konstanter</b>		<i>Symbolisk logik (jämförelse)</i>
0		falsk, false
1		sann, true
<b>Operationer</b>		
+		ELLER, OR
.		OCH, AND
-		ICKE, NOT
<b>Axiom</b>		
$0 + 0 = 0$	(A1)	false OR false = false
$1 \cdot 1 = 1$	(A2)	true AND true = true
$1 + 1 = 1$	(A3)	true or true = true
$0 \cdot 0 = 0$	(A4)	false AND false = false
$0 + 1 = 1 + 0 = 1$	(A5)	
$1 \cdot 0 = 0 \cdot 1 = 0$	(A6)	
$\bar{0} = 1$	(A7)	NOT(false)=true
$\bar{1} = 0$	(A8)	NOT(true)=false

Operationerna plus (+) och gånger (·) i den boolska algebran har inget att göra med vanlig addition eller multiplikation. Valet av dessa har gjorts med tanke på att axiomen A1-A6 liknar den vanliga algebrans axiom.

En Boolesk variabel karakteriseras av att den bara kan anta värdena 0 och 1. Ur axiomen A1 till A8 kan följande satser (teorem) härledas. Dessa kan sedan användas som räknelagar vid manipulering av Booleska uttryck.

**Tabell 2.2** Boolesk algebra - Räknelagar för en variabel**BOOLESK ALGEBRA -RÄKNELAGAR FÖR EN VARIABEL**

$$\begin{aligned} x + x &= x & (L1) \\ x \cdot x &= x & (L2) \\ x + \bar{x} &= 1 & (L3) \\ x \cdot \bar{x} &= 0 & (L4) \\ x + 1 &= 1 & (L5) \\ x \cdot 0 &= 0 & (L6) \\ x + 0 &= x & (L7) \\ x \cdot 1 &= x & (L8) \\ \bar{(\bar{x})} &= x & (L9) \end{aligned}$$

**Tabell 2.3** Boolesk algebra - Räknelagar för flera variabler**BOOLESK ALGEBRA -RÄKNELAGAR FÖR FLERA VARIABLER**

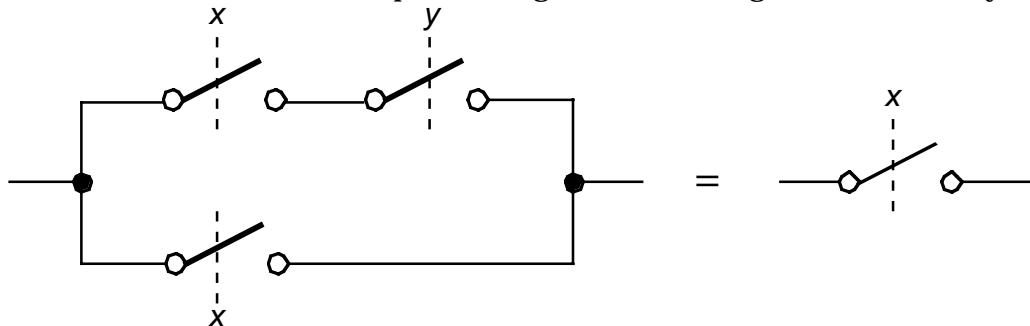
$(x + y) + z = x + (y + z)$	(L10) Associativa lagarna
$x(y \cdot z) = (x \cdot y)z$	(L11)
$x + y = y + x$	(L12) Kommutativa lagarna
$x \cdot y = y \cdot x$	(L13)
$x(y + z) = x \cdot y + x \cdot z$	(L14) Distributiva lagarna
$x + yz = (x + y)(x + z)$	(L15)
$x + x \cdot y = x$	(L16) Absorptionslagarna
$x \cdot (x + y) = x$	(L17)
$x \cdot y + \bar{x} \cdot z = x \cdot y + \bar{x} \cdot z + y \cdot z$	(L18) Konsensuslagarna
$(x + y)(\bar{x} + z) = (x + y)(\bar{x} + z)(y + z)$	(L19)
$\overline{(x + y)} = \bar{x} \cdot \bar{y}$	(L18) De Morgans Lagar
$\overline{(x \cdot y)} = \bar{x} + \bar{y}$	(L19)

Att räknelagarna med variabler inblandade stämmer kan enkelt visas med *perfekt induktion*. Genom att antalet konstanter i boolesk algebra är två, får vi ett ändligt antal kombinationer att prova. Räknelag L1 kan verifieras med följande tabell samt utnyttjande av axiomen.

x	V.L.	H.L.
	$x + x$	x
0	$0 + 0 = 0$	0
1	$1 + 1 = 1$	1

Många av ovanstående räknelagrar kan illustreras med hjälp av switchar som ett hjälpmittel för att förstå dessa.

Genom att studera följande figur där absorptionslagen  $x + x \cdot y = x$  illustreras med hjälp av switchar, inser man att den ena parallella grenen är onödig ur funktionell synvinkel.

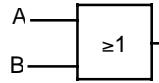
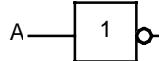
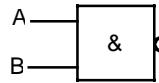
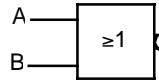
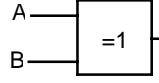


**Figur 2.6:** Absorptionslagen L16 tolkad med hjälp av switchar

## 2.3 Grindlogik

Tabellen som följer sammanfattar de vanligast förekommande grindvarianterna.

**Tabell 2.4** Logiska grindar

Funktion	Symbol	Boolesk	Sanningstabell															
AND		$C = A \cdot B$	<table border="1"> <tr> <td>A</td><td>B</td><td>C</td></tr> <tr> <td>0</td><td>0</td><td>0</td></tr> <tr> <td>0</td><td>1</td><td>0</td></tr> <tr> <td>1</td><td>0</td><td>0</td></tr> <tr> <td>1</td><td>1</td><td>1</td></tr> </table>	A	B	C	0	0	0	0	1	0	1	0	0	1	1	1
A	B	C																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$C = A + B$	<table border="1"> <tr> <td>A</td><td>B</td><td>C</td></tr> <tr> <td>0</td><td>0</td><td>0</td></tr> <tr> <td>0</td><td>1</td><td>1</td></tr> <tr> <td>1</td><td>0</td><td>1</td></tr> <tr> <td>1</td><td>1</td><td>1</td></tr> </table>	A	B	C	0	0	0	0	1	1	1	0	1	1	1	1
A	B	C																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
NOT		$B = \bar{A}$	<table border="1"> <tr> <td>A</td><td>B</td></tr> <tr> <td>0</td><td>1</td></tr> <tr> <td>1</td><td>0</td></tr> </table>	A	B	0	1	1	0									
A	B																	
0	1																	
1	0																	
NAND		$C = \overline{A \cdot B}$	<table border="1"> <tr> <td>A</td><td>B</td><td>C</td></tr> <tr> <td>0</td><td>0</td><td>1</td></tr> <tr> <td>0</td><td>1</td><td>1</td></tr> <tr> <td>1</td><td>0</td><td>1</td></tr> <tr> <td>1</td><td>1</td><td>0</td></tr> </table>	A	B	C	0	0	1	0	1	1	1	0	1	1	1	0
A	B	C																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$C = \overline{A + B}$	<table border="1"> <tr> <td>A</td><td>B</td><td>C</td></tr> <tr> <td>0</td><td>0</td><td>1</td></tr> <tr> <td>0</td><td>1</td><td>0</td></tr> <tr> <td>1</td><td>0</td><td>0</td></tr> <tr> <td>1</td><td>1</td><td>0</td></tr> </table>	A	B	C	0	0	1	0	1	0	1	0	0	1	1	0
A	B	C																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
Exklusiv OR		$C = A \oplus B$	<table border="1"> <tr> <td>A</td><td>B</td><td>C</td></tr> <tr> <td>0</td><td>0</td><td>0</td></tr> <tr> <td>0</td><td>1</td><td>1</td></tr> <tr> <td>1</td><td>0</td><td>1</td></tr> <tr> <td>1</td><td>1</td><td>0</td></tr> </table>	A	B	C	0	0	0	0	1	1	1	0	1	1	1	0
A	B	C																
0	0	0																
0	1	1																
1	0	1																
1	1	0																

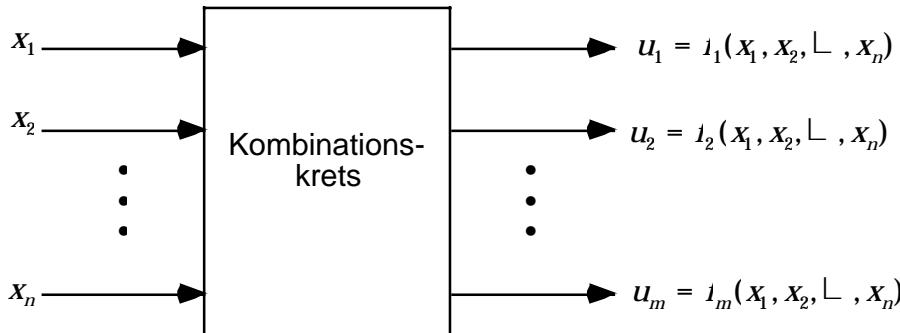
En EOR-funktion beskrivs ofta med hjälp operatorn  $\oplus$  för addition modulo 2. Exklusivt ELLER innebär att ett udda antal ettor på ingångarna ger en etta som utsignal, alla andra fall ger en nolla som utsignal. Sambandet mellan operatorn  $\oplus$  och Boolesk algebra är:

$$C = A \oplus B = A \cdot \bar{B} + \bar{A} \cdot B$$

I digitaltekniken finns två typer av kretsar: Kombinationskretsar och sekvenskretsar. I följande två stycken beskriver vi det karakteristiska för dessa två typer av kretsar.

Med hjälp av ovanstående grindar kan sedan olika typer av *kombinationskretsar – combinational circuits* byggas. En kombinationskrets definieras av att utgångsvärdet

$\mathbf{U}(t)$  vid en godtycklig tidpunkt endast beror av ingångsvärdet  $\mathbf{X}(t)$  vid samma tidpunkt. Detta innebär att vi har ett rent funktionellt samband mellan utsignaler  $\mathbf{U} = [u_1, u_2, \dots, u_m]$  och insignalerna  $\mathbf{X} = [x_1, x_2, \dots, x_n]$ , dvs  $\mathbf{U} = f(\mathbf{X})$ . Ett kombinatoriskt näts funktion (beteende) kan beskrivas utan att tiden behöver användas som variabel.

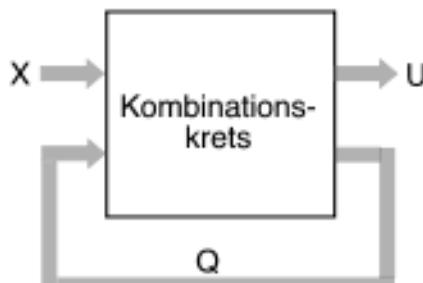


Figur 2.7: Kombinationskretsen

I ett kombinationsnät finns det ingen återkoppling tillbaka i nätet av en utsignal från en grind.

En *sekvenskrets* – *sequential circuit* kan ej beskrivas utan att tidsbegreppet införes. Ett sekvensnät har någon form av minne eller med andra ord ett *tillstånd*. En sekvenskrets kan betraktas som en kombinationskrets där ett antal signaler i kretsen återkopplas tillbaka och fungerar som insignalerna. Signalerna som återkopplas tillbaka är systemets tillståndsvariabler och betecknas normalt med  $Q = [q_1, q_2, \dots, q_r]$ .

Sekvenskrets=Kombinationskrets + återkoppling



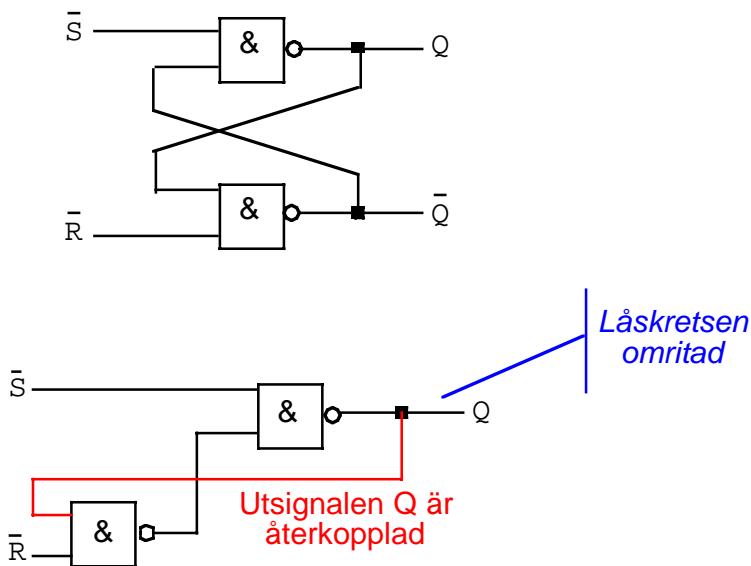
Figur 2.8: Sekvenskretsen

Insignalerna till ett kombinatoriskt nät inkommer i parallel form medan ett sekvensnät kan acceptera att insignalerna alternativt inkommer i seriell form. Om en sekvenskrets används så kan vi välja mellan att konstruera för en parallel databehandling eller en seriell databehandling.

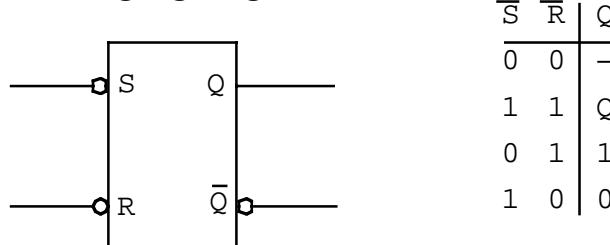
## 2.4 Vippor och latchar – minneselement

Den bistabila *läskretsen* utgör ofta en grund för att åstadkomma ett minneselement i digitala system. Ett minneselement är den enklaste formen av sekvenskretsar och används för att lagra en bit. Det engelska ordet för läskrets är *latch*.

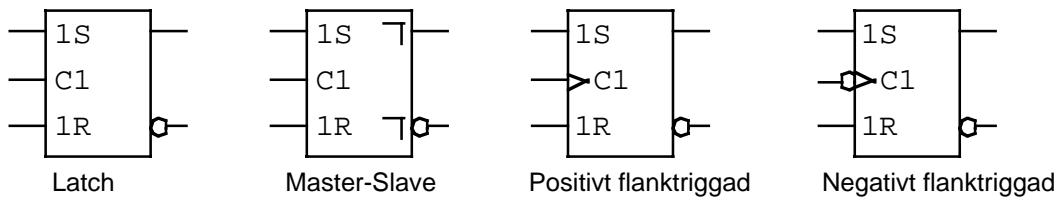
Läskretsen som följer i figuren är av typen  $\overline{\text{SR}}$ -latch, Set-Reset-latch med aktivt låga ingångar. Utsignalen är tillståndssignalen  $Q$ . För att skriva in en logisk 1:a, SET, skall insignalerna vara  $\overline{\text{SR}} = 01$  och för att skriva in en 0:a, RESET, skall de vara  $\overline{\text{SR}} = 10$ . I läsläge så är  $\overline{\text{SR}} = 11$  och läskretsen minns det som har skrivits in tidigare, d.v.s. i detta läge är utgångarna låsta och kan ej ändra värde.

**Figur 5.1:** Låskrets är den enklaste formen av en sekvenskrets, används för att lagra en bit

Observera att  $\overline{SR} = 00$  är en ogiltig insignal, varför?

**Figur 2.9:** IEC-symbol för låskretsen

Latchar och vippor av SR-typ finns i ett antal varianter. Skillnaden mellan en latch och en vippa är att alla förändringar på utgången från en vippa sker på en positiv eller negativ klockflank (C1-ingången). För en latch sker förändringarna så länge klocksignalen antingen är låg eller hög. SR-latchen är den mest förekommande asynkrona sekvenskretsen då den utgör en byggsten i statiska halvledarminnen för att lagra en bit.

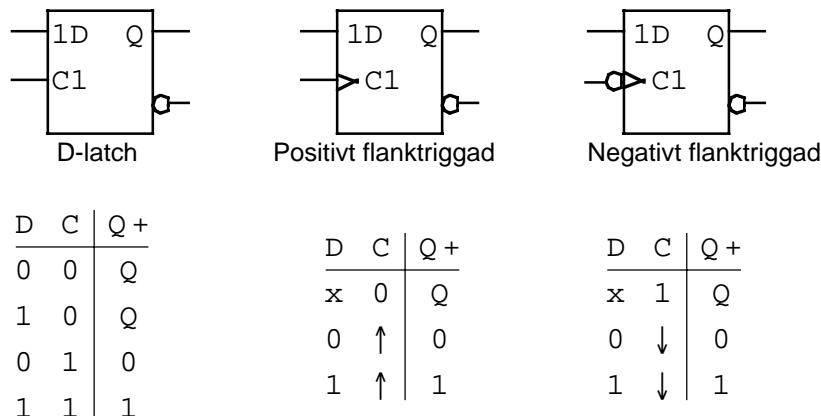


S	R	C	Q +	S	R	C	Q +	S	R	C	Q +	S	R	C	Q +
0	0	x	Q	0	0	Π	Q	x	x	0	Q	x	x	1	Q
1	0	0	Q	1	0	Π	1	1	0	↑	1	1	0	↓	1
1	0	1	1	0	1	Π	0	0	1	↑	0	0	1	↓	0
0	1	0	Q	1	1	Π	?	0	0	↑	Q	0	0	↓	Q
0	1	1	0					1	1	↑	?	1	1	↓	?
1	1	x	?												

**Figur 2.10:** Vippor och latchar av SR-typ

SR-latchen har en klockingång  $C$ . När klockingången är låg är utgångsvärdena frysta ( $Q^+ = Q$ ). Då klockingången är hög speglar utgången  $Q$  momentant vad som händer på ingången  $S$  (set) respektive  $R$  (reset).

För den positivt flanktriggade SR-vippan sker alla förändringar på utgången  $Q$  då en 0 till 1 övergång görs på klockingången. Utgångsvärdet bestäms då av värdena på insignalerna  $S$  respektive  $R$ .

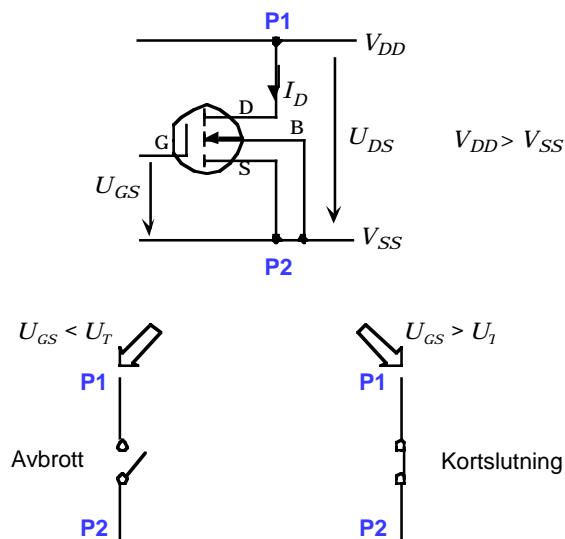
**Figur 2.11:** Vippor och latchar av D-typ

D-latchen har en dataingång  $D$  och en klockingång  $C$ . Ingången  $D$  är den som bestämmer tillståndet  $Q$  (utsignalen) och klocksignalen  $C$  har funktionen att öppna respektive stänga latchen. När klocksignalen  $C$  är hög så är latchen *transparent* då utgången  $Q$  i princip står i direkt förbindelse med ingången  $D$ . När klocksignalen går låg läses utgångsvärdet  $Q$  till det värde insignalen  $D$  har i detta ögonblick.

## 2.5 CMOS, den digitala kretsens byggsten

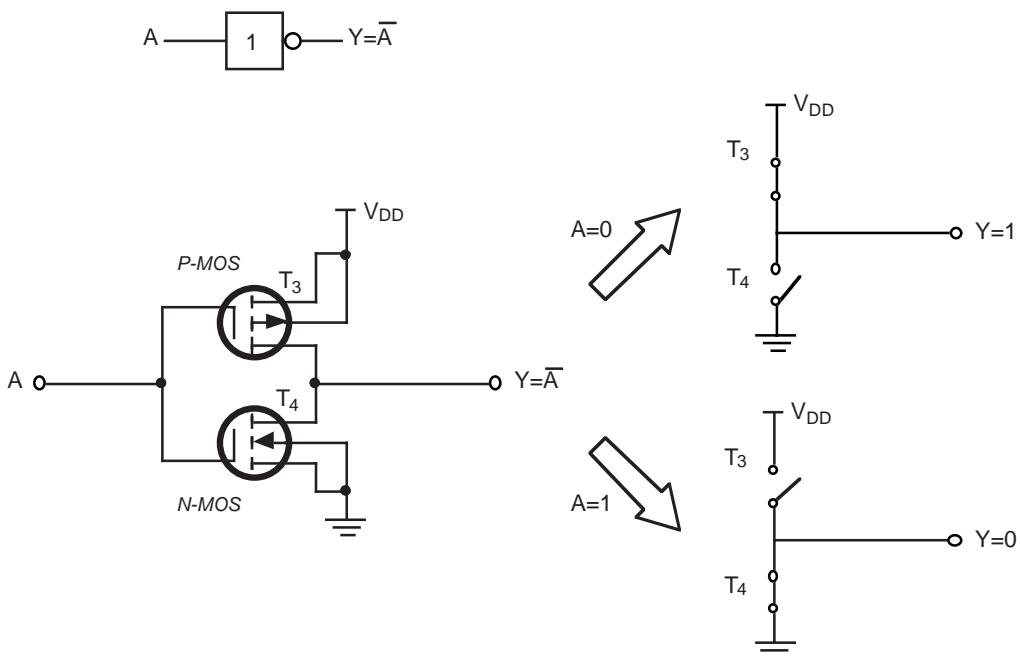
Att konstruera logiska grindar och vippor kan göras med många olika slags teknologier. Den mest förekommande teknologin i dagens läge är den som är baserad på användning av CMOS-teknologin.

För en beskrivning av de elektriska egenskaperna hos NMOS- och PMOS-transistorer härvisar vi till någon lärobok i digitalteknik. Vi kommer till stor del att modellera ledande transistorer som en kortslutning (resistans 0) och icke ledande transistor som ett avbrott (resistans  $\infty$ ). MOS-transistorerna är spänningstryrda och styrs med hjälp av gate-source-spänningen  $U_{GS}$ . När denna spänning är mindre än tröskelspannningen  $U_T$  är transistorn icke ledande och fungerar som en öppen switch. I det andra fallet när spänningen är större än tröskelspannningen, är transistorn ledande och fungerar som en sluten switch.



*Figur 2.12: Förenklad modell av NMOS-transistorn*

Grundkretsen i *CMOS-teknologin* är en inverterare som byggs upp av två transistorer, en PMOS-transistor (T3<sup>1</sup>) och en NMOS-transistor (T4), därav namnet **Complementary Metal Oxide Semiconductor**. En transistor i CMOS-teknologin fungerar som en **strömbrytare**, d.v.s. leder (kortslutning) respektive leder ej (avbrott). Utsignalen  $Y$  kan anta två logiska tillstånd 0 respektive 1. Det logiska tillståndet 0 motsvarar **Låg utspänning, 0V** och logiska tillståndet 1 motsvarar **Hög utspänning, matningspänningen  $V_{DD} = +5\text{ V}$** . Vid låg utspänning, logisk 0:a, leder transistor T4 (kortsluten) och transistor T3 är strypt (avbrott). Utgången  $Y$  står alltid i kontakt med jord. För att T4 skall leda krävs att insignalen  $A$  har en hög spänning, d.v.s. en logisk 1:a. Det omvänta gäller för hög utsignal.

*Inverterare***Figur 2.13:** CMOS-inverterare

<sup>1</sup>Valet av beteckningar har gjorts för att passa med CMOS-inverteraren med en 3S-utgång, se nästa avsnitt i kapitlet om bussproblematiken.

## 2.6 Bussproblematiken

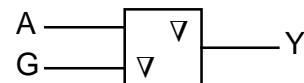
Bussproblemet består i att vi skall få olika digitala enheter att "prata" med varandra, över samma kommunikationslinjer. Alla enheter kan inte "prata" samtidigt, d.v.s. lägga ut data samtidigt på en databusslinje. Om två enheter lägger ut data samtidigt på en busslinje får vi en busskollision.

Det finns i princip två lösningar för att lösa bussproblemet: öppen-kollektorutgång respektive tristate-utgång. En 3-state-utgång har tre tillstånd som namnet anger, först de två logiska signalnivåerna för 0:a respektive 1:a och sen ett tredje *högimpedanstillstånd*. Högimpedanstillståndet kan ses som att utgången frikopplas (fritt svävande) från den bussledning den är ansluten till.

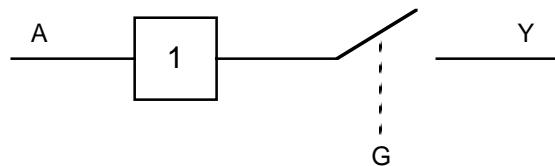
Funktionstabell

G	A	Y
1	x	Högohmigt
0	0	0
0	1	1

IEC-symbol för en 3S-grind



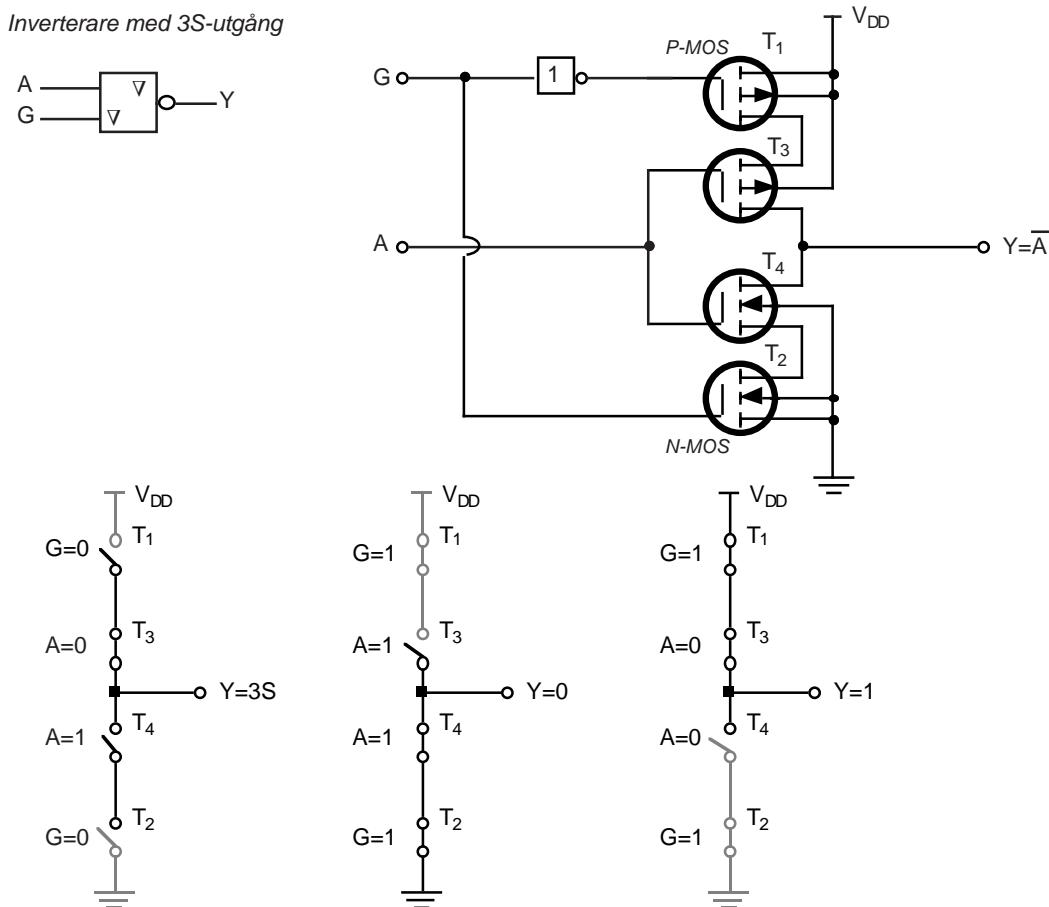
Principen för en 3S-grind



Figur 2.14: 3S-grind

Främkopplingen av en utgång sker med en extra styringång ofta kallad "enable" eller "tristate". Tristate-principen kan tillämpas på alla typer av grindar, då "3-state" endast innebär att en utgång främkopplas och därmed ej påverkar den på utgången anslutna signalledningen. Databussen är nästan alltid en 3-state-buss.

För att erhålla en CMOS-inverterare som har det tredje högimpediva tillståndet kan grundkretsen för en inverterare modifieras en aning och en extra styrsignal läggas till.

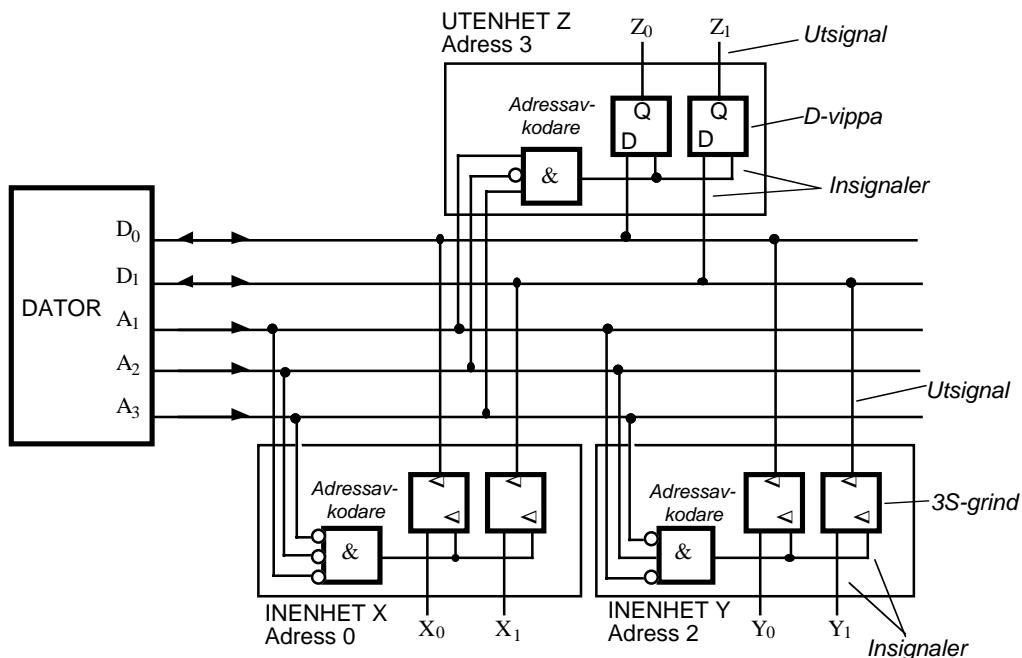


**Figur 2.15:** CMOS-inverterare med 3S-utgång

Styrsignalen G=H, logisk 1:a, medförför att kretsen fungerar som en inverterare. Transistorerna T<sub>1</sub> och T<sub>2</sub> leder.

Styrsignalen G=L, logisk 0:a, medförför att transistorerna T<sub>1</sub> och T<sub>2</sub> ej leder. Detta motsvarar avbrott och transistor T<sub>3</sub> står ej i förbindelse med matningsspänningen  $V_{DD}$  och transistor T<sub>4</sub> står ej i förbindelse med jord. Utgången Y ligger i detta fallet i ett tredje tillstånd, ett högimpediva tillstånd. En bra bild av det högimpediva tillståndet är att se utgången Y som en lös tamp som ej är anslutet till någonting, d.v.s ingen ström kan gå via utgången Y.

I figuren som följer visas ett förenklat datorsystem med en databuss bestående av 2 bussledningar och en adressbuss bestående av 3 bussledningar. Till systemet finns två inenheter anslutna och en utenhet, detta sett från processorn. Observera att databussen är dubbelriktad, datorn kan skriva data till minne och inenheter och läsa data från minne och inenheter. Inenheter, d.v.s. I/O-enheter som skall läsas av processorn är anslutna till databussen med 3-state-utgång. Utenheterna, I/O-enheter som processorn kan skriva till, är normalt någon form av minne eller register, i sin enklaste form är det några klockade D-vippor.



**Figur 2.16:** IO-enheter anslutna till en buss

## 2.7 Uppgifter

### U1: Verifiering av likheter (perfekt induktion)

Verifiera likheterna genom att för samtliga värden beräkna vänsterledet (VL) och högerledet (HL) och sedan jämföra dessa. Detta sätt att verifiera likheter brukar kallas *perfekt induktion*.

a)  $\bar{x}y(x + \bar{y})(x + \bar{y} + xy) = 0$

xy	VL	HL
00		
01		
10		
11		

b)  $x\bar{y} + \bar{x}y = \overline{(xy + \bar{x}\bar{y})}$

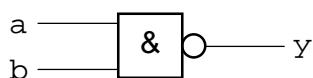
xy	VL	HL
00		
01		
10		
11		

### U2: Teorifrågor om logik, boolsk algebra, etc.

a) Avgör vilka av följande påståenden som är sanna (S) respektive falska (F):

- 1 Om  $x \bar{y} + \bar{x} y = 0$  så är  $x=y$ .
- 2 Om  $x=y$  så är  $x y + \bar{x} \bar{y} = 1$
- 3 Om  $x \bar{y} + \bar{x} y = x \bar{z} + \bar{x} z$  så är  $y=z$ .

b) Denna koppling realisering funktionen



- A)  $y = \bar{a} \cdot \bar{b}$
- B)  $y = \overline{\bar{a} \cdot b}$
- C)  $y = \bar{a} + \bar{b}$
- D)  $y = \overline{\bar{a} + b}$

c) Vilket uttryck utgör inversen till  $\overline{(a+b)+c}$

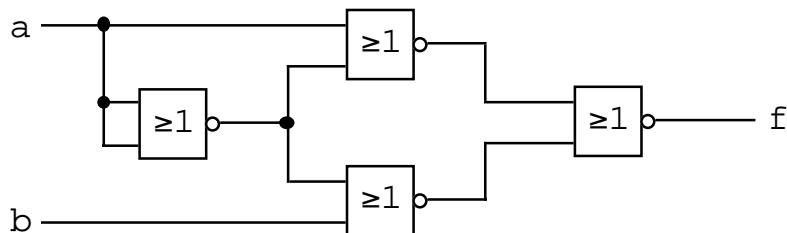
- A)  $(a+b)\bar{c}$
- B)  $ab\bar{c}$
- C)  $(a+b)c$
- D) någonting annat.

d) Inversen till  $(x + \bar{y})(\bar{z} + x \cdot y)$  är

- A)  $(\bar{x} + y)(z + \overline{x \cdot y})$

- B)  $x \cdot \bar{y} + \bar{z}(x + y)$
- C)  $\bar{x} \cdot y + \bar{z}(x + y)$
- D)  $\bar{x} \cdot y + z \cdot (\bar{x} + \bar{y})$

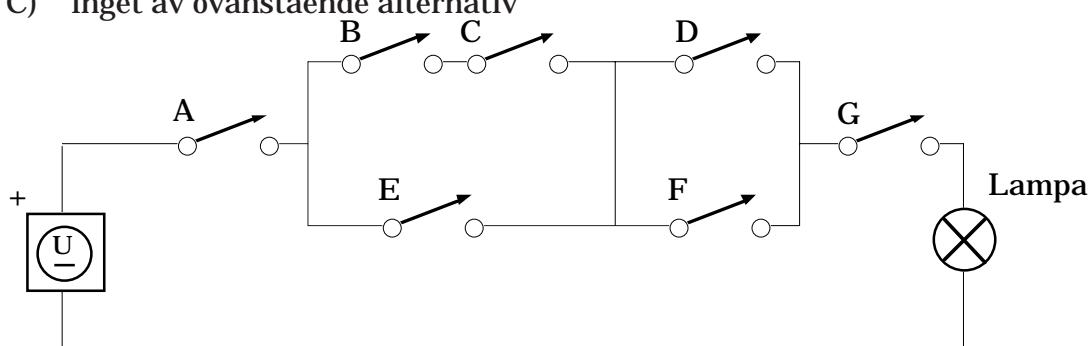
e) Vilken funktion realiseras nedanstående nät?



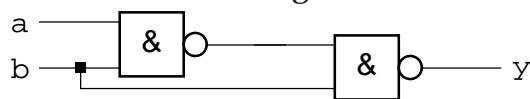
- A)  $f = a \cdot b$
- B)  $f = \bar{a} + b$
- C)  $f = \overline{a \cdot \bar{b}} + \bar{a} \cdot b$
- D)  $f = \overline{a \cdot b + \bar{a} \cdot \bar{b}}$

f) Vilket av följande algebraiska uttryck beskriver kretsen

- A)  $A \cdot (B \cdot C + E)(D + F) \cdot G$
- B)  $A \cdot (B \cdot C + E + D + F) \cdot G$
- C) inget av ovanstående alternativ



g) Vilken funktion realiseras nedanstående grindnät?



- A)  $y = \bar{a} \cdot b$
- B)  $y = \overline{\bar{a} \cdot b}$
- C)  $y = \bar{a} + b$
- D)  $y = \overline{\bar{a} + b}$

h) Ange sanningstabellen för en EOR-grind med två ingångar. Vilket Boolskt uttryck får detta? Förfar på analogt sätt med en EOR-grind med 3 ingångar.

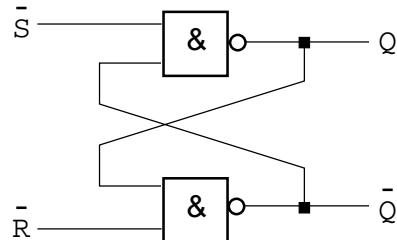
i) Visa hur en två-ingångars EOR-grind kan användas som en styrbar inverterare.

### U3: SR-latchar och SR-vippor

a) Den bistabila vippan är en av grundstenarna för att realisera minneselement i digitaltekniken. Redogör för den bistbila vippans funktion.

b) Vad blir utgångens Q:s värde vid följande sekvenser av insignalerna?

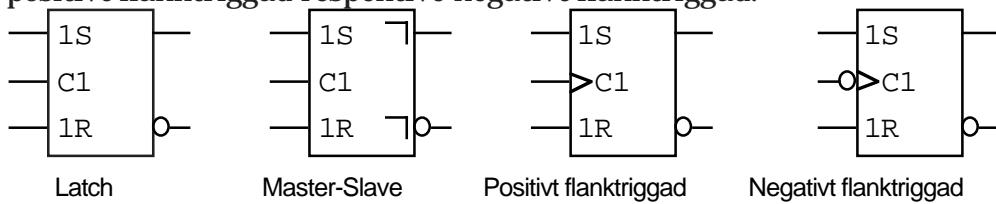
$\bar{S} \quad \bar{R}: \quad 10 \quad 11 \quad 01 \quad 11 \quad 01 \quad 11 \quad 11 \quad 10 \quad 11 \quad 10 \quad 11 \quad 11 \quad 10 \quad 00$



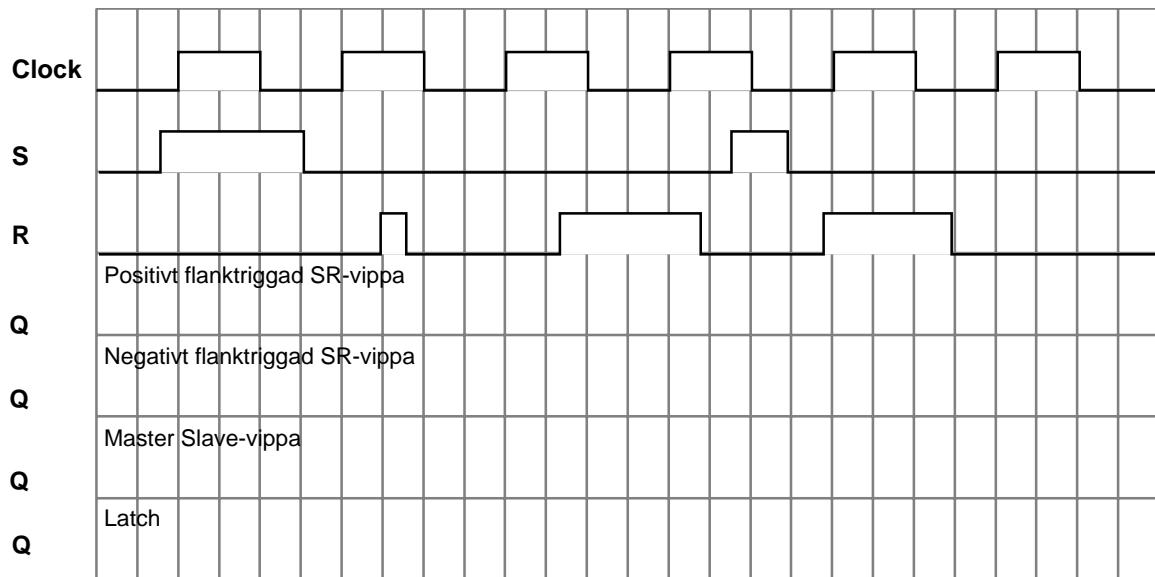
Figur 5.1: Låskrets

c) Bygg en SR-vippa med NOR-kretsar. Är denna vippa känslig för 1:or eller 0:or på ingångarna?

d) Rita i följande diagram utsignalen Q för en SR-latch och SR-vippor av typ master-slave, positivt flanktriggad respektive negativt flanktriggad.



Figur 5.2: Några olika typer av låskretsar och vippor av SR-typ

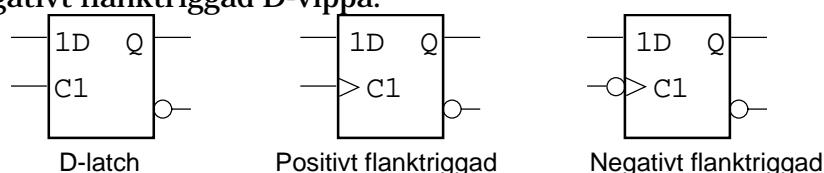


e) Förklara begreppen *förberedande ingång* och *verkställande ingång*. Vilka tidsmässiga relationer råder mellan en förberedande respektive en verkställande ingång. Utgå från en av SR-vipporna och redogör för vilka insignalerna som är förberedande

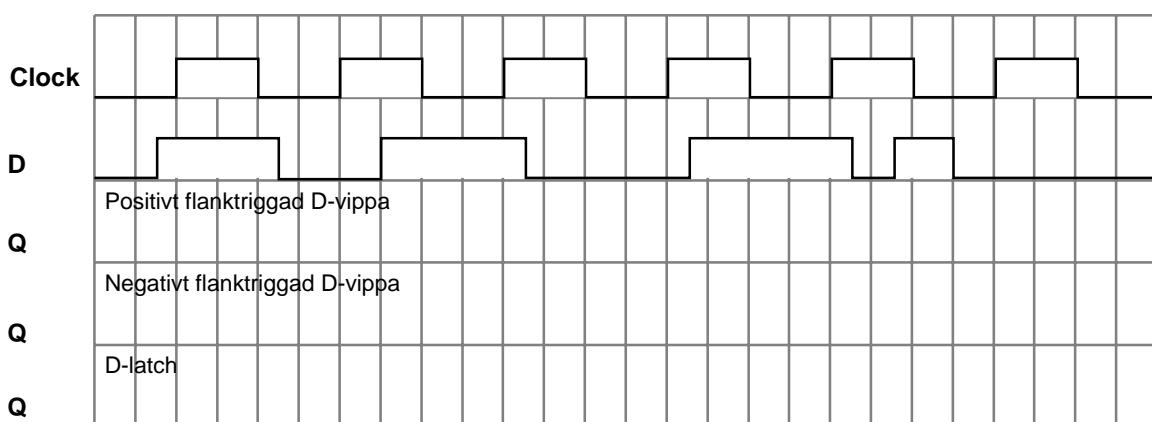
respektive verkställande.

#### **U4: D-latchar och D-vippor**

- a) En D-latch kan konstrueras med hjälp av en SR-latch och en inverterare. Visa med en bild hur detta görs.
- b) Rita i följande diagram utsignalen Q för en D-latch, positivt flanktriggad respektive negativt flanktriggad D-vippa.



**Figur 5.3:** D-latch och D-vippor



#### **U5: Teorifrågor om bussproblematiken**

- a) Redogör för hur en 3S-grind i CMOS-teknologi fungerar.
- b) Vad innebär bussproblematikproblemet?

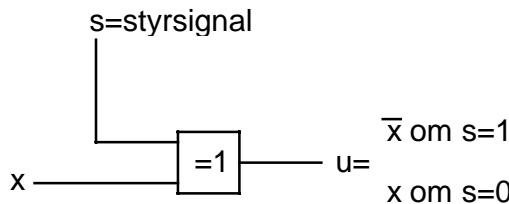
### **2.8 Svar**

#### **U1: Verifiering av likheter (perfekt induktion)**

#### **U2: Teorifrågor om logik, boolsk algebra, etc.**

- a) **1, 2**, och **3** är sanna.
- b) C)  $y = \bar{a} + \bar{b}$
- c) B)  $ab\bar{c}$

- d) D)  $\bar{x} \cdot y + z \cdot (\bar{x} + \bar{y})$   
e) B)  $f = \bar{a} + b$   
f) A)  $A \cdot (B \cdot C + E)(D + F) \cdot G$   
g) B)  $y = \bar{a} \cdot b$   
h)  
i)



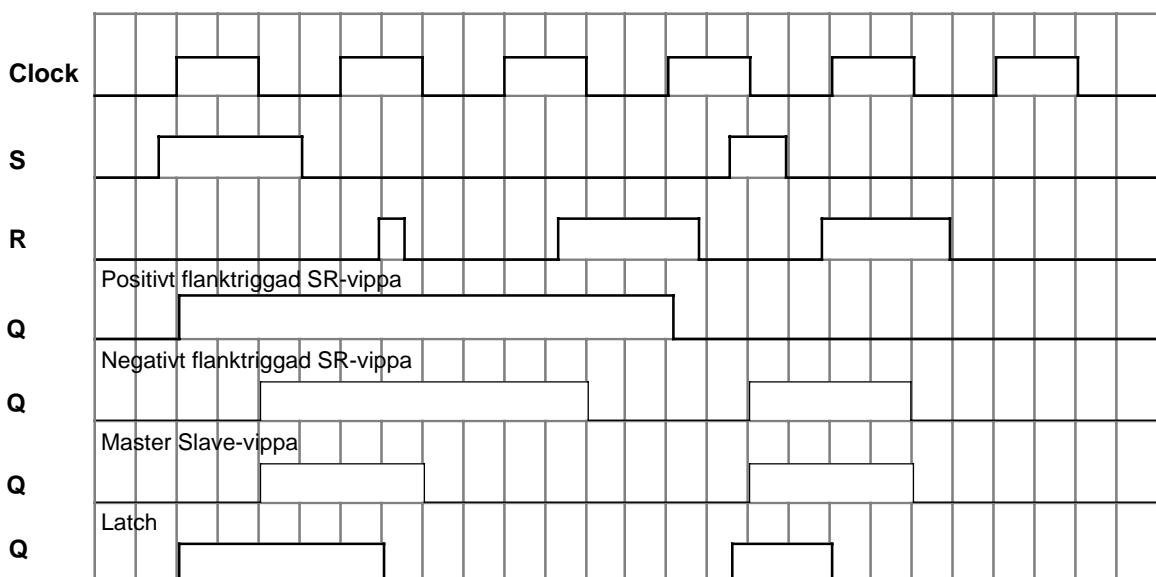
### U3: SR-latchar och SR-vippor

- a) -  
b) SR:    10    11    01    11    01    11    11    10    11    10    11    11    10    00  
Q       0      0      1      1      1      1      1      0      0      0      0      0      0      1

Observera att 00 är en otillåten kommbination, detta leder till att Q=1 och att  $\bar{Q}=1$ .

c) Känslig för 1:or.

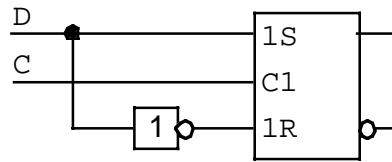
d)



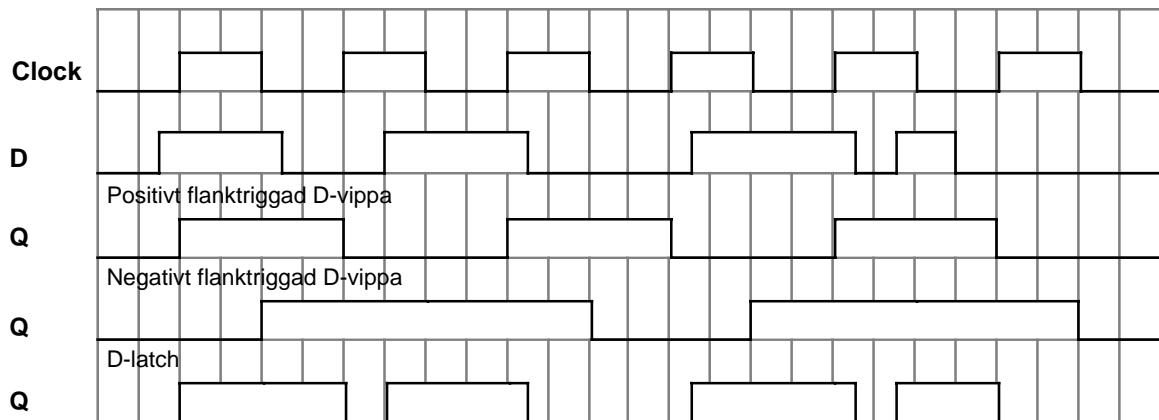
- e) Om vi utgår t.ex. från en klockad SR-vippa, måste insignalerna Set och Reset vara stabila innan klockflanken kommer på klockingången (C). Insignalerna Set och Reset brukar benämñas förberedande ingångar och klockingågen verkställande ingång. De förberedande ingångarna måste vara stabila innan den aktiva ingången påverkas.

### U4: D-latchar och D-vippor

a)



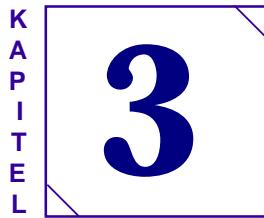
b)



## U5: Teorifrågor om bussproblematiken

a) -

b) Bussproblematiken är problemet med att kunna få flera digitala enheter att dela på samma kommunikationslinje. Bara en enhet åt gången är tillåten att sända (prata), övriga kan då ta emot (avlyssna) den som sänder.



$$01011010_2 = 5A_{16} = 90_{10}$$

# Talsystem, koder och binär aritmetik

---

## 3.1 Talsystem

De decimala, hexdecimala och binära talsystemen är *positionssystem*, där vikten av varje siffra bestäms av dess position i talet samt av talsystemets bas. Ett decimalt tal t.ex. 251.46 kan skrivas på så sätt att vi mer explicit anger varje siffras vikt i detta tal:

$$251.46 = 2 \cdot 10^2 + 5 \cdot 10^1 + 1 \cdot 10^0 + 4 \cdot 10^{-1} + 6 \cdot 10^{-2}$$

Basen  $B$  i det decimala talsystemet är 10, i det hexadecimala talsystemet 16 och i det binära talsystemet 2. Basen anger det antal sifversymboler som används i talsystemet. I det decimala talsystemet använder vi siffrorna 0,1,2, ..,9. Och i det binära siffrorna 0 och 1. Däremot i det hexadecimala som kräver 16 siffror använder vi siffrorna 0,1,2, .., 9 samt A,B, C, D, E och F, där A motsvarar 10, B motsvarar 11, .., och F motsvarar 15.

	B=10	B=16	B=2
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	9	1001	
10	A	1010	
11	B	1011	
12	C	1100	
13	D	1101	
14	E	1110	
15	F	1111	

**Figur 3.1:** Talen 0 till 15 i det binära respektive hexadecimala talsystemet

## 3.2 Omvandling mellan talsystem

Omvandling mellan det binära och hexadecimala talsystemet är lätt att göra, då en hexdecimal siffra motsvaras av fyra binära siffror. Det binära helttalet  $100111100011100_2$  kan således lätt skrivas om till ett hexadecimalt. Utgångspunkt för grupperingen är binärpunkten, åt vänster för heltalsdelen och till höger för fraktionsdelen

$$\underbrace{1001}_9 \quad \underbrace{1111}_F \quad \underbrace{0001}_1 \quad \underbrace{1100}_C \quad = 9F1C_{16}$$

Omvandling från det binära/hexadecimala till det decimala talsystemet är också förhållandevis rättframtid. Exemplet som följer visar hur du kan göra enligt handräkningsmetodiken:

$$\begin{aligned} (1010.0101)_2 &= 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} = \\ &= 8 + 2 + 0.25 + 0.0625 = 10.3125 \\ (1010.0101)_2 &= (10.3125)_{10} \end{aligned}$$

Omvandling från det decimala till t.ex. det binära talsystemet är ej lika lätt att göra. Detta på grund av att det per decimal siffra åtgår i genomsnitt 3.32 bitar (inget heltal).

Vid omvandling från ett decimalt tal t.ex.  $N_0.N_{-1} = 163.306$  till ett binärt/hexadecimalt tal, delas det decimala talet upp i en heltalsdel  $N_0 = 163$  och en fraktionsdel  $N_{-1} = 0.306$ . Rent praktiskt i vårt fall så gör vi förslagsvis omvandlingen från decimalt till hexadecimalt tal för att få så lite räkningsarbete som möjligt. Sedan är det lätt att ta fram det binära talet om så önskas.

Konvertering av heltalsdelen  $N_0$  till ett annat talsystem, sker genom upprepad heltalsdivision med basen  $B$  för det talsystem man vill konvertera till. Kvoten går vidare till nästa led i divisionen och resten blir en position i det omvandlade talet:

$$\begin{aligned} 163 / 16 &= 10 + \boxed{3}/16 \quad \text{Minst signifikanta siffran är 3.} \\ 10 / 16 &= 0 + \boxed{10}/16 \quad \text{Nest signifikanta siffran är A.} \end{aligned}$$

Konvertering av fraktionsdelen  $N_{-1}$  sker på analogt sätt fast genom upprepad multiplikation med basen  $B$  för det talsystem man vill konvertera till. I varje led så separeras heltalsdelen och fraktionsdelen. Heltalsdelen bildar en siffra i det konverterade talet och fraktionsdelen går vidare till nästa led:

$$\begin{aligned} 0.306 \times 16 &= \boxed{4} + 0.896 \quad \text{Mest signifikanta siffran är 4.} \\ 0.896 \times 16 &= \boxed{14} + 0.336 \quad \text{Näst mest signifikanta siffran är E.} \\ 0.336 \times 16 &= \boxed{5} + 0.376 \\ &\text{etc.} \end{aligned}$$

Konverteringen av fraktionsdelen brukar normalt ej gå jämnt upp, utan måste avbrytas när den upplösning man arbetar med har uppnåtts. Det hexadecimala talet blir:

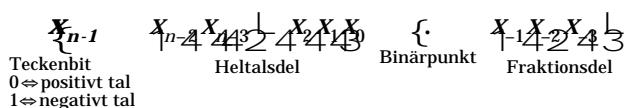
$$(A3.4E)_{16} = (10100011.01001110)_2$$

### 3.3 Binär representation av negativa tal

Det finns olika metoder för att koda negativa tal binärt. De två vanligaste metoderna är *tecken-belopp-kodning* respektive *2-komplementkodning*. För att representera heltalet används normalt 2-komplementkod. Anledningen till detta är att det blir väldigt lätt att hårdvarumässigt realisera subtraktionen i form av en addition. Tecken-beloppkod används vid kodning av mantissan i flyttal (implementering av reella tal binärt).

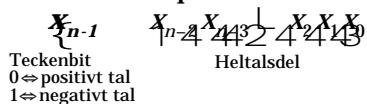
#### 3.3.1 Tecken-belopp-kodning

Vid tecken-belopp-kodning av negativa och positiva tal i det binära talsystemet införs en bit som fungerar som teckenbit. Teckenbiten placeras längst till vänster i bitmönstret och resterande bitar till höger representerar beloppet av talet. Det generella formatet vid teckenbeloppkodning blir enligt följande:



där  $x_i$  är en binär siffra med vikten  $2^i$ .

I de flesta tillämpningar saknas en fraktionsdel och det binära heltalet på  $n$ -bitarsformat blir på formen:



Talområdet för heltal i tecken-belopp-kod blir:  $-(2^{n-1} - 1) \leq x \leq +(2^{n-1} - 1)$

Det binära talet 1101 i 4-bitarsformat blir skriven som ett decimalt tal:

$-1 \cdot (1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0) = -5$ . I tabellen som följer visas fyra bitars tal  $x_3x_2x_1x_0$  med teckenbeloppkodning:

Decimalt tal	Binärt tal i tecken-belopp
-7	1111
-6	1110
-5	1101
-4	1100
-3	1011
-2	1010
-1	1001
0	1000
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

*Figur 3.2: Teckenbeloppkodning*

### 3.3.2 1- och 2-komplementkodning

Vid 1-komplementkodning eller 2-komplementkodning av negativa och positiva tal i det binära talsystemet införs en bit som fungerar som teckenbit, denna bit finns längst till vänster. Komplementkodning är ett sätt att representera negativa tal med positiva tal.

#### ■ Exempel 3.1: Negativa och positiva decimaltal i 2-komplementrepresentation

I detta exempel skall vi koda några decimaltal med 8 bitars binär representation. Negativa tal kodas med 2-komplementkodning. Talområdet blir med 8 bitars tal:

$$-2^{8-1} \leq d \leq +\left(2^{8-1} - 1\right)$$

$$-128 \leq d \leq +127$$

Om vi har ett positivt decimaltal  $d$  så representeras det binärt på vanligt sätt:

$$d = (10)_{10} = (00001010)_2 = \mathbf{x}$$

Ett negativt decimaltal däremot representeras binärt (2-komplementkodning) av det tal man får då man adderar det negativa talet med  $2^n = 2^8 = 256$ :

$$d = (-10)_{10} \Rightarrow (2^8 - 10)_{10} = (246)_{10} = (11110110)_2 = \mathbf{x}$$

Observera här att vi konverterar det decimala talet 246 till ett binärt tal representerat med 8 bitar. Decimaltalet -10 blir i 8 bitars 2-komplementkodning 11110110, talet -10 representeras alltså med talet 245.

Observera att det binära talet betecknas med  $\mathbf{x}$  och det decimala talet med  $d$ .

**Slut exempel 3.1** ■

Ett decimalt tal  $d$ , inom talområdet  $-2^{n-1} \leq d \leq +\left(2^{n-1} - 1\right)$  representeras av ett binärt tal  $\mathbf{x}$  med  $n$  bitar enligt följande regler då negativa tal 2-komplementkodas:

$$\mathbf{x} = \begin{cases} \text{bin}(d) & \text{då } x \geq 0 \\ \text{bin}(d + 2^n) & \text{då } x < 0 \end{cases}$$

där  $\text{bin}(d)$  betyder den binära representationen av  $d$ .

#### 1-komplement

Det som presenteras i det som följer är ett alternativt sätt än föregående att hantera 2-komplementkodningen av negativa decimala tal som skall representeras binärt. Negativa tal som kodas i 1-komplementkod bildas genom att till motsvarande positiva tal invertera samtliga bitar:

$$(5)_{10} = (0101)_2 \xrightarrow{\text{1-komplement}} (1010)_2 = (-5)_{10}$$

#### 2-komplement

2-komplementet till ett binärt tal  $\mathbf{x}$  bildas genom:

1. Bilda 1-komplementet till det binära talet  $\mathbf{x}$ , d.v.s. invertera alla positioner i  $\mathbf{x}$ .
2. Addera 1 till minst signifikanta positionen.

För att representera negativa tal binärt låter man det negativa talet representeras som 2-komplementet till motsvarande positiva tal, inklusive teckenbiten.

Decimalt tal	Binärt tal i 2-komplement	Binärt tal i 1-komplement
-8	1000	
-7	1001	1000
-6	1010	1001
-5	1011	1010
-4	1100	1011
-3	1101	1100
-2	1110	1101
-1	1111	1110
0	0000	0000
-0	0000	1111
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	0101
6	0110	0110
7	0111	0111

Figur 3.3: 2- och 1-komplementkod

För ett tal  $\mathbf{x}$  i 2-komplementkodning:

$$\mathbf{x} = \underbrace{x_{n-1}}_{\substack{\text{Teckenbit} \\ \text{Vikt } -2^{n-1}}} \underbrace{x_{n-4}x_{n-3}x_{n-2}x_{n-1}}_{\substack{\text{Heltalsdel} \\ \text{Vikt } 2^0, 2^1, 2^2, 2^3}}$$

gäller att teckenbiten har negativ vikt,  $-2^{n-1}$ , och de övriga positionerna  $i = 0, 1, \dots, n-2$  har positiv vikt,  $+2^i$ .

Talområdet för  $n$ -bitars heltal i 2-komplementkod blir:  $-2^{n-1} \leq d \leq +2^{n-1} - 1$ . Observera att talområdet ej är symmetriskt.

### ■ Exempel 3.2: 2-komplementkod

Antag att vi använder 8 bitars binära heltal. Hur representeras det decimala talet -117 binärt i 2-komplementkod? Vi skriver först 117 binärt:  $(117)_{10} = (01110101)_2$ . Efter detta skall vi bilda tvåkomplementet till det binära talet:

10001010 Ett - komplement

$$\begin{array}{r} 01110101 \\ \overset{2\text{-komplement}}{\Rightarrow} + \frac{00000001}{10001011} \end{array} \text{Addera 1 i lsb} \quad \text{Två - komplementet}$$

Svaret blir alltså  $(10001011)_2 = (-117)_{10}$

Slut exempel 3.2 ■

## 3.4 Binär aritmetik

### 3.4.1 Addition och subtraktion

#### 3.4.1.1 Addition av binära tal

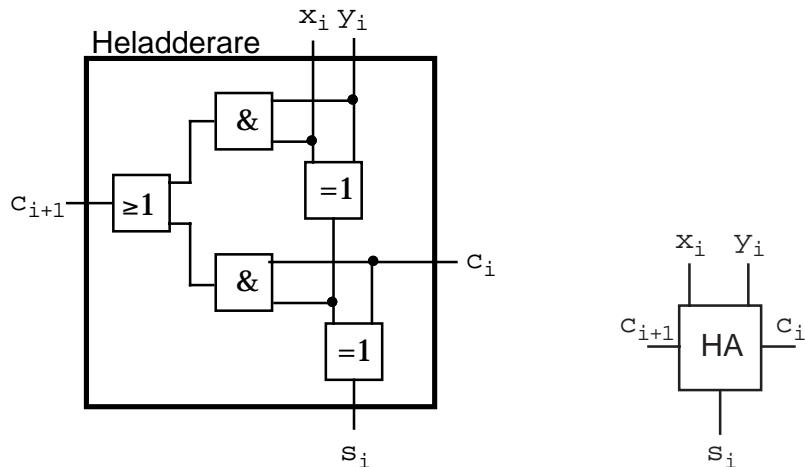
Addition av binära 2-komplemental sker på samma sätt som vanlig addition, med det förbehållet att vi arbetar i det binära talsystemet istället för det decimala. Additionen går från minst signifikanta sifferposition till mest signifikanta, d.v.s. från höger till vänster. Vi åskådliggör addition av en godtycklig position  $i$  i två binära tal  $\mathbf{x}$  och  $\mathbf{y}$ :

$$\begin{array}{r} c_i \\ \dots x_{i+1} \ x_i \ x_{i-1} \dots \\ + \dots y_{i+1} \ y_i \ y_{i-1} \dots \\ \hline \dots s_{i+1} \ s_i \ s_{i-1} \dots \end{array}$$

Summasiffran  $s_i$  i position  $i$  bildas som en funktion av  $f(x_i, y_i, c_i)$  där  $c_i$  är *minnessiffran - carry* som fås vid addition av position  $i-1$ . Från position  $i$  genereras också en minnessiffra  $c_{i+1}$  till position  $i+1$ :

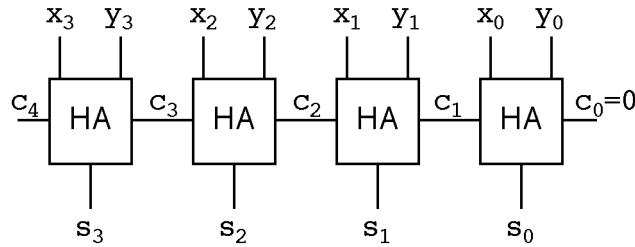
$$\begin{array}{cccccccccc} & & & & & & & & & \\ & c_i & & & & & & & & \\ & x_i & & & & & & & & \\ & + \frac{y_i}{c_{i+1} \quad s_i} & & & & & & & & \\ & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \\ & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \\ & + \frac{0}{0 \quad 0} & + \frac{1}{0 \quad 1} & + \frac{0}{0 \quad 1} & + \frac{1}{1 \quad 0} & + \frac{0}{0 \quad 1} & + \frac{1}{1 \quad 0} & + \frac{0}{1 \quad 0} & + \frac{1}{1 \quad 1} & \end{array}$$

En heladderare åstadkommes enkelt med några EOR-, OR och AND-grindar.



*Figur 3.4: Heladderare*

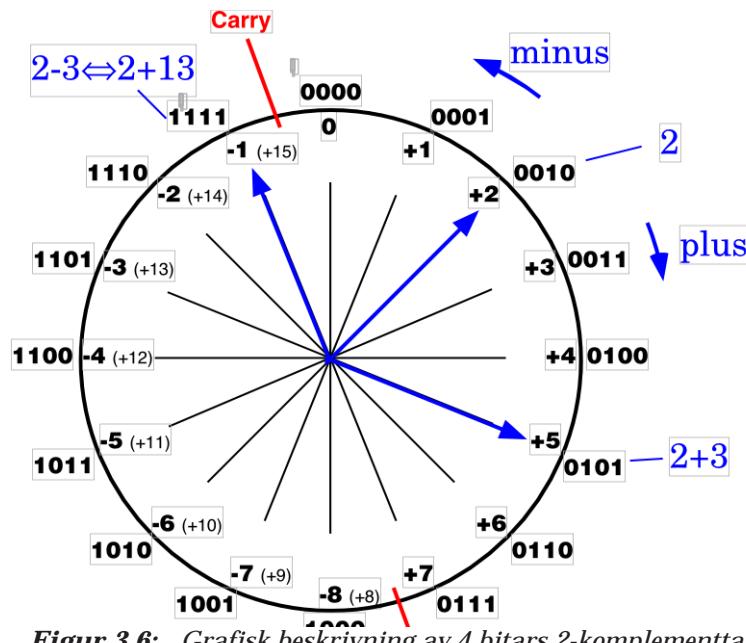
En adderare för binära tal i 4-bitarsformat på formen  $\mathbf{x} = [x_3 x_2 x_1 x_0]$  kan enkelt realiseras med heladderaren (HA) som byggsten:



Figur 3.5: Adderare för 4-bitars tal.

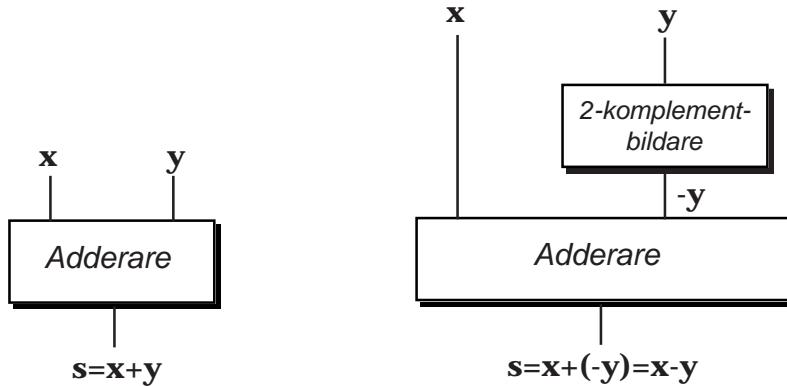
### 3.4.1.2 Grafisk addition och subtraktion av 4 bitars 2-komplementtal

I figuren som följer åskådliggör vi grafiskt fyra bitars 2-komplementtal, med fyra bitar kan vi representera totalt  $2^4 = 16$  heltal. I cirkulärgrafen visas kodning av negativa tal enligt 2-komplementkodning med fet stil, inom parantes visas kodning utan teckenbit. All databehandling i en dator görs på 0:or och 1:or, vilken betydelse vi ger för dem bestämmer vi som programmerare.



Figur 3.6: Grafisk beskrivning av 4 bitars 2-komplementtal

Addition av två tal t.ex.  $2+3$ , innebär att pilen med start i  $+2$  stegas fram 3 steg medurs. Subtraktionen  $2-3$  kan göras på analogt sätt, utgående från pilen i  $+2$  så stegas denna moturs 3 steg. Det som är intressant med 2-komplementrepresentationen är att en subtraktion kan utföras som en addition. Talet  $+3_{10} = 0011_2$  har tvåkomplementet  $1101_2$ , om vi betraktar detta som ett binärt positivt heltal är detta lika med 13. Subtraktionen  $2-3$  kan alltså uttryckas som en addition  $(2 + 13)_{10} = 15_{10} = 1111_2$ , d.v.s. med start i  $+2$  stegar vi 13 steg medurs, där  $1111_2$  är koden för  $-1$ .

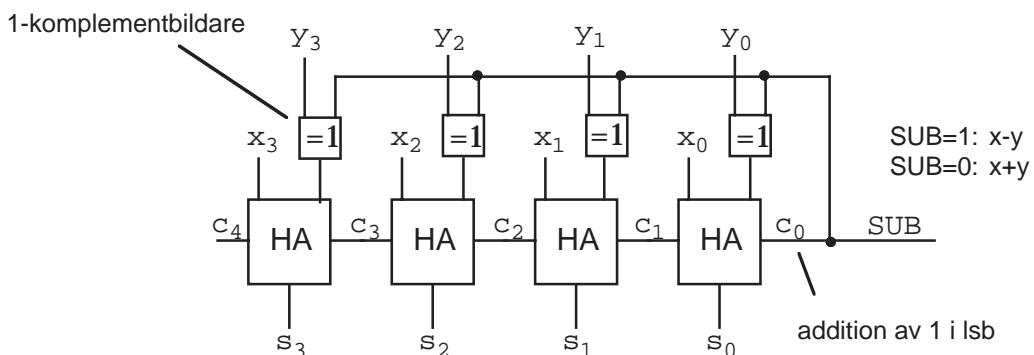


$$(2)_{10} - (3)_{10} = (2)_{10} + (-3)_{10} = (-1)_{10}$$

$$(0010)_2 - (0011)_2 = (0010)_2 + (1101)_2 = (1111)_2$$

**Figur 3.7:** Subtraktion som en addition vid 2-komplementkodning

En kombinerad additions/subtraktionskrets kan åstadkommas av additionskrets som kompletteras med extra logik i form av ett antal EOR-grindar och en styrsignal SUB som anger om subtraktion eller addition skall utföras. När styrsignalen SUB=0 fungerar kretsen som adderare och utför operationen  $x + y$  och när SUB=1 kommer subtraktionen  $x - y$  att utföras. Subtraktionen genomförs som en addition genom att 2-komplementet till  $y$  bildas då SUB=1.

**Figur 3.8:** Additions- och subtraktionskrets

Ovanstående figur visar på ett bra sätt varför 2-komplementkodning av negativa tal föredras framför tecken-beloppkodning. Hårdvaran för en kombinerad additions-/subtraktionskrets blir väldigt enkel, då den bygger på en additionskrets.

### 3.4.1.3 Spill – overflow

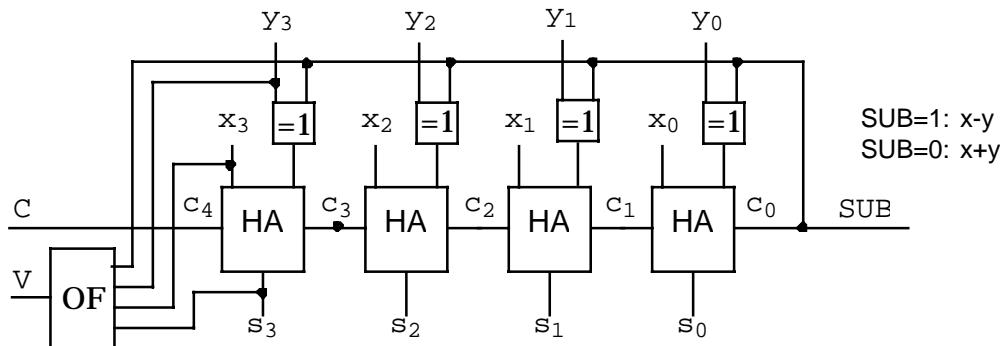
*Spill – overflow* i en additionskrets uppstår när resultatet av additionen faller utanför det givna talområdet. Talområdet bestäms av antalet bitar som används i de binära talen.

Om det binära talen ej har teckenbit har vi en enkel spillindikator i den mest

signifikanta minnessiffran i en additionskrets (se t.ex. adderaren för 4 bitars tal). Denna kommer vi att beteckna  $C$  som står för engelskans carry, som på svenska kan översättas till minnesiffra i detta sammanhang.

När de binära talen har en teckenbit och är kodade i tvåkomplementkod blir spillindikatorn lite mer komplicerad. Spill kan aldrig inträffa vid addition om talen har olika tecken. På analogt sätt kan spill aldrig inträffa vid subtraktion om talen har samma tecken. Om vi studerar additionskretsen enbart, kan spill inträffa när vi adderar två negativa eller två positiva tal.

I figuren som följer visas en fyra bitars additions-/subtraktionskrets med två flaggsignalen  $C$  och  $V$ . Spillindikatorn  $V$  används för att detektera spill för tal i tvåkomplementkod, flaggans namn kommer från  $V$ :et i oOverflow (spill på svenska). Carrysignalen  $C$  kan användas på flera olika sätt bl. a. som spillindikator för binära tal utan teckenbit.



**Figur 3.9:** Fyra bitars additions/subtraktionskrets med en spillindikator  $V$

Spillflaggorna kan också bildas på följande sätt med logiskt AND och logiskt OR på de ingående teckenbitarna. Följande formler gäller vid addition

$$C = x_3 \cdot y_3 + y_3 \cdot \bar{s}_3 + x_3 \cdot \bar{s}_3$$

$$V = x_3 \cdot y_3 \cdot \bar{s}_3 + \bar{x}_3 \cdot \bar{y}_3 \cdot s_3$$

och följande vid subtraktion

$$C = \bar{x}_3 \cdot y_3 + y_3 \cdot s_3 + \bar{x}_3 \cdot s_3$$

$$V = x_3 \cdot \bar{y}_3 \cdot \bar{s}_3 + \bar{x}_3 \cdot y_3 \cdot s_3$$

### 3.5 BCD – Binary Coded Decimal

BCD som står för Binary Coded Decimal är en binär kod för att representera numeriska tal. Varje decimal siffra representeras av 4 bitar. Kodens brukar ibland även kallas 8421-kod efter positionsvikterna. De decimala siffrorna kodas som 4 bitars positiva binära heltal enligt tabellen:

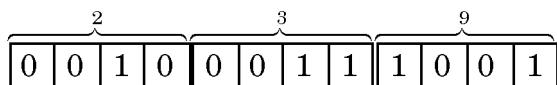
Decimalt tal	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

**Figur 3.10:** BCD-tabell

Observera att koderna 1010 till 1111 används ej och är ogiltiga.

### ■ **Exempel 3.3: BCD-kodning**

Det decimala talet 239 blir i BCD-kod:



**Slut exempel 3.3** ■

## 3.6 ASCII

ASCII är en förkortning av American Standard Code for Information Interchange, d.v.s. Amerikanska standardkoden för informationsutbyte. Standarden definierar en 7 bitars kod för att representera det latinska alfabetet, numeriska tecken, specialtecken, etc. binärt. Med 7 bitar kan totalt 128 tecken kodas. ASCII är konstruerat för att användas vid informationsutbyte mellan utrustningar av olika fabrikat. Den används t.ex. som standard vid digital kommunikation över en telefonlinje. Normalt läter man koden vara på 8 bitar, som är lika med en byte, och är enklare att hantera av en dator. Detta gör att 128 extra teckenkoder erhålls, dessa kan användas på olika sätt bl. a. för specialtecken och grafiska tecken.

### 3.6.1 Tabellen

ASCII-tecken finns i både amerikansk och svensk standard. I tabellen nedan ges alla ASCII-tecken. *Synliga tecken* är sådan som finns på tangentbordet dessa har heltalsvärdet mellan 32 och 126 (0x20 till 0x7E). Tecken med ordningsvärdet mindre än eller lika med 31 (0x1F) är så kallade *styrtecken* (osynliga tecken).

	O	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00 NUL	01 SOH	02 STX	03 ETX	04 ETO	05 ENQ	06 ACK	07 BEL	08 BS	09 HT	10 LF	11 VT	12 FF	13 CR	14 SO	15 SI
1	16 DLE	17 DC1	18 DC2	19 DC3	20 DC4	21 NAK	22 SYN	23 ETB	24 CAN	25 EM	26 SUB	27 ESC	28 FS	29 GS	30 RS	31 US
2	SP	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Figur 3.11: ASCII-tabellen (amerikansk standard)

	B	C	D
5	91	92	93
	Ä	Ö	Å

	B	C	D
7	123	124	125
	ä	ö	å

Figur 3.12: Korrigering av den Amerikanska standarden av ASCII-tabellen för att få med specifikt svenska bokstäver

### 3.6.2 ASCII-koder och C-programmering

För att kunna ange till exempel ett ny rad tecken som har heltalsvärdet 10 (se ASCII-tabellen) som ej finns med som ett tecken på tangentbordet brukar *escape-sekvenser* användas. Som escape-tecken används backslash \, detta innebär att tecknet som kommer efter skall tolkas annorlunda jämfört med dess vanliga betydelse. Så ett ny rad tecken skrivs i C som '\n', observera att detta är bara ett enda tecken fast det skrivs med 2 tecken.

Teckennamn	Skrivs i C	Heltalsvärde
null	'\0'	0
backspace	'\b'	8
tab	'\t'	9
newline	'\n'	10
formfeed	'\f'	12
carriage return	'\r'	13
double quote	'\"'	34
single quote	'\''	39
backslash	'\\'	92

Figur 3.13: Användning av escape-sekvenser för att nå vissa osynliga samt vissa specialtecken

## 3.7 Några format för binära filer/data

Normalt pratar man om två olika filtyper i datorvärlden: Rena *textfiler* som ibland också brukar benämñas ASCII-filer och *binärfiler*. Ideen med detta avsnitt är att presentera några olika sätt att koda om rena binärfiler till textfiler då dessa ofta är bekvämare att använda.

En textfil byggs till största delen upp av de synliga tecknen i ASCII-tabellen, förutom detta används några av de osynliga tecknen som formateringskoder såsom tabulatertecknet, ny rad tecknet (LF), och vagnreturtecknet (CR). En textfil kan då visuellt betraktas med ett ordbehandlingsprogram, datorn betraktar fortfarande varje tecken som binärdata.

Binärfiler innehåller binärt data och kan ej på samma sätt som en texfil betraktas med ett ordbehandlingsprogram. Binärfilen kan innehålla binärt data som kan representeras som synliga ASCII-tecken men innehåller också binärt data som ej kan representeras som synliga tecken. Typiskt är binärfiler ofta program- och applikationsfiler som kan exekveras av en dator, andra typer av binärfiler kan innehålla binärdatal för programmering av PROM-minnen, etc. Ofta försöker man koda om dessa filer till ASCII-filer, då detta bl. a. har den fördelen med att kunna vara läsbara av ett textredigeringsprogram.

### 3.7.1 BinHex-formatet

En textfil har den fördelen att det är lätt att sända och ta emot sådana filer för de flesta elektroniska postssystem. Däremot kan det vara lite mer komplicerat att sända och ta emot en ren binärfil. *BinHex* är en metod för att koda binärt data så att det endast består av synliga och utskrivbara ASCII-tecken. Observera att även en vanlig ASCII-fil kan kodas enligt BinHex-formatet, vanligtvis är detta ej nödvändigt.

BinHex är bland annat en de facto standard för kodning av filer för MacIntosh-datorer. Den används både vid sändning av filer via elektroniska postsystem och för lagring av MacIntoshfiler på FTP<sup>1</sup>-servrar.

Vid kodning enligt BinHex-standarden så grupperas det binära datat i grupper om 6 bitar, därav namnet BinHex. Varje binärt värde om sex bitar kan sedan kodas efter sitt

<sup>1</sup>File Transfer Protocol. Ett Internet-protokoll för att sända filer.

binärvärde genom tabellslagning i tabellen som följer:

	000	001	010	011	100	101	110	111
	0	1	2	3	4	5	6	7
000	!	"	#	\$	%	&	'	(
	8	9	10	11	12	13	14	15
001	)	*	+	,	-	0	1	2
	16	17	18	19	20	21	22	23
010	3	4	5	6	8	9	@	A
	24	25	26	27	28	29	30	31
011	B	C	D	E	F	G	H	I
	32	33	34	35	36	37	38	39
100	J	K	L	M	N	P	Q	R
	40	41	42	43	44	45	46	47
101	S	T	U	V	X	Y	Z	[
	48	49	50	51	52	53	54	55
110	'	a	b	c	d	e	f	h
	56	57	58	59	60	61	62	63
111	i	j	k	l	m	p	q	r

Figur 3.14: Teckenmängd som används i BinHex-kodning

### ■ Exempel 3.4: BinHex-kodning

Antag att du har en fil som består av följande binärdata ordnat i 8-bitarsgrupper:

0	1	0	0	0	0	0	1
0	1	0	0	0	0	1	0
0	1	0	0	0	0	1	1

Vid kodning enligt BinHex-standarden organiseras bitarna i grupper om 6:

0	1	0	0	0	0	16	'3'
0	1	0	1	0	0	20	BinHex-kodning '8'
0	0	1	0	0	1	9	⇒ '*'
0	0	0	0	1	1	3	'\$'

I BinHex-kodning kommer binärdatat att representeras av de ASCII-kodade tecknen: **38\*\$**. På grund av kodningssättet kommer den kodade filen att bli 33% större jämfört med den okodade, detta gäller om varje ASCII-tecken representeras av 8 bitar.

**Slut exempel 3.4 ■**

En fil i BinHex-formatet inleds alltid med teckensträngen: **(This file must be converted with BinHex 4.0)**. Detta gör det möjligt för ett BinHex-avkodningsprogram att identifiera filen som en BinHex-fil. Den kodade delen av filen kan inledas med ett huvud med information om: filens orginalnamn, typ av fil, skapare av filen (program), etc. Därefter kommer den kodade filen mellan ett startkolonstecken : och ett slutkolonstecken :.

### ■ Exempel 3.5: Början på en BinHex-fil

(This file must be converted with BinHex 4.0)

```
:%NATG'96GfPdBfJJ-5i` ,R0TG! "6594%8dP8)3#3" )Je!*!%DHp6593K!!%!! )J
eFNAKG3)$!*!$&J!A)#!16 'PdC90hDA4MD#a,M! !N" (Zh`#3!k- "e!%$!VS!N!-
```

**Slut exempel 3.5 ■**

### 3.7.2 Motorolas S-records

Filer innehållande Motorolas S-records är läsbara med ett textredigeringsprogram. Filen byggs upp med hjälp av 8 olika posttyper, så kallade S-records. Strukturen på en post är enligt figuren som följer:

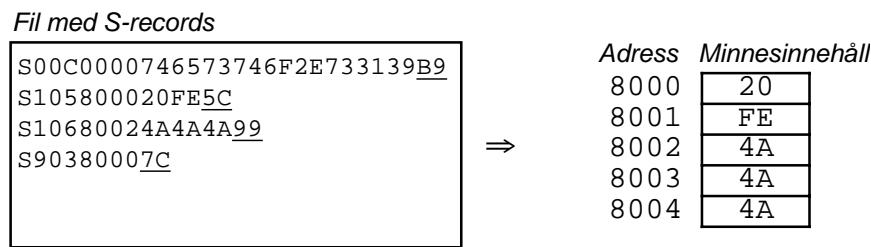
Fält	Storleken i tecken
Posttyp	2
Postens längd	2
Minnesadress	4, 6, eller 8
Kod/data	0 - 2N
Checksumma	2

*Figur 3.15: Strukturen på en S-record*

Posttypen består av två tecken och inleds med ett S som kan tolkas som starttecken på raden där efter kommer en siffra som anger typen på posten. Giltiga värden är: S0,S1,S2,S3,S5,S7,S8, eller S9. Tolkningen av posttyperna är enligt följande:

- S0** Posthuvud som för ett block av S-records, innehåller beskrivande information som identifierar blocket av S-records som följer. Adressfältet är typiskt noll. Kod/datafältet innehåller typiskt en "hexifierad" teckensträng.
- S1** Kod/datapost innehållande en 2 bytes adress, d.v.s. 4 hexadecimala siffror.
- S2** Kod/datapost innehållande en 3 bytes adress, d.v.s. 6 hexadecimala siffror.
- S3** Kod/datapost innehållande en 4 bytes adress, d.v.s. 8 hexadecimala siffror.
- S5** Räknepost innehållande information om antalet S1,S2, och S3 poster som fanns i det block som har sänts. Räknevärdet finns i adressfältet, kod/datafältet är tomt. Normalt sett så används ej S5-poster.
- S7** Termineringspost för ett block av S-records. En 4 bytes adress innehåller adressen där exekveringen startar, kod/datafältet är tomt.
- S8** Termineringspost för ett block av S-records. En 3 bytes adress innehåller adressen där exekveringen startar, kod/datafältet är tomt.
- S9** Termineringspost för ett block av S-records. En 2 bytes adress innehåller adressen där exekveringen startar, kod/datafältet är tomt.

■ **Exempel 3.6:** Exempel på en fil med S-records



Radchecksumman som står sist på en rad (understruket) är vald på sådant sätt, att om man betraktar alla tal som bytetal, skall summan av dessa bli  $(FF)_{16}$ . Om vi som exempel tar raden med radchecksumman  $(5C)_{16}$ , ser vi om vi utför additionen  $(05 + 80 + 00 + 20 + FE + 5C)_{16} = (FF)_{16}$  att detta stämmer.

**Slut exempel 3.6** ■

## 3.8 Uppgifter

### U1: Teorifrågor om talsystem, koder och aritmetik

- Vilken vikt har position  $N$  i ett heltal i ett positionssystem med basen  $B$ ?
- Vilka är vikterna för de 8 minst signifikanta positionerna i ett binärt heltal?
- Vad menas med att ett talsystem är ett positionssystem?
- Ett specifikt 8-bitars värdet i minnet kan betyda olika saker beroende av vilket sammanhang det förekommer. Det hexadecimala värdet 0x3A kan tolkas på olika sätt. Ett av nedanstående alternativ är ej rätt.
  - Såsom ett decimaltal med värdet 58.
  - Operationskoden för instruktionen PULD (Pull Double Accumulator from Stack)
  - BCD-kodade talet 310
  - ASCII-värdet för tecknet :.
- Informationen i de 8 bitarna 11001000 kan ges olika betydelse. Vilket av följande alternativ är ej rätt?
  - BCD talet 128
  - Decimala heltalet 200
  - Tecken-belopp-talet -72
  - En instruktionskod (maskinkod) för EORB.
- Varför föredras 2-komplementkodning vid kodning av negativa tal?
- Hur många bitar går det på en hexadecimal respektive en decimal siffra?

### U2: Binära talsystemet

- Omvandla följande decimala tal till binär form:

$10_{10}$

$213_{10}$

$512_{10}$

$94_{10}$

- Omvandla följande binära heltal till decimal form:

$101_2$  $11110_2$  $110110_2$  $110111101_2$ **c)** Omvandla följande decimala tal till binär form: $43.32_{10}$  $7.5_{10}$  $17.15_{10}$ **d)** Omvandla följande binära heltal till decimal form: $11.11_2$  $11011.001_2$ **e)** Vilken ordlängd i antal bitar krävs för att koda 1000 olika möjligheter respektive  $10^6$  möjligheter?**f)** Addera följande binära tal:

$$\begin{array}{r} 10100 \\ + \quad 00011 \\ \hline \end{array}$$

$$\begin{array}{r} 1010 \\ + \quad 0011 \\ \hline \end{array}$$

$$\begin{array}{r} 0111 \\ + \quad 0001 \\ \hline \end{array}$$

$$\begin{array}{r} 0111 \\ + \quad 0011 \\ \hline \end{array}$$

### U3: Hexadecimala talsystemet

**a)** Omvandla följande decimala tal till hexadecimal form: $10_{10}$  $213_{10}$  $512_{10}$  $94_{10}$ **b)** Omvandla följande hexadecimala tal till decimal form: $111_{16}$  $5FF_{16}$  $ABCD_{16}$  $2A0_{16}$ **c)** Omvandla följande binära heltal till hexadecimal form: $101_2$  $11110_2$  $110110_2$  $110111101_2$ 

### U4: BCD (Binary Coded Decimal)

**a)** Skriv följande decimala tal i BCD-kod:

14

98

144

55

**b)** Skriv följande BCD-kodade binärtal till decimal form

10011000

11101110101

### U5: ASCII (American Standard Code of Information Interchange)

**a)** Vilket ASCII-värde (heltalsvärde) har följande tecken?

a

b

c

d

A

B

C

D

0

1

2

3

**b)** I vilken ordning sorteras följande ord om ASCII-värdena används respektive "normal" ordningsföljd:

myra, Orsa, orre, Myrra

**U6: Tecken-belopprepresentation**

- a) Hur representeras -5 i tecken-belopprepresentation? Använd 8 bitars ord längd.
- b) Använd tecken-belopp-representation för att skriva följande decimala tal i binär form med ord längden lika med 8 bitar

$$87_{10} \quad -87_{10}$$

**U7: 1-komplementsrepresentation**

- a) "+6" representeras av "00000110". Vad blir representationen av "-6" i 1-komplement?
- b) Vad blir 1-komplementet till följande binära heltal?

$$0100_2 \quad 1100_2 \quad 0000_2$$

**U8: 2-komplementsrepresentation**

- a) "+6" representeras av "00000110". Vad blir representationen av "-6" i 2-komplement?
- b) Vad blir 2-komplementet till följande binära heltal?
- c) Hur representeras -25 med 8 bitar respektive 16 bitar i 2-komplementrepresentation?
- d) Vilket är det största respektive minsta tal som kan representeras i 2-komplementrepresentation med användande av en byte.
- e) Vilket är det största respektive minsta tal som kan representeras i 2-komplementrepresentation med användande av två byte.
- f) Vilket är det största respektive minsta tal som kan representeras i 2-komplementrepresentation med användande av fyra byte.
- g) Slutförl följande tabell av ekvivalenta värden, antag att 16 bitar används. Använd ej räknedosa utan gör arbetet med hjälp av handräkning.

Decimalt med teckenbit	Decimalt utan teckenbit	Hexadecimalt	Oktalt	Binärt
-10000	55536	D8F0	154360	1101100011110000
127				
		FE27		
	60000			
				0110000001000111
			127	
-1				
				0000000011101110
		0BCA		
			166377	
12359				
		7FFF		
		8000		

- h) Rita ett blockschema över en kombinerad additions- och subtraktionskrets där negativa tal representeras enligt 2-komplementmetoden.

### U9: Addition av tal med 2-komplementsrepresentation

Fullfölj följande additioner av åtta bitars tal i 2-komplementrepresentation. Indikera resultatet, carry-biten C, overflowbiten V och om resultatet är korrekt eller fel.

a)

$$\begin{array}{r}
 00000110 \quad ( \quad ) \\
 + 00001000 \quad ( \quad ) \\
 \hline
 = \quad \quad \quad ( \quad ) \quad V: \quad C:
 \end{array}$$

b)

$$\begin{array}{r}
 01111111 \quad ( \quad ) \\
 + 00000001 \quad ( \quad ) \\
 \hline
 = \quad \quad \quad ( \quad ) \quad V: \quad C:
 \end{array}$$

c)

$$\begin{array}{r}
 00000100 \quad (\quad) \\
 + 11111110 \quad (\quad) \\
 \hline
 = \quad (\quad) \text{ V:} \quad \text{C:}
 \end{array}$$

**d)**

$$\begin{array}{r}
 00000010 \quad (\quad) \\
 + 11111100 \quad (\quad) \\
 \hline
 = \quad (\quad) \text{ V:} \quad \text{C:}
 \end{array}$$

**e)**

$$\begin{array}{r}
 11111110 \quad (\quad) \\
 + 11111010 \quad (\quad) \\
 \hline
 = \quad (\quad) \text{ V:} \quad \text{C:}
 \end{array}$$

**f)**

$$\begin{array}{r}
 10000001 \quad (\quad) \\
 + 11000010 \quad (\quad) \\
 \hline
 = \quad (\quad) \text{ V:} \quad \text{C:}
 \end{array}$$

**g)**

$$\begin{array}{r}
 10111111 \quad (\quad) \\
 + 11000001 \quad (\quad) \\
 \hline
 = \quad (\quad) \text{ V:} \quad \text{C:}
 \end{array}$$

**h)**

$$\begin{array}{r}
 00010000 \quad (\quad) \\
 + 01000000 \quad (\quad) \\
 \hline
 = \quad (\quad) \text{ V:} \quad \text{C:}
 \end{array}$$

**i)**

$$\begin{array}{r}
 11111010 \quad (\quad) \\
 + 11111001 \quad (\quad) \\
 \hline
 = \quad (\quad) \text{ V:} \quad \text{C:}
 \end{array}$$

**j)**

$$\begin{array}{r}
 01111110 \quad (\quad) \\
 + 00101010 \quad (\quad) \\
 \hline
 = \quad (\quad) \text{ V:} \quad \text{C:}
 \end{array}$$

**k)** Kan spill (overflow) inträffa vid addition om båda talen har samma tecken?

**l)** Kan spill (overflow) inträffa vid addition om båda talen har olika tecken?

## U10: Antal bitar/bytes vid olika typer av kodning

- a) Jämför antalet bitar som krävs för att representera tal inom följande decimala talområden:

Talområde	Antal bitar i binär kod	Antal bitar BCD-kod	Antal bitar ASCII-kod
0-9			
0-99			
0-999			
0-9999			

- b) Hur många 8-bitars minnesceller behövs för att lagra ASCII-representationen av namnet "FRED"?

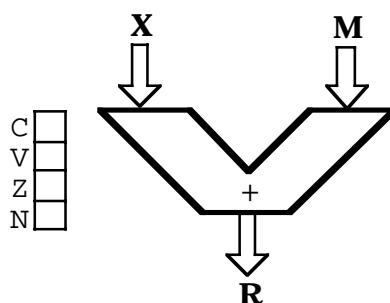
A) 16      B) 4      C) 7      D) 2

- c) Hur många 8-bitars minnesceller behövs för att lagra ASCII-representationen av namnet "FRED" om lagringen görs som i programspråket C med en null-terminerad sträng?

A) 16      B) 4      C) 5      D) 2

## U11: Aritmetisk logisk enhet – ALU

- a) ALU:n arbetar med en 8 bitars ordlängd, bitformatet är  $x = x_7x_6x_5x_4x_3x_2x_1x_0$ , analogt för M och R.



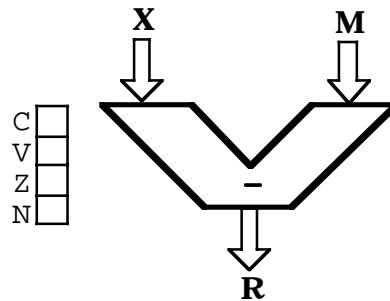
Vilket av följande påståenden är rätt om en ALU som är inställd för addition  $R=X+M$ ?

- A) Spill kan inträffa om talen har olika tecken.
- B) Spillflaggan bildas av följande logiska uttryck:  

$$V = \underline{x}_7 \cdot \underline{m}_7 \cdot \underline{r}_7 + \underline{x}_7 \cdot \underline{m}_7 \cdot \underline{r}_7$$
- C) Minnessiffran bildas av:  

$$C = \underline{x}_7 \cdot \underline{m}_7 \cdot \underline{r}_7 + \underline{x}_7 \cdot \underline{m}_7 \cdot \underline{r}_7$$
- D) Minnessiffran får vi alltid om talen har samma tecken.

b)



Vilket av följande påståenden är rätt om en ALU som är inställd för subtraktion  $R=X-M$ ? Utgå t.ex. från SUB-instruktionen i manualen för AVR-processorns instruktionsuppsättning för att lösa denna uppgift.

- A) Spillflaggan bildas av följande logiska uttryck:

$$V = \underline{x_7} \cdot \underline{m_7} + \underline{x_7} \cdot \underline{m_7} \cdot \underline{r_7}$$

- B) Om talen inte har någon teckenbit är spillflaggan V för 2-komplementoverflow ointressant.  
 C) Minnessiffra får vi alltid om talen har samma tecken.  
 D) Spill kan inte inträffa om talen har olika tecken.

### 3.9 Svar

#### **U1: Teorifrågor om talsystem, koder och aritmetik**

- a)  $B^N$   
 b)  $2^0=1, 2^1=2, 2^2=4, 2^3=8, 2^4=16, 2^5=32, 2^6=64, 2^7=128$   
 c)  
 d) C)  
 e) A) BCD talet 128  
 f)  
 g) 4 respektive 3.3 bitar.

#### **U2: Binära talsystemet**

a)  $10_{10} = \underline{\underline{1010}}_2$

$213_{10} = \underline{\underline{11010101}}_2$

$512_{10} = \underline{\underline{1000000000}}_2$

#### **Omvandling av $94_{10}$**

$$\begin{aligned} 94/2 &= 47 + 0/2 \\ 47/2 &= 23 + 1/2 \\ 23/2 &= 11 + 1/2 \\ 11/2 &= 5 + 1/2 \\ 5/2 &= 2 + 1/2 \\ 2/2 &= 1 + 0/2 \\ 1/2 &= 0 + 1/2 \end{aligned}$$

↑ Minst signifikanta biten (positionen)  
 ↓ Mest signifikanta biten (positionen)

$94_{10} = \underline{\underline{1011110}}_2$

b)  $101_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = \underline{5}_{10}$

$$11110_2 = \underline{30}_{10}$$

$$110110_2 = \underline{54}_{10}$$

$$110111101_2 = \underline{445}_{10}$$

c) **Omvandling av 43.32**

Heltals delen  $N_0=43$  och bråkdelens  $N_1=0.32$

Omvandling av  $N_0$

$$\begin{aligned} 43/2 &= 21 + 1/2 \\ 21/2 &= 10 + 1/2 \\ 10/2 &= 5 + 0/2 \\ 5/2 &= 2 + 1/2 \\ 2/2 &= 1 + 0/2 \\ 1/2 &= 0 + 1/2 \end{aligned}$$

*Minst signifikanta biten (positionen)*

*Mest signifikanta biten (positionen)*

Omvandling av  $N_1$

$$\begin{aligned} 0.32 \cdot 2 &= 0.64 + 0 \\ 0.64 \cdot 2 &= 0.28 + 1 \\ 0.28 \cdot 2 &= 0.56 + 0 \\ 0.56 \cdot 2 &= 0.12 + 1 \\ 0.12 \cdot 2 &= 0.24 + 0 \\ 0.24 \cdot 2 &= 0.48 + 0 \\ 0.48 \cdot 2 &= 0.96 + 0 \\ 0.96 \cdot 2 &= 0.92 + 1 \end{aligned}$$

*Mest signifikanta biten (positionen)*

*Minst signifikanta biten (positionen)*

$$43.32_{10} = \underline{101011.01010001 \dots}_2$$

$$7.5_{10} = \underline{111.1}_2$$

$$17.15_{10} = \underline{10001.001001100110011 \dots}_2$$

d)  $11.11_2 = \underline{3.75}_{10}$

$$11011.001_2 = \underline{27.125}_{10}$$

e) 10 respektive 20 boolska variabler.

f) 10111      1101      1000      1010

### U3: Hexadecimala talsystemet

a)  $10_{10} = \underline{A}_{16}$

**Omvandling av 213<sub>10</sub>**

$$\begin{aligned} 213/16 &= 13 + 5/16 \\ 13/16 &= 0 + 13/16 \end{aligned}$$

$$213_{10} = 13 \cdot 16^1 + 5 \cdot 16^0 = \underline{D5}_{16}$$

$$512_{10} = \underline{200}_{16}$$

$$94_{10} = \underline{5E}_{16}$$

b)  $111_{16} = \underline{273}_{10}$

$$5FF_{16} = \underline{1535}_{10}$$

$$ABCD_{16} = 10 \cdot 16^3 + 11 \cdot 16^2 + 12 \cdot 16^1 + \underline{13} = 43981_{10}$$

$$2A0_{16} = \underline{672}_{10}$$

c)  $101_2 = \underline{5}_{16}$

$$11110_2 = \underline{1E}_{16}$$

$$110110_2 = \underline{36}_{16}$$

$$110111101_2 = 1 \ 1011 \ 1101 = \underline{1BD}_{16}$$

#### **U4: BCD (Binary Coded Decimal)**

a)  $14_{10} = \underline{0001 \ 0100}_{BCD}$

$$98_{10} = \underline{1001 \ 1000}_{BCD}$$

$$144_{10} = \underline{0001 \ 0100 \ 0100}_{BCD}$$

$$55_{10} = \underline{0101 \ 0101}_{BCD}$$

b)  $1001 \ 1000_{BCD} = \underline{98}_{10}$

$$111 \ 0111 \ 0101_{BCD} = \underline{775}_{10}$$

#### **U5: ASCII (American Standard Code of Information Interchange)**

a) -

b) ASCII-sortering: Myrra, Orsa, myra, orre

Normal sortering: myra, Myrra, orre ,Orsa

#### **U6: Tecken-belopprepresentation**

Svaren ges i 8 bitars tecken-beloppkod (TBK).

a)  $(-5)_{10} = (10000101)_{TBK}$

b)  $(87)_{10} = (01010111)_{TBK}, (-87)_{10} = (11010111)_{TBK}$

**U7: 1-komplementsrepresentation**

a) 11111001

b)  $0100 \xrightarrow{1\text{-komplement}} 1011, \quad 1100 \xrightarrow{1\text{-k}} 0011, \quad 0000 \xrightarrow{1\text{-k}} 1111$

**U8: 2-komplementsrepresentation**

a) 11111010

b)  $0100 \xrightarrow{2-K} 1100, \quad 1100 \xrightarrow{2-K} 0100, \quad 0000 \xrightarrow{2-K} 0000$

c)  $(25)_{10} = (0000000000011001)_2$

$0000000000011001 \xrightarrow{2-K} 111111111100111$

$(-25)_{10} = (111111111100111)_2$

Hur representeras -25 med 8 bitar respektive 16 bitar i 2-komplementrepresentation?

d) Största tal är 127 och minsta tal är -128

e) Största tal är  $2^{15} - 1$  och minsta tal är  $-2^{15}$ 

f) -

g)

Decimalt med teckenbit	Decimalt utan teckenbit	Hexadecimalt	Oktalt	Binärt
-10000	55536	D8F0	154360	1101100011110000
127	127	007F	000177	0000000001111111
		FE27		
	60000			
				0110000001000111
87	87	0057	000127	0000000001010111
-1	65536	FFFF	177777	1111111111111111
238	238	00EE	000356	0000000011101110
		0BCA		
			166377	
12359				
32765	32765	7FFF	077777	0111111111111111
-32768	32768	8000	100000	1000000000000000

## U9: Addition av tal med 2-komplementsrepresentation

$$\begin{array}{cccccccc}
 a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 \\
 + m_7 & m_6 & m_5 & m_4 & m_3 & m_2 & m_1 & m_0 \\
 \hline
 r_7 & r_6 & r_5 & r_4 & r_3 & r_2 & r_1 & r_0
 \end{array}$$

C-flaggan bildas direkt av bit  $r_8$ , dvs om en minnessiffra genereras till ledet efter kolumn 7. Spillflaggan V bildas genom att studera de bitar som kan betraktas som teckenbitar dvs  $a_7$ ,  $m_7$  och  $r_7$ . Om talen som adderas har samma värde på teckenbiten skall också resultatet ha detta värde, om inte detta gäller har vi spill.

- a) 00001110 V=0, C=0
- b) 10000000 V=1, C=0
- c) 00000010 V=0, C=1
- d) 11111110 V=0, C=0
- e) 11111000 V=0, C=1
- f) 01000011 V=1, C=1
- g) 10000000 V=0, C=1
- h) 01010000 V=0, C=0
- i) 11110011 V=0, C=1

j 10101000 v=1, c=0

### **U10: Antal bitar/bytes vid olika typer av kodning**

a)

Talområde	Antal bitar i binär kod	Antal bitar BCD-kod	Antal bitar ASCII-kod
0-9	4	4	8
0-99	7	8	16
0-999	10	12	24
0-9999	14	16	32

b) B) 4

c) C) 5

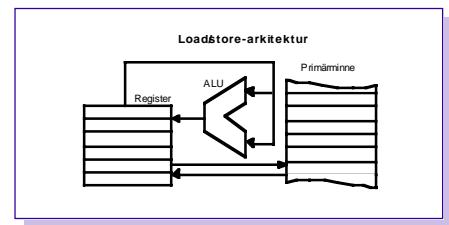
### **U11: Aritmetisk logisk enhet - ALU**

a) B)

b) B)

K  
A  
P  
I  
T  
E  
L

# 4



## AVR 8 bitars datorer

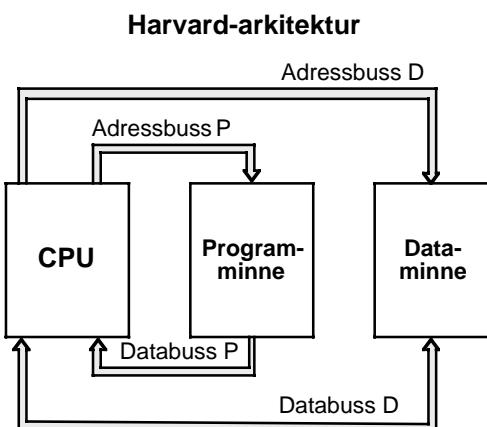
Vi kommer i detta kapitel behandla en speciell mikrostyrenhet i ATMEL:s familj av 8 bitars RISC-processor under märkesnamnet AVR. Mikrostyrenheten heter AVR ATmega16. När du läser detta kapitel har du nyttja av att även ha tillgång till den tekniska referensmanualen "ATmega16" från ATMEL. Detta är den bästa informationskällan om denna processors uppbyggnad.

### 4.1 Lite om datorarkitektur

#### 4.1.1 Harvard- eller von-Neumann-arkitektur

En CPU med Harvard-arkitektur har separata datavägar för att överföra instruktioner och data till CPU:n. Namnet Harvard-arkitektur kommer från en IBM-dator som levererades till Harvard-universitetet 1944. Detta var en av de tidigaste elektromekaniska datorerna. För att bygga denna dator åtgick mer än 750000 komponenter, vilket gjorde att datorn vägde ansenliga 5 ton. Jämför detta med den enchipsdator som vi använder i denna kurs, som har en mycket större beräkningskapacitet och väger försvinnande lite i jämförelse. Instruktionerna till denna IBM-dator sparades på en håslagen pappersremsa och datat i låskretsar uppbyggt med hjälp av reläer.

Harvardarkitekturen karakteriseras av de två address- respektive databussar som finns i ett sådant system.

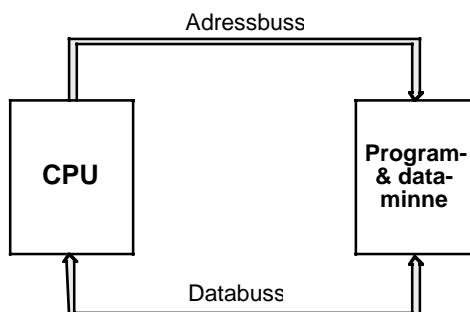


**Figur 4.1:** Harvard-arkitektur

Fördelen med Harvardarkitekturen är att exekveringshastigheten , d.v.s. den tid det tar att exekvera ett program kan minskas jämfört med andra arkitekturlösningar.

En annan typ av datorarkitektur som används vid konstruktion av processorer brukar benämnas efter den ungerske matematikern John von Neumann (1903-1957) och kallas följaktligen ”*von Neumann dator*”. En von Neumann-dator lagrar både program och data i ett minne, ideen går under namnet ”the Stored Program Concept”. I en datorarkitektur som är baserad på von Neumann arkitektur har man en gemensam dataväg för instruktioner och data. Detta innebär att endast en address- och en databuss finns i ett sådant system.

#### von Neumann-arkitektur



**Figur 4.2:** von Neumann-arkitektur

Vilka egenskaper har respektive arkitektur?

#### Harvard-arkitektur

Exekvering av en instruktion på 1 klockcykel.

Parallel hämtning av instruktioner och data.

Instruktioner och data är alltid separerade.

Kod respektive data kan ha olika bitbredd.

#### von Neumann arkitektur

Exekvering av en instruktion tar typiskt flera klockcykler.

Seriell hämtning av instruktioner och data.

En minnestruktur.

Processorer med von Neumann-arkitektur finns i alla persondatorer som har Intel 80x86/Pentium-processorer. Även Motorolas processorfamiljer med beteckningen 68000 respektive 68xx har denna arkitektur. Harvardarkitekturen återfinns ofta i signalprocessorer, Atmel AVR-familj, Microchips PIC-familj och i persondatorer av typen Apple MacIntosh som är bestyckade med PowerPC-processorer (tillverkare IBM och Motorola).

### 4.1.2 CPU-arkitektur: CISC eller RISC

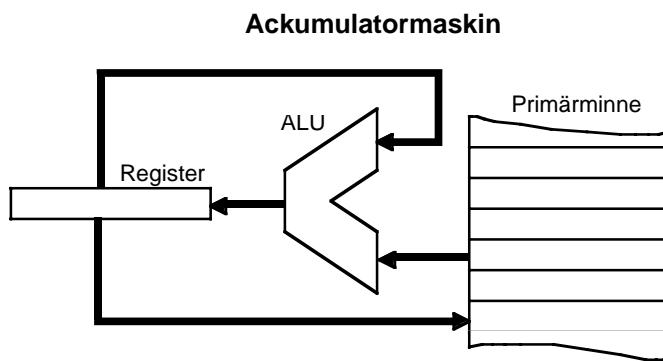
Två andra begrepp som också är relaterade till föregående avsnitt om Harvard- och von Neumann arkitektur är CISC och RISC.

*CISC* — *Complex Instruction Set Computer* är den ursprungliga CPU-arkitekturen. Det som karakteriseras CISC är att instruktionerna till CPU:n kan vara komplexa i den meningen att de har en hög funktionalitet och är specialiserade. En typisk instruktion finns i flera upplagor beroende på vilken adresseringsmod som används (direkt minnesadressering, indexerad minnesadressering, etc.). Specialiserade instruktioner kan skapas av CISC-konstruktören för att understödja olika tillämpningar. I CISC-processorn 68HC12 från Motorola finns specialiserade instruktioner för att implementera fuzzy logic regulatorer och digitala filter. Instruktioner och adresseringsmoder skapas för att understödja kodgenerering från C/C++-kompilatorer.

En motreaktion mot CISC kom på 1980-talet i form av *RISC* — *Reduced Instruction Set Computer*. RISC karakteriseras av en liten mängd av primitiva (enkla) instruktioner för att kunna utföra elementära operationer. Komplexa funktioner får byggas upp med hjälp av de primitiva instruktionerna. Alla instruktioner exekveras på en klockcykel. Alla instruktioner fungerar på likartat sätt mot registren i CPU:n.

En modern CPU är oftast en blandning av CISC- och RISC-tänkande. Det bästa plockas från båda ansatserna till CPU-konstruktion.

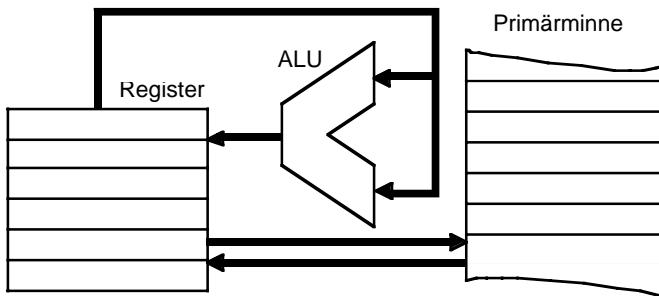
En *ackumulatormaskin* är en CPU med CISC- respektive von Neumann arkitektur. Datorn karakteriseras av ett eller några beräkningsregister, där ena operanden till ALU:n hämtas från minnet och den andra från ett register.



*Figur 4.3: Ackumulatormaskin*

En CPU med *load/store*-arkitektur är en CPU med RISC- respektive Harvard-arkitektur. Datorn karakteriseras av många register där båda operanderna till ALU:n hämtas ifrån registerbanken. Kommunikationen med dataminnet sker med hjälp av två instruktioner load-ladda från minnet och store-spara i minnet.

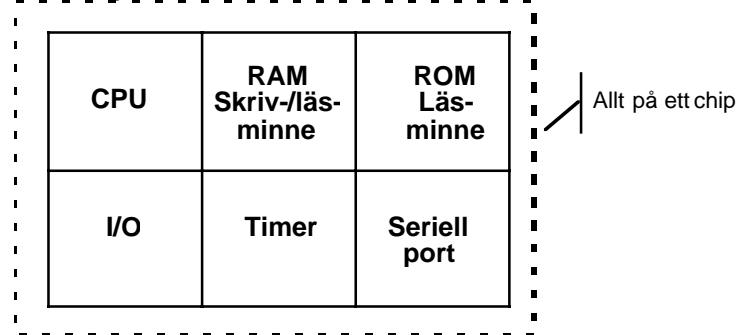
### Load/store-arkitektur



**Figur 4.4:** Load/store-arkitektur

## 4.2 ATMEL ATmega16

En *mikrostyrenhet — microcontroller* är en enhet integrerad i en krets (ett chip). En komplett dator innehåller förutom en CPU också olika minnestyper (RAM, ROM, EEPROM, FLASH, etc.), I/O-enheter, seriella kommunikationsportar, timer, etc.



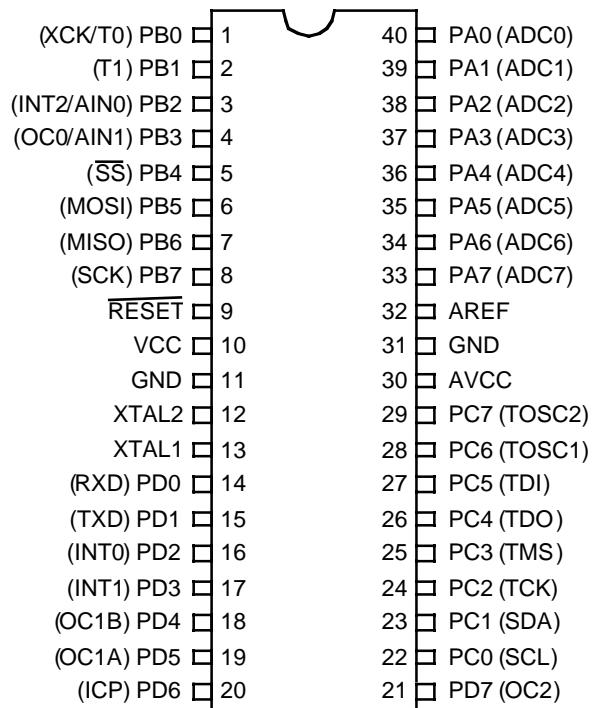
**Figur 4.5:** Typisk uppbyggnad av en mikrostyrenhet

En mikrostyrenhet är konstruerad för att fungera i olika tillämpningar. Priset på en sådan krets är normalt ganska lågt, vilket innebär att inget slöseri görs med kiselytan. Konstruktionen byggs upp av kompakta och effektiva komponenter tänkta att lösa ofta förekommande tillämpningsproblem.

Låg effektförbrukning kan fås för applikationer som använder batterimatning genom att matningspåningen då kan väljas till  $V_{DD} = 2.7$  V. I applikationer som får sin matningsspänning via nätet väljs vanligtvis  $V_{DD}$  till 5V. Förlusteffekten  $P_f$  för en CMOS-krets beror av klockfrekvensen  $f$  och den genomsnittliga kapacitativa lasten  $C_L$ , som skall laddas upp och ur under en klockperiod av matningsspänningen  $V_{DD}$  enligt formeln  $P_f = f \cdot C_L \cdot V_{DD}^2$ . Av formeln ser vi att effektförbrukningen då den lägre

matningsspänningen väljes blir  $\frac{2.7^2}{5^2} \cdot 100 = 30\%$  av vad den skulle ha blivit om den högre matningsspänningen hade valts. Nackdelen med lägre matningsspänning är att systemet kan bli mer störningskänsligt, detta kan givetvis kompenseras med hjälp av att ha detta i åtanke vid konstruktion av systemet.

En bra första start för att studera mikrostyrenheten ATMEL ATmega16 är att titta på kretsens pinnkonfigureringsdiagram.

**Figur 4.6:** Pinn-konfigureringsdiagram för ATmega16

### 4.2.1 Portar

Mikrostyrenheten har en mängd olika IO-funktioner: analoga insignalер, räknare, digitala in- och utsignalер, etc. Dessa funktioner står i kontakt med omvärlden via någon av processorns pinnar och de speciella register i processorns som kallas *portar*, d.v.s. in- och utgångar mot omvärlden. Kretsens gränssnitt mot omvärlden består av 32 anslutningar (pinnar), grupperade som 4x8 anslutningar vilka benämns PORTA (PA7-PA0) till PORTD (PD7-PD0). Portarna kan programmeras till att ha olika funktionalitet beroende av tillämpning.

### 4.2.2 Externa avbrott

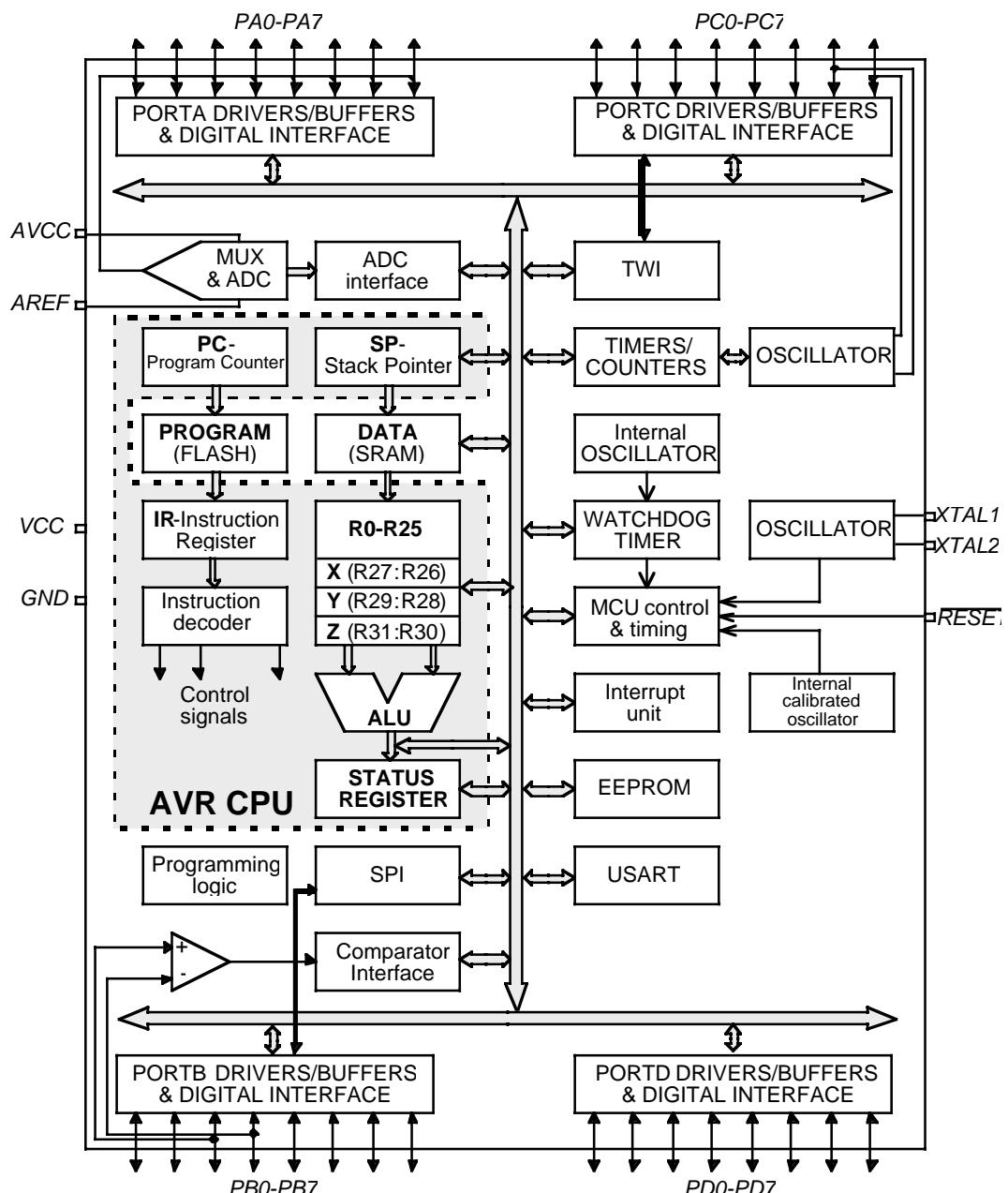
Tre stycken externa avbrottspinnar finns på processorn: RESET, INT0, och INT1. Pinnarna INT0, och INT1 (INTerrupt) är tänkta att användas för externa I/O-enheter som på något sätt vill tala om för processorn att en händelse har inträffat som behöver betjänas, d.v.s. aktuell programexekvering måste avbrytas för att exekvera ett avbrottsprogram som en reaktion och svar på den händelse som avbrottssignalen innebär.

#### 4.2.2.1 Reset-pinnen

Om du studerar figuren över ATmega16:s pinnar, ser du att det finns en pinne med namnet RESET. Avbrottsslinjen RESET används vid spänningsspäslag för att processorn ska starta upp på ett i förväg känt sätt. Signalen är aktivt låg och skall var det en viss tid för att gå in i återstartsmoden. Vidare så brukar denna avbrottsslinje på en persondator, vara forbunden med en tryckknapp för att kunna starta om datorn vid något slag av programhaveri.

### 4.2.3 Blockdiagram

Mikrostyrkretsarna under namnet AVR från ATMEL är en encykels RISC-maskin med 32 generella register å 8 bitar vardera. En instruktion tar en klockcykel att utföra. AVR är en CPU konstruerad efter principerna för RISC-maskiner, trots detta har den relativt många instruktioner varav en del är CISC-likt. Som alla moderna CPU-konstruktioner är den konstruerad för att understödja effektiv kodgenerering från C-kompilatorer. Genom att den har många register kan komplexa beräkningar och parameteröverföring till C-funktioner ske utan att allt för mycket flyttande av data mellan CPU och minne behöver göras.



Figur 4.7: Blockdiagram över ATmega16-datorn

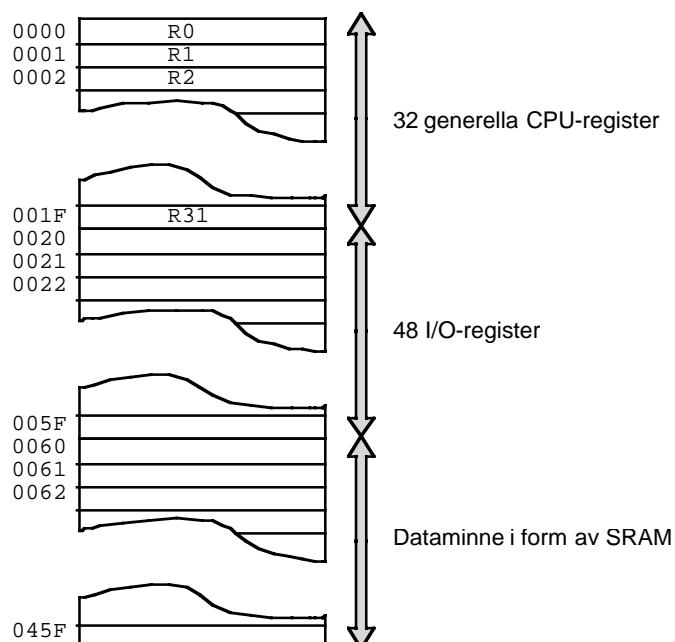
Den takt med vilken en processor arbetar med brukar benämnes *maskincykel*. Om vi klockar processorn med 8MHz så blir tiden för en maskincykel lika med 125

nanosekunder.

#### 4.2.4 Minnesmapp för ATmega16

En *minnesmapp* är en beskrivning av hur adressrymden används för att accessa olika typer av minnen och I/O-enheter i systemet. Minnesmappen kan vara mer detaljerad än så och innehålla information om vad som är program- respektive dataminne, adresser till funktioner och globala variabler i programmen, etc. Beroende på att AVR-processorer har en Harvard-arkitektur så finns det en minnesmapp för dataminne (SRAM) och en minnesmapp för programminne (FLASH). Förutom detta så ligger även ett EEPROM-minne i en egen *adressrymd*, detta minnes främsta användning är för permanentlagring av parametrar. Vi har alltså 3 olika minnesmappar att hålla reda på.

I figuren som följer visas minnesmappen för data. Denna är uppdelad i 3 olika delar: De 32 generella CPU-registrena kan nås via adresserna 0x0000 till 0x001F i dataminnet. Processorns I/O-register ligger från address 0x0020 till 0x005F. Resten av adressrymden från 0x0060 till 0x0045F kan användas för vanlig datalagring.



**Figur 4.8:** Dataminnesmapp för ATmega16

### 4.3 Uppgifter

#### U1: Teorifrågor om datorarkitektur

- Rita ett blockdiagram för en CPU med Harvardarkitektur.
- Rita ett blockdiagram för en CPU med von Neumann-arkitektur.
- Vad står CISC för?
- Vad står RISC för?

## **U2: Teorifrågor om AVR RISC-processorer**

- a) Hur många olika minnesmappar finns det i en AVR-processor? Namnge dessa.
- b) Antag att matningsspänningen  $V_{DD}$  i ett datorsystem byggt med hjälp av CMOS-teknologi sänks från 5V till 3V. Hur mycket minskar effektförbrukningen?
- c) Vad är fel om begreppet minnesmapp för en dator:
  - A) En bild av hur den tillgängliga adressrymden för en dator används.
  - B) En dator med en 16 bitars adressbuss kommer att ha en minnesmapp bestående av totalt 32756 adresser.
  - C) I minnesmappen specificeras ofta vart läsminnets, skriv-läsminnets respektive I/O-minnets adressområden.
  - D) En processor med von Neuman arkitektur har en minnesmapp och en dator med Harvard-arkitektur har två minnesmappar,
- d) En maskencykel är på 250 nanosekunder, vilken klockfrekvens matas processorn med?
- e) Registerparet som betecknas med Z vilka register är det frågan om?

## **4.4 Svar**

### **U1: Teorifrågor om datorarkitektur**

- a) Se sidan 67 och figuren om Harvard-arkitektur.
- b) Se sidan 68 och figuren om von-Neumann-arkitektur
- c) Complex Instruction Set Computer
- d) Reduced Instruction Set Computer

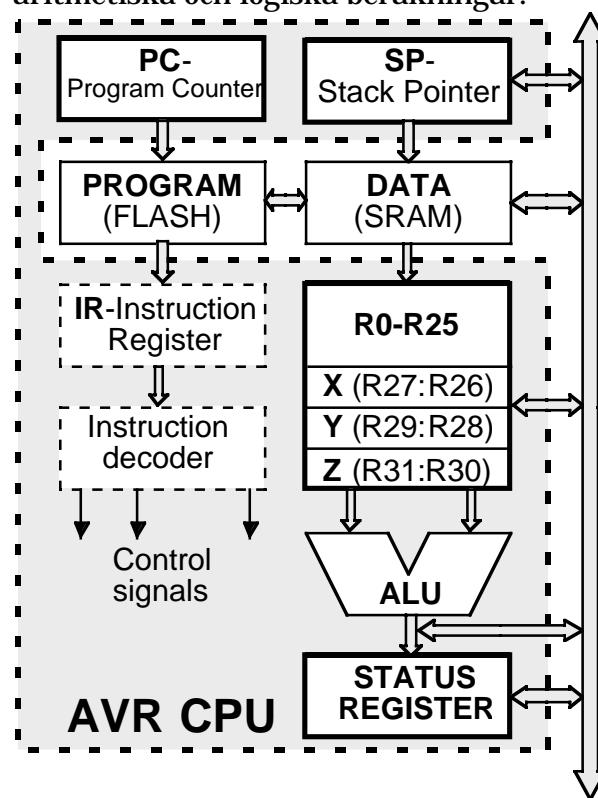
### **U2: Teorifrågor om AVR RISC-processorer**

- a) 3 stycken. Dataminne, programminne och EEPROM-minne.
- b) Effektförbrukningen blir 36% av vad den hade varit med den högre matningsspänningen. Minskningen blir alltså 74%.
- c) B)
- d) 4MHz.
- e) R31:R30

# Instruktioner som AVR-processorn kan utföra

## 5.1 Principerna för AVR RISC CPU:n

Vi skall i detta avsnitt gå igenom principerna för hur en CPU eller med synonyma ord centralenheten eller processorn fungerar. Vi slår fast ännu en gång att en CPU är en beräkningsmaskin för aritmetiska och logiska beräkningar.



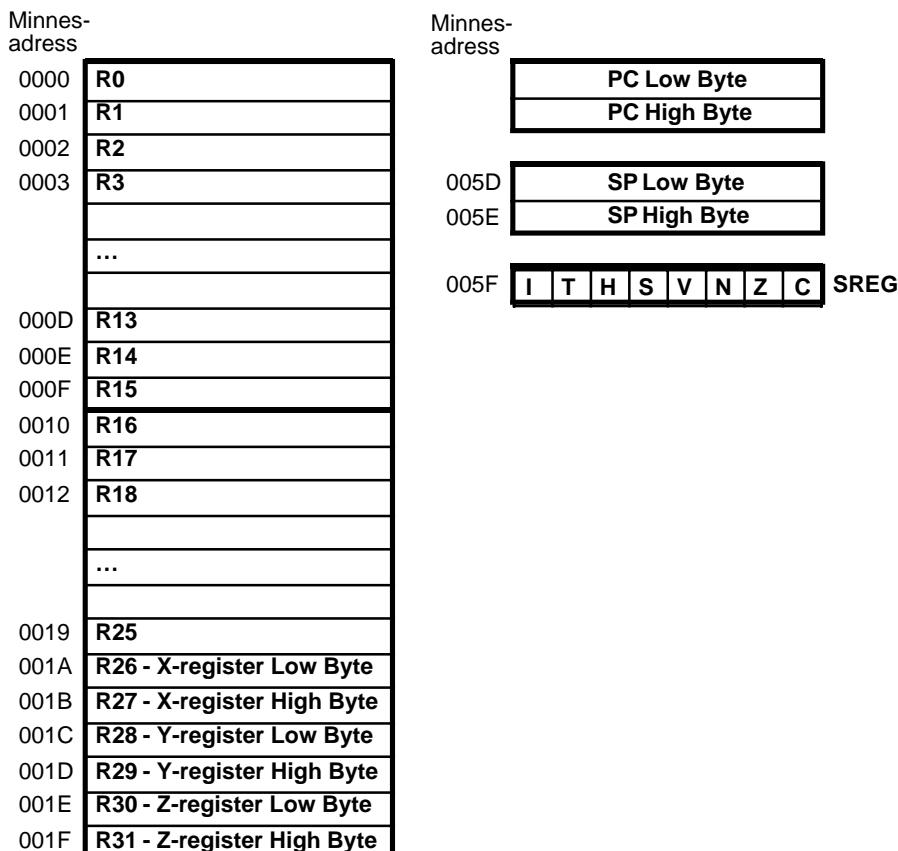
*Figur 5.1: CPU*

Det som karakteriseras AVR-processorn är att den har en Harvard-arkitektur samt att den är av typen RISC. Harvardarkitekturen har som grundläggande ide att ha separata datavägar för att överföra instruktioner och data till processorn. Instruktionshämtning och instruktionsexekvering sker alltså parallellt. Medan en

instruktion exekveras hämtas nästa instruktion i programmet in till processorn. Detta gör att en instruktion kan exekveras på en klockcykel. Processorn har en enkel uppbyggnad, som är ett av RISC-arkitekturens kännetecken.

### 5.1.1 Registerfil på 32 generella register på en byte vardera

Processorn har en *registerfil* bestående av 32 stycken generella register på en byte vardera. Accesstiderna för skrivning respektive läsning är på en klockcykel.



Figur 5.2: Programmerarens modell av AVR CPU:n

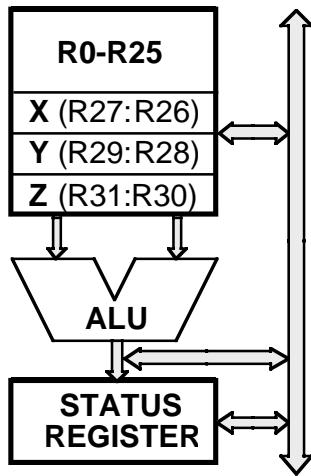
Stackpekarregistret och statusregistret SREG har dataminnesadress däremot har programräknarregistret ingen dataminnesadress.

Sex av registrena: R26 – R31 kan även användas som tre 16-bitars adressregister (pekare) och benämns då som X-, Y-, och Z-registrena. Användningen av dessa för adressgenerering ges i samband med genomgången av processorns instruktionsrapportar längre fram i detta kapitel.

### 5.1.2 ALU-enheten

Två godtyckligt valda register från registerfilen fungerar som operander i en ALU-operation. Resultatet från en ALU-operation sparas tillbaka till ett register i registerfilen. Hela operationen tar en klockcykel. ALU-enheten supportar aritmetiska och logiska operationer samt bitoperationer mellan register eller mellan ett register och en konstant. Statusregistret SREG uppdateras efter varje operation för att spegla

resultatet av operationen.



*Figur 5.3: ALU-enheten*

Följande typer av operationer kan utföras mellan ALU-enheten och registerfilen:

- En 8-bitars operand och ett 8-bitars resultat.
- Två 8-bitars operander och ett 8-bitars resultat.
- Två 8-bitars operander och ett 16-bitars resultat.
- En 16-bitars operand och ett 16-bitars resultat.

Operanderna hämtas från registerfilen och resultatet skrivs tillbaka till registerfilen.

### 5.1.3 Statusregistret - SREG

Statusregistret SREG innehåller information om resultatet av den senast exekverade ALU-operationen. Informationen är av typen: resultatet är lika med noll (Z-biten i registret), resultatet är negativt (N-biten i registret), etc. Informationen kan sedan användas för att realisera villkorlig programexekvering (if-then-else-satser) eller villkorliga programlooopar (while-satser, for-satser).



*Figur 5.4: Statusregistret SREG*

Statusregistret finns i dataminnet på adressen  $005F_{16}$ .

Statusregistret innehåller 8 styr och flaggbitar:

- **Bit 7 – I: Global Interrupt Enable**

Den globala avbrottssbiten *I* används för att tillåta respektive ej tillåta avbrottsgenerering av processorn. När denna bit är satt till 1 tillåts avbrott. Om biten är satt till 0 tillåts inga avbrott oavsett hur de individuella avbrottsskällorna (I/O-enheter oftast) är programmerade. *I*-biten kan manipuleras på tre olika sätt: Instruktionerna SEI (Set I, Global Interrupt Enable) och CLI (Clear I, Global Interrupt Disable) ger en direkt påverkan av biten. Ett avbrott nollställer *I*-biten automatiskt, detta görs genom hårdvaran i processorn.

- **Bit 6 – T: Bit Copy Storage**

Bitkopieringsinstruktionerna BLD (BitLoaD) och BST (BitSTore) använder  $T$ -biten som källa eller destination. En bit från ett register i registerfilen kan kopieras till  $T$ -biten med BST-instruktionen.  $T$ -biten kan sedan med BLD-instruktionen kopieras till en bit i ett register i registerfilen.

- **Bit 5 – H: Half Carry Flag**

Half Carry - flaggan  $H$  indikerar att en halv carry har inträffat. Flaggan bildas med hjälp av de fyra minst signifikanta bitarna i de två operanderna till ALU:n. Denna flagga är speciellt användbar tillsammans med BCD-aritmetik, där fyra bitar används vid kodning av en decimal siffra,

- **Bit 4 – S: Sign Bit,  $S = N \oplus V$**

Teckenbiten  $S$ -biten är alltid en exklusivt eller operation mellan negativflaggan  $N$  och flaggan för 2-komplementspill  $V$ .

- **Bit 3 – V: Two's Complement Overflow Flag**

2-kopmlementspillflaggan  $V$  stödjer 2-komplementaritmetik.

- **Bit 2 – N: Negative Flag**

Negativ-flaggan  $N$  indikerar att ett resultat har blivit negativt i en aritmetisk eller logisk operation.

- **Bit 1 – Z: Zero Flag**

Zero-flaggan  $Z$  indikerar att ett resultat har blivit noll i en aritmetisk eller logisk operation.

- **Bit 0 – C: Carry Flag**

Carry-flaggan  $C$  indikerar en minnessiffra i en aritmetisk eller logisk operation.

### 5.1.4 Programräknaren PC (instruktionspekarregister)

En *instruktion* är en operation som processorn utför. Instruktionerna eller med ett annat ord programmet finns i programminnet (FLASH-minne i AVR-processorn).

*Programräknaren – program counter* är ett register för *programstyrning*. Detta register innehåller adressen till instruktionen som skall hämtas härnäst.

Programräknarregistret brukar i processorer av märket Intel benämñas instruktionspekarregistret, vilket säger lite mer om funktionen hos detta register, registret pekar alltså på nästa instruktion i programmet som skall exekveras.

### 5.1.5 Stackpekarregistret – SP

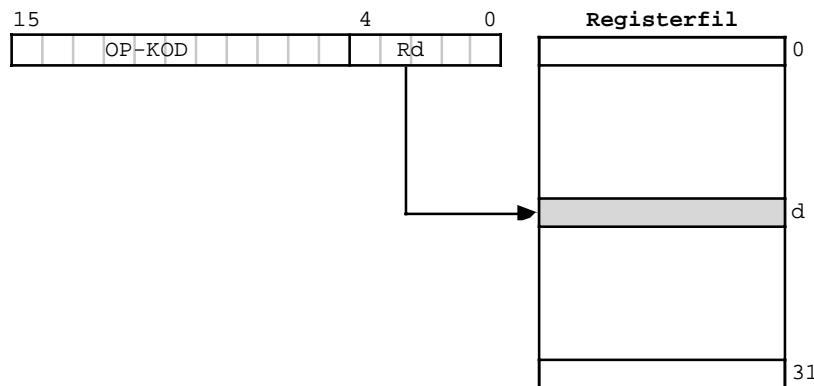
Stackpekarregistret *SP* används som adresspekarregister för att administrera en datastack i processorn. Stacken används för 3 olika ändamål: Återhoppsadresser sparas vid funktionsanrop (subrutinanrop) för att kunna hitta tillbaka till anropsstället i programmet efter avslutat funktionsanrop. Lokala variabler till en funktion kan finnas på stacken om processorns register ej räcker till. Det tredje området för stackens användning är för att kunna implementera avbrott av ett exekverande program för att sedan kunna exekvera en avbrottsfunktion. Stackens användning kommer vi att komma tillbaka till längre fram i kompendiet.

## 5.2 Instruktioner för att hämta och lagra data

I detta delkapitel skall vi ta upp några grundläggande instruktioner för att hämta in data till processorn från minnet, bearbeta data och sedan spara resultatet i minnet. Detta är också kärnan i ett program.

### 5.2.1 Direkt adressering av ett register

De flesta instruktionerna innehåller någon form av adressinformation för att adressera de tre olika adressrymderna i processorn: Programminnet, dataminnet och EEPROM-minnet. Den enklaste formen av adressering är direkt adressering av ett register i registerfilen (de 32 första adresserna i dataminnet). Instruktionskoden för en instruktion av denna adresseringstyp består av 1 ord (2 bytes) med 2 fält. Fält 1 är operationskoden (OP-KOD) och fält 2 (Rd) är adressering av ett register i registerfilen.

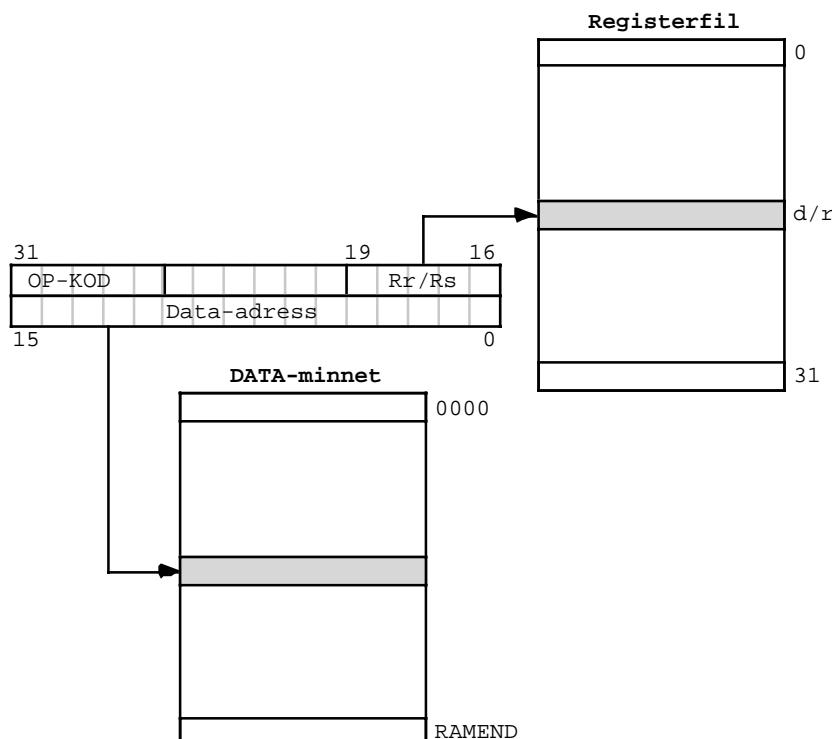


**Figur 5.5:** Direkt adressering av ett register

Utvärt register Rd fungerar som en operand till instruktionen med instruktionskoden OP-KOD.

### 5.2.2 Direkt dataadressering

Instruktioner som gör en direktadressering av dataminnet är på 2 ord (4 bytes). Dataadressen är på 16 bitar. Förutom detta så innehåller instruktionen en angivelse av destinations-/käll-register på 5 bitar. Typiska instruktioner för direktadressering av dataminnet är LDS- och STS-instruktionerna som beskrivs längre fram i detta kapitel.



*Figur 5.6: Direkt dataadressering*

### 5.2.3 Ladda ett register direkt från dataminnet

Instruktionen *LDS – LoaD direct from data Space* laddar en byte från dataminnet till ett av registrenna 0 till 31 i registerfilen. Dataminnet byggs upp av tre delar: registerfilen, I/O-registren och SRAM-minnet.

**Operation:**

$$\begin{aligned} \text{Rd} &\leftarrow (k) \\ \text{PC} &\leftarrow \text{PC} + 2 \end{aligned}$$

**Syntax:**

LDS Rd, k                                     $0 \leq d \leq 31, 0 \leq k \leq 65535$

**32 bitars op-kod:**

1	0	0	1	0	0	0	d	d	d	d	0	0	0	0
k	k	k	k	k	k	k	k	k	k	k	k	k	k	k

**Statusregistret SREG:**

I	T	H	S	V	N	Z	C
-	-	-	-	-	-	-	-

**Klockcykler:**

2

**Exempel:**

LDS R4, 0x00F0 ;Ladda R4 från adress 0x00F0 i

```

ADD      R4,R3      ;dataminnet
        ;Addera R3 till R4
STS      0x00F0,R4  ;Spara tillbaka till
                    ;dataminnet address 0x00F0

```

### 5.2.4 Ladda register 16 till 31 med en 8 bitars konstant

Instruktionen *LDI – Load Immediate* laddar en 8 bitars konstant direkt till ett av registrena 16 till 31

**Operation:**

$Rd \leftarrow K$   
 $PC \leftarrow PC + 1$

**Syntax:**            LDI     $Rd, K$                        $16 \leq d \leq 31, 0 \leq K \leq 255$

**16 bitars op-kod:**

1	1	1	0	K	K	K	K	d	d	d	d	K	K	K	K
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Observera att fältet för att ange register är på 4 bitar.  
För att få aktuellt register som används finns en offset-skillnad på 16:  $Rd = d + 16$ .

**Statusregistret SREG:**

I	T	H	S	V	N	Z	C
-	-	-	-	-	-	-	-

**Klockcykler:**            1

**Exempel:**            LDI     $R17, 32$     ;Konstanten 32 ladda till R17  
                        LDI     $R18, 0xF0$    ;Konstanten 0xF0 ladda till R1

### 5.2.5 Spara ett register direkt i dataminnet

Instruktionen *STS – STore direct to data Space* sparar ett av registrena 0 till 31 direkt i dataminnet. Dataminnet byggs upp av tre delar: registerfilen<sup>1</sup>, I/O-register och SRAM-minnet.

**Operation:**

$(k) \leftarrow Rd$   
 $PC \leftarrow PC + 2$

**Syntax:**            STS     $k, Rd$                        $0 \leq d \leq 31, 0 \leq k \leq 65535$

**32 bitars op-kod:**

1	0	0	1	0	0	1	d	d	d	d	0	0	0	0
k	k	k	k	k	k	k	k	k	k	k	k	k	k	k

**Statusregistret SREG:**

I	T	H	S	V	N	Z	C
-	-	-	-	-	-	-	-

**Klockcykler:**            2

**Exempel:**            LDS     $R4, 0x00F0$    ;Ladda R4 från adress 0x00F0 i  
                        ;dataminnet

<sup>1</sup>Registerfilen är registrena R0 till R31

```
ADD      R4,R3      ;Addera R3 till R4
STS      0x00F0,R4  ;Spara tillbaka till
                  ;dataminnet address 0x00F0
```

## 5.2.6 Sammanfattning av instruktioner för dataöverföring

Det finns en mängd olika instruktioner för dataöverföring. Move-instruktionerna (MOV,MOVW) för att kopiera data mellan olika register, load-instruktioner (LDI,LD,LPM) för att ladda ett register med information från dataminnet eller programminnet samt store-instruktioner (ST,STS,STD) skriva från ett register till dataminnet.

Mnemonic	O p.	Beskrivning	Funktion	Flaggor
MOV	Rd, Rr	Move Between Registers	$Rd \leftarrow Rr$	None
MOVW	Rd, Rr	Copy Register Word	$Rd+1:Rd \leftarrow Rr+1:Rr$	None
LDI	Rd, K	Load Immediate	$Rd \leftarrow K$	None
LD	Rd, X	Load Indirect	$Rd \leftarrow (X)$	None
LD	Rd, X+	Load Indirect and Post-Inc.	$Rd \leftarrow (X), X \leftarrow X + 1$	None
LD	Rd, -X	Load Indirect and Pre-Dec.	$X \leftarrow X - 1, Rd \leftarrow (X)$	None
LD	Rd, Y	Load Indirect	$Rd \leftarrow (Y)$	None
LD	Rd, Y+	Load Indirect and Post-Inc.	$Rd \leftarrow (Y), Y \leftarrow Y + 1$	None
LD	Rd, -Y	Load Indirect and Pre-Dec.	$Y \leftarrow Y - 1, Rd \leftarrow (Y)$	None
LDD	Rd, Y+q	Load Indirect with Displacement	$Rd \leftarrow (Y + q)$	None
LD	Rd, Z	Load Indirect	$Rd \leftarrow (Z)$	None
LD	Rd, Z+	Load Indirect and Post-Inc.	$Rd \leftarrow (Z), Z \leftarrow Z + 1$	None
LD	Rd, -Z	Load Indirect and Pre-Dec.	$Z \leftarrow Z - 1, Rd \leftarrow (Z)$	None
LDD	Rd, Z+q	Load Indirect with Displacement	$Rd \leftarrow (Z + q)$	None
LDS	Rd, k	Load Direct from SRAM	$Rd \leftarrow (k)$	None
ST	X, Rr	Store Indirect	$(X) \leftarrow Rr$	None
ST	X+, Rr	Store Indirect and Post-Inc.	$(X) \leftarrow Rr, X \leftarrow X + 1$	None
ST	X, Rr	Store Indirect and Pre-Dec.	$X \leftarrow X - 1, (X) \leftarrow Rr$	None
ST	Y, Rr	Store Indirect	$(Y) \leftarrow Rr$	None
ST	Y+, Rr	Store Indirect and Post-Inc.	$(Y) \leftarrow Rr, Y \leftarrow Y + 1$	None
ST	Y, Rr	Store Indirect and Pre-Dec.	$Y \leftarrow Y - 1, (Y) \leftarrow Rr$	None
STD	Y+q, Rr	Store Indirect with Displacement	$(Y + q) \leftarrow Rr$	None
ST	Z, Rr	Store Indirect	$(Z) \leftarrow Rr$	None
ST	Z+, Rr	Store Indirect and Post-Inc.	$(Z) \leftarrow Rr, Z \leftarrow Z + 1$	None
ST	Z, Rr	Store Indirect and Pre-Dec.	$Z \leftarrow Z - 1, (Z) \leftarrow Rr$	None
STD	Z+q, Rr	Store Indirect with Displacement	$(Z + q) \leftarrow Rr$	None
STS	k, Rr	Store Direct to SRAM	$(k) \leftarrow Rr$	None
LPM		Load Program Memory	$R0 \leftarrow (Z)$	None
LPM	Rd, Z	Load Program Memory	$Rd \leftarrow (Z)$	None
LPM	Rd, Z+	Load Program Memory and	$Rd \leftarrow (Z), Z \leftarrow Z + 1$	None
SPM		Store Program Memory	$(Z) \leftarrow R1:R0$	None
IN	Rd, P	In Port	$Rd \leftarrow P$	None
OUT	P, Rr	Out Port	$P \leftarrow Rr$	None
PUSH	Rr	Push Register on Stack	$STACK \leftarrow Rr$	None
POP	Rd	Pop Register from Stack	$Rd \leftarrow STACK$	None

**Figur 5.7:** Dataöverföringsinstruktioner

## 5.3 Instruktioner för att bearbeta data

AVR-processorn har ett flertal aritmetiska och logiska instruktioner som arbetar med 8 bitars data (bytedata), men det finns även några instruktioner som hanterar addition och subtraktion av 16 bitars konstanter. Multiplikationsinstruktioner för 8 bitar gånger 8 bitar finns i ett antal varianter. Vilka utgör basen för att kunna multiplicera tal med större bitbredd än 8 bitar.

Mnemonic	Op.	Beskrivning	Funktion	Flaggor
ADD	Rd, Rr	Add two Registers	$Rd \leftarrow Rd + Rr$	Z,C,N,V,H
ADC	Rd, Rr	Add with Carry two Registers	$Rd \leftarrow Rd + Rr + C$	Z,C,N,V,H
ADIW	Rdl,K	Add Immediate to Word	$Rdh:Rdl \leftarrow Rdh:Rdl + K$	Z,C,N,V,S
SUB	Rd, Rr	Subtract two Registers	$Rd \leftarrow Rd - Rr$	Z,C,N,V,H
SUBI	Rd, K	Subtract Constant from Register	$Rd \leftarrow Rd - K$	Z,C,N,V,H
SBC	Rd, Rr	Subtract with Carry two	$Rd \leftarrow Rd - Rr - C$	Z,C,N,V,H
SBCI	Rd, K	Subtract with Carry Constant	$Rd \leftarrow Rd - K - C$	Z,C,N,V,H
SBIW	Rdl,K	Subtract Immediate from Word	$Rdh:Rdl \leftarrow Rdh:Rdl - K$	Z,C,N,V,S
AND	Rd, Rr	Logical AND Registers	$Rd \leftarrow Rd \bullet Rr$	Z,N,V
ANDI	Rd, K	Logical AND Register and	$Rd \leftarrow Rd \bullet K$	Z,N,V
OR	Rd, Rr	Logical OR Registers	$Rd \leftarrow Rd \vee Rr$	Z,N,V
ORI	Rd, K	Logical OR Register and	$Rd \leftarrow Rd \vee K$	Z,N,V
EOR	Rd, Rr	Exclusive OR Registers	$Rd \leftarrow Rd \oplus Rr$	Z,N,V
COM	Rd	One's Complement	$Rd \leftarrow \$FF - Rd$	Z,C,N,V
NEG	Rd	Two's Complement	$Rd \leftarrow \$00 - Rd$	Z,C,N,V,H
SBR	Rd,K	Set Bit(s) in Register	$Rd \leftarrow Rd \vee K$	Z,N,V
CBR	Rd,K	Clear Bit(s) in Register	$Rd \leftarrow Rd \bullet (\$FF - K)$	Z,N,V
INC	Rd	Increment	$Rd \leftarrow Rd + 1$	Z,N,V
DEC	Rd	Decrement	$Rd \leftarrow Rd - 1$	Z,N,V
TST	Rd	Test for Zero or Minus	$Rd \leftarrow Rd \bullet Rd$	Z,N,V
CLR	Rd	Clear Register	$Rd \leftarrow Rd \oplus Rd$	Z,N,V
SER	Rd	Set Register	$Rd \leftarrow \$FF$	None
MUL	Rd, Rr	Multiply Unsigned	$R1:R0 \leftarrow Rd \times Rr$	Z,C
MULS	Rd, Rr	Multiply Signed	$R1:R0 \leftarrow Rd \times Rr$	Z,C
MULSU	Rd, Rr	Multiply Signed with Unsigned	$R1:R0 \leftarrow Rd \times Rr$	Z,C
FMUL	Rd, Rr	Fractional Multiply Unsigned	$R1:R0 \leftarrow (Rd \times Rr) \ll 1$	Z,C
FMULS	Rd, Rr	Fractional Multiply Signed	$R1:R0 \leftarrow (Rd \times Rr) \ll 1$	Z,C
FMULSU	Rd, Rr	Fractional Multiply Signed with	$R1:R0 \leftarrow (Rd \times Rr) \ll 1$	Z,C

**Figur 5.8:** Aritmetiska och logiska instruktioner

På grund av processorns uppbyggnad är antalet instruktionskoder begränsat. Detta gör att en instruktion som SUBI (subtrahera ett konstant värde från ett register) finns men ej som kanske förväntat en motsvarande instruktion för addition ADDI. Detta innebär att om du vill addera ett positivt värde till ett register måste du göra detta genom att använda SUBI-instruktionen.

### ■ Exempel 5.1: Addition med hjälp av subtraktion

Antag att vi vill addera konstanten +10 till registret R18. Tyvärr så existerar inte instruktionen ADDI men ändå kan man använda instruktionen SUBI. Vi får alltså utföra additionen som en subtraktion:

$$R18 = R18 + 10 \Leftrightarrow R18 = R18 - (-10)$$

Den decimala heltalskonstanten +10 kan skrivas binärt som  $00001010_2 = 0A_{16}$ , heltalskonstanten -10 fås binärt med 2-komplementrepresentation till  $11110110_2 = F6_{16}$ .

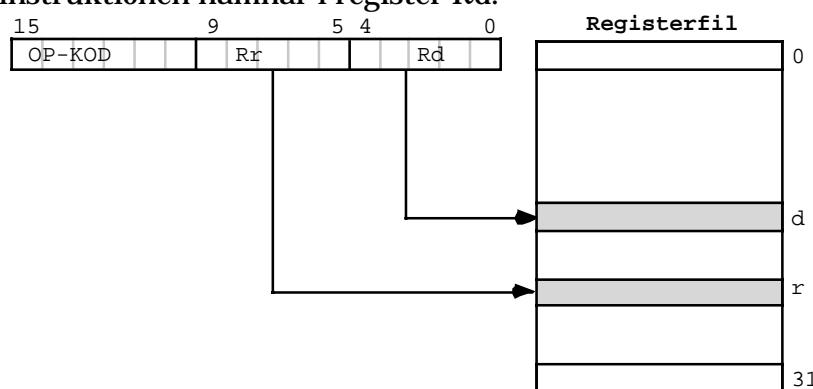
Assemblerkoden för addition av en konstant till ett register med hjälp av subtraktion blir:

LÄGE	OPERATION	OPERAND	KOMMENTAR
SUBI	R18 , 0xF6	; R18 $\leftarrow$ R18 - 0xF6	

**Slut exempel 5.1** ■

### 5.3.1 Direkt adressering av 2 register

Många instruktioner kräver två operander, till exempel addition. Operanderna hämtas från registerfilen. De två operanderna anges av fälten Rr och Rd i instruktionen. Resultatet av instruktionen hamnar i register Rd.



Figur 5.9: Direkt adressering med två register

### 5.3.2 Addition utan minnessiffra (carry)

Instruktionen ADD – ADD without carry adderar två register utan att ta hänsyn till carry-flaggan C.

**Operation:**  $Rd \leftarrow Rd + Rr$   
 $PC \leftarrow PC + 1$

**Syntax:** ADD Rd ,Rr       $0 \leq d \leq 31, 0 \leq d \leq 31$

**16 bitars op-kod:**

0	0	0	0	1	1	r	d	d	d	d	r	r	r	r
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

**Statusregistret SREG:**

I	T	H	S	V	N	Z	C
-	-	↔	↔	↔	↔	↔	↔ <sup>1</sup>

Flaggorna bildas utifrån följande ekvation:  $R = Rd + Rr$

$$H = R_d \cdot R_r + R_r \cdot \overline{R}_d + \overline{R}_r \cdot R_d$$

$$S = N \oplus V$$

$$V = R_d \cdot R_r \cdot \overline{R}_7 + \overline{R}_d \cdot R_r \cdot \overline{R}_7$$

$$N = R_7$$

$$Z = \overline{R}_7 \cdot \overline{R}_6 \cdot \overline{R}_5 \cdot \overline{R}_4 \cdot \overline{R}_3 \cdot \overline{R}_2 \cdot \overline{R}_1 \cdot \overline{R}_0$$

$$C = R_d \cdot R_r \cdot \overline{R}_7 + R_r \cdot \overline{R}_7 + \overline{R}_7 \cdot R_d$$

**Klockcykler:** 1

**Exempel:** ADD R2,R1 ; R2=R2+R1  
ADD R4,R4 ; Addera R4 till sig själv  
; R4=R4+R4

### ■ Exempel 5.2: Addition av två heltalskonstanter

Antag att vi skall addera heltalen 25 och 10. Resultatet skall sedan skrivas till adress 0x0100, vilket är adressen för en enbytes global variabel med namnet summa. Uttryckt i högnivåspråket C är det följande vi skall beräkna:

```
unsigned char summa;
summa = 25+10;
```

Följande assemblerprogram utför ovanstående:

<u>LÄGE</u>	<u>OPERATION</u>	<u>OPERAND</u>	<u>KOMMENTAR</u>
-------------	------------------	----------------	------------------

LDI	R2, 25	; R2 ← 0x19	R2 <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>19</td></tr></table>	19
19				
LDI	R1, 10	; R1 ← 0x0A	R1 <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0A</td></tr></table>	0A
0A				
ADD	R2, R1	; R2 ← R2 + R1 ,	R2 <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>23</td></tr></table>	23
23				
STS	0x0100, R2	;(0x0100)← R2	0100 <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>23</td></tr></table>	23
23				

Ovanstående programkod skulle kunna skrivas om för att förtydligas lite med hjälp av assemblerdirektivet .equ (equate), med vilkens hjälp det är möjligt att namnge numeriska värden. Det numeriska värdet representeras då av en alfanumerisk symbol, med hjälp av detta direktiv kan assemblerprogrammet göras mer förklarande.

<u>LÄGE</u>	<u>OPERATION</u>	<u>OPERAND</u>	<u>KOMMENTAR</u>
-------------	------------------	----------------	------------------

<sup>1</sup> ↔ betyder att flaggans värde uppdateras, – betyder att flaggans värde är oförändrad.

```

.equ      K1, 25
.equ      K2, 10
.equ      summa, 0x0100

LDI      R2, K1
LDI      R1, K2
ADD      R2, R1
STS      summa, R2

```

Observera att programmet är ekvivalent med föregående, maskinkodsrepresentationen av programmet är identiskt i båda fallen.

**Slut exempel 5.2** ■

## 5.4 Några hoppinstruktioner för ovillkorliga hopp

Ett hopp i ett program innebär att programräknaren *PC* laddas med ett helt nytt värde. Detta bryter det sekventiella exekveringsförloppet där programräknaren stegas upp för att peka på nästa instruktion.

De program som vi skriver innehåller alltid en cyklick exekvering, d.v.s. programmet körs om och om igen. Vi har behov av att kunna implementera en oändlig loop som håller igång den programkod som skall exekveras cyklick. En loop i ett program innebär att ett hopp måste utföras i loopens slut till loopens start.

### 5.4.1 JMP – JuMP

Instruktionen *JMP – JuMP* kan hoppa till godtycklig address i programminnet. Hoppet är ett så kallat absoluthopp där hoppadressen anges med 22 bitar, vilket ger oss en hoppräckvidd på  $4M^1$  (megaord). Hoppet är ovillkorligt, dvs det utförs alltid.

**Operation:**  $PC \leftarrow k$

**Syntax:**  $JMP \quad k \quad 0 \leq k \leq 4M$

**32 bitars op-kod:**

1	0	0	1	0	1	0	k	k	k	k	1	1	0	k
k	k	k	k	k	k	k	k	k	k	k	k	k	k	k

**Statusregistret SREG:**

I	T	H	S	V	N	Z	C
-	-	-	-	-	-	-	-

**Klockcykler:** 3

**Exempel:**  $MOV \quad R1, R0 \quad ; \quad R1=R0$   
 $JMP \quad farplace \quad ; \quad \text{Ovillkorligt hopp}$

<sup>1</sup> $1K=2^{10}$ ,  $1M=1K \times 1K=2^{20}$

## 5.4.2 RJMP – Relative JuMP

Instruktionen *RJMP – Relative JuMP* kan hoppa relativt programräknarens aktuella värde med ett offset inom intervallet -2048 till 2047

**Operation:**  $PC \leftarrow PC + k + 1$

**Syntax:** RJMP k  $-2048 \leq k \leq +2047$

**16 bitars op-kod:**

1	1	0	0	k	k	k	k	k	k	k	k	k	k	k	k
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

**Statusregistret SREG:**

I	T	H	S	V	N	Z	C
-	-	-	-	-	-	-	-

**Klockcykler:** 3

**Exempel:**

error:	CPI	R16,0x42 ; Jämför R16 med 0x42
	BRNE	error ; Hopp om R16 ≠ 0x42
	RJMP	ok ; Ovillkorligt hopp
	ADD	R16,R17 ; R16=R16+R17
	INC	R16 ; Addera +1 till R16
ok:	NOP	

I exemplet används instruktionen BRNE – BRanch Not Equal som är ett exempel på en instruktion för villkorligt hopp. En instruktion för ett villkorligt hopp utförs om vissa villkor på flaggorna i statusregistret SREG är uppfyllda annars så fortsätter exekveringen med instruktionen efter BRNE. I exemplet ovan sker det villkorliga hoppet till instruktionen ADD. Villkorliga hopp kommer att behandlas i ett kapitel längre fram.

### ■ Exempel 5.3: Oändlig loop

Antag att vi har följande C-program innehållande en oändlig loop för cyklisk exekvering:

```
whi l e ( 1 )
{
    //...
}
```

Assemblerkodning av detta skulle kunna göras på följande sätt, C-koden har vi med som kommentar:

LÄGE	OPERATION	OPERAND	KOMMENTAR
;	;	---	<u>while</u> ( 1 )
;	;		{
while_loop:			
;	;	•••	
;	;		}

```
RJMP      while_loop
```

**Slut exempel 5.3** ■

## 5.5 Subrutinanrop utan parameteröverföring

Vi har behov av att kunna modularisera vårt program i form av subrutiner/funktioner. På assemblernivå så kallas vi en funktion för en *subrutin*. I ett inledningsskede kommer vi att använda subrutiner som ej kräver några parametrar. Betraktat som en C-funktion kommer vi att ha följande prototyp för denna:

```
void Func();
```

d.v.s. inga parametrar vid funktionsanropet och funktionen returnerar inget (void).

### Funktionsdefinition

```
void Func()
{
    // ...
}
```

### Funktionsanrop

```
Func();
// nästa programsats
```

### Subrutindefinition

```
Func:
;
RET //Return, återvänd
```

### Subrutinanrop

```
RCALL Func
; nästa instruktion
```

**Figur 5.10:** Jämförelse mellan funktions- och subroutinebegreppen

Ett subrutinanrop är ett hopp i programmet. För att utföra hoppet används instruktionen RCALL eller CALL, där ordet CALL står för anrop av en subrutin. Till skillnad mot vanliga hopp så har vi en restriktion, vi måste på något sätt efter att subrutinanropet har utförts hitta tillbaka till instruktionen efter RCALL/CALL. Ett subrutinanrop med t. ex. RCALL innebär förutom hoppet, att även programräknaren PC får ett nytt värde samt att en *återhoppsadress* till instruktionen efter RCALL sparas på stacken. Vi kommer att behandla detta mer noggrannt längre fram. En subrutin avslutas alltid med instruktionen RET, som står för RETurn, d.v.s. återvänd. Informationen varit vi skall återvända finns sparad överst på stacken i form av en återhoppsadress som programräknaren PC laddas med.

### 5.5.1 RCALL – Relative CALL to subroutine

Instruktionen *RCALL – Relative CALL to subroutine* kan utföra ett subrutinanrop relativt programräknarens aktuella värde med ett offset inom intervallet -2048 till 2047. Om subrutinen ej ligger inom adressintervallet  $[PC - 2048 + 1, PC + 2048]$  får instruktionen CALL användas istället.

**Operation:**  $SP \leftarrow SP - 2$  Öka på stacken med 2 bytes.  
 $M_{(SP+1)} : M_{(SP+2)} \leftarrow PC + 1$  Återhoppsadress sparas på stacken  
 $PC \leftarrow PC + k + 1$  Hopp till subrutinen, **relativhopp**

**Syntax:** RCALL k  $-2048 \leq k \leq +2047$

**16 bitars op-kod:**

1	1	0	1	k	k	k	k	k	k	k	k	k	k	k	k
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

**Statusregistret SREG:**

I	T	H	S	V	N	Z	C
-	-	-	-	-	-	-	-

**Klockcykler:** 3

**Exempel:** **RCALL** routine ; Anrop av subrutinen

...

**routine:** PUSH R14 ; Spara R14 på stacken  
 ...  
 POP R14 ; Återställ R14 från stacken  
**RET** ; Återvänd från subrutinanropet

### 5.5.2 RET – Relative CALL to subroutine

Instruktionen *RET – RETurn from subroutine* gör att vi återvänder tillbaka till instruktionen efter anropsstället. Programräknaren laddas med en återhoppsadress från stacken.

**Operation:**  $PC \leftarrow M_{(SP+1)} : M_{(SP+2)}$  Återhoppsadress laddas till PC  
 $SP \leftarrow SP + 2$  Frisläpp 2 bytes på stacken

**Syntax:** RET

**16 bitars op-kod:**

1	0	0	1	0	1	0	1	0	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

**Statusregistret SREG:**

I	T	H	S	V	N	Z	C
-	-	-	-	-	-	-	-

**Klockcykler:** 4

**Exempel:** **RCALL** routine ; Anrop av subrutinen

...

**routine:** LDI R20,1  
 ...  
**RET** ; Återvänd från subrutinanropet

## 5.6 Uppgifter

### U1: Teorifrågor om AVR-processorn

- a) Vilka typer av operationer tillåter AVR-processorn?
- b) Vilken funktion har programräknaren *PC*?
- c) Vilket alternativt namn brukar programräknarregistret ha i Intel-processorer?
- d) Hur genomförs ett absoluthopp i ett program?
- e) Vilken skillnad föreligger mellan ett relativhopp och ett absoluthopp?
- f) Ett subrutinanrop är det ett hopp eller ej?
- g) Antag att du vill addera konstanten +1 till registret R18, hur gör du?
- h) Varför finns inte instruktionen ADDI då instruktionen SUBI existerar?
- i) Då alla instruktioner i AVR-processorn upptar antingen 2 eller 4 byte så är det naturligt att räkna i ord istället för byte. Ett ord är lika med 2 byte. En minnesadress kan då antingen uppges som bytadress eller ordadress. För att få ordadressen delar man bytadressen med två och tar heltalsdelen av detta. Vilken ordadress får vi då bytadressen är 0x0100?
- j) Vilka bytadresser får vi då ordadressen är 0x0080?

### U2: Några frågor om instruktioner

- a) Skriv instruktionen för att ladda register R8 med konstanten 0xAA
- b) Skriv instruktionen för att ladda register R8 med från dataminnesadressen 0x0078
- c) Skriv instruktionen för att addera register R2 och R3, resultatet skall hamna i R3.
- d) Skriv instruktionen för att addera register R2 och R3, resultatet skall hamna i R2.
- e) Skriv instruktionerna för att addera två stycken 16 bitars tal i registerparen *R3 : R2* respektive *R5 : R4*. Resultatet skall läggas i registerparet *R3 : R2*.
- f) Skriv instruktionerna för att subtrahera två stycken 16 bitars tal i registerparen *R3 : R2* respektive *R5 : R4*. Resultatet skall läggas i registerparet *R3 : R2*.
- g) Bestäm resultatet och hur flaggorna N och Z påverkas då följande instruktioner exekveras:

LDI        R20, 0x87  
ANDI       R20, 0x78

- A) N=0 Z=0    B) N=0 Z=1    C) N=1 Z=1    D) N=1 Z=0

- h) I de följande uppgifterna skall du studera några olika instruktioner. Antag att minnesinnehållet är enligt följande:

7000	3B
7001	A5
7002	?
7003	?

Ange för var och en av följande uppgifter vad som sparas på adress 0x7002 i minnet.

LDS	R20, 0x7000
COM	R20
STS	0x7002, R20

- i) begi n:
- |     |             |
|-----|-------------|
| LDI | R20 , 0x0A  |
| LSL | R20         |
| LSL | R20         |
| STS | 0x7002, R20 |
- stop:
- |      |      |
|------|------|
| RJMP | stop |
|------|------|

- j) Skriv de assemblerinstruktioner som behövs för att invertera bit 5 i U utan att övriga bitar påverkas. Antag att U är adressen till en 1 bytes data i minnet:

U: . byte 0 ; 

D7	D6	D5	D4	D3	D2	D1	D0
----	----	----	----	----	----	----	----

- k) Skriv de assemblerinstruktioner som behövs för att ettställa bit 5 i U utan att övriga bitar påverkas.
- l) Skriv de assemblerinstruktioner som behövs för att nollställa bit 5 i U utan att övriga bitar påverkas.

### U3: Från C-kod till assemblerkod

- a) Skriv de instruktioner som behövs för att utföra följande tilldelningssats i C:

char a, b, c;

...

a = b+c;

Antag att variablerna a,b och c är globala och har dataminnesadresserna 0x00F0, 0x00F1 respektive 0x00F2.

- b) Skriv de instruktioner som behövs för att utföra följande tilldelningssats i C:

int a, b, c;

...

a = b+c;

Antag att variablerna a,b och c är globala och har dataminnesadresserna 0x00F0, 0x00F2 respektive 0x00F4.

## U4: Disassemblering

I följande frågor skall du dissassemblera maskinkoden och ange vilken instruktion/instruktioner det är fråga om samt ange operanderna till respektive instruktion.

- a) I minnet på byteaddress 0x0100 och 0x0101 ligger 2 bytes med data. Instruktionen är LDI, vilka operander har vi till denna?

0100	C0	E0
------	----	----

- b) Instruktionen är ANDI, vilka operander har vi till denna?

00F2	C7	70
------	----	----

## 5.7 Svar

### U1: Teorifrågor om AVR-processorn

- a) Aritmetiska, logiska och bit-operationer.
- b) -
- c) Instruktionspekarregister.
- d) Ett absoluthopp innebär att programräknaren laddas med ett helt nytt värde.
- e) -
- f) Hopp i programmet.
- g) Flera lösningar finns på detta problem, vi redovisar två av dem:

Addition av två register:

```
LDI      R19 , 0x01
ADD      R18 , R19      ; R18=R18+R19=R18+1
```

Direkt addition av en konstant till ett register. Tyvärr så finns ej instruktionen ADDI – ADD Immediate, ändemot så finns instruktionen SUBI – SUBtract Immediate. Vi kan då göra additionen av  $+1=0x01$  som en subtraktion av  $-1=0xFF$ :

```
SUBI    R18 , 0xFF      ; R18=R18-0xFF=R18-(-1)=R18+1
```

Tvåkomplementet till  $0x01$  är  $0xFF$ . Ett alternativt sätt att skriva instruktionen är på formen:

```
SUBI    R18 , -1
```

- h) Antalet instruktionskoder att använda är begränsade, därför slösar man ej dessa i onödan. Instruktionen ADDI som ej finns och SUBI är ekvivalenta, d.v.s. utbytbara mot varann, t.ex. så kan en addition av en konstant utföras som en subtraktion av en konstant.
- i)  $0x0100 / 2 = 0x0080$
- j)  $2 * 0x0080 = 0x0100$ . Vi har alltså byteadresserna  $0x0100$  och  $0x0101$

### U2: Några frågor om instruktioner

- a) LDI      R8, 0xAA

b) LDS R8, 0x0078

c) ADD R3, R2

d) ADD R2, R3

e) ADD R2, R4

ADC R3, R5

f) SUB R2, R4

SBC R3, R5

g) B) N=0 Z=1

h) 7002 C4i) 7002 28j) LDS R19, U  
LDI R20, 0x20  
EOR R19, R20  
STS U, R19k) LDS R19, U  
SBR R19, 0x20  
STS U, R19l) LDS R19, U  
CBR R19, 0x20  
STS U, R19

### U3: Från C-kod till assemblerkod

a) LDS R1, 0x00F0 ; R1=a  
     LDS R2, 0x00F1 ; R2=b  
     ADD R2, R1 ; R2=R2+R1=a+b  
     STS 0x00F1, R2 ; c=R2

b) LDS R0, 0x00F0 ; a low byte  
     LDS R1, 0x00F1 ; a high byte  
     LDS R2, 0x00F2 ; b low byte  
     LDS R3, 0x00F3 ; b high byte  
     ADD R0, R2 ; c=a+b=R0: R1  
     ADC R1, R3  
     STS 0x00F4, R0 ; (00F4)<-R0  
     STS 0x00F5, R1 ; (00F5)<-R1

### U4: Disassemblering

I minnet ligger följande maskinkod vilken instruktion eller instruktioner är det?

- a) LDI R28 , 0x00

Observera att vi har little endian lagring i minnet. Detta innebär att mest signifika byten (E0) lagras på högsta adressen och minst signifkanta (C0) på lägsta adressen.

0100 

C0	E0
----	----

 $\Rightarrow E0C0 \Rightarrow 1110\ 0000\ 1100\ 0000$

Identifiering med operationskodsdefinitionen för LDI

1	1	1	0	K	K	K	K	d	d	d	d	K	K	K	K
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

ger att  $K = 0x00$  och att  $d = 0xC = 12$ .

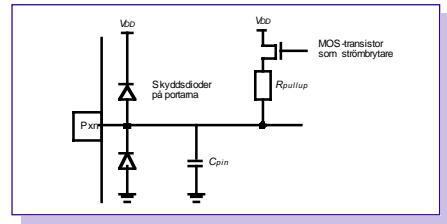
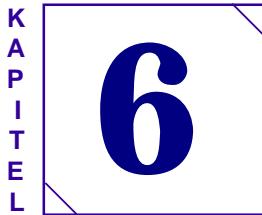
$$Rd = d + 16 = 12 + 16 = 28$$

- b) ANDI R28, 0x07

00F2 

C7	70
----	----

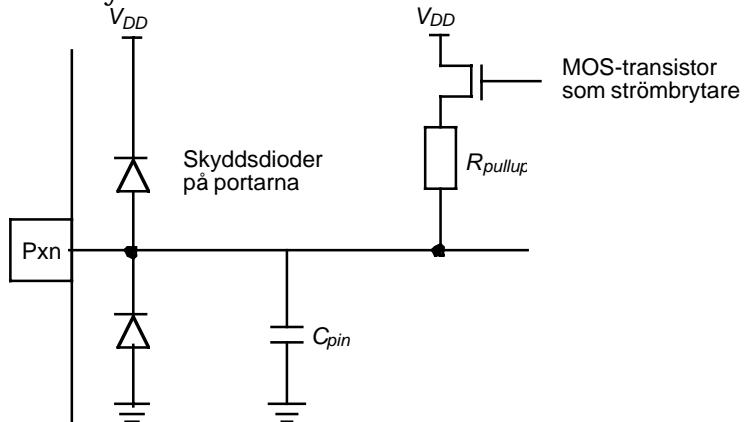
 $\Rightarrow 70C7 \Rightarrow 0111\ 0000\ 1100\ 0111$



## Digitala in- och utsignaler

AVR ATmega16 har 32 portar, vilka är uppdelade i 4 grupper med namnen PORTA (PA7-PA0), PORTB (PB7-PB0), PORTC (PC7-PC0) och PORTD(PD7-PD0). Varje grupp består alltså av 8 bitar, 1 byte. Den mest grundläggande funktionen för en port är som digital in- eller utgång. De flesta portarna kan även användas för alternativa funktioner som analog inport (0-5V), timeringång, PWM-utgång, seriell kommunikation, etc.

Elektriskt så skyddas en port mot för höga respektive för låga spänningar med hjälp av skyddsdioder som kortsluter antingen till jord eller till matningsspänningen  $V_{DD}$ . En port har förmåga att både driva respektive sänka en ström. Portens drivförmåga är konstruerad för att kunna driva en lysdiod, d.v.s. leverera tillräckligt med ström för att få ett bra ljusflöde från lysdioden.



**Figur 6.1:** Ekvivalent elektriskt schema för en portpinne på processorn

Om porten är programmerad att fungera som digital ingång finns möjlighet att programvässigt koppla in ett pull-up motstånd. Pull-up motståndets funktion är att ”dra upp” spänningen till att ligga i närheten av matningspänningen då den digitala insignalen är logiskt 1. Om ett pull-up motstånd skall kopplas in eller ej beror av hur den digitala insignalen är konstruerad.

Till varje portgrupp x (A,B,C eller D) finns det 3 stycken I/O-register associerat: PORTx, DDRx respektive PINx.

I/O-adress	Minnesadress	Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x1B	0x003B	<b>PORTA</b>	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0
0x1A	0x003A	<b>DDRA</b>	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0
0x19	0x0039	<b>PINA</b>	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0
0x18	0x0038	<b>PORTB</b>	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
0x17	0x0037	<b>DDRB</b>	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0
0x16	0x0036	<b>PINB</b>	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
0x15	0x0035	<b>PORTC</b>	PORTC7	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0
0x14	0x0034	<b>DDRC</b>	DDC7	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0
0x13	0x0033	<b>PINC</b>	PINC7	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0
0x12	0x0032	<b>PORTD</b>	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
0x11	0x0031	<b>DDRD</b>	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
0x10	0x0030	<b>PIND</b>	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0

**Figur 6.2:** Portar för digitala in- och utsignaler

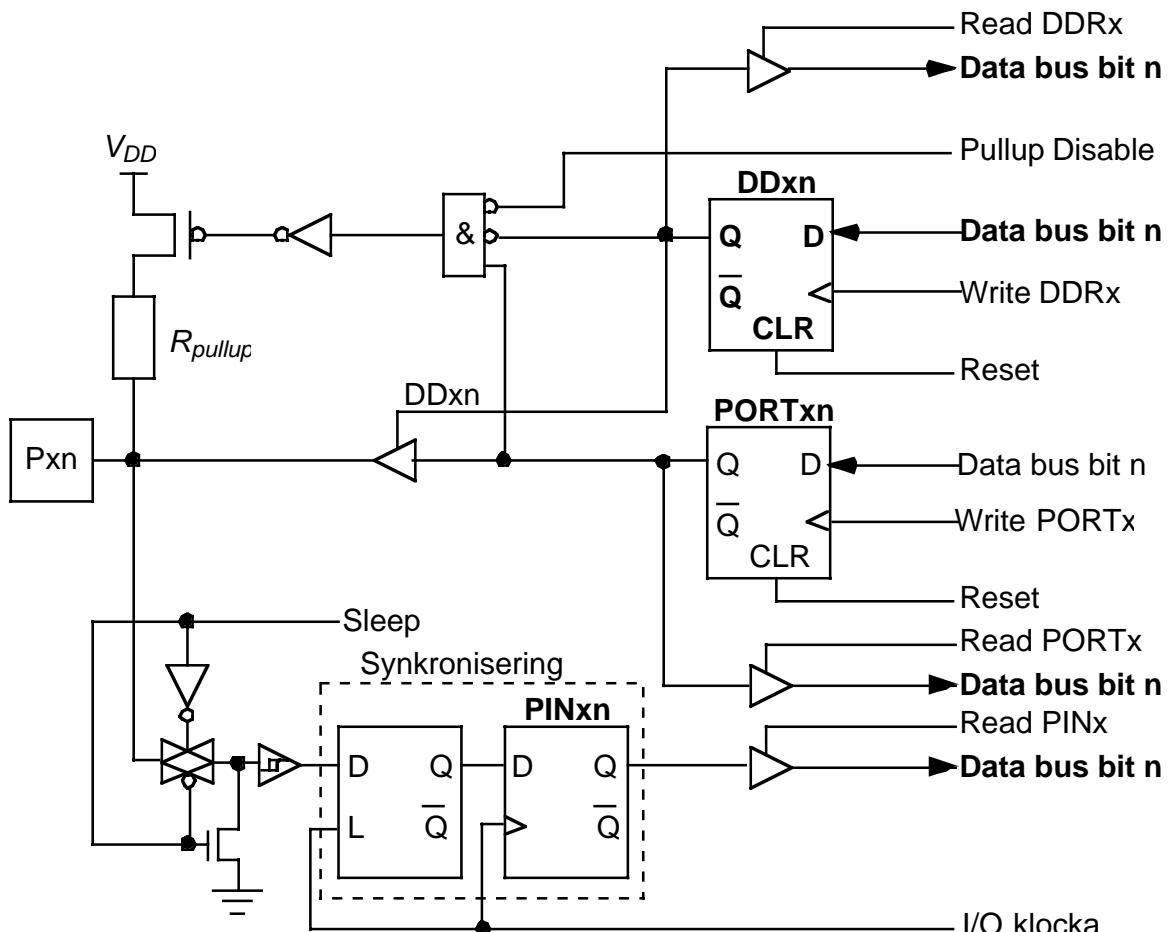
Registret *DDR<sub>x</sub>* – Data Direction Register *x* är registret som bestämmer vilka portpinnar som skall fungera som digitala ingångar respektive digitala utgångar. Varje enskild portpinne kan alltså programmeras individuellt. Om en portpinne har programmerats att fungera som digital utgång så skall korresponderande bit i *PORT<sub>x</sub>*-registret hålla detta utgångsvärde. Om en portpinne fungerar som digital ingång kan detta ingångsvärde avläsas i registret *PIN<sub>x</sub>* – Port IN *x*.

Totala antalet I/O-register i ATmega16 är 64 stycken. I/O-adressen är ett heltal i intervallet 0 till 63. Alla I/O-register finns också minnesmappade i dataminnet. Adressen i dataminnet ges av I/O-adressen plus ett offset på 0x20 (hexadecimalt).

Figuren nedan ger en logisk bild av hur en portpinne fungerar och hur pinnens funktion är relaterat till de tre registrena PORTx, DDRx och PINx. I varje register används 1 bit för en portpinne. Varje bit finns lagrat i en D-vippa. Totalt används 3 bitar per portpinne.

Bortkoppling av alla PULL UP motstånd för alla portar görs med biten *PUD – Pull up Disable* i registret *SFIOR – Special Function I/O Register*.

Biten DDxn respektive PORTxn är både skriv och läsbara, däremot så kan biten PINxn enbart läsas.



**Figur 6.3:** Logiskt schema för en portpinne på processorn

Vid RESET på processorn så läggs pinnen i ett tri-state-läge genom tvångssättning av bitarna {DDxn, PORTxn} till 0b00. Detta innebär att porten kan ses som ett avbrott ur en elektrisk synvinkel.

Om en portpinne skall fungera som utgång skall datariktningsbiten DDxn programmeras till en 1:a. Detta kopplar ut registerbiten PORTxn till portpinnen, genom att skriva 0 eller 1 till denna bit sätts utgångsvärdet.

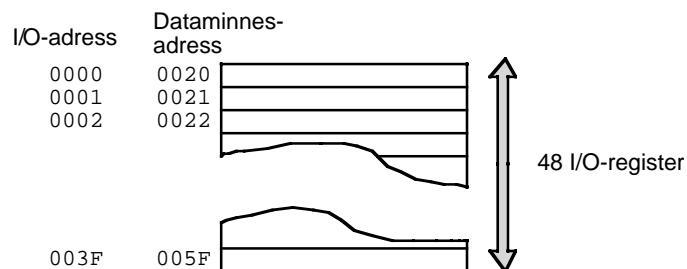
Om en portpinne skall fungera som ingång skall datariktningsbiten DDxn programmeras till en 0:a. Därefter kan portpinnens ingångsvärde avläsas via registerbiten PINx. Notera hur en D-latch används för att skapa en insignal till D-vippan PINxn som är stabil innan klockning av vippan tillåtes.

Oanslutna pinnar bör ha definierat värde, vilket görs bäst genom att koppla in det

interna pullup-motståndet.

## 6.1 Instruktioner för IO-registermanipulering

I/O-registrena i processorn ligger minnesmappade i dataminnet. Vid konstruktionen av AVR-processorn har man understött med instruktioner som kan manipulera med dessa I/O-register. Observera att I/O-registrena kan adresseras på två olika sätt: Antingen med instruktioner som använder I/O-adresseringsmoden eller med instruktioner som använder dataminnesadressering.

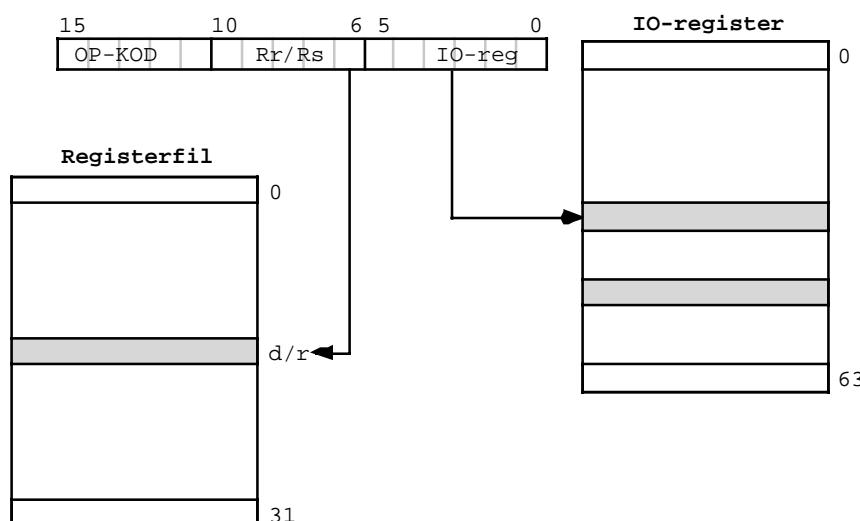


Figur 6.4: Datamine-smapp för ATmega16

Vi har en offsetskillnad mellan datamine-sadressen och I/O-adressen i storleken  $32=0x20$ . Flera instruktioner arbetar mot I/O-registrena med hjälp av I/O-adressen, det är alltså endast instruktioner som kan användas mot I/O-registrena.

### 6.1.1 I/O-adressering

Speciella instruktioner finns för att adressera den del av dataminnet där processorns I/O-register finns förlagda. I/O-registerarean är 64 byte i de flesta AVR-kretsarna. Förutom detta så innehåller instruktionen en angivelse av ett destinations-/källregister.



Figur 6.5: I/O-adressering

Då I/O-registerarean också är datamine-smappad så kan de vanliga instruktionerna för att läsa och skriva i dataminnet användas.

## 6.1.2 Ladda ett I/O-register till ett register i registerfilen

Instruktionen *IN* – *Load an I/O-location into a register* läser ett I/O-register till ett av registren 0 till 31

**Operation:**

$Rd \leftarrow I/O(A)$   
 $PC \leftarrow PC + 1$

**Syntax:**

IN  $Rd, A$   $0 \leq d \leq 31, 0 \leq A \leq 63$

**16 bitars op-kod:**

1	0	1	1	0	A	A	d	d	d	d	A	A	A	A
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

I T H S V N Z C

**Statusregistret SREG:**

-	-	-	-	-	-	-	-	-
---	---	---	---	---	---	---	---	---

**Klockcykler:**

1

**Exempel:**

```
IN      R14, 0x16    ;Läser PORTB till register R14
CPI     R14, 8       ;Jämför R14 med 8
BREQ   exit         ;Hopp om lika (BRanch EQual)
...
exit:
```

## 6.1.3 Skriv ett register i registerfilen till ett I/O-register

Instruktionen *OUT* – *Store register to I/O-location* skriver ett av registren 0 till 31 till ett I/O-register.

**Operation:**

$I/O(A) \leftarrow Rr$   
 $PC \leftarrow PC + 1$

**Syntax:**

OUT  $A, Rr$   $0 \leq r \leq 31, 0 \leq A \leq 63$

**16 bitars op-kod:**

1	0	1	1	1	A	A	r	r	r	r	A	A	A	A
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

I T H S V N Z C

**Statusregistret SREG:**

-	-	-	-	-	-	-	-	-
---	---	---	---	---	---	---	---	---

**Klockcykler:**

1

**Exempel:**

```
LDI     R12, 0xFF    ;R12 är lika med 0xFF
OUT    0x17, R12    ;Skriver till DDRB
```

### ■ Exempel 6.1: PORTD med pinnarna PD3-PD0 som utgångar och PD7-PD4 som ingångar

Exemplet visar på hur pinnarna PD7-PD4 i PORTDsätts till ingångar, med pull-up-motstånd anslutna till PD7 och PD6. Pinnarna PD3-PD0 sätts till utgångar med

värdena 0011.

<u>LÄGE</u>	<u>OPERATION</u>	<u>OPERAND</u>	<u>KOMMENTAR</u>
#define PIND	0x10		
#define DDRD	0x11		
#define PORTD	0x12		
; Define pull-ups and set outputs high			
; Define directions for port pins			
LDI	r17,0b00001111		
OUT	DDRD,r17		
LDI	r16,0xF0		
OUT	PORTD,r16		
; Insert nop for synchronization			
NOP			
; Read port pins			
IN	r16,PIND		

**Slut exempel 6.1 ■**

## 6.2 Assemblerinstruktioner för bithantering

För att manipulera och testa enskilda bitar finns det ett antal olika assemblerinstruktioner. I tabellen nedan sammanfattas dessa.

Mnemonic	Op.	Beskrivning	Funktion	Flaggor
LSL	Rd	Logical Shift Left	$Rd(n+1) \leftarrow Rd(n)$ , $Rd(0) \leftarrow 0, C \leftarrow Rd(7)$	Z,C,N,V,H
LSR	Rd	Logical Shift Right	$Rd(n) \leftarrow Rd(n+1)$ , $Rd(7) \leftarrow 0, C \leftarrow Rd(0)$	Z,C,N,V
ROL	Rd	Rotate Left Through Carry	$Rd(0) \leftarrow C$ , $Rd(n+1) \leftarrow Rd(n), C \leftarrow Rd(7)$	Z,C,N,V,H
ROR	Rd	Rotate Right Through Carry	$Rd(7) \leftarrow C$ , $Rd(n) \leftarrow Rd(n+1), C \leftarrow Rd(0)$	Z,C,N,V
ASR	Rd	Arithmetic Shift Right	$Rd(n) \leftarrow Rd(n+1)$ , $n=0..6$	Z,C,N,V
SWAP	Rd	Swap Nibbles	$Rd(3..0) \ll Rd(7..4)$	None
BSET	s	Flag Set	SREG(s) $\leftarrow 1$	SREG(s)
BCLR	s	Flag Clear	SREG(s) $\leftarrow 0$	SREG(s)
SBI	P, b	Set Bit in I/O Register	I/O(P, b) $\leftarrow 1$	None
CBI	P, b	Clear Bit in I/O Register	I/O(P, b) $\leftarrow 0$	None
BST	Rr, b	Bit Store from Register to T	T $\leftarrow Rr(b)$	T
BLD	Rd, b	Bit load from T to Register	Rd(b) $\leftarrow T$	None
SEC		Set Carry	C $\leftarrow 1$	C
CLC		Clear Carry	C $\leftarrow 0$	C
SEN		Set Negative Flag	N $\leftarrow 1$	N
CLN		Clear Negative Flag	N $\leftarrow 0$	N
SEZ		Set Zero Flag	Z $\leftarrow 1$	Z
CLZ		Clear Zero Flag	Z $\leftarrow 0$	Z
SEI		Global Interrupt Enable	I $\leftarrow 1$	I
CLI		Global Interrupt Disable	I $\leftarrow 0$	I
SES		Set Signed Test Flag	S $\leftarrow 1$	S
CLS		Clear Signed Test Flag	S $\leftarrow 0$	S
SEV		Set Two's Complement	V $\leftarrow 1$	V
CLV		Clear Two's Complement	V $\leftarrow 0$	V
SET		Set T in SREG	T $\leftarrow 1$	T
CLT		Clear T in SREG	T $\leftarrow 0$	T
SEH		Set Half Carry Flag in SREG	H $\leftarrow 1$	H
CLH		Clear Half Carry Flag in SREG	H $\leftarrow 0$	H

**Figur 6.6:** Bit- och bittestinstruktoner

En delmängd av ovanstående instruktioner är användbara för att hantera enskilda strömställare, tangentbord, lysdioder, etc. som är anslutna till IO-register för digitala in- och utsignaler.

### ■ Exempel 6.2: Bitmanipulering

Som ett första litet exempel börjar vi med att nollställa bitarna PD1 och PD6 i PORTD, övriga bitar lämnas oförändrade:

```
CBI      PORTD, 6
CBI      PORTD, 1
```

En strömställare är ansluten till PD7, vi skall utföra olika bearbetningar beroende på om den är 1 eller 0.

```
IN       R0, 0x??    ;R0=PORTD
ANDI    R0, 0x80
BREQ    PD7_0

PD7_1:  ; ;---PD7=1---
...
RJMP    PH7_END

PD7_0:  ; ;---PD7=0---
...
RJMP    PH7_END

PD7_END:
```

**Slut exempel 6.2 ■**

## 6.3 Lysdioder

I en diod i vilken det flyter en ström frigörs energi vid rekombination av hål och elektroner i PN-övergången. I en vanlig kiseldiod frigörs denna energi i form av värme. I dioder tillverkade av speciella ämne såsom galliumarsenid, galliumfosfid eller galliumarsenidfosfid frigörs energin istället i form av ljus (fotoner). Dessa ljusemitterande (ljusavgivande) dioder kallas *lysdioder*. Ofta så används den engelska förkortningen *LED - Light Emitting Diode*.



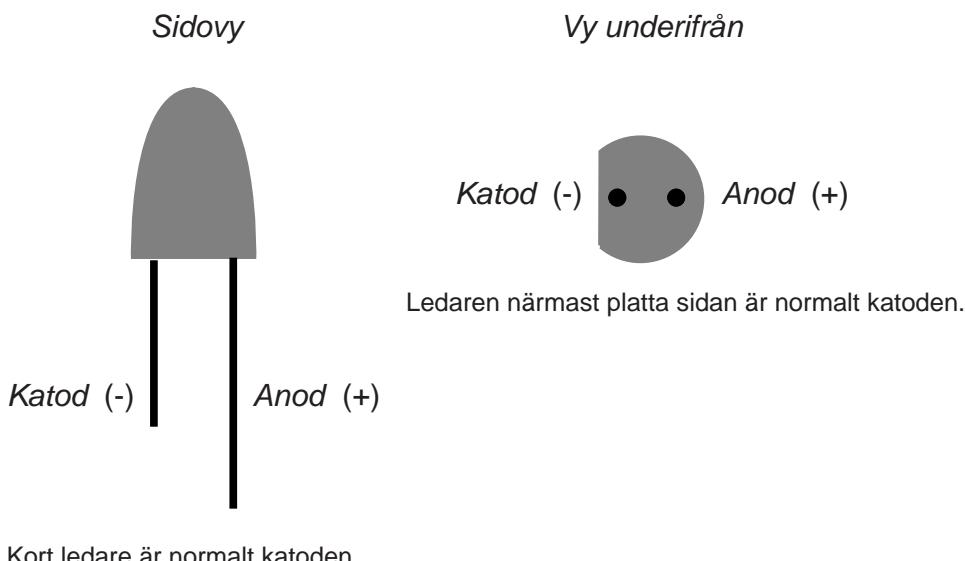
**Figur 6.7:** Kretssymbol för en lysdiod

Följande gäller för några typer av lysdioder:

- Galliumarsenid, **GaAs**, avger infrarött till rött ljus, framspänningsfall typiskt 1.4 V.
- Galliumarsenidfosfid, **GaAsP**, avger rött till gult ljus, framspänningsfall typiskt 2 V.
- Galliumfosfid, **GaP**, avger grönt till blågrönt ljus, framspänningsfall typiskt 3 V.
- Galliumnitrid, **GaN**, avger blått ljus.

Kan färs att lysa med vitt sken genom att lägga ett fosforskikt på diodens plasthölje. Detta gör att en del av det blåa ljuset omvandlas till gult ljus, som sedan genom kombination med det resterande blåa blir ett starkt vitt ljus.

Ljusstyrkan hos en lysdiod ökar med ökande ström.



Kort ledare är normalt katoden.

**Figur 6.8:** Katod och anod på en lysdiod

## 6.4 Strömställare

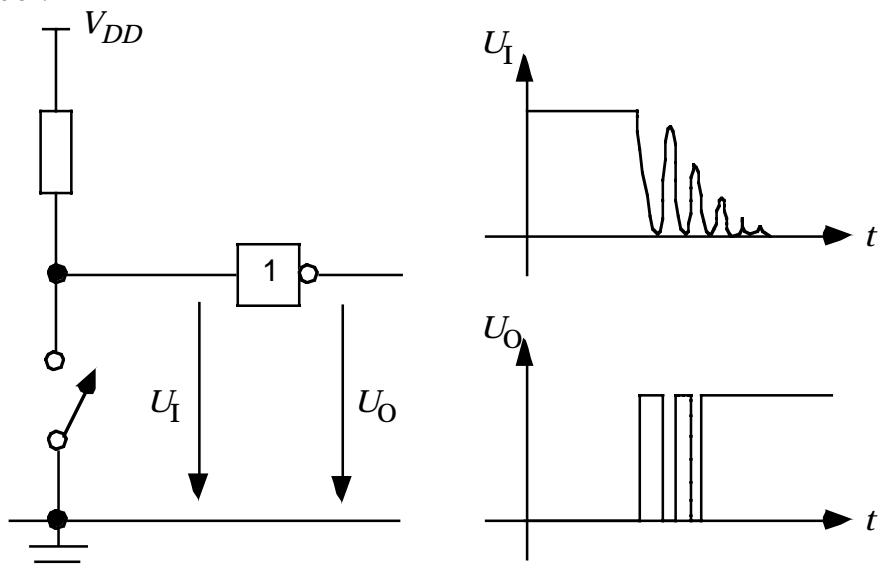
*Strömställare* är ett samlingsnamn för alla komponenter som sluter, bryter eller kopplar om ström. En strömställare är oftast i våra sammanhang en elektromekanisk komponent. Vi kommer i princip bara att hantera små spänningar och strömmar och kommer därmed ej att stöta på de problem som uppstår vid brytning av stora spänningar och strömmar.

Olika typer av strömställare finns:

- *Vippströmställare*. En vippströmställare har 2 eller flera distinkta lägen för vilket det erfodras en viss mekanisk kraft för att ändra läge.
- *Skjutströmställare*. Förekommer ofta i DIL-kapsel för kretskortsmontage som miniatyrströmställare.
- *Tryckströmställare*.
- *Tangentbordsströmställare*.
- *Tungelement*. Magnetkänslig strömställare. I en glaskolv finns en metalltunga mellan två elektroder, då denna påverkas av ett magnetfält slutes strömkretsen.
- *Reläer*. Vanligtvis mekaniska kontakter som styrs genom magnetisk kraftpåverkan från en spole.
- *Gränslägesbrytare*.
- etc.

### 6.4.1 Kontaktavstudsning i elektromekaniska strömställare

*Kontaktstudsar* är något som de flesta elektromekaniska strömställare uppvisar. Under en kort tid då strömställaren växlar läge kommer signalen från denna att vara obestämd på grund av mekaniska studsar. Den tid det tar för de mekaniska studarna att klinga av beror på strömställarens konstruktion, normalt så är tiden mindre än 25 millisekunder.

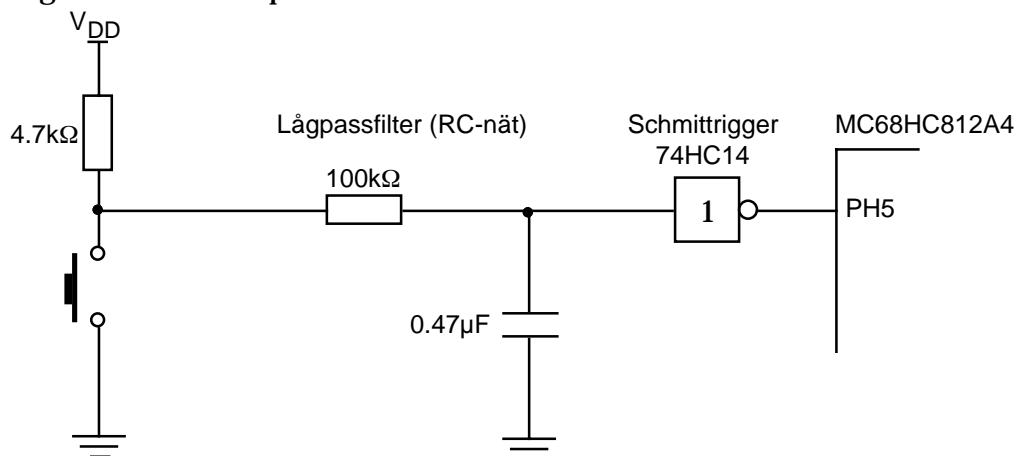


Figur 6.9: Effekterna av kontaktstudsar

Olika sätt att lösa kontaktstudsproblemet finns både i form av hårdvarumässiga och mjukvarumässiga. Oftast är det dock bekvämast att använda de mjukvarumässiga metoderna. För ordningens skull redovisar vi också de hårdvarumässiga lösningarna, då dessa samtidigt ger oss en viss insikt i problematiken kring mekaniska studar.

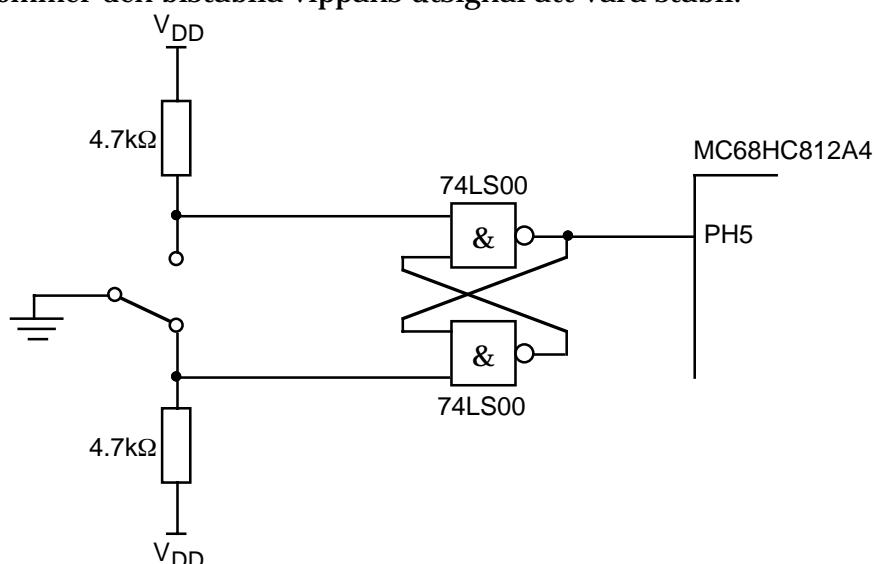
#### 6.4.1.1 Hårdvarumässig lösning på kontaktstudsproblemet

En lågpassfiltrering av signalen från strömställaren tar bort de högfrekventa komponenterna. Signalen över kondensatorn kommer att ha reducerade amplitudtoppar samt en större stigtid jämfört med den ofiltrerade signalen. Schmittriggerns hysteres gör sedan att de små variationer som utsignalen har från lågpassfiltret (som är insignalen till Schmittriggern) ej förmår att ändra Schmittriggern när omslagsnivån väl har passerats.



**Figur 6.10:** Avstudsning med lågpassfilter och en Schmitttrigger

Om en tvälägesomkopplare används så kan en alternativ lösning med en bistabil vippa användas (SR-vippa). Observera att vid omkoppling sker studsningen mot en och samma pol, detta gör att NAND-grinden som är ansluten till polen vid första kontaktstudsens kommer att slå om och ge utsignalen 1 som därmed tvingar den andra NAND-grinden att ge utsignalen 0. Oavsett hur många kontaktstudsar som sker efter den första kommer den bistabila vippans utsignal att vara stabil.



**Figur 6.11:** Avstudsning med SR-vippa (bistabil vippa)

#### 6.4.1.2 Mjukvarumässig lösning på kontaktstudsproblemet

En mjukvarumässig lösning av kontaktstudsproblemet är normalt önskvärd då detta kostar mindre. Detta kan göras på olika sätt beroende på applikationens natur:

- *Vänteloopslösning*. Vänta 20-50 millisekunder från det första kontaktögonblicket innan programmet får fortsätta. Detta bör göra att alla kontaktstudsar har tagit slut och därmed är dolda för programmet.
- *Räkneloopslösning*. Läs av strömställaren ett upprepat antal gånger. Antag att alla studsarna är slut då samma tillstånd har uppmäts t.ex. 50 ggr. Denna lösning ger ett snabbare programförlopp än lösningen med vänteloop.
- *Fördröjt tillslag*. I häftet "Signaler och system" och i kapitlet om Logikprogrammering och sekvenstyrning finns det beskrivit hur tillslagsfördräjning kan göras i ett program som exekveras cykliskt med periodtiden  $T$  sekunder. Principen baserar sig på att räknare används för att hålla reda på tiden.

### ■ Exempel 6.3: Avläsning av en starttangent (vänteloopslösning)

I detta lilla exempel utgår vi ifrån att det på port PB5 finns en tryckknapp ansluten och att denna är aktiv låg. Vi skall alltså vänta in tills denna blir 0, därefter skall olika programfunktioner utföras.

För att vänta ut kontaktstudsarna har vi en programmässig vänteloop i form av subrutinen `wait(milliseconds)`. Till denna skickar vi med en parameter i registerparet R25:R24 (datatypen int) som anger hur många millisekunders väntefördräjning vi skall ha. Observera att fördräjningsrutinerna är gjorda med förutsättningen att vi har en 16MHz klocka till processorn. Om vi klockar med annan frekvens får dessa rutiner modifieras.

Efter 50 ms räknar vi med att alla kontaktstudsar är borta. Detta är nödvändigt då exekveringen av subrutinen `doTheDirtyWork` ej tar mer än högst 1 ms.

```
.global bouncing

#define PORTB 0x18
#define DDRB 0x17
#define PINB 0x16

bouncing:

wait_start:
    SBIC    PINB,5           ;Skip next instruction if Bit 5
                           ;in PINB is Cleared
    RJMP    wait_start

    LDI     R24,50            ;Wait 50 milli seconds
    LDI     R25, 0
    RCALL   wait_ms

    RCALL   doTheDirtyWork

wait_release:
    SBIS    PINB,5           ;Skip next instruction if Bit 5
                           ;in PINB is Set
    RJMP    wait_release

    LDI     R24,50            ;Wait 50 milli seconds
```

```

LDI      R25, 0
CALL    wait_milliseconds

RET

doTheDirtyWork:
    ; something to do

RET

; ; ; ===Subroutine wait_milliseconds=====
; ; ;   C-prototype: void wait_milliseconds(int milliseconds);
; ; ;                           Parameter 1 is passed in registerpair
R24:R25
    .global wait_milliseconds

wait_milliseconds:
wait_milliseconds_loop:
    RCALL    wait1ms
    SBIW    R24,1           ; Subtract R25:R24=R25:R24-1
    BRNE    wait_milliseconds_loop
wait_milliseconds_end:

RET

; ; ; ===Subroutine wait1ms=====
; ; ;   C-prototype: void wait1ms(void);
    .global wait1ms

wait1ms:
    CALL    wait250microseconds
    CALL    wait250microseconds
    CALL    wait250microseconds
    CALL    wait250microseconds
    RET

```

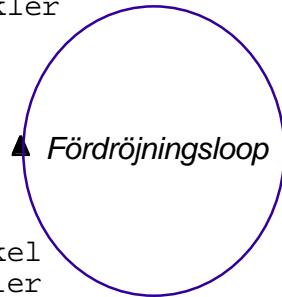
```

; ; ==Subroutine wait250microseconds=====
wait250microseconds:
    ; Register R25-R18 can be used as scratch pad register
    ; due to the GNU C calling conventions.

    ; 16MHz clock <-> 1 machine cycle = 62.5 ns
    LDI    R18, 210
wait_loop:
    LD     R19,X      ;2 maskincykler
    LD     R19,X
    DEC   R18       ;1 maskencykel
    BRNE wait_loop ;2maskencykler
                    ;vid hopp
                    ;annars 1 maskencykel

RET

```



I fördräjningsrutinen `wait250microseconds` har vi en fördräjningsloop, som räknar variabel i denna används register **R18**. Antalet maskencykler för ett varv i loopen är

$$8 \cdot 2 + 1 + 2 = 19 \text{ maskencykler}$$

Om en maskencykel är på 62.5 nanosekunder (16MHz klockning) ger detta en p

$$19 \cdot 62.5 = 1187.5 \text{ ns}$$

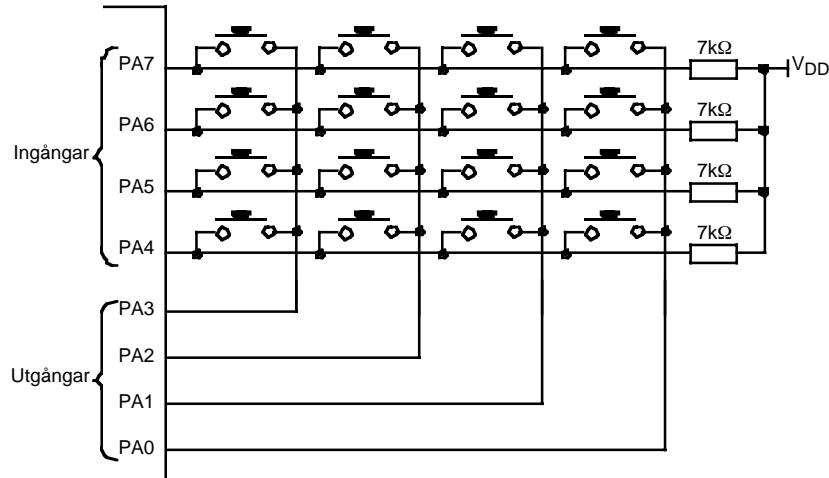
Totalt så genomlöper vi denna loop 210 gånger och tiden blir därmed

$$210 \cdot 1187.5 = 249375 \text{ nanosekunder} = 0.25 \text{ millisekunder}$$

**Slut exempel 6.3** ■

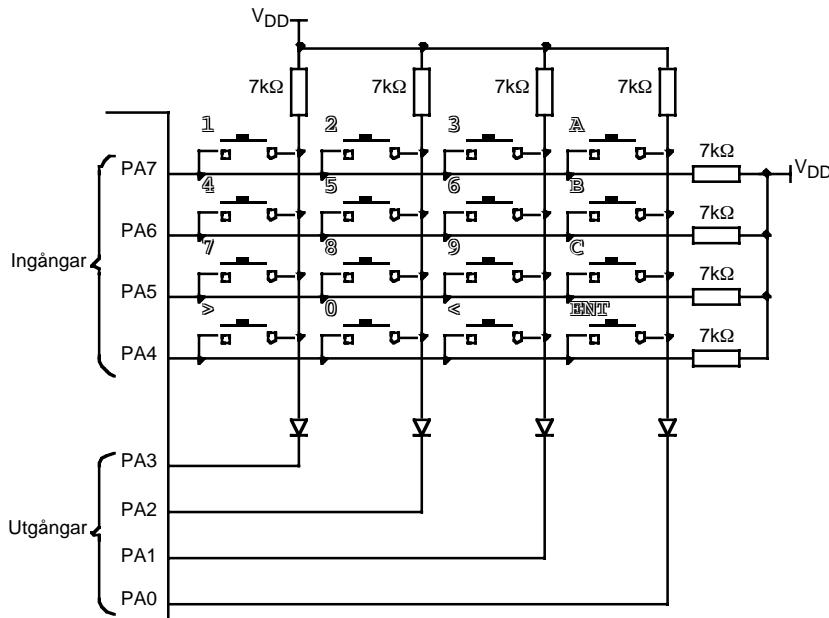
#### 6.4.2 Många tryckknappar <-> tangentbord

Om antalet tryckknappar i en tillämpning är mindre än 8 finns det sällan anledning till att organisera dem så att antalet portar som behöver utnyttjas ej blir för stort. Från programmerarens synvinkel så kan alltid en grupp av tryckknappar betraktas som ett tangentbord även om de är oberoende av varann. I figuren som följer visas hur 16 tryckknappar kan organiseras så att endast 8 pinnar på processorn för digitala in-/utgångar behöver användas. Observera att tryckknapparna ej fysiskt måste vara monterade i en matris såsom figuren visar, utan detta är en logisk vy sett från programmerarens synvinkel. Antalet portar som åtgår är lika med summan av antalet rader och kolumner i matrisen. Jämför detta med fallet att en port används per tangent.

**Figur 6.12:** 4x4 matris av tryckknappar

Utgångarna är normalt höga. Vid läsning av en kolumn av tryckknappar sätts motsvarande utgång låg. Sedan läses alla tryckknappar av i denna kolumn, om en tryckknapp är nedtryckt skall motsvarande radbit vara 0, annars är den 1.

Ovanstående koppling kan ge upphov till problem om fler än en tangent per rad är nedtryckt i form av att kolumnutgångar med olika tillstånd kommer att kortslutas via tangenterna. En lösning på detta problem är att isolera kolumnerna med hjälp av dioder.

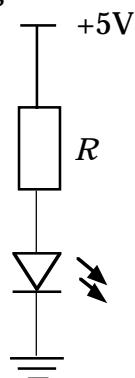
**Figur 6.13:** Tangentbord med kolumnisolering via dioder

Om fler tryckknappar än två är nedtryckta samtidigt kan ovanstående lösning ge felaktigt avlästa värden. Antag att tangenterna i positionerna (3,1), (4,1) och (4,2) är nedtryckta samtidigt, vilka tangenter kommer du då att detektera som aktiva då du scannar tryckknappsmatrisen? En mjukvarumässig lösning på detta problem kan åstadkommas genom att endast acceptera att en tryckknapp i taget per kolumn får vara nedtryckt.

## 6.5 Uppgifter

### U1: Digitala in- och utportar

- a) Då reset görs på processorn initieras alla register som kan fungera antingen som digital ingång eller digital utgång till att fungera som digitala ingångar. Varför? Tänk i termer av elektriska egenskaper.
- b) Visa i assemblerkod hur datariktningsregistret för PORTA skall initieras om PA7-PA4 skall vara ingångar och PA3-PA0 skall vara utgångar.
- c) Antag att en strömställare är ansluten till PA5 och att du vill känna av om den har nedtryckts. Strömställaren ger noll vid nedtryckt tillstånd. Visa också med ett kopplingsschema hur strömställaren är ansluten.
- d) En lysdiod behöver en ström på ungefär 10 mA för att lysa "bra", d.v.s. för att få önskad ljusstyrka. Vilket värde skall vi välja på förkopplingsmotståndet? Räkna med ett framspänningsfall på ungefär 2V.



- A)  $300\Omega$     B)  $1k\Omega$     C)  $2k\Omega$     D)  $600\Omega$

- e) Antag att PA7-PA4 är programmerade som utgångar och att PA3-PA0 är ingångar. Följande läge gäller för registren:

PORATA	1	0	0	1	0	1	1	0	1B (003B)
DDRA	1	1	1	1	0	0	0	0	1A (003A)
PINA	1	0	0	1	0	1	1	0	1C (0039)

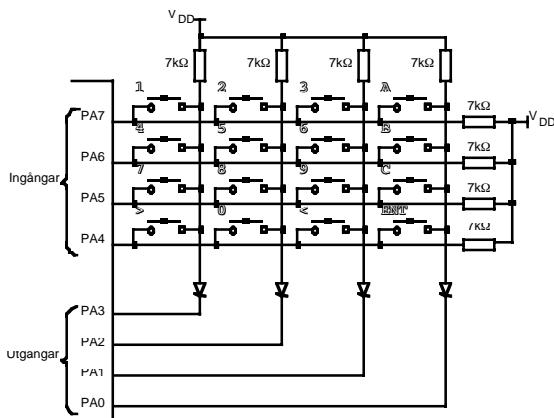
Vilket värde har PORATA efter det att följande programkod har genomlöpts?

```

SBIS      PINA, 2           ; Skip next instruction if Bit 2
                           ; in PINA is Set
RJMP      L1
SBI       PORTA, 6          ; Set Bit in I/O Register
RJMP      L2
L1:        ADD      R20, R21
L2:
    
```

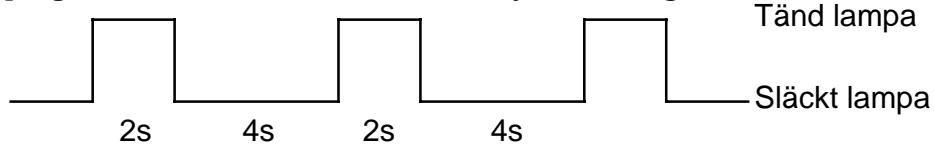
- A) E6    B) 16    C) C6    D) D6

- f) Antag att tangenten "5" är nedtryckt. Vid avläsning av denna, vilket värde skall utgångarna ha och vilket värde får ingångarna?



## U2: Vänteloop

- a) Vad är nackdelen med att använda en vänteloop i ett program för att åstadkomma en tidsfördröjning?
- b) Skriv ett program som tänder och släcker en lysdiod enligt nedanstående sekvens



Använd en programvarumässig tidsfördröjning.

## 6.6 Svar

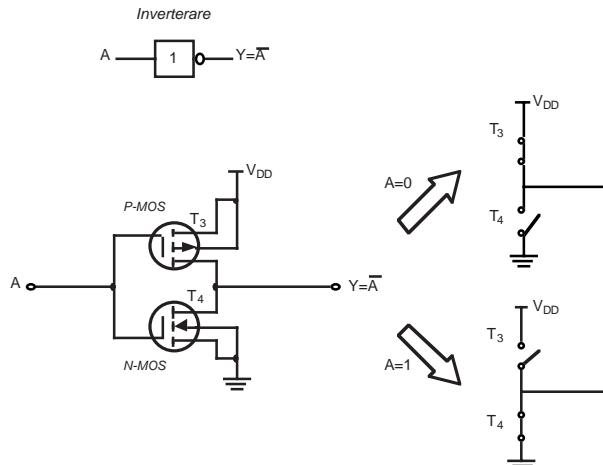
### U1: Digitala in- och utportar

- a) Problem uppstår alltid då två utgångar kopplas mot varann. En utgång ger antingen 0 eller 1. Detta innebär att utgången antingen står direkt eller indirekt i kontakt med 0V (jord) eller med 5V (matningsspänning). Om nu två utgångar kopplas ihop så innebär det i det olyckligste fallet att vi får en kortslutning mellan 0V och 5V. Detta gör att en stor ström kommer att gå igenom utgångstransistorerna och med en därtill relativt sett stor effektutveckling i form av värme i dessa. Vilket i slutändan kan förstöra utgångstransistorerna. I strömbrytarmodellen har vi förenklat transistorn. I praktiken så har vi en spänning över drain och source, d.v.s. transistorn har en resistans större än 0. Effektutvecklingen i transistorn i form av värme blir

$$P = U_{DS} \cdot I_{DS}$$

Av säkerhetsskäl görs därför alla DIO-portar till digitala ingångar. En ingång kan ej driva en ström och därmed får vi inga problem oavsett vad som är anslutet till porten. Man får sedan hoppas att programmeraren vet vad som är anslutet till portarna så att datariktningsregistret programmeras rätt.

Ha gärna en CMOS-inverterare som en bild att hänga upp ovanstående resonemang på. Problem uppstår då två sådana utgångar kopplas mot varann:



b)

```
#define PINA      0x19
#define DDRA      0x1A
#define PORTA     0x1B

; ; DDRA=0x0F;
LDI      R18,0x0F
OUT      DDRA , R18
```

c)

```
SBIC    PINA,5           ; Skip next instruction if Bit 5
;in PINB is Cleared
RJMP    L_PA5_1
L_PA5_0:
; ; Tryckknappen aktiv låg
; ; Här utför vi något nyttigt
...
RJMP    L_PA5_END
L_PA5_1:
L_PA5_END:
```

d) A) 300Ω

e) D) D6

f) Ingångar är PA7-PA4, utgångar är PA3-PA0

För att avläsa tangent '5' skall utgångarna vara 1011, på ingångarna kommer vi att avläsa 1011.

## U2: Vänteloop

- a) Nackdelen är att CPU:n inte används för något "nyttigt" arbete. Metoden med väntelooopar kan vara svår att använda om beräkningsintensiva saker skall utföras av processorn. I vårt fall så fungerar vänteloopen ypperligt, då vi ej har något annat som skall utföras.

b)

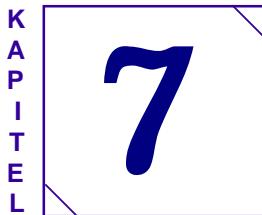
**Pseudokod i C**

```
void pattern(void)
{
    DDRA = 0x01; //PA0 programmed as output
    while (1)
    {
        PORTA = 0x01;
        wait_milliSeconds(2000);
        PORTA = 0x00;
        wait_milliSeconds(4000);
    }
}
```

**Assemblerkod**

```
#define PINA      0x19
#define DDRA      0x1A
#define PORTA     0x1B

.text
.global pattern
;;; --- void pattern(void)
;;;
pattern:
;;; --- DDRA = 0x01; //PA0 programmed as output
    LDI      R25, 0x01
    OUT     DDRA, R25
;;; --- while (1)
;;; ---
mainWhileStart:
;;; ---     PORTA = 0x01;
    LDI      R25, 0x01
    OUT     PORTA, R25
;;; ---     wait_milliSeconds(2000);
    LDI      R24,160
    LDI      R25,15
    RCALL   wait_milliSeconds
;;; ---     PORTA = 0x00;
    LDI      R25, 0x00
    OUT     PORTA, R25
;;; ---     wait_milliSeconds(4000);
    LDI      R24,208
    LDI      R25,7
    RCALL   wait_milliSeconds
;;; --- }
    RET
```



# GNU assembler/länkare

För en fullständig redogörelse för GNU assemblern med namnet "GNU as" hänvisar vi till manualen för denna. I detta kapitel tar vi upp det mest primära om denna assembler. Förutom assemblern skall vi också behandla länkaren som går under namnet "GNU ld".

## 7.1 Maskinberoende syntax

Assemblern "GNU as" är en generell assembler som fungerar med en mängd olika processorer. Med maskinberoende syntax menas de delar som är beroende av vald processor, maskinberoende syntax består huvudsakligen av instruktionsuppsättningen för vald processor.

Som *whitespace-tecken* räknas blanktecknet eller tabulatortecknet, i godtycklig ordning. Whitespace-tecken används för att separera symboler och för att göra program mer läsbara. För assemblern gäller att den räknar en godtycklig sekvens av whitespace-tecken som ett blanktecken.

Kommentarer i ett assemblerprogram kan göras på olika sätt. Dels som det görs i C:

```
/*
   Detta är en kommentar.
*/
```

Det andra sättet, som är vanligt i olika assemblerspråk är att använda semikolon för att starta en *radkommentar*. En radkommentar gäller fram till slutet av raden:

```
LDI    R2,20      ;Här börjar radkommentaren
LDS    R1,0x0200
ADDA   R2,R1      ;Nästa radkommentar
STS    0x0201,R2
```

Symboler är ett grundläggande koncept i ett programmeringsspråk. Programmeraren använder symboler för att namnge saker, länkaren använder symboler för att länka ihop programmoduler till en exekverbar enhet och debuggerprogramvaran använder symboler vid avlusning av programmet. En *symbol* i ett assemblerprogram byggs upp med hjälp av följande tecken:

```
a,b,c,...,x,y,z,A,B,C,...,X,Y,Z
0,1,2,3,4,5,6,7,8,9
-
.
$
```

Inget symbolnamn får inledas med en siffra. Det finns ingen restriktion på längden av

en symbol, utan ett godtyckligt antal tecken kan användas.

En *programsats – statement* i assemblerspråket avslutas med ett nytradtecken ('\n'). Om en programsats behöver mer utrymme än en rad, kan denna utsträckas till att gälla flera rader genom att lägga ett backslashstecken (\) omedelbart före nyradtecknet.

En programsats börjar med ingen eller flera *adresslappar – labels*, valfritt följt av en nyckelsymbol som bestämmer vilken typ av programsats det är frågan om. Om symbolen inleds med en punkt är det frågan om ett assemblerdirektiv, som normalt är maskinberoende, d.v.s. gäller för alla typer av datorer/processorer. Om symbolen börjar med en bokstav är det frågan om en instruktion i assemblerspråket för vald processor. En instruktion kommer att assembleras till en maskinspråksinstruktion. Syntaxen för en programssats i assemblerspråket är enligt en av följande tre typer:

<u>LÄGE</u>	<u>OPERATION</u>	<u>OPERAND</u>	<u>KOMMENTAR</u>
label:	.directive	följt av något annat	
another_label:	instruction	operand_1, operand_2, ...	; Tom programsats

Ett assemblerprogram brukar skrivas i fyra kolumner: lägeskolumn, operationskolumn, operandkolumn och en kommentarkolumn. Om en label finns på en rad så skall den alltid inleda raden.

Observera att en label eller med det svenska ordet adresslapp eller adressetikett är en symbol som omedelbart åtföljs av ett kolon (:). Som ordet adresslapp antyder används labels för att referera till olika minnesadresser. Dessa är normalt av någon av följande typ: adresser till variabler och konstanter, hopplägen för villkorliga eller repetitiva programsatser eller start på funktioner. En label är en symbolisk adress. Motsvarigheten till labels i högnivåspråket C är identifierare. En identifierare i C är ett variabelnamn, konstantnamn eller ett funktionsnamn.

En *konstant* är ett tal skrivit så att dess värde alltid är känt. Några exempel på konstanter är:

```
.byte    74,0112,092,0X4A,0X4a,'J' ;J på olika sätt
       .ascii  "Ring i klockan \7"      ;Strängkonstant
PI:     .float   3.1415922653589      ;Flyttalskonstant
MAX:    .int     32756                 ;Heltalskonstanter
       .int     1,2,3,4
```

En *teckenkonstant* är ett tecken som upptar en byte i minnet, ett tecken är ett litet heltalet och kan användas i olika beräkningsuttryck. En *strängkonstant* innesluts mellan två citattecken, vilket skrivas på samma sätt som i C. De 32 första tecknen i ASCII-tabellen används som styrtecken och är osynliga, på tangentbordet finns ingen tangent för dessa. För att kunna använda dessa i en teckensträng måste en konvention tillgripas som enbart använder synliga tecken (tangentbordstecken), för detta ändamål används så kallade escape-tecken. Som escapestecken används backslashstecknet '\', vilket leder till att tecknet därefter skall omtolkas till ett av de osynliga tecknen i ASCII-tabellen.

**Figur 7.1:** Tillåtna escapesekvenser i GNU as

Heltalskonstanter kan skrivas på olika sätt, antingen binärt, oktalt, hexadecimalt eller decimalt.

Ett binärt heltal inleds med 0b eller 0B följd av noll eller flera binära siffror (0 eller 1).

Ett oktalt heltal inleds med 0 följt av noll eller flera oktala siffror (0,1,2,3,4,5,6,7).

Ett decimalt heltal startar med en siffra som ej är 0 därefter följt av noll eller flera decimala siffror (0,1,2,3,4,5,6,7,8,9).

Ett hexadecimalt heltal inleds med 0x eller 0X följt av noll eller flera hexadecimala siffror (0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f,A,B,C,D,E).

Flyttalskonstanter skriver vi på samma sätt som i C.

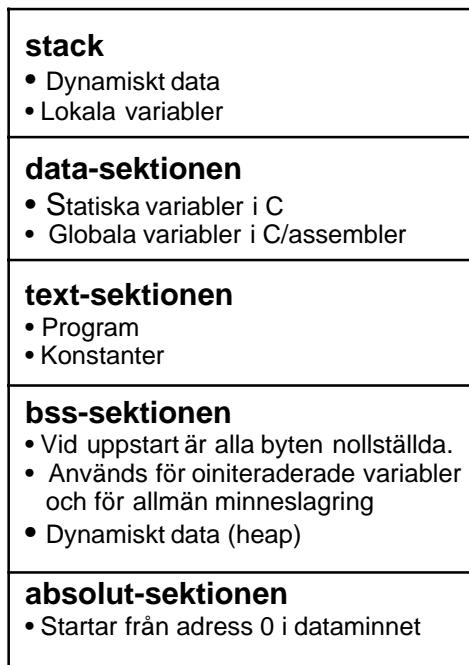
## 7.2 Sektioner och relokering av kod/data

En *sektion – section* är enkelt uttryckt ett kontinuerligt adressområde, där inga hål finns från starten till slutet. Alla minnesadresser i en sektion har samma egenskaper, t.ex. skulle en sektion kunna ha enbart läsegenskapen, i en sådan sektion kan program och konstanta data lagras. Sektionerna återspeglar i grunden vilken typ av minnen som finns i datorsystemet: skriv-/läs-minne, enbart läsminne, etc.

Länkaren "GNU ld" läser ett antal objektkodsfiler, d.v.s. partiella program, och kombinerar dessa till ett exekverbart program. Assemblern "GNU as" skapar objektkodsfiler utifrån källkodsfiler. Alla dessa objektkodsfiler antas starta från adress 0 (titta gärna på en listfil från assemblern). Länkarens uppgift är att tilldela de slutgiltiga adresserna för de partiella program som länkas ihop så att de ej överlappar varann.

GNU ld hanterar enbart fyra olika typer av sektioner:

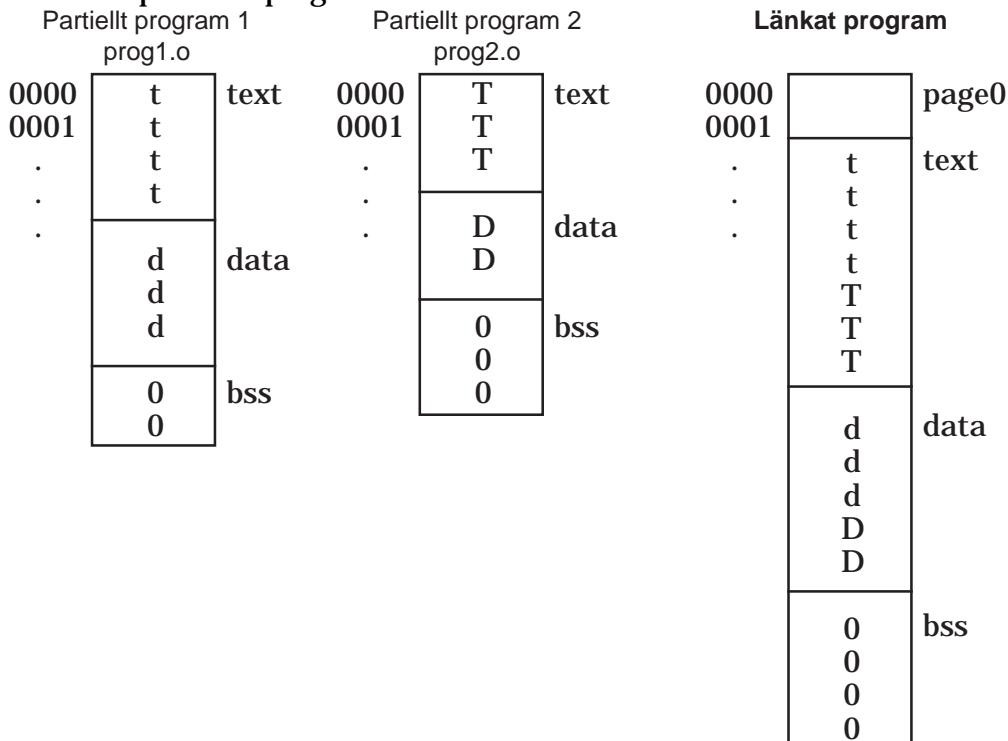
- **text-sektionen**  
Textsektionen är tänkt att vara oföränderlig och i denna placeras program i form av maskinspråksinstruktioner, konstanter, etc. I AVR-processorn finns text-sektionen i FLASH-minnet.
- **data-sektionen**  
Databsektionen är tänkt att vara förändringsbar i denna placeras globala variabler, etc. I AVR-processorn finns data-sektionen i SRAM-minnet.
- **bss-sektionen**  
Denna sektion innehåller ett minnesutrymme som nollställs vid reset. Den används för oinitialiserade variabler och som ett allmänt datautrymme. I AVR-processorn finns bss-sektionen i SRAM-minnet.
- **absolut-sektionen**  
Adress 0 i denna sektion "relokeras" alltid till runtime-adressen 0.



*Figur 7.2: Sektioner i ett GNU C/Assemblerprogram*

För de program som vi skriver i assembler kommer enbart två sektioner att vara av intresse: dasektionen och textsektionen. Dasektionen i AVR-processorn består fysiskt av SRAM-minnet och textsektionen av FLASH-minnet (programminnet).

Figuren som följer visar på hur länkningsprogrammet *relocates* olika sektioner i de partiella programmen (objektkodsfiler). Relokering betyder på ren svenska att länkaren åter (re) bestämmer en ny plats i minnet (locate) för de olika sektionerna i de partiella programmen.



Figur 7.3: Länkningsprocessen

## 7.3 Assemblerdirektiv

Ett assemblerdirektiv är ett kommando till assemblatorn. Direktivet översätts inte till maskinkod, d.v.s. genererar ingen instruktionskod till processorn. Alla assemblerdirektiv inleds med en punkt ('.'), resten av namnet är bokstäver.

Observera att syntaxen för de olika direktiven anges i rubrikerna som följer.

### 7.3.1 .org new\_location , fill

Används för att sätta *lokaliseringräknaren* – *location counter*. Denna räknare styr var i minnesmappen som maskinkod, data och variabler placeras i minnesarean.

Följande deklaration

```
.org      0x0100
```

specificerar att program/data som därefter matas in kommer att starta på adress 100 hexadecimalt.

Lokaliseringräknaren anges alltid relativt början av en sektion (text, data, bss).

När lokaliseringssökningen avancerar d.v.s. ändrar värde kommer mellanliggande bytes att fyllas med **fill**, om detta utelämnas sker fyllningen med 0x00.

### 7.3.2 .equ symbol, uttryck

Används för att tilldela en symbol ett värde. Ofta används den för att namnge adresser, så kallade *symboliska adresser*. Mer allmänt är det frågan om att skapa symboliska namn för heltalsvärdet, detta för att öka läsbarheten och förståelsen för vad som görs i assemblerprogrammet.

Deklaration av en symbolisk adress kan göras på följande sätt:

```
.equ ADR1, 0x0100
```

Detta innebär att den absoluta adressen 100 hexadecimalt associeras till symbolen ADR1. Därefter kan det symboliska namnet ADR1 användas på alla ställen i programmet där vi vill göra någon skriv- eller läsoperation på adress  $100_{16}$ . Det symboliska namnet ADR1 är ekvivalent med det numeriska värdet 0x100. Jämför gärna med makrodefinitioner i C, där det ekvivalenta skrivasättet skulle vara:

```
#define ADR1 0x100
```

### 7.3.3 .comm symbol, längd

Används för att specificera datautrymme, d.v.s. att reservera plats i minnet för ett minnesblock med storleken **längd** och namnet (label) **symbol**. Detta minnesblock kommer ej att initieras.

Till exempel:

```
.comm CH, 1
```

kommer att reservera en byte för data. Adressen till denna minnescell nås genom den symboliska adressen CH.

På liknande sätt kommer:

```
.comm TABELL, 40
```

reservera 40 byte för data till en struktur med namnet TABELL (symbolisk adress).

Direktivets namn är en förkortning av det engelska ordet common, som har betydelsen gemensam och allmän. Detta leder till att symbolen samtidigt blir känd utanför källkodsfilen och finns med i symboltabellslistningen. Direktivet ersätter i princip en kombination av global- och space-direktivet:

```
.global TABELL
```

```
TABELL: .space 40
```

### 7.3.4 .byte *uttryck\_1*, ..., *uttryck\_n*

Används för att definiera en eller flera 8-bitars konstanter eller textsträngar. Följande är några exempel:

```
K1:      .byte  10           ;Konstant med värdet 10
MASK:    .byte  0b1101100      ;Bitmaskkonstant
TAB:     .byte  0x0A,0xF3,0x1F,0x11 ;Tabell med fyra värden
```

### 7.3.5 .word *uttryck\_1*, ..., *uttryck\_n*

Används för att definiera en eller flera 16-bitars heltalskonstanter eller initierade variabler. Används på samma sätt som .byte-direktivet. Följande är några exempel:

```
K5:      .word   10          ;Konstant med värdet 10
MASK2:   .word   0b1111000011110000 ;Bitmaskkonstant, 16 bitar
TAB2:    .word   0x1234, 0x2322, 0x0000, 0xFFFF
                    ;Tabell med fyra värden
```

### 7.3.6 .long *uttryck\_1*, ..., *uttryck\_n*

Används för att definiera en eller flera 32-bitars heltalskonstanter eller initierade variabler. Följande är några exempel:

```
K5:      .long   10          ;Konstant med värdet 10
MASK2:   .long   0b11110000111100001111000011110000
                    ;Bitmaskkonstant, 32 bitar
TAB2:    .word   0x00001234, 0x11112322, 0x33330000
                    ;Tabell med tre värden
```

### 7.3.7 .float *uttryck\_1*, ..., *uttryck\_n*

Används för att definiera en eller flera flyttalskonstanter eller initierade variabler.

```
PI       .float  3.14
```

### 7.3.8 .ascii "*sträng\_1*", ..., "*sträng\_n*"

Direktivet .ascii förväntar sig av noll eller teckensträngar i en kommaseparerad lista. Varje sträng assembleras sedan till att ligga tecken för tecken i en konsekutiv adressordning. Observera att strängen ej är null-terminerad. Några exempel på strängar är:

```
str:     .ascii  "Detta är en sträng\0"
namn:    .ascii  "Putte Plutt"
```

### 7.3.9 .asciz "*sträng\_1*", ..., "*sträng\_n*"

Fungerar som .ascii-direktivet med den skillnaden att assemblern NULL-terminerar

varje sträng. Detta innebär att vi slipper att manuellt NULL-terminera strängen med escapesekvensen \0.

```
str:    .asciz  "Detta är en sträng"
```

### 7.3.10 .data

Anger för assemblatorn GNU att följande programsatser skall assembleras till att ligga i datasektionen.

### 7.3.11 .text

Anger för assemblatorn GNU att följande programsatser skall assembleras till att ligga i textsektionen.

### 7.3.12 .global symbol

Detta direktiv används för att göra en symbol global, vilket gör att den kan refereras i andra källkodsfiler. Direktivet är nödvändigt för att kunna skriva subrutiner i assembler som skall anropas från ett C-program.

```
.global Summara
Summara:
    ADD      R0 ,R1
    ADD      R0 ,R2
    ADD      R0 ,R3
    RET
```

## 7.4 Exempel: Lysdiodprogram

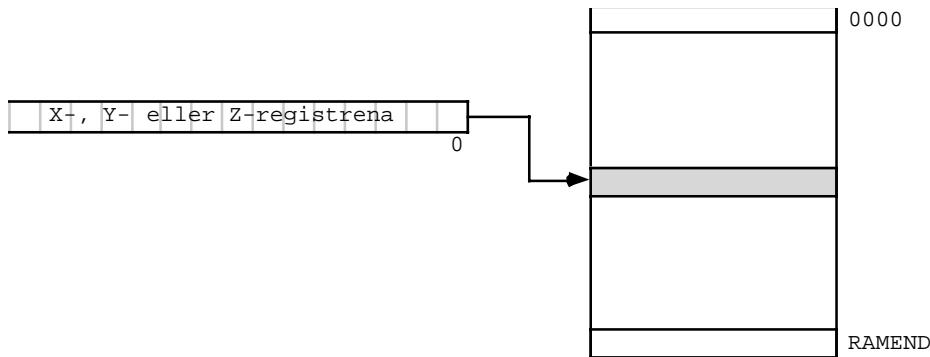
Anslutna till PORTB på processorn finns det 8 lysdioder som skall tändas och släckas efter följande mönster:

00000000
10000001
11000011
11100111
11111111
11100111
11000011
10000001

Lysdiодerna skall uppdateras med ett nytt värde efter 200 ms.

### 7.4.1 Indirekt dataadressering

Vid *indirekt dataadressering* används någon av registrena X (R27:R26), Y (R29:R28) eller Z (R31:R30) som en adresspekare in i dataminnet (SRAM-minnet). Denna adresseringsmod är användbar då man arbetar med pekare och vektorer.



**Figur 7.4:** Indirekt dataadressering

I exemplet som följer kommer vi att använda denna adresseringsmod för att indexera oss i en vektor med lysdiodsmönster.

#### 7.4.2 Pseudokodslösning i C

Ett C-program är en lösning på en högre abstraktionsnivå än ett assemblerprogram. Ett sätt att konstruera en lösning på en högre nivå vid assemblerprogrammering är att göra en PSEUDO-kodslösning antingen i ett högnivåspråk eller något slags pseudospråk.

I lösningen som följer har vi skapat en global vektor med namnet LEDS som innehåller de 8 lysdiodsmönstren. För att plocka ut rätt mönster ur tabellen har vi indexeringsvariabeln *i*, som är en 0 till 7 rundräknare.

```

void wait_milli_seconds(int value);

const unsigned char LEDS[8]={0x00, 0x81, 0xC3, 0xE7, 0xFF, 0xE7, 0xC3, 0x81};

int main(void)
{
    int i;

    i=0;
    DDRB=0xFF; //PB7-PB0 are programmed as outputs

    while (1)
    {
        PORTB = LEDS[i];
        i++;
        i = i & 0x07;

        wait_milli_seconds(500);
    }

    return 0;
}

```

Observera, att istället för att använda en if-sats för att sätta *i* till 0 då den blir större än 7

i++;

```
if ( i > 7 )
    i = 0;
```

så gör vi en enkel maskningsoperation

```
i++;
i = i & 0x07;
```

Detta går bra att göra då antalet intressanta indexeringsvärden är 8 (0 till 7) och representeras med 3 bitar, där vi utnyttjar alla 3 bitar. Om antalet indexeringsvärden hade varit t. ex. 10 hade detta ej gått, för att representera indexeringsvärdena åtgår 4 bitar där vi dock ej utnyttjar alla kombinationer. Ovanstående metod fungerar då antalet värden i tabellen är  $8 = 2^3$ ,  $16 = 2^4$ , etc.

### 7.4.3 Assemblerlösning

En lösning i assembler kan se ut på följande sätt:

```
.global leds_a1

/*====NAMNGIVNA NUMERISKA VÄRDEN=====*/
.equ      PINB, 0x16
.equ      DDRB, 0x17
.equ      PORTB, 0x18

/*====DATA=====*/
.data

.global LEDS_

LEDS_: .byte 0x00, 0x81, 0xC3, 0xE7
       .byte 0xFF, 0xE7, 0xC3, 0x81

/*==== leds_a1 =====*/
.text
leds_a1:

;; i=0;
LDI      R28, 0x00
LDI      R29, 0x00

;; DDRB=0xFF; //PB7-PB0 are programmed as outputs
LDI      R24, 0xFF
OUT      DDRB, R24

;; while ( 1 )
while_loop:
```

Z-registret är detsamma som registerparet R31:R30, Det första problemet består i att ladda detta register med adressen till vektorn. Adressetiketten LEDS\_ är på 16 bitar. Med hjälp av funktionerna lo8 och hi8 kan minst signifika respektive mest signifika byten i denna adress plockas ut.

```
; {
;;      PORTB = LEDS[ i ];
LDI      R30, lo8(LEDS_)
LDI      R31, hi8(LEDS_)
```

Variablen *i* är detsamma som registerparet R29:R28. Då varje element i tabellen är på 1 byte får adressen till indexerat element som LEDS\_+ *i* =R31:R30+R29:R28. Resultatet av additionen finns i registerparet R31:R30 som också är lika med Z-registret.

```
ADD      R30, R28
ADC      R31, R29
```

Läsning görs nu i dataminnet på den address som pekas ut av Z-registret.

```
LD       R24, Z
OUT     PORTB, R24

; ; i++;
ADIW    R28, 0x01

; ; i= i & 0x07;
ANDI    R28, 0x07      ; 7
ANDI    R29, 0x00      ; 0
```

Parameterpassningen till subrutinen wait\_milliseconds sker enligt konventionerna för GNU C-kompilatorn (behandlas i nästa kapitel). Detta innebär att om argument 1 är på 2 byte skall det skickas med i registerparet R25:R24.

```
; ; wait_milliseconds(500);
LDI     R24, 0xF4      ; 244
LDI     R25, 0x01      ; 1
RCALL   wait_milliseconds
mainEnd:
        RJMP   while_loop
```

Notera att vi har globaldeklarerat adressläget leds\_a1 för att vi skall kunna anropa assemblerprogrammet från main-funktionen i ett C-program.

#### 7.4.4 Anrop av assemblerprogrammet från main-funktionen

Anrop av en assemblersubrutin kan antingen ske från en C-funktion eller från en annan assemblersubrutin (med CALL/RCALL-instruktionerna). Då vårt huvudprogram utgörs av main-funktionen som är skriven i C, kommer vi att anropa vår assemblersubrutin som en funktion i C. Kompilatorn kräver att vi tillhandahåller en funktionsprototyp som beskriver hur ett giltigt funktionsanrop görs av vår subrutin. Assemblersubrutinen anropas i C-programmet som en funktion som ej tar några parametrar och ej returnerar något värde.

```
void leds_a1(void);

void main()
{
    while (1)
    {
        leds_a1();
    }
}
```

## 7.5 GNU Make - Generering av exekverbara filer

Programmet GNU Make utför kommandon som finns i en "makefil". Standardnamnet på denna fil är "Makefile". En sådan finns i varje projektatalog som vi skapar. I vårt sammanhang skall vi använda GNU Make för att på ett smidigt sätt generera exekverbara filer för vår AVR-processor, genom omkompileering av ändrade källkodsfiler och länkning av dessa.

En enkel makefil kan bestå av "regler" med följande syntax och utseende:

```
målfil : beroende_fil1, beroende_fil2, ... ,beroende_filn
        kommando1
        kommanod2
        ...
        ...
```

En *målfil* är normalt en exekverbar fil eller en objektkodsfil, d.v.s. filer av typen ".s19", ".elf", ".b", eller ".o". En *beroendefil* är en indatafil som behövs för att på något sätt skapa *målfilen*. Ett *kommando* är en åtgärd som skall utföras av GNU Make. Så fort en beroendefil har ändrats kommer målfilen skapas på nytt. Vilket även medför att de olika kommandona kommer att utföras för att skapa en ny målfil.

En variabel i makefilen definieras å följande sätt:

*VARIABEL* = <värdet>

vilken sedan kan anropas i make-filen enligt:

`$(VARIABEL)`

En annan typ av regel kan vara en åtgärd som skall utföras i form av ett antal kommandon som alltid skall utföras. Ett exempel på en sådan regel är 'clean', som i detta sammanhang används för att ta bort gamla filer (remove).

```
Definition av variabeln FINISH
FINISH = @echo Errors: none
BEGIN = @echo ----- begin -----
END = @echo ----- end -----

# Eye candy.
begin:
        $(BEGIN)
finished:
        $(FINISH)           Användning av variabeln FINISH
end:
        $(END)
```

clean: begin clean\_list finished end

clean\_list :

```
    $(REMOVE) $(TARGET).hex
    $(REMOVE) $(TARGET).eep
    $(REMOVE) $(TARGET).obj
    $(REMOVE) $(TARGET).cof
    $(REMOVE) $(TARGET).elf
```

```

$(REMOVE) $(TARGET).map
$(REMOVE) $(TARGET).obj
$(REMOVE) $(TARGET).a90
$(REMOVE) $(TARGET).sym
$(REMOVE) $(TARGET).lnk
$(REMOVE) $(TARGET).lss
$(REMOVE) $(OBJ)
$(REMOVE) $(LST)

```

Om du vill rensa bort gamla filer kan du ge make-kommandot med argumentet clean:

```
make clean
```

Sådana åtgärdsregler kan du själv enkelt skapa i din “Makefile” för olika behov som kan uppstå.

För en fullständig beskrivning av GNU Make hänvisas till manualen för denna. Den finns bland annat i HTML-format för läsning med webläsare.

### ■ **Exempel 7.1: Beroendeförhållanden i en makefil**

I en makefil finns följande rader:

```
lab1.o: lab1.s
        $(AS) lab1.s -o lab1.o
```

Följande tolkning kan göras:

- lab1.o: lab1.s  
Objektkodsfilen “lab1.o” är beroende av källkodsfilen “lab1.s”.
- Om “lab1.s” är nyare än “lab1.o” kommer kommandot:  
\$(AS) lab1.s -o lab1.o  
att utföras.

Observera att kommandoraden alltid måste TABBAS ut och att det ej duger med mellanslag.

**Slut exempel 7.1 ■**

### ■ **Exempel 7.2: Rekursiv kontroll av beroendeförhållanden**

I en makefil finns följande rader:

```

OBJS =vectors.o \
       main.o
project1: $(OBJS)
          $(CC) $(LDFLAGS) -o $@ $(OBJS) $(LIBS)
main.o: main.s
        $(AS) main.s -o main.o

```

*Implicit regel, behöver ej anges.  
En implicit regel finns inbyggd i make.  
Finns med för tydlighetens skull.*

Följande tolkning kan göras:

1. En variabel med namnet OBJS (för objektkodsfiler) tilldelas de objektkodsfiler som ingår i projektet
2. När make-programmet kommer till raden med project1 konstateras att denna beror av objektkodsfilerna, som hålls i variablen OBJS. Make-programmet kommer nu att rekursivt undersöka om det finns beroenden för “vectors.o” respektive “main.o” i make-filen.
3. Objektkodsfilen “main.o” är beroende av källkodsfilen “main.s”. Om “main.s” är av ett nyare datum än “main.o” kommer kommandot

```
$(AS) main.s -o main.o
```

att utföras.

4. När punkt 3 har utförts, kommer makeprogrammet att återgå till beroenderaden för project1 och utföra kommandot:

```
$(CC) $(LDFLAGS) -o $@ $(OBJS) $(LIBS)
```

## Slut exempel 7.2 ■

### 7.5.1 Makefile i våra projektmappar

Programmet make (make.exe) har som indata en textfil med kommandon och beroenden som skall tolkas. Om man inte anger namnet på indatafilen när make-programmet startas kommer programmet att försöka öppna en fil med standardnamnet "makefile". I våra programprojektmappar har vi alltid en fil med detta namn.

I våra projekt kommer vi att använda en makefile som är gjord av Eric Weddington. Denna utgör en bra grund att stå på och kan modifieras efter behov. Jag kommer att lägga in kommentarer och markera vissa delar som är väsentliga för oss som användare.

Observer att en kommentar i makefilen inleds med tecknet #.

```
# WinAVR Sample makefile (c) 2002-2003 Eric B. Weddington
# Released to the Public Domain
# Please read the make user manual!
#
#
```

Anropet av makefilen kan göras på 3 olika sätt. Normalt så startar vi upp make utan argument, vilket är ekvivalent med att anropa make med argumentet "all". Detta kommando gör att källkodsfiler som behöver kompileras kompileras och därefter länkas ett nytt program ihop av alla objektkodsfiler, som tillhör projektet. Om det finns behov av att ta bort alla filer som skapas automatiskt vid kompilering, länkning, etc. kan detta göras genom att skicka med argumentet "clean".

```
# On command line:
# make all = Make software.
# make clean = Clean out built project files.
# make coff = Convert ELF to COFF using objtool.

#
# To rebuild project do make clean then make all.
#
```

Först kommer en definition av ett antal variabler som används längre fram. Den viktigaste för oss är MCU-variabeln som anger vilken AVR-processor vi kommer att skapa ett program för.

```
# MCU name
MCU = atmega16

# Output format. (can be srec, ihex)
FORMAT = ihex
```

TARGET-variabeln anger namnet på den fil som innehåller main-funktionen. I våra projekt låter vi normalt denna fil heta "main.c"

```
# Target file name (without extension).
TARGET = main
```

```
# Optimization level (can be 0, 1, 2, 3, s)
# (Note: 3 is not always the best optimization level.
OPT = s
```

Alla i programprojektet ingående källkodsfiler i C-kod respektive assemblerkod skall anges med hjälp av variablerna SRC och ASRC.

```
# List C source files here.
```

```
SRC = $(TARGET).c \
    io_ai.c \
    io_di.c \
    io_do.c \
    io_usart.c \
    delay_loop.c \
    lcd4.c
```

```
# List Assembler source files here.
```

```
ASRC = bouncing.s delay.s leds_a1.s
```

```
# Optional compiler flags.
```

```
CFLAGS = -g -O$(OPT) -funsigned-char \
    -funsigned-bitfields -fpack-struct \
    -fshort-enums -Wall -Wstrict-prototypes -Wa,-ahlms=$(<:.c=.lst)
```

```
# Optional assembler flags.
```

```
ASFLAGS = -Wa,-ahlms=$(<:.s=.lst), -gstabs
```

```
# Optional linker flags.
```

```
LDFLAGS = -Wl,-Map=$(TARGET).map,--cref
```

```
# Additional library flags (-lm = math library).
```

```
LIBFLAGS = -lm
```

---

```
# Define directories, if needed.
```

```
DIRAVR = c:/winavr
DIRAVRBIN = $(DIRAVR)/bin
DIRAVRUTILS = $(DIRAVR)/utils/bin
DIRINC =
DIRLIB = $(DIRAVR)/avr/lib
```

```
# Define programs and commands.
```

```
SHELL = sh
CC = avr-gcc
OBJCOPY = avr-objcopy
OBJDUMP = avr-objdump
NM =avr-nm
REMOVE = rm -f
COPY = cp
ELFCOFF = objtool
```

```
HEXSIZE = @avr-size --target=$(FORMAT) $(TARGET).hex
ELFSIZE = @avr-size $(TARGET).elf
```

```
FINISH = @echo Errors: none
```

```
BEGIN = @echo ----- begin -----
```

```

END = @echo ----- end -----

# Define all object files.
OBJ = $(SRC:.c=.o) $(ASRC:.s=.o)

# Define all listing files.
LST = $(ASRC:.s=.lst) $(SRC:.c=.lst)

# Combine all necessary flags and optional flags.
# Add target processor to flags.
ALL_CFLAGS = -mmcu=$(MCU) -I. $(CFLAGS)
ALL_ASFLAGS = -mmcu=$(MCU) -I. -x assembler-with-cpp $(ASFLAGS)
ALL_LDFLAGS = -mmcu=$(MCU) $(LDFLAGS)

```

Regeln "all" anger hur målsystemet skall byggas ihop.

```

# Default target.
all: begin gccversion sizebefore $(TARGET).elf $(TARGET).hex
      $(TARGET).eep $(TARGET).lss sizeafter finished end

# Eye candy.

begin:
      $(BEGIN)

finished:
      $(FINISH)

end:
      $(END)

# Display size of file.

sizebefore:
      @echo Size before:
      -$(HEXSIZE)

sizeafter:
      @echo Size after:
      $(HEXSIZE)

# Display compiler version information.
gccversion :
      $(CC) --version

# Target: Convert ELF to COFF for use in debugging / simulating
# in AVR Studio.

coff: $(TARGET).cof end

%.cof: %.elf
      $(ELFCOFF) loadelf $< mapfile $*.map writecof $@

# Create final output files (.hex, .eep) from ELF output file.

```

```
% .hex: %.elf
        $(OBJCOPY) -O $(FORMAT) -R .eeprom $< $@

%.eep: %.elf
        -$(OBJCOPY) -j .eeprom --set-section-
flags=.eeprom="alloc,load" --change-section-lma .eeprom=0 -O
$(FORMAT) $< $@

# Create extended listing file from ELF output file.
%.lss: %.elf
        $(OBJDUMP) -h -S $< > $@
        $(NM) -a -g -n $<

# Link: create ELF output file from object files.
.SECONDARY : $(TARGET).elf
.PRECIOUS : $(OBJ)
%.elf: $(OBJ)
        $(CC) $(ALL_LDFLAGS) $(OBJ) $(LIBFLAGS) --output $@

# Compile: create object files from C source files.
%.o : %.c
        $(CC) -c $(ALL_CFLAGS) $< -o $@

# Assemble: create object files from assembler source files.
%.o : %.s
        $(CC) -c $(ALL_ASFLAGS) $< -o $@
```

Regeln "clean" rensar bort alla automatgenererade filer och minskar därmed minnesutrymmet som projektet upptar till en bråkdel.

```
# Target: clean project.
clean: begin clean_list finished end

clean_list :
        $(REMOVE) $(TARGET).hex
        $(REMOVE) $(TARGET).eep
        $(REMOVE) $(TARGET).obj
        $(REMOVE) $(TARGET).cof
        $(REMOVE) $(TARGET).elf
        $(REMOVE) $(TARGET).map
        $(REMOVE) $(TARGET).obj
        $(REMOVE) $(TARGET).a90
        $(REMOVE) $(TARGET).sym
        $(REMOVE) $(TARGET).lnk
        $(REMOVE) $(TARGET).lss
        $(REMOVE) $(OBJ)
        $(REMOVE) $(LST)

# Automatically generate C source code dependencies.
# (Code taken from the GNU make user manual.)
# Note that this will work with sh (bash) and sed that is
# shipped with WinAVR (see the SHELL variable defined above).
# This may not work with other shells or other sed's.
%.d: %.c
```

```

set -e; $(CC) -MM $(ALL_CFLAGS) $< \
| sed 's/\($*\)\.o[ :]*/\1.o $@ : /g' > $@; \
[ -s $@ ] || rm -f $@

# Remove the '-' if you want to see
# the dependency files generated.
-include $(SRC:.c=.d)

# Listing of phony targets.
.PHONY : all begin finish end sizebefore sizeafter gccversion
coff clean clean_list

```

## 7.6 Uppgifter

### **U1: Teorifrågor om GNU as**

- a) Föklara vad en label är.
- b) Ge ett svenskt ord för det engelska ordet label.
- c) Ett assemblerprogram byggs upp med hjälp av assemblerinstruktioner och assemblerdirektiv. Föklara vilken skillnad som råder mellan instruktion och direktiv.
- d) Vilka är reglerna för hur ett symboliskt namn bildas i GNU assemblern respektive GNU C?
- e) I ett assemblerprogram finns följande deklaration:  

```
u:      .space 4
```

Adressetiketten u har det numeriska värdet 0x0080. Vilket numeriskt värde har då u+3?
- f) Ett minnesutrymme skall reserveras. I C-kod skulle följande deklaration ha gjorts:  

```
unsigned int gN = -2;
```

Reservera detta minnesutrymme i assemblerkod:
- g) Antag att adressetiketten VAR har värdet 0x7000. Ange vad som hamnar i minnet från adress 0x7000 då följande assemblerdirektiv används:  

```
VAR:    .byte   0xAA
        .byte   0b10001011
        .word   -20
        .long   20
```

Ange det hexadecimala värdena.
- h) En vektor i C-kod :  

```
char vec[4]={ 1, 2, 3, 4 };
```

Skall deklareras i assembler.

## **U2: Några små additionsprogram**

- a) Skriv ett assemblerprogram som adderar två 8-bitars tal lagrade på minnesplatserna vU och vV. Resultatet skall lagras som ett 8-bitarstal på minnesplatsen vZ.
- b) Skriv ett assemblerprogram som adderar två 16-bitars tal lagrade på minnesplatserna vU och vV. Resultatet skall lagras som ett 16-bitarstal på minnesplatsen vZ.

## **U3: Från C-kod till assemblerkod**

- a) Skriv de instruktioner som behövs för att utföra följande tilldelningssats i C:

`char` va, vb, vc;

...

a = b+c;

Antag att variablerna va,vb och vc är globala och skall deklareras i asssembler.

- b) Skriv de instruktioner som behövs för att utföra följande tilldelningssats i C:

`int` a, b, c;

...

a = b+c;

Antag att variablerna va,vb och vc är globala och skall deklareras i asssembler.

## **7.7 Svar**

### **U1: Teorifrågor om GNU as**

- a) En label är en adressetikett, dvs ett symboliskt namn för en address.

- b) -

- c) -

- d) En symbol (identifierare) bildas av följande tecken:

a,b,c,...,x,y,z,A,B,C,...,X,Y,Z

0,1,2,3,4,5,6,7,8,9

-

.

\$

Ett symbolnamn får ej starta med en siffra.

- e)  $u+3=0x0080+0x0003=0x0083$

- f) gN: .word -2

g)

7000	AA
7001	8B
7002	FF
7003	EC
7004	00
7005	00
7006	00
7007	14

h)

```
.global vec
vec: .byte 1,2,3,4
```

## U2: Några små additionsprogram

a)

```
.data
.comm vU,1
.comm vV,1
.comm vZ,1

.text
.global addera
addera:
LDS R24, vU
LDS R25, vV
ADD R25,R24
STS vZ ,R25
RET
```

b)

```
.data
.comm vU,2
.comm vV,2
.comm vZ,2

.text
.global addera2
addera2:
LDS R24, vU
LDS R25, vU+1
LDS R22, vV
LDS R23, vV+1
ADD R24,R22
ADC R25,R23
STS vZ ,R24
STS vZ+1 ,R25
RET
```

### U3: Från C-kod till assemblerkod

- a) Reservering av minnesutrymme kan göras på olika sätt. Direktivet comm har den fördelen att adressläget blir globalt automatiskt

```
.data
.comm    va, 1    //variable 1

.global vb      //variable 2
vb:     .space 1

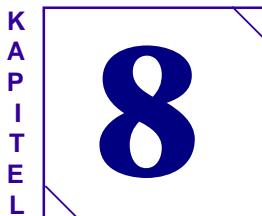
.comm    vc, 1

.text
...
LDS      R20, va      ; R20=a
LDS      R21, vb      ; R21=b
ADD      R21, R20    ; R21=R21+R20=a+b
STS      vc, R21    ; c=R21
```

b)

```
.data
.comm    va, 2    //2 bytes minnesutrymme
.comm    vb, 2
.comm    vc, 2

.text
...
LDS      R0, va      ; va low byte
LDS      R1, va+1    ; va high byte
LDS      R2, vb      ; vb low byte
LDS      R3, vb+1    ; vb high byte
ADD      R0, R2      ; vc=va+vb=R0: R1
ADC      R1, R3
STS      vc, R0      ; vc low byte = R0
STS      vc+1, R1    ; vc high byte = R1
```



## C och assembler

---

### 8.1 Högnivåspråket C på maskinnivå

Hur använder GNU C-kompilatorn registrenna och stacken för att implementera funktioner och parameteröverföring? I tabellen som följer står f() för en funktion och f(p8) står för en funktion med en parameter på 1 byte (8 bitar). Parametern kan då vara av typen unsigned char eller signed char. På analogt sätt står p16 för en två bytes parameter av typen int, unsigned int eller en pekare. Parametern p32 är en fyra bytes parameter av typen long eller float.

<u>Parameters</u>	<u>Param 1</u>	<u>Param 2</u>	<u>Param 3</u>	<u>Param 4</u>
f(p8)	R25: R24			
f(p16)	R25: R24			
f(p16, p16)	R25: R24 <sup>1</sup>	R23: R22		
f(p16, p16, p16)	R25: R24	R23: R22	R21: R20	
f(p32, p32)	R25: R24: R23: R22	R21: R20: R19: R18		
f(p32, p32, p32)	R25: R24: R23: R22	R21: R20: R19: R18	R17: R16: R15: R14	

Om du skall skriva en subrutin i assembler som skall kunna anropas från en C-funktion, bör du alltid kolla upp med den aktuella versionen av C-kompilatorn om vilka register parametrarna hamnar i. Enklast gör du detta genom att skriva en C-funktion och studera den assemblerkoden som genereras från denna, på så sätt kan du övertyga dig om vilka register och parametrar som hör ihop.

#### Returneringsvärde (return value)

p8	R25: R24 <sup>2</sup>
p16	R25: R24
p32	R25: R24: R23: R22

<sup>1</sup>MSB left, LSB right (r25 - lower, r24 - higher, and so on)

<sup>2</sup>R25 is cleared

**Hur får registerna användas i en C-funktion?**

- R0 Får användas utan restriktioner, s. k. scratch pad register.
- R1 0 (förutsätts alltid att vara 0 av C-kompilatorn)
- R2-R17 Skall alltid återställas till sitt ursprungliga värde innan en funktion anropas (register preserved by caller).
- R18-R25 Används för att överföra parametrar till en funktions.
- Får användas utan restriktioner, s. k. scratch pad register.
- R26-R27 Får användas utan restriktioner, s. k. scratch pad register.
- R28-R29 Skall alltid återställas till sitt ursprungliga värde innan en funktion anropas (register preserved by caller).
- R30-R31 Får användas utan restriktioner, s. k. scratch pad register.

Sett från GNU-kompilatorns synvinkel finns det två typer av register: Register som får användas utan restriktioner så kallade scratch pad register (kladblocksregister) och register vars innehåll ej får förändras av en funktion, på engelska benämns dessa register som "register preserved by caller". Register som ej får förändras kan användas i en funktion om de sparar på stacken<sup>1</sup> och återställs innan ett funktionsanrop eller då vi lämnar en funktion.

**8.2 Uppgifter****U1: Teorifrågor om C på maskinnivån**

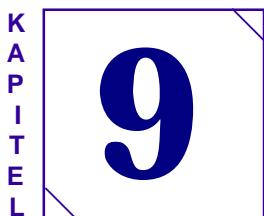
- a) En funktion har följande prototyp: int Func(); I vilka register skall funktionen returnera beräknat resultat?
- b) En funktion har följande prototyp: int Func(int x, int v); Till vilka register skall parametrarna x och v överföras?

**8.3 Svar****U1: Teorifrågor om C på maskinnivån**

- a) R25:R24
- b) Parameter x överförs i registerparet R25:R24 och parameter v överförs i registerparet R23:R22.

---

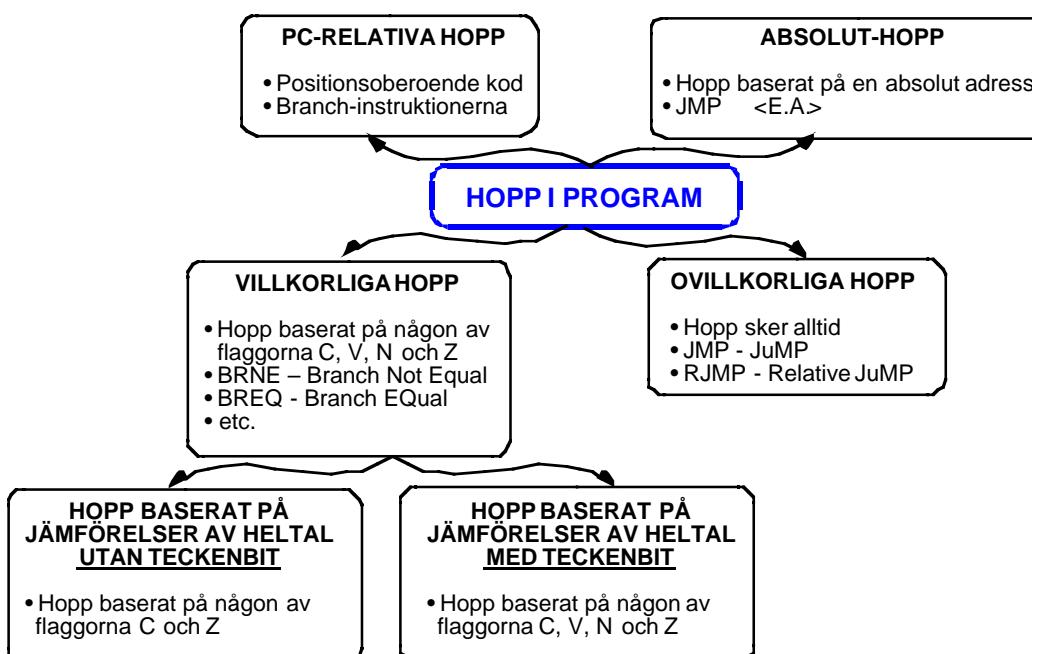
<sup>1</sup>Se mer i kapitel 10 om stacken och dess användning.



# Villkorliga och repetetiva programsatser i assembler

## 9.1 Hopp i ett program

För att implementera högnivåspråksstrukturer, som if-else-satser, while-satser, etc. krävs på maskinnivån att hopp kan utföras i programmet från en programminnesadress till en annan. Ett hopp innebär på maskinnivån att programräknaren PC laddas med ett nytt värde, programexekveringen skall fortsätta från en annan plats i minnet. Vid sekventiell exekvering av instruktioner så stegas programräknaren upp för att peka på instruktionen efter den som exekveras. Till en processor av typen AVR finns det en mängd olika hoppinstruktioner och begrepp relaterat till dessa. I figuren som följer sammanfattas dessa begrepp:



*Figur 9.1: Begrepp relaterat till hopp i ett program*

Ett ovillkorligt hopp är ett hopp som alltid utförs. *Ovillkorliga hopp* kan göras på två olika sätt: Det första är *absolut hopp* där programräknaren PC i processorn laddas med ett nytt värde i form av en fullständig minnesadress.

$$PC \leftarrow \text{Adress}$$

Den andra type av hopp som är viktigt bland annat för att implementera positionsoberoende kod är, PC-relativa hopp. Ett *PC-relativt hopp* görs genom att addera ett offset till programräknarens aktuella värde.

$$PC \leftarrow PC + \text{offset}$$

Med *positionsoberoende kod* menas programkod som kan läsas in till godtycklig plats i programminnet för att sedan exekveras. I våra små inbyggnadssystem är det sällan nödvändigt, däremot måste koden till program skrivna för en persondator alltid vara positionsoberoende.

Villkorliga hopp används för att implementera villkorliga programsatser av typen if-else samt för repetetiva programsatser av typen while. Ett *villkorligt hopp* utförs om något villkor är uppfyllt. På maskinnivå i en processor baserar sig de villkorliga hoppen på flaggorna i statusregistret SREG. Vissa villkor på flaggorna skall vara uppfyllda för att ett hopp skall ske, annars så stegas programräknaren fram till nästa instruktion.



**Figur 9.2:** Statusregistret SREG

I statusregistret SREG är det endast de aritmetiska flaggorna (V,N,Z,C) som är av intresse för villkorliga hopp.

På maskinnivå baserar sig de villkorliga hoppen på en jämförelse av två tal med eller utan teckenbit:

$$\text{tal1} \geq \text{tal2}, \text{ tal1} > \text{tal2}, \text{ tal1} = \text{tal2}, \text{ tal1} \neq \text{tal2}, \text{ tal1} < \text{tal2}, \text{ tal1} \leq \text{tal2}$$

Jämförelsen av talen görs genom att utföra subtraktionen

$$\text{tal1} - \text{tal2}$$

och sedan resultatet med noll

$$\text{tal1} - \text{tal2} \geq 0, \text{ tal1} - \text{tal2} > 0, \text{ tal1} - \text{tal2} = 0,$$

$$\text{tal1} - \text{tal2} \neq 0, \text{ tal1} - \text{tal2} < 0, \text{ tal1} - \text{tal2} \leq 0$$

Olika kombinationer på flaggorna V, N, Z och C motsvarar olika jämförelser.

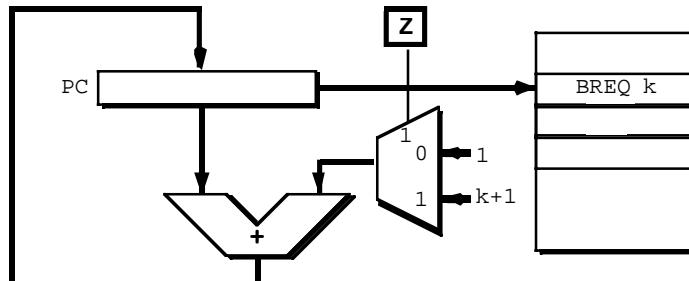
Antag att vi vill göra ett hopp om talen är lika ( $\text{tal1} - \text{tal2} = 0$ ) respektive olika ( $\text{tal1} - \text{tal2} \neq 0$ ), vilket krav skall vi då ha på flaggorna? I detta fallet räcker det med att basera hoppen på zero-flaggan Z. Om talen är lika ( $\text{tal1} - \text{tal2} = 0$ ) är Z=1, om talen är olika ( $\text{tal1} - \text{tal2} \neq 0$ ) är Z=0.

### 9.1.1 BREQ – Hoppa om lika med (BRanch if EQual)

Instruktionen *BREQ – BRanch if EQual*, utför ett hopp om Z-flaggan i statusregistret SREG är 1 (om flaggan är noll utförs inget hopp). Om instruktionen exekveras direkt efter någon av instruktionerna CP, CPI, SUB eller SUBI, så kommer hoppet att utföras endast om de två operandregistrena Rd och Rr är lika.

**Operation:**

if Z=1 then PC  $\leftarrow$  PC + k + 1 else PC  $\leftarrow$  PC + 1  
**alternativ tolkning**  
if Rd=Rr then PC  $\leftarrow$  PC + k + 1 else PC  $\leftarrow$  PC + 1

**Dataflöde:****Syntax:**BREQ k  $-64 \leq k \leq 63$ **16 bitars op-kod:**

1	1	1	1	0	0	k	k	k	k	k	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---

I T H S V N Z C

**Statusregistret SREG:**

-	-	-	-	-	-	-	-
---	---	---	---	---	---	---	---

**Klockcykler:**

1 om inget hopp utförs och 2 om ett hopp utförs.

**Exempel:**

CP	R0 ,R1	;Jämför R0 med R1 genom att ; utföra R0-R1
BREQ	LIKA	;Hopp till läget LIKA om Z=1 ;dvs om R0=R1.
• • •		

LIKA: LDI R0 ,0x00

### 9.1.2 Hopp baserat på jämförelse av två tal

AVR-processorn har ett antal villkorliga hoppinstruktioner för att avgöra relationen mellan två tal.

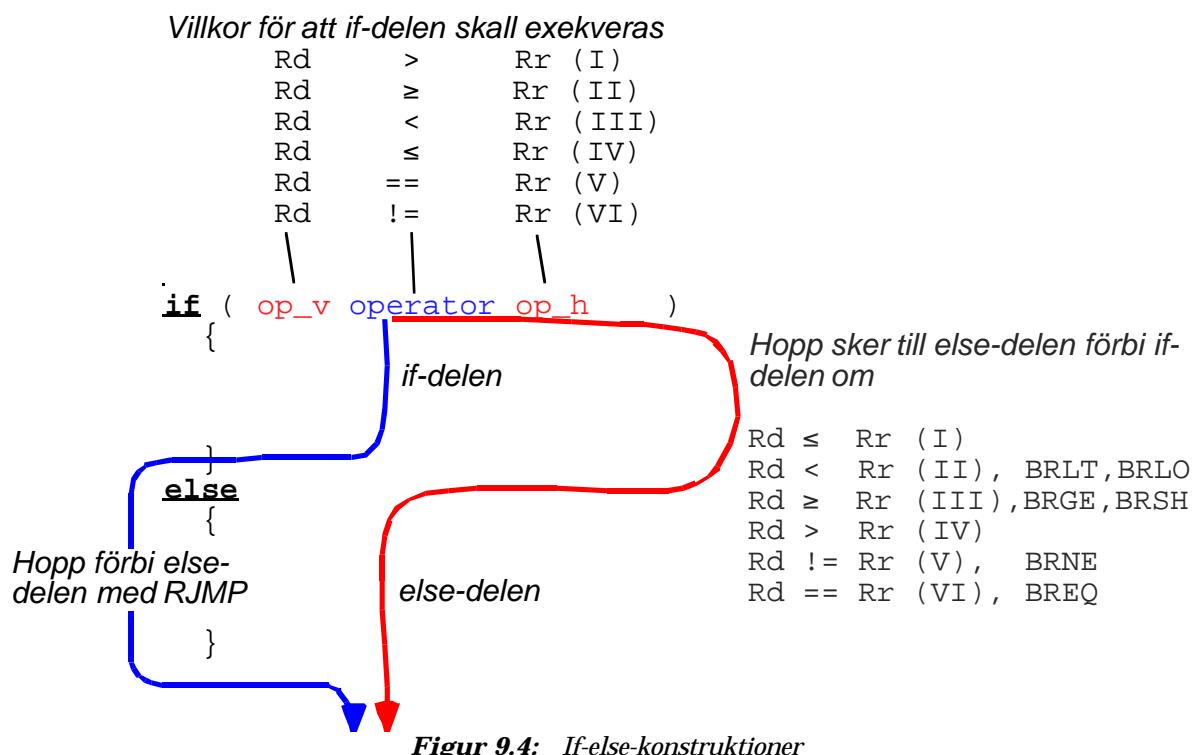
Test	Boolsk test	Mnemonic	Kompl. test	Boolsk test	Mnemonic	Talformat
Rd > Rr	$Z \bullet (N \oplus V) = 0$	BRLT(1)	$Rd \leq Rr$	$Z + (N \oplus V) = 1$	BRGE*	Signed
$Rd \geq Rr$	$(N \oplus V) = 0$	BRGE	$Rd < Rr$	$(N \oplus V) = 1$	BRLT	Signed
$Rd = Rr$	$Z = 1$	BREQ	$Rd \neq Rr$	$Z = 0$	BRNE	Signed
$Rd \leq Rr$	$Z + (N \oplus V) = 1$	BRGE(1)	$Rd > Rr$	$Z \bullet (N \oplus V) = 0$	BRLT*	Signed
$Rd < Rr$	$(N \oplus V) = 1$	BRLT	$Rd \geq Rr$	$(N \oplus V) = 0$	BRGE	Signed
$Rd > Rr$	$C + Z = 0$	BRLO(1)	$Rd \leq Rr$	$C + Z = 1$	BRSH*	Unsigned
$Rd \geq Rr$	$C = 0$	BRSH BRCC	$Rd < Rr$	$C = 1$	BRLO BRCS	Unsigned
$Rd = Rr$	$Z = 1$	BREQ	$Rd \neq Rr$	$Z = 0$	BRNE	Unsigned
$Rd \leq Rr$	$C + Z = 1$	BRSH(1)	$Rd > Rr$	$C + Z = 0$	BRLO*	Unsigned
$Rd < Rr$	$C = 1$	BRLO BRCS	$Rd \geq Rr$	$C = 0$	BRSH BRCC	Unsigned
Carry	$C = 1$	BRCS	No carry	$C = 0$	BRCC	Simple
Negative	$N = 1$	BRMI	Positive	$N = 0$	BRPL	Simple
Overflow	$V = 1$	BRVS	No overflow	$V = 0$	BRVC	Simple
Zero	$Z = 1$	BREQ	Not zero	$Z = 0$	BRNE	Simple
1. Byt plats på Rd och Rri operationen före testet, i.e., CP Rd,Rr -> CP Rr,Rd						

**Figur 9.3:** Hoppinstruktioner

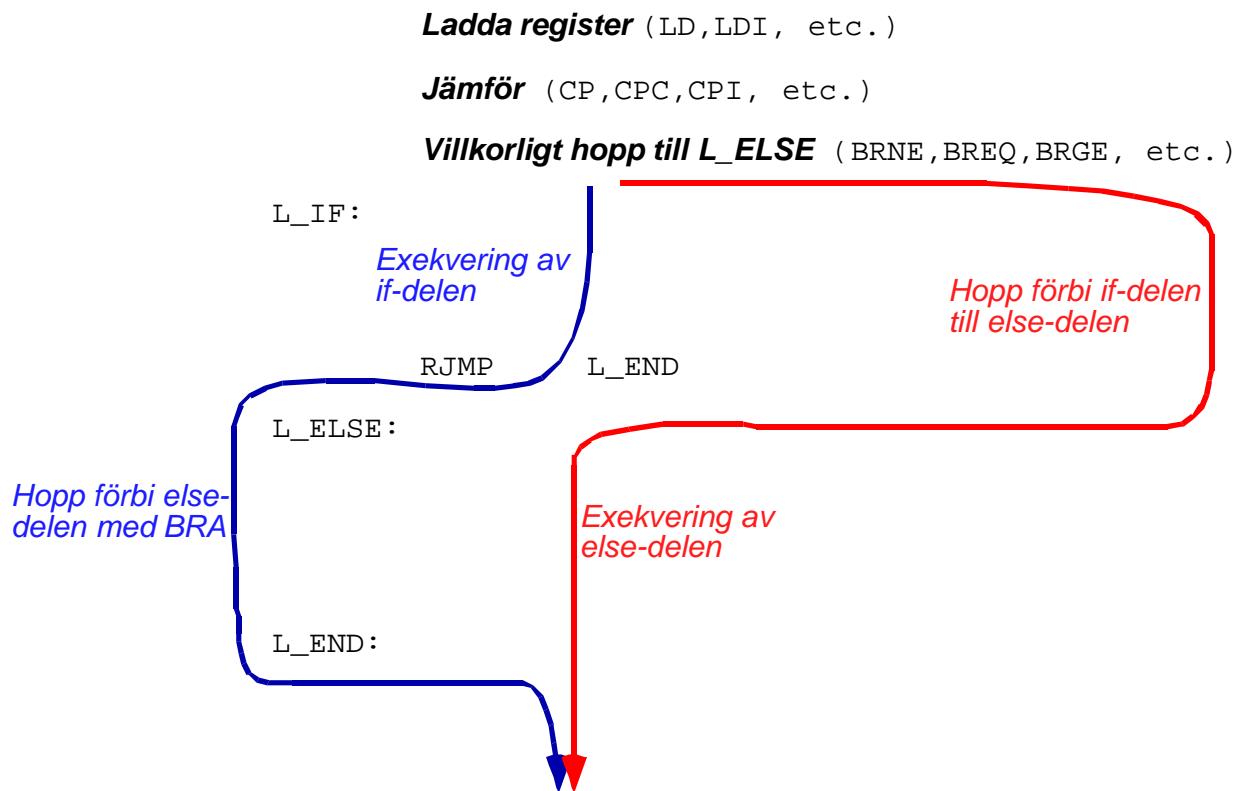
Processorn stödjer hoppinstruktioner för  $=$  (BREQ),  $\neq$  (BRNE),  $\geq$  (BRGE, BRSH) samt  $<$  (BRLT, BRLO). Om vi vill göra ett hopp då  $Rd > Rr$  så finns det t.ex. ingen instruktion BRGT-BRanch Greater Than. Ett enkelt sätt att lösa detta är att att byta plats på registrervärdena Rd och Rr och utföra det eventuella hoppet med BRLT.

## 9.2 If-else-konstruktioner på assembler

Det finns olika sätt att betrakta villkorliga hopp i assemblerkod. Ett sätt som vi skall använda är att knyta an till en enkel if-else-struktur. Beteckningen Rd i figuren nedan står för ett av processorns register vilket jämförs med ett annat register som betecknas med Rr. Ett testvillkor i en if-sats måste konstrueras på ett sådant sätt att vi i slutändan har ett enkel test mellan två processorregister. Med användade av test- och jämförelseinstruktioner kan vi sedan påverka processorns flaggor så att dessa speglar relationen mellan registrena Rd och Rr och därefter utföra ett villkorligt hopp baserat på någon eller några av flaggorna.



Vi kommer att välja att göra det villkorliga hoppet till else-delen och om inget hopp sker exekverar vi if-delen och hoppar sedan ovillkorligt förbi else-delen.



**Figur 9.5:** If-else-strukturen i assembler

### ■ Exempel 9.1: If-else i assembler med jämförelser på tal av typen char

Antag att vi har följande programexempel i C-kod:

char u, v, z; //Tre hel tal svarar till tre teckenbitar

//...

```
if ( u < v )
    z=u;
else
    z=v;
```

Ovanstående programsats kan skrivas om till

```
if ( u - v < 0 )
    z=u;
else //Hopp till else-del en sker på u-v≥0
    z=v;
```

Om hopp sker till else-delen skall hoppet ske på  $u-z \geq 0$ . Detta innebär att hoppinstruktionen BRGE – BRanch Greater or Equal skall användas.

Ovanstående kan implementeras i assembler på följande sätt:

.data

```

.comm u,1      ;8 bitars heltal med teckenbit
.comm v,1      ;8 bitars heltal med teckenbit
.comm z,1      ;8 bitars heltal med teckenbit

.text

LDS    R18 , u
LDS    R20 , v
CP     R18 , R20 ;R18-R20
BRGE  L01_ELSE ;Hopp om R18≥R20

L01_IF:
; ; ... u<v ...
LDS    R18 , u    //Onödigt då R18=u redan
STS    z , R18
RJMP  L01_END

L01_ELSE:
; ; ... u ≥ v ...
LDS    R18 , v
STS    z , R18

L01_END:

```

I exemplet ovan ser vi att för evaluering av en olikhet måste vi på något sätt utföra en subtraktion som ställer flaggorna i statusregistret SREG. I exemplet ovan valde jag CP-instruktionen, SUB-instruktionen hade dock gått lika bra.

**Slut exempel 9.1** ■

### ■ Exempel 9.2: If-else i assembler med jämförelser på tal av typen int

Antag att vi har följande programexempel i C-kod:

```
int u, v, z; //Tre heltalsvariabler med teckenbit
```

```
//•••
```

```
if ( u > v)
z=u;
else
z=v;
```

Ovanstående programsats kan skrivas om till

```
if ( u - v > 0)
z=u;
else          //Hopp till else-del en sker på u-v≤0
z=v;
```

Vi vill kunna hoppa till else-delen då  $u-v \leq 0$ . Tyvärr så finns ingen instruktion ~~BRLE~~—~~Branch Less or Equal~~. Hur bär vi oss åt då? Vi negerar vänsterled och högerled i jämförelsen, samtidigt som vi byter från mindre än < till större än >.

```
if ( -u + v < 0)
z=u;
else          //Hopp till else-del en sker på v-u≥0
z=v;
```

Ovanstående kan implementeras i assembler på följande sätt:

```
.data
.comm u,2      ;16 bitars heltal med teckenbit
```

```

.comm v,2      ; 16 bi tars hel tal med teckenbit
.comm z,2      ; 16 bi tars hel tal med teckenbit

.text

LDS    R18 , u
LDS    R19 , u+1
LDS    R20 , v
LDS    R21 , v+1
CP     R20 , R18 ; R20:R21 - R18:R19
CPC   R21 , R19
BRGE  L01_ELSE ; Hopp om R20:R21≥R18:R19

L01_IF:
;; ... u>v -> z=u...
STS   z+1 , R18
STS   z , R19
BRA  L01_END

L01_ELSE:
;; ... u ≤ v -> z=v...
STS   z+1 , R20
STS   z , R21

L01_END:

```

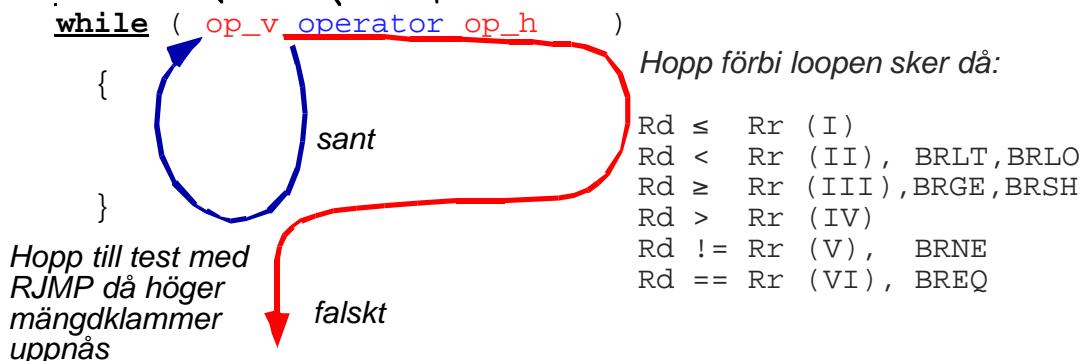
**Slut exempel 9.2** ■

## 9.3 Repetition där antalet varv i loopen ej är känt

TVÅ olika typer av repetition förekommer: I det ena fallet är antalet repetitioner i loopen känt och vi använder någon form av räknare för att implementera denna. I det andra fallet, är antalet varv i loopen okänt och loopen hålls igång med hjälp av ett testvillkor. Detta testvillkor kan vara mer eller mindre komplicerat. I slutända skall vi komma fram till ett test där processorns register ingår och som i sin tur påverkar flaggorna i statusregistret SREG.

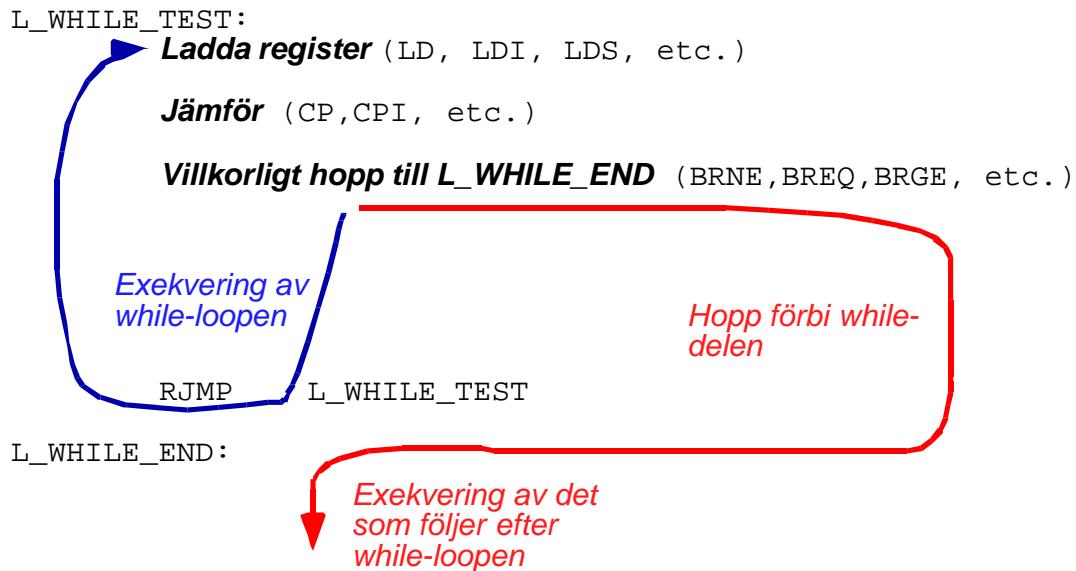
Villkor för att loopen skall exekveras

Rd >	Rr (I)
Rd $\geq$	Rr (II)
Rd <	Rr (III)
Rd $\leq$	Rr (IV)
Rd ==	Rr (V)
Rd !=	Rr (VI)



Figur 9.6: While-konstruktion

I assembler byggs en while-loop upp med hjälp av två hopplägen. Det första hoppläget, L\_WHILE\_TEST, är starten till while-testet, d.v.s. om loopen skall exekveras eller ej. Det andra hoppläget, L\_WHILE\_END, är ett hopp ut ur loopen om testvillkoret blir falskt.



**Figur 9.7:** While-konstruktionen i assembler

## 9.4 Operationer på en vektor

Ett vanligt förekommande programmeringsproblem är att traversera (gå igenom) en vektor. Ett typiskt exempel på detta är summering av alla tal i en vektor. I C-kod kan detta uttryckas på följande sätt:

```

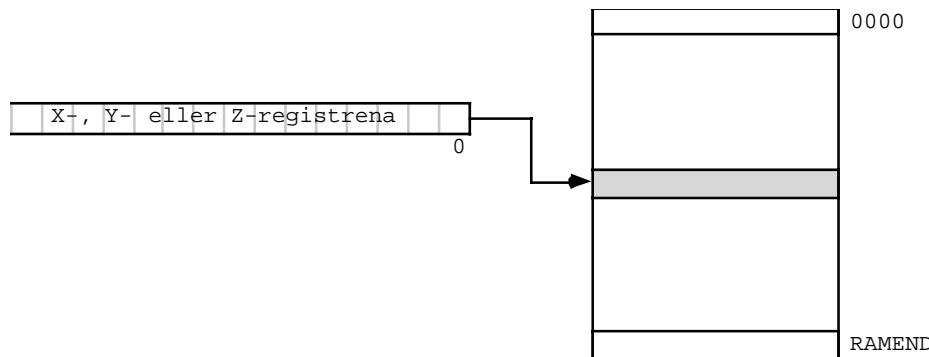
volatile int i;
int sum = 0;
int main(void)
{
    i = 0;
    while (i < 8)
    {
        sum = sum + i;
        i++;
    }
}
    
```

Frågan är nu hur man löser detta på maskinnivå i vår AVR-processor? Innan vi går in på detta skall vi repetera en adresseringsmod som är grunden för att kunna realisera

traversering av en vektor på ett effektivt sätt.

### 9.4.1 Indirekt dataadressering

Vid *indirekt dataadressering* används något av registren X (R27:R26), Y (R29:R28) eller Z (R31:R30) som en adresspekare in i dataminnet (SRAM-minnet). Denna adresseringsmod är användbar då man arbetar med pekare och vektorer.



**Figur 9.8:** Indirekt dataadressering

X-, Y- och Z-registrenna håller 16 bitars adresser. Frågan är då hur laddar vi en 16 bitars address till något av registrenna då alla dataoperationer i processorn är på 8 bitar? Till hjälp har vi två funktioner, en som plockar den högsta byten respektive en som plockar den lägsta byten i en 16 bitarsadress. Funktionerna heter hi och lo.

Antag att en initerad vektor med startadressen (label) iVector finns:

```
i Vector: .word 100, 200, 300, 400, 500, 600, 700, 800
```

Nu vill vi ladda adressen iVector till register Z, d.v.s. registerparet R31:R30, detta görs på enklast sätt med instruktionen LDI (LoaD Immediate):

```
LDI      R30, lo8(i Vector)
LDI      R31, hi 8(i Vector)
```

Då vår processor gör de flesta dataoperationerna på 8 bitars data är vi tvungna att ladda 16-bitarsadressen till register R31:R30 med hjälp av två 8-bitarsoperationer. Funktionen **lo8** ger de 8 minst signifikanta bitarna i ett 16-bitarsvärdet och funktionen **hi8** ger de 8  mest signifikanta bitarna i ett 16-bitarsvärdet.

### 9.4.2 Summering av talen i en vektor

I assemblerkodningen som följer visar vi på hur summering av 8 tal i en vektor kan göras.

Först i assemblerprogrammet kommer deklarationen av de globala variablerna:

```
.data
.global i Vector
; ; int i Vector[8]={100, 200, 300, 400, 500, 600, 700, 800};
i Vector: .word 100, 200, 300, 400, 500, 600, 700, 800
```

```
; ; int sum;
.sum: .global sum
      .word 0

; ; int i ;
.i: .global i
     .word 0
```

Initiering av de globala variablerna, summationsvariablen *sum* och indexeringsvariablen *i*, görs genom att kopiera värdet i register R1 till dessa. I konventionerna för GNU C gäller att detta register alltid innehåller värdet 0.

```
.text

...
;; sum = 0;

STS      sum+1, R1      ; R1=0 i n GNU C fÖr AVR
STS      sum , R1

;; i = 0;

STS      i +1, R1
STS      i , R1
```

Studera koden nedan, som visar på ett sätt att implementera while-loopen, titta speciellt på kommentarerna i koden.

```
; ; while ( i < 8 )

while_test:
    LDS    R24 , i
    LDS    R25 , i +1
    SBIW  R24 , 8          ; Subtraktion i -8
    BRGE  whi_le_end       ; Hopp ut ur loopen dä i -8 ≥ 0

    ; ; {
    ; ;     sum = sum + i Vector[i];
    LDS    R30 , i           ; Z = R31:R30 = 2*i + i Vector
    LDS    R31 , i +1
    ADD    R30 , R30
    ADC    R31 , R31
    SUBI  R30 , lo8(-i Vector)
    SBCI  R31 , hi8(-i Vector)

    LDS    R24 , sum
    LDS    R25 , sum+1
    LD     R18 , Z
    LDD    R19 , Z+1
    ADD    R24 , R18
```

OBServera

```

ADC      R25    , R19
STS      sum+1 , R25
STS      sum   , R24

;; i++;

LDS      R24 , i
LDS      R25 , i +1
ADIW   R24 , 1
STS      i +1 , R25
STS      i   , R24

RJMP    whi_le_test

```

whi\_le\_end:

Observera hur adressen till elementet i vektorn bildas. I C-kod skriver vi på följande sätt:

i Vector[i]

Varje element i denna vektor är av typen int och tar därmed 2 byte i minnesutrymme. På maskinnivå adresseras vektorn med en effektiv address som finns i Z-registret, för att beräkna rätt address vid indexeringen måste vi ta hänsyn till storleken på den datatyp som varje element i vektorn är av:

$Z = R31:R30 = 2*i + i\text{Vector}$

## 9.5 Uppgifter

### U1: Teorifrågor om villkorliga och repetitiva programsatser

- Vilken skillnad föreligger mellan SUB- och CP-instruktionerna?
- När två heltal av datatypen unsigned int ingen teckenbit jämförs, skall ett hopp utföras då  $\text{tal1} \geq \text{tal2}$ , vilken hoppinstruktion skall användas?
- När två heltal av datatypen int teckenbit jämförs, skall ett hopp utföras då  $\text{tal1} \geq \text{tal2}$ , vilken hoppinstruktion skall användas?
- Vilket värde har register R20 efter att nedanstående kod har exekverats?

```

LDI      R20, 0
LDI      R21, 1
SUBI   R21, 2
BREQ   stop
SUBI   R20, 0xFF
stop:
RJMP    stop

```

### U2: Villkorliga hopp för att implementera if-else-satser

- Koda följande if-else-sats i assembler:

**if ( uc == 0 )**

uc=0x01;

där variabeln uc är en global variabel på adressen 0x009A

char uc;

- b) Koda följande if-else-sats i assembler:

```
if ( vc > 50 )
    vc = 0;
else
    vc = vc+1;
```

där variabeln uc är en global variabel på adressen 0x0097

char vc;

## U3: Villkorliga hopp för att implementera repetetiva satser

- a) Koda följande while-sats i assembler:

```
i =0;
sum=0;

while (i <1000)
{
    sum=sum+i ;
    i++;
}
```

där variablerna är av heltalsdatatypen int och lokalt deklarerade i en funktion.  
Antag att registerparet R25:R24 används för den lokala variabeln sum och  
registerparet R19:R18 används för den lokala variabeln i.

## 9.6 Svar

### U1: Teorifrågor om villkorliga och repetitiva programsatser

- a) CP och SUB instruktionen utför båda en subtraktion som kommer att påverka flaggor i statusregistret SREG. Skillnaden är att resultatet av subtraktionen kastas i CP-fallet och sparas i ett register i SUB-fallet.
- b) BRSH (se sid 10 i manualen "AVR Instruction set")
- c) BRGE (se sid 10 i manualen "AVR Instruction set")
- d) R20 har värdet 0x01=1

```
LDI    R20, 0      ; R20=0
LDI    R21, 1      ; R21=1
SUBI   R21, 2      ; R21=R21-2=1-2=-1=0xFF -> Z=0
BREQ  stop        ; Inget hopp sker då Z=0
SUBI   R20, 0xFF   ; R20=R20-0xFF=R20+0x01=0x01
stop:
RJMP  stop
```

## U2: Villkorliga hopp för att implementera if-else-satser

- a) I lösningen nedan används AND-instruktionen för att påverka flaggan Z i statusregistret SREG. Flaggan Z utnyttjas sedan för att avgöra om ett hopp skall ske eller ej. Observera att CPI-instruktionen går lika bra att använda (tar ett ord mera i minnesutrymme).

```
; if ( uc == 0 )
LDS      R16, 0X009A
AND      R16, R16           ; CPI   R16, 0x00 går lika bra
BRNE    L_ELSE

; uc=0x01;
L_IF:
LDI      R16, 0X01
STS      0X009A, R16

L_ELSE:
```

- b) Olika sätt finns att koda if-else-satsen. Vi visar först på kompilatorns variant

```
; if ( vc > 50 )
LDS      R17, 0X0097
CPI      R17, 0X33          ; 51, jämför med 51 inte 50
BRGE    L_IF
RJMP    L_ELSE

; vc = 0;
L_IF:
STS      0X0097, R1
RJMP    L_IF_END

; else
; vc = vc+1;
L_ELSE:
LDS      R18, 0X0097
SUBI    R18, 0xFF
STS      0X0097, R18

L_IF_END:
```

En andra variant på att koda if-else-satsen är enligt metoden som anges i detta kompendium. I denna variant vill vi hoppa till else-delen endast om  $vc \leq 50$  (inget hopp om  $vc > 50$ ). Tyvärr så finns ingen hoppinstruktion för villkoret  $\leq$ , hur gör vi då? Studera följande omskrivningar av testuttrycket:

$$vc \leq 50$$

$$vc - 50 \leq 0$$

$$-vc + 50 \geq 0$$

$$50 - vc \geq 0$$

Vi gör hoppet till else-delen genom att först göra subtraktionen  $50 - vc$  detta för att påverka flaggorna i statusregistret SREG, därefter använder vi hoppinstruktionen

**BRGE – BRanch Greater or Equal ( $\geq$ ).**

```

;; if ( vc > 50 )
LDI      R17, 0X32          ; 50
LDS      R16, 0x0097        ; R17=vc
CP      R17, R16           ; R17-R16=50-vc
BRGE    L_ELSE

;; vc = 0;
L_IF:
STS      0X0097, R1
RJMP    L_IF_END

;; else
;; vc = vc+1;
L_ELSE:
LDS      R18, 0X0097
SUBI    R18, 0xFF
STS      0X0097, R18

L_IF_END:

```

**U3: Villkorliga hopp för att implementera repetetiva satser**

- a) Vi presenterar först GNU C-kompilatorns lösning av problemet:

```

;; i=0;
LDI      R24, 0X00          ; 0
LDI      R25, 0X00          ; 0

;; sum=0;
MOVW    R18, R24

;; while (i < 1000)
;; {
L_WHILE:
;;     sum=sum+i;
ADD      R24, R18
ADC      R25, R19

;;     i++;
SUBI    R18, 0xFF          ; 255
SBCI    R19, 0xFF          ; 255
LDI     R20, 0X03          ; 3
CPI     R18, 0XE8          ; 232
CPC     R19, R20
BRLT    L_WHILE

```

Om vi studerar kodutläggningen för ovanstående while-sats i C så motsvarar den en do-while-sats där vi har looptestet sist.

En andra variant på att koda while-satsen är enligt metoden som anges i detta kompendium. Hopp ut ur loopen sker på villkoret att  $i \geq 1000 \Leftrightarrow i - 1000 \geq 0$ , om det motsatta gäller,  $i < 1000$ , så ligger vi kvar i loopen. Instruktionen BRGE utför hopp på villkoret  $\geq$ .

```

; ; i =0;
LDI      R24, 0X00          ; 0
LDI      R25, 0X00          ; 0

; ; sum=0;
LDI      R18, 0X00          ; 0
LDI      R19, 0X00          ; 0

; ; while (i <1000)
L_WHILE_TEST:
LDI      R20, 0X03          ; 3
CPI      R18, 0XE8          ; 232
CPC      R19, R20
BRGE    L_WHILE_END

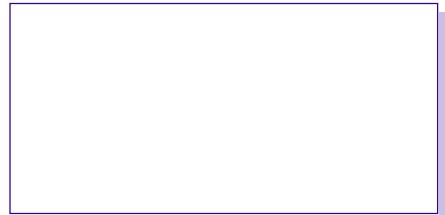
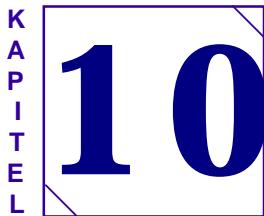
; ; {
; ;     sum=sum+i;
ADD    R24, R18
ADC    R25, R19

; ;     i++;
SUBI   R18, 0xFF          ; 255
SBCI   R19, 0xFF          ; 255

; ; }
RJMP   L_WHILE_TEST

L_WHILE_END:

```



# Modulär programmering med subrutiner

---

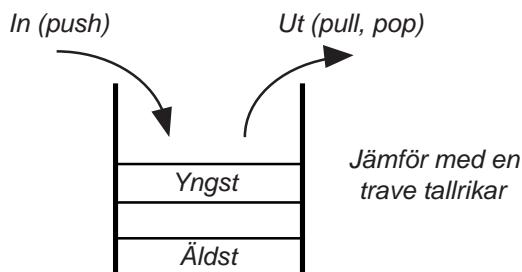
## 10.1 Modularisering av programmet med subrutiner

Från högnivåprogrammering i C känner vi igen begreppet funktion. En funktion är ett stycke programkod som kan anropas en eller flera gånger i ett program, funktionen finns endast i en upplaga. I en mikrodator implementeras detta med hjälp av subrutiner och en stack. En *subrutin* är en programmodul som utför en begränsad och väl definierad uppgift. Huvudprogrammet byggs sedan upp med anrop till olika subrutiner. Observera att det vi kallar subrutin i ett assemblerprogram är en implementering av det vi kallar funktion/procedur i ett högnivåspråk.

### 10.1.1 Stacken

De flesta har en intuitiv förtäelse för dataabstraktionen *stack*, baserat på erfarenhet från vardagslivet. Ett exempel på en stack<sup>1</sup> är en hög med papper på skrivbordet eller en stapel av ett antal tallrikar i ett porslinskåp. I båda fallen är det mest karakteristiska att det som ligger på toppen är det som är lättast åtkomligt. Att lägga till en ny enhet görs lättast genom att lägga den ovanpå alla de andra.

En *stack* är en *linjär lista* av LIFO-typ (Last-In-First-Out). Med detta menas att det element som skall tas bort först är det som lades in sist.



*Figur 10.1: Illustration av hur stacken fungerar*

Stacken är en mycket vanligt förekommande datastruktur då man behöver bearbeta data i motsatt ordning som det kom in.

<sup>1</sup>stack | [hö]stack; stapel, hög, mängd, massa

I mikrodatorn finns det en stack som administreras av stackpekaregistrerat *SP*. Denna stack har ett flertal användningsområden. Det vi behandlar i detta kapitel är användningen av stacken för att lagra återhoppsadresser vid subrutinanrop, föra över parametrar till en subrutin samt att reservera minne för lokala variabler.

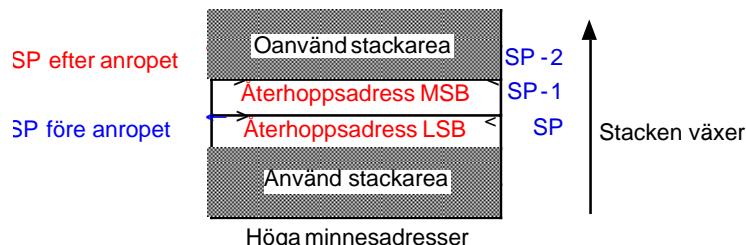
AVR-processorn stödjer en stack genom användning av stackpekaregistrerat *SP* i processorn. Stackpekaregistreret *SP* är 16 bitar brett, detta gör att stacken kan placeras varsomhelst i adressområdet på 64 Kbytes för dataminnet (SRAM). Stackregistret finns mappat i dataminnet på adresserna 0x005D och 0x005E, de finns i I/O-delen av dataminnet och kan nås med I/O-adresserna 0x3D<sup>1</sup> och 0x3E.



**Figur 10.2:** Stackpekaregistreret

Vid RESET på processorn skall stackpekaren initieras till den SRAM-adress där stacken skall ligga. Observera att stackpekaren alltid pekar på den lediga plats där skrivning av nästa data som läggs på stacken skall ske. Stacken växer mot lägre minnesadresser och minskar mot högre minnesadresser.

Stacken används automatiskt av processorns hårdvara vid subrutinanrop. Ett subrutinanrop görs genom användning av instruktionen CALL (Call of Subroutine) eller RCALL (Relative Call of subroutine). RCALL-instruktionen utför ett PC-relativt hopp medan CALL-instruktionen normalt används för att göra absoluthopp. Efter ett subrutinanrop förändras stacken på följande sätt:



**Figur 10.3:** Stackens innehåll efter det att instruktionerna CALL eller RCALL har exekverats

En subrutin avslutas med instruktionen RET (RETurn from subroutine) som manipulerar stacken på motsatta sättet jämfört med CALL-instruktionen. RET-instruktionen läser återhoppsadressen från stacken och laddar programräknaren PC med denna och programexekveringen fortsätter sedan på denna adress.

### 10.1.2 CALL – CALL to subroutine

Instruktionen *CALL – CALL to subroutine* kan utföra ett subrutinanrop till godtycklig address inom programminnet (FLASH-minnet).

#### Operation:

$$\begin{aligned} \text{SP} &\leftarrow \text{SP} - 2 \\ M_{(\text{SP}+1)} : M_{(\text{SP}+2)} &\leftarrow \text{PC} + 2 \end{aligned}$$

Öka på stacken med 2 bytes.  
Återhoppsadress sparas på stacken

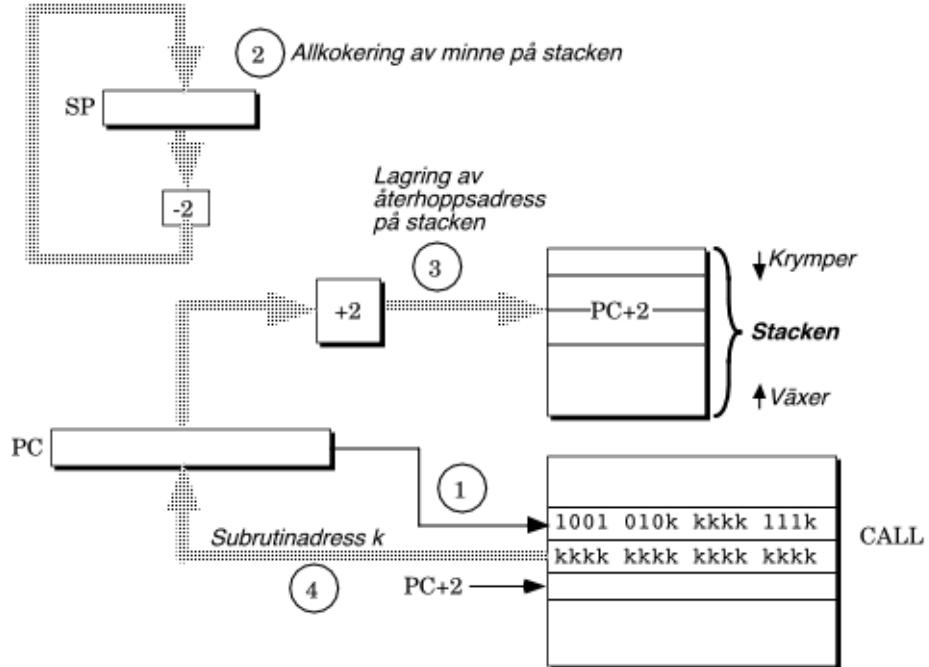
$$\text{PC} \leftarrow k$$

Hopp till subrutinen, **absoluthopp**

<sup>1</sup>Offsetskillnad på 0x20 mellan I/O-adress och dataminnesadress.

**Syntax:**

CALL k

**Dataflöde med en 16 bitars PC:**

- 1) Programräknaren pekar på instruktionen `CALL` som består av 2 ord.
- 2) Stackpekaren minskas med två, d.v.s. utrymmet för återhopsadressen upptar 2 bytes. Stacken växer mot lägre minnesadresser.
- 3) Stacken växer genom att återhopsadressen, d.v.s. adressen till nästa instruktion efter `CALL`, lägges på stacken.
- 4) Programräknaren laddas med adressen till subrutinen.

**32 bitars op-kod:**

1	0	0	1	0	1	0	k	k	k	k	1	1	1	k
k	k	k	k	k	k	k	k	k	k	k	k	k	k	k

**Statusregistret SREG:**

I	T	H	S	V	N	Z	C
-	-	-	-	-	-	-	-

**Klockcykler:**

4 maskincykler med 16 bitars PC

**Exempel:**

CALL routine ; Anrop av subrutinen

...

```

routine:    PUSH    R14      ; Spara R14 på stacken
            ...
            POP     R14      ; Återställ R14 från stacken
            RET      ; Återvänd från subrutinanropet

```

### 10.1.3 RET – RETurn from subroutine

Instruktionen *RET – RETurn from subroutine* gör att vi återvänder till instruktionen efter anropsstället. Programräknaren laddas med en återhoppsadress från stacken.

**Operation:**

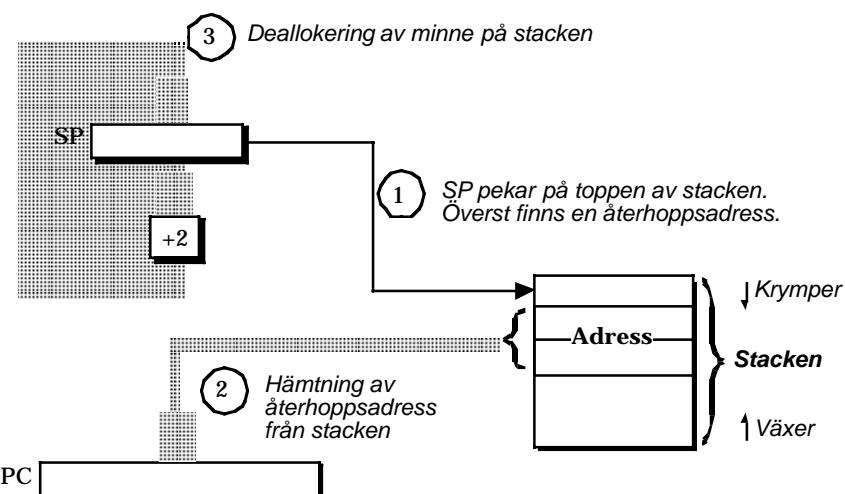
$$\begin{aligned} \text{PC} &\leftarrow M_{(SP+1)} : M_{(SP+2)} \\ \text{SP} &\leftarrow \text{SP} + 2 \end{aligned}$$

Återhoppsadress laddas till PC  
Frisläpp 2 byte på stacken

**Syntax:**

RET

**Dataflöde:**



**16 bitars op-kod:**

1	0	0	1	0	1	0	1	0	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

**Statusregistret SREG:**

I	T	H	S	V	N	Z	C
-	-	-	-	-	-	-	-

**Klockcykler:**

4

**Exempel:**

RCALL routine ; Anrop av subrutinen

...

routine:

...

RET ; Återvänd från  
; subrutinanropet

## ■ Exempel 10.1: Vad händer på stacken vid ett subrutinanrop

Följande lilla C-program skall analyseras:

```

int Add_int_int(int a, int b)
{
    return a+b;
}

int Calculate()
{
    return Add_int_int(10, 20);
}

volatile int m=0;

int main(void)
{
    while (1)
    {
        m = Calculate();
    }
}

```

Nyckelordet volatile används för att kompilatorn ej skall kodoptimera. Anropet på Calculate är helt onödigt då resultatet ej används, men görs för att vi skall kunna studera stackens användning vid ett funktionsanrop. Genom volatile slår vi av kodoptimeringen och får med koden för detta onödiga funktionsanrop.

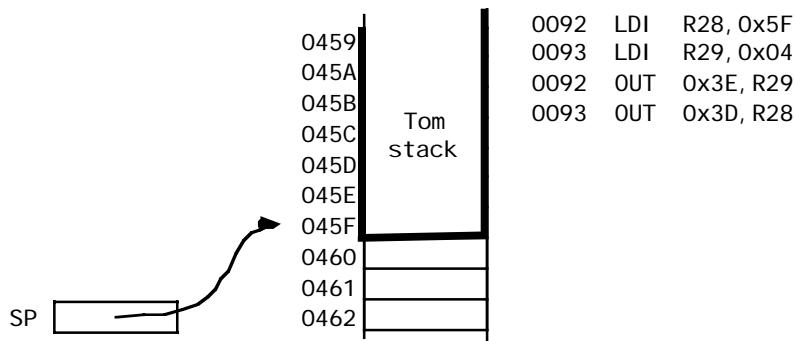
Funktionen main innehåller ett anrop på funktionen Calculate som i sin tur har ett anrop på funktionen Add\_int\_int. För att förstå vad som händer på stacken är vi tvungna att studera maskinkoden som kompilatorn har genererat. Följande maskinkodsdump är tagen från AVRstudio och simulatorn i denna:

<b>@00000088: Add_int_int</b>				
674:       return a+b;				
+00000088: 0F86           ADD      R24, R22	Add without carry			
+00000089: 1F97           ADC      R25, R23	Add with carry			
675:     }				
+0000008A: 9508           RET	Subroutine return			
<b>@0000008B: Calculate</b>				
679:       return Add_int_int(10, 20);				
+0000008B: E164           LDI      R22, 0x14	Load immediate			
+0000008C: E070           LDI      R23, 0x00	Load immediate			
+0000008D: E08A           LDI      R24, 0x0A	Load immediate			
+0000008E: E090           LDI      R25, 0x00	Load immediate			
+0000008F: 940E0088     CALL     0x00000088	Call subroutine			
680:     }				
+00000091: 9508           RET	Subroutine return			
<b>@00000092: main</b>				
684:       int main(void)				
685:     {				
+00000092: E5CF           LDI      R28, 0x5F	Load immediate			
+00000093: E0D4           LDI      R29, 0x04	Load immediate			
+00000094: BFDE           OUT     0x3E, R29	Out to I/O location			
+00000095: BFCB           OUT     0x3D, R28	Out to I/O location			

```

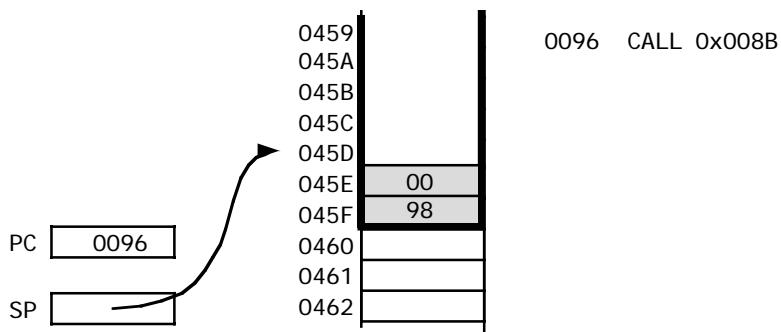
689:      m = Calculate();
+00000096: 940E008B    CALL    0x0000008B      Call subroutine
+00000098: 93900097    STS     0x0097, R25      Store direct to data space
+0000009A: 93800096    STS     0x0096, R24      Store direct to data space
+0000009C: CFF9        RJMP   -0x0007      Relative jump
@
```

Vi börjar med att studera vad som händer i main-funktionens början. Fyra assemblerinstruktioner exekveras. Registerparet R29:R28 laddas med konstanten 0x045F som skrivs till I/O-adresserna 0x3E och 0x3D, detta är en initiering av stackpekarregistret. Stacken växer mot lägre minnesadresser och stackpekaren visar på nästa lediga minnesadress där något kan läggas. Vår bild av stacken blir i form av en behållare, där vi kan lägga i och ta ur data efter FIFO-principen.



**Figur 10.4:** Exekvering av de fyra första instruktionerna i main-funktionen

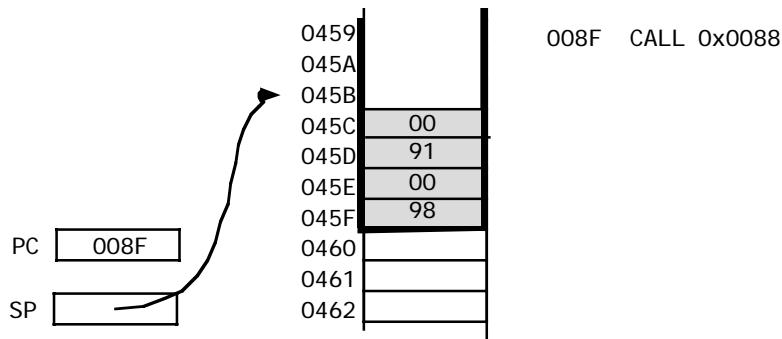
När programräknaren *PC* har värdet 0x0096 sker ett anrop av subrutinen Calculate. När instruktionen CALL utförs, görs förutom hoppet till subrutinen även en skrivning av återhoppsadressen<sup>1</sup> till stacken. Återhoppsadressen är alltid adressen till instruktionen efter CALL, i detta fallet adressen 0x0098.



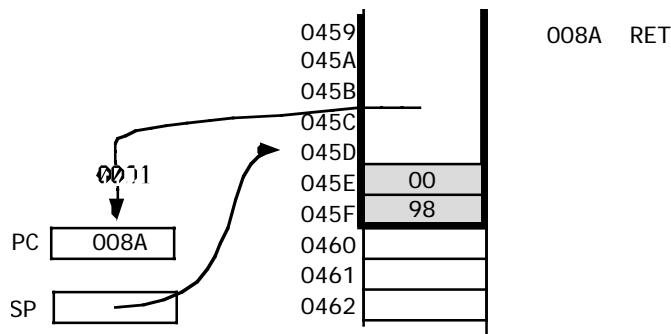
**Figur 10.5:** Exekvering av instruktionen CALL Calculate

I funktionen Calculate sker ett anrop av funktionen Add\_int\_int, detta gör att stacken växer med 2 byte då återhoppsadressen stackas.

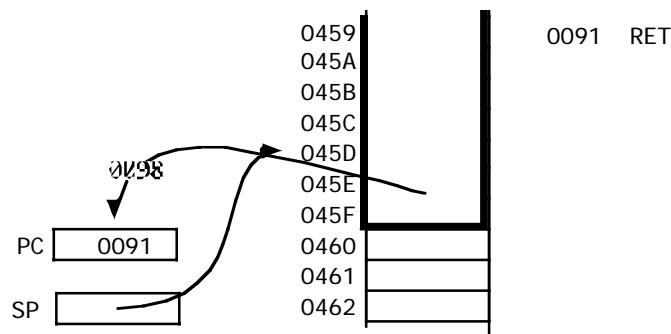
<sup>1</sup>Observera att återhoppsadressen sparas i big endian format. Datalagring är little endian.

**Figur 10.6:** Exekvering av instruktionen CALL Add\_int\_int

Återhopsinstruktionen RET (return) plockar de två översta byten på stacken och laddar programräknaren med detta. I och med detta avslutar vi exekveringen av Add\_int\_int och återvänder till anropsstället i funktionen Calculate.

**Figur 10.7:** Exekvering av instruktionen RET i funktionen Add\_int\_int

När instruktionen RET i funktionen Calculate exekveras, så förbrukas de två sista byten på stacken och vi återvänder tillbaka till main-funktionen och den oändliga while-loopen.

**Figur 10.8:** Exekvering av instruktionen RET i funktionen Calculate

**Slut exempel 10.1** ■

## 10.2 Temporär datalagring på stacken

Det finns instruktioner som tillåter individuella register i processorn att skrivas ner till stacken och att läsning görs från stacken till individuella register. Detta innebär att stacken kan användas för temporär datalagring och tillåter även konstruktioner såsom rekursiva funktioner och reentrantna subrutiner.

Temporär datalagring på stacken används för att implementera två viktiga ideer i ett högnivåspråk, nämligen lokala variabler och överföring av parametrar/argument vid funktionsanrop. I vår AVR-processor med totalt 32 register utnyttjas registren i första hand för lokala variabler och parameteröverföring, när registren ej räcker till utnyttjas stacken. I processorer som ej är av typen RISC, utan av CISC, sker överföring av parametrar och reservering av minne för lokala variabler normalt på stacken.

### **10.2.1 PUSH – PUSH register on stack**

Instruktionen *PUSH – PUSH register on stack* ökar på stacken med 1 byte. Det som läggs på stacken är innehållet i ett av processorns 32 register.

**Operation:**  $M_{(SP)} \leftarrow Rr$       Registret Rr skrivs till stacken  
 $SP \leftarrow SP - 1$       Reservering av 1 byte på stacken

**Syntax:**      PUSH Rr       $0 \leq r \leq 31$

**16 bitars op-kod:**

1	0	0	1	0	0	1	d	d	d	d	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

**Statusregistret SREG:**

I	T	H	S	V	N	Z	C
-	-	-	-	-	-	-	-

**Klockcykler:**      2

**Exempel:**      CALL routine ; Anrop av routine  
 ...  
 routine: PUSH R14 ; Spara R14 på stacken  
 PUSH R13 ; Spara R13 på stacken  
 ...  
 POP R13 ; Återställ R13  
 POP R14 ; Återställ R14  
 RET ; Återvänd från subrutinen

### 10.2.2 POP – POP register from stack

Instruktionen *POP – POP register from stack* läser den byte som finns överst på stacken till registret Rr. Samtidigt så minskas stacken med 1 byte.

**Operation:**  $SP \leftarrow SP + 1$   
 $Rr \leftarrow M_{(SP)}$

**Syntax:** POP Rr  $0 \leq r \leq 31$

**16 bitars op-kod:**

1	0	0	1	0	0	0	d	d	d	d	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

**Statusregistret SREG:**

I	T	H	S	V	N	Z	C
-	-	-	-	-	-	-	-

**Klockcykler:** 4

**Exempel:**

```

CALL routine ; Anrop av routine
...
routine: PUSH    R14    ; Spara R14 på stacken
        PUSH    R13    ; Spara R13 på stacken
        ...
        POP     R13    ; Återställ R13
        POP     R14    ; Återställ R14
        RET      ; Återvänd från subrutinen

```

## 10.3 Överföring av parametrar till subrutiner

Det finns olika metoder att skicka med parametrar vid anrop av en subrutin. I punktform så redogör jag dessa:

- 1) Parametrar skickas med i någon eller några av processorns register R0 till R31  
Denna metod är att föredra om antalet parametrar är få och antalet register är många. Med denna metod kan både indata ges till subrutinen och utdata returneras från subrutinen.
- 2) Parametrar skickas med på stacken. Dessa fungerar i princip som lokala variabler. Detta är den mest generella metoden och används av de flesta kompilatorer.
- 3) Parametrar skickas med via globala variabler. Denna metod bör undvikas, i vissa sammanhang speciellt vid datautbyte mellan parallelexekverande tasks (program) kan det vara befogat.

I praktiken kan man använda en blandning av dessa metoder.

## **10.4 Uppgifter**

### **U1: Teorifrågor om subrutiner och parameteröverföring**

- a) Redogör för principerna för en stack.
- b) Redogör för tre olika sätt att föra över parametrar till en subrutin.
- c) En RISC-processor som AVR-processorn karakteriseras av att den har en väldigt stor registerbank (32 stycken 8-bitarsregister), hur löser GNU-kompilatorn i första hand parameteröverföring till funktioner/subrutiner respektive allokering av minne för lokala variabler?
  - A) I första hand väljs något register om brist på register uppstår används stacken.
  - B) Stacken används i första hand. Register används för beräkningar.
- d) Vad är det som lagras på stacken då instruktionen CALL eller RCALL används?
- e) Vilka av följande påståenden om subrutiner är felaktiga?
  - A) När CALL används lagras en återhoppsadress på stacken.
  - B) En subrutin måste avslutas med ett RET för att kunna hitta tillbaka till anropsstället.
  - C) En stack är ej nödvändigt för att implementera subrutiner.
  - D) Om man hoppar till en subrutin med JMP spårar programmet ur totalt då RET-instruktionen exekveras.
- f) Vilken I/O-adress respektive dataminnesadress har stackpekarregistret SP?

### **U2: Kodning av några subrutiner**

- a) Koda en subrutin swap\_r20\_r21 som byter plats på registrena R20 och R21. Alla andra register måste ha oförändrade värden efter denna operation.  
Tips: Spara ett register på stacken och använd detta som en temporär lagringsplats. Återställ detta register efter att bytet har skett. Ett anrop på subrutinen ser ut på följande sätt:  

```
RCALL    swap_r20_r21
```
- b) Gör en alternativ kodning av subrutinen swap\_r20\_r21 genom att enbart använda dig av EOR-instruktionen. Denna lösning har fördelen att stacken ej behöver användas.

### **U3: Kodning av subrutiner som shall kunna anropas från C**

- a) Koda följande funktion som är skriven i C, i assembler:

```
int max(int a, int b)
{
  if ( a > b )
    return a;
  else
    return b;
```

```
}
```

Parameteröverföringen görs enligt GNU-kompilatorns konventioner. Parametern a förs över i register R25:R24 och parametern b i register R23:R22. Värdet som funktionen returnerar läggs i register R25:R24.

b) Visa hur ett anrop av max-subrutinen görs på assemblernivå, utgå ifrån följande exempel:

```
v = max(10, 20);
```

Antag att den globala variabeln v finns på adressen 0x0090 och 0x0091.

c) Koda följande funktion i assembler

```
int sum_vector(char *vec)
{
    char i=0;
    int sum;

    for (i=0; i<8; i++)
        sum=sum+vec[i];
}
```

## 10.5 Svar

### **U1: Teorifrågor om subrutiner och parameteröverföring**

- a) -
- b) 1. Via globala variabler  
2. Via processorns register  
3. Via stacken.
- c) A)
- d) Återhoppsadressen, adressen till instruktionen direkt efter CALL i minnet.
- e) C)
- f) I/O-adress: 0x3D och 0x3E  
Dataminnesadres 0x5D och 0x5E

### **U2: Kodning av några subrutiner**

- a) Variant 1:

```
swap_r20_r21:
    PUSH   R25
    MOV    R25 , R20    ;R25<-R20
    MOV    R20 , R21    ;R20<-R21
    MOV    R21 , R25    ;R21<-R25
    POP    R25
    RET
```

Variant 2:

```
swap_r20_r21:
    PUSH   R21
    MOV    R21 , R20      ;R21<-R20
    POP    R20
    RET
```

- b) Ursprungsvärdena i register R20 respektive R21 kallas vi A respektive B. För att förstå hur det fungerar skall man komma ihåg att EOR-operationen  $A \oplus A = 0$

```
swap_r20_r21:
    ; ; R20 = A
    ; ; R21 = B
    EOR    R20 , R21      ; R20 = R20  $\oplus$  R21 = A  $\oplus$  B
    EOR    R21 , R20      ; R21 = R21  $\oplus$  R20 = B  $\oplus$  A  $\oplus$  B = A
    EOR    R20 , R21      ; R20 = R20  $\oplus$  R21 = A  $\oplus$  B  $\oplus$  A = B
    RET
```

### U3: Kodning av subrutiner som skall kunna anropas från C

- a) Först presenteras en kodning där vi följer reglerna från föregående kapitel för att koda if-else-satser. Enligt denna metod skall vi utföra villkorligthopp till else-delen. Vi hoppar till else-delen då  $a \leq b$ , problemet är att denna villkorliga hoppinstruktion ej finns. En omskrivning av testvillkoret behöver göras som leder till att vi kan använda en hoppinstruktion som finns:

$$a - b \leq 0 \Leftrightarrow b - a \geq 0$$

```
.global max

;; int max(int a, int b)
max:
    ;;
    ; if ( a > b )
    CP      R22, R24
    CPC     R23, R25
    BRGE   max_else

max_if:
    ;;
    ; return a;      a finns redan i R25: R24
    RJMP   max_if_end

    ;;
    ; else
max_else:
    ;;
    ; return b;
    MOVW   R24, R22
max_if_end:
    ;;
    }
    RET
```

GNU-kompilatorn ger följande lösning:

```
.text
.global max

;; int max(int a, int b)
```

```

max:
;;  {
;;    if ( a > b )
CP      R22, R24
CPC     R23, R25
BRGE   max_else

;;      return a;
MOVW   R22, R24
;;    else
;;      return b;
max_else:
MOVW   R24, R22
;;  }
RET

```

b) Visa hur ett anrop av max-subrutinen görs på assemblernivå, utgå ifrån följande exempel:

```

;; v = max(10, 20);
LDI    R24, 0x0A //Överföring av 10
LDI    R25, 0x00
LDI    R22, 0x14 //Överföring av 20
LDI    R23, 0x00
CALL   max        //Anrop av max
STS    0x0090, R24 //Tilldelning till v
STS    0x0091, R25

```

c) -

# Index

---

1-komplementkod.....	44
2-komplement .....	45
2-komplementkodning.....	43
3-state-buss .....	32

**A**

absolut-sektionen.....	118
absoluthopp.....	87, 139
ackumulatormaskin .....	69
ADD.....	85
ADD without carry .....	85
Adressbussen.....	11, 14
adresslappar.....	116
adressledningar .....	9
adressrymd.....	73
assembler.....	16
assemblerare.....	16
assemblerprogram.....	16
assemblerspråk.....	16
ATMEL.....	67

**B**

BCD.....	49
big-endian.....	10
binary digit.....	7
BinHex.....	52
binära kvantiter .....	20
binära variabler.....	20
binärfiler .....	52
binärt heltal.....	117
binärt ord.....	9
bit.....	7
Bitordningen .....	10
blanktecknet .....	115
BREQ.....	139
bss-sektionen .....	118
buss.....	11
busskollision .....	32
Bussproblemet.....	32
byte .....	7

**C**

CALL .....	156
CALL to subroutine .....	156
carry.....	46
Central Processing Unit.....	7
centralenhets.....	7

CISC .....	69
CMOS-teknologin .....	31
combinational circuits .....	26
Complex Instruction Set Computer .....	69
control bus .....	11
CPU .....	7
<b>D</b>	
data .....	7
data-sektionen .....	118
Databussen .....	11, 14
Dataledningarna .....	9
DDRx .....	97
decimalt heltal .....	117
digital .....	8
<b>E</b>	
escape-sekvenser .....	51
escapetecken .....	116
<b>F</b>	
funktioner .....	22
<b>G</b>	
GNU as .....	115, 118
GNU ld .....	115
GNU Make .....	126-127
Gränslägesbrytare .....	105
<b>H</b>	
Heltalskonstanter .....	117
hexadecimalt heltal .....	117
hi8 .....	124
högimpedanstillstånd .....	32
<b>I</b>	
I/O-port .....	12
I/O-portar .....	13
IN .....	100
indirekt dataadressering .....	122, 148
inenhet .....	7
inre buss .....	11
instruktion .....	78
instruktionspekarregistret .....	78
<b>J</b>	
JMP .....	87
JuMP .....	87
<b>K</b>	
kombinationskretsar .....	26
Kommentarer .....	115
kompilator .....	15
konstant .....	116

konstanter .....	22
Kontaktstudsar .....	105

**L**

labels .....	116
latch .....	27
LDI .....	81
LDS .....	80
LED .....	104
Light Emitting Diode.....	104
little-endian.....	10
lo8 .....	124
LoaD direct from data Space .....	80
LoaD Immediate.....	81
load/store-arkitektur.....	69
location counter .....	119
Lokala variabler .....	78
lokaliseringräknaren.....	119
lysdioder.....	104
läskretsen .....	27
Länkaren.....	118

**M**

makefile.....	128
maskinberoende syntax.....	115
maskincykel .....	72
maskininstruktion.....	16
maskinoberoende syntax.....	115
Maskinspråk .....	15
memoteknik.....	16
microcontroller .....	70
mikroprogram .....	18
mikrostyrenhet .....	70
minne .....	7, 9
minnesadresseringsregistret.....	11
minnescell .....	9
minnesmapp.....	73
minnesmappad .....	12
minnessiffran .....	46
mnemonics.....	16

**N**

null-terminerad .....	121
numerisk beräkningsmaskin .....	6

**O**

oktalt heltal .....	117
Ord längden .....	9
OUT .....	100
overflow .....	48
Ovillkorliga hopp .....	139

**P**

parallelbuss .....	14
PC-relativt hopp.....	139
perfekt induktion .....	24
PINx.....	97
POP .....	163
POP register from stack .....	163
portar .....	71
PORTx .....	97
positionsoberoende kod.....	139
positionssystem.....	41
primärminne.....	12
program.....	7
program counter .....	78
Programräknaren.....	78
programsats.....	116
programstyrning.....	78
PUD.....	98
PUSH.....	162
PUSH register on stack .....	162

**R**

radkommentar .....	115
RCALL .....	89
Reduced Instruction Set Computer.....	69
reentranta subrutiner .....	162
register preserved by caller.....	137
registerfil .....	76
rekursiva funktioner .....	162
Relative CALL to subroutine .....	89
Relative JuMP .....	88
relocates.....	119
relokerar .....	119
Reläer.....	105
RET .....	90, 158
RETurn from subroutine .....	90, 158
RISC .....	69
RJMP .....	88
runtime-adressen .....	118

**S**

sanningstabell.....	21
scratch pad register.....	137
section.....	118
sektion .....	118
Sekundärminne.....	12
sekvenskrets.....	27
sequential circuit .....	27
seriell buss .....	14
SFIOR.....	98
Skjutströmställare.....	105
slutledning .....	22
Spill.....	48
stack .....	155
statement .....	116

Statusregistret .....	77
STore direct to data Space.....	81
strängkonstant.....	116
Strömställare.....	105
STS.....	81
Styrbussen .....	11, 14
subrutin.....	89, 155
subrutinanrop .....	156
symbol .....	115
symbolisk logik.....	22
symboliska adresser .....	120
 <b>T</b>	
Tangentbordsströmställare .....	105
tecken-belopp-kodning.....	43
teckenkonstant.....	116
temporär datalagring.....	162
text-sektionen .....	118
textfiler .....	52
tillstånd.....	27
transparent .....	29
truth table.....	21
Tryckströmställare.....	105
Tungelement.....	105
 <b>U</b>	
utenhet .....	7
 <b>V</b>	
variabler.....	22
villkorligt hopp.....	139
Vippströmställare.....	105
von Neumann.....	68
von Neumann dator.....	68
 <b>W</b>	
whitespace-tecken .....	115
 <b>Y</b>	
yttre buss.....	11
Yttra enheter .....	12
 <b>A</b>	
åtehoppsadress.....	89
återhoppsadressen.....	160
Återhoppsadresser.....	78