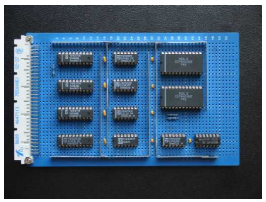
Digital- och datorteknik



Föreläsning #17

Biträdande professor Jan Jonsson

Institutionen för data- och informationsteknik
Chalmers tekniska högskola

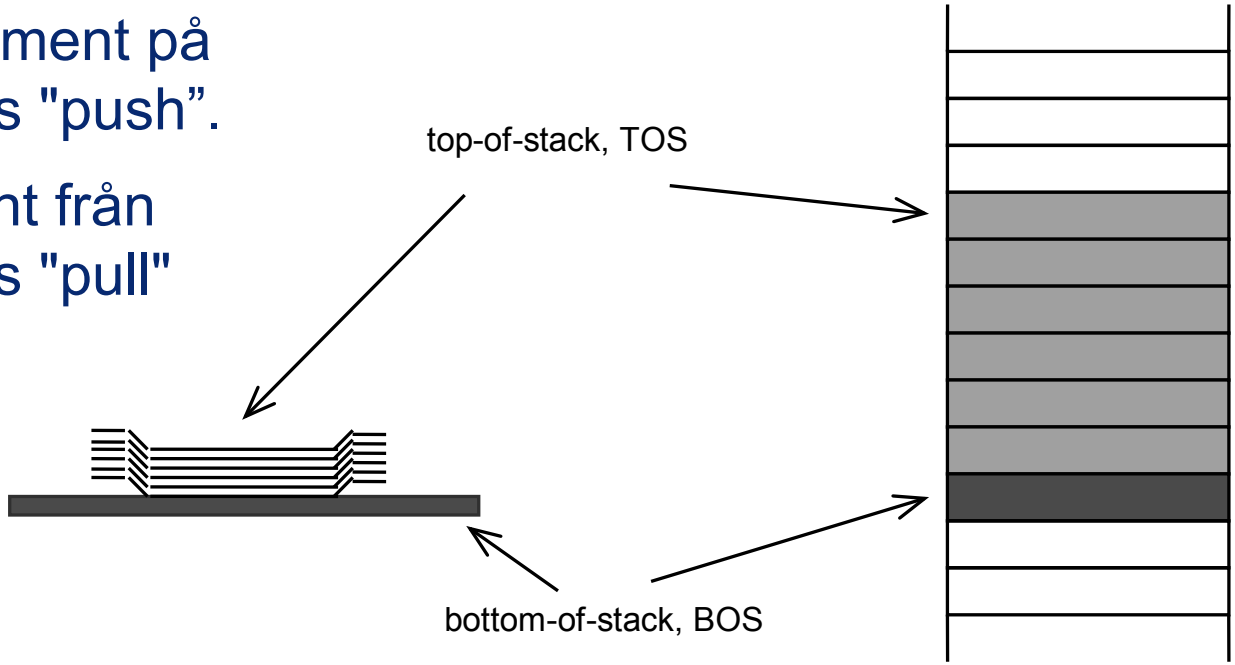
Stackoperationer

”Tallriksmodellen”

Element kan endast läggas till och tas bort ovanifrån, d v s via toppen av stacken. Denna princip för att komma åt elementen kallas "sist in - först ut" (eng. last in - first out, LIFO).

Att lägga nya element på stacken benämns "push”.

Att hämta element från stacken benämns "pull" (ibland "pop").

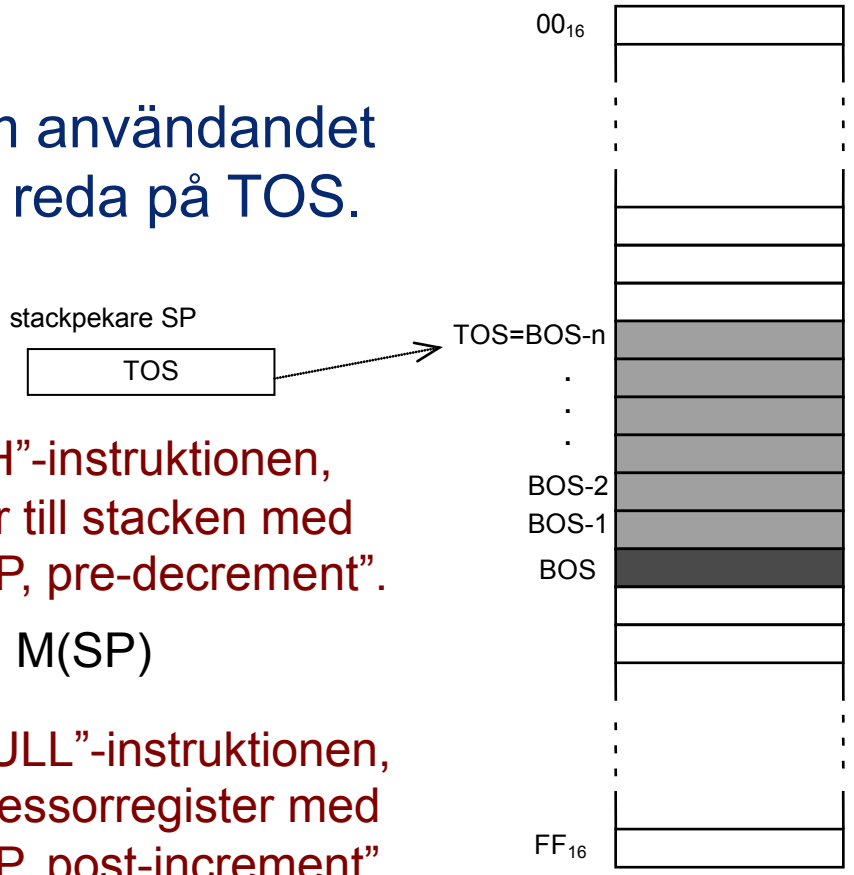


Stackoperationer

”Tallriksmodellen”

Stacken realiseras i datorn genom användandet av en stackpekare SP, som håller reda på TOS.

Stacken initieras genom att SP laddas med BOS.



Ett element läggs på stacken med ”PUSH”-instruktionen, som lagrar data från ett processorregister till stacken med adresseringsmetoden ”register indirect SP, pre-decrement”.

T ex: PSHA SP - 1 → SP, A → M(SP)

Ett element hämtas från stacken med ”PULL”-instruktionen, som hämtar data från stacken till ett processorregister med adresseringsmetoden ”register indirect SP, post-increment”.

T ex: PULX M(SP) → X, SP + 1 → SP

Stackoperationer

Demonstrationsexempel 1 – stackoperationer

a) Ge en sekvens av instruktioner som definierar en stack med början på adress 20_{16} och placerar operanderna 06_{16}, 35_{16} och 28_{16} på stacken.

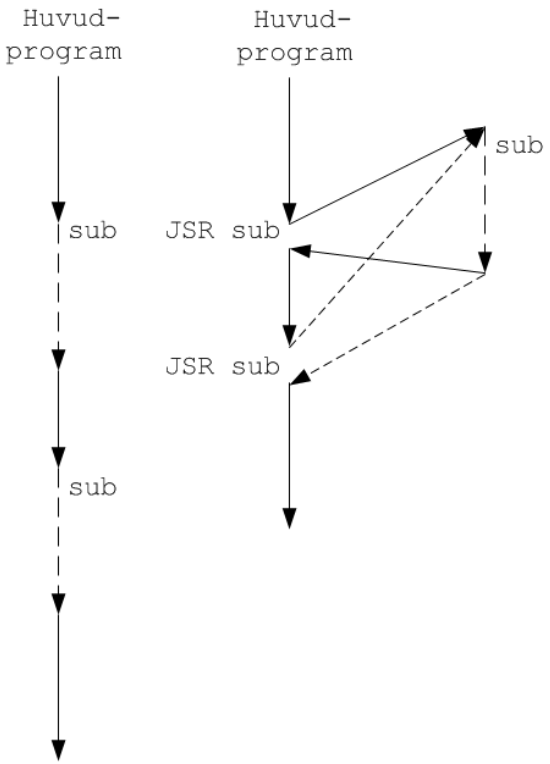
b) Vad är stackpekarens innehåll när sekvensen i a) genomlöpts av processorn?

Modularisering i subrutiner

Behovet av modularisering

I ett välstrukturerat program bör man se till att organisera sin programkod i subrutiner. Fördelarna med detta är:

- Återanvändning: en funktion som behöver användas ofta i olika delar av programmet kan placeras i en modul för att undvika att programmerarna ”återuppfinner hjulet” när de skriver sina individuella programdelar.
- Effektivisering: flera programmerare kan arbeta parallellt och bidra till koden genom att ha ansvar för var sin subrutin.



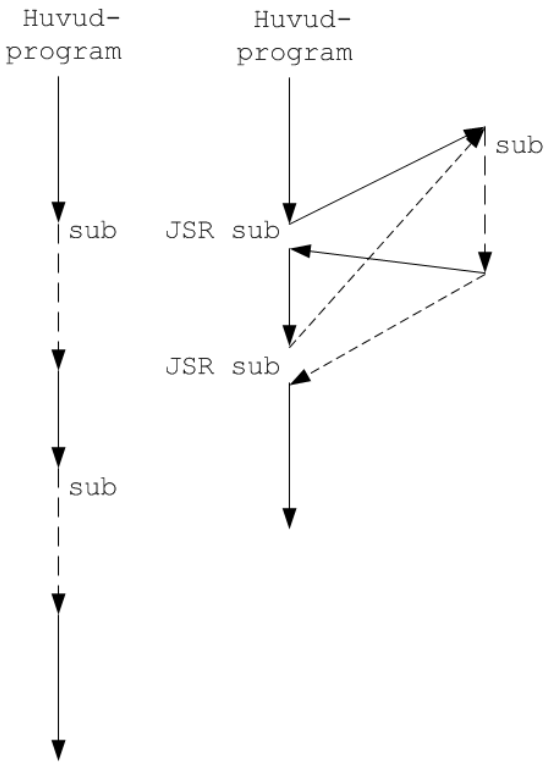
Modularisering i subrutiner

Anrop till subrutin med ”bokmärke”

Vid anrop (hopp) till en subrutin måste det garanteras att programflödet kan återvända till korrekt (anropande) läge i programmet.

Detta kan åstadkommas genom att ett ”bokmärke” sparas på stacken vid anropet. Detta bokmärke utgörs av innehållet i PC vid tidpunkten för anropet till subrutinen.

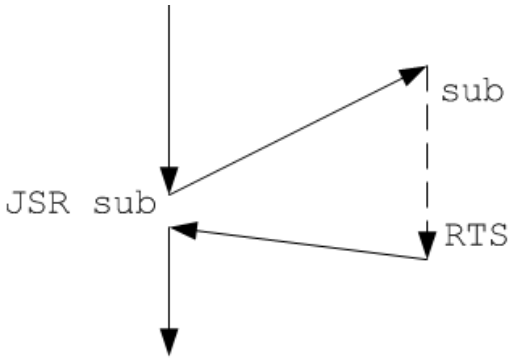
Tack vare stackens egenskaper (LIFO) går det att, inifrån en subrutin, göra ett anrop till ännu en subrutin, och ändå komma tillbaka till det ursprungliga anropet.



Modularisering i subrutiner

Anrop till subrutin med ”bokmärke”

Anrop till subrutin görs med JSR eller BSR (”jump / branch to subroutine”).



JSR Adr SP - 1 → SP, PC → M(SP), Adr → PC

BSR Adr SP - 1 → SP, PC → M(SP), PC + Offset → PC

BSR använder samma adresseringsmetod (PC-relativ adressering) som BRA, BEQ etc

Återhopp från subrutin görs med RTS (”return from subroutine”).

RTS M(SP) → PC, SP + 1 → SP

Modularisering i subrutiner

Tillfällig lagring av data i subrutin

Stackens egenskaper (LIFO) möjliggör inte bara att man kan lagra bokmärken på ett praktiskt sätt; det går också att tillfälligt lagra data på stacken medan subrutinens kod exekveras:

- Innehållet i register vars data inte får förstöras kan sparas (med PUSH-instruktioner) i början av subrutinens kod, och återställas (med PULL-instruktioner) i slutet av koden.

T ex:	PSHA	SP - 1 → SP, A → M(SP)	[spara A]
	PSHCC	SP - 1 → SP, CC → M(SP)	[spara flaggor]
	…		
	PULCC	M(SP) → CC, SP + 1 → SP	[återställ flaggor]
	PULA	M(SP) → A, SP + 1 → SP	[återställ A]

Modularisering i subrutiner

Tillfällig lagring av data i subrutin

Stackens egenskaper (LIFO) möjliggör inte bara att man kan lagra bokmärken på ett praktiskt sätt; det går också att tillfälligt lagra data på stacken medan subrutinens kod exekveras:

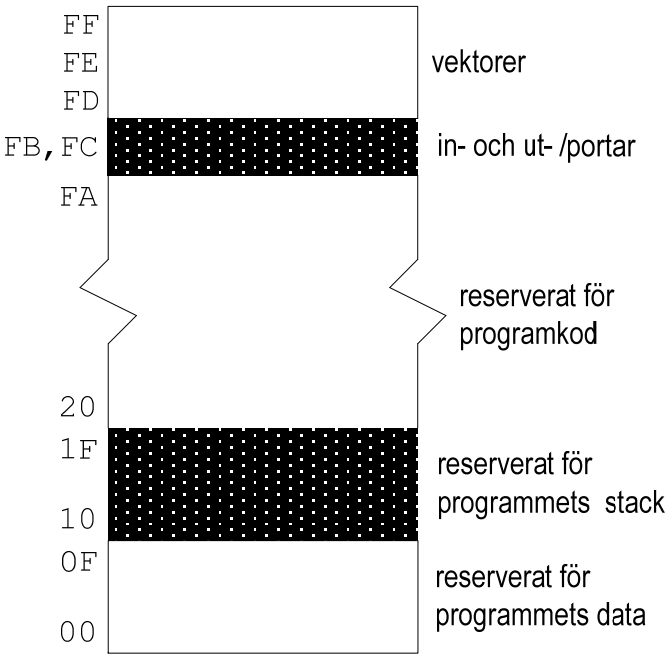
- Extra utrymme för lagring av data kan göras tillgängligt i början av koden respektive tas bort i slutet av koden. Varje byte i detta utrymme kan sedan ses som en tillfällig variabel som adresseras via metoden ’indirekt SP-register med offset’.

```
T ex:   LEASP -8,SP      SP - 8 → SP          [ skapa utrymme ]
        CLR 3,SP         0 → M(SP+3)          [ nollställ en variabel ]
        …
        LDA 0,SP         M(SP)  → A           [ läs annan variabel ]
        LEASP 8,SP       SP + 8 → SP          [ ta bort utrymme ]
```

Assemblerprogrammering

Primärminnets användning

Den rekommenderade användningen av primärminnet i FLIS-processorn är:



- Adress 20_{16} – FA_{16}: Programkod
- Adress 00_{16} – $0F_{16}$: Globala variabler
  (data med relativt lång livslängd)
- Adress 10_{16} – $1F_{16}$: Programstack
  (data med relativt kort livslängd)
- Adress FB_{16} – FC_{16}: In- och ut-portar
  (reserverat för kommunikation med sensor och/eller ställdon)
- Adress FD_{16} – FD_{16}: Vektorer
  (reserverat för lagring av speciella hoppadresser)

Assemblerprogrammering

Assemblerdirektiv – etiketter

Vi har ju sett att när man skriver program i assemblerspråk har man ersatt maskininstruktionernas binära representation med symboliska namn på instruktionerna. T ex skriver vi LDA #7, istället för den två byte långa sekvensen F0 07, när vi vill ange den instruktion som laddar register A med konstanten 7.

Vi skulle på ett liknande sätt vilja ersätta lägen (minnesadresser) i primärminnet med symboliska namn. Detta låter sig göras i assemblerspråk om man sätter en etikett (”label”) framför den instruktion eller databyte vars adress vi vill kunna referera till.

```
Exempel:        Loop    DECA
                        BNE  Loop
```

Assemblerprogrammering

Assemblerdirektiv – startadress

För att en etikett skall kunna användas i programmet måste dess läge (d v s minnesadress) kunna beräknas. Detta är möjligt om det är känt var i minnet programkod och data är placerade.

Med hjälp av assemblerdirektivet ORG har programmeraren möjlighet att ange en startadress för kod eller data.

Exempel:

```
        ORG $20
Start   LDSP #$20      ; etiketten 'Start' får värdet 20(16)
        LDA #7
        BRA  Skip
        NOP
Skip    INCA           ; etiketten 'Skip' får värdet 27(16)
```

Assemblerprogrammering

Assemblerdirektiv – initierade data

Förutom programkod är det också möjligt att lägga in datavärden i minnet när man skriver i assemblerspråk. Dessa värden kommer att finnas på det angivna läget då programmet startar, och kallas därför initierade data.

Med hjälp av assemblerdirektivet FCB (form constant byte) kan en eller flera bytes av kända data läggas på en angiven plats i minnet.

Exempel:

```
        ORG $20
Start   LDSP #$20       ; etiketten 'Start' får värdet 20_16
        …
        ORG $FF
        FCB  Start      ; värdet 20_16 läggs i resetvektor
```

Assemblerprogrammering

Assemblerdirektiv – initierade data

Förutom programkod är det också möjligt att lägga in datavärden i minnet när man skriver i assemblerspråk. Dessa värden kommer att finnas på det angivna läget då programmet startar, och kallas därför initierade data.

Med hjälp av assemblerdirektivet FCS (form constant string) kan en sträng med ASCII-tecken läggas på en angiven plats i minnet.

Exempel:

```
        LDX #Hello      ; etiketten 'Hello' har värdet 60(16)
        LDA 0,X
        …
        ORG $60
Hello   FCS  "Hello world"   ; sträng med 11 ASCII-tecken
```

Assemblerprogrammering

Assemblerdirektiv – icke initierade data

Det är också möjligt att i förväg reservera utrymme i minnet. Minnesinnehållet i detta utrymme är odefinierat då programmet startar, och är huvudsakligen tänkt att användas för lagring av globala variabler (data med relativt lång livslängd).

Med hjälp av direktivet RMB (reserve memory bytes) kan utrymme för en eller flera bytes reserveras på en angiven plats i minnet.

```
Exempel:            ORG $0
          Count     RMB 1       ; plats för räknarvariabel
                                ; etiketten 'Count' får värdet 00_16
          List      RMB 8       ; plats för lista med 8 element
                                ; etiketten 'List' får värdet 01_16
```

Assemblerprogrammering

Assemblerdirektiv – övrigt

- Egendefinierade symboler: definieras med direktivet EQU

  T ex: Max_val EQU 37
  Min_val EQU 12
  Range EQU (Max_val–Min_val)

- Hexadecimal konstant: talet föregås av tecknet $.

  T ex: $40 = $40_{16} = 64_{10}$

- Binär konstant: talet föregås av tecknet %.

  T ex: %1011 = $1011_2 = B_{16} = 11_{10}$

- Teckenkonstant: tecknet omges av enkla citationstecken.

  T ex: ’A’ = $41_{16} = 65_{10}$ = ASCII-värdet för stora A

Assemblerprogrammering

Demonstrationsexempel 2 – subrutiner

Skriv en subrutin AddAY som adderar två 8-bitars tal utan tecken. De två talen skall finnas i Y- resp. A-registret vid anrop av subrutinen. Mest signifikanta byten skall returneras i Y-registret och minst signifikanta byten i A-registret.

Av de interna registren får subrutinen endast påverka Y- och A-registren samt flaggregistret. Z-flaggan skall sättas till 1 om summan är lika med noll. De övriga flaggornas värden saknar betydelse.

Subrutinens kod skall placeras med början på adress 50_{16}.

Visa även hur stackens innehåll ser ut om subrutinen anropas från ett huvudprogram vars kod börjar på adress 30_{16}. Top-of-stack (TOS) sätts till 20_{16},