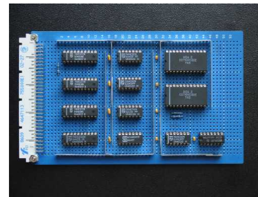
CHALMERS
UNIVERSITY OF TECHNOLOGY

Digital- och datorteknik



Föreläsning #16

Biträdande professor Jan Jonsson

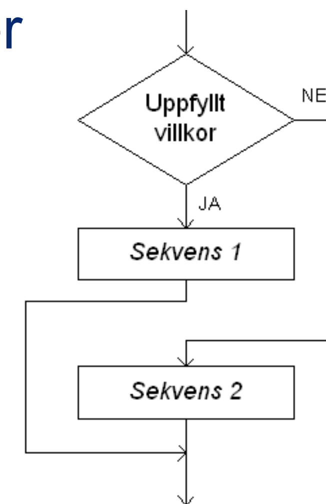
Institutionen för data- och informationsteknik
Chalmers tekniska högskola

Kontroll av programflöde

Behovet av ändring av programflödet

För att kunna skriva ett avancerat datorprogram som anpassar sitt beteende utgående från ett beräkningsresultat eller på grund av interaktion med datoranvändaren måste datorn tillhandahålla instruktioner som möjliggör ändring av programflödet.

Med möjlighet till val i programflödet kan datorn t ex utföra en av två möjliga sekvenser av maskininstruktioner beroende på om ett visst logiskt villkor är uppfyllt eller ej.



För att konstruera sådana val i ett program behövs stöd i datorn för både ovillkorliga och villkorliga programflödesändringar ("hopp").

Kontroll av programflöde

Ovillkorliga hoppinstruktioner

’Jump’-instruktionen

Hoppdestinationens läge i primärminnet (effektivadressen, EA) anges av instruktionens operandinfo.

T ex: JMP $58 $58_{16} \rightarrow PC$

’Branch always’-instruktionen

Hoppdestinationens läge i primärminnet (EA) utgörs av PC-registrets nuvarande värde plus en offset (i 2-komplementform). Värdet på ’offset’ anges av operandinfo.

T ex: BRA $58 $PC + (58_{16} – PC) \rightarrow PC$

’offset’ = operandinfo

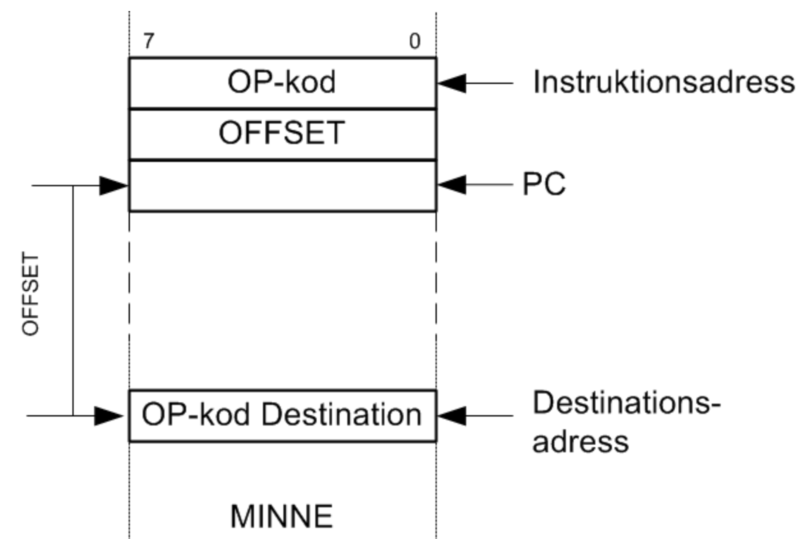
Kontroll av programflöde

Ovillkorliga hoppinstruktioner

’Branch always’-instruktionen (forts.)

För beräkning av ’offset’ gäller följande:

- Instruktionens operationskod och operandinfo förutsätts ha hämtats när EA beräknas i datorn. Därför måste PC antas peka på den efterföljande instruktionens operationskod när värdet på ’offset’ beräknas.
- Instruktionens blivande läge i primärminnet måste vara känt när värdet på ’offset’ beräknas.



Kontroll av programflöde

Demonstrationsexempel #1 – ovillkorliga hopp

a) "Jump"-instruktionen JMP $50 är placerad med början på adress 37_{16}.
Vad är dess maskinkod?

b) "Branch"-instruktionen BRA $50 är placerad med början på adress 37_{16}.
Vad är dess maskinkod?

Kontroll av programflöde

Demonstrationsexempel #2 – ovillkorliga hopp

a) "Jump"-instruktionen JMP $05 är placerad med början på adress 20_{16}.
Vad är dess maskinkod?

b) "Branch"-instruktionen BRA $05 är placerad med början på adress 20_{16}.
Vad är dess maskinkod?

Kontroll av programflöde

Villkorliga hoppinstruktioner

De villkor som styr datorns programflöde kan vara godtyckligt komplicerade. Den automatiska styrenheten, som skall utföra programflödesändringen, kan inte själv göra de beräkningar som behövs för att avgöra om ett villkor är uppfyllt eller ej.

Styrenheten har bara tillgång till datavägens flaggbitar, och det förutsätts därför att dessa redan har manipulerats på ett sådant sätt att de representerar det villkor som styr programflödet.

Exempel: om programflödet skall ändras när addition av två tal med tecken ger spill måste V-flaggan ha satts till 1 vid additionsinstruktionen.

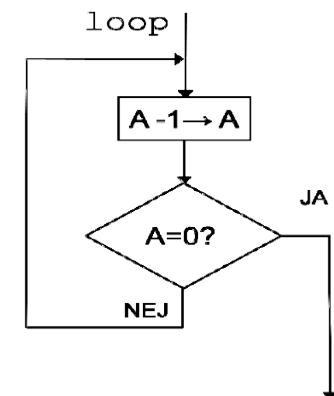
Exempel: om programflödet skall ändras när värdet på ett tal är skilt från 0 måste Z-flaggan ha nollställts av någon lämplig instruktion.

Kontroll av programflöde

Villkorliga hoppinstruktioner

De villkorliga hoppinstruktionerna har stora likheter med den ovillkorliga ’branch always’-instruktionen (BRA), i den mening att samtliga använder PC-relativ adressering.

Skillnaden gentemot BRA-instruktionen är att de villkorliga instruktionerna enbart sätter PC till hoppdestinationen (EA) om ett givet villkor är uppfyllt. I annat fall kommer PC att peka på efterföljande instruktion.



T ex: BNE \$23 if (Z = 0) PC + (23_{16} – PC) → PC

Kontroll av programflöde

Villkorliga hopp med enkla villkor

Följande ’branch’-instruktioner utför en ändring av programflödet utgående från värdet på en given flaggbit:

Instruktion	Villkor	
BMI	N	“Branch if minus”
BPL	N’	“Branch if plus”
BEQ	Z	“Branch if equal”
BNE	Z’	“Branch if not equal”
BCS	C	“Branch if carry set”
BCC	C’	“Branch if carry clear”
BVS	V	“Branch if overflow set”
BVC	V’	“Branch if overflow clear”

Kontroll av programflöde

Villkorliga hopp med jämförelsevillkor

En vanligt förekommande situation vid val i programflödet är att man jämför två tal med varandra, och gör en ändring av programflödet beroende på talens inbördes relation. Detta kan exempelvis ske när man vill kontrollera huruvida ett värde ligger inom ett givet intervall eller när man sorterar en lista med tal.

I FLIS-datorn utförs jämförelsen med en CMP-instruktion, som beräknar differensen mellan de två talen och sätter flaggbitarna utgående från resultatet. Den erhållna differensen sparas ej.

T ex: CMPA #6 *jämför innehållet i register A med konstanten* 6_{10}

CMPX $93 *jämför innehållet i register X med innehållet i minnescellen på adress* 93_{16}

Kontroll av programflöde

Villkorliga hopp med jämförelsevillkor

De grundläggande utfall som kan erhållas från CMP-instruktionen är större än (>), lika med (=) och mindre än (<). Vi kan också få de logiska komplementen (≤, ≠, ≥) till dessa utfall.

Frågan är nu vilka flaggbitar som skall undersökas för att korrekt representera respektive villkor?

Utfallet ’lika med’ (=) motsvarar villkoret Z. Dess komplement ’inte lika med (≠) motsvarar villkoret Z’.

De övriga utfallen är lite mer komplicerade att analysera, då de flaggbitar som skall undersökas beror på vilken talrepresentation som gäller. Precis som vid vår tidigare analys av ALU-aritmetik behöver vi beakta två fall: tal utan tecken och tal med tecken.

Kontroll av programflöde

Villkorliga hopp med jämförelsevillkor (tal utan tecken)

Antag att talet X jämförs med talet Y medelst subtraktionen X – Y.

Vid utfallet ’mindre än’ (<) måste subtraktionen ha gett spill (C=1), då en negativ differens inte kan representeras för tal utan tecken.

Utfallet ’mindre än’ (<) motsvarar alltså villkoret C.

Vi kan nu se att utfallet ’mindre eller lika med’ (≤) fås när antingen C=1 eller Z=1, d v s utfallet motsvarar villkoret C + Z.

Utfallet ’större än’ (>), som är det logiska komplementet till ’mindre eller lika med’, motsvarar därför villkoret (C + Z)’.

Utfallet ’större eller lika med’ (≥), som ju är det logiska komplementet till ’mindre än’, motsvarar villkoret C’.

Kontroll av programflöde

Villkorliga hopp med jämförelsevillkor (tal <u>med</u> tecken)

Antag att talet X jämförs med talet Y medelst subtraktionen X – Y.

Vid utfallet ’mindre än’ (<) måste resultatet har blivit negativt (N=1), givet att spill inte uppstått (V=0). Om subtraktionen gett spill (V=1) måste resultatet istället ha blivit positivt (N=0).

Utfallet ’mindre än’ (<) motsvarar alltså villkoret $N \cdot V' + N' \cdot V = N \oplus V$.

Utfallet ’mindre eller lika med’ (≤) motsvarar villkoret $(N \oplus V) + Z$.

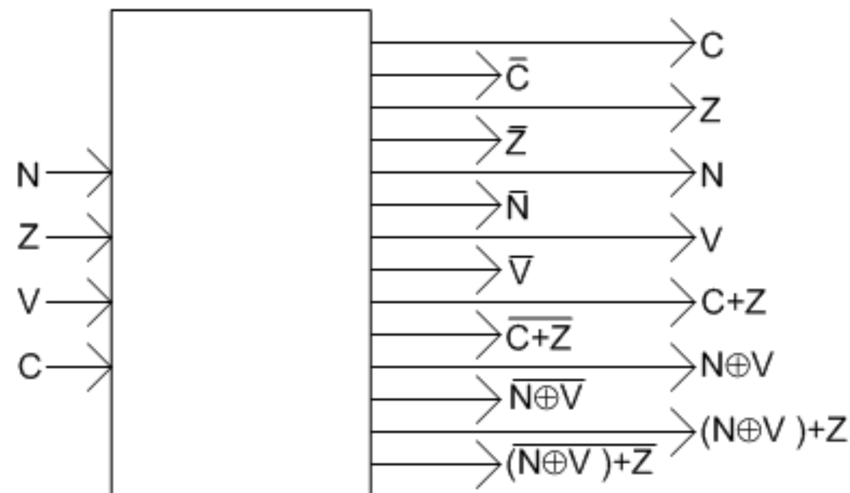
Utfallet ’större än’ (>), det logiska komplementet till ’mindre eller lika med’, motsvarar villkoret $((N \oplus V) + Z)'$.

Utfallet ’större eller lika med’ (≥), det logiska komplementet till ’mindre än’, motsvarar villkoret $(N \oplus V)'$.

Kontroll av programflöde

Villkorliga hopp med jämförelsevillkor

Vi har därmed fått en förklaring till varför nedanstående avkodning av flaggbitarna görs i styrenheten.



Dessa villkorssignaler finns tillgängliga i Digiflisps ”Instruction builder” och på kopplingsplattan i Lab 3, för att möjliggöra implementering av villkorliga hopp.

Kontroll av programflöde

Villkorliga hopp med jämförelsevillkor (sammanställning)

Hoppvillkor för villkorliga hopp för tal utan respektive med tecken.

Notera terminologin som används för instruktionsnamnen.

	Hoppvillkor utan tecken		Relation	Hoppvillkor med tecken		
"higher"	BHI	(C + Z)'	X > Y	((N ⊕ V) + Z)'	BGT	"greater"
"higher or same"	BHS	C'	X ≥ Y	(N ⊕ V)'	BGE	"greater or equal"
"equal"	BEQ	Z	X = Y	Z	BEQ	"equal"
"not equal"	BNE	Z'	X ≠ Y	Z'	BNE	"not equal"
"lower or same"	BLS	C + Z	X ≤ Y	(N ⊕ V) + Z	BLE	"less or equal"
"lower"	BLO	C	X < Y	N ⊕ V	BLT	"less than"

Kontroll av programflöde

Demonstrationsexempel #3 – villkorliga hopp

För vilka värden på W utförs hoppet nedan?

```
LDA         #$85
CMPA        #W
B(Villkor)  Hopp
```

om det villkorliga hoppet är:

a) BHI
b) BHS
c) BLS
d) BLO
e) BGT
f) BGE
g) BLE
h) BLT

Stackoperationer

Behovet av tillfällig datalagring

I många sammanhang kan man som programmerare hamna i en situation där man har behov av tillfällig lagring av data, men man har inte tillräckligt många register till förfogade i processorn.

Det är därför lämpligt att reservera en del av primärminnet för sådan tillfällig datalagring. Denna del av minnet bör inte vara för stor för att inte äta upp utrymme för lagring av program, men får samtidigt inte vara för snålt tilltagen så att man inte kan göra den temporära lagring som programmet behöver.

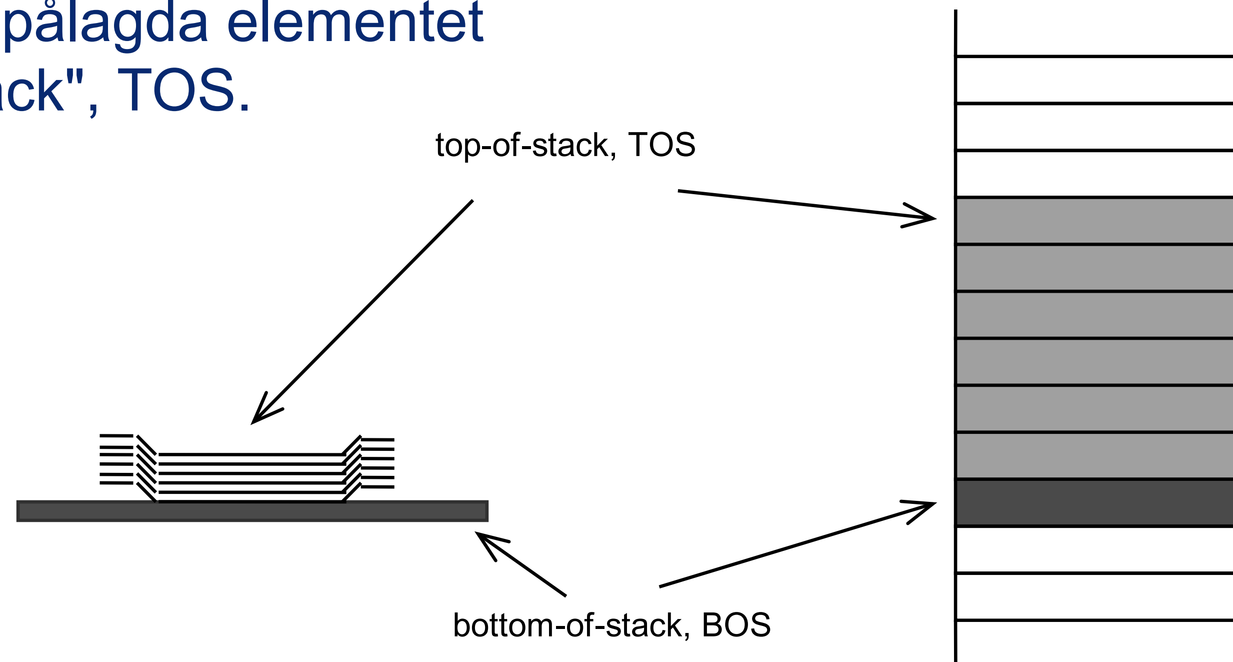
En annan viktig aspekt är att programmeraren på ett enkelt sätt skall kunna lägga till och ta bort data i denna del av minnet.

Detta kan lösas med en datastruktur, kallad ”stack” (stapel).

Stackoperationer

”Tallriksmodellen”

På en stack lagras binära ord på precis samma sätt som man staplar tallrikar på en hylla, d v s bildligt talat ovanpå varandra. Basläget kallas "bottom-of-stack", BOS och läget för det sist pålagda elementet kallas "top-of-stack", TOS.



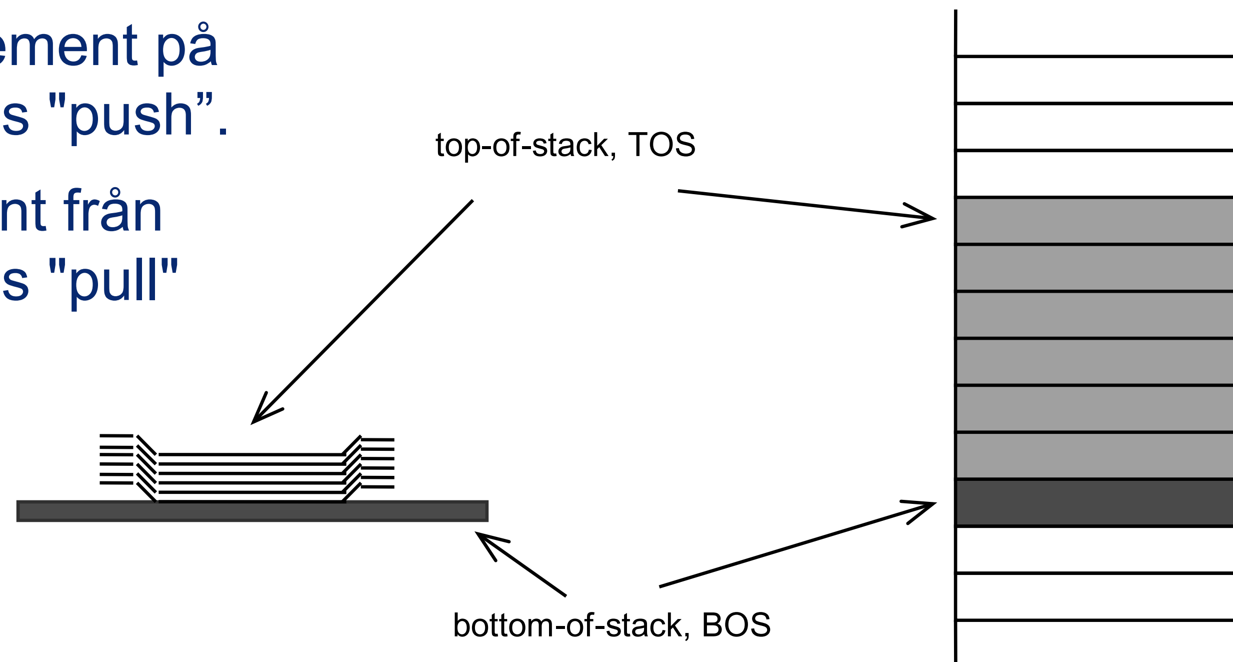
Stackoperationer

”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").

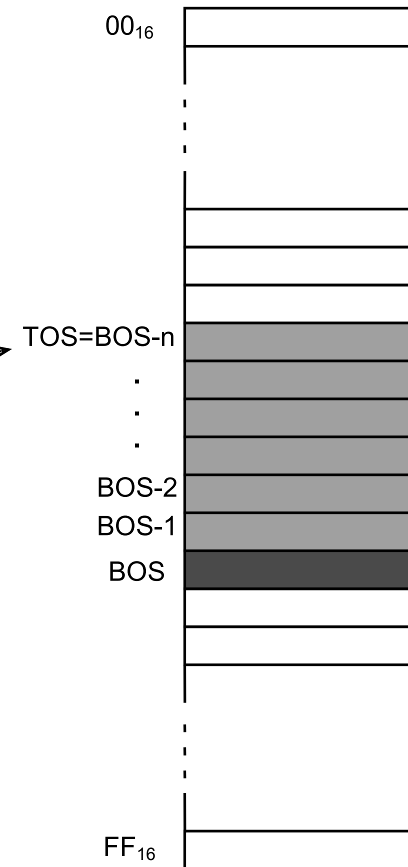


Stackoperationer

”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

Användningsområden

Stacken används huvudsakligen i följande situationer:

- Vid anrop till subrutin: spara automatiskt en kopia av PC-registrets nuvarande innehåll (ett ”bokmärke”) på TOS.

  Anrop till subrutin görs med JSR eller BSR

  Återhopp från subrutin (till ”bokmärke”) görs med RTS

- Under anrop till subrutin: skapa utrymme för lokala variabler och spara kopior av register som manipuleras i subrutinen.

  Minnesutrymme för variabler skapas / tas bort med LEASP n,SP

  Registerinnehåll sparas / återställs med PUSH / PULL