ME – en dator

1 Inledning

ME är en påhittad dator, men den har likheter med riktiga datorer: det finns ett maskinspråk med olika instruktioner, det finns minne och register, och så vidare. Vi kommer att skriva program i datorns assemblerspråk. Instruktionerna i assemblerspråket motsvarar direkt maskininstruktioner men skrivs med symboliska namn i stället för med nollor och ettor.

Till sist beskrivs en emulator[1] för assemblerspråket i datorn.

2 ME-datorn

2.1 Datorns uppbyggnad och assemblerspråk

ME-datorn har ett minne om 1000 minnesceller, numrerade 0–999. Datorn har alltså en ”adressrymd” om 1000 minnesceller — jämför detta med en persondator som har några miljarder minnesceller. I varje minnescell kan man lagra ett heltal, positivt eller negativt — jämför detta med en persondator där man förutom heltal också kan lagra flyttal (reella tal).

Förutom i minnet kan man lagra data i register. Det finns fem register: `R1`, `R2`, `R3`, `R4` och `R5`. Varje register kan innehålla ett positivt eller negativt heltal. Anledningen till att man har register är att det går snabbare att komma åt ett tal i ett register än att hämta det från minnet. När man gör beräkningar brukar man därför ha operanderna till beräkningen i register, även om detta inte är nödvändigt i ME-datorn.

ME-datorn har instruktioner för aritmetik (de fyra räknesätten) och för villkorliga och ovillkorliga hopp. En stoppinstruktion och instruktioner för inläsning och utskrift finns också.

Ett program i datorns assemblerspråk består av ett antal satser. En sats består av en instruktion och noll till tre parametrar. Parametrarna anger var indata till instruktionen ska hämtas eller var resultatet av instruktionen ska lagras. Exempel på en sats i språket:

```
add r1,100,m(356) ! betyder: addera innehållet i register r1
                  ! och talet 100, lagra resultatet i
                  ! minnescellen med adressen 356
```

Här är instruktionen `add`, som alltid ska ha tre parametrar. De båda första parametrarna adderas och resultatet lagras i den tredje parametern. I exemplet är den första parametern ett register, den andra parametern en konstant och den tredje parametern en minnescell. Det finns fyra olika typer av parametrar:

1. Ett konstant värde: det angivna värdet används direkt som parameter. Man kan naturligtvis inte använda denna parametertyp som resultatparameter.
2. Register: ett av registren `R1`, `R2`, `R3`, `R4` eller `R5`.
3. Minne: en av minnescellerna. Skrivs `M(adress)`, till exempel `M(356)`.

[1] Emulera betyder efterlikna. En definition av ordet emulator: A device, computer program, or system that accepts the same inputs and produces the same outputs as a given system.

`move`	`p1,p2`	Kopiera `p1` till `p2`.
`add`	`p1,p2,p3`	Addera `p1` och `p2`, lägg resultatet i `p3`.
`sub`	`p1,p2,p3`	Subtrahera `p2` från `p1`, lägg resultatet i `p3`.
`mul`	`p1,p2,p3`	Multiplicera `p1` och `p2`, lägg resultatet i `p3`.
`div`	`p1,p2,p3`	Dividera `p1` med `p2`, lägg resultatet i `p3` (heltalsdivision, resten kastas bort).
`jump`	`lab`	Hoppa till `lab`, som ska vara ett läge.
`jpos`	`p1,lab`	Hoppa till lab om `p1` $>= 0$, annars fortsätt exekveringen med nästa sats.
`jneg`	`p1,lab`	Hoppa till lab om `p1` < 0, annars fortsätt.
`jz`	`p1,lab`	Hoppa till `lab` om `p1` $= 0$, annars fortsätt.
`jnz`	`p1,lab`	Hoppa till `lab` om `p1` $\neq 0$, annars fortsätt.
`read`	`p1`	Läs in ett talvärde, lagra det i `p1` (detta är egentligen ett anrop till operativsystemet).
`print`	`p1`	Skriv ut `p1` (också detta är ett anrop till operativsystemet).
`stop`		Avsluta exekveringen.

Tabell 1: Instruktioner i ME-datorn.

4. Minne indirekt: den minnescell vars adress finns i register `R1`, `R2`, `R3`, `R4` eller `R5`. Skrivs `M(register)`, till exempel `M(R1)`. Denna parametertyp förklaras utförligare i avsnitt 2.2.

Det finns en femte parametertyp som kallas *läge* (engelska label, ”etikett”) och som bara används i samband med hoppinstruktioner. Ett läge är ett namn efterföljt av kolon som man skriver först på en programrad. I en hoppinstruktion är läget ”målet” för hoppet. Exempel:

```
back:   sub  r1,1,r1
        ...
        jump back
```

I tabell 1 visas alla olika instruktioner som man kan använda i ME-datorn. `p1`, `p2` och `p3` är parametrar som kan vara av godtycklig typ.

ME-datorn skiljer sig i en del avseenden från många moderna datorer, som är så kallade RISC-datorer (Reduced Instruction Set Computer). En sådan dator har man försökt göra så enkel som möjligt genom att hålla nere antalet olika instruktioner och genom att göra de instruktioner som finns så enkla som möjligt. Till exempel kanske det i en RISC-dator inte finns multiplikations- och divisionsinstruktioner, eftersom dessa operationer kan byggas upp med hjälp av andra instruktioner. Men framförallt är RISC-datorer inte så generella som ME-datorn när det gäller parametrar. Till exempel kräver en `add`-instruktion i en RISC-dator normalt att alla parametrarna är register.

Med hjälp av de beskrivna instruktionerna kan man skriva både enkla och mer komplicerade program. Vi visar några exempel på program, där vi först skriver Javasatser och därefter motsvarande ME-satser. I alla exemplen utnyttjar vi i Javasatserna tre heltalsvariabler `x`, `y` och `z`. I ME-programmen lagras dessa variabler i minnescellerna `M(0)`, `M(1)` respektive `M(2)`.

Först visar vi ett exempel som bara innehåller tilldelningssatser och beräkning av ett aritmetiskt uttryck:

```
x = 10;
y = 20;
z = 2 * (x + 1) - 3 * y;
```

```
move 10,m(0)    ! x = 10
move 20,m(1)    ! y = 20
add  m(0),1,r1  ! r1 = x + 1
mul  2,r1,r1    ! r1 = 2 * r1
mul  3,m(1),r2  ! r2 = 3 * y
sub  r1,r2,m(2) ! z = r1 - r2
```

De båda första Javasatserna motsvaras av var sin ME-sats. I den tredje Javasatsen finns en beräkning av ett aritmetiskt uttryck. Denna beräkning har vi varit tvungna att bryta ner i mindre

delar som kan uttryckas med ME-satser. Vi använder register för att lagra mellanresultat. När man har ett program i högnivåspråk, till exempel Java, så är det kompilatorn som bryter ner komplicerade programstrukturer till satser i maskinspråket.

Man kan också uttrycka mera komplicerade programstrukturer med hjälp av ME-satser, till exempel `if`-satser:

```
x = scan.nextInt();
y = scan.nextInt();
if (x >= 0) {
    x = x + 1;
}
if (x > y) {
    z = x;
} else {
    z = y;
}
System.out.println(z);
```

```
      read  m(0)          ! läs in x
      read  m(1)          ! läs in y
      jneg  m(0),neg      ! hoppa till neg om x < 0
      add   m(0),1,m(0)   ! x = x + 1
neg:  sub   m(1),m(0),r1  ! r1 = y - x
      jpos  r1,else       ! hoppa om x <= y
      move  m(0),m(2)     ! z = x;
      jump  comm          ! hoppa till comm
else: move  m(1),m(2)     ! z = y
comm: print m(2)          ! skriv ut z
```

Notera att vi har varit tvungna att formulera om villkoren för att kunna uttrycka dem med ME-instruktioner. I satsen `if (x > y) ...` börjar vi med att räkna ut `y - x` och utför `else`-grenen om `y - x` är < 0, det vill säga om `x` är $\leq$ `y`.

Till sist ska vi visa ett exempel på hur en `while`-sats i Java skrivs med hjälp av ME-satser. En `for`-sats kan ju uttryckas med hjälp av en `while`-sats, så en `for`-sats skriver man på ungefär samma sätt. Här beräknar vi summan $y = 1 + 2 + 3 + \ldots + 99$:

```
x = 1;
y = 0;
while (x < 100) {
    y = y + x;
    x = x + 1;
}
System.out.println(y);
```

```
      move  1,m(0)          ! x = 1
      move  0,m(1)          ! y = 0
loop: sub   m(0),100,r3     ! r3 = x - 100
      jpos  r3,lpend        ! hoppa om x >= 100
      add   m(1),m(0),m(1)  ! y = y + x
      add   m(0),1,m(0)     ! x = x + 1
      jump  loop            ! hoppa till loop
lpend: print m(1)           ! skriv ut y
```

När man har sett dessa exempel och tänkt sig in i hur arbetsamt det skulle vara att översätta ett större Javaprogram till ME-satser eller till någon annan dators assemblerspråk så uppskattar man antagligen att man normalt inte skriver program i sådana språk. De enda tillfällen då man själv måste skriva program i assemblerspråk är när man måste ha mycket nära kontakt med datorns hårdvara, till exempel i de inre delarna i ett operativsystem.

Några avslutande anmärkningar om ME-datorns assemblerspråk:

- Man skiljer inte på små och stora bokstäver i program. En `add`-instruktion kan till exempel skrivas `add`, `ADD`, `Add`, ...
- En kommentar inleds med `!` och sträcker sig till slutet av raden (motsvarar Javas `//`).
- Man kan stoppa in blanka rader var som helst i ett program.

2.2 Indirekt adressering – överkurs

I föregående avsnitt fanns inga exempel där vi utnyttjade den fjärde parametertypen, ”minne indirekt”. Detta beror inte på att denna parametertyp skulle vara oviktig — tvärtom är det nog så att den är den parametertyp som förekommer oftast i program som produceras av kompilatorer för högnivåspråk. Det beror på att minnet i sådana program inte kan adresseras statiskt med fixa adresser som `M(0)` eller `M(1)`, eftersom kompilatorn inte kan förutsäga var en struktur kommer att finnas i minnet.

Vi tar följande Javaklass som exempel:

```
class A {
    private int x;
    public A() {
        x = 0;
    }
    public void increment() {
        x = x + 1;
    }
}
```

Satsen `x = x + 1` ska översättas till en `add`-instruktion: `add ?,1,?` där frågetecknen anger adressen till `x` i minnet. Men man kan ju skapa flera `A`-objekt som hamnar på olika platser i minnet, som vi gör i följande satser:

```
A pa1 = new A();
pa1.increment();
A pa2 = new A();
pa2.increment();
```

I minnet kan det ut så här efter satsen `A pa2 = new A()` (detta är en förenklad bild):

0	920	pa1
1	921	pa2
2	0	
3	0	
...	...	
920	1	A-objekt
921	0	A-objekt

Vi ser att `pa1.x` här har adressen `920`, `pa2.x` har adressen `921`. Vi kan nu lösa problemet att adressera `x` i de olika objekten om vi ser till att vi, när vi kommer till metoden `increment`, alltid har adressen till det aktuella objektet (`this`-referensen) i ett bestämt register. Om vi antar att detta register är `R3` kan additionsinstruktionen skrivas:

```
add m(r3),1,m(r3) ! r3 innehåller en adress. Addera 1 till
                  ! minnescellen på den adressen
```

För att utföra operationen `increment` på ett objekt laddar man in adressen till objektet, till exempel `pa1` eller `pa2`, i `R3` och anropar metoden. Hur man utför metodanrop beskriver vi i avsnitt 2.3.

Ett annat tillfälle då man måste utnyttja indirekt adressering är när man hanterar vektorer. Detta gäller även om en vektor är placerad på en bestämd plats i minnet: om man har en vektor `v` på en bestämd plats så känner man adresserna till alla elementen och kan därmed direkt adressera `v[i]` om `i` är en konstant. Men om `i` är en variabel så måste man beräkna `i`-värdet, addera det till vektorns startadress och lägga summan i ett register och till sist utnyttja indirekt adressering för att komma åt `v[i]`.

Vi visar hur det kan gå till i ett exempel:

```
int[] v = new int[10];
for (int i = 0; i < 10; i++) {
    v[i] = i * i;
}
int sum = 0;
for (int i = 0; i < 10; i++) {
    sum = sum + v[i];
}
System.out.println(sum);
```

Vi förutsätter här att vektorn `v` hamnar i minnet på adress `25` och framåt. Variabeln `i` lagrar vi i register `R1`, variabeln `sum` i `R3`.

```
        move  0,r1         ! i = 0
sqr:    sub   r1,10,r2     ! r2 = i - 10
        jpos  r2,sqend     ! hoppa till sqend om i - 10 >= 0 dvs om i >= 10
        add   25,r1,r2     ! r2 = adressen till v[i]
        mul   r1,r1,m(r2)  ! v[i] = i * i
        add   r1,1,r1      ! i++
        jump  sqr
sqend:  move  0,r3         ! sum = 0
        move  0,r1         ! i = 0
add:    sub   r1,10,r2
        jpos  r2,aend
        add   25,r1,r2
        add   r3,m(r2),r3  ! sum = sum + v[i]
        add   r1,1,r1      ! i++
        jump  add
aend:   print r3           ! skriv ut sum
```

2.3 Subrutiner – överkurs

En subrutin är ett programavsnitt som man kan hoppa till från olika platser i ett program. När man återvänder från en subrutin kommer man tillbaka till den plats varifrån man hoppade. När man hoppar till en subrutin säger man att man *anropar* subrutinen. Subrutiner kan ha *parametrar* dvs värden som man tar med sig till subrutinen när man anropar den. Subrutiner finns i de flesta programspråk även om benämningen kan variera mellan olika språk: metod, funktion, procedur, ...

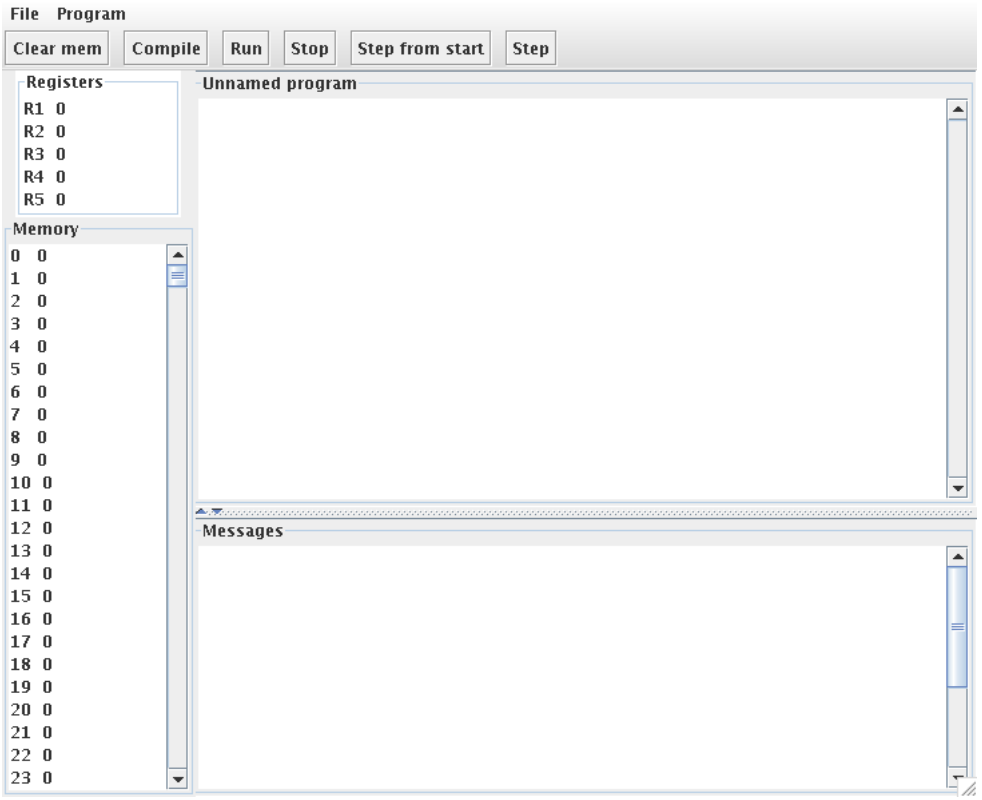
När det gäller subrutiner finns det tre problem som måste lösas:

1. Att hoppa till subrutinen. Detta är inte svårt; det gör man med en `jump`-instruktion. Problemet blir att återvända från subrutinen; se punkt 3.
2. Att bestämma hur parametrarna ska föras över till subrutinen. Detta kan göras på olika sätt; här förutsätter vi att parametrarna lagras i register `R2`, `R3` och `R4`. Man får alltså ha högst tre parametrar till en subrutin. Om subrutinen är en funktion så förutsätter vi att resultatet av funktionen förs tillbaka i register `R1`.
3. Att återvända till den plats i programmet varifrån subrutinen anropades. Detta löser vi genom att vi förutsätter att denna plats finns i register `R5`. Innan man hoppar till subrutinen måste man alltså ladda in anropsplatsen i detta register.

I nedanstående programavsnitt visar vi ett exempel med en subrutin `write` som kvadrerar och skriver ut innehållet i register `R2`. Subrutinen anropas från två platser i programmet, med parametervärdet `10` respektive `20`.

```
        move  10,r2    ! parameter i första anropet
        move  bk1,r5   ! plats att återvända till i första anropet
        jump  write
bk1:    move  20,r2    ! parameter i andra anropet
        move  bk2,r5   ! plats att återvända till i andra anropet
        jump  write
bk2:    stop

write:  mul   r2,r2,r2 ! subrutinen: kvadrera parametern r2
        print r2       ! skriv ut r2
        jump  r5       ! hoppa tillbaka
```



Figur 1: METool, fönster.

Som synes har vi utökat assemblerspråket som beskrevs i tabell 1 med möjligheten att ladda in ett läge i ett register och att hoppa till det läge som finns i ett register.

3 En emulator för ME-datorn

I detta avsnitt beskrivs ett program, METool, som emulerar en ME-dator enligt beskrivningen i de tidigare avsnitten. METool, som är skrivet i Java, kan bland annat läsa in ett ME-program och utföra instruktionerna i programmet. I METool kan man också se innehållet i register och minne under exekvering av ett program, och (om man vill) editera ME-program.

När man startat METool får man upp ett fönster (se figur 1). Till vänster i fönstret visas innehållet i register och minne, uppe till höger skriver man program, nere till höger visas meddelanden.

Det finns två menyer:

File Menyalternativ för att öppna och spara program.

New Töm programfönstret och meddelandefönstret.

Open Öppna ett ME-program dvs läs in programmet från en fil och visa det i programfönstret. Filer som innehåller ME-program bör ha tillägget *.mep*. Man kan editera ett ME-program i en editor och sedan läsa in det i METool, men man kan också editera program direkt i programfönstret. Då har man inte tillgång till så många editeringskommandon; bara CONTROL-X (klipp ut), CONTROL-C (kopiera) och CONTROL-V (klistra in).

Save Spara ME-programmet som finns i programfönstret i en fil.

Save as... Spara ME-programmet i en ny fil.

Exit Avsluta METool.

Program Menyalternativ för att kompilera och köra ME-program. Dessa menyalternativ finns också som knappar i verktygsraden under menyraden.

Clear mem Nollställ register och minne.

Compile Kompilera programmet. Eventuella felmeddelanden visas i meddelandefönstret.

Run Exekvera det kompilerade programmet från början till slut (från den första satsen till `stop`-satsen).

Stop Avbryt exekvering av programmet. Används om ett program har ”gått i loop”.

Step from start Påbörja stegvis exekvering av programmet.

Step Exekvera en sats i programmet. Den sats som står i tur att exekveras markeras i programfönstret.