Datorteknik
BCD-aritmetik

—

Michael Josefsson

Version 0.1 2019

Innehåll

1. Inledning

Vi har tills nu studerat binära tal med basen två och dess ettor och nollor. En annan talbas som kan vara värd att känna till är den decimala, trots att processorn och logiken fortfarande är binär. Med *binärkodade decimala siffror, Binary Coded Digits*, BCD, avses här att de decimala siffrorna `0`–`9` kodas med fyra bitar vardera. Med fyra bitar har vi det hexadecimala talsystemet med sexton symboler `0000`..`1111`. I BCD-fallet används dock bara de som motsvarar de tio symbolerna `0`–`9`:

Bin	Hex	BCD
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	x
1011	B	x
1100	C	x
1101	D	x
1110	E	x
1111	F	x

De med 'x' markerade positionerna i BCD-kolumnen är inte tillåtna.

Omvandling mellan decimalt tal och BCD sker med tabellen ovan, siffra för siffra.

Exempel: Omvandla det decimala talet 6591 till BCD

```
6591 = 0110 0101 1001 0001 = 0110010110010001
```

Den resulterande bitsträngen är i detta fall 16 bitar lång och skulle precis få plats i två byte. Generellt kan en byte husera två BCD-kodade decimala siffror. ■

Man förstår av exemplet att BCD-kodning inte är optimal användning av bitarna, `1010`–`1111` används ju aldrig. Med rent binär tolkning, där alla bitar används, är `6591 = 11001 1011 1111` dvs 13 bitar och 3 bitar kompaktare. Man kanske kan undra varför BCD då blivit en så använd kodning?

Bland fördelarna kan nämnas att utskrifter förenklas avsevärt då talet redan är i presentabel form siffra för siffra. Om det binära talet `1100110111111` skall skrivas ut decimalt måste först de enskilda siffrorna skiljas ut genom en tidsödande ”dela med tio”-process.[1]

Beräkningarna liknar dessutom de vi är vana vid för hand, varför algoritmer är relativt ”rakt på” att implementera och felsöka.

Precis som vid binära tal kan inte avrundningsfel inträffa vid addition, subtraktion och multiplikation. Av denna anledning är BCD-kodning vanlig i kassaregister där varje öre måste räknas. I detta fall unnar man sig dessutom lyxen att *alltid* använda heltal och placera en decimalpunkt två siffror från höger: `123.45`.

[1] $\frac{6591}{10} = 1100110111111/tio = 1010010011.0001 = 659.1$, $1010010011/tio = 100001.1001 = 65.9$, $1000001/tio = 0110.0101 = 6.5$ och slutligen $0110/tio = .0110 = 0.6$. Divisionen lämnas som en övning till läsaren.

2. Aritmetik

I detta avsnitt beskrivs de grundläggande aritmetiska operationerna addition, subtraktion, multiplikation och division. Läs inte allt på en gång, låt addition sjunka in ordentligt. Allt senare bygger på den.

2.1. Addition

Addition av BCD-kodade tal utförs som vanliga decimala tal, med ett viktigt undantag: **om resultatsiffran efter additionen blivit större än 9 adderas +6**. Anledningen till detta inses om man genomför additionen `9 + 1` som med fyra bitar blir hexadecimalt `A`, när det i själva verket skulle bli `0`, dvs man måste vid behov hoppa över de icke tillåtna symbolerna `A–F`.

Denna åtgärd kallas decimaljustering, *decimal adjust*, och flera processorer har stöd för BCD-addition inbyggt.[1]

Exempel: Addera 5 + 6 som BCD-kodade tal.

```
  5 = 0101
+ 6 = 0110
-----------
     1011   = 11, > 9
     0110   + 6
----c------
    1 0001   = '11' i BCD
```

Additionen gav ett resultat, `1011` som inte är inom det tillåtna området `0–9`, varför en ytterligare addition med +6 genomfördes för att få rätt siffra. Samtidigt noterar man en utgående carry `c` från denna siffra till nästa siffra och resultatet är BCD-mässigt korrekt. ■

[1] I Motorola M68000 är stödet inbyggt med instruktioner som *Add BCD* och *Sub BCD*, (`abcd`) och (`sbcd`). I Z80 heter instruktionen *Decimal Adjust Accummulator* (`daa`)

Med flera siffror är förfarandet analogt med observationen att man i varje addition kan behöva addera +6. Om den utgående carryn adderas till siffran 9 blir summan `A` och även i detta fall måste addition med +6 utföras. Ett exempel demonstrerar:

Exempel: Addera 38 + 75 som BCD-kodade tal.

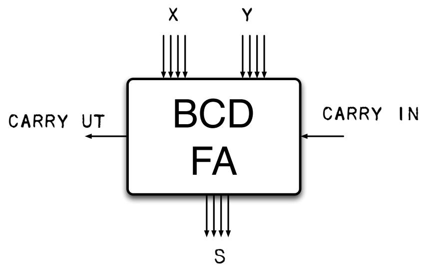
För tydlighets skull används här och i fortsättningen ett mellanslag mellan de olika siffrorna.

```
  38 = 0011 1000
+ 75 = 0111 0101
----------c------
       1010 1101   = $A respektive $D
       0110 0110     +6 respektive +6
-----c-----------
  0001 0001 0011   = '113'
```

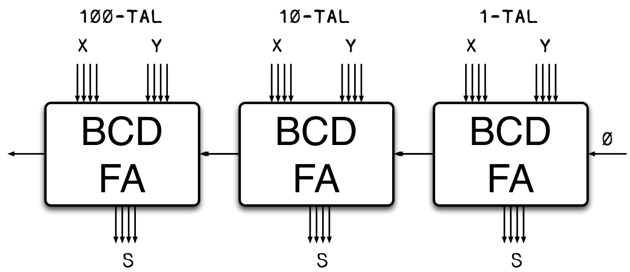
Notera hur en den högra additionen med +6 gav en carry (c) till siffran till vänster. ■

2.2. BCD-adderare

På samma sätt som man beskriver addition av binära tal med en fulladderare, FA, kan en fulladderare för BCD ritas som en komponent som adderar två BCD-siffror plus en inkommande carry och genererar en BCD-siffra och en utgående carry:

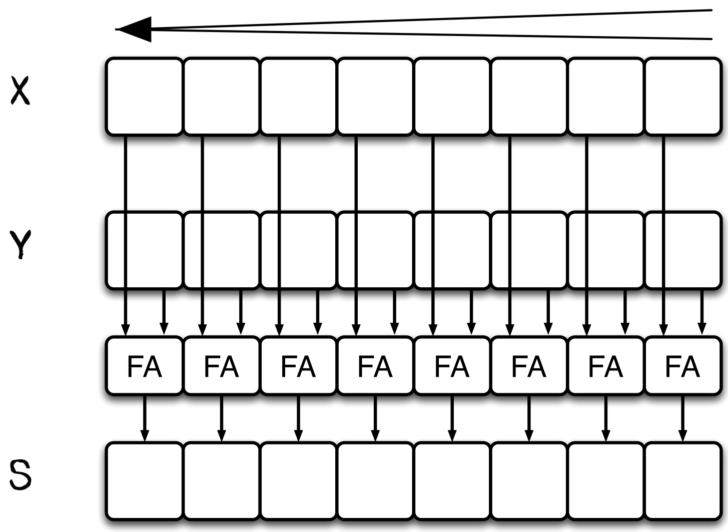


Blocket FA ovan måste alltså innehålla logik som vid behov genomför den önskade extra additionen med +6. Addition av flersiffriga tal utförs med kedjor av fulladderare, en per siffra i talområdet:



Fulladderarkedjan kan således användas för att addera två åtta-siffriga tal genom att, i en loop, addera talen två och två, och i varje loopvarv i erhålla en carry- och en summabit:

$$(C_i, S_i) = X_i + Y_i + c_{i-1} \quad med \quad c_{-1} = 0$$



I ett program behövs endast en (1) programrutin för fulladderare. Denna rutin underkastas sedan i tur och ordning de ingående talens siffror med början i minst signifikant siffra, från höger till vänster.

2.3. Subtraktion

Subtraktion utförs enklast som addition med omvänt tecken. Hittills har talen antagits vara teckenlösa men för subtraktion krävs alltså att vi inför tal med tecken.

Tio-komplementet Precis som i tvåkomplementsfallet för basen 2, finns ett *tiokomplement* för basen 10. Genom att komplementera ett tal byts talets tecken. I tvåkomplementsfallet sker detta genom att ett-komplementera (invertera) varje siffra (bit) och addera +1. Tio-komplementering görs på liknande sätt genom att nio-komplementera och addera +1.

I det binära fallet används tvåkomplement och en bit, teckenbiten, indikerar att talet är negativt. I fallet med BCD är situationen något otympligare. Ett ensiffrigt BCD-tal kan med tecken tolkas som:

utan tecken	0	1	2	3	4	5	6	7	8	9
med tecken	0	1	2	3	4	–5	–4	–3	–2	–1

varför alla tal med tecken som inleds med 5–9 i själva verket är negativa tal. Det finns alltså inte längre bara en *bit* som anger om talet är negativt.

En konsekvens av detta är att talområdet inte kan vara $[-9xxx, +9xxx]$ utan enbart $[-5xxx, +4xxx]$. För att undvika detta utnyttjas *tecken-beloppsrepresentation* där beloppet, som ju alltid är positivt, blir [0000, 9999], och minustecknet hanteras på annat sätt.

Nio-komplementet av en siffra, X, är den siffra som skall läggas till för att summan skall bli 9 enligt nedanstående tabell:

X	0	1	2	3	4	5	6	7	8	9
9–X	9	8	7	6	5	4	3	2	1	0

Exempel: Bestäm BCD-koden för det negativa talet –963.

Talet är BCD-kodat så att byta tecken på det är detsamma som att tio-komplementera det. Med tabellen ovan, siffra för siffra och en avslutande addition fås

```
     963   talet
     036   9-komplement
      +1
-------------
     037   10-komplement
```

En kontroll är att `963 + 037 = 1) 000` dvs de tre sista siffrorna adderar ihop till `000`. Den därvid uppkomna carryn kan man bortse ifrån.[2]

I full BCD-detalj kan komplementeringen utföras som

```
     963   1001 0110 0011   talet
     036   0000 0011 0110   9-komplement
      +1   0000 0000 0001
-------------
     037   0000 0011 0111  10-komplement
```

■

Exempel: Ett annat tal, 960, tio-komplementeras på samma sätt. Glöm inte att decimaljustera resultatet.

```
     960   1001 0110 0000   annat tal
     039   0000 0011 1001   9-komplement
      +1   0000 0000 0001
---------------------------
     03A   0000 0011 1010   10-komplement
      +6   0000 0000 0110   DAA
---------------------------
           0000 0100 0000   10-komplement ('040')
```

Som kontroll ser man att $960 + 40 = 10\,000$. ■

Subtraktion Med tio-komplementet kan nu subtraktionen $\mathcal{X} - \mathcal{Y}$ utföras som

$$\mathcal{X} - \mathcal{Y} = \mathcal{X} + (-\mathcal{Y})$$

Exempel: Utför subtraktionen 2943 – 698.

Först tas `-698` fram...

```
   +0698    0000 0110 1001 1000
            1001 0011 0000 0001   9-komplement
      +1    0000 0000 0000 0001   +1
----------------------------------
   -0698    1001 0011 0000 0010  Koll: 0698 + 9302 = 10000
```

... och sedan görs en avslutande addition av de två talen:

[2] För tre siffror utför man egentligen 1000-komplementet! Och detta genom att beräkna $1000 - x$. Och så vidare.

```
    2943   0010 1001 0100 0011
 +  9302   1001 0011 0000 0010
-------------c----------------
           1011 1100 0100 0101
       +   0000 0110 0000 0000  Decimaljustera de > 9
------------------------------
           1100 0010 0100 0101
       +   0110 0000 0000 0000  Decimaljustera de > 9
------------------------------
           0010 0010 0100 0101  Resultat: +2245
```

Här har decimaljusteringen utförts parallellt med alla siffror för att kunna betraktas. Med en ensiffras fulladderare sker denna decimaljustering i varje siffra dock automatiskt. ∎

2.4. Multiplikation

Multiplikation utförs i huvudsak som upprepad addition men med lite annorlunda disposition än vanligt. En metod anpassad och strömlinjeformad för datorhårdvara blir enklare att implementera än skolmetoden. För enkelhets skull används enbart positiva tal i algoritmen. Den i algoritmerna nedan förekommande additionen utförs i BCD-fallet som komplett addition med decimaljustering.

Först visas multiplikationen på vanligt sätt för att visa de olika delresultaten när algoritmen senare anpassas för datorimplementering.

Exempel: Multiplicera talen 1234 och 4321.

Beräkningarna utförs som med decimala tal för att principen bättre skall framgå.

```
          1234     multiplikator
          4321     multiplikand
--------------
          1234     1234 * 1
         2468      1234 * 2
        3702       1234 * 3
       4936        1234 * 4
--------------
       5332114     1234*4321
```

∎

Algoritmen ovan är känd. Ett problem ur vår synvinkel är att ett antal delresultat måste samlas på hög innan en avslutande, tämligen massiv, addition förlöser det det hela till ett resultat. Det är en nackdel. Men en nackdel som dock kan lösas genom att införa *partialprodukter*, dvs addera så fort något kan adderas.

Not 1. En multiplikation av två fyra-siffrors tal, en så kallad *4x4*-multiplikation kan inte ge mer än 4 + 4 = 8 siffror i resultatet.

Med ledning av detta ansätts en nollställd partialprodukt av önskad längd redan i början:

Exempel: Partialprodukt

```
         1234      multiplikator
_________4321_     multiplikand
     00000000      partialprodukt p_0  = 0
_________1234_     1234 * 1
         1234      partialprodukt p_1
________2468__     1234 * 2
        25914      partialprodukt p_2
_______3702___     1234 * 3
       396114      partialprodukt p_3
______4936____     1234 * 4
      5332114      partialprodukt p_4 = 1234*4321
```

■

Ur denna uppställning och beräkning kan man se hur vissa siffror ändras och hur vissa *aldrig* ändras: I p_1 är sista siffran, 4, redan färdig, klar och beräknad. I p_2 tillkommer näst sista siffran, som sedan aldrig ändras, och så vidare. I varje steg tillverkas alltså en ytterligare siffra som är klar och kan läggas åt sidan. Additionen är i varje steg enbart fyra siffror plus fyra andra siffror! Tyvärr är denna addition mer och mer åt vänster. . . Men, då det enligt ovan, i varje steg, producerats en siffra till höger kan man skifta partialprodukten ett steg åt höger och på så sätt hålla additionerna rakt under varann.

Exempel: Partialprodukt och högerskift

```
         1234      multiplikator
_________4321_     multiplikand
    0000 0000      partialprodukt p_0  = 0
_________1234_     1234 * 1
    0000 1234      partialprodukt p_1
     000 0123 4    skiftad
_________2468_     1234 * 2
     000 2591 4    partialprodukt p_2
      00 0259 14   skiftad
_________3702_     1234 * 3
      00 3961 14   partialprodukt p_3
       0 0396 114  skiftad
_________4936_     1234 * 4
       0 5332 114  partialprodukt p_4
         0533 2114 skiftad = 1234*4321
```

■

Not 2. Generellt erhålls slutresultatet av en *n x n*-multiplikation efter *n* steg. Multiplikationen är i 4x4-fallet alltid i 4 steg.

Not 3. Övre halvan av resultet återfinns under ursprungsoperanderna, undre halvan har producerats under processens gång.

Not 4. För en *n x n*-multiplikation behövs enligt ovan endast en 2*n*-adderare. Detta gäller generellt.

Multiplikation med tecken Multiplikationen ovan är beskriven för teckenlösa tal. Med tecken-belopp-representerade tal multipliceras talens belopp och det resulterande tecknet fås genom att betrakta ursprungstalens tecken där olika tecken ger negativt resultat medan samma tecken alltid ger positivt tecken på resultatet.

Multiplikationsdetaljer I beräkningen ovan utfördes delprodukterna, till exempel `3702 = 1234·3` utan förklaring. Man kan notera att en sådan delprodukt som mest är en multiplikation mellan ett fyra-siffrigt tal och ett en-siffrigt. Ett enkelt och direkt sätt att utföra denna multiplikation med ett en-siffrigt tal är direkt upprepad addition. Denna sorts "multiplikation" tar olika lång tid beroende på antalet additioner men tar under alla förhållanden aldrig mer än högst 9 additioner.

Fler möjligheter

De ensiffriga multiplikationerna kan beräknas på tre sätt:

- upprepade additioner,
- tabelluppslagning i en multiplikationstabell eller
- använda stöd i befintlig hårdvara.

Metoderna med upprepade additioner eller tabelluppslagning är enklast att implementera. Upprepad addition tar, som nämnts, olika lång tid — och i bland väldigt lång tid — beroende på operanderna. Tabelluppslagning tar plats (82 värden) men kan koda hela resultatet i en byte för en total tabellstorlek av 82 bytes. Symmetrier (`3·4 = 4·3`) kan användas för att få ner tabellstorleken ytterligare. Inte sällan finns multiplikationsinstruktioner i en processor. I fallet med 8x8-till-16-bitars instruktion kan denna användas för 1x1-siffras multiplikation (`9·9 = 81`) enligt ovan, men också för 3x1- eller 2x2-siffrors multiplikation (`999·9 = 8991`, `99·99 = 9801`).

8x8-multiplikation Med 8 siffror i både multiplikand och multiplikator blir algoritmen något mer omfattande att visualisera men stegen är desamma:

Exempel: Partialprodukt och högerskift 8x8 siffror.

I detta exempel ansätts plats för maximalt resultat om 16 siffror:

```
                 1736 5289               multiplikator
                 3247 5178               multiplikand
---------------------------
       0000 0000 0000 0000               p_0  = 0
               1 3892 2312               1736 5289 * 8
---------------------------
               1 3892 2312               p_1
                 1389 2231 2             >>
               1 2155 7023               1736 5289 * 7
---------------------------
               1 3544 9254 2             p_2
                 1354 4925 42            >>
                 1736 5289               1736 5289 * 1
---------------------------
                 3091 0214 42            p_3
                  309 1021 442           >>
                 8682 6445               1736 5289 * 5
---------------------------
                 8991 7466 442           p_4
                  899 1746 6442          >>
               1 2155 7023               1736 5289 * 7
---------------------------
               1 3054 8769 6442          p_5
                 1305 4876 9644 2        >>
                 6946 1156               1736 5289 * 4
---------------------------
                 8251 6032 9644 2        p_6
                  825 1603 2964 42       >>
                 3473 0578               1736 5289 * 2
---------------------------
                 4298 2181 2964 42       p_7
                  429 8218 1296 442      >>
                 5209 5867               1736 5289 * 3
---------------------------
                 5639 4085 1296 442      p_8
                 0563 9408 5129 6442     >> svar 16 siffror
---------------------------
```

Den avslutande produkten blir `0563940851296442`. Kontrollera alla siffror på din miniräknare! Om svaret skall ha 8 siffror blir det en användarfråga om det är den första eller sista delen som skall redovisas, eller om ett felmeddelande skall ges ”Svaret får inte plats” så fort en partialprodukt överstiger 8 siffror. ■

2.5. Division

Division utförs i huvudsak som ”för hand” med upprepad subtraktion och förflyttning i sidled. Liksom i fallet med multiplikation vill man placera alla subtraktioner och, i förekommande fall, återställande additioner i direkt under varann.

Först visas divisionen

$$\frac{43665}{123} = 355$$

enligt ”liggande stolen”:

Exempel: Liggande stolen

```
          123
      _______
355 | 43665
       -355      1 * 355
       ----
         816
        -710     2 * 355
       -----
         1065
        -1065    3 * 355
        -----
            0
```

■

Den inledande subtraktionen är 436−355 och går att göra 1 gång innan resultatet blir negativt. Sedan flyttas nämnaren ett steg höger, nästa siffra ur täljaren läggs till och subtraktionen upprepas tills resultatet blir negativt (2 gånger nu) och metoden fortsätter tills alla siffror i täljaren använts.[3]

Not. Subtraktionen upprepas tills resultatet är < 0. Den sista subtraktionen återtas och räknas inte in i kvoten.

[3] Försåvitt inte decimaltal tillåts, ty då fortsätter man på den inslagna vägen, fast med nollor från täljaren, tills önskat antal decimaler producerats.

8/8-division Metoden går att modifiera med motsvarande knep som i multiplikationsfallet. Utförd i 8 decimala siffror kan den beskrivas enligt nedan där nämnaren är fix i position och storlek medan täljaren flyttas ett steg åt vänster i varje omgång. Nämnaren subtraheras upprepade gånger från täljarens siffror och antalet subtraktioner noteras.

Exempel:

```
      |00000000|00043665|    startpositioner
              <<             skift 0
1.    |00000000|0043665-|
    - |00000355|             0 subtraktioner
      |00000000|0043665-|
              <<             skift 1
2.    |00000000|043665--|
    - |00000355|             0 subtraktioner
      |00000000|043665--|
              <<             skift 2
3.    |00000000|43665---|
    - |00000355|             0 subtraktioner
      |00000000|43665---|
              <<             skift 3
4.    |00000000|43665---|
    - |00000355|             0 subtraktioner
      |00000000|43665---|
              <<             skift 4
5.    |00000004|3665----|
    - |00000355|             0 subtraktioner
      |00000004|3665----|
              <<             skift 5
6.    |00000043|665-----|
    - |00000355|665-----|    0 subtraktioner
      |00000043|665-----|
              <<             skift 6
7.    |00000436|65------|
    - |00000355|             1 * 355 = 81 subtraheras
      |00000081|65------|
              <<             skift 7
8.    |00000816|5-------|
    - |00000710|             2 * 355 = 710 subtraheras
      |00000106|5-------|
              <<             skift 8, sista
      |00001065|--------|
    - |00001065|             3 * 355 = 1065 subtraheras
              0              rest = 0
```

■

Resultatet avläses till höger, uppifrån och ned: 00000123. En 8/8-division tar på detta sätt alltid 8 steg. Det går att snabba upp den genom att studera de ingående talen och vänsterskifta tills första siffran ≠ 0 finns i entalspositionen men strömlinjeformningen enligt ovan är generell och direkt. Divisionen gick här jämnt ut med resten = 0. I det allmänna fallet ger divisionen dock rest.

3. Implementation

Medan man mycket väl *skulle* kunna spara BCD-talen som tal med tecken i tio-komplementsform är det sannolikt en onödig komplikation. Den stora fördelen med talen i tecken-belopp-representation är att de alltid är redo att skrivas ut utan behandling.

Förslagsvis lagras alla resultat i tecken-belopp-representation. Det underlättar felsökning och kontroll av minnesinnehåll då alla tal är direkt avläsningsbara.

Ett sätt att använda 8 bytes för 8 BCD-siffror är att låta en byte innehålla en BCD-siffra i sin lägsta nibble. Den högsta nibbeln är då fri att använda för att till exempel signalera talets tecken. Det positiva talet 13587195 lagras som

\$01	\$03	\$05	\$08	\$07	\$01	\$09	\$05
högsta byte							lägsta byte

medan det negativa talet –13587195 lagras med en ”tecken-nibble”

\$81	\$03	\$05	\$08	\$07	\$01	\$09	\$05
högsta byte							lägsta byte

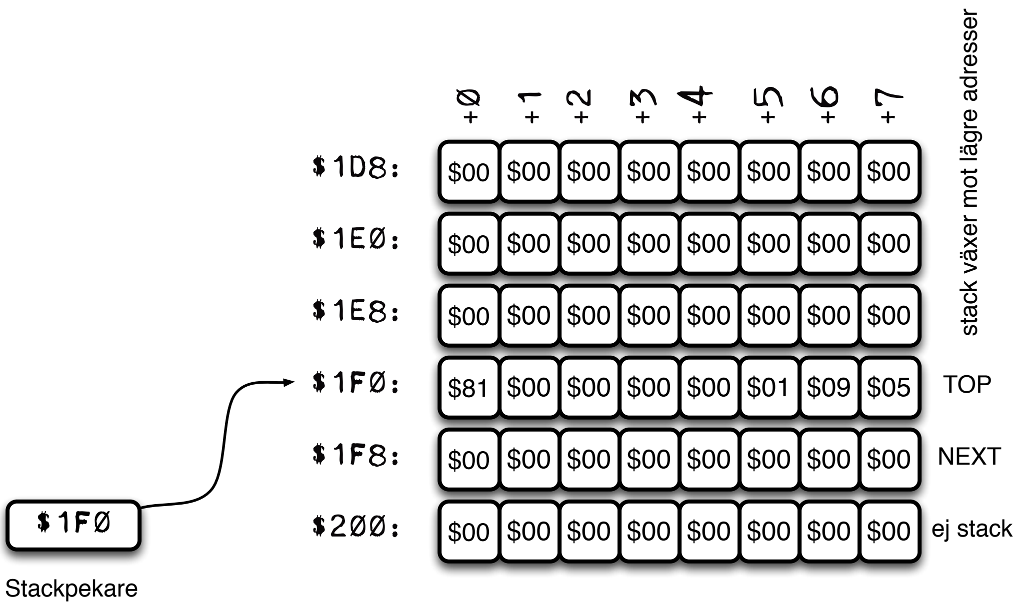
Med teckenbiten i högsta byten kan utskrift ske genom att först skriva ut ett eventuellt tecken och sedan ignorera alla `$00`-or tills siffrorna `195` i talet kommer.

En möjlighet är att överföra talet som skall skrivas ut till en intern buffert för utskriftsformattering utan onödiga nollor innan strängen skickas till displayen: `-00001234`→`-1234`.

Minneslayout Som vanligt är en *stack* en lämplig ADT för att lagra numeriska värden för aritmetiska beräkningar. Vill man inte fysiskt flytta runt alla värden på stacken (`push`, `pop` osv) är en stackpekare att föredra. Pekare används sedan hela tiden för att hämta och lagra enskilda siffror vid beräkningarna.

För enkelhets skull kan man låta alla beräkningar utföras med alla siffror, dvs för att addera två tal måste alla 8 siffror adderas även om talen i sig inte kräver alla 8 positioner.[1]

[1] Ett litet offer för ökad enkelhet på bekostnad av högsta effektivitet (om högsta effektivitet är målet använder man inte BCD ändå, right?)



Figuren visar en tänkt minneslayout utförd som en stack där varje tal tar 8 bytes i anspråk. Stackpekaren pekar på senast ditlagda tal, här talet –195 respektive 0. Med en bestämd maximal tallängd om 8 bytes är upp/ned-räkningen till nästa/föregående tal alltid samma.[2]

Fördelen är att stacken enkelt kan manipuleras med rutiner som `drop` och `dup` men även med mer komplexa rutiner som `+`, `-` och `negate` med flera.[3] Ett tal behöver aldrig *raderas* från stacken, ej heller nollställas, det räcker att peka om stackpekaren. I figuren ovan innehåller de inte utnyttjade talen enbart `$00`, detta är inte nödvändigt och i verkligheten osannolikt.

En ytterligare fördel är att stacken även kan används *internt* i respektive aritmetisk beräkning för att lagra mellanresultat! Genom att nå respektive siffra i talen med pekare kan effektiva programflöden uppnås.

[2]Faktum är att de tre lägsta bitarna i adressen är respektive byte-adress inom ett tal.

[3]Lägg märke till att en subtraktion utförs som ”call negate, call plus” om en paraeterstack används.

A. 8x8-multiplikation med upprepad addition

Här visas multiplikationen 1234·4321 med upprepad addition utskriven i varje steg.

Exempel: Partialprodukt och högerskift, alla steg

```
          1234      multiplikator
          4321      multiplikand
 ---------------
     0000 0000      partialprodukt p_0  = 0
          1234      1234 * 1
 ---------------
     0000 1234      partialprodukt p_1
           123 4    skiftad
          1234      1234 * 2
          1234
 ---------------
     0000 2591 4    partialprodukt p_2
           259 14   skiftad
          1234      1234 * 3
          1234
          1234
 ---------------
     0000 3961 14   partialprodukt p_3
           396 114  skiftad
          1234      1234 * 4
          1234
          1234
          1234
 ---------------
     0000 5332 114  partialprodukt p_4
          0533 2114 skiftad = 1234*4321
```

■

B. *Double dabble*-algoritmen

Ibland kan befintlig hårdvara för binär multiplikation användas. För att använda sådana basen-2-resultat i decimalaritmetiken måste de först omvandlas till BCD-tal.

För omvandling mellan binära tal och BCD-tal kan en variant av den s.k. *double dabble*-metoden användas. I grova drag utförs successiva vänsterskiftningar av det binära talet och siffran 6 adderas[1] om resultatet i en BCD-kolumn (här ″10″ och ″1″) blir större än 9:

Exempel: Omvandla $\texttt{1100}_2$ till BCD med *double dabble*-algoritmen

Op	10	1	Binär-tal
Start	0000	0000	1100
Skift	0000	0001	100
Skift	0000	0011	00
Skift	0000	0110	0
Skift	0000	1100	← större än 9
6	0000	0110	← addera 6
+	0001	0010	
BCD	1	2	← svar i BCD

Med resultatet i BCD-form kan det användas för att adderas till partialprodukten i den ursprungliga multiplikationen.

—o=Ö=o—

[1] I en vanlig implementation av metoden utförs istället addition med 3 innan vänsterskiftningen.