

INSTITUTIONEN FÖR DATAVETENSKAP
UNIVERSITETET OCH TEKNISKA HÖGSKOLAN
LINKÖPING

Vägen mot en minimal Forth arkitektur

av

Mikael Patel

INDUSTRISERIEN

CADLAB, Januari 1990

Postadress:

Institutionen för datavetenskap
Universitetet i Linköping och
Tekniska Högskolan
581 83 Linköping



Mailing address:

Department of Computer and
Information Science
Linköping University
S-581 83 Linköping, Sweden

Vägen mot en minimal Forth arkitektur

av

Mikael Patel

E-mail: mip@ida.liu.se

Sammanfattning: Denna rapport visar att ett trådat programmeringsspråk som Forth kan realiseras med ett mycket litet antal primitiva operationer. Genom att successivt bryta ner operationerna i Forth kan man finna en minimal uppsättning som också är minimal med avseende på antal grindar vid realisering i hårdvara. Den föreslagna hårdvaruarkitektur kan med hjälp av en enkel metodik lätt utvidgas beroende på krav på prestanda. Den grundläggande hårdvarukonstruktionen kvarstår och endast nya operationer introduceras på hårdvarunivån. Man får härigenom en flytande gräns mellan hård- och mjukvara. En genomgång av de aritmetriska operationerna och kontrollstrukturerna i Forth genomförs detaljerat för att visa hur man kan finna den minimala uppsättning primitiver som krävs för att realisera Forth.

Postadress:

Institutionen för datavetenskap
Universitetet i Linköping och
Tekniska Högskolan
581 83 Linköping

Mailing address:

Department of Computer and
Information Science
Linköping University
S-581 83 Linköping, Sweden

VÄGEN MOT EN MINIMAL FORTH ARKITEKTUR

1990-01-29

Mikael Patel

Laboratoriet för datorstödd elektronikkonstruktion (CADLAB)
Institutionen för datavetenskap
Universitetet i Linköping och
Tekniska Högskolan
581 83 LINKÖPING

Sammanfattning: Denna rapport visar att ett trådat programmeringsspråk som Forth kan realiseras med ett mycket litet antal primitiva operationer. Genom att successivt bryta ner operationerna i Forth kan man finna en minimal uppsättning som också är minimal med avseende på antal grindar vid realisering i hårdvara. Den föreslagna hårdvaruarkitektur kan med hjälp av en enkel metodik lätt utvidgas beroende på krav på prestanda. Den grundläggande hårdvarukonstruktionen kvarstår och endast nya operationer introduceras på hårdvarunivån. Man får härigenom en flytande gräns mellan hård- och mjukvara. En genomgång av de aritmetriska operationerna och kontrollstrukturerna i Forth genomförs detaljerat för att visa hur man kan finna den minimala uppsättning primitiver som krävs för att realisera Forth.

Nyckelord: Trådade programmeringsspråk, Forth-83, Datorarkitektur, Stackarkitekturer, Minimal instruktionsuppsättningsdatorsystem (MISC).

1. INTRODUKTION

Trådade programmeringsspråk som Forth [1] tillåter sig definieras med ett fåtal primitiva operationer. Större delen av programmeringsspråket kan beskrivas i sig själv. I denna rapport skall vi söka den minimala uppsättning primitiver som behövs för att realisera programmeringsspråket Forth. Avsikten är att visa att en mycket liten uppsättning instruktioner behövs och att man sedan kan genom analys av applikationer kan avgöra hur och med vilka operationer som den innersta kärnan av instruktionerna skall utökas.

Genom att finna den minimala uppsättningen av operationer kan vi konstruera en grundläggande hårdvaruarkitektur och sedan genom en uppsättning konstruktionsregler visa hur denna maskin systematiskt kan utökas beroende på krav på prestanda. Valet av de operationer som skall realiseras i hårdvara beror främst på hårdvarukostnad och frekvensen av användning i applikationer. Om t.ex. multiplikation är en mycket vanligt förekommande operation bör den stödjas i hårdvara för att ge applikationen högre prestanda. Vidare skall påpekas att arkitekturen också kan användas för att underlätta emulering av den virtuella Forthmaskinen på andra fysiska processorer på marknaden genom att ge förståelse för vilka operationer som är viktiga att realisera i maskininstruktioner hos den underliggande processorn.

Den virtuella Forthmaskinen igenkänns på att den är en sk dual stackmaskin. Två stackar används för att dels överföra parametrar till och resultat från procedurer och dels för att spara returadresser vid proceduranrop. Genom språkets grundläggande funktionella natur vävs operationer och kontrollstrukturer samman utan behov av lokala variabler. I och med att returadresstacken endast används vid anrop av och retur från procedurer kan denna stack användas fritt inom proceduren. Teoretiskt har man visat att det räcker med två stackar för att erhålla samma uttryckskraft som den teoretiska Turingmaskinen [2]. Detta innebär att godtyckligt många nivåer av interpretatorer kan vävas samman med hjälp av de två stackarna och därigenom emulera en Turingmaskin med oändligt minne. Den virtuella Forth maskinen är därmed en mer universell beräkningsmaskin jämfört med en traditionell registermaskin [6].

Trådad kod har visat sig vara det implementeringsmässigt mest effektiva kodningsförfarandet [5] med hänsyn till att modern programutvecklingsmetodik bygger på att små korta subrutiner används för att abstrahera programstrukturen. Kodningsförfarandet ger både minnes- och processoreffektivitet då alla

operationerna hanteras symmetriskt och den extremt låga kostnaden för ett subrutinsanrop i Forth jämfört med programmeringsspråk som t.ex. Pascal och Ada. I dessa språk krävs många minnesoperationer för att realisera ett subrutinsanrop då traditionella processorer saknar hårdvarustöd för denna typ av operation.

Åtkomsten av returadressstacken och möjligheten att härigenom direkt manipulera exekveringsadressen gör att den minimala maskinen inte behöver realisera hopp som en primitivoperation. Detta kan åstadkommas som en högnivåoperation. Hur detta kan realiseras kommer att visas i kommande avsnitt.

Rapporten har följande uppläggning: I första avsnittet genomförs en analys av vilka primitiver som behövs för att realisera aritmetriska och logiska operatorer samt kontrollstrukturer i Forth. Sedan beskrivs dessa primitiver i ett sk registeröverföringsspråk för att ge en djupare förståelse för vilka transaktioner som krävs på hårdvarunivån. Realisering av högnivåoperationerna behandlas därefter då dessa påverkar instruktionshämtningen hos den virtuella maskinen. Sist behandlas en möjlig hårdvarurealisering av de primitiva operationerna och strukturen hos en minimal dual stackarkitektur beskrivs i detalj.

Genomgående krävs elementära kunskaper om Forth. För den ovane ges här en mycket kort introduktion till språket.

En kort introduktion till Forth

"FORTH is like the Tao: it is a Way,
and is realized when followed.
Its fragility is its strength;
its simplicity is its direction."

Michael Ham, winning entry in Mountain View Press's contest to describe FORTH in twenty-five words or less.

Procedurer och funktioner skrivs som sk kolondefinitioner. Dessa börjar med operationen kolon och slutar med semikolon. Som skiljetecken mellan ord används genomgående blankt. Ordet efter kolon är namnet på den nya definitionen (proceduren). Parenteser behandlas som kommentarer i språket och det tillhör god programmeringsstil att beskriva hur definitionen påverkar parameterstacken eftersom all parameteröverföring är implicit. Normalt används en notation som symboliskt visar hur stacken ser ut före och efter operationen.

```
: example ( before -- after) code ;
```

Eftersom Forth skrivs i postfix (omvänd polsk) notation exekveras koden i exakt den ordningen som den är skriven med vissa undantag för kontrollstrukturer då hopp kan ske i koden. Ett infixuttryck kan lätt skrivas om till postfix genom att flytta operationen mellan två operander till efter dessa.

```
2 + 5          => 2 5 +  
X < Y          => X Y <
```

Vid mer komplicerade uttryck måste man också tänka på att överföra operationerna till rätt följd beroende på prioritet och parenteser i infixuttrycket.

```
(2 + 5) * 7    => 2 5 + 7 *  
2 + (5 * 7)   => 2 5 7 * +
```

Postfixkod exekveras i den ordning det står från vänster till höger. Enda undantaget är kontrollstrukturer som kan ge upphov till ändring av flödet genom en definition (hopp). Forth är ett strukturerat språk då det gäller kontrollstrukturer då explicita hopp finns inte. Villkorssatsen, IF-ELSE-THEN, skrivs också i en speciell postfixform.

```
condition IF sectiontrue THEN  
condition IF sectiontrue ELSE sectionfalse THEN
```

Om villkoret, parametern till IF, är sant (ej noll) utförs koden mellan IF och THEN eller IF och ELSE. Om parametern till IF är falskt (noll) sker ett hopp till koden efter ELSE eller till slutet av

villkorssatsen om ELSE saknas. I och med Forths definition av värdena för sant och falskt behövs inte en speciell test-om-sant operation. Nedan kommer också ett itertivtuttryck att användas. Detta liknar mycket Pascals WHILE-sats (men har kraftfullare) och har följande struktur:

```
BEGIN section1 condition WHILE section2 REPEAT
```

Första avsnittet och villkoret utförs och sedan om parametern till WHILE är sant (ej noll) utförs andra avsnittet och behandlingen upprepas. I annat fall, om parametern är falskt (noll), sker ett hopp till efter REPEAT. I Forth ges ofta grundläggande primitiver som utför ett antal sammanslagna operationer ett namn som motsvarar sammanslagningen. En operation som adderar med ett kallas följaktligen 1+ och en operation som testar mot noll 0=.

2. SÖKANDE EFTER PRIMITIVA OPERATIONER

En fråga som ofta har ställs mig då jag presenterar trådade programmeringsspråk är; Vilka primitiver som måste programmeras och vilken är den minimala uppsättningen? Jag skall här visa att man kan härleda en mycket liten uppsättning operation och en tillhörande arkitektur för att realisera Forth. Konstruktionen är inte denna snabbast minimala uppsättningen snarare den billigaste och därmed, i viss mån, den långsammaste. Framställningen här är för att visa att en mycket liten uppsättning räcker och att denna kan sedan utökas efter applikationsbehov. Denna minimala dual stack arkitektur kan sedan genom en enkel metodik utvidgas för att uppnå högre prestanda.

Aritmetriska operationer: plus och minus

Vilka primitiver behövs? Låt oss bryta ner problemet något för att lättare kunna ta oss an det. Kan de aritmetriska operationerna, *plus*, *minus*, etc., realiseras med ett antal primitiver? Och i så fall vilka? Återigen söker vi den billigaste lösningen sett ur hårdvarusynpunkt så en lösning med plus och minus som implicerar en aritmetrisk logisk enhet vore önskvärd.

En gammal metod som bl.a. har används i mekaniska räknedosor för att åstadkomma *plus* är att använda upprepade subtraktion och addition med ett. Minus kan sedan realiseras med hjälp av plus genom att först negera talet. Vidare är negation av ett två-komplements tal samma som att negera ett-komplement (invertera) och addera ett [7].

```
: + ( x y -- z)
  dup 0<                               ( Check direction )
  if begin                               ( Counting downwards)
    dup                                 ( While not zero do)
    while
      1+ swap 1- swap                 ( Increment y and decrement x)
    repeat
  else
    begin                               ( Counting upwards)
      dup                               ( While not zero do)
      while
        1- swap 1+ swap              ( Decrement y and increment x)
      repeat
  then
  drop ;                               ( Drop y and return the result)

: negate ( x -- y) not 1+ ;
: - ( x y -- z) negate + ;
```

Låt oss nu fortsätta sökandet efter den minimal uppsättningen operationer och försöka åstadkomma de operationer som används med hjälp av andra primitiva operationer. Vi har nu problemet att finna primitiver för att åstadkomma operationen *test-om-negativt*, hopp- och stackoperationer.

Booleska och relationsoperationer

Hur kan man åstadkomma *test-om-negativt*? Detta kan vi skriva om som en test av mest signifikanta biten i talet. Genomgående antar jag att vi har att göra med en 16-bitsrealisering av den minimala Forth

maskinen därav konstanten -32768 (hex 8000) som motsvarar ett 16-bitstal med den mest signifikanta biten satt till ett övriga bitar till noll:

```
-32768 constant minint
```

```
: boolean ( x -- flag) 0= 0= ;
: 0< ( x -- flag) minint and boolean ;
: 0> ( x -- flag) dup 0< swap 0= or not ;
: = ( x y -- flag) - 0= ;
: < ( x y -- flag) - 0< ;
: > ( x y -- flag) - 0> ;
```

Relationsoperatorerna, *lika-med*, *mindre-än* och *större-än*, definieras nu lätt med det vi har fått ihop genom att använda subtraktion och test i relation till noll istället. Ett undantag är operationen *lika-med* eftersom en effektivare realisering kan åstadkommas med hjälp av *exklusivt-eller*.

```
: = ( x y -- flag) xor 0= ;
```

Listan över primitiver kan minskas ytterligare genom att de logiska operatorerna kan åstadkommas med hjälp av en funktionellt komplett binäroperator som t.ex. *nand* eller *nor*. De övriga operatorerna, *icke*, *och*, *eller* och *exklusivt-eller* definieras här med hjälp av *nand*:

```
: not ( x -- y) dup nand ;
: and ( x y -- z) nand not ;
: or ( x y -- z) not swap not nand ;
: xor ( x y -- z) over over not nand >r swap not nand r> nand ;
```

Stackoperationer

Nu är vi framme vid stackoperationerna. Samtliga stackoperationerna kan skrivas om som minnesaccess-funktioner genom att t.ex. första cellen i minnet som en tillfällig lagringsplats. Åter skall det nog påpekas att vi inte söker en realisering med hög prestanda utan en med minimalt antal primitiver. I avsnittet om hårdvarurealisering kommer dock flera av stackoperationerna att erbjudas direkt i hårdvara utan extra kostnad eftersom dessa är implicita i de övriga primitiverna.

```
variable t
: t! ( x -- ) t ! ;
: t@ ( -- x) t @ ;
```

Först två hjälpdefinitioner för att flytta data till och från den tillfälliga lagringsplatsen i minnet. Nu kan stackoperationerna ges med hjälp av dessa.

```
: drop ( x -- ) t! ;
: dup ( x -- x x) t! t@ t@ ;
: swap ( x y -- y x) t! >r t@ r> ;

: rot ( x y z -- y z x) >r swap r> swap ;
: over ( x y -- x y x) >r dup r> swap ;
: ?dup ( x -- [0] or [x x]) dup if dup then ;

: r@ ( -- x) r> r> dup >r swap >r ;
```

Aritmetriska operationer: fortsättning

Vad som saknas nu är att visa att de övriga aritmetriska operationerna kan beskrivas med de befintliga primitiverna. Först de elementära operationerna. Genom att använda regler om inverstransformationer och hur operationer uppför sig under dessa förhållanden kan *subtraktion-med-ett* operationen definieras i form av ett-komplement och *addition-med-ett* [7].

```
: 1- ( x -- y) not 1+ not ;
: 2+ ( x -- y) 1+ 1+ ;
: 2- ( x -- y) not 2+ not ;
```


Absolutbelopp, min- och maxfunktionern kan enkelt definieras i de övriga:

```
: abs ( x -- y) dup 0< if negate then ;
: min ( x y -- z) over over > if swap then drop ;
: max ( x y -- z) over over < if swap then drop ;
```

Nu till de sista aritmetriska operationerna; *multiplikation* och *division*. Dessa kan realiseras som upprepad addition eller subtraktion men tecknet på operanderna måste först kontrolleras:

```
: * ( x y -- z)
dup                                ( Check not zero)
if over 0< over 0< xor >r         ( Calculate sign of result)
  0 rot abs rot abs                ( Use absolute values)
  begin
    dup                            ( While not zero do)
    while
      swap rot over +              ( Add to accumulator)
      swap rot 1-                  ( And decrement counter)
    repeat
      drop drop                    ( Drop temporary parameters)
      r> if negate then            ( Check sign for negate)
    else
      swap drop                    ( Return zero)
    then ;

: 2* ( x -- y) 2 * ;

: /mod ( x y -- r q)
dup                                ( Check not zero division)
if over 0< >r                      ( Save sign of dividend)
  over 0< over 0< xor >r          ( Calculate sign of result)
  0 rot abs rot abs                ( Use the absolute values)
  begin
    swap over - dup 0< not        ( Calculate next remainder)
    while                          ( Check not negative)
      swap rot 1+                  ( Increment quotient)
      rot rot                       ( And go again)
    repeat
      + swap                        ( Restore after last loop)
      r> if negate then            ( Check sign of quotient)
      r> if swap negate swap then  ( Check sign of remainder)
    then ;

: / ( x y -- q) /mod swap drop ;
: mod ( x y -- r) /mod drop ;
: 2/ ( x -- y) 2 / ;
```

Så långt de aritmetriska operationerna. Vad som nu återstår att bearbeta är dels hantering av kompilerade talen (literals) och kontrollstrukturen.

Kompilerande operationer och kontrollstrukturer

Konstanter i kod måste vid exekveringstillfället lägga sitt värde på parameterstacken. Detta sker genom funktionen (*literal*). För att ge en definition som blir oberoende antal bitar per ord i en implementeringsmaskin används *cell* och *cell+*:

```
2 constant cell
: cell+ ( x -- y) 1+ 1+ ;

: (literal) ( -- n) r> dup cell+ >r @ ;
: compile ( -- ) r> dup cell+ >r @ , ;
: literal ( n -- ) compile (literal) , ; immediate
```

Literal är ett komplicerat ord, som alltså kompilerar då det exekveras och det gör den alltid då den är immediate.

För att hantera minnesallokering måste också ett antal primitiver skapas. Dessa följer traditionella implementeringsmetoder. Ovan används operationen *komma* som allokerar och tilldelar minne i den sk ordlistan (eng. *dictionary*). För att hantera allokering används normalt fyra primitiver. Först en variabel som håller adressen till nästa fria minnescell i ordlistan sedan en funktion som ger innehållet av denna variabel samt en funktion för allokering av minne.

```
variable dp ( -- addr)

: here ( -- x) dp @ ;
: allot ( n -- ) dp +! ;
: , ( x -- ) here ! cell allot ;

: +! ( n addr -- ) dup @ rot + swap ! ;
```

För att avsluta denna analys av vilka primitiver som en minimal virtuell Forthmaskin behöver skall vi nu betrakta kontrollstrukturerna i Forth. Normalt är även dessa beskrivna med hjälp av ett antal primitiver. Vi behöver realisera två hoppoperationer, dels ovillkorligt hopp och dels villkorligt hopp. Dessa kommer att använda innehållet av efterföljande minnescell som hoppadress. För att inte ge en rekursiv definition av den minimala maskinen använder jag här direktadresser och inte relativa adresser.

```
: (branch) ( -- ) r> @ >r ;
```

Ovillkorligt hopp ställer inte till några större problem. Det är bara att hämta återhoppadressen och sedan vad den pekar på och använda detta som returadress. Villkorligt hopp är något svårare eftersom vi måste avgöra utifrån parametern om ett hopp skall göras eller om hoppadressen skall passeras i koden.

```
: (?branch) ( flag -- )
  0= dup r@ @ and           ( If true then branch using address)
  swap not r> cell+ and     ( Else skip inline address)
  or >r ;                   ( Compose new return address)
```

Nu har vi alla byggstenarna för att kunna konstruera kontrollstrukturerna i Forth. Fyra hjälpprimitiver används för att abstrahera markering och upplösning av hoppadresser:

```
: >mark ( -- addr) here 0 , ;
: >resolve ( addr -- ) here swap ! ;
: <mark ( -- addr) here ;
: <resolve ( addr -- ) , ;
```

Först villkorsstrukturen som kommer att kompilera hopp framåt i koden:

```
: if ( flag -- ) compile (?branch) >mark ; immediate
: else ( -- ) compile (branch) >mark swap >resolve ; immediate
: then ( -- ) >resolve ; immediate
```

Sist gruppen av iterativa konstruktioner:

```
: begin ( -- ) <mark ; immediate
: while ( flag -- ) compile (?branch) >mark ; immediate
: repeat ( -- ) compile (branch) swap <resolve >resolve ; immediate
```

Sammanställning

Nu har vi byggt upp de grundläggande operationerna i språket från en mycket begränsad uppsättning primitiver. För de definitioner ovan har endast följande primitiver används; `1+ 0= nand @ ! >r r>`. Totalt krävs *sju primitiver* plus två för att hantera anrop och retur från procedurer (`exit`).

Man kan observera ett antal intressanta egenskaper hos denna uppsättning. För det första saknas hopp som en primitivoperation. Denna kan, som har visats, realiseras med minnesaccess och stackfunktioner.

Den minimala arkitekturen för Forth behöver alltså inte erbjuda hoppinstruktioner. Dessa kan byggas av de nio primitiverna. Stackoperationer kan också byggas. Detta lämnas oss med en mycket liten och symmetrisk lista primitiver.

Det skall påpekas att den lista av primitiver kan lätt utvidgas och härigenom ge bättre realiseringar. Genom att låta addition och logiskt shift vara primitiver kan multiplikation och division omskrivas till mer effektiva former. Den basuppsättning som här presenteras bygger på minimal hårdvara. Mer om detta i avsnittet om hårdvarurealisering.

3. PRIMITIVA OPERATIONER

Efter analysen av de operationer som behövs är den slutgiltiga listan följande åtta operationerna:

```
1+ ( x -- y)
0= ( x -- flag)
nand ( x y -- z)

>r ( x -- )
r> ( -- x)

@ ( addr -- x)
! ( x addr -- )

exit ( -- )
```

En nionde primitiv uppstår för att hantera subrutinsanrop (kolondefinitioner). Dessa primitiver kan nu beskrivas med hjälp av ett registeröverföringsspråk[†] (RTL) för att ge oss mer information om möjliga hårdvarurealiseringar av den inre strukturen av en minimal Forthmaskin. Genom att studera de grundläggande transaktionerna mellan maskinens inre enheter kan vi härleda en lämplig sammankoppling [8] som erbjuder önskad nivå på kostnad, prestanda, parallellism etc. Vi måste först namnge olika register, stackar och minnet i systemet:

tos	top of parameter stack register
ps	parameter stack
ip	instruction pointer
rs	return stack
ir	instruction register
ma	memory address port
md	memory data port
mm	main memory

Man kan observera att operationerna är dels med en parameter (unär) och dels med två parametrar (binär). *Ett-plus* och *noll-lika-med* kan beskrivas som följande registeröverföring:

```
unary ( x -- y)      function1(tos)  -> tos

1+ ( x -- y)        tos + 1 -> tos
0= ( x -- y)        tos = 0 -> tos
```

De binära operatorerna använder både toppregistret (tos) och parameterstacken (ps). Operationerna har följande allmänna form:

```
binary ( x y -- z)   function2(pop(ps), tos) -> tos
```

[†] Registeröverföringsspråket som har använts har följande syntax:

```
<source> -> <destination> , <parallel assignment>
<source> -> <destination> ; <sequential assignment>
<source> och <destination> är register, stack, port eller minne.
```



```
nand ( x y -- z)      nand(pop(ps), tos) -> tos
```

Den minimala uppsättningen med instruktioner kan enkelt utvidgas med andra unära och binära operatörer utan någon speciell hantering. De operationer som nu återstår att beskriva i registeröverföringsform har med transporten av data mellan de olika enheterna; Till och från minnet och returadresstacken.

```
>r ( x -- )      pop(ps) -> tos, tos -> push(rs)
r> ( -- x )      pop(rs) -> tos, tos -> push(ps)

@ ( addr -- x)   tos -> ma, mm[ma] -> md, md -> tos
! ( x addr -- )  tos -> ma, pop(ps) -> md, md -> mm[ma]; pop(ps) -> tos
```

Att tilldela minnet givet en adress och data på parameterstacken kommer att kräva två faser. De övriga operationerna kan göras på en klockcykel. Nu återstår bara de två grundläggande kontrollprimitiverna; anrop och retur från subrutin:

```
call ( -- )      ir -> ip, ip -> push(rs)
exit ( -- )      pop(rs) -> ir
```

För att fullständigt beskriva den minimala Forth processorn krävs till sist att den övergripande instruktionshämtning och exekvering definieras på registeröverföringsform:

```
fetch ( -- )      ip -> ma, mm[ma] -> ir <phase1>
perform ( -- )    do(ir), ip + 1 -> ip <phase2>
```

Den utförande fasen (perform) kräver att operationen avkodas och exekveras av maskinen (do). Instruktionspekaren (ip) har valts att inkrementeras parallellt med den utförande fasen. Detta medför att hårdvarukraven för denna del av processorn minimeras.

4. HÖGNIVÅOPERATIONER

Högnivåoperationerna har beskrivits med kolondefinitioner men några eftertankar måste gå till hur den innersta adressinterpretatorn kommer att arbeta. Om man vill undvika detta helt kan man enkelt ge kolon denna definition:

```
: : ( -- ) create ] does> >r ;
```

Det vill säga en högnivåoperation är en kolondefinition som vid exekvering tar referensen till dess kropp (body) och anser att detta är returadressen. Detta medför att kolondefinitionen utförs. Vidare definieras VARIABLE och CONSTANT som följande högnivåoperationer.

```
: variable ( -- ) create 0 , does> ;
: constant ( x -- ) create , does> @ ;
```

I en virtuellmaskin måste trådning kunna separeras från maskininstruktioner. En metod att åstadkomma detta är att igenkänna det faktum att i ett teckenadresserat (byte) minne kommer trådar att vara adress med jämn minnesadress. Härigenom kan detta användas som en signal till den innersta interpretatorn vad som är en subrutinsanrop. Kort sagt adresser är instruktioner för subrutingsanrop.

Av utrymmesskäl och då det inte är av vikt för den minimala instruktionsuppsättningen för Forth har jag valt att inte beskriva hantering av CREATE och DOES>. För den vetgirige kan bilagan hjälpa då jag där presenterar en Forthsimulator av den minimala maskinen.

5. HÅRDVARUREALISERING

Den virtuella maskinen kräver fem funktionsmoduler; 1) topp av parameterstacksregister (tos), 2) parameterstack, 3) instruktionspekare (ip), 4) returadresstack och 5) primärminne. Vid en hårdvarurealisering av de ovan nämnda primitiverna, kan man genom att analysera hur data flyttas

mellan de olika enheterna i den virtuella maskinen dra slutsatsen att en tvåbusskonstruktion räcker då endast två flöden av data sker samtidigt. Två portar krävs mellan processor och minnet; en för minnesadressen (ma) och en för att transportera data till och från minnet (md).

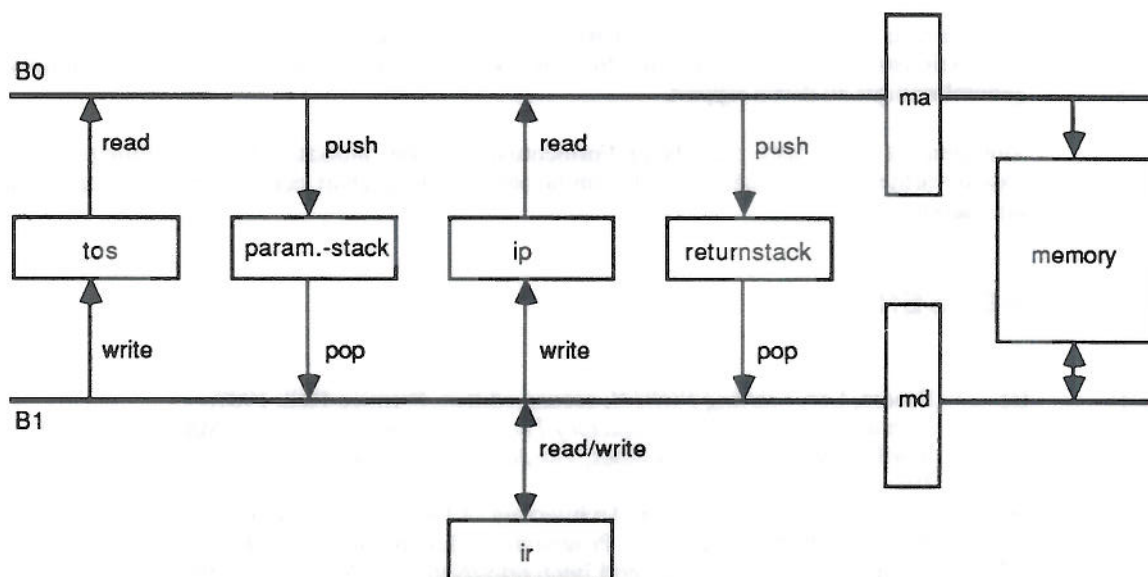


Fig. 1: En möjlig hårdvarustruktur för en dual stacksarkitektur

Genom tvåbusskonstruktionen kan de flesta av de primitiva operationerna genomföras inom en klockcykel inklusive instruktionshämtningen. Unära operationer (en parameter) utförs direkt på toppregistret (tos). Binära operationer kräver att data hämtas från parameterstacken. I och med denna grundkonstruktion kan vilken unär- eller binäraoperation görs om till en hårdvaruprimitiv genom att realisera funktionen i hårdvara kring toppregistret (tos). Men det räcker med bara 1+, 0= och NAND till att börja med.

För att nå bättre prestanda kan man undersöka frekvensen av användningen av de olika operationerna i olika Forthapplikationer. Naturligtvis kommer det att visa sig att stack-, hopp- och aritmetriska operationer såsom plus och minus har mycket hög frekvens av användning och bör därför realiseras i hårdvara. I föregående kapitel beskrevs den generella hanteringen av unära och binära operationer i arkitekturen. Det räcker med att utöka logiken som kringgärdar toppregistret (tos) för att uppnå högre prestanda för dessa typer av operationer.

Den valda framställningen är inte fulländad eller optimal. Andra definitioner av t.ex. multiplikation med hjälp av shift och addition ger bättre prestanda. Återigen medför dessa typer av omdefinitioner endast små lokala förändringar i Fortharkitekturen.

SLUTSATSER

Genom att dissekera de traditionella operationerna i Forth har jag visat att dessa kan realiseras med en mycket liten uppsättning primitiver. Målsättning har varit att dels finna den minimala mängden dels att specificera en hårdvarumodell för realisering av dual stack arkitekturer som kan utvidgas beroende på applikationsområde. Genom det minimala kravet på hårdvara (grindar) kan detta kontrollmaskineri användas i många tillämpningar där det annars inte skulle vara möjligt att välja en processor för kontrolluppgiften. Den förenklade konstruktionen kan mycket lätt utvidgas för att uppnå högre nivå av parallellism genom t.ex. uppdelning av bussarna. Härigenom kan gränsen, en instruktion per klockcykel, brytas.

TILLKÄNNAGIVANDE

Jag vill rikta ett stort tack till Mitch Bradley och Peter da Silva för flera förslag som medförde att instruktionsuppsättningen blev mindre. Ett speciellt tack till Per Alm för sina kommentarer och genomläsningar av denna rapport.

Slutligen ett varmt tack till alla Er Forthentusiaster som motsätter Er påtryckningarna från C och Pascal-världen och tror på ett språk som är oändligt utbyggbart och med en omedelbar och intensiv interaktion.

REFERENSER

- [1] Brodie, Leo, *Starting FORTH*, second edition, Prentice-Hall, 1987.
- [2] Krishnamurthy, E.V., *Introductory Theory of Computer Science*, Macmillan, 1983.
- [3] Starling, M.K., Forth Machines, *The Journal of Forth Application and Research*, Vol.2, No.1, 1984.
- [4] Hayes, J.R., Lee, S.C., The Architecture of FRISC 3: A Summary, *Proc. of the 1988 Rochester Forth Conference*, Programming Environments, 1988.
- [5] Pittman, T., Two-Level Hybrid Interpreter/Native Code Execution for Combined Space-Time Program Efficiency, *Proc. of the SIGPLAN'87 Symposium Interpreters and Interpretive Techniques*, 1987.
- [6] Hallberg, T.-J., *Hur datorer fungerar*, Studentlitteratur, 1979.
- [7] Danielsson, P.-E., Bengtsson, L., *Digital teknik*, Studentlitteratur, 1986.
- [8] Hwang, K., Briggs, F.A., *Computer Architecture and Parallel Processing*, McGraw-Hill, 1985.
- [9] Brodie, Leo, *Thinking FORTH, A Language and Philosophy for Solving Problems*, Prentice-Hall, 1984.

BILAGA

Nedan följer en realisering av den härledda minimal maskinen för Forth i Forth själv. Kod har bl.a. använts för att simulera och verifiera arkitekturen.

```
.( Loading Minimal Forth Machine definitions...) cr

\ An implementation of a Minimal Forth Machine
\ Written by Mikael Patel, Copyright (C) 1990.
\ Started on: 10 December 1989
\ Last updated: 15 January 1990
\ Dependencies: (TILE Forth 2.48) none

vocabulary minimal

minimal definitions

forth

\ Hardware Devices: Registers and Stacks
: register ( -- | Create a register variable)
  create 0 , ( Create symbol and initiate)
  does> @ ; ( Fetch register value)

: .register ( x -- | Print value of register)
  . ; ( Print as a number)

: -> ( x -- | Assign operator for registers)
  ' >body ( Access preceding symbol)
  [compile] literal ( May compile as literal)
  compile ! ; immediate ( and store value to body)

: stack ( n -- | Create a stack with n-depth)
  create ( Create symbol for stack)
  here swap 2+ cells allot ( Allocate stack area)
  here over cell + ! ( Initiate the stack bottom)
  here swap ! ; ( Initiate the stack pointer)

: push ( x s -- | Push value onto stack)
  cell negate over +! @ ! ; ( Decrement and store value)

: pop ( s -- x | Pop value from stack)
  dup @ @ cell rot +! ; ( Fetch value and increment)

: .stack ( s -- )
  dup cell + @ swap @ ( Fetch stack pointer and bottom)
  ?do i @ . cell +loop ; ( Fetch values and display)

\ Forth Machine Registers
register ir ( Instruction register)
register ip ( Instruction pointer)
16 stack rp ( Return address stack)
register tos ( Top of stack register)
16 stack sp ( Parameter stack)

\ Instruction execution trace flag
variable trace
```

```

\ Print machine state
: .registers ( -- | Display the machine state)
  ." ir: " ir .name space      ( Print name of instruction)
  ." ip: " ip cell - .register  ( Print value of instr. pointer)
  ." rp: " rp .stack           ( Print contents of return stack)
  ." tos: " tos .register      ( Print value of top of stack)
  ." sp: " sp .stack cr ;      ( Print contents of param. stack)

\ Forth Machine Instructions
: instruction ( -- | Create a instruction symbol)
  create ;

: code ( -- x | Compile instruction code)
  minimal      ( Select from minimal vocabulary)
  [compile] ['] ( Compile symbol value)
  forth ; immediate ( Go back to the forth vocabulary)

\ The list of possible instructions
instruction 1+
instruction 0=
instruction NAND
instruction >R
instruction R>
instruction !
instruction @
instruction EXIT
instruction HALT

: CALL ( -- )
  ip rp push      ( Push instruction pointer)
  ir >body -> ip ; ( And fetch new value)

\ The Minimal Forth Machine
: fetch-instruction ( -- ir)
  ip @ dup -> ir      ( Fetch the next instruction)
  ip cell + -> ip ;   ( Increment instruction pointer)

: processor ( -- | The virtual machine)
  begin
    fetch-instruction
    trace @ if .registers then
      case
        code 1+ of tos 1+ -> tos      endof
        code 0= of tos 0= -> tos      endof
        code NAND of sp pop tos and not -> tos endof
        code >R of tos rp push sp pop -> tos endof
        code R> of tos sp push rp pop -> tos endof
        code ! of sp pop tos ! sp pop -> tos endof
        code @ of tos @ -> tos      endof
        code EXIT of rp pop -> ip    endof
        code HALT of true abort " HALT" endof
      CALL
    endcase
  again ;

: run ( addr -- | Run the virtual machine)
  -> ip      ( Assign instruction pointer)
  0 -> tos  ( Initiate top of stack)
  ." RUN" cr ( And run the processor)
  processor ;

```

\ A simple compiler for the Minimal Forth Machine

minimal

```
: CREATE ( -- ) create ;
: COMPILE ( -- ) compile compile ; immediate

: DEFINE ( -- ) CREATE ] ;
: END ( -- ) COMPILE EXIT [compile] [ ; immediate
: BLOCK ( n -- ) cells allot ;
: DATA ( -- ) , ;
```

\ Variable management

```
DEFINE [VARIABLE] ( -- addr) R> END
: VARIABLE ( -- addr) CREATE COMPILE [VARIABLE] 1 BLOCK ;
```

\ Constant management

```
DEFINE [CONSTANT] ( -- n) R> @ END
: CONSTANT ( n -- ) CREATE COMPILE [CONSTANT] DATA ;
```

\ Basic stack manipulation functions and some utility functions

VARIABLE T

```
DEFINE T! ( x -- ) T ! END
DEFINE T@ ( -- x) T @ END
```

```
DEFINE DROP ( x -- ) T! END
DEFINE DUP ( x -- x x) T! T@ T@ END
DEFINE SWAP ( x y -- y x) T! >R T@ R> END
DEFINE ROT ( x y z -- y z x) >R SWAP R> SWAP END
DEFINE OVER ( x y -- x y x) >R DUP R> SWAP END
DEFINE R@ ( -- x) R> R> DUP >R SWAP >R END
```

\ Logical function

```
DEFINE BOOLEAN ( x -- flag) 0= 0= END
DEFINE NOT ( x y -- z) DUP NAND END
DEFINE AND ( x y -- z) NAND NOT END
DEFINE OR ( x y -- z) NOT SWAP NOT NAND END
DEFINE XOR ( x y -- y) OVER OVER NOT NAND >R SWAP NOT NAND R> NAND END
```

\ Relation operations

-2147483648 CONSTANT MININT (-- x | 32-bit minimum integer value)

```
DEFINE 0< ( x -- flag) MININT AND BOOLEAN END
DEFINE 0> ( x -- flag) DUP 0< SWAP 0= OR NOT END
DEFINE = ( x y -- flag) XOR 0= END
```

\ Primitive arithmetic functions

```
DEFINE 1- ( x -- y) NOT 1+ NOT END
DEFINE 2+ ( x -- y) 1+ 1+ END
DEFINE 2- ( x -- y) NOT 1+ 1+ NOT END
```

\ Cell sizes and functions

4 CONSTANT CELL

```
DEFINE CELL+ ( x -- y) 1+ 1+ 1+ 1+ END
```


\ Branch instructions

```
DEFINE (BRANCH) ( -- ) R> @ >R END
DEFINE (?BRANCH) ( flag -- ) 0= DUP R@ @ AND SWAP NOT R> CELL+ AND OR >R END
```

\ Compiler functions

forth

```
: >MARK ( -- addr) here 0 , ;
: >RESOLVE ( addr -- ) here swap ! ;
: <MARK ( -- addr) here ;
: <RESOLVE ( -- addr) , ;
```

minimal

```
: IF ( flag -- ) COMPILE (?BRANCH) >MARK ; immediate
: ELSE ( -- ) COMPILE (BRANCH) >MARK swap >RESOLVE ; immediate
: THEN ( -- ) >RESOLVE ; immediate
: BEGIN ( -- ) <MARK ; immediate
: WHILE ( flag -- ) COMPILE (?BRANCH) >MARK ; immediate
: REPEAT ( -- ) COMPILE (BRANCH) swap <RESOLVE >RESOLVE ; immediate
: UNTIL ( flag -- ) COMPILE (?BRANCH) <RESOLVE ; immediate
```

\ Simple arithmetrical functions

```
DEFINE U+ ( x y -- z) BEGIN DUP WHILE 1- SWAP 1+ SWAP REPEAT DROP END
DEFINE U- ( x y -- z) BEGIN DUP WHILE 1+ SWAP 1- SWAP REPEAT DROP END
DEFINE NEGATE ( x -- y) NOT 1+ END
DEFINE + ( x y -- z) DUP 0< IF U- ELSE U+ THEN END
DEFINE - ( x y -- z) NEGATE + END
```

\ More relation functions

```
DEFINE < ( x y -- flag) - 0< END
DEFINE > ( x y -- flag) - 0> END
```

\ Some test code just to show that it works

```
4 CONSTANT X
2 CONSTANT Y
```

```
DEFINE TEST ( -- )
  X Y + Y > HALT
END
```

TEST run



Arbetsinr., institution, fakultet
Division, department, faculty

Department of Computer and
Information Science
Institutionen för datavetenskap

ISBN:	
ISSN:	ISSN-0281-4250
Rapportnr: Report no:	LiTH-IDA-R-90-02
Upplegans storlek: Number of copies:	
Datum: Date:	Januari 1990

Projekt: Project:	
Titel: Title:	Vägen mot en minimal Forth arkitektur
Författare: Author:	Mikael Patel

Uppdragsgivare: Commissioned by:	
Dnr: Call no:	

Rapporttyp: Kind of report:	<input type="checkbox"/> Examensarbete/Final project <input type="checkbox"/> Delrapport/Progress report <input type="checkbox"/> Reserapport/Travel report <input type="checkbox"/> Slutrapport/Final report <input checked="" type="checkbox"/> Övrig rapport/Other kind of report
--------------------------------	--

Rapportspråk: Language:	<input checked="" type="checkbox"/> Svenska/Swedish <input type="checkbox"/> Engelska/English <input type="checkbox"/> _____
----------------------------	--

Sammanfattning (högst 150 ord): Abstract (150 words):	<p>Sammanfattning: Denna rapport visar att ett trådat programmeringsspråk som Forth kan realiseras med ett mycket litet antal primitiva operationer. Genom att successivt bryta ner operationerna i Forth kan man finna en minimal uppsättning som också är minimal med avseende på antal grindar vid realisering i hårdvara. Den föreslagna hårdvaruarkitektur kan med hjälp av en enkel metodik lätt utvidgas beroende på krav på prestanda. Den grundläggande hårdvarukonstruktionen kvarstår och endast nya operationer introduceras på hårdvarunivån. Man får härigenom en flytande gräns mellan hård- och mjukvara. En genomgång av de aritmetiska operationerna och kontrollstrukturerna i Forth genomförs detaljerat för att visa hur man kan finna den minimala uppsättning primitiver som krävs för att realisera Forth.</p>
Nyckelord (högst 8): Keywords (8):	Trådade programmeringsspråk, Forth-83, Datorarkitektur, Stackarkitekturer, Minimala instruktionsuppsättningsdatorsystem

Bibliotekets anteckningar

IDA - institutionen för datavetenskap vid Universitetet och Tekniska högskolan i Linköping

omfattar ämnena *administrativ databehandling*, *datalogi*, och *telesystem*, med undervisning främst inom civilingenjörsutbildningen, datavetenskaplig linje och systemvetenskaplig linje. Inom IDA finns en bred forskningsverksamhet och ett forskarutbildningsprogram som leder fram till licentiat- och/eller doktors-examen. Institutionens forskning är organiserad i grupper (laboratorier), som bl.a. arbetar inom områdena algoritmkomplexitetsteori, applikationssystem, datorstöd för automation, CAD för elektronik, biblioteks- och informationssystem, logik-programmering, databehandling av naturligt språk, programmeringsmiljöer, kunskapsrepresentation i logik samt administrativ databehandling.

Industriserien

är en serie forskningsrapporter som särskilt vänder sig till den som arbetar med program- och maskin-varufrågor i industriell miljö eller inom den offentliga sektorns databehandling. I serien ingår dels rapporter från forskningsprojekt i Linköping som är av särskilt intresse för praktiker, dels översikter över aktuell forskning i världen i övrigt. Dessutom ingår presentationer av målsättning och strategi för institutionens forskning.

Hittills har utkommit:

- 1990** **Mikael Patel:** Vägen mot en minimal Forth arkitektur. (LiTH-IDA-R-90-02)
- 1989** **Anders Törne:** Computer Assistance in Automation. (LiTH-IDA-R-89-51)
Bengt Lennartsson: Industriell programvaruteknik - en programförklaring. (LiTH-IDA-R-89-43)
Arja Vainio-Larsson, Peter Åberg: Utvärdering av människa-maskin gränssnitt (Delrapport II). (LiTH-IDA-R-89-31)
Arja Vainio-Larsson: Hypertext/hypermedia från idé till praktisk tillämpning. (LiTH-IDA-R-89-15)
Olof Johansson: A Perspective on Engineering Database Research. (LiTH-IDA-R-89-14)
- 1988** **Mikael R.K. Patel:** The State of the Art of High Level Synthesis: Some Reflections from the 25th ACM/IEEE Design Automation Conference. (LiTH-IDA-R-88-40)
Tomas Sokolnicki: Intelligent Tutoring Systems - Craft or Technology? (LiTH-IDA-R-88-33)
Johan Hultman, Anders Nyberg: Realizing Action Plans and Response Rules in a System Tool for an Autonomous Vehicle. (LiTH-IDA-R-88-24)
Arja Vainio-Larsson, Rebecca Orring, Peter Åberg: Metoder för utvärdering av människa-maskin gränssnitt: En litteraturstudie. (LiTH-IDA-R-88-20)
Jonas Löwgren: History, State and Future of User Interface Management Systems. (LiTH-IDA-R-88-14)
- 1987** **Arne Jönsson:** Naturligt språk för användardialog och databasförfrågningar. En sammanfattning av ett föredrag vid konferensen *Effektiv människa - dator-interaktion*, Linköping, 18-19 augusti 1987. (LiTH-IDA-R-87-25)
- 1985** **Harold W. Lawson Jr.:** Swedish Participation in the Malaysian National Microelectronics Programme Also in *Proc. of ERSA (Economic Relations with Southeast Asia) Symposium*, Stockholm, October 21-25, 1985. (LiTH-IDA-R-85-11)

IDA - institutionen för datavetenskap
vid Universitetet och Tekniska högskolan i Linköping

Tidigare rapporter i industriserien:

- 1985 **Harold W. Lawson, Jr.:** Sabbatical Report. (LiTH-IDA-R-85-05)
Sture Hägglund, Henrik Nordin, Roland Rehnert, Kristian Sandahl: Utveckling av kunskapsbaserade applikationssystem i samverkan högskola - näringsliv (LiTH-IDA-R-85-09)
- 1984 **Erik Sandewall:** Fjärde generationens Programvaruutbildning. (LiTH-IDA-R-84-13)
Michael Pääbo: CAD-elektronik idag och i framtiden. (LiTH-IDA-R-84-03)
Bengt Lennartsson: Programvarumiljöer. Produktionsteknik för programvara i Ada och andra språk. (LiTH-IDA-R-84-01)
- 1983 **Erik Sandewall:** Datavetenskaplig utvecklingsmiljö och kunskapsöverföringsprogram. (LiTH-IDA-R-83-10)
Sture Hägglund: Kunskapsbaserade expertsystem. Ny teknik för applikationsutveckling i nästa generations programvarusystem. (LiTH-IDA-R-83-07)
Pär Emanuelson: Programtransformationer (LiTH-IDA-R-83-06)
Mats Lenngren et al.: Datorgrafikdagar 7-9 juni 1983. (LiTH-IDA-R-83-05)
Michael Pääbo: Introduktion till datorgrafik. (LiTH-IDA-R-83-02)
- 1982 **Software Systems Research Center. Progress Report 1982.**
(Översikt över forskningsprojekt, personal och publikationer vid Datalogicentrum, LiTH.)
Hans Grunditz, Uwe Hein, Erik Tengvald: Artificiell intelligens i framtidens CAD/CAM-system (LiTH-MAT-R-82-32)
(Föredrag vid de nordiska CAD/CAM-dagarna, Göteborg, november 1982)
Anders Haraldsson: INTERLISP - en avancerad integrerad programmeringsomgivning för LISP-språket. (LiTH-MAT-R-82-29)
(Föredrag vid Nord-Data 82 i Göteborg, juni 1982.)
Sture Hägglund: Informationshantering i det elektroniska kontoret. (LiTH-MAT-R-82-27, ingår även i rapportserien från LIBLAB)
(Föredrag vid den 5:e Nordiska IoD-konferensen, Trondheim, juni 1982)
Erik Sandewall: Ny teknik i kontorsdatasystem. (LiTH-MAT-R-82-17)
(Föredrag vid Nord-Data 82 i Göteborg, juni 1982.)
- 1981 **Software Systems Research Center. Progress Report 1981.**
Dan Strömberg: Datorn - hjälpreda eller hot i det lilla företaget? (LiTH-MAT-R-81-06)
Peter Fritzson: Distribuerad PATHCAL: Förslag till ett distribuerat interaktivt programmeringssystem för PASCAL. (LiTH-MAT-R-81-05)
Sture Hägglund m fl: 80-talets elektroniska kontor: Erfarenheter från LOIS-projektet. (LiTH-MAT-R-81-04)
Ola Strömfors, Lennart Jonesjö: The Implementation and Experiences of a Structure-Oriented Text Editor. (LiTH-MAT-R-81-03)
samt
Ola Strömfors: ED3 - Användarhandledning.
- 1980 **Kenth Ericson, Hans Lunell:** Redskap för kompilatorframställning (LiTH-MAT-R-80-39)
Software Systems Research Center. Preliminary Report 1980.