HITAC 5020 ソフトウエアシステム(2)
HARP 5020

中田育男，高須昭輔，稲富　彬

1. HARP 5020言語としてはIBM社のFORTRAN IV言語をそのまま採用しているが，さらにHITAC 5020の機能を生かす為と，debugging を容易にするために，いくつかのstatementを追加してある．

ここではHITAC 5020のA register (accumulator相当)とB register (index register相当)とをHARP 5020でどのように生かしたかということ，とdebugging statementとしてどのようなものを用意しているかということを略述する．

2. HITAC 5020の2番地から15番地まではA registerと呼ばれるaccumulatorの機能を持つた遅延線registerである．そのうち2番地から7番地まではB registerと呼ばれ，addressのmodifierとしての機能も持つている．

HARP 5020のobject programではこのB registerをDo loopのcontrolとaddress modify及びaddress計算のために使用し8番地から15番地までのA registerを数式演算用のaccumulator として使用する．

このようにaddress演算と数式演算とが独立のregisterで行なえることはcompilerのobject programにとつて大変都合が良い．

例えばarray elementに関する数式演算をしているとき，途中であるarray elementの所存番地を知るためのaddress演算が必要になつても数式演算の途中の結果を一時格納する必要がない．

さらに数式演算用のaccumulator がいくつかあることは演算の途中の結果を一時格納する手間を減らすことになる．

例えばaccumulatorが一つしかない計算機の場合は

$$A=(B+C)/(D+E)$$

という数式を左からscanしてcompileすればobject programは次のよう

になるであろう．

```
CLA   B
ADD   C
STO   WS1     B+C→WS1
CLA   D
ADD   E
STO   WS2     D+E→WS2
CLA   WS1
DIV   WS2     WS1/WS2→A
STO   A
```

(9 steps)

上の例を右からscanすれば

```
CLA   E
ADD   E
STO   WS1
CLA   C
ADD   B
DIV   WS1
STO   A
```

(7 steps)

となつて，少し短かくなる．

HITAC 5020の場合は左からscanしても右からscanしても変りないが，今，左からscanしたとすれば

```
CAF   A1, B
AF    A1, C     (B+C)→A1
CAF   A2, D
AF    A2, E     (D+E)→A2
DF    A1, A2    (A1)/(A2)→A1
TRE   A1, A
```

(6 steps)

となつて6 stepsですむ.

[注]

CAFはClear Add Floatingのmnemonic code.

A1はA1というA registerを示す.

AFはAdd Floating.

DFはDivide Floating.

TRFはTransfer Rounded Floating.

さらにもう一つ

A=(B*C+D*E)/(F+G)

という数式を例にとつてみるとaccumulatorが一つしかない計算機の場合は左からscanして13 steps, 右からscanして11 stepsかかる. accumulatorが二つある計算機の場合は左からscanして9 steps, 右からscanして10 stepsかかるが, accumulatorが三つあれば左からscanしても9 stepsですむ.

HARP 5020では8番地から15番地までの8 wordのA registerを, single length integerの演算のときは4ケのaccumulatorとして使い, double length integerのときは2ケのaccumulatorとして使い, single及びdouble precision floatingのときは4ケのaccumulatorとして使い, さらにsingle及びdouble precision complexのときには一組のcomplex accumulatorとして使うことによつて演算速度を速くしている.

3. compilerを作るときまず第一に考えなければならないことは, いかにして能率の良いobject programを作り出すかということであろう. その場合一番問題になるのはloopを如何にして速くまわすかということである. loopを速くまわすためにはloopのcontrolを能率よくすることも大事であるが, loopというものは多くの場合arrayに関する演算をするとき使われるから, Do loopの中でarray elementの番地をどのようにして作り出すかということが大きな問題になる.

今

DIMENSION A(10,15) B(15,12)

としたときmemoryの割りつけをA(1,1), A(2,1),(3,1), A(10,1), A(1,2),A(2,2),……,A(9,15),A(10,15)とcolumn wiseにしたとすればA(I,J)の番地は

A(0,0)の番地＋I＋10×J

として求められるが，このaddress演算をどこでやるかが問題になる．

例えば

```
Do 10  I=1,10
.......
Do 10  J=1,12
.......
Do 10  K=1,15
.......
X=…… +A(I,K)*B(K,J)
```

というloopを速くまわすためには最小loopであるKのloopを速くしなければならない．そのためには

A(I,K)*B(K,J)

の計算を

CAF A1, A_{00} +I#B_1

MF A1, B_{00} +J×12#B_2

のかたちにすればよい．ただしここでA_{00}，B_{00}はそれぞれA(0,0), B(0,0)の番地であり，#B_1はB register B1によつてaddress partをmodifyすることを示す．B1にはK×10，B2にはKの値を入れておけばよい．そしてloopを1回まわるたびにB1の内容を10増やし，B2の内容を1増やして，B2の内容が15になるまでKのloopをまわすのである．しかしこの方法ではIのloopでIの値が変化したときA_{00}＋Iを計算して上の命令の番地部にうめ込みJのloopでJの値が変化したときB_{00}＋J×12を計算して上の命令の番地部にうめ込んでおかなければならない．

そこでもう一つの方法として番地部にはA_{00}及びB_{00}を入れておき B registerにI+K×10とJ×12+Kを入れておく方法が考えられるが，この

方法ではB registerの増減のしかたとloopの判定のしかたが難しい.

これらの方法を使えばたしかに最小loopは速くなるがcompilerの負担が大きい. またこれらの方法で一番能率を良くするためには比較的多くのB registerを必要とする.

そこでcompilerの負担を小さくし, しかもobject programの能率をあまり落さない方法として, 次のような方法を採用した.

それはaddress演算の際必要になる掛算をcompiling timeに実行して表にしておき, object timeにはその表をひいてaddress partに加えこむようにする方法である.

例えば前の例で

CAF A1,A(I,K)

とするところにはA'+1番地以降に

0,10,20,·····,140

を入れておき

MNI ～,A'#B1

CAF A1,A_{01} #B2

とする. ここでMNIはmodify next instructionという命令であつて, この命令のeffective addressで示される番地の内容を次の命令のaddress (modifyをすませたあとのaddress)に加え込めという命令である. 今B1にKの値, B2にIの値を入れておけば

A_{01} #B2 は

A(0,1)の番地+I

であり, それにMNI命令によつてA'+K番地の内容, すなわち(K−1)×10が加えられるから結局CAF命令のeffective addressは

A(0,1)の番地+I+(K−1)×10=A(I,K)の番地

となる.

三次元のarrayについてもMNI命令のa part (A registerの指定をするところ)とindirect addressingの機能を使うことによつてMNI命令1つですませることができる.

例えば

DIMENSION A(10,12,15)

でA(I,J,K)番地の内容をとり出すためにはIの値がB register B1に，JがB2に，KがB3に入つていれば

A′+1番地以降に

0,10×12,2×10×12,・・・・・,14×10×12

A″+1番地以降に

A_{011},A_{011}+10,A_{011}+2×10,・・・・・,A_{011}+11×10

という表を用意しておいて

MNI B1,A′#B3

CAF A1,A″#B2*

とすればよい．ここで*はindirect addressingを示す．

A″#B2*によつてA″+J番地の内容A_{011}+(J−1)×10がとり出され，それにMNI命令によつて，B1の内容IとA′#B3番地の内容(K−1)×10×12が加えられ結局effectine addressは

A_{011}+I+(J−1)×10+(K−1)×10×12=A(I,J,K)の番地

となるのである．

4. HARP言語でプログラムを書けばもつと機械語に近い言語で書く場合よりはるかにdebuggingはらくになるが，しかしそれでも現在多くのmachine timeがdebuggingのために使われている．そこでdebugging が能率よく出来るようにするためにHARP 5020には次のようなdebugging statementを設けた．

(1) DEBUG VALUES

DEBUG VALUES statementは

DEBUG VALUES n_1, n_2 IF(e) a_1, a_2, a_3, ...

又は

DEBUG VALUES n_1, n_2 a_1, a_2, a_3, ...

n_1, n_2はstatement number，eはlogical expression，

a_iはvarible name

statement number n_1 のついた statement と statement number n_2 のついた statement の間で，a_i を左辺に持つ arithmetic statement があればその statement を実行したときにそのとき a_i に与えられた値を a_i の name と一緒に印字する．IF(e) があれば e の値が true であるときだけ印字する．

a_i が array name であるときは左辺が a_i(C*I+C′) というかたちの arithmetic statement についてその statement を実行したときに，そのとき a_i(C*I+C′) に与えられた値を a_i の name とそのときの subscript の値と一緒に印字する．

a_i が一つも書いてなければその program のすべての variable について上のことを行なう．

例えば

```
DEBUG VALUES   100,200   IF(I.GT.3) A

100 .......
    .......
    DO 20   I=1,10
    .......
20  A(I)=......
    .......
200 .......
    .......
```

とすれば I=1,2,3 のときは A(I) は印字しないが I が 4 以上のとき，例えば I=5 のときに statement number 20 のついた statement を実行すればそのあとで

DEBUG A(5)=±×.×××…×E±××

とそのときの値を印字する．

(2) DEBUG FLOW

DEBUG FLOW statement は

DEBUG FLOW n_1, n_2

n_1, n_2 は statement number

statement number n_1 のついた statement と，statement number n_2 のついた statement の間に

① unconditional GO TO statement

② computed GO TO statement

③ assigned GO TO statement

④ arithmetic IF statement

があつたらその statement を実行するときに，どこへ control が移るかを印字する．

例えば

```
DEBUG FLOW  100,200
100·····
10 GO TO 30
   ·····
200·····
```

とすれば 10 の statement を実行するときに

DEBUG GO TO 30

と印字する．

(3) DEBUG DUMP

DEBUG DUMP statement は

DEBUG DUMP n IF(e) $a_1, a_2, \cdots\cdots$

又は

DEBUG DUMP n $a_1, a_2, \cdots\cdots$

n は statement number

e は logical expression

a_i は variable name

statement number n のついた statement を実行したときに e の値が true であつたら（IF(e) がなければ常に true とする） a_i ($i=1,2,\cdots\cdots$) の値を印字する．a_i が array であつたらその array 全部の値を印字する．

(4) DEBUG TERMINATE

DEBUG TERMINATE statement は

DEBUG TERMINATE n IF(e)

又は

DEBUG TERMINATE *n*

n は statement number

e は logical expression

statement number *n* のついた statement を実行したとき（その statement が control statement であつたらそれを実行する前）に *e* の値が true であつたら（IF(*e*) がなければ常に）

DEBUG TERMINATE *n*

と印字してこの program の実行を終了する．

(5)

FUNCTION 又は SUBROUTINE subprogram に DEBUG statement があつたら program 実行の際その subprogram を call したときにまず

DEBUG xxxxx

と印字する．但し xxxxx はその subprogram name である．

「HARP5020」のその後の展開

1．HARP 5020 という名前について

FORTRAN という名前は IBM 社のものであると思って HARP という名前を付けたが，その後 1966 年に FORTRAN 言語は国際規格（FORTRAN 66 と呼ばれる）となったので，それ以後このような名前が付けられることはなくなった．

本論文では，英語の用語に対応する日本語がまだなかったので英語の単語をそのまま使っている．statement を「文」などとしたのは FORTRAN 66 の JIS 化のときである（島内先生の提案）．

2．register の使いかた

regisiter が複数個ある計算機はそれまでにはなかったし，コンパイラのアルゴリズムとしては，式を左から scan（して目的コードを生成）するよりは右から scan したほうが良い場合がある，くらいのことしか知られていなかった．この論文では，regisiter が複数個ある場合はどちらから scan してもあまり変わらないとしているが，複数個の register をうまく生かす方法が別にないか気にはなっていた．

すこし後で，この部分のコンパイラのコーディングを始める直前になって複数個の register を最適に利用する方法を思いついて，それをコンパイラに組み込むことが出来た．

その方法は，式を抽象構文木で表現し，register を多く必要とする枝の方を先に計算するように目的コードを生成する方法である．

このコンパイラは 1965 年の東京大学大型計算機センタの開所にあわせて多くの人に使われるようになった．

この仕事が一段落した後で，研究所の人間は論文も書かなければならないと言われて，この方法を C.ACM に投稿した．

Nakata, I.: On Compiling Algorithms for Arithmetic Expressions, *C.ACM*, Vol.10, No.8, pp.492-494, Aug. 1967.

この論文はいくつかの論文で引用されたが，

Sethi, R. and Ullman, J.D.: The Generation of Optimal Code for Arithmetic Expressions, *J. ACM*, vol.17, no.4, pp.715-72, 1970.

は，筆者の論文の不備な部分を補い，この方法が最適であることを示した．今でもこのアルゴリズムは使われているが，Sethi と Ullman のアルゴリズムと呼ばれている．なお，何年か後に，同様のアルゴリズムがすでに

Ershov, A.P.: On Programming of Arithmetic Operations, *C.ACM*, vol.1, no.8, pp.3-6, 1958.

で発表されていることが見つけられた．

3．配列要素のアドレス計算

本論文の方法を使えば，コンパイラの最適化としては，内側のループから B register を割り当てていくことだけすれば，それなりに効率の良い目的コードを生成できるので，コンパイルも実行も速いコンパイラを実現することが出来た．

この方法では，配列を引数として渡すとき，その配列の先頭番地と，アドレス計算用の掛算の表の両方を渡す必要がある．すなわち，配列の構造を渡す必要がある．筆者はそれが自然であると思っていたのであるが，多くのコンパイラでは先頭番地しか渡していなかった．FORTRAN の規格を作るときにそれが言語仕様として明文化されてしまったので（JIS FORTRAN を作るときは筆者は反対したのであるが），この方法は規格にあわないものとなってしまった．

マトリックスの掛算のプログラム

```
      DO 30 I = 1,L
      DO 20 J = 1,M
      C(I,J) = 0.0
      DO 10 K = 1,N
10    S = S + A(I,K)*B(K,J)
20    C(I,J) = S
30    CONTONUE
```

の最内ループの目的コードは，この方法で 8 命令とすることが出来た．同じ頃発表された IBM360 シリーズの FORTRAN G コンパイラのそれは 13 命令であったが，後に H コンパイラで 5 命令が達成された．

RISC マシンが出現したときは，命令が単純であってもコンパイラが最適化するから問題ないとされていた．最適化すれば 8 命令ですむであろうと思って筆者が調べたところ，1991 年ころは 11 命令で，1994 年ころになって 8 命令が達成されていた．2 回のアンローリングをして 7 命令というのもあった．

CP-PACS（HP 社の PA-RISC をベースに筑波大と日立で開発）では 4 回のアンローリングで 3.5 命令である（掛算と加算を一緒に行う命令がある）．