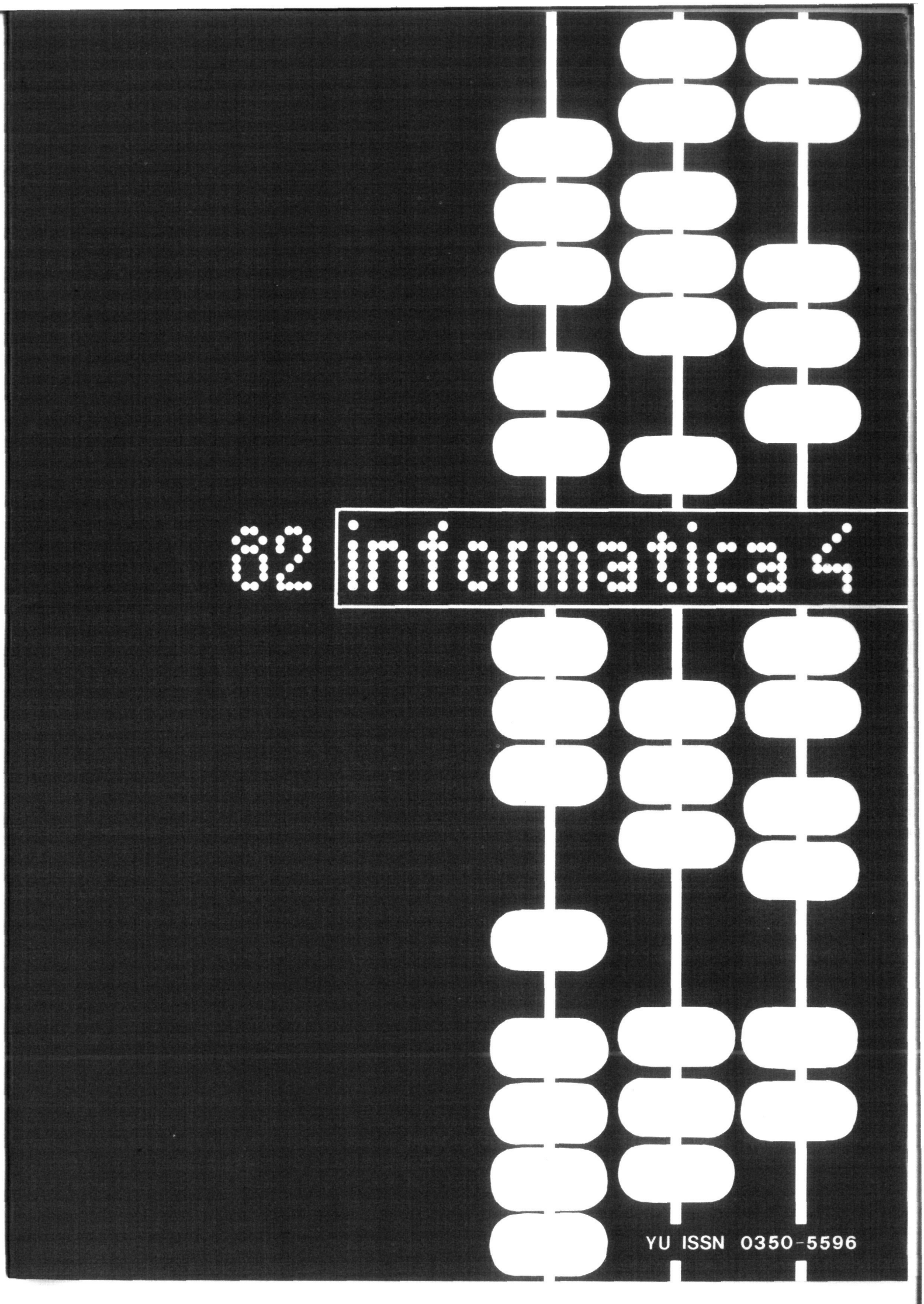
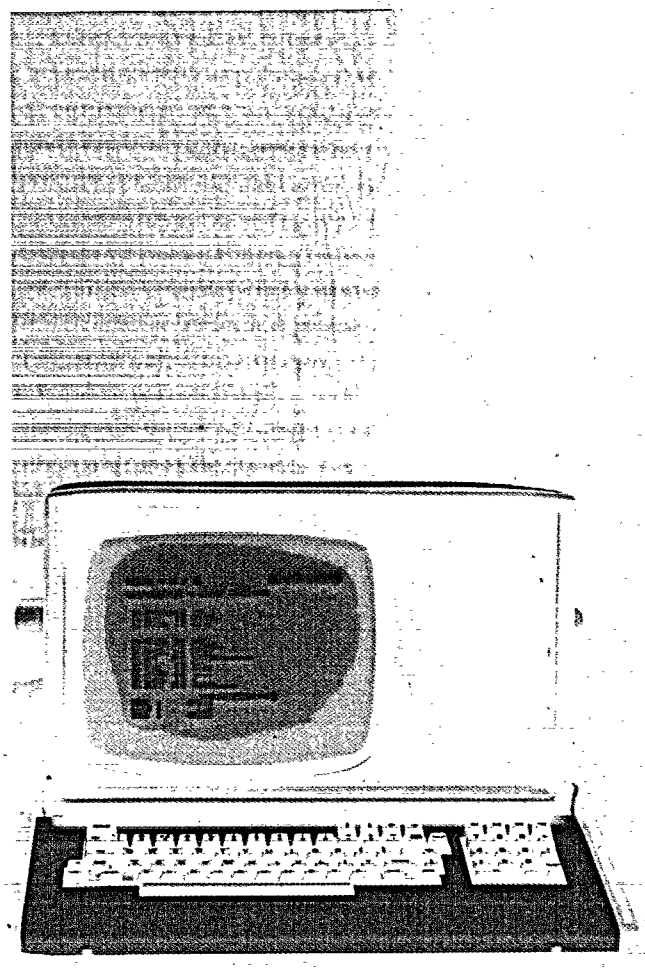


YU ISSN 0350-5596

Iskra Delta



MIPOS – UČINKOVIT, SODOBNO ZASNOVAN, STANDARDEN PAKET APLIKACIJ, KI REŠUJE NAJOBČUTLJIVEJŠE PROBLEME OBDELAVE POSLOVNIH PODATKOV, KOT STA AŽURNOST IN DOSEGLJIVOST INFORMACIJ ZA TAKOJŠNJE POSLOVNE ODLOČITVE, KI JIH NAREKUJE GOSPODARSKI TRENUTEK. NEMENJEN JE MANJŠIM DELOVNIM ORGANIZACIJAM, ORGANIZACIJSKIM ENOTAM, KJER LAHKO KOMPLEKSNO ZADOVOLJI POTREBE PO OBDELAVI POSLOVNIH PODATKOV.

VSE NADALJNJE INFORMACIJE VAM NUDI: ISKRA DELTA Tržno komuniciranje 61000 Ljubljana, Titova 52

računalniški sistemi delta®

informatica

Published by INFORMATIKA, Slovene Society for Informatics, 61000 Ljubljana, Parmova 41, Yugoslavia

Headquarters: Informatica, Parmova 41, 61000 Ljubljana, Phone: (061) 312-988, Telex: 31366 Delta

Annual subscription rate for abroad is US $ 22 for companies, and US $ 7,5 for individuals.

Printed by: Tiskarna KRESIJA, Ljubljana

DESIGN: Rasto Kirn

JOURNAL OF COMPUTING AND INFORMATICS

YU ISSN 0350–5596

VOLUME 6, 1982 – No. 4

CONTENTS

informatica

Časopis izdaja Slovensko društvo INFORMATIKA, 61000 Ljubljana, Parmova 41, Jugoslavija

Uredništvo in uprava: Informatica, Parmova 41, 61000 Ljubljana, telefon (061) 312-988, teleks: 31366 YU DELTA

Letna naročnina za delovne organizacije je 500,00 din, za redne člane 200,00 din, za študente 100,00/50,00 din, posamezne številke 100,00 din
Žiro račun št.: 50101-678-51841
Stališče uredništva se lahko razlikuje od mnenja avtorjev.

Pri financiranju revije sodeluje tudi Raziskovalna skupnost Slovenije.

Na podlagi mnenja Republiškega sekretariata za prosveto in kulturo št. 4210-44/79 z dne 1.2.1979, je časopis oproščen temeljnega davka od prometa proizvodov.

Tisk: Tiskarna KRESIJA, Ljubljana

Grafična oprema: Rasto Kirn

ČASOPIS ZA TEHNOLOGIJO RAČUNALNIŠTVA IN PROBLEME INFORMATIKE
ČASOPIS ZA RAČUNARSKU TEHNOLOGIJU I PROBLEME INFORMATIKE
SPISANIE ZA TEHNOLOGIJA NA SMETANJETO I PROBLEMI OD OBLASTA NA INFORMATIKATA

YU ISSN 0350–5596

LETNIK 6, 1982 – Št. 4

VSEBINA

RAČUNARI UPRAVLJANI TOKOM PODATAKA

SLOBODAN RIBARIĆ

TEHNIČKA VOJNA AKADEMIJA KoV JNA, ZAGREB

UDK: 681.3:519.685

U ovom članku opisane su dvije arhitekture računarskih sistema: konvencionalna arhitektura računara s upravljačkim tokom (control-flow computer) ili von Neumannova arhitektura i arhitektura računara upravljanog tokom podataka (data-flow computer). Računarski sistem na bazi von Neumannovog modela (i njegovih modifikacija) zbog svoje sekvencijalne prirode ne može efikasno iskoristiti inherentni paralelizam prisutan u problemu i postići perfomansu koja se zahtijeva na području umjetne inteligencije, obrade i raspoznavanja slika, globalnih vremenskih modela i sl. U radu je prikazan Adamsov model paralelne obrade koji predstavlja osnovu na kojoj se temelji arhitektura računara upravljanog tokom podataka. Računarski sistemi upravljani tokom podataka svojom koncepcijom omogućuju efikasno korištenje inherentnog paralelizma, a uz korištenje svih prednosti koje pruža LSI tehnologija omogućit će i dostizanje perfomasi iznad 1000 MFLOPS.

DATA-FLOW COMPUTERS. - In this paper are described two computer system architectures: a conventional control-flow computer architecture or von Neumann architecture and data-flow computer architecture. Due to its sequential nature, the computer system based on von Neumann model (and its modifications) cannot make effective use of inherent parallelism present in the problem and, in this way, achieve the performance required for the field of artificial intelligence, picture processing and recognition, global weather models, etc. Futhermore, in this work is presented Adams's model of parallel processing on which a data-flow computer architecture is based. The conception of data-flow computer systems makes possible the effective use of inherent parallelism, and together with all the advantages offered by LSI technology it will be possible to achieve performance above 1000 MFLOPS.

1. UVOD

Sve šira primjena računarskih sistema u ovom desetljeću na slijedećim područjima:

- područje vrlo složenih obrada (npr. globalni vremenski modeli, problemi s područja atomske fizike, kriptografije, složeni hidrodinamički problemi),
- problemi s područja koja sadrže visok stupanj inherentnog paralelizma u podacima ili u postupcima obrade (ili u jednom i drugom). Na primjer, područje umjetne inteligencije, obrada i raspoznavanje slika, obrada radarskih signala i sl.,
- područje visoko pouzdanih sistema za upravljanje i nadgledavanje npr., vojni sistemi protivprojektilne zaštite , uvjetuje [1]:
- visoku performansu takvih računarskih sistema, naravno, uz prihvatljivu cijenu (npr. performansa iznad 1000 MFLOPS Million Floating-Point Operations Per Second),
- iskorištenje svih prednosti koje nude tehnologije visokog (LSI) i vrlo visokog (VLSI) stupnja integracije - upotrebu velikog broja, ali samo nekoliko različitih tipova građevnih blokova, od kojih svaki ima vrlo visok omjer logika/broj izvoda (logic-to-pin ratio),
- korištenje inherentnog paralelizma u cilju brže obrade,
- postojanje takve programske opreme koja će omogućiti efikasno korištenje računarskih resursa i iskorištenje svih potencijalnih mogućnosti takvih velikih računarskih sistema.

Ispunjenjem gornjih uvjeta postići će se visok kvalitet odgovora računarskog sistema koji je proporcionalan količini izvršenih izračuna [2].

Za ilustraciju zahtijevanog kvaliteta odgovora, ocijenimo performansu računarskog sistema za obradu slike u realnom vremenu. Uzmimo primjer jedne tipične obrade: Neka je slika rezolucije 512×512 slikovnih elemenata (pixel), a vrijednost slikovnog elementa neka je kvantizirana u 256 razina (8 bita $\times 3$ za boju). U ovom kontekstu pod obradom slike u realnom vremenu podrazumijevamo obradu u vremenu manjem od vremena potrebnog za obnavljanje slike na TV zaslonu (data acqusition time of TV frame) i ono iznosi $4 \cdot 10^{-2}$ s.

Tipična obrada slike, npr., filtriranje uz pomoć maske dimenzija 3×3 slikovnih elemenata ili ortogonalna transformacija slike, zahtijeva od deset do stotinu instrukcija po slikovnom elementu.

Potrebna performansa računarskog sistema (izražena u MIPS - Million Instruction Per Second) je:

$$BSE \times BISE \times BSL \times 10^{-6},$$

gdje je:

BSE – broj slikovnih elemenata,

BISE – broj instrukcija za slikovni element,

BSL – broj slika u sekundi.

Za naš slučaj zahtjevana performansa iznosi:

$$(512 \times 512) \times (10 \div 100) \times (1/(4 \times 10^{-2})) \times 10^{-6},$$

odnosno 65,636 ÷ 656,36 MIPS-a.

Tako visoka performansa za velike računarske sisteme na osnovi von Neumannovog modela (i njegovih modifikacija) predstavlja nedostižan san.

Na primjer, veliki računarski sistem kao što je IBM-ov 3081 iz tzv. novog vala H ima performansu 10.4 MIPS-a, Amdahlov računarski sistem 5860 ima 13.9 MIPS-a, a Hitachijev AS/9000 DPC ima 15.9 MIPS-a [3].

U ovom radu opisane su dvije arhitekture računarskih sistema: konvencionalna arhitektura računara s upravljačkim tokom (control-flow computer) ili von Neumannova arhitektura računara i arhitektura računara upravljanog tokom podataka (data-flow computer).

Računarski sistemi na bazi von Neumannovog modela (i njegovih modifikacija) zbog svojih karakteristika ne mogu ispuniti navedene uvjete (visoka performansa, iskorištenje inherentnog paralelizma i sl.).

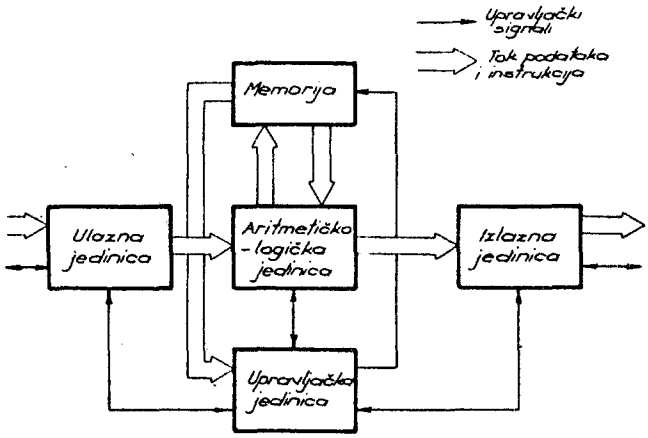
Računarski sistemi upravljani tokom podataka svojom koncepcijom omogućuju efikasno korištenje inherentnog paralelizma prisutnog u algoritmu, a uz korištenje svih prednosti koje donosi LSI i VLSI tehnologija omogućit će i dostizanje zahtijevanih performansi (>1000 MFLOPS). U prvom dijelu članka, osim opisa arhitekture von Neumannovog računara, dana je i kritika takve arhitekture. U drugom dijelu članka prikazan je model paralelne obrade i arhitektura računara upravljanog tokom podataka.

2. RAČUNAR S UPRAVLJAČKIM TOKOM (CONTROL--FLOW COMPUTER) I KRITIKA VON NEUMANNOVE ARHITEKTURE

Konvencionalna arhitektura računara je ona s upravljačkim tokom ili von Neumannova arhitektura. U jednom od najznačajnijih radova na području arhitekture računara (autori Burks, Goldstine i von Neumann [4]) koji je nastao nekih petnaestak godina prije pojave samog pojma arhitektura računara* [5], detaljno je razrađena koncepcija računara sa pohranjivanjem programa (stored-program computer). Taj rad je imao jak utjecaj ne samo na prvu generaciju računara, već i na sve naredne.

Von Neumannov model (SISD-Single Instruction Stream-Single Data Stream kategorije arhitekture / prema Flynnu [6]/) s izvjesnim modifikacijama, suvereno vlada već u četiri generacije računara. Opišimo, ukratko, osnovne značajke te arhitekture:

- von Neumannov model računara čine četiri funkcionalne komponente (aritmetičko-logička jedinica, upravljačka jedinica, U/I jedinica, memorija (sl. 1).



Sl. 1. Von Neumannov model računara

- primjenjuje koncepciju pohranjivanja instrukcija i podataka u istoj memoriji,
- memorija je jednodimenzionalna (pojavljuje se kao vektor sastavljen od riječi) sa sekvencijalnim načinom adresiranja (koncept linearnog adresiranja - memorijske lokacije imaju adrese 0000, 0001, 0002, ...),
- nema eksplicitne razlike izmedu instrukcija i podataka. Instrukcije se mogu smatrati kao podaci i biti modificirane (npr. adresni dijelovi instrukcijske riječi),
- značenje pojedinih podataka nije sadržano u samim podacima, na primjer, ni u čemu se ne razlikuje (eksplicitno) skup bitova koji predstavlja broj sa pomičnim zarezom od skupa bitova koji predstavljaju niz znakova (character string),
- posljednje i najvažnije, izvođenje programa je sekvencijalno, odnosno to je računar sa upravljačkim

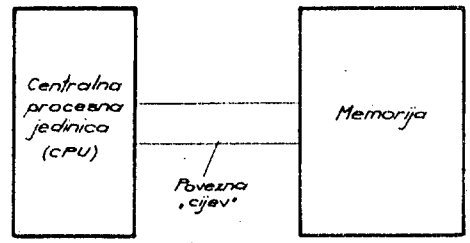
* Izraz „arhitektura računara" uveli su IBM-ovi konstruktori za opisivanje računarskih sistema serije IBM 360.

tokom (control-flow computer). U von Neumannovom modelu računara program se izvodi točno prema uredenju (nizu) instrukcija u programu. Dakle, tok izračuna je odreden slijedom instrukcija koje se pojavljuju u programu - što znači da je izračun odreden upravljačkim tokom u programu. Taj upravljački tok je sekvencijalni: instrukcije adresirane programskim brojilom slijedno se pribavljaju iz memorije i izvode. Sadržaj programskog brojila se u fazi PRIBAVI (Fetch) inkrementira i pokazuje na slijedeću instrukciju.

Za više od tri decenije, von Neumannov model računara doživljavao je promjene i poboljšanja koje su se odrazile na performansu računara, ali su još uvijek ostala poboljšanja u implementaciji dotične arhitekture, a ne poboljšanja u arhitekturi [7]:

- uvodenje indeksnih registara kao dodatnih adresnih registara (računar Datatron, tvrtka Electro Data Corp., 1953. godine),
- mikroprogramirana upravljačka jedinica (M.V. Wilkes, 1951.; računar IBM 360, tvrtka IBM, 1964. godine),
- primjena registara opće namjene (general-purpose register), taj koncept uveden je po prvi put u računaru Pegasus, tvrtke Ferranti L.t.d, 1956. godine,
- uvodenje priručne (cache) memorije (računar IBM 360 model 85, tvrtka IBM, 1968. godine),
- virtualna memorija - koncept koji omogućuje da se memorija sastavljena iz tri hijerarhijska nivoa pojavljuje u odnosu na program kao jedinstvena velika memorija čija je brzina skoro jednaka brzini memorije u prvom hijerarhijskom nivou (računar ATLAS, razvijen na Manchester University, 1959. godine),
- primjena prekidnog sistema (interrupt system) (Univac 1103, tvrtke Univac, / ERA, 1954. godine),
- aritmetika s brojevima sa pomičnim zarezom (IBM NORC, IBM 704, tvrtka IBM, 1954. godine),
- koncept asinhronog U/I, primjena funkcionalno nezavisnih jedinica - U/I kanala (LARC, tvrtka Remington Rand, 1956. godine),
- primjena protočne (pipeline) organizacije aritmetičko-logičke jedinice i više istovrsnih aritmetičkih sklopova (CDC 6600, tvrtka Control Data Corporation, 1964. godine).

Von Neumannov rad, nastao daleke 1946. godine, mnogi autori nazivaju genijalnim i najznačajnijim [5], [8] (u radu su prisutna razmatranja prekrivanja U/I aktivnosti sa aktivnosti procesora, hijerarhija memorijskog sistema i sl.). U novije vrijeme von Neumannov model doživljava i oštre kritike [7], [9] koje ne umanjuju značaj rada, već su više samo posljedica promjena koje su nastale u posljednjih trideset i pet godina.



Sl. 2. Pojednostavnjeni model von Neumannovog računara po J. Backusu

Model von Neumannovog računara, J. Backus predstavlja u pojednostavnjenom obliku (sl. 2) kao centralnu procesnu jedinicu (CPU) koja je pomoću prijenosne povezne cijevi (connecting tube) spojena sa memorijom [9]. Zadatak programa je da ulazne podatke transformira u izlazne, odnosno da mijenja sadržaj memorije u kojoj su pohranjeni ulazni podaci. To se u von Neumannovom modelu izvodi stalnim slanjem riječi naprijed-natrag riječ po riječ kroz povezanu cijev. Veliki dio prometa riječi u cijevi nisu korisni podaci već instrukcije i adrese. Očito da u takvom modelu veza izmedu memorija i centralne procesne jedinice predstavlja kritičnu komponentu. Tu poveznu cijev J. Backus naziva "von Neumannovo usko grlo". J. Backus tvrdi da sigurno mora postojati "... manje primitivan način ..." stvaranja velikih promjena u memoriji od onog sa slanjem ogromnog broja riječi naprijed-natrag kroz cijev. Ne samo to, koncept von Neumanna on naziva "intelektualnim uskim grlom" jer primorava na način razmišljanja: jedna riječ u vremenu (word-at-a-time thinking), umjesto da upućuje na rješavanje većih cjelina, odnosno paralelno rješavanje problema.

Iako je arhitektura von Neumannovog računara snažno utjecala i na programske jezike tvoreći razred von Neumannovih jezika (npr. Fortran, Algol 68 i dr.), izmedu koncepcije arhitekture takvog računara i koncepcije viših programskih jezika postoji veliki tzv. semantički jaz (semantic gap [7]), koji se ogleda u neprilagodenosti arhitekture strukturi viših programskih jezika (tipovi podataka, obrada nizova, blok strukture, procedure), što ima za posljedicu nepouzdanost programske opreme (software unreliability), složenost prevodilaca i postojanje zahtjeva za prekomjernom veličinom memorije. Taj semantički jaz je sve veći što su viši programski jezici funkcionalnije orijentirani (npr. APL,

FFP - Formal Functional Programming System [9]).

Pokušaji da se ostvare uvjeti (postavljeni u uvodu) primjenom konvencionalne arhitekture računara (računari sa upravljačkim tokom ili von Neumannovi računari) su bezuspješni iz slijedećih razloga:

- sekvencijalna priroda konvencionalnog računarskog sistema onemogućuje iskorištenje inherentnog paralelizma prisutnog u problemu,
- metode poboljšanja performanse na sve četiri razine (sklopovi i uređaji, algoritmi u funkcionalnim jedinicama, organizacija sistema, sistemska programska oprema [10]) su u velikoj mjeri iscrpljene, štoviše, takva arhitektura se u izvjesnoj mjeri udaljuje od mogućnosti iskorištenja prednosti koje nudi noviji razvoj LSI i VLSI tehnologije [1],
- prisustvo semantičkog jaza [7].

Konvencionalna von Neumannova arhitektura ne zadovoljava, kakva arhitektura računarskog sistema treba biti?

Odgovor na ovo pitanje traži se u paralelnim računarskim sistemima: multiprocesori [10], matrični procesori (array processor) [11], asocijativni procesori [12], sistolična polja [13]. Kao jedna od alternativa nudi se arhitektura koja predstavlja korjenitu promjenu u odnosu na von Neumannovu arhitekturu - računari upravljani tokom podataka (data-flow computer).

3. MODEL PARALELNE OBRADE

Modeli paralelne obrade u kojima se usmjereni grafovi upotrebljavaju za prikaz izračuna predstavljaju osnovu na kojoj se temelji koncepcija arhitekture računara upravljanog tokom podataka (radovi R.M. Karpa i R.E. Millera; J.E. Rodrigueza [1], D.A. Adamsa [14]).

D.A. Adams predlaže model M koji se sastoji iz primitivnih čvorova (primitive nodes), proceduralnih čvorova i usmjerenih grana kojima je dodijeljena funkcija privremenog pohranjivanja podataka (FIFO* struktura memorije), koji čine usmjereni graf. Primitivni čvorovi p_1, p_2, ... p_n predstavljaju izračunske korake i definirani su na slijedeći način:

Svaki primitivni čvor p_i, $1 \leq i \leq n$ ima uređen skup od ℓ ulaza podataka: d_{i1}, d_{i2}, ..., $d_{i\ell}$, i skup od m izlaza podataka: $d_{i\ell+1}$, $d_{i\ell+2}$, d_m.

Svaki čvor ima jednoznačnu funkciju

$$f_i : (V_{i1}, \dots, V_{i\ell}) \rightarrow (v_{i\ell+1}, \dots, v_{im}),$$

gdje je V_{ij} ulazni podatak - element iz skupa podataka ili φ-nul element, a v_{ij} izlazni podatak iz čvora. Primitivni čvor predstavlja gradevni blok iz kojih se sastavlja program. Funkcije f koje izvodi primitivni čvor mogu u zavisnosti od klase izračuna, biti u opsegu od logičkih operacija pa sve do složenih operacija (na primjer, množenje matrica ili množenje matrice s vektorom).

Svakom ulazu čvora p_i, $k \leq i \leq n$ može se dodijeliti binarna varijabla koja se naziva ulazni status S_{ij}, $1 \leq j \leq \ell$, koji poprima vrijednost L ili U. Čvor p_i ima tada definiranu jednoznačnu funkciju:

$$g_i : (S_{i1}, \dots, S_{i\ell}, V_{i1}, V_{i2}, \dots, V_{i\ell}) \rightarrow (S'_{i1}, \dots, S'_{i\ell}),$$

gdje u slučaju $S_{ij} = L$ V_{ij} poprima vrijednost φ-nul elementa. Nije dozvoljeno da sve vrijednosti V_{ij}, $j = 1,2, \dots \ell$ istovremeno poprime vrijednost φ.

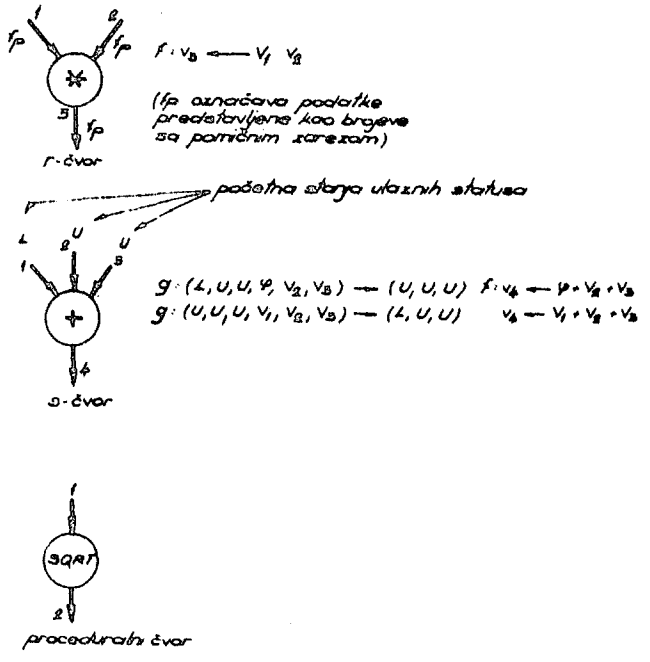
Čvorovi koji su definirani samo sa funkcijom f nazivaju se r-čvorovi, dok se oni definirani sa f i g nazivaju s-čvorovi.

Ulazni status postavljen za s-čvor može biti ili L (locked - zaključan) ili U (unlocked - otključan).

Vrijednost na zaključnom ulazu ostaje i ne koristi se u izračunu sve dok taj ulaz ne postane otključan. Funkcija g u s-čvoru, pomoću L i U ulaznih statusa, izabire podatke iz medusobno nezavisnih izvora.

Proceduralni čvor je čvor čija operacija može biti opisana drugim grafom.

Slika 3 prikazuje pojedine vrste čvorova u modelu M.



Sl. 3. Tipovi čvorova u modelu M

* FIFO - First-In-First-Out

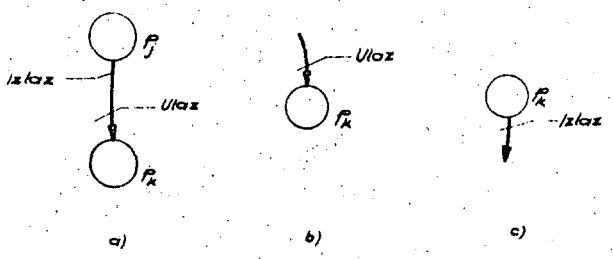
Usmjerene grane povezuju čvorove i služe za prijenos i privremeno pohranjivanje podataka određenog tipa. Podaci tvore FIFO strukturu (rep čekanja), a njihova struktura odnosno tip podataka definiran je gramatikom (Definicija 1 [14]).

Graf procedura (ili samo graf) G sastoji se od skupa čvorova $\eta = \{p_1, p_2, \dots, p_n\}$ i skupa usmjerenih grana $\epsilon = \{e_1, e_2, \dots, e_m\}$ pri čemu svaka usmjerena grana $e_1 \in \epsilon$ mora zadovoljavati jedan od slijedeća tri uvjeta (sl. 4):

1) mora biti usmjerena iz izlaza čvora $N_j \in \eta$ u ulaz čvora $N_k \in \eta$,

2) usmjerena samo prema ulazu čvora $N_k \in \eta$ s tim da se uređen skup usmjerenih grana zove ulazni skup od G i različit je od praznog skupa,

3) usmjerena samo iz izlaza čvora $N_j \in \eta$. Uređen skup usmjerenih grana naziva se izlazni skup G i različit je od praznog skupa.



Sl. 4. Uvjeti za grane u graf proceduri

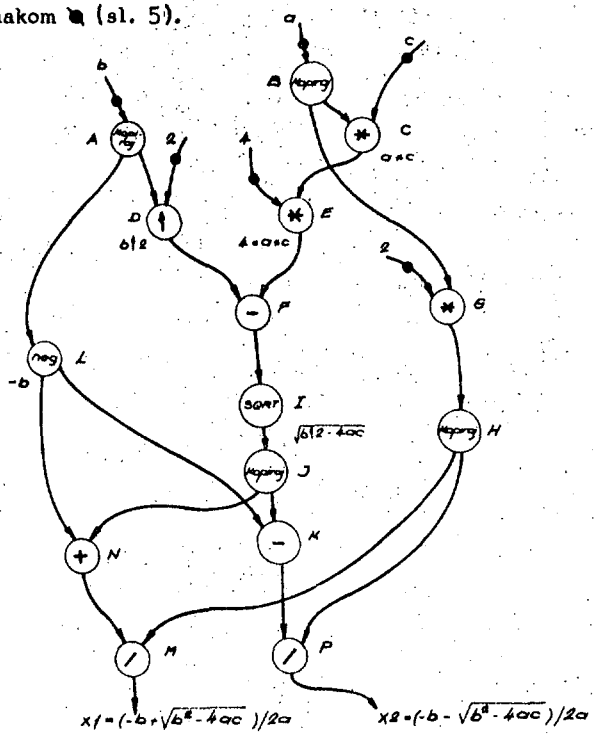
Graf program $\mathcal{P}$ definiran je kao uređen par (G, X), gdje je G graf procedura, a X skup svih graf procedura. Uvjeti koje graf program mora zadovoljavati dani su u [14].

Osnovna značajka modela M je da bilo koji izračunski korak (aktiviranje čvora p_i) može započeti u onom trenutku kada usmjerene grane koje ulaze u taj čvor sadrže podatke koji su potrebni za izračun. Broj izračunskih koraka koji mogu biti izvršeni u danom trenutku dinamički je određen samo raspoloživim podacima. Kažemo da je uvjetovan tokom podataka (data-flow).

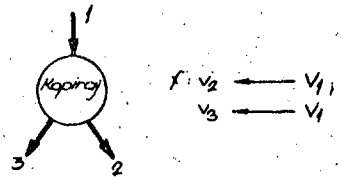
Model M ima jedno važno svojstvo: programi izraženi pomoću graf programa su deterministički, odnosno rezultati konačnog izračuna ne zavise od broja upotrebljenih procesora, njihove relativne brzine ili od izabranog slijeda aktiviranja pojedinih čvorova (naravno, onih koji imaju raspoložive sve potrebne podatke).

Slika 5 prikazuje graf program (data-flow program) za izračun korijena kvadratne jednadžbe.

r-čvor sa funkcijom kopiraj opisan je na sl. 6. Početni uvjeti su na usmjerenim granama označeni sa znakom ● (sl. 5).



Sl. 5. Data-flow program za izračun korijena kvadratne jednadžbe



Sl. 6. r-čvor sa funkcijom KOPIRAJ

Razmotrimo izvođenje programa sa slike 5.

S obzirom na osnovnu značajku modela M, u prvom trenutku raspoloživi su podaci za čvorove A i B, izračunski koraci (funkcija KOPIRAJ) se u njima izvode istovremeno.

Nakon toga, postupak izračuna može biti prikazan slijedećom sekvencom (u istom redu označeni čvorovi izvode istovremeno specificirane operacije):

D - C - G - L

E - H

F

I

J

N - K

M - P.

Iz primjera programa vidimo da priroda problema uvjetuje raspoloživost operanada, i ukoliko je problem takav da ima visok stupanj paralelizma prisutan u podacima i operacijama, tada je omogućeno istovremeno izvođenje velikog broja operacija. Odnosno, pri takvom

izvođenju izračunskih koraka sačuvan je inherentni paralelizam prisutan u problemu.

4. RAČUNARI UPRAVLJANI TOKOM PODATAKA (DATA-FLOW COMPUTER)

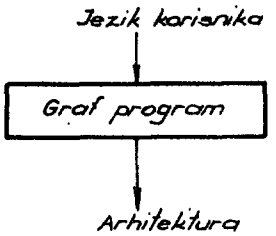
Računari upravljani tokom podataka predstavljaju korjenitu promjenu u odnosu na von Neumannovu arhitekturu. Osnovna razlika leži u tome na koji se način odlučuje o pribavljanju slijedeće instrukcije za izvršenje. U konvencionalnom von Neumannovom računaru (računaru sa upravljačkim tokom) izvođenje programa je određeno prema uređenom slijedu instrukcija u programu (control-flow).

Kod računara upravljanih tokom podataka pribavlja se ona instrukcija (ili instrukcije) koja u danom trenutku ima sve potrebne podatke raspoložive (u skladu s modelom M). Dakle, izvođenje programa je upravljano sa raspoloživosti podataka. Takav računar ne treba konvencionalnu upravljačku jedinicu, ne treba programsko brojilo, a ni upravljački tok ugrađen u program u obliku slijeda instrukcija. S obzirom na gornja svojstva takav računar omogućava iskorištenje prirodnog i inherentnog paralelizma koji je prisutan u algoritmima.

Računar upravljan tokom podataka ima slijedeće četiri bitne karakteristike [15]:

- operacija se izvodi u trenutku raspoloživosti svih potrebnih ulaznih podataka (activation by availability),
- nakon što je operacija izvršena, ulazni podatak koji je sudjelovao u toj operaciji nije više raspoloživ kao ulazni podatak za neku drugu operaciju (FIFO-struktura podataka u modelu M),
- ne primjenjuje se koncept pohranjivanja podataka poput onog u konvencionalnom računaru, (npr. oznaka i pohranjivanje varijabli),
- nema ograničenja u pogledu upravljanja slijedom izvođenja; izuzev onih koja su dana tokom podataka (nema elemenata upravljanja tokom instrukcija).

Pored mogućnosti iskorištenja inherentnog paralelizma prisutnog u algoritmu, računar upravljan tokom podataka ima mogućnost dinamičkog prilagođavanja svoje strukture strukturi algoritma. U stvari, takav računar se može smatrati kao oblik arhitekture zasnovan na jeziku (language - based architecture) [1] u kojem je graf program taj osnovni jezik, odnosno graf program (data-flow program graph) služi kao međustupanj između višeg programskog jezika korisnika i arhitekture računara (sl. 7). Viši programski jezik (textual data-flow programming language) ima neka svojstva konvencionalnih von Neumannovih jezika i svojstava funkcionalnih jezika. Na primjer, program u višem jeziku (sličnom PASCAL-u) za izračun korijena kvadratne jednadžbe ima oblik [15]:



Sl. 7. Graf program kao međustupanj između jezika i arhitekture [1]

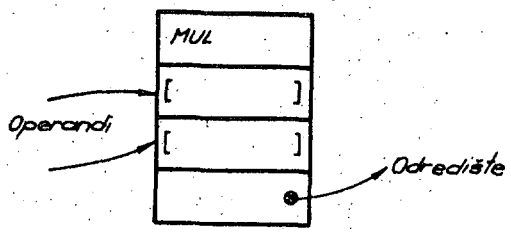
```
FUNCTION quad-roots INPUT (a,b,c: REAL)
                    OUTPUT (x1, x2: REAL)
  VAR temp: REAL
  BEGIN
    temp : = sqrt (b**2-4*a * c)
      x1 : = (-b + temp)/(2*a)
      x2 : = (-b - temp)/(2*a)

  END
VAR i1, i2, i3 : REAL
    Γ1, Γ2, : REAL
BEGIN
    .
    .
    .
i1 : = ... ; i2 : = ... ; i3 : = ... ;
          .
          .
          .
(Γ1, Γ2) : = quad-roots (i1, i2, i3);
          .
          .

END
```

Prevodioci za više programske jezike za računare upravljane tokom podataka, na osnovi zavisnosti podataka i operacija, generiraju graf programe. Primjeri viših programskih jezika za taj tip računara su ID jezik razvijen na University of California i VAL nastao na MIT-u [16].

Od arhitekture računara upravljanog tokom podataka traži se da omogući efikasnu implementaciju tako dobivenih graf programa. Očito da dinamičko prilagođavanje strukture strukturi algoritma postavlja velike zahtjeve, koji su i inače prisutni u visoko paralelnim računskim strukturama, u pogledu prospojne mreže

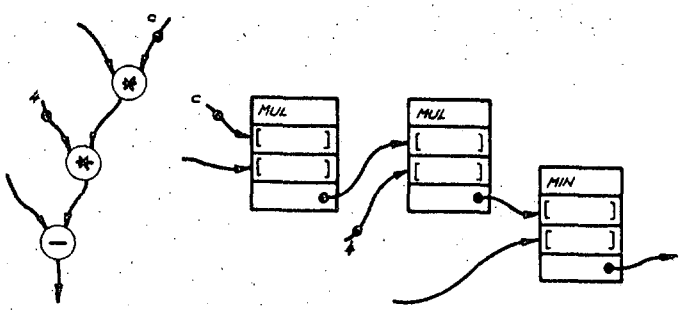
(communication network) koja usmjerava rezultate od/ /prema procesnim elementima i instrukcijskim riječima. Naime, instrukcijska riječ tzv. kalup aktivnosti (activity template) ima slijedeću strukturu:

- polje operacijskog koda koji specificira operaciju koja će biti izvršena,
- dva ili više prijemna polja operanada, koja čekaju vrijednosti operanada,
- jedno ili više polja koje određuju odredište rezultata operacija izvedenih nad operandima.



Sl. 8. Struktura kalupa aktivnosti activity template

Slika 8 prikazuje strukturu kalupa aktivnosti (activity template). Slika 9 prikazuje detalj graf programa za izračun korijena kvadratne jednadžbe prikazanog pomoću kalupa aktivnosti.



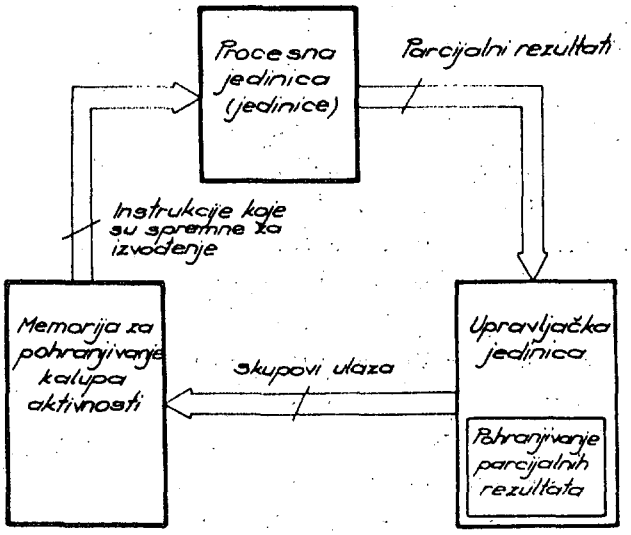
Sl. 9. Struktura kalupa aktivnosti za detalj graf programa

Instrukcijska riječ, odnosno kalup aktivnosti može biti pribavljen za izvršenje u onom trenutku kada je kalup popunjen potrebnim operandima.

Razmotrimo sada mehanizam djelovanja računara upravljanog tokom podataka. Strukturu takvog računara prikazuje slika 10.

Upravljačka jedinica ima zadatak da na temelju parcijalnih rezultata i polja koji određuje odredište rezultata (paketi rezultata) tvori skupove parcijalnih rezultata.

Polje koje određuje odredište rezultata sadrži i informaciju o broju potrebnih operanda za kalup aktivnosti. Na osnovi te informacije upravljačka jedinica određuje kada je skup operanada (ulaza) kompletiran. Tako kompletiran skup operanada prosljeđuje se memo-



Sl. 10. Struktura računara upravljanog tokom podataka

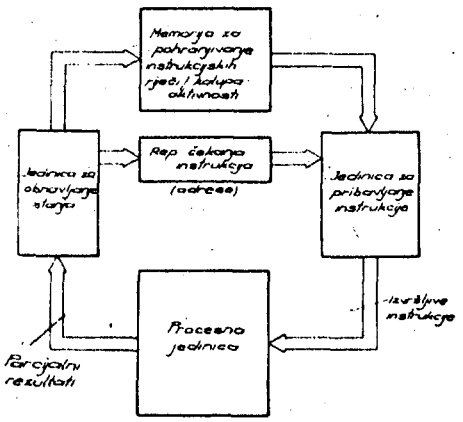
riji za pohranjivanje kalupa aktivnosti. Upotpunjeni kalupi aktivnosti (nazivaju se operacijski paketi (sl. 11)) skupom operanada šalju se prema procesnoj jedinici koja izvodi operacije specificirane poljem operacijskog koda i generira pakete rezultata (result packet, sl. 11) za svako odredišno polje.

operacijski paket: <oper. kôd, operandi, odredišta>

paket rezultata: < vrijednost, odredište >

Sl. 11. Struktura operacijskog paketa i paket rezultata

Memorija za pohranjivanje kalupa aktivnosti (prema graf programu) realizirana je pomoću memorije s izravnim pristupom. Upravljačka jedinica pored upravljačke logike ima asocijativnu memoriju za pohranjivanje parcijalnih rezultata. Procesna jedinica se, obično, sastoji iz većeg broja identičnih procesora razine složenosti mikroprocesora [17].



Sl. 12. Struktura računara upravljanog tokom podataka razvijenog na MIT-u

Slika 12 prikazuje strukturu računara upravljanog tokom podataka koji je razvijen na MIT-u [1]. Graf program koji je opisan skupom kalupa aktivnosti pohranjen je u memoriji. Svaki kalup aktivnosti ima adresu koja ulazi u rep čekanja instrukcija u trenutku kada je instrukcija spremna za izvršenje. Jedinica za pribavljanje instrukcija uzima iz repa čekanja adresu spremne instrukcije i čita kalup aktivnosti iz memorije, tvori operacijski paket i prosljeđuje ga procesnoj jedinici. Procesna jedinica izvodi operaciju određenu s operacijskim kodom i pakete rezultata šalje u jedinicu za obnavljanje stanja (update). Ona proslijeđuje parcijalne rezultate prema kalupima aktivnosti (oni su određeni odredišnom adresom) i ispituje da li su svi zahtijevani operandi za pojedine kalupe aktivnosti prispjeli. Ako jesu, sklop za obnavljanje stanja prenosi adresu takvog kalupa u rep čekanja instrukcija. Očito da broj adresa koje čekaju u repu čekanja instrukcija označava i stupanj paralelnosti primjenjenog algoritma.

Primjenom većeg broja procesora (npr. polja procesora) u procesnoj jedinici koji su preko prospojne mreže (communication network) spojeni sa jedinicom za pribavljanje instrukcija postiže se efikasno iskorištenje paralelnosti algoritma. U ovom slučaju paralelno izvođenje je uslovljeno veličinom i složenošću prospojne mreže.

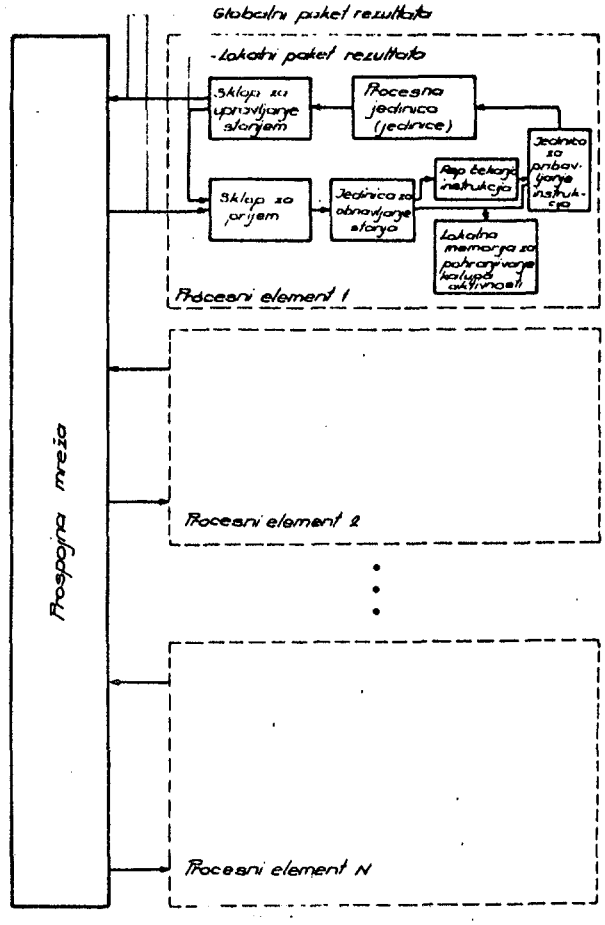
Multiprocesor upravljan tokom podataka [1], [18] pruža još veće mogućnosti u iskorištenju inherentnog paralelizma. U takvom računaru graf program je razdijeljen na dijelove koji su distribuirani pojedinim procesnim elementima. Procesni elementi imaju svoje lokalne memorije za pohranjivanje kalupa aktivnosti. U tom slučaju razlikuju se dva tipa paketa rezultata:

- lokalni, koji ostaju unutar procesnog elementa,
- globalni; paket rezultata koji u odredišnom polju nema adresu lokalne memorije vlastitog procesnog elementa već mora, preko prospojne mreže, biti usmjeren u neki drugi procesni element.

Slika 13 prikazuje strukturu multiprocesora upravljanog tokom podataka [1].

Opis multiprocesora upravljanog tokom podataka dan je u lit. [18], a u radu [19] razmatra se primjena funkcionalnih memorija [20] u jednom takvom sistemu.

Analize performansi računara upravljanog tokom podataka su pokazale da je postignuto povećanje brzine 40 : 1 u odnosu na konvencionalne visoko performansne računare (high-speed computer) u rješavanju problema metodom mreža i izračuna FFT (Fast Fourier Transform) [15].



Sl. 13. Struktura multiprocesora upravljanog tokom podataka

Računar upravljan tokom podataka specijalno namijenjen za obradu GCM (General Circulation Model) modela za prognozu vremena ima performansu veću za dva reda veličine u odnosu na IBM 360/95 [21].

5. ZAKLJUČAK

Primjena računarskih sistema na područjima meteorologije, kriptografije, umjetne inteligencije, obrade slika, obrade radarskih signala i sl. uvjetuje njihovu visoku performansu. Uprkos posezima na sve četiri razine (sklopovi, algoritmi, organizacija, programska oprema) u cilju poboljšanja performansi, računarski sistemi konvencionalne arhitekture ne mogu postići zahtijevane performanse. Jedan od glavnih ograničavajućih faktora je njihova serijska priroda izvođenja izračuna.

Razvoj tehnologije LSI i VLSI omogućio je korjenite zaokrete u odnosu na von Neumannovu arhitekturu. Jedan takav zaokret predstavlja arhitektura računara upravljanog tokom podataka. Ona omogućuje efikasno iskorištenje inherentnog paralelizma prisutnog u problemu, odnosno algoritmu, omogućuje korištenje VLSI

komponenti i razvoj programske opreme koja će omogućiti efikasno korištenje računarskih resursa. Rezultati koji potvrđuju gornje pretpostavke predstavljaju plodove istraživanja na području graf-programa, viših programskih jezika za opisivanje graf-programa i arhitekture računara upravljanog tokom podataka. Ta istraživanja se odvijaju na MIT-u (J.B. Dennis), University of California (K.P. Gostelow, Arviud), University of Utah (A.L. Devis), University of Manchester (J.R. Gurd) i u CERT-DERI u Toulouse (J.C. Syre) u Francuskoj.

LITERATURA

[1] J.B. Dennis, Data Flow Supercomputers, Computer, Vol. 13. No. 11., novembar 1980, str. 48-56.

[2] L.S. Haynes, Highly Parallel Computing, Computer, Vol. 15, No. 1., januar 1982, str. 7-8.

[3] D.F. Farmer, IBM-Compatible Giants Datamation, Vol. 27, No. 13, str. 92-104.

[4] A.W. Burks, H.H. Goldstine, J. Neumann, Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, u C.G. Bell, A. Newell, Computer Structures: Readings and Examples, McGraw-Hill, New York 1971.

[5] G.J. Lipovski, K.L. Doty, Developments and Directions in Computer Architecture, Computer, Vol. 11, No. 8, august 1978, str. 54-57.

[6] M.J. Flynn, Some Computer Organizations and Their Effectivenes, IEEE Trans. on Comput., Vol C-21, No. 9, septembar 1972, str. 948-960.

[7] G.J. Myers, Advances in Computer Architecture, J. Wiley, New York, 1978.

[8] E.E. Swartzlander editor , Computer Design Development, Hayden Book Co., New Jersey, 1976. str. 219.

[9] J. Backus, Can Programming Be Liberated from Neumann Style and Its Algebra of Programs, Com. of the ACM, Vol. 21, No. 8, august 1978., str. 613-641.

[10] P.H. Enslow Jr., Multiprocessors and Other Parallel Systems, An Introduction and Overview, u Computer Architecture, Workshop of the Gesellechaft für Informatik, Erlangen, May 1975, Springer-Verlag, Berlin, 1976.

[11] K.J. Thurber, L.D. Wald, Associative and Parallel Processors, ACM Comp. Surveys, Vol. 7, No. 4, decembar 1975, str. 215-255.

[12] S.S. Yau, H.S. Fung, Associative Processor Architecture - A Survey, ACM Comp. Surveys, Vol. 9, No. 1, mart 1977, str. 3-27.

[13] H.T. Kung, Why Systolic Architectures?, Computer, Vol. 15, No. 1, januar 1982, str. 37-46.

[14] D.A. Adams, A Model for Parallel Computations, u Parallel Processor Systems, Technologies and Applications, ed. L.C. Hobbs et al., Spartan Books, New York, 1970.

[15] P.C. Treleaven, Exploiting Program Concurrency in Computing Systems, Computer, Vol. 12, No. 1., januar 1979, str. 42-50.

[16] L.S. Haynes, et al., A Survey of Highly Parallel Computing, Computer, Vol. 15, No. 1, januar 1982, str. 9-24.

[17] S. Ribarić, Arhitektura mikroprocesora, Tehnička knjiga, Zagreb 1982.

[18] J. Rumbaugh, A Data Flow Multiprocessor, IEEE Trans. on Comp., Vol. C-26, No. 2, februar 1977, str. 138-146.

[19] Jan Chudik, Architecture of the Data Flow Computer Based on Functional Memories, VI bosansko-hercegovački simpozij iz informatike, Jahorina 1982.

[20] P.L. Gardner, Functional Memory and Its Microprogramming Implications, IEEE Trans. on Comp., Vol. C-20, No. 7, juli 1971, str. 764-775.

[21] J.B. Dennis, K.K.S. Weng, Application of Data Flow Computation to the Weather Problem, u High Speed Computer and Algorithm Organization, ed. D.J. Kuck et al, Academic Press, Inc., New York, 1977.

ZNAČILNOSTI INTELOVIH 16–BITNIH MIKRORAČUNALNIKOV

DAVID ČUK,
EDO GERMADNIK

UDK: 681.3.06 INTEL

INSTITU „JOŽEF STEFAN", LJUBLJANA

V članku so opisane značilnosti Intel-ove družine 16-bitnih mikroračunalnikov 8086. Namen članka je seznaniti bralca z Intel-ovim principom gradnje 16-bitnega mikroračunalnika in glavnimi komponentami družine 8086. Kratko so opisani tudi operacijski sistemi, ki podpirajo družino 8086 v različnih področjih njihove uporabe. Navedeni so tudi rezultati primerjave družine 8086 z mikroračunalnikom MC68000 in miniračunalnikom PDP 11/44.

THE INTEL 16-BIT MICROCOMPUTER FAMILY CHARACTERISTICS. In this paper the Intel 16-bit microcomputer family 8086 is described. A purpose of this article is to introduce the reader to the Intel 16-bit microcomputer philosophy and the main 8086 family components. Available 8086 family operating systems for different types of applications are presented. Results of the benchmark test of the 8086 family with the microcomputer MC68000 and the minicomputer PDP 11/44 are also discussed.

0. UVOD

Po podatkih iz literature (npr. Electronics, maj 1982) so Intel-ovi mikroračunalniki daleč najbolj uporabljeni na svetu, saj presegajo tri četrtine vseh prodanih mikroračunalnikov.

Dejstvo je, da je Intel v Jugoslaviji, pa tudi v Sloveniji, zelo razširjen. Predvsem gre za uporabo 8-bitnih mikroračunalnikov, v posameznih institutih in delovnih organizacijah pa tudi že razvijajo 16-bitne mikroračunalnike, drugi pa se odločajo o nadaljni poti.

Namen tega članka je strnjen prikaz Intel-ovega načina gradnje 16-bitnih mikroračunalnikov in prikaz obstoječe sistemske programske opreme zanje.

V članku je predvsem prikazana zgradba družine 8086, katere glavne značilnosti so /1/:

- posamezne sistemske naloge so distribuirane med različne specializirane komponente,
- zmožnost multiprocesiranja je vgrajena že v sama vezja.

1. iAPX 86

iAPX 86 je sklop enega ali več vezij iz družine 8086, ki predstavljajo funkcionalno celoto.

Sestavni deli so naslednja vezja:

8086 - centralna procesna enota,

8087 - aritmetični koprocesor,

8089 - procesor vhodno/izhodnih enot,

80130 - procesor operacijskega sistema.

Imamo naslednje oznake:

iAPX 86/10 = 8086

iAPX 86/11 = 8086 + 8089

iAPX 86/20 = 8086 + 8087

iAPX 86/21 = 8086 + 8087 + 8089

iAPX 86/30 = 8086 + 80130

1.1. Multiprocesiranje

Multiprocesiranje ima, predvsem pri srednjih in večjih sistemih veliko prednosti pred pristopom z eno samo zmogljivo centralno procesno enoto:

- procesna opravila so porazdeljena med več specializiranih procesorjev, ki so optimizirani za izvajanje določenih tipov opravil,
- možnost paralelnega procesiranja,
- okvara enega dela sistema le delno vpliva na delovanje celotnega sistema,
- razdelitev celotne naloge na več enostavnih opravil, ki jih lahko razvijamo paralelno, skrajša čas razvoja in olajša kasnejše spremembe posameznih opravil.

Družina 8086 je grajena za multiprocesiranje, to pomeni, da ima že vgrajene elemente za medprocesorsko povezavo in koordinacijo.

Gradnja podpira dva tipa procesorjev:

- neodvisne procesorje;

- koprocesorje.

Neodvisni procesorji so tisti, ki izvajajo vsak svoje lastno zaporedje instrukcij (npr. 8086, 8089), medtem ko koprocesorji dobijo svoje instrukcije od drugega procesorja (npr. 8087).

Koprocesor paralelno razpoznava instrukcije drugega procesorja in če je določena instrukcija zanj, jo tudi izvrši. Koprocesor pravzaprav razširja nabor instrukcij določenega procesorja.

Družine 8086 rešuje problem medprocesorske koordinacije z:

- mehanizmom dodeljevanja vodila preko vgrajene logike ali zunanjega vezja in je nevidno za programsko opremo;

- vsak procesor vključuje signal LOCK in se ga lahko programsko aktivira. S tem preprečimo drugim procesorjem dostop do skupnih virov sistema (spomin, vhodno/izhodne enote). Vsak procesor vsebuje instrukcijo, ki testira in poveča vsebino pomnilniške besede pri blokiranem vodilu. Ta instrukcija omogoča izvedbo mehanizma semaforja za krmiljenje dosega večjega števila procesorjev do skupnih računalniških virov (več o Dijkstra-ovem semaforju npr. v literaturi /2/).

Splošna struktura vodil za medsebojno povezavo posameznih procesorjev družine 8086 je prikazana na sliki 1.

V sistemu imamo:

- pomnilniške module, skupni so za vse procesne module;

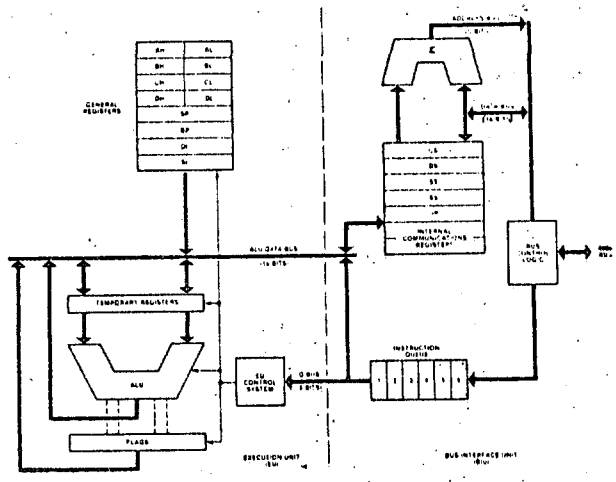
- vhodno/izhodne module, ti so prav tako skupni za vse procesne module;

- procesne module, ti vsebujejo enega ali več procesorjev povezanih preko lokalnega vodila, lahko pa vsebujejo tudi privaten pomnilnik in vhodno/izhodne enote, te niso dostopne ostalim procesnim modulom v sistemu.

Povezava med posameznimi moduli je preko sistemskega vodila. Intel je že za 8-bitno družino mikroračunalnikov razvil večprocesorsko vodilo MULTIBUS /4/, kasneje pa ga je razširil še za 16-bitne mikroračunalnike. Vodilo je v svetu zelo razširjeno in je postalo tudi IEEE standard.

1.2. Centralna procesna enota 8086

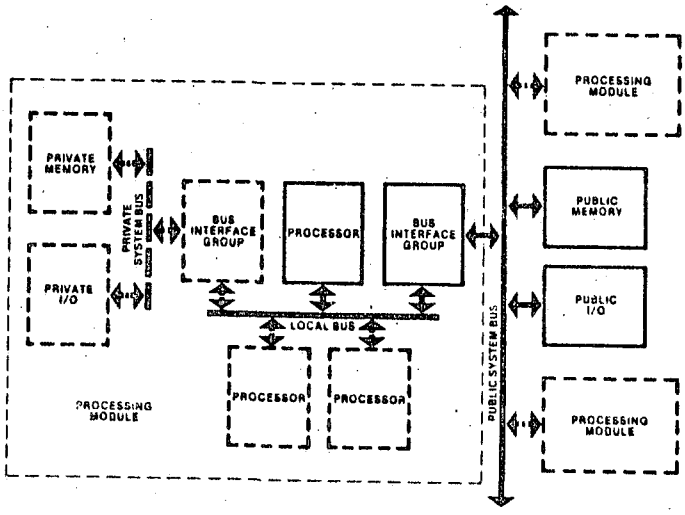
Osnovo družine 8086 predstavlja centralna procesna enota (CPU) 8086. Funkcionalno enaka je tudi CPU 8088, le da ima ta samo 8-bitno podatkovno vodilo za povezavo s pomnilnikom in vhodno/izhodnimi enotami. Shema CPU 8086 je na sliki 2.



Slika 2: Blok diagram CPU 8086

Detajlnejši opis CPU 8086 je npr. v literaturi /1/, tu pa bi navedli le nekatere najpomembnejše značilnosti:

- CPU 8086 lahko efektivno uporabimo v različno zahtevnih aplikacijah, kar ji omogočata dva različna načina delovanja. Tako vsebujejo pri minimalnem načinu delovanja nekateri kontakti procesorja signale, ki direktno krmilijo pomnilnik in



Slika 1: Struktura vodil družine 8086

vhodno/izhodne enote (s tem se zmanjša število vezij na minimalno), pri maksimalnem načinu pa vsebujejo ti kontakti signale za krmiljenje dodatnih vezij, ki omogočajo multiprocesiranje;

- performance 5MHz 8086 je približno 7 do 10 krat večja od perforance 2 MHz 8080A. Velika zmogljivost je dosežena z notranjo zgradbo s pomočjo razdelitve opravil med izvajalno enoto (EU) in enoto za povezavo na vodilo (BIU). Tako EU izvaja instrukcije, BIU pa skrbi za prenos instrukcij in podatkov iz pomnilnika in za zapis rezultatov. Izvajanje obeh enot se večji del vrši paralelno;

- CPU 8086 je grajena za delo s procesorjem vhodno izhodnih enot (IOP) 8089, in drugimi neodvisnimi procesorji in koprocesorji;

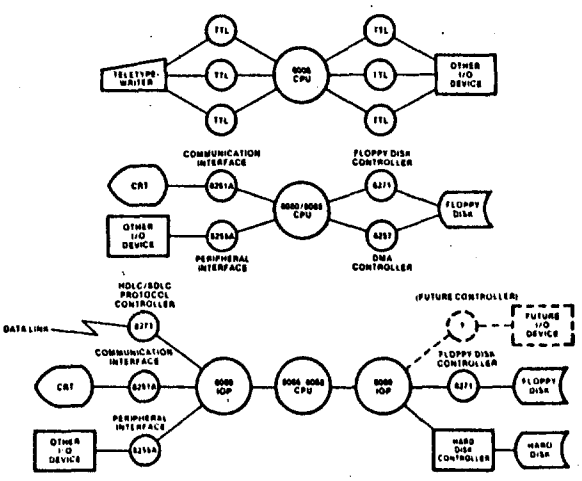
- CPU 8086 direktno podpira programiranje v višjenivojskih jezikih (npr. PL/M-86). Nabor instrukcij je simetričen, to omogoča direktne operacije nad operandi v spominu ali na skladu. Mehanizem naslavljanja direktno podpira polja, strukture in druge višjenivojske programske podatkovne strukture;

- CPU 8086 lahko naslavlja 1M zlogov pomnilnika, ki je razdeljen v logične segmente z do 64 K zlogov. CPU lahko istočasno dosega štiri takšne segmente (programski, podatkovni, skladovni in dodatni segment). Segmentacija pomnilnika, skupaj z instrukcijami za manipulacijo z nizi, omogoča enostavno izvedbo dinamičnega relociranja programov in podatkov.

1.3. Procesor vhodno/izhodnih enot 8089

Procesor vhodno/izhodnih enot 8089 (IOP) je neodvisen procesor družine 8086. Uporabljamo ga v sistemih, kjer so intenzivne vhodno/izhodne operacije. IOP 8089 je mikroprocesor z dvema DMA kanaloma, katerega nabor instrukcij je prilagojen vhodno/izhodnim operacijam.

IOP 8089 je izraz splošnega trenda odnosa med CPU in vhodno/izhodnimi enotami pri prvih treh generacijah mikroprocesorjev (glej sliko 3).



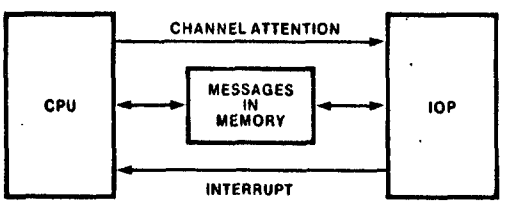
Slika 3: Odnos CPU - vhodno/izhodne enote

V prvi generaciji so bile povezave z vhodno/izhodnimi enotami narejene s TTL komponentami, prenosi so bili na nivoju bitov, obseg enot pa je bil majhen, in še te so bile počasne.

V drugi generaciji so bili razviti krmilniki enot na enem vezju, ki so razbremenjevali CPU najnižjega nivoja, tako da je CPU prenašala cele zloge. Razvoj DMA krmilnika je omogočil uporabo zelo hitrih enot, prenos pa se je izvršil na nivoju blokov podatkov. Krmilniki pa so še vedno zahtevali določeno akcijo CPU ob začetku in koncu prenosa, pa tudi med DMA ciklom je bil CPU zaustavljen.

IOP 8089 predstavlja tretjo generacijo procesiranja vhodno/izhodnih enot. Ta razbremenjuje CPU še zgoraj navedenih opravil. CPU sedaj samo pripravi v pomnilniku sporočilo z vpisom zahtevane vhodno/izhodne operacije (slika 4). IOP prebere sporočilo, izvrši vhodno/izhodno operacijo in obvesti CPU, da je operacija zaključena. Med izvrševanjem le te lahko CPU vrši druga opravila.

IOP 8089 lahko vrši istočasno povezavo z dvema perifernima enotama.



Slika 4: Povezava med CPU in IOP

1.4. Numerični koprocesor 8087

8087 je numerični komprocesor družine 8086 in izvaja operacije v standardnem IEEE formatu realnih števil (80-bitni). Koprocesor 8087 uporabljamo v numerično intenzivnih aplikacijah. Hitrost izvrševanja operacij je približno 100 krat večja, kot če iste vršimo s programom za CPU 8086. Tako je približni čas za izvršitev množenja dveh realnih števil 18 mikro sec., deljenja 39 mikro sec., kvadratnega korena pa 36 mikro sec.

1.5. Procesor operacijskega sistema 80130

Vemo, da predstavlja razvoj programske opreme poglavitnih del cene določene aplikacije. Zato se že dlje časa pojavlja misel o vgraditvi standardnih funkcij v integrirano vezje. Vezje 80130 je 16 K statične sistemske programske opreme (jedro operacijskega sistema za delo v realnem času iRMX 86) skupaj s krmilnikom prekinitev in tremi časovniki, ki podpirajo programsko opremo.

80130 rešuje 35 zahtevkov za delo v realnem času, ki obravnavajo:

- dejavnosti,
- komunikacijo in sinhronizacijo med dejavnostmi,
- prekinitve,
- razporejanje pomnilnika.

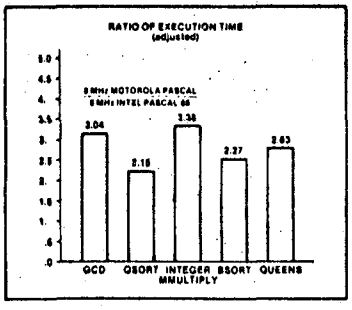
Vezje 80130 velikokrat zadošča za izvedbo enostavnejših aplikacij, za komplekšnejše pa jih lahko uporabnik nadgradi s svojim operacijskim sistemom ali pa uporabi Intel-ov operacijski sistemi iRMX 86.

1.6. Zmogljivost iAPX 86

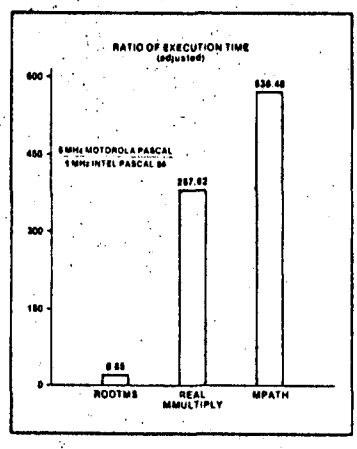
1.6.1. Primerjava z MC68000

Dejstvo je, da se danes piše večina programov v višjih programskih jezikih. Za 16-bitne mikroračunalnike velja ocena, da je 80 % aplikacij pisanih v visokih programskih jezikih. Iz tega razloga so v tej točki na kratko povzeti rezultati poročila /5/ za višji programski jezik Pascal-86 ver. 1.0, ki teče na Intel-ovem razvojenem sistemu Intellec (iAPX86/20 z 5 MHz uro) ter Pascalom firme Motorola (Motorola Pascal ver. 1.2), ki teče na razvojnem sistemu EXORMAC (MC68000 z 8 MHz uro).

V poročilu je primerjanih 44 programov glede na velikost generiranih strojnih kod. Od teh je primerjanih 8 programov še glede na hitrost izvajanja. Vsi programi imajo enako izvorno obliko za oba sistema. Izmerjeni rezultati so prikazani na slikah 5, 6 in 7.



Slika 6: Rezultat testa za nenumerične programe



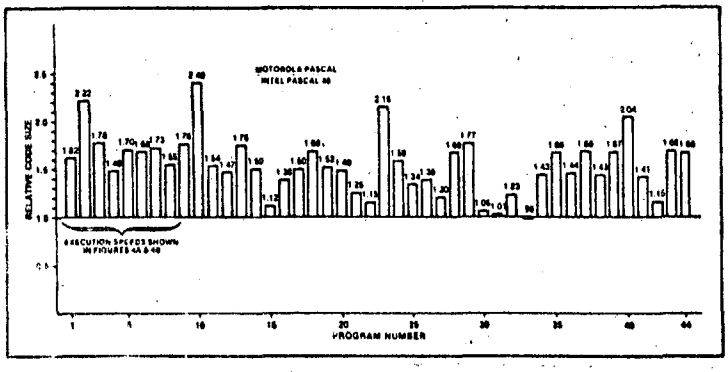
Slika 7: Rezultat testa za numerične programe

Obseg strojnih kod je za vse programe v mejah 0.9 - 2.5. Le pri enem programu je razmerje 0.9, kar pomeni da pri tem primeru generira Pascal Motorole manj kod kot Intel-ov Pascal-86. V vseh drugih primerih pa Motorola Pascal generira večje število strojnih kod (glej sliko 5).

Izvajalni časi instrukcij so korigirani na nič čakalnih stanj. (Intel-ov in Motorol-in razvojni sistem uporabljata čakalna stanja v strojnih ciklih procesorja), čas pa je korigiran za 8 MHz delovanje procesorja 8086. Izračunano razmerje je podano na slikah 6 in 7.

Program v Pascal-86 teče najmanj dvakrat hitreje kot Motorola Pascal. Pri numeričnih programih kjer poleg procesorja 8086 sodeluje še koprocesor 8087 je Pascal-86 cca 250 krat hitrejši od Motorola Pascal-a.

Vzroke za zgoraj navedene podatke lahko iščemo v arhitekturni zasnovi procesorja, njegovi ortogonalnosti, simetričnosti in zgradbi, ki podpira visoke programske jezike. Posebno je treba proučiti možnosti za naslavljanje operanda, pozivanje in izstopanje iz procedur, učinkovito kodiranje instrukcij in podobno.



Slika 5: Relativni obseg programov

Navedimo nekatere bistvene razlike vezij 8086 in MC68000:

- naslavljanje operandov: Vsak prevajalnik si rezervira nekatere registre procesorja za izvajanje določenih funkcij visokega programskega jezika. V tabeli 2 je seznam registrov, ki jih uporabljata oba prevajalnika.

FUNKCIJA	PASCAL-86	PASCAL 68000
kazalec sklada	SS + SP	A7
kazalec dinamičnih spremenljivk	SS + BP	A6
kazalec na podatke	DS	A5
kazalec kod	CS + IP	PC

Tabela 2.

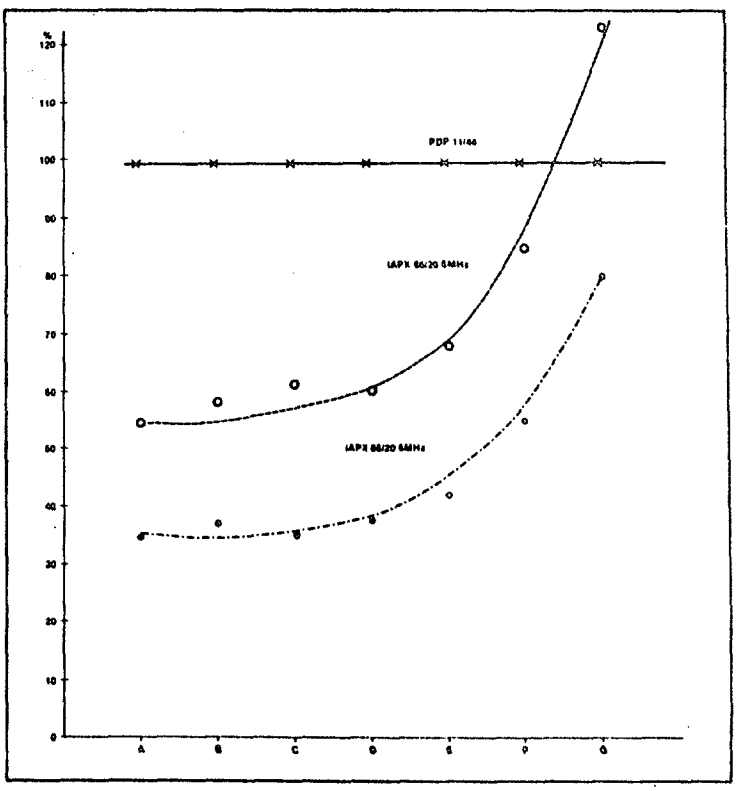
Pascal-86 uporablja specialne registre katerih značilnosti so že določene z zgradbo procesorja. Procesorsko vezje MC68000 pa ima vse registre splošne. Ta razlika v arhitekturi procesorjev omogoča bolj učinkovito kodiranje instrukcij procesorja 8086.

- doseganje podatkovnih polj in struktur: Prevajalnik pogosto uporablja dva indeksna registra in zamik za učinkovit doseg elementov polj in zapisov ker MC68000 ne dopušča uporabe 16-bitnega zamika (le 8-bitni) je pri dvojno indeksnem naslavljanju potrebno pri večjih zapisih uporabiti dodatni naslovni register in dodatno instrukcijo LEA in/ali ADD.

- območje naslavljanja: Zdi se, da veliko območje kontinuiranega naslavljanja MC68000 dopušča več fleksibilnosti, kot segmentna arhitektura procesorja 8086. Vendar je ta prednost zmanjšana zaradi omejenega 8 ali 16-bitnega zamika. Motorola Pascal uporablja register A5 za kazalec podatkov. Zaradi pomanjkanja instrukcije za 32-bitni zamik je zato tudi za MC68000 možna učinkovita generacija kod le za podatke znotraj območja 64 K.

- vstop in izstop iz procedur: Visoki programski jeziki uporabljajo principe modularnosti in uporabljajo procedure (funkcije, podprograme). Zgradba CPU-ja mora biti taka, da učinkovito podpira urejanje dinamičnih podatkovnih struktur pri vstopu in izstopu iz procedure. 68000 uporablja instrukciji LINK in UNLK, vendar so za celotno pripravo vseh parametrov potrebne še dodatne instrukcije. Ker 68000 nima instrukcije RETURN in POP, ki počisti sklad po izstopu iz procedure, je potrebno urejevati sklad eksplicitno, za izstop iz procedure pa je potreben indirekten skok. Instrukcija RETURN in POP procesorja 8086 omogočata učinkovito generacijo kod, ker z eno instrukcijo počisti sklad in prenese kontrolo na instrukcijo, ki sledi klicu procedure.

- kodiranje instrukcij: Ker je to obsežno področje, je tukaj naveden samo en primer. Pri 8086 je dolžina neposrednega operanda neodvisna od velikosti operanda, kar je dokaj pomembno saj prevajalnik pogosto uporablja neposredne operande majhne velikosti. Pri MC68000 pa je velikost takojšnjega operanda enaka velikosti operanda. MC68000 ima le nekaj instrukcij z 4-bitnim neposrednim operandom, ki pa se omejujejo le na instrukcije nad registri.



Slika 8: Hitrost izvajanja iAPX 86/20 : PDP 11/44

1.6.2. Primerjava z minirašunalnikom PDP 11/44

Merjene so bile hitrosti izvajanja sedmih testnih programov v FORTRANU na iAPX 86/20 in na PDP-11/44 z enoto hitrega spomina in z numeričnim procesorjem za računanje v plavajoči vejici.

Rezultati tega testa so prikazani na sliki 8.

Iz rezultatov je razvidno, da 8086 z numeričnim koprocesorjem 8087 razpolaga z lastnostmi, ki so enakega velikostnega razreda kot lastnosti minirašunalnika, katerega cena je veliko večja od cene mikrorašunalnika.

2. RAZVOJ iAPX 86

Firma Intel je razširila ponudbo 16-bitnih mikrorašunalnikov z dvema mikroprocesorjema:

- iAPX 186
- iAPX 286

V članku bi kratko navedli le nekatere glavne značilnosti obeh.

iAPX 186 je namenjen za množično uporabo, karakteristika je nizka cena. Vsebuje približno 10 komponent običajnega sistema iAPX 86. Tako vsebuje tudi 2 DMA kanala, časovnike, krmilnik prekinitev. Je programsko navzgor kompatibilen z iAPX 86.

iAPX 286 je namenjen za zahtevnejše aplikacije, karakteristika je velika zmogljivost (približno 5 krat večja od iAPX 86). Nekatere značilnosti so:

- krmilje za upravljanje pomnilnika,
- večnivojska zaščita programov,
- podpira 1 G zlogov logičnega adresnega prostora in 16 M zlogov fizičnega,
- programska kompatibilnost z iAPX 86.

3. SISTEMSKA PROGRAMSKA OPREMA DRUŽINE 8086

Dejstvo je, da predstavlja programska oprema večji del stroškov pri realizaciji določene aplikacije. Programsko opremo sestavlja tako sistemska kot aplikativna programska oprema. Različne aplikacije mikrorašunalnikov, tu so predvsem mišljeni 16-bitni, zahtevajo dokaj različno programsko opremo, tako glede obsega, kot tudi glede karakteristike.

Namen te točke je prikaz obstoječih najvažnejših operacijskih sistemov (O.S.) za družino 8086, z grobo navedbo karakteristik posameznih sistemov, medtem ko detajlnejša analiza posameznih O.S., ki je potrebna pri realizaciji konkretne aplikacije, presega obseg tega članka, in bo podana kdaj kasneje. Izbira O.S. in druge sistemske programske opreme predstavlja namreč zelo pomembno odločitev, saj običajno zahteva zamenjava O.S. tudi veliko sprememb v aplikativni programski opremi.

Velik korak k poenotenju aplikativne programske opreme je naredil Intel /7/ z definicijo standardnega programskega vmesnika med operacijskim sistemom in ostalimi sistemskimi oz. aplikativnimi programi (universal development interface - UDI).

Ta vmesnik se sestoji iz definicije standardnih sistemskih zahtevkov (27) in formatov podatkov. UDI lahko uporabimo za katerikoli rašunalnik, ker se ne veže na nabor ukazov.

Tabela 3 prikazuje sistemsko programsko opremo za iAPX 86, ki je kompatibilna z UDI.

OPERACIJSKI SISTEM	VISJENIVOJSKI JEZIK
ISIS	PASCAL
iRMX-86	FORTRAN
CP/M-86	BASIC
MP/M-86	C
MP/NET-86	COBOL
XENIX	PL/M
MS/DOS	JOVIAL
OASIS	ADA

Tabela 3.

Izbira operacijskega sistema zavisi od značilnosti aplikacije in od razpoložljive materialne opreme. Glede na zelo širok spekter uporabe 16-bitnih mikrorašunalnikov, lahko postavimo dve skrajni uporabi /8/:

- osebni mikrorašunalniki,
- večuporabniški zmogljivi mikrorašunalniki.

Značilnosti obeh skupin mikrorašunalnikov so prikazane v tabeli 4 /8/.

	OSEBNI	VEČUPORABNIŠKI
Obseg O.S.	do 15 K	do 100 K
Št. uporabnikov	eden	do 16 (ali več)
Obseg pomnilnika	do 48 K	do 16 M
Zunanji pomnilnik	do 160 K	več kot 5 M
Podpora pri razvoju	mala	velika
Podpora pri izvajanju	enostavne datoteke	kompleksne zahteve

Tabela 4.

Za osebne mikrorašunalnike sta se uveljavila predvsem dva operacijska sistema:

- CP/M-86,
- MS-DOS.

Primerjavo obeh sistemov lahko najdemo v literaturi /9/. Značilno za te sisteme je, da ne izkoriščajo vseh zmožnosti 16-bitnega procesorja.

Za zmogljive, večuporabniške mikrorašunalnike so se uveljavili predvsem O.S. na osnovi UNIX-a, to so O.S., ki so ali podobni UNIX-u ali pa razviti iz UNIX-a, najbolj znani so:

- XENIX /8/,
- IDRIS /10/,
- CHROMIX,

Operacijski sistem, ki je večuporabniška nadgradnja O.S. CP/M-86 je:

- MP/M-86,

operacijski sistem namenjen predvsem za poslovne aplikacije je:

- OASIS-86,

operacijski sistem za delo mikroračunalnika v realnem času pa je:

- iRMX-86 /11/.

Vidimo, da imamo širok spekter operacijskih sistemov namenjenih za različna področja uporabe. Podrobnejše značilnosti zgoraj navedenih operacijskih sistemov so navedene v pripisanih člankih in v dokumentacijah razvijalcev.

4. ZAKLJUČEK

V članku so strnjeno zapisane glavne značilnosti materialne in programske opreme Intel-ove družine 16-bitnih mikroračunalnikov 8086, ki bodo morda razvijalca mikroračunalnikov uvedbe v poglobljen študij zgradbe in uporabe 16-bitnih mikroračunalnikov v najrazličnejših možnih aplikacijah. Najtežja je vsekakor izbira ustrezne sistemske programske opreme in izdelava aplikativne programske opreme.

LITERATURA

/1/ The 8086 Family User's Manual, Intel Co., October 1979.

/2/ E.W. Dijkstra: Hierachical Ordering of Sequential Processes, Acta Informatica 1, 1971.

/3/ Systems Data Catalog 1982, Intel Corporation.

/4/ iSBC Applications Manual, AP-28A MULTIBUS Interfacing, Intel Co. 1979.

/5/ M. Moore, J. Crawford, R. Jaswa, K. Shoemaker, H. Kop: iAPX 86 System Benchmark Report, Intel Co., February 1982.

/6/ MC 68000 Microprocessors User's Manual, Motorola Inc. 1980.

/7/ A. Hartman: Standard Software Interface Boosts Program Portability, Electronics, May 19, 1982, 157-159.

/8/ A. Lewis: 16-bit Operating Systems, a Whole New Ball Game, Computer Design, June 1982, 197-204.

/9/ R. Taylor, P. Lemmons: A Comparation of CP/M-86 and MS-DOS, Byte, July 1982, 330-356.

/10/ P.J. Plauger, M.S. Krieger: Unix-like Software Runs on Mini-and Microcomputers, Electronics, March 24, 1981, 125-129.

/11/ J.P. Collins, M.M. Lewitt: An Object Oriented Operating System for Microcomputers, Computer Design, June 1982, 165-172.

PROGRAMIRANJE V ADI II

ANTON P. ŽELEZNIKAR

ISKRA DELTA, LJUBLJANA

UDK: 681.3.06 ADA:519.682

Drugi del članka prinaša kratek priročnik jezika Janus Ada in opisuje leksikalne elemente (identifikatorji, numerične konstante, bazna cela števila, znakovni nizi in konstante, komentar, rezervirane besede), deklaracije in tipe (izpeljani, skalarni, preštevni, znakovni, boolovski, celoštevilski, zlogovni, diskretni, poljski, zapisni, variantni, dostopni tip), imena in izraze (indeksne, izbrane komponente, atributi, izrazi, operatorji, tipska konverzija, kvalificirani izrazi, alokatorji), stavke (sestavljeni, NULL, prireditveni, IF, CASE, LOOP, FOR, WHILE, bločni, EXIT, RETURN, GOTO, ASM stavek), podprograme (parametri, telo, poziv, procedura, funkcija), pakete (specifikacija, telo) in pravila vidljivosti identifikatorjev.
PROGRAMMING IN ADA II. The second part of the article shows a short guide of the Janus Ada Language and describes lexical elements (identifiers, numeric constants, based integers, character constants and strings, comment, reserved words), declarations and types (derived, scalar, enumeration, character, Boolean, integer, byte, discrete, array, record, variant, access type), names and expressions (indexed, selected components, attributes, operators, type conversion, qualified expressions, allocators), statements (compound, NULL, assignment, IF, CASE, LOOP, FOR, WHILE, bločni, EXIT, RETURN, GOTO, ASM stavek), subprograms (parameters, body, call, procedure, function), packages (specification, body), and visibility rules of identifiers.

4. Kratek opis jezika Janus Ada

4.1. Leksikalni elementi

Uporabljajo se vsi ASCII znaki, ki se notranje prevedejo v velike črke; izjema so seveda znakovne konstante in nizi.

Programski simboli so identifikatorji, števila, nizi, znakovne konstante, posebni simboli in konstante. Dodatni presledki se lahko vstavljajo kjerkoli in nimajo vpliva.

Identifikatorji so imena in znak podčrta (´_´) je v njih veljaven.

Numerične konstante so cela in realna števila in v njih smemo uporabljati podčrto, ki ne povzroča učinkov. Numerične konstante v Adi so "literali".

Bazna cela števila lahko imajo bazo med vrednostima 2 in 36. Janus Ada ne dovoljuje baznih realnih števil. Podčrta se lahko uporablja tudi v teh številih in je brez učinka. Pri bazah, večjih od 10, se uporabljajo črke abecede namesto manjkajočih številčnih znakov (črka Z ali z predstavlja številko 35). Npr.

2#1111_0000# ali 16#F0#

je celo število 240 in

32#G9# ali 8#411#

predstavlja celo število 265.

Znakovna (karakterska) konstanta je tiskan znak med enojnima narekovajema, npr.

´a´ ´=´ ´´´ ´ ´

kjer imamo črko a, znaka = in ´ ter presledek.

Znakovni niz je zaporedje nič ali več znakov med dvojnima narekovajema ("). Tako je "" prazen niz (dolžina nič), """" je pa en dvojni narekovaj (dolžina 1).

Komentar začenja z dvema pomišljajema (-) in preneha s koncem vrstice. Učinek komentarja je enak učinku presledka. Npr.

```
-- To je komentar.
END; -- Komentar se lahko začne kjerkoli.
-----------------------------------------
-- Prva dva pomišljaja začenjata komentar.
```

Pragma posreduje prevajalniku določeno navodilo. Pragma je vobče sestavljena iz rezervirane besede PRAGMA, imena pragme in včasih tudi iz argumenta (imena, celega števila ali niza).

V Janus Adi (verzija 1.4.3) imamo tele rezervirane besede:

ALL	AND.	ARRAY	AT
BEGIN	BODY	CASE	CONSTANT
DECLARE	DO	ELSE	ELSEIF
END	EXIT	FOR	FUNCTION
GOTO	IF	IN	IS
LOOP	MOD	NEW	NOT
NULL	OF	OR	OTHERS
OUT	PACKAGE	PRAGMA	PRIVATE
PROCEDURE	RANGE	RECORD	REM
RETURN	REVERSE	SUBTYPE	THEN
TYPE	USE	WHEN	WHILE
WITH	XOR		

Verzija 1.4.5 Janus Ade dodaja tem rezerviranim besedam še

ACCESS	DELTA	DIGIT$	LIMITED

Dodatna rezervirana beseda je ASM (ta ni v standardni Adi). Besede, kot so INTEGER, FLOAT, STRING, CHARACTER, BYTE, FALSE in TRUE, niso rezervirane in jih pišemo z malimi črkami.

Dopustni so tile alternativni znaki: vertikalno črto '|' lahko zamenjamo s '!', znak '#' z dvopičjem ':' in dvojni narekovaj '"' z znakom '%'.

Pogojni prevod Ada vrstice se lahko uporablja tam, kjer imamo na začetku vrstice znak '@'. Prevajanje take vrstice nastopi le, če smo uporabili stavek

```
PRAGMA condcomp(on);
```

na začetku programa s takimi vrsticami.

4.2.Deklaracije in tipi

V Adi je moč graditi nove tipe. Vsak identifikator mora biti deklariran, vendar označitve, bločni in zančni identifikatorji niso deklarirani in so logično ločeni od ostalih identifikatorjev.

Deklaracija tipa uvede ime za dano definicijo tipa. To ime predstavlja tip v deklaracijah spremenljivk, zapisnih polj, formalnih parametrov in v drugih podobnih deklaracijah. Obstajata dve vrsti tipskih deklaracij: za tip in za podtip. Tipska deklaracija uvede nov, različen tip. Podtipska deklaracija oblikuje nov, vendar združljiv tip. Podtipi so na določen način omejeni. Podtip se izvede iz tipa, vendar z dodatnimi omejitvami glede na uporabo tipa (v obliki območja). Združljivi tip in njegova podtipa so npr.

```
TYPE pomn_naslov IS NEW integer;
SUBTYPE naslov IS pomn_naslov
                        RANGE 0..16#2000#;
SUBTYPE prog_naslov IS pomn_nasl;
```

Subtipske deklaracije lahko uporabljajo samo tipske marke in območja (subtype_indication).

Definicija izpeljanega tipa oblikuje nov, nezdružljiv tip z enakimi lastnostmi, kot jih ima izvorni tip, npr.

```
TYPE kazalec IS NEW integer;
-- integer je tu izvorni tip
```

Tako lahko ločimo podatke različnih tipov, ki so bili izvedeni iz istega tipa. Npr.

```
TYPE naslov IS NEW integer;
TYPE kazalec IS NEW integer RANGE 0..max;
-- Izpeljani tipi lahko uvedejo območja
```

V tem primeru naslova ne moremo prirediti kazalcu in obratno. Taka zaščita lahko povzroči vrsto napak.

Skalarni tipi so diskretni in realni. Diskretni tipi so preštevni in celoštevilski ter se uporabljajo za indeksiranje in ponavljanje v zankah. Numerični tipi so celoštevilski in realni. Vsi skalarni tipi so urejeni. Območje vrednosti skalarnega tipa je lahko omejeno. Območje ima obliko L..D, kjer sta L in D enakega tipa. Za ničto območje velja, da je L večje od D. Pridevka tipa T sta T'FIRST in T'LAST (najmanjša in največja vrednost tipa).

Definicija preštevnega tipa uvede množico vrednosti z listo identifikatorjev. Npr.

```
TYPE dan IS (po, to, sr, ce, pe, so, ne);
SUBTYPE delovni_dan IS dan RANGE po .. pe;
```

Tip karakterjev (znakov) (CHARACTER) je preddefiniran in označuje polno množico 128 ASCII znakov. Znakovne konstante so tega tipa.

Preddefinirani boolovski tip je prešteven:

```
TYPE Boolean IS (false, true);
```

Logični in relacijski operatorji vračajo rezultate tega tipa.

Preddefinirani celoštevilski tip vsebuje cela števila v območju -32768 .. 32767. Drugi celoštevilski tipi se oblikujejo z območnimi deklaracijami. Npr.

```
TYPE štev_vrstice IS RANGE 1..5000;
```

Tudi tip zlog je v Janus Adi preddefiniran in ustreza strojnemu zlogu procesorjev 8080A in Z80. To ni celoštevilski tip, je pa z njim združljiv v območju 0 .. 255.

Uporaba Ada območij ni stanovitna, saj ima tri vrste območij, ki so znana kot "subtype indications", "integer type definitions" in "discrete range". Del teh konceptov je tudi "range constraint".

V Adi se najpogosteje uporablja koncept "subtype indication", ko imamo npr.

```
SUBTYPE indeks IS integer RANGE 0 .. 100;
TYPE prevod IS ARRAY (character)
  OF character RANGE 'A' .. 'Z';
-- Tu je 'subtype indication' del, ki se
--    nahaja za besedo OF
```

Diskretno območje se uporablja v zvezi s tipskim imenom ali brez njega, in sicer v deklaracijah indeksov polja, v FOR zankah ter v območjih CASE označitev, npr.

```
FOR i IN 0 .. 10 LOOP
TYPE tabela IS ARRAY ('A'..'Z') OF integer;
WHEN integer RANGE 0..200 =>
```

Primera območnih omejitev (range constraint) pa sta:

```
TYPE nasl IS RANGE 0 .. 201;
```

kar je ekvivalentno

```
TYPE nasl IS NEW integer RANGE 0 .. 201;
```

in

```
TYPE realno IS DIGITS 10
                        RANGE -1.0E30 .. 1.0E30;
```

Pridevki diskretnega tipa T so določeni takole:

T'POS(X)	X mora biti tipa T. Rezultat za POS je pozicijsko število vrednosti X v tipu T. Ta rezultat je kar X, če je T celoštevilski tip oziroma je položaj pri preštevnem tipu.
T'VAL(N)	N mora biti celoštevilskega tipa. Rezultat je vrednost tipa T pri položaju N.
T'SUCC(X)	Ta pridevek vrne naslednika vrednosti X. Napaka nastopi, če X nima naslednika (X = T'LAST).
T'PRED(X)	Ta pridevek vrne predhodnika vrednosti X. Pri X = T'FIRST nastopi napaka.

Dva načina realnega tipa sta tip s fiksno in tip s plavajočo vejico. Pri poslovnih uporabah so priporočljivi tipi s fiksno vejico, pri znanstvenih pa tipi s plavajočo vejico (majhne in velike vrednosti).

Dva tipa s plavajočo vejico sta preddefinirana: "float" in "long_float". Plavajoči tip ima približno 6,5 mest in eksponentno območje med -38 in 38. Dolgi plavajoči tip ima 12 mest in eksponentno območje med 1440 in -1440. Tako imamo npr.

```
TYPE plavajoče IS DIGITS 6
                  RANGE -1.0E38 .. 1.0E38;
TYPE dolgo_plavajoče IS DIGITS 12;
```

Polje je sestavljeno iz liste elementov enakega tipa. Posamezen element je izbran z uporabo indeksne vrednosti. Polje se lahko obdeluje kot skupni predmet ali pa kot zaporedje elementov. Indeks je diskretnega tipa, njegove vrednosti so v določenem območju. Dinamična polja imajo nestatične meje območij, vendar se ne smejo uporabljati v obliki zapisnih (record) polj ali kot elementni tipi drugih polj. Npr.

```
TYPE tabela IS ARRAY (character) OF integer;
TYPE lista IS ARRAY (1 .. 100) OF tabela;
   -- To je polje polj
TYPE dva IS ARRAY
  (barva RANGE belo .. modro, 0 .. 20)
  OF dan;  -- To je dvorazsežno polje
```

Niz v Janus Adi so posebne vrste in niz je enorazsežno polje znakov. Obseg polja je določen z deklaracijo. Elementi niza so dosegljivi z indeksi (tako kot pri poljih), vendar ima niz še dodatne lastnosti:

dolžina: niz ima dolžino trenutno vsebujočih znakov v njem
priredljivost: tipska pravila ne veljajo za nize in različni nizni tipi se lahko medsebojno prirejajo, dokler rezultatni niz ne preseže dolžine
primerljivost: nizi se lahko primerjajo z uporabo operatorjev =, <=, <, >, >=
nizne konstante: nizna konstanta se lahko uporablja kot niz

Npr.

```
ime_zbirke: string(20); -- V imenu je lahko
                        --   do 20 znakov
odgovor: string(80);
```

Največje število znakov je 255.

Zapisni (record) predmet je struktura s poimenovanimi polji, ki so lahko različnih tipov. Zapis se lahko uporablja kot en sam predmet ali po delih. Zapisi v Janus Adi so enaki onim v Pascalu. Zapisi zapisov niso dopusutni. Npr.

```
TYPE datum IS RECORD
  dan: integer RANGE 1 .. 31;
  mesec: ime_meseca;
  leto: integer RANGE 1850 .. 2200;
END RECORD;
TYPE sekljanje IS RECORD
  niz: string (30);
  kazalec: integer RANGE 0..dolžina_simbola;
END RECORD;
```

Variantni zapisi so posebnost jezika Janus Ada. Variantni deli v zapisih določajo alternativne liste polj v odvisnosti od vrednosti posebnega polja, ki ga imenujemo privesek (tag). Na ta način je moč prihraniti pomnilni prostor (neuporabljena polja ne zasedejo prostor) in povečati nazornost zapisne strukture. Variantni privesek mora imeti pravilno vrednost, da se omogoči dostop na specifično polje.

Sintaksa variantnega zapisa je enaka sintaksi CASE stavka, le da je tipsko ime dano pri imenu priveska. Npr.:

```
TYPE oseba IS RECORD
  starost: integer RANGE 0 .. 120;
  narodnost: nar_tip;
  CASE spol: mo_zen IS
    WHEN moski =
      last_avta: boolean;
      poročen: boolean;
    WHEN ženski =
      otroci: integer RANGE 0 .. 20;
      zaposlena: boolean;
    WHEN OTHERS =
      NULL;
  END CASE;
END RECORD;
```

Izbira primera OTHERS določa vrednost, ki ni pokrita z ostalima izbirama. Variantni deli so lahko vgnezdeni, vendar je dovoljena le ena varianta na komponentno listo. Torej je neveljavno tole:

```
TYPE neveljavno IS RECORD
  -- nekaj polj
  CASE a: integer IS
    -- variantni del
  END CASE;
  CASE b: boolean IS
    -- variantni del
  END CASE;
END RECORD;
```

Tipi dostopa (dostopni tipi) in tipi pascalskih kazalcev so si podobni ter omogočajo dinamično oblikovanje predmetov. Predmeti, ki so bili oblikovani z deklaracijami ali alokatorji, se lahko navajajo z imeni; pri dostopnih tipih te možnosti ni.

Dostopni tip kaže le na tisti tip predmetov, za katerega je bil dostopno opredeljen. Ta strogost ohranja lastnost doslednega preizkušanja tipov v Adi. Dostopna vrednost NULL označuje, da ni mogoče dostopiti k nobenemu predmetu, vendar se dostopne spremenljivke implicitno (avtomatično) ne inicializirajo na vrednost NULL. Dostopna vrednost se lahko svobodno prireja dostopnim spremenljivkam enakega tipa. Vse dostopne spremenljivke, ki imajo enako vrednost, navajajo enak predmet (tudi pri spremembi predmeta). Npr.:

```
TYPE kazal IS ACCESS integer;
  a, b: kazal;
  ...
  BEGIN
    a := NEW integer; -- to je alokator,
                      -- glej kasneje
```

```
    a.ALL := 6;
    b := a;
    b.ALL := 10;
      -- tu postane a.ALL = 10 (ne ostane 6)
...
```

Deli predmeta, na katerega kaže dostopni tip, lahko vsebujejo dostopne tipe enakega ali različnega tipa. To omogoča oblikovanje rekurzivnih in medsebojno odvisnih dostopnih predmetov. Ker morajo biti vsi tipi deklarirani pred uporabo, se lahko vzame nepopolna tipska deklaracija le tedaj, ko imamo deklaracije dostopnega tipa. Npr.:

```
TYPE stev_vrstice; -- nepopolna tipska
                   --   deklaracija
TYPE kazal_vrst IS ACCESS stev_vrstice;
TYPE stev_vrstice IS RECORD
     vrstica: integer;
     naslednja: kazal_vrst;
END RECORD;

TYPE beseda; -- nepopolna tipska deklaracija
TYPE kazalec IS ACCESS beseda;
TYPE beseda IS RECORD
     vrednost: string (20);
     levo, desno: kazalec;
     vrst_stev: kazal_vrst;
END RECORD;
```

D e k l a r a t i v n i d e l je lista deklaracij, ki se nanašajo na določen del programa. Deklarativni deli se pojavljajo v blokih, podprogramih in paketih. Predmeti morajo biti deklarirani pred njihovo uporabo. Če se podprogramska deklaracija pojavi v deklarativnem delu, se mora podprogramsko telo pojaviti v istem deklarativnem delu. Janus Ada ne zahteva posebnega vrstnega reda deklaracij v deklaracijskem delu. V Adi morajo podprogramska telesa slediti vsem drugim deklaracijam.

4.3. Imena in izrazi

I m e n a predstavljajo predmete, ki jih je deklariral uporabnik ali sistem. Imena so spremenljivke, tipi, podprogrami in paketi. Druga imena predstavljajo predmetne komponente, atribute (pridevke) in označitve. Imena, ki so funkcijski pozivi, so opisana kasneje. Najenostavnejša oblika imena je identifikator, ki je bil določen z deklaracijo. Npr.:

```
indeks  -- ime skalarne spremenljivke
mreža   -- ime poljske spremenljivke
pomično -- ime tipa
abs     -- ime funkcije
```

I n d e k s n a k o m p o n e n t a je lahko element polja le v Janus Adi in je ime z listo izrazov v oklepajih. Ime je določeno z imenom polja, izrazi pa so indeksi komponent. Indeksi morajo biti enakega tipa, kot so oni v polju. Npr.:

```
tabela(i)    -- element enorazsežnega polja
mreža(1,j+1) -- element dvorazsežnega polja
polje_polj(x)(y)  -- element tega polja je
                  --   sam polje
```

Vendar Janus Ada ne dopušča dostopa do predmetov polja (kazalec kot dostopni tip) prek indeksnih komponent.

R e z i n e v Janus Adi še niso implementirane.

I z b r a n e k o m p o n e n t e se uporabljajo pri izbiri zapisnih polj in za izbiro predmetov v paketih. Izbrana komponenta je ime s piko in identifikatorjem. Če je ime pred piko zapis, določa identifikator za piko polje zapisa. Če je zapis variantni zapis, lahko identifikator polja predstavlja polje v variantnem delu. Vsako dostopljeno polje v variantnem delu naj bi obstajalo skladno s trenutno vrednostjo privesnega (tag) polja. Vendar prevajalnik za Janus Ado nima variantnega preizkusa in nelegalne variante ne povzročajo signalov napake. Kadar je ime pred piko paket, določa identifikator za piko predmet v paketu. Primeri izbranih komponent so tile:

```
oseba.priimek  -- komponenta zapisa
oseba.kraj.populacija  -- komponenta zapisa
tabela_sekljanja(simbol).dolžina  -- kom-
                              -- ponenta zapisa
sklad.vrh  -- spremenljivka paketa
sklad.prazno()  --  poziv funkcije v paketu
```

Za zapisno strukturo z vgnezdenimi zapisnimi strukturami mora biti navedeno ime za vsako ravnino, ko se poimenuje vgnezdena komponenta.

Izbrana komponenta se lahko uporabi za dostop k zapisnim poljem in/ali za dostop k celotnemu predmetu, na katerega kaže dostopna vrednost. Z obliko identifikator.ALL je določen celoten predmet, na katerega kaže dostopna spremenljivka. V času poteka se pojavi sporočilo o napaki, če je imela dostopna spremenljivka vrednost NULL.

A t r i b u t i določajo lastnosti poimenovanih predmetov. Atribut je ime, ki mu sledi apostrof in identifikator brez ali z argumenti. Atributi niso rezervirani, uporabnik jih ne more deklarirati. Npr.:

```
barva'pred(i)   --  vrednost pred vrednostjo
                --  'i' v tipu barva
tip_kazalec'first  --  najmanjša vrednost v
                   --  tipu tipskega območja
integer'size    --  število bitov v predsta-
                --  vitvi celega števila
```

Atributi jezika Janus Ada so tile:

```
X'address, T'size, T'first, T'last,
T'pos(X1), T'val(Z), T'pred(X1), T'succ(X1),
P'location
```

Tu je X predmet ali podprogram, T je tip ali podtip, X1 je tipa T, Z je tipa celo število in P je ime trenutnega paketa.

L i t e r a l označuje eksplicitno vrednost danega tipa. Število ali preštevni literal označuje vrednost skalarnega tipa. Primeri so:

```
10  --  celoštevilski literal
rdeče  --  preštevni literal (barva)
byte(11)  --  konverzija celoštevilskega li-
          --  terala v tip BYTE
'a'  --  znakovni literal
```

A g r e g a t i še niso implementirani v Janus Adi.

I z r a z je formula, ki določa izračun vrednosti; ta je numerična ali nenumerična. Izraz je sestavljen iz operandov in operatorjev. Operandi imajo vrednost določenega tipa. Dovoljeni operandi so imena, literali, tipske konverzije in izrazi v oklepajih. Če je operand ime, mora biti spremenljivka, konstanta, funkcijski poziv, atribut ali komponenta polja ali zapisa; ne more pa biti proceduro, tipsko ali paketno ime. Tip izraza je odvisen od tipov operandov in operatorjev. Tip, ki ga producira operator, je odvisen od tipov operandov. Primeri operandov so:

```
4.0  -- število
globina  -- vrednost spremenljivke
dolžina(x)  -- funkcijski poziv
(števnik + 10)  -- izraz v oklepajih
```

Primeri izrazov so:

```
globina  -- operand
dolžina(x)**3  -- potenciranje
obseg_tabele MOD obseg_odseka  -- multipli-
                               -- kativni operator
-5.1  -- unarni minus
NOT globalno  -- unarni operator
ključ = "JANUS"  -- relacijski operator
i IN 1 .. 10  -- operator pripadnosti
(a AND b) OR c  -- logični izraz
x**2 - 5.2*u*v  -- sestavljen izraz
```

O p e r a t o r j i, ki se uporabljajo v izrazih, so zbrani v dodatku A (v prvem delu članka) in njihove uporabe ne bomo posebej opisovali.

T i p s k a k o n v e r z i j a je dovoljena med nekaterimi tipi, in sicer med vsemi numeričnimi in med tipi, ki imajo skupen starševski tip. Primeri so:

```
svetla_barva(črno)  -- napaka območja
float(i)  -- "i" je celo število
integer(pi)  -- dobimo vrednost 3
```

Janus Ada ne dovoljuje tipske konverzije IN OUT in OUT parametrov.

K v a l i f i c i r a n i i z r a z i povzročijo, da postane tip izraza enak določenemu tipu ali podtipu. Tu ni konverzije. Npr.:

```
barva'(modro)  -- tip za modro je barva
pomnas'(1000)  -- tip za 1000 je pomnil-
               -- niški naslov
```

A l o k a t o r (dodeljevalnik) oblikuje nov predmet, ko je izvršen in vrne dostopno vrednost za navajanje predmeta. Janus Ada ne pozna inicializacije dodeljenega (alociranega) predmeta. Npr.:

```
a := NEW znak;
bolek := NEW beseda;
```

Alokator dodeli pomnilnik za novi predmet iz podatkovne strukture, imenovane kup (gomila, angl. heap). Kup ni urejen, kot sta sklad in polje. Kup se uporablja v sistemu za različne namene, npr. pri vračanju velikih predmetov iz funkcij, zbirk, kot začasni pomnilnik za različne tipe parametrov. Pri tem se lahko pojavi prestop obsega kupa. Kup in sklad rasteta v pomnilniku eden proti drugemu. Do prestopa obsega kupa lahko pride pri preobsežni rekurziji. Janus prevajalnik ima proceduro 'dispose' (odpravi), ki vrne uporabljeni pomnilni prostor v kup. Ta procedura se mora poklicati za vsak odprevljeni predmet. Velja:

```
PROCEDURE dispose (kazalec: IN OUT
                        določen_dostopni_tip);
```

'dispose' postavi kazalec na NULL po vrnitvi.

S t a t i č n e oziroma l i t e r a l n e izraze izračuna že kompilator; v njih so dovoljeni tile predmeti:

- številske in znakovne konstante
- imenovane konstante in preštevni literali
- unarna operatorja + in - ter binarni operatorji +, -, *, /, MOD in REM
- atributske lokacije, obseg in naslov (če je predmet statično dodeljen)
- atributi pos, val, succ in pred, če je argument statičen izraz
- tipske konverzije in kvalificirani izrazi statičnih izrazov
- relacijski operatorji <, <=, >, >=, =, /=

V statičnih izrazih ni dovoljena uporaba logičnih operatorjev NOT, AND, OR, XOR in operatorja pripadnosti. Bilo bi povsem neobičajno, če bi se ti operatorji pojavili v popolnoma konstantnih izrazih. Primeri statičnih izrazov so:

```
13 + 29
(člen + 1)/obseg  -- tu sta člen in obseg
                  -- poimenovani konstanti
člen > 200
obseg*množica'size  -- obseg je poimenovana
                    -- konstanta
integer(člen)
naslov'(16#110#)
character'val(11)
```

4.4. Stavki

Programi so sestavljeni iz s t a v k o v. Stavki so enostavni (prireditveni) in kompleksni (IF stavek). Kompleksen stavek vsebuje več stavkov. Stavki so lahko označeni in o z n a č i t e v je zaprta v dvojne kotne oklepaje, npr.:

<< označitev >>

Označitve se uporabljajo samo v povezavi z GOTO stavki in morajo biti enkratne (enoumne).

N U L L s t a v e k je brez učinka ter se ga vstavlja namesto nekega drugega stavka, ki naj bi se tam pojavil. Takšna mesta so IF veje, CASE udi in LOOP telesa. NULL stavek lahko uporabimo tudi ob označitvi.

P r i r e d i t v e n i s t a v e k nadomesti trenutno vrednost spremenljivke z novo vrednostjo, določeno z desnim izrazom. Spremenljivka in izraz morata biti enakega tipa in vrednost izraza mora biti v območju podobmočja spremenljivke. Npr.:

```
i, j: integer RANGE 1 .. 6;
k, m: integer RANGE 1 .. 23;
j := i;  -- identični območji
k := j;  -- združljivi območji
j := k;  -- preizkus je možen le v času
         -- izvajanja ter lahko povzroči
         -- napako območja
```

P r i r e d i t v e p o l j a so enake drugim prireditvam, tipi se morajo natanko ujemati.

N i z n e p r i r e d i t v e so dopustne le v Janus Adi. Namenska spremenljivka mora imeti dovolj prostora. (V Adi so dolžine enake.)

I F s t a v e k omogoča izbiro zaporedja stavkov v odvisnosti od vrednosti enega ali več pogojev. Izraz v pogoju mora biti boolovski. Imamo navadni THEN, ELSE in ELSEIF del. Npr.:

```
IF akcija = redukcija THEN
   reduciraj;
ELSEIF akcija = pomik THEN
   pomakni;
ELSE
   reduciraj_delno;
END IF;

IF(indeks < maks) AND (si(indeks) = ' ')
THEN indeks := indeks+1;
ELSE
   napaka(1);  -- prestop
END IF;

IF niz = "" THEN
   IF program = janus THEN  -- vgnezdeni IF
```

```
      piši(test);              --   stavek
    ELSEIF znak = 'A' THEN
      beri(test);
    END IF;
END IF;
```

C A S E s t a v e k izbere eno od več alternativnih stavčnih zaporedij. Izbira je odvisna od vrednosti izraza na začetku CASE stavka. CASE stavek se uporablja namesto zaporedja IF stavkov oziroma njihovih delov. Npr.:

```
CASE znak IS
  WHEN 'A'..'Z' => NULL;
  WHEN 'a'..'z' => veliko(znak);
  WHEN '0'..'9' => število(znak);
  WHEN '.'      => konec_programa;
  WHEN ',' | ';' => ločilo(znak);
  WHEN OTHERS   => posebno(znak);
END CASE;
```

Ta CASE stavek je okrajšava za:

```
IF     znak IN 'A'..'Z' THEN
  NULL;
ELSEIF znak IN 'a'..'z' THEN
  veliko(znak);
ELSEIF znak IN '0'..'9' THEN
  število(znak);
ELSEIF znak = '.'       THEN
  konec_programa;
ELSEIF (znak = ',' OR znak = ';') THEN
  ločilo(znak);
ELSE
  posebno(znak);
END IF;
```

Izraz mora biti diskretnega tipa. CASE označitve morajo biti konstante, lahko so iz diskretnega območja, kjer so meje statični izrazi. Če se ne uporabi OTHERS za nenavedene možnosti, se v času izvajanja pojavi napaka. Če pa se OTHERS uporabi, mora biti zadnje in le samo v CASE stavku.

L O O P s t a v e k ima tri stavčne oblike. Osnovna zanka se konča z uporabo EXIT stavka, ki je kjerkoli v zanki. Npr.:

```
LOOP  -- repeat stavek v Pascalu
  IF znak IN 'a'..'z' THEN  -- spremeni v
                            -- velike črke
    znak := character'val
                (character'pos(znak)-32);
  END IF;
  znak := naslednji_znak();
  EXIT WHEN NOT(znak IN '0'..'z');
                        --  until v Pascalu
END LOOP;
```

F O R z a n k a je osnovna zanka z iteracijskim delom, ki določa število ponovitev. Pri navadni obliki se zančni števnik povečuje, lahko pa ga tudi dekrementiramo z uporabo besede REVERSE. Parameter zanke je bralna spremenljivka in ne more biti IN OUT ali OUT parameter. Npr.:

```
i, j: integer;
BEGIN
  j := 0;
  vsota: FOR i IN 1 .. 179 LOOP
           j := j + 1;
         END LOOP vsota;
END;

FOR i IN REVERSE 1..13 LOOP  -- dekrement.
  FOR zn IN character RANGE 'A'..'Z' LOOP
    -- inkrementiranje
    --  'zn' je tipa character, 'i' tipa
    --      integer
    -- sledijo stavki zanke
  END LOOP;
END LOOP;
```

```
SUBTYPE malo IS integer RANGE 1 .. 10
    -- deklaracije in stavki
FOR števnik IN malo LOOP
    -- števnik teče od 1 do 10
    -- sledijo stavki
END LOOP;
```

Če se zančni identifikator uporablja, kot npr. 'vsota', potem mora ta identifikator slediti 'END LOOP'.

Tretja oblika osnovne zanke je W H I L E z a n k a . WHILE pogoj se preizkusi pred izvršitvijo zanke. Npr.:

```
WHILE a(i).dell /= vrednost LOOP
   IF i < maks THEN
     i := i + 1;
   ELSE
     EXIT;
   END IF;
END LOOP;
```

Uporaba zančnih identifikatorjev sicer ni potrebna, je pa zlasti pri vgnezdenih zankah koristna, ko izstopamo iz notranjih zank v zunanje. Npr.:

```
zunanja: FOR i IN 4..13 LOOP
  srednja: FOR i IN 0..99 LOOP
    notranja: LOOP
                k := zunanja.i + i;
                EXIT notranja
                  WHEN k >= zunanja.i * i;
                EXIT zunanja;
              END LOOP notranja;
              EXIT;
            END LOOP srednja;
          END LOOP zunanja;
```

Z b l o k i lahko ločimo zaporedje stavkov in deklaracij od glavnega programskega poteka brez uporabe posebne procedure. V primeru uporabe bločnega identifikatorja na začetku bloka, moramo tega uporabiti tudi za besedo 'END', ki blok končuje.

Blok se lahko pojavi na mestu stavka. Blok lahko nadomesti enkratno uporabo procedure in tako časa izvajanja ne povečuje s procedurnim pozivom. Vendar v bloku deklarirane spremenljivke niso dosegljive prek blokovnega imena.

E X I T s t a v e k povzroči končanje (njega) obdajajoče zanke. Izstop se nanaša na najbolj notranjo zanko, ki ta EXIT stavek obdaja, če seveda ni drugače določeno (npr. za EXIT stavkom v prejšnjem primeru ali s pogojem). Npr.:

```
preizkus: LOOP
            j := j + 1;
            IF i = j THEN
              EXIT preizkus;
            END IF;
            EXIT preizkus WHEN i = j;
          END LOOP preizkus;
```

IF stavek v gornjem primeru ima enak učinek kot EXIT stavek pod njim.

R E T U R N s t a v e k povzroči takojšnje končanje obdajajoče funkcije ali procedure. Funkcija mora vrniti vrednost, proceduri pa je ni treba. Vrnitev ne more prenesti krmiljenja izven paketnega telesa. Funkcija se mora končati z RETURN stavkom, proceduri pa tega ni treba. Npr.:

```
PROCEDURE zapusti IS
BEGIN
  IF pravilno THEN
    RETURN;
  END IF;
END zapusti;
```

```
FUNCTION je (tip: kazalec) RETURN boolean IS
BEGIN
  RETURN bazni_tip(tip) = 1;
END je;
```

G O T O s t a v e k ima za posledico takojšnji prenos krmiljenja na označeni stavek. Ta stavek in pripadajoči GOTO stavek morata biti v istem podprogramu. Npr.:

```
<<spet>>
IF vozlišče(i) < ime THEN
   IF levo(i) /= 0 THEN
      i := levo(i);
      GOTO spet;
   END IF;
   -- dodatni stavki
END IF;
```

Uznačitev se tu lahko pojavi samo pred stavkom (ne pred delom stavka). Nepriporočljivo je skakanje v bloke, IF, CASE in LOOP stavke z uporabo GOTO stavkov. Janus Ada ne omogoča skokov v bloke in zanke.

A S M s t a v k i so posebnost Janus Ade in omogočajo medvrstično vstavitev binarnih podatkov (v kodni del objektnega izhoda). Tako se lahko vstavijo podatkovne tabele in strojni kod v Janus Ada program. Pravilo ASM stavka je:

```
asm_statement ::=
ASM static_expression {, static_expression};
```

Izraz celoštevilskega tipa mora izračunati prevajalnik in podatki se vstavijo v kodni tok brez interpretacije. Pri podatku z vrednostjo med 0 in 255 se generira en sam zlog. Npr.:

```
ASM lhld, nasl'address; -- 'lhld' je opera-
    -- torski kod 16#2A#(procesor 8080A)
ASM 0,1,2,3;   -- vstavi zloge 0, 1, 2, 3
```

Najbolj uporabljana atributa sta "address" in "location". V X'address je X vobče predmet ali podprogram. Ta pridevek vrne začetni strojni naslov predmeta X. Rezultatni tip je celo število. Za address pridevek v ASM stavku morajo biti izpolnjeni tile pogoji: če je X predmet, mora biti X deklariran kot paketni predmet. Naslov (X) mora biti deklariran kot paketna spremenljivka na najbolj zunanji ravnini programa. Če je X podprogram, mora ASM stavek slediti ali biti vključen v podprogramsko telo. Podprogramski naslov se lahko uporabi le, če je podprogramsko telo že bilo prevedeno in vnaprejšnje deklaracije so tu brez učinka.

V P'location je P ime trenutnega paketa. P'location vrne trenutni lokacijski števnik za izhodni izvirni kod. V ASM stavku omogoča ta pridevek "relativne" skoke in je v grobem ekvivalenten pomenu '$' v zbirnem jeziku. Skoki na Janus Ada označitve niso implementirani (možna je uporaba GOTO stavkov med ASM stavki).

ASM stavki se uporabljajo le za krajša kodna zaporedja. Daljše rutine v zbirnem jeziku lahko oblikujejo zbirni paket in se uporabijo na zbirni ravnini.

4.5. Podprogrami

Podprogram je procedura ali funkcija. Procedurni poziv je stavek, funkcija pa vrne vrednost in se uporablja kot ime v izrazih.

P o d p r o g r a m s k a d e k l a r a c i j a določa proceduro ali funkcijo in je ekvivalentna vnaprejšnji procedurni deklaraciji v Pascalu. S to deklaracijo se določi ime podprograma, njegovi formalni parametri (če obstajajo) in tip vrnjene vrednosti (če ta možnost obstaja). Dejanski kod, ki temu podprogramu pripada, se deklarira v podprogramskem telesu. Podprogramska deklaracija je potrebna le tedaj, ko se podprogram uporabi, preden je bilo njegovo telo definirano (npr. v paketu ali v rekurzivnih programih). Primeri podprogramskih deklaracij so:

```
PROCEDURE preišči_drevo;
PROCEDURE vejitev(list: OUT vozlišče);
FUNCTION skup_imenovalec (m,n: integer)
                          RETURN integer;
FUNCTION križni produkt (x,y: matrika)
                          RETURN real;
```

Identifikator za formalni parameter se lahko uporabi le enkrat. Zaradi tega je

```
PROCEDURE napaka (dobro, slabo: IN boolean;
                  slabo: OUT integer);
```

nepravilno, ker se "slabo" pojavi dvakrat.

Janus Ada podprogrami se lahko kličejo rekurzivno in imajo lastnost ponovnega vstopa (v zbirnem jeziku ta druga lastnost vobče ne velja za subrutine in jo moramo programsko zagotoviti).

F o r m a l n i p a r a m e t r i podprograma so lokalne spremenljivke ali konstante. Ti parametri lahko imajo tri načine: IN, OUT in IN OUT. IN parameter dobi vrednost z dejanskim parametrom; OUT parametru, ki ustreza dejanskemu parametru, se priredi rezultatna vrednost. IN OUT parameter ima lastnost IN in OUT parametra.

P o d p r o g r a m s k o t e l o določa izvršljivi kod podprograma. Pri uporabi vnaprejšnje deklaracije mora biti v celoti ponovljena podprogramska specifikacija. Npr.:

```
PROCEDURE vnaprej(i: integer);

PROCEDURE primer IS
BEGIN
   vnaprej(1);
   -- nadalnji stavki
END; -- nimamo opcijskega identifikatorja

PROCEDURE vnaprej(i: integer) IS
BEGIN
   -- stavki procedure
END vnaprej;
```

Pri vsakem podprogramskem pozivu se oblikuje povezava med formalnimi in dejanskimi parametri, nakar se izvršijo stavki telesa. Podprogram se konča z RETURN stavkom ali s končnim stavkom podprograma. Po izvršitvi telesa se parametri načina OUT in IN OUT kopirajo v dejanske parametre in krmiljenje se vrne v točko, iz katere je klicajoči program pozval podprogram.

P o d p r o g r a m s k i p o z i v se nanaša na proceduro ali funkcijo. Ta poziv določa korespondenco med dejanskimi in formalnimi parametri. Dejanski parameter je spremenljivka, konstanta ali vrednost izraza. Poziv brezparametrične funkcije ima prazen del v oklepajih. Npr.:

```
procedurni_poziv; -- ni parametrov
proc_s_parametri(indeks, 1, ab);
odgovor := vprašanje(stavek1, stavek2);
           -- "vprašanje" je funkcija
vozlišče := nasl_vozlišča();  -- ni pa-
                              -- rametrov
vsota := seštej(delna_vsota)*2;
```

D e j a n s k i p a r a m e t e r IN OUT in

OUT načina mora biti spremenljivka (navadna, indeksirana, izbrána) in ne sme biti tipska konverzija kake spremenljivke. Izrazi s tipsko konverzijo so dovoljeni kot dejanski parametri za IN način. Izraz se izračuna pred dejanskim pozivom. Dejanski in formalni parametri morajo biti enakega tipa.

F u n k c i j a je podprogram, ki vrne vrednost. Funkcijski parametri so lahko le IN načina. Funkcija se konča z RETURN stavkom, ki vrne vrednost funkcije. Če se funkcija konča s svojim koncem, njena vrednost ni določena. To možnost funkcijskega končanja moramo preprečiti. Npr.:

```
FUNCTION prazno (znak: IN character)
                          RETURN boolean IS
BEGIN
   RETURN znak := ' ';
END prazno;
```

Vrnjena vrednost funkcije je lahko poljubnega tipa in vračajo se lahko tudi neskalarne vrednosti (npr. nizne).

4.6. Paketi

P a k e t i se uporabljajo za združevanje podatkov in operacij, ki logično spadajo skupaj. Uporabimo pa jih lahko tudi za prikrivanje (zasebnost) tiste informacije, ki je za programerja nebistvena; ta način uporabe imenujemo podatkovna abstrakcija.

Paketi se uporabljajo pri realizaciji ločenega prevajanja v Janus Adi. Zaradi ustreznih Ada deklaracij lahko prevajalnik za Janus Ado avtomatično naloži in uporablja predmete paketa. Tako obstaja varnost tipskega preizkušanja tudi pri ločenem prevajanju. Pri vrsti pascalskih prevajalnikov te lastnosti pri ločenem prevajanju paketov nimamo.

Paket ima dva dela: specifikacijo in telo. Paketno ime telesa in specifikacije se morata ujemati z imenom pripadajočih zbirk. Vgnezdenja paketov niso dovoljena. Le eno paketno telo ali specifikacija se lahko nahaja v zbirki izvirnega teksta.

P a k e t n a s p e c i f i k a c i j a je del paketa, ki je viden za druge pakete, ločeno prevedene. Specifikacija obkroži podatkovne definicije in podprograme za obdelavo, hkrati pa skrije implementacijske (programske) podrobnosti v paketnem telesu. Prevod paketne specifikacije generira v Janus Adi informacijsko zbirko za uporabo v drugih paketih; ta zbirka ima poseben sufiks '.SYM' .

Paketno telo (če obstaja) ima sufiks '.PKG' . Specifikacijska zbirka naj bi imela sufiks '.LIB' , tako da je ne bi zamenjevali s SYM zbirko. Informacijo paketne specifikacije lahko uporablja več programov. Specifikaciji ni treba določevati podprogramov, določuje pa lahko množico konstant, tipov in spremenljivk, kot to narekujejo potrebe drugih paketov. V tem primeru ni potreben obstoj paketnega telesa, saj izvršljivi kod ni specificiran.

Paketna specifikacija, ki določa podprograme za druge pakete, mora imeti paketno telo in obe imeni se morata ujemati, tako da prevajalnik lahko avtomatično naveže paketno telo na specifikacijo. Zaradi tega mora biti specifikacija prevedena pred pripadajočim telesom. Paketna specifikacija lahko definira največ 255 imen. Primer:

```
PACKAGE specif IS  --  ni besede BODY
  -- ta specifikacija nima paketnega telesa
  maks_obseg: CONSTANT := 101;
  TYPE obs IS integer RANGE 0 .. maks_obseg;
  TYPE spr IS (nič, ena, dve, tri);
  točka, teme: obs;
END specif;

WITH specif;  --  vzemi "specif" informacijo
PACKAGE spec IS
  --  ta specifikacija mora imeti telo
  maks_celo: CONSTANT := 32767;
  TYPE zapis IS RECORD
    poljel: specif.obs;
    polje2: integer;
  END RECORD;

  PROCEDURE naj_zap(a_obs: specif.obs;
                    a_celo: integer);
  FUNCTION glej_obs(zap_v: zapis)
                    RETURN specif.obs;
END spec;
```

P a k e t n o t e l o je del paketa z deklariranimi izvršljivimi stavki. Glavni programi v Janus Adi morajo biti paketna telesa. Pomnilni prostor se paketnim predmetom dodeljuje statično, tako da se njihove vrednosti ne zgubijo med klicanjem paketnih funkcij in procedur.

Ime paketa se mora u j e m a t i z imenom zbirke, tako da prevajalnik in povezovalnik lahko delujeta avtomatično. Če vsebuje paketno telo paketno specifikacijo, potem mora biti ta specifikacija prevedena pred telesom. Primer:

```
WITH obratno, prenosno; -- Tu imamo omejitev
   -- dolžine imen na 8 znakov (CP/M zbirke)
PACKAGE BODY preizkus IS
   -- Primer glavnega programa
   TYPE tabela IS ARRAY (1 .. 10, 1 .. 10)
                                 OF integer;
   mult_tab: tabela;
BEGIN
   FOR i IN 1 .. 10 LOOP
      FOR j IN 1 .. 10 LOOP
         mult_tab(i,j) := j*i;
      END LOOP;
   END LOOP;
   --  Dodatni stavki
END preizkus;
```

Paketno telo ne potrebuje stavčnega zaporedja in tako zaporedje se največkrat uporabi za inicializacijo glavnega programa. Uporabnik lahko oblikuje podprogramski paket v obliki knjižnice.

Zasebni tipi se uporabljajo za podatke podprogramov, za prikrivanje predmetnih podrobnosti uporabniku. Cilj tega je zaščita ali abstrakcija. Deklaracija zasebnega tipa se lahko pojavi le v paketni specifikaciji. Imamo dve obliki zasebnih deklaracij: PRIVATE in LIMITED PRIVATE.

Deklaracija zasebnega tipa povzroči, da postane tipsko ime dosegljivo za druge pakete, vidljivost pa ostane ohranjena le v paketu s to deklaracijo. Zasebni tipi se lahko uporabljajo le za deklaracijo, prireditev in primerjavo na enakost in neenakost.

Zunaj določujočega paketa je zaseben tip le zaseben. Drugi paketi ne poznajo dejanskega tipa, kar pa ni pomembno. Drugi paketi tako ne morejo uporabljati te informacije. Ni pa dopustna uporaba zasebnega tipa v paketih, kot je npr.:

- deklariranje imenovane konstante zasebnega tipa
- inicializacija spremenljivke ali parametra z literalom
- dereferenciranje zasebnega tipa kot zapisa

O m e j e n i z a s e b n i t i p i so podobni zasebnim, le da so bolj omejeni pri uporabi. Velja:

- prireditve in primerjave niso dopustne
- deklaracija spremenljivke tega tipa ne sme imeti inicializacije
- konstante tega tipa ne morejo biti deklarirane

Te omejitve se razširjajo še na polja in zapise, če ti vsebujejo elemente ali polja omejenega zasebnega tipa. Npr.:

```
PACKAGE osebje IS
   -- Ta paket predstavlja varen način pre-
   -- gledovanja kandidatov za zaposlitev.
   -- Dejansko ime in biografski podatki so
   -- za pregledovalca maskirani.

   TYPE oseba IS LIMITED PRIVATE;
   TYPE delodajalec IS LIMITED PRIVATE;
   TYPE izkušnje IS LIMITED PRIVATE;
   TYPE zdravje IS (trdno, netrdno);
   TYPE poklic IS (teh,e_inž,s_inž,kem,fiz);

   FUNCTION poišči_osebo (delo: poklic)
                              RETURN oseba;
     -- Vrni potencialnega kandidata za do-
     -- ločeno delo

   PROCEDURE natisni_pregled (kdo: oseba);
     -- Natisni pregled podatkov za zaposli-
     -- tvenega kandidata

PRIVATE
   TYPE oseba IS integer;
   TYPE delodajalec IS NEW oseba;
   SUBTYPE izkušnje IS niz(255);
END osebje;
```

4.7. Pravila vidljivosti

S posebnimi pravili je določeno *območje*, v katerem se identifikator lahko uporablja. Zamisli različnih vidljivosti v sodobnih programirnih jezikih predstavljajo spekter, ki ga lahko ponazorimo s tole shemo:

```
<------------spekter vidljivosti------------>

 zbirnik     C          Pascal     Ada      ?
 BASIC                  Algol
 COBOL       FORTRAN                PL/I
```

Ta spekter se razprostira od enostavnega do zapletenega, kjer enostavno pomeni eno samo definicijo identifikatorja, kompleksno pa dopušča več načinov za več pomenov identifikatorja.

Koncepta območja in vidljivosti sta v bistvu enaka. To, kar je v območju podprograma lokalno, je vidno samo zanj; globalno je po definiciji vidno tudi izven območja procedure. V novem območju lahko ima identifikator drug (lokalen, globalen) pomen. Podobno kot za procedure velja tudi za bloke. Pri zapisih lahko uporabljamo še operacije s piko, ko dostopamo k poljem zapisa. Npr.:

```
A_zapis.notr := 1; A_zapis.znak := '%';
A_zapis.zastavica := A_zapis.notr > malo;
```

Zapis je območje in podobno velja tudi za paket. Imena s piko lahko tako razširimo od zapisa na poimenovane pakete, podprograme, bloke, zanke (vse te možnosti v Janus Adi še niso implementirane). Na ta način postane lokalna spremenljivka tudi globalno dostopna. Ta način dostopanja k maskiranim (lokalnim) imenom omogoči nazadnje tudi ločeno prevajanje paketov. Tudi WITH stavek omogoča uporabniku dostop do imen iz paketne specifikacije z uporabo izrazave s piko.

Zapisi s piko so zelo pripravni za dobivanje enoumnih imen sicer maskiranih predmetov, postanejo pa zelo obsežni pri ločenem prevajanju. Zaradi te neprimernosti se uporablja s t a v č n i č l e n U S E . Ta člen omogoča uporabniku dostop do predmetov iz paketa z USE deklaracijo, kot da bi bili ti predmeti lokalni.

USE člen ima le nekaj omejitev. USE člen ne maskira drugih definicij v trenutno kompilirani enoti; to velja za imena iz drugih paketov z USE členi. USE člen se obnaša kot deklaracija na najbolj zunanji ravnini: ničesar ne skriva, toda je lahko sama maskirana z deklaracijami obdajajočih območij. USE člen se lahko pojavi v vsakem vgnezdenem podprogramu ali v osnovnem bloku; moč teh deklaracij pa bo izginila, ko bomo izstopili iz območja deklaracij. Zapisi s piko pa obdržijo svojo moč neglede na veljavnost območja USE deklaracije. To je pomembno pri lokalni definiciji imena, ki je v USE paketu. Npr.:

```
USE kup;
USE bit_lib, io, utility;
```

Paket z imenom STANDARD vsebuje definicije za preddefinirane tipe in podprograme. Uporabniku je ta paket brez nadaljnega dosegljiv in rutine se nalagajo avtomatično v program. Znanje o tem paketu je potrebno uporabniku le tedaj, ko so preddefinirani predmeti maskirani z drugimi deklaracijami v programu. V takih primerih bo uporabljen zapis s piko za dostopanje v preddefinirano okolico, ki je zbrana v zbirki STANDARD.SYM; te zbirke uporabnik ne sme modificirati. Paketi z WITH členi so implicitno definirani v paketu STANDARD. To omogoča uporabniku dostop v paket tudi pri njegovem skritem imenu, dokler je paket STANDARD viden.

Slovstvo k drugemu delu

((10)) S.Patchen: A Preview of Supersoft's Ada Compiler. Lifelines 3 (1982), No.4, 29 - 30.

((11)) J.G.P.Barnes: Programming in Ada. Addison-Wesley, 1982.

((12)) Janus Package User Manuals, 8080 Version 3. July 1982. RR Software, P.O.Box 1512, Madison, Wisconsin 53701.

Dodatek B

Ta dodatek prikazuje sintakso jezika Janus Ada; ta sintaksa se seveda razlikuje od sintakse uradnega jezika Ada. Rezervirane besede so izpisane z velikimi črkami zaradi boljše preglednosti. Ta sintaksa velja za verzijo 1.4.3, popravki za verzijo 1.4.5 (verzija 1.4.4 se ni pojavila) bodo dani kasneje. Številke na levem robu se nanašajo na podpoglavja priročnika v članku (poglavje 4).

SINTAKSA JEZIKA JANUS ADA (VERZIJA 1.4.3)

```
4.1
identifier ::= letter {[underscore]
                  letter_or_digit}
letter_or_digit ::= letter | digit
letter ::= upper_case_letter |
           lower_case_letter
numeric_literal ::= decimal_number |
                    based_number
decimal_number ::= integer [. integer]
                   [E exponent]
integer ::= digit {[underscore] digit}
exponent ::= [+] integer | - integer
based_integer ::= base # extended_digit
                  {[underscore] extended_digit} #
base ::= integer
extended_digit ::= digit | letter
character_literals ::= ' character '
character_string ::= " {character} "
pragma ::= PRAGMA identifier
           [(argument {, argument})];
argument ::= identifier | character_string |
             number
4.2
declaration ::= object_declaration |
                 type_declaration |
                 subtype_declaration |
                 number_declaration |
                 subprogram_declaration |
                 package_declaration
object_declaration ::=
      identifier_list : [CONSTANT]
        subtype_indication [ := expression] ;
    | identifier_list : [CONSTANT]
        array_definition ;
number_declaration ::= identifier_list :
        CONSTANT := literal_expression ;
identifier_list ::= identifier {, identifier}
type_declaration ::= TYPE identifier IS
                       type_definition ;
type_definition ::=
                 enumeration_type_definition |
                 integer_type_definition |
                 real_type_definition |
                 array_type_definition |
                 record_type_definition |
                 derived_type_definition |
                 private_type_definition
subtype_declaration ::= SUBTYPE identifier IS
                          subtype_indication ;
subtype_indication ::= type_mark [constraint]
type_mark ::= type-name | subtype_name
constraint ::= range_constraint
derived_type_definition ::=
                      NEW subtype_indication ;
range_constraint ::= RANGE range
range ::=
       simple_expression .. simple_expression
enumeration_type_definition ::=
   (enumeration_literal {,enumeration_literal})
enumeration_literal ::= identifier |
                        character_literal
integer_type_definition ::= range_constraint
real_type_definition ::= accuracy_constraint
accuracy_constraint ::=
     DELTA simple_expression [range_constraint]
array_type_definition ::=
                      ARRAY index_constraint OF
                        subtype_indication
index_constraint ::=
          (discrete_range {, discrete_range})
discrete_range ::=
         type_mark [range_constraint] | range
record_type_definition ::= RECORD
                             record_body
                           END RECORD
record_body ::= component_list  variant |
                component_list |
                variant | NULL ;
component_list ::= {component_declaration}
component_declaration ::=
      identifier_list : subtype_indication ; |
      identifier_list : array_type_definition ;
variant ::=
    CASE discriminant_name : subtype_indication
     IS
     {WHEN choice {| choice} => component-list}
    END CASE;
choice ::= simple_expression |
           discrete_range | OTHERS
declarative_part ::= {declarative_item}
                     {representation-specification}
                     {program_component}
program_component ::= package_declaration |
                      package_body
declarative_item ::= declaration | use_clause
4.3
name ::= identifier | indexed_component |
          selected_component | attribute |
          function_call
indexed_component ::=
              name (expression {, expression})
selected_component ::= name . identifier |
                       name . ALL
attribute ::=
            name ' identifier [( expression )]
literal ::= numeric_literal |
            enumeration_literal |
            character_string | NULL
expression ::= relation {AND relation} |
               relation {OR relation} |
               relation {XOR relation} |
               relation {AND THEN relation} |
               relation {OR ELSE relation}
relation ::= simple_expression
      [relational_operator simple_expression] |
        simple_expression [NOT] IN range |
        simple_expression [NOT] IN
          subtype_indication
simple_expression ::= [unary_operator]
                      term {adding_operator term}
term ::= factor {multiplying_operator factor}
factor ::= primary [** primary]
primary ::= literal | name | function_call |
       type_conversion | qualified_expression |
         ( expression )
logical_operator ::= AND | OR | XOR
relational_operator ::=  = | /= | < | <= | > |
                         >=
adding_operator ::=  + | - | &
unary_operator ::=  + | - | NOT
multiplying_operator ::=  * | / | MOD | REM
exponentiating_operator ::=  **
type_conversion ::= type_mark ( expression )
qualified_expression ::=
                     type_mark ' ( expression )
4.4
sequence_of_statements ::=
                          statement {statement}
statement ::= {label} simple_statement |
              {label} compound_statement
simple_statement ::= assignment_statement |
                     procedure_call |
                     exit_statement |
                     return_statement |
                     goto_statement |
                     null_statement |
                     asm_statement |
                     inout_statement
compound_statement ::= if_statement |
                       case_statement |
                       loop_statement |
                       block
label ::= <<identifier>>
null_statement ::= NULL ;
assignment_statement ::=
                  variable_name := expression ;
```

```
if_statement ::=
    IF condition THEN sequence_of_statements
 {ELSEIF condition THEN sequence_of_statements}
    [ELSE sequence_of_statements]
     END IF ;
condition ::= Boolean_expression
case_statement ::=
     CASE expression IS
       {WHEN choice {| choice} =>
                sequence_of_statements}
     END CASE ;
loop_statement ::= [loop_identifier :]
        [iteration_clause]
         basic_loop [loop_identifier] ;
basic_loop ::= LOOP
                 sequence_of_statements
               END LOOP
iteration_clause ::= FOR loop_parameter IN
                     [REVERSE] discrete_range |
                     WHILE condition
loop_parameter ::= identifier
block ::= [block_identifier :]
            [DECLARE declarative_part]
            BEGIN
              sequence_of_statements
            [EXCEPTION {exception_handler}]
            END [block_identifier] ;
exit_statement ::= EXIT [loop_name]
                     [WHEN condition] ;
return_statement ::= RETURN [expression] ;
goto_statement ::= GOTO label_name ;
asm_statement ::= ASM static_expression
                  {, static_expressjion} ;
inout_statement ::=
     OUT expression {, expression} ; |
     IN variable_name {, variable_name} ;
4.5
subprogram_declaration ::=
                     subprogram_specification ;
subprogram_specification ::=
           PROCEDURE identifier [formal_part] |
           FUNCTION identifier [formal_part]
             RETURN subtype_indication
formal_part ::= ( parameter_declaration
                  {; parameter_declaration} )
parameter_declaration ::= identifier_list :
       mode  subtype_indication [:= expression]
mode ::= [IN] | OUT | IN OUT
subprogram_body ::= subprogram_specification
     IS declarative_part
          BEGIN
               sequence_of_statements
               [EXCEPTION {exception_handler}]
          END [identifier] ;
procedure_call ::=
           procedure_name [actual_parameters] ;
function_call ::=
            function_name  actual_parameters |
            function_name ()
actual_parameters ::=
                 ( expression {, expressjion} )
4.6
package_declaration ::= package_specification ;

package_specification ::=
     PACKAGE identifier IS
           {declarative_item}
           [PRIVATE {declarative_item}
              {representation_specification}]
     END [identifier]
package_body ::=
        PACKAGE BODY identifier IS
          declarative_part
          [BEGIN
             sequence_of_statements
             [EXCEPTION {exception_handler}]
          ]
           END [identifier] ;
private_type_definition ::= PRIVATE
4.7
use_clause ::=
           USE package_name {, package_name} ;
```

```
4.8
compilation ::=
   context_specification  package_declaration |
     context_specification  package_body
context_specification ::= {with_clause}
with_clause ::= WITH unit_name {, unit_name} ;
4.9
exception_declaration ::=
                  identifier_list : EXCEPTION ;
exception_handler ::=
               WHEN exception_choice
                   {| exception_choice} =>
                     sequence_of_statements
exception_choice ::= exception_name | OTHERS
4.10
representation_specification ::=
                          address_specification
address_specification ::= FOR name USE AT
                    static_simple_expression ;
```

SINTAKSNE SPREMEMBE
jezika Janus Ada (verzija 1.4.5)

Nova verzija jezika Janus Ada se približuje standardu Ade in zato ukinja nekatere sintaksne kategorije verzije 1.4.3, uvaja pa tudi nove. Določene sintaksne kategorije se zaradi tega spremenijo.

Nove sintaksne kategorije so:

```
access_type_definition,
incomplete_type_declaration,
allocator
```

Ukinjajo se:

```
inout_statement,
representation_specification,
address_specification
```

Modificirane so tele kategorije:

```
argument,
declaration,
object_declaration,
number_declaration,
type_definition,
constraint,
accuracy_constraint,
declarative_part,
primary,
simple_statement,
parameter_declaration,
package_specification,
private_type_definition
```

Nove sintaksne kategorije imajo tele definicije:

```
access_type_definition ::=
                        ACCESS subtype_indication
incomplete_type_declaration ::=
                              TYPE identifier ;
allocator ::= NEW type_mark ;
```

Nekatere modifikacije definicij so:

```
declaration ::=
            ... | incomplete_type_declaration
type_definition ::=
                  ... | access_type_definition
constraint ::= ... | (static_expression)
primary ::= ... | allocator
simple_statement ::=
                     '(' brez inout_statement ')'
private_type_definition ::=
                           ... | LIMITED PRIVATE
```

... pomeni dodatek v obstoječem pravilu.

VIŠEPROCESORSKI SISTEMI S MIKROPROCESORIMA IM6100

M. JELAVIĆ

INSTITUT „RUDJER BOŠKOVIĆ", ZAGREB

UDK: 681.3.06.IM6100

U ovom radu razmatraju se dvije jednostavnije sistemske organizacije višeprocesorskih sistema s mikroprocesorima IM6100. Dan je opis i blok shema sistemskih organizacija sa višesabirničkim direktnim i posrednim pristupom sklopovskim globalnim sredstvima. Razradjene su osnovne programske strukture za isključivo dodjeljivanje globalnih sklopovskih sredstava i manipulacije sa globalnim varijablama - semaforima. Na temelju zahtijeva da vrijeme arbitraže za globalno sredstvo mora biti puno kraće od zauzeća i upotrebe globalnog sredstva izveden je zaključak o nivou paralelizma pogodnim za implementaciju na takvim organizacijama.

MULTIPROCESSOR SYSTEMS WITH MICROPROCESSORS IM6100

In this paper two simple multiprocessor organizations with microprocessors IM6100 are considered. Discriptions of multibus with direct and indirect access to global resources are given. Program structures for mutual exclusion at hardware level and for manipulation with global variables semaphores are developed. Following the requirement that the time for global resource arbitration must be much shorter than the time the global resource is used a concluison about convenient level of parallelism for implementation on particular organization is made.

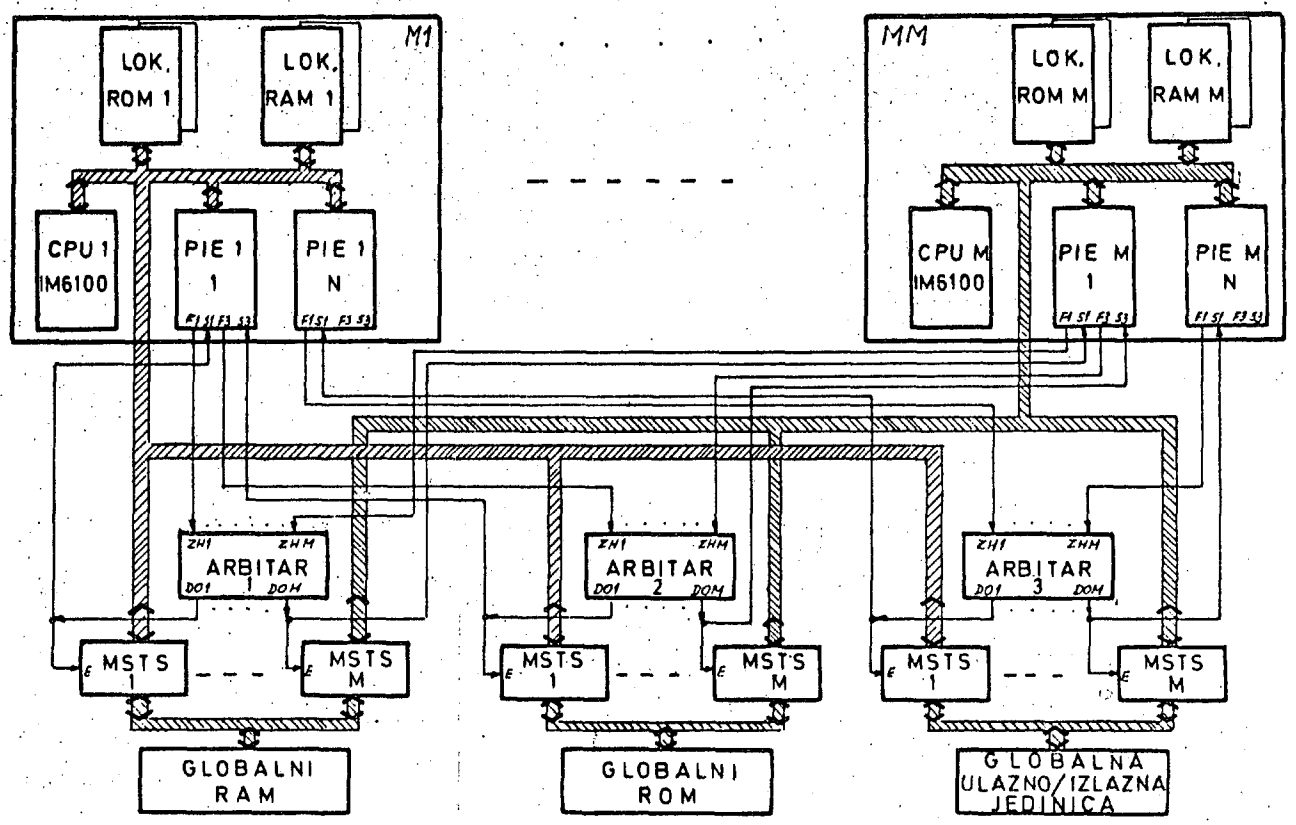
1. UVOD

Općeniti cilj razvoja digitalnih računala uz povećanje pouzdanosti rada je i smanjenje odzivnog vremena. Jedan od puteva da se to postigne je smanjenje vremena osnovnog ciklusa instrukcije. U tehnološkom smislu to se može postići većom gustoćom pakiranja elemenata na čipu. No sve veća gustoća pakiranja praćena je sve većim problemima oko realizacije te posebnog problema oko efikasnog odvoda topline.

Iako je poluvodička tehnologija napravila veliki kvantitativni i kvalitativni skok što se tiče pakiranja elemenata, jasno je da se takav razvoj neće moći nastaviti istim tempom /1/. Kako su zahtijevi za većom brzinom i pouzdanošću računala sve veći paralelno sa tehnološkom evolucijom razvija se takva organizacija i programska podrška računarskih sistema koja bi mogla podržavati i iskorištavati paralelne aktivnosti unutar sistema. Kao rezultat toga pojavljuju se računarski sistemi sa više procesnih jedinica - višeprocesorski sistemi.

Unatoč podosta neriješenih problema, organizacija funkcijskih jedinica kao što su mikroprocesori, memorije, ulazno-izlazne jedinice itd. u višeprocesorske sisteme kada se želi veća pouzdanost i bolja propusna moć sistema danas je postala opće prihvaćena tehnika /2/.

Razvoj komercijalnih višeprocesorskih sistema ovisan je i općenito diktiran dostupnošću pojedinih komponenata na tržištu. Iako danas najnovije generacije mikroprocesora sadrže sve potrebne uvjete za uspješno i efikasno projektiranje višeprocesorskih sistema opravdano je razmotriti mogućnosti organizacije višeprocesorskih sistema sa mikroprocesorima koji nemaju ugradjena ta svojstva, a sa kojima se ima iskustva u gradnji standardnih mikroprocesorskih sistema. Naročito je to opravdano u uvjetima jugoslavenskog tržišta gdje je dostupnost i mogućnost nabave takvih, a naročito novijih komponenti mala. Takav pokušaj je napravljen u ovom radu, a odnosi se na razmatranje dvije jednostavnije sistemske organizacije višeprocesorskih



Slika 1. Organizacija sa direktnim pristupom

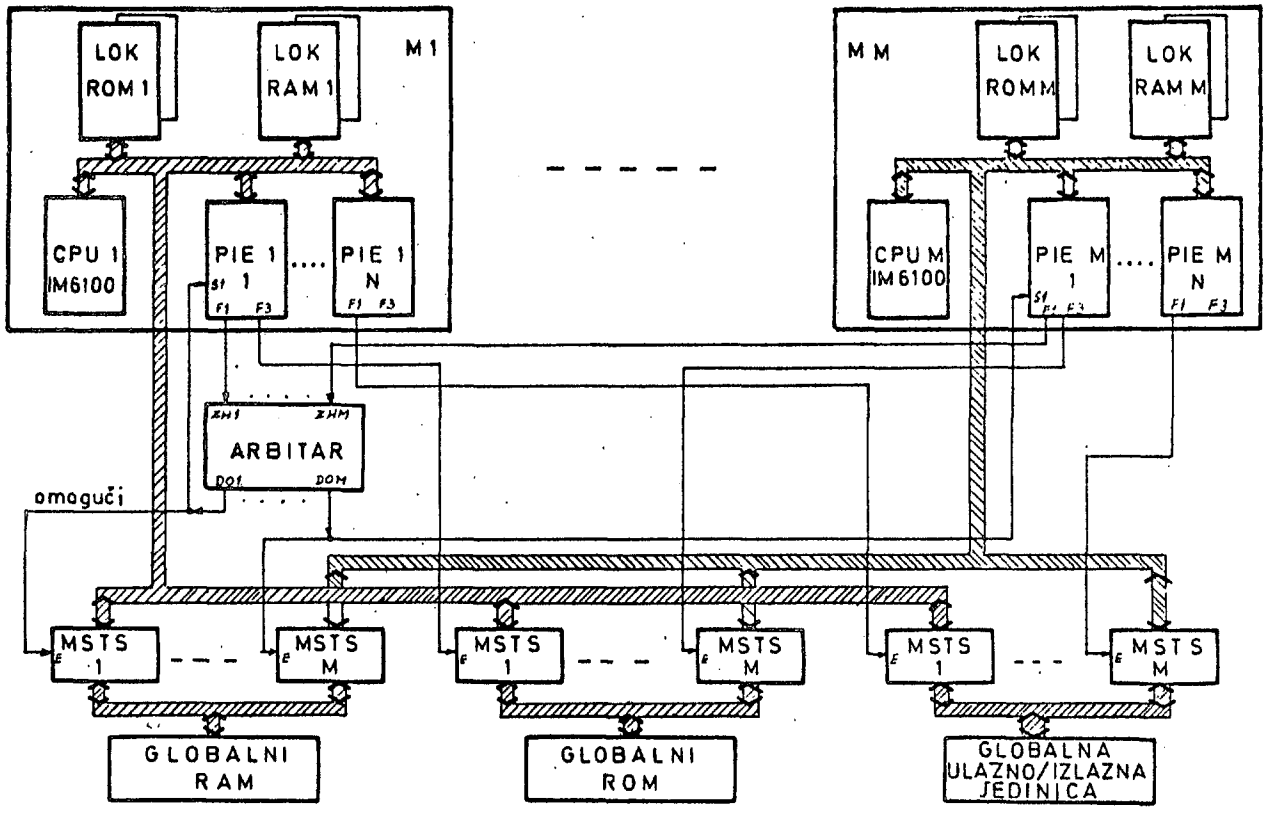
sistema s mikroprocesorima IM6100.

2. SISTEMSKA ORGANIZACIJA SA VIŠESABIRNIČKIM DIREKTNIM PRISTUPOM

Takav oblik organizacije prikazan je slikom 1. Standardnim konfiguracijama mikroprocesorskih sistema M1.....MM dodani su medjusklopovi koji omogućuju zajedničku upotrebu globalnih sredstava[x]. Medjusklopovi se sastoje od sklopova za riješavanje konflikta oko dodjele sklopovskih globalnih sredstava- arbitara i dvosmjernih sklopova sa 3 stanja koji povezuju sabirnice globalnih sredstava sa lokalnim sabirnicama pojedinih konfiguracija M1....MM. Sklop za arbitražu prihvaća zahtijeve za globalnim sredstvima ZH i po unaprijed utvrdjenom kriteriju odredjuje kojem od zahtijeva će globalno sredstvo biti dodijeljeno /3/,/4/. Svaka konfiguracija M generira signale za kontrolu i upravljanje medjusklopova preko veznog elementa IM6101 PIE (Peripheral Interface Element). Na temelju kontrolnih i upravljačkih signala mikroprocesora IM6100 vezani element generira 4 zastavice F1, F2, F3, i F4 od kojih se F1 i F3 generiraju direktno (posebne instrukcije), a F2 i F4 posredno (mijenjajući A registar veznog elementa) /5/.

Zahtijevi za globalnim sredstvima generiraju se preko zastavica F1 i F3. Zahtijev se proslijedjuje arbitru koji generiranjem jednog od DO signala jednoznačno odredjuje kojem od zahtijeva će biti udovoljeno. Isti taj signal preko S linije (SENSE) veznog elementa signalizira da je pristup sklopovskom globalnom sredstvu slobodan, a ujedno i povezuje sabirnice dodijeljenog globalnog sredstva sa sabirnicama konfiguracije M koja je generirala zahtjev. Jednoznačnost dodjeljivanja globalnog sredstva osigurana je samom građom arbitra /3,/4/. Svakom globalnom sredstvu pridjeljen je sklop za arbitražu, pa takvu konfiguraciju nazivamo višesabirničkom sa direktnim pristupom. Prestankom zahtjeva tj. skidanjem zastavice F onemogućuje se pristup globalnom sredstvu koje

[x] Pod globalnim sredstvima podrazumjevamo sklopovske i programske strukture koje mogu biti korištene od više konfiguracija.



Slika 2. Organizacija sa posrednim pristupom

sada može biti dodijeljeno slijedećoj konfiguraciji M.Zahtijev za globalnim sredstvom realiziran je slijedećim programskim odsječkom:

		broj stanja instrukcije
SFLGx	/postavi zastavicu Fx	(17)
SKIPx	/globalno sredstvo je /dodijeljeno ? DO=1?	(17)
JMP .-1	/ ne!	
-	/ da!	

a oslobadjanje globalnog sredstva instrukcijom:

CFLGx	/oslobodi globalno /sredstvo i resetiraj Fx	(17)

Zahtijev za globalnim sredstvom odvija se u vječitoj petlji čekanja. Zamjenom instrukcije JMP .-1 sa JMP WAIT može se realizirati eventualna lista čekanja za konkretno globalno sredstvo ali to ovisi o vrsti programske podrške implementirane na datom sistemu.

3 SISTEMSKA ORGANIZACIJA SA VIŠESABIRNIČKIM POSREDNIM PRISTUPOM

Karakteristika te organizacije (Sl.2) je da posjeduje samo jedan sklop za arbitražu koji je pridjeljen globalnom RAM-u. Logika za riješavanje konflikta oko dodijele globalnog RAM-a riješava se u jednom nivou isto kao i za sva globalna sredstva u organizaciji sa višesabirničkim direktnim pristupom i to postavljanjem zastavice i ispitivanjem DO signala arbitra. Medjutim u ovoj organizaciji za sva ostala globalna sredstva postupak riješavanja konflikta odvija se u dva nivoa. Prvi je kao što je već spomenuto sklopovska arbitraža za globalni RAM preko zastavice Fl, a drugi je ispitivanje rezervirane memorijske lokacije RAM-a koju nazivamo semaforom S. Za svako globalno sredstvo postoji po jedna rezervirana lokacija u RAM-u koja indicira zauzeće globalnog sredstva ako je S=0 ili da je sredstvo slobodno ako je S=1. Nakon ispitivanja semafora S ako je globalno sredstvo slobodno, postavljamo zastavicu F koja povezuje lokalne sabirnice konfiguracije M sa sabirnicama traženog sredstva. I ovdje je osigurana jednoznačnost pristupa globalnom sredstvu. Za globalni RAM to je osigurano sklopovskom arbitražom, a za ostala globalna sredstva indikacijom zabrane preko semafora. Kako je sklopovskom arbitražom za globalni RAM omogućen i jednoznačan pristup semaforima to je uvjet jednoznačnosti dodjele globalnog sredstva samo jednoj konfiguraciji M zadovoljen. Nakon oslobadjanja globalnog RAM-a pristup semafo-

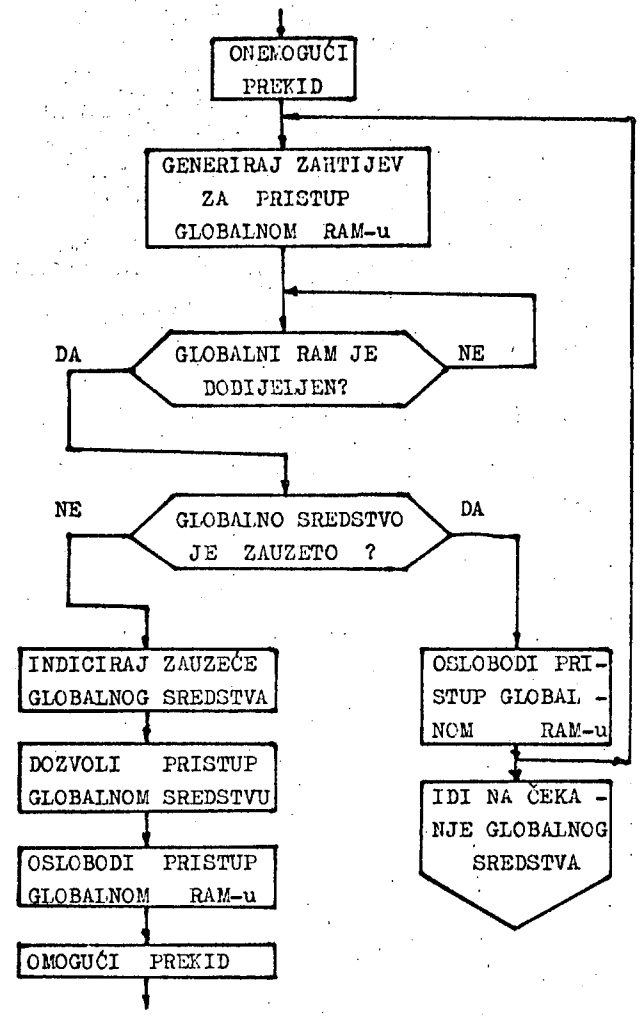
rima je omogućen ostalim konfiguracijama M koje će naći da je globalno sredstvo zauzeto, što u stvari znači zabranu postavljanja zastavice F za povezivanje sabirnica. Programski odsječci za zauzeće i oslobadjanje globalnog sredstva na gore opisani način su slijedeći:

```
                                        broj stanja
                                        instrukcije
IOF         / onemogući prekid          (17)
SFLG1       / F1=1! generiranje zahtijeva
            / za pristup globalnom RAM-u (17)
SKIP1       / DO1=1? globalni RAM je
            / dodijeljen?               (17)
JMP .-1     / ne!
CLA CLL     / da! očisti akumulator     (10)
TAD S       / S u akumulator            (10)
SZA CLA     / S=0? globalno sredstvo je
            / zauzeto ?                 (10)
JMP .+3     / ne!                       (10)
CFLG1       / da! oslobodi pristup
            / globalnom RAM-u
JMP .-10
DCA S       / S=0! indikacija zauzeća
            / globalnog sredstva        (11)
CFLGx       / dozvoli pristup globalnom
            / sredstvu                  (17)
CFLG1       / oslobodi pristup globalnom
            / RAM-u                     (17)
ION         / omogući prekid            (17)
```

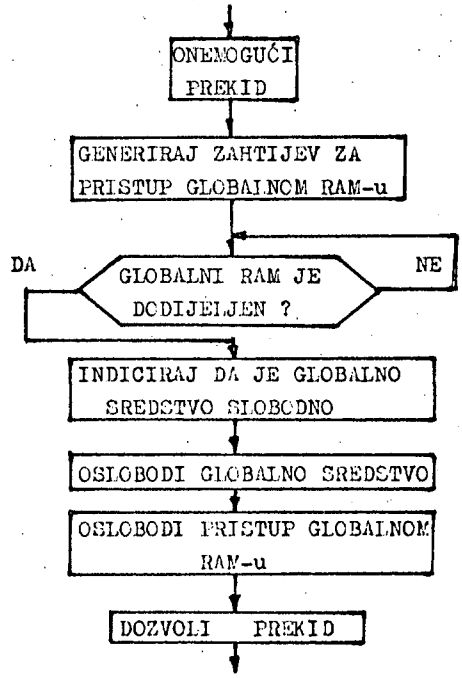
slijed instrukcija za zauzeće globalnog sredstva

```
IOF         / onemogući prekid          (17)
SFLG1       / generiranje zahtijeva za
            / pristup globalnom RAM-u   (17)
SKIP1       / globalni RAM je dodijeljen?(17)
JMP .-1     / ne!
CLA IAC     / da! akumulator = 1        (10)
DCA S       / S=1! indiciraj da je
            / globalno sredstvo slobodno (11)
SFLGx       / oslobodi globalno sredstvo (17)
CFLG1       / oslobodi pristup globalnom
            / RAM-u                     (17)
ION         / omogući prekid            (17)
```

slijed instrukcija za oslobađanje globalnog sredstva



Dijagram toka programskog odsječka za zauzeće globalnog sredstva



Dijagram toka programskog odsječka za oslobađanje globalnog sredstva

Na isti način kao i u prethodnoj organizaciji zauzeće globalnog sredstva ne mora se odvijati u vječnoj petlji čekanja. Umjesto naredbe JMP-10 može se upotrebiti JMP WAIT i na taj način postaviti odredjenu konfiguraciju M u listu čekanja na dotično globalno sredstvo. Jasno je da je uvjet da se to može napraviti da jezgro operacionog sistema ima mogućnost manipuliranja sa listama čekanja za pojedina globalna sredstva.

4. ANALIZA VIŠEPROCESORSKIH SISTEMSKIH ORGANIZACIJA

Sistemske organizacije na Sl. 1 i Sl. 2 zahtijevaju eksplicitno traženje pristupa globalnim sredstvima (preko zastavica i semafora). Razlog leži u tome što instrukcijski repertuar mikroprocesora IM6100 sa kojima su realizirane spomenute organizacije višeprocesorskih sistema ne sadrži instrukcije koje omogućuju ispitivanje i promjenu memorijskih lokacija RAM-a u jednom ciklusu instrukcije u toku kojega bi zadržavali isključivi pristup toj memorijskoj lokaciji. Zbog toga se isključivi pristup globalnim sredstvima postiže eksplicitnim metodama koje su općenito gledano neefikasne ali jedine moguće u ovom slučaju. Eksplicitne metode zahtijevaju više vremena, jer podrazumijevaju izvršavanje instrukcija rezerviranih za postavljanje zahtjeva za sklopovskim globalnim sredstvima i manipuliranje sa semaforima. Da bi sistem bio efikasan, vrijeme potrebno za arbitražu za nekim globalnim sredstvom mora biti puno manje (za red veličine) od vremena korištenja globalnog sredstva. Ako sa t_a označimo vrijeme arbitraže za globalno sredstvo (izvršavanje instrukcija rezerviranih za zauzeće globalnog sredstva), sa t_o vrijeme oslobađanja globalnog sredstva (izvršenje instrukcija rezerviranih za oslobađanje globalnog sredstva), sa t_A vrijeme u kojem arbitar odluči kojem od zahtijeva će pridjeliti globalno sredstvo i povezati sabirnice, sa t_O vrijeme koje je potrebno arbitru za resetiranje DO signala nakon prestanka zahtijeva ZH i sa t_{ZG} vrijeme zauzeća globalnog sredstva možemo napisati relaciju: $t_{ZG} \geq 10 \left[(t_a + t_o) + t_A + t_O \right]$ Kako je[x]:

$t_A \ll t_a$

$t_O \ll t_o$

možemo pisati:

$t_{ZG} \geq 10\ (t_a + t_o)$

Ako t_a i t_o izrazimo sa brojem stanja instrukcija za slijed instrukcija za zauzeće i oslobađanje globalnog sredstva za obadvije organizacije dobijemo slijedeće rezultate[xx].

t'_a = 34 stanja za org. sa direktnim pristup.

t'_o = 17 stanja za org. sa direktnim pristup.

t'_a = 153 stanja za org. sa posrednim pris.

t'_o = 123 stanja za org. sa posrednim pris.

Konačno dobijamo rezultat:

$t'_{ZGd} \geq 510$ stanja za organizaciju sa direktnim pristupom i

$t'_{ZGp} \geq 2760$ stanja za organizaciju sa posrednim pristupom

Trajanje stanja instrukcije ovisi o taktu oscilatora mikroprocesora, pa je:

$t_{ZGd} \geq 510 T_o$

$t_{ZGp} \geq 2760 T_o$ gdje je T_o trajanje jednog stanja mikroprocesora IM6100 (tab. 1)

	IM6100 Vcc=5.0V fc= 4MHz	IM6100A Vcc=10.0V fc= 8MHz	IM6100C Vcc=5.0 fc= 3.3MHz
To	500 ns	250 ns	600 ns
t_{ZGd}	$\geq$255 μs	$\geq$127,5 μs	$\geq$ 306 μs
t_{ZGp}	$\geq$1,38 ms	$\geq$ 0.69 ms	$\geq$1.656 ms

tab. 1

t_{ZG} određuje zapravo najmanje vrijeme korištenja globalnog sredstva da bi bio zadovoljen prije navedeni zahtijev o efikasnosti sistema, a implicitno i nivo paralelizma pogodan za implementaciju na spomenutim sistemskim organizacijama. t'_{ZG} podijeljen sa prosječnim brojem stanja po instrukciji koji iznosi 12,2 /6/ daje približan broj instrukcija koje bi se trebale izvršiti između dvije uzastopne dodjele globalnog sredstva. Broj instrukcija za organizaciju sa direktnim pristupom iznosi 42, a za organizaciju sa posrednim pristupom 227. To sugerira implementaciju paralelizma /7/ na nivou logičkih cijelina zadatka za organizaciju sa direktnim pristupom i na nivou zadataka za organizaciju sa posrednim pristupom.

---

x za opravdanost tih uvjeta vidjeti /3/ u kojem se pokazuje da je vrijeme t_A u realiziranim arbitrima veličine par stotina nanosekundi, ovisno od broja zahtijeva ZH koje arbitar može poslužiti, a $t_O \ll t_A$ dok je prosječno izvršavanje instrukcije IM6100 sa prosječnim brojem stanja 12,2 kod 4MHz više od 6 μs.

xx uzet je slijed instrukcija kod slučaja da je globalno sredstvo slobodno. Nalaženje da je sredstvo zauzeto i problem zagušenosti cjelokupnog sistema spada u kategoriju problema interferencije u višeprocesorskim sistemima.

5. ZAKLJUČAK

Dvije opisane konfiguracije sistema pokazuju relativno jednostavan način realizacije višeprocesorskih sistema sa mikroprocesorima koji nisu predvidjeni za takve namjene i sa međusklopovima koji su relativno lako dostupni. 4 vrste arbitara detaljno su obrađeni u literaturi /3/, a realizirani su sklopovima srednjeg i malog stupnja integracije. Broj konfiguracija standardnih mikroprocesorskih sistema M nije ograničen iako će se odraziti na efikasnost sistema posebno u slučaju organizacije sa posrednim pristupom. Uočava se da implementiranje većeg nivoa paralelizma povlači za sobom kompleksnija sklopovska riješenja. Daljnji korak u rješavanju implementacije većeg paralelizma bio bi implementacija rezerviranih instrukcija koje bi mogle osiguravati čitanje sa isključivim pristupom memoriji i pisanje. To bi se moglo postići normalnim instrukcijama ograničenim na odredjeni adresni prostor uz asistenciju kompleksnijeg međusklopovskog riješenja. To bi nadalje omogućilo implementaciju Dijkstrinih algoritama /8/ za ulazak u kritični odsječak.

Treba napomenuti da ovako koncipirane organizacije dopuštaju paralelan rad onoliko konfiguracija mikroprocesorskih sistema koliko ih je uključeno u organizaciju i da nisu ograničene samo na realizaciju sa mikroprocesorom IM6100.

Osiguravanjem isključivog pristupa globalnim sredstvima kao i uvođenjem globalnih varijabli - semafora stvorene su pretpostavke za efikasnu implementaciju jezgre višeprocesorskog sistema, koja bi osiguravala komunikaciju i koordinaciju između raznih zadataka nekog kompleksnog programa te efikasno korištenje skupih ulazno-izlaznih jedinica.

LITERATURA:

/1/ CAY WEITZMAN, "Distributed Micro/Minicomputer Systems", Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1980.

/2/ PHILIP H. ENSLOW,JR., "Multiprocessors and Parallel Processing", Comtre Corporation John Wiley & Sons, 1974.

/3/ MARINO JELAVIĆ, "Rasporedjivanje procesa u računalima s više procesora" Magistarski rad, Elektrotehnički fakultet, Zagreb 1982.

/4/ M. JELAVIĆ, "The structure and analysis of arbiters based on priority, ring, distributed and FIFO selection", 18th Internation Symposium on Mini and Microcomputess and their Application, Davos, March-2-5. 1982. /u štampi/.

/5/ "Intersil IM6100 CMOS 12 bit Microprocesor", 10900 N. Tantau Ave., Cupertino, Calif. 95014.

/6/ N. BOGUNOVIĆ, M. JELAVIĆ, "An universal microprocessor system for data acquisition and registration", IMEKO Symposium on Computerized Measurement, Dubrovnik, Yugoslavia, 1981, p.287-290

/7/ ANTHONY RALSTON, CHESTER L. MEEK, "Encyclopedia of Computer Science", Petrocelli/Charter 1976.

/8/ EDWARD G. COFFMAN, JR., PETER J. DENNING "Operating System Theory", Prentice H all, Inc. Englewood Cliffs, New Jersey, 1973, p. 1-82.

VMDP – SOFTVERSKI MONITOR ZA ON LINE PRIKAZ AKTIVNOSTI RAČUNARA CDC 3170

ZLATKO DOLENEC

UDK: 681.326.0

TEHNIČKA VOJNA AKADEMIJA KoV JNA
ZAGREB

Prikazani su namjena, način rada i ugradnja monitora VMDP u operativni sistem MASTER/CDC 3oooL. Monitor VMDP (Video Monitor Dump Program) izgrađen je kao rezidentni task operativnog sistema koji u kontroliranim vremenskim intervalima nadgleda sistemske tabele i daje na ekranu terminala prikaz aktivnosti batch programa i CPU taskova, aktivnost suspenzije, iskoristivost memorije i CPU, te neke podatke za analizu performansi sistema. Sinhronizacija rada monitora VMDP i ostalih komponenti operativnog sistema postignuta je međusobnim isključenjem.

VMDP - A SOFTWARE MONITOR FOR ON LINE DISPLAY THE CDC 317o COMPUTER SYSTEM ACTIVITIES

The objective, modes of action and insertion the VMDP software monitor into MASTER/CDC 3oooL operating system are described. VMDP (Video Monitor Dump Program) monitor was built as resident operating system task that inspects system tables in controlled time intervals and displays batch jobs, CPU tasks and suspension activities, memory and CPU utilization, and some data for performance analyses. Synchronization between VMDP monitor and another parts of MASTER operating system was achieved with mutual exclusion.

1. Uvod

Postojeća konfiguracija računara CDC 317o instaliranog na TVA KoV JNA ima konzolni pisač kao sredstvo komunikacije između operatora i sistema. Sporost pisača i prisustvo papira na kojem je i polazište i odredište poruka između operatora i sistema, onemogućavaju ili čine krajnje neekonomičnim njegovu upotrebu za prikaz većih količina podataka koji se ažuriraju u kratkim vremenskim intervalima.

Sistem CDC 317o ima mogućnost da se komunikacija operatora i sistema vrši putem posebnog videoterminala i uz podršku odgovarajućeg sistemskog softvera, ali taj dio opreme nije nabavljen uz postojeću konfiguraciju. To je razlog da je sada broj upita i odgovora koji se mogu dobiti putem konzolnog pisača veoma skroman. Podaci o aktivnosti sistema dobivaju se jednokratno na poseban zahtjev (komandu) i daju samo informaciju o tome koji batch programi su aktivni, a koji čekaju na aktiviranje.

Motiva za izgradnju monitora VMDP bilo je više. Prvo, to je bila potreba da se programska aktivnost na nivou korisnika i operativnog sistema učine vidljivima iz edukativnih razloga za slušaoce specijalnosti računarske tehnike i širi krug korisnika profesionalno vezanih uz računar. Zatim, ugradnjom u sistem vlastitog interaktivnog procesora sistemskih kontrolnih komandi, udvostručen je broj batch programa koji mogu raditi istovremeno (sa tri na šest), a to je povećalo napor operatora da formira optimalno punjenje računara, s obzirom na to da dio programa može ući u sistem putem terminala, dakle bez njegove kontrole.

Jedan od zadataka monitora VMDP bio je taj da omogući operatoru uvid u aktivnost sistema i zauzeće njegovih kritičnih resursa. Prikaz dinamike korištenja pojedinih resursa sistema omogućuju direktno ili indirektno mjerenje i analizu nekih performansi sistema.

Pored toga, način realizacije monitora VMDP i njegovo mjesto unutar operativnog sistema MASTER omogućuju on line analizu situacije u slučaju neobičnih manifestacija u radu sistema ili u slučaju njegovog potpunog zastoja (deadlock). To je značajni dobitak za održavanje sistema, s obzirom na to da se takva analiza u operativnom sistemu MASTER može napraviti samo dugotrajnim i kompliciranim off line postupkom (dump rezidentnih tabela na magnetsku traku, ponovna inicijalizacija operativnog sistema i dump sadržaja trake na linijski štampač, pa tek tada analiza zatečenog stanja sistemskih tabela).

2. Zadaci i način rada monitora VMDP

Monitor VMDP može raditi u statičkom i dinamičkom režimu rada. Dinamički režim rada naziva se MONITOR mod, a statički DUMP mod. U dinamičkom režimu rada VMDP automatski i bez intervencije operatora nadgleda, obrađuje i prikazuje različite aktivnosti računarskog sistema.

U statičkom režimu rada se nadgledanje aktivnosti sistema vrši jednokratno, na poseban zahtjev operatora. Upravljanje radom monitora

VMDP vrši se interaktivno sa istog terminala na na kojem se prikazuju podaci o aktivnosti sistema.

MONITOR mod

Zadatak je monitora VMDP u režimu rada MONITOR mod, da u određenim intervalima vremena vrši nadgledanje sistemskih tabela operativnog sistema MASTER i daje prikaz slijedećih podataka na ekranu terminala CDC 211 :

- pokretni indikator zauzeća fizičke memorije raspoložive za korisnike
- imena aktivnih batch programa i iznos dotad utrošenih resursa svakog aktivnog programa
- imena batch programa koji čekaju da budu aktivirani i iznos resursa koje će svaki od njih tražiti od sistema
- imena CPU taskova koji čekaju na CPU
- imena CPU taskova suspendiranih na disk
- imena trenutno aktivnih CPU taskova nekog aktivnog batch programa
- ključni podaci iz vektora stanja (deskriptor hardverskog i softverskog konteksta taska) bilo kojeg taska poznatog operativnom sistemu (uključujući i taskove koji su vlasništvo operativnog sistema)
- kumulirano vrijeme rada CPU za korisnike
- kumulirano vrijeme CPU za rad operativnog sistema MASTER
- faktor napredovanja svakog aktivnog batch programa u vremenu
- faktor napredovanja operativnog sistema MASTER u vremenu
- iznos logičke memorije raspoložive za batch programe
- vrijednost kvantuma vremena (time slice), ime verzije operativnog sistema pod kojom računar radi, sistemsko vrijeme i datum.

Većina ovih podataka prikazuje se na ekranu istovremeno, a samo neki podaci se odabiru selektivno putem odgovarajuće komande.
Slijedeće komande (odvojene zarezom) mijenjaju način rada monitora i sadržaj dijelova ekrana:

TIM1, TIM2, PAUS, JOBT, TASK, MONI, XXXX

Komandama TIM1 i TIM2 mijenjaju se dužine vremenskih intervala u kojima VMDP vrši uvid u stanje sistema i prikaz tog stanja na ekranu.
Komandom PAUS se monitor VMDP stavlja u mirovanje kroz zadati broj minuta (max 99), a terminal CDC 211 stavlja na raspolaganje sistemu. Nakon isteka zadatog broja minuta, VMDP se javlja operatoru porukom na konzolnom pisaču, tražeći ponovnu inicijalizaciju (dodjela terminala CDC 211) ili novi interval mirovanja.

Komandom TASK se VMDP stavlja u takav način rada da se u lijevom dijelu prve linije ekrana prikazuju određeni podaci iz vektora stanja bilo kojeg taska u sistemu. Ti podaci su status taska (spremnost, neaktivnost, čekanje na neki resurs, čekanje na završetak IO operacije, itd), dinamički prioritet taska, ime taska pozivnika i još neki drugi podaci. U komandi TASK identifikaciju taska čine njegovo ime i ime batch programa u kojem je task invociran. Izostavljanje imena batch programa znači da se radi o tasku koji pripada operativnom sistemu MASTER.

Komandom JOBT se u prvoj liniji ekrana ispisuju imena aktivnih taskova batch programa koji je u komandi naveden svojim imenom. Aktivni taskovi u gornjem kontekstu su oni koji su se u času monitoriranja sistema nalazili u listi čekanja za CPU.

Komandom XXXX monitor VMDP iz stanja MONITOR mod prelazi u statički režim rada - DUMP mod. Komandom MONI pokreće se obrnuti postupak, dakle prelaz iz DUMP moda u MONITOR mod.

Nakon inicijalizacije ili nakon komande MONI, VMDP uvijek radi u MONITOR modu. Unutar MONITOR moda ciklički u vremenu ponavljaju se dva stanja. To su MONITOR mod 1 i MONITOR mod 2.

Svako stanje predstavlja određeni tip nadgledanja aktivnosti sistema. U MONITOR mod 1 vrši se nadgledanje, obrada i prikaz aktivnosti sistema na nivou batch programa (job scheduling), a u MONITOR mod 2 na nivou taskova (task scheduling).
Ta stanja se ponavljaju u vremenskim intervalima 3ooo milisekundi za MONITOR mod 1 i 5oo milisekundi za MONITOR mod 2. To su default vrijednosti koje se komandama TIM1 i TIM2 mogu mijenjati u toku rada monitora. Ekran terminala CDC 211 ima veličinu od 1o4o znakova (13 linija po 8o znakova). Slika 1. prikazuje primjer sadržaja ekrana u (default) MONITOR modu.
Linijom 4 ekran je podijeljen na dva dijela. Ispod te linije prikazani su podaci o aktivnosti sistema na nivou batch programa, i to tako da su na desnoj strani prikazani podaci o aktivnim batch programima (najviše 6), a na lijevoj podaci o programima koji čekaju da budu aktivirani (najviše 7).

Za aktivne programe ti podaci su navedeni ispod parola koje imaju slijedeće značenje:

JOB INIT - ime aktivnog batch programa

SCHT - broj minuta CPU scheduliranih za taj program

TIM% - postotak utrošenog vremena SCHT

LIN% - postotak utrošenih linija linijskog štampača (schedulirana vrijednost se ne prikazuje)

B - broj scheduliranih diskova

COR - broj memorijskih parcela koje u tom trenutku zauzimaju taskovi batch programa

FN - faktor napredovanja programa u vremenu između dva uzastopna stanja MONITOR mod 1

```
1   MEMORY %    o=====  33=====  66=====  9o=====          SUSP:
2   RDY TASK:   *IDL      X        FTNU     COSY     *MON
3       MONITOR:                    A C T I V E      o9/36/18       TIME2: 5oo MS
4   --- KRT TVA ---------     VIDEO MONITOR DUMP PROGRAM     -------- o5/o9/82 -----
5   JOB SCHD CL  COR TIM BYC DISKB          SCHT TIM% LIN%  JOB INIT  B  COR  FN
6   UCBLIVA   B  6o  25  1    1   AVLM  4    3o   73   13   FTNUAGC       42  27
7   XFERLIST  B  1o   5  o        PMEM 116    1o   16    1   ABSGSP    o   23  32
8   FTNU1561  I  45   5  o        MA24   2    2o    5    o   IJMTDoo3      12   5
9   FTNU1562  I  45   5  o        JTBL   6     5   37   52   COSYLIST      27  13
1o  MVRVUC5   C  73  25  o        EDIT  3Z
11                                 QUAN  3o
12                                   TIME1:    3ooo MS     SUM CORE US: 86 %
13    USER TIME:    2o73 SEK   MASTER TIME:    1386 SEK              MASTER FN: 22
```

Slika 1. Primjer sadržaja ekrana u (default) MONITOR modu. Četiri batch programa rade konkurentno, a njih pet čeka na aktiviranje. U listi čekanja na CPU nalazi se pet taskova poredanih po prioritetu, a na vrhu liste je task koji trenutno drži procesor.

Za programe koji čekaju na aktiviranje ti su podaci:

JOB SCHED - ime programa

CL - klasa (prioritet) programa

COR - schedulirani broj memorijskih parcela logičke memorije

TIM - schedulirani broj minuta CPU

BYC - broj koji kaže koliko puta je program bio viđen za aktiviranje

DISKB - broj scheduliranih diskova

Postotak trenutno upotrebljene korisničke (fizičke) memorije prikazan je uz parolu SUM CORE US. Nakon punjenja rezidentnog dijela operativnog sistema u memoriju, veličina memorije raspoloživa za korisničke taskove prikazana je uz parolu PMEM.
USER TIME je kumulativno vrijeme rada CPU koje ide na račun korisnika, a MASTER TIME je kumulativno vrijeme rada CPU za sistemsku egzekutivu i taskove operativnog sistema. Podatak koji je naročito ineresantan operatoru je AVLM. To je veličina trenutno raspoložive logičke memorije, a to je ona memorija za koju se natječu batch programi kad čekaju na aktiviranje. Početna vrijednost logičke memorije je u instalacionoj proceduri postavljena na vrijednost raspoložive fizičke memorije (PMEM) uvećanu za određeni postotak. Time je omogućeno da u konkurentni rad krene više batch programa nego što bi to bilo moguće računajući raspoloživu stvarnu fizičku memoriju, s obzirom na to da zahtjevi za stvarnom memorijom unutar batch programa variraju u toku vremena. Konflikte u zahtjevima za stvarnom memorijom operativni sistem razrješava suspenzijom taskova na magnetski disk.

Svi podaci u donjih devet linija ažuriraju se u vremenskom intervalu uz TIM1. Linija broj 3 daje informacije o stanju rada monitora VMDP (ACTIVE ili STOPPED), trenutno vrijeme u sistemu i interval vremena u kojem se ponavlja stanje MONITOR mod 2 (TIME2).

U stanju MONITOR mod 2 vrši se ažuriranje podataka u prve dvije linije. U default MONITOR modu (kada ne djeluju komande TASK i JOBT) se u prvoj liniji ekrana iza parole MEMORY u taktu intervala TIME2 pokreću markeri koji brojem ispred sebe daju grubu indikaciju postotka zauzeća fizičke korisničke memorije. Lista imena taskova (najviše 8) koji čekaju na CPU prikazana je u drugoj liniji iza parole RDY TASK. Taskovi su, od lijeva na desno poredani prema rastućem prioritetu koji tog trenutka imaju. U prvoj liniji, desno od parole SUSP nalaze se imena taskova koji su zbog konflikta u zahtjevu za stvarnom memorijom suspendirani na disk. Komandama TASK i JOBT se iz prve linije uklanjaju pokretni markeri, a na njihovo mjesto dolaze podaci u skladu sa komandom. Nakon razaranja taska ili terminiranja batch programa, monitor VMDP u prvoj liniji ekrana ponovno prikazuje pokretne markere zauzeća korisničke fizičke memorije.

Izmjena stanja MONITOR mod 1 i MONITOR mod 2 čini osnovu u dinamici rada monitora VMDP. Način izmjene tih stanja i globalnu shemu zbivanja prikazuje program vmdp na slici 2.

Centralno mjesto svakog stanja je pristup podacima u određenim sistemskim tabelama operativnog sistema, te obrada i transfer tih podataka u buffer u kojem sa nalazi maska sadržaja ekrana (slika 1.) terminala CDC 211. Sistemske tabele pristupne su samo rutinama egzekutive EXEC i taskovima operativnog sistema koji imaju posebne privilegije. Odnos monitora VMDP i ostalih komponenti operativnog sistema MASTER je konkurentan, a ne kooperativan. To znači da monitor VMDP (u implementaciji to je task operativnog sistema) ne izmjenjuje podatke sa drugim dijelovima operativnog sistema i jedini njegov izravni kontakt sa egzekutivom su sistemske tabele kao zajedničke varijable. Problem sinhronizacije u radu monitora i drugih dijelova operativnog sistema prilikom pristupa sistemskim tabelama, riješen je uvođenjem kri-

```
program  vmdp;
.
type statio = (finish,active,abnorm);
.
var status: statio;
    acc,tim1,tim2: real;
.
.
komunikacija sa operatorom i vezivanje na termunit;
unos maske ekrana sa diska;
inicijalizacija konstanti i kontrolnih varijabli;
.
tim1:=3000;
tim2:=500;
.
repeate
   begin (*MONITOR mod 1*)
     disable interrupt;
      monitoriranje batch program aktivnosti;
     enable interrupt;
     obrada i transfer podataka na termunit;
   end
   acc:=0;
   repeate
       begin (*MONITOR mod 2*)
         disable interrupt;
          monitoriranje aktivnosti taskova;
         enable interrupt;
         obrada i transfer podataka na termunit;
           read(termunit,..,status,ready);
           pause(tim2);
         case status of
           active: acc:=acc+tim2;
           finish: begin
                     read(termunit,..,status,wait);
                     case status of
                       finish:  begin
                                  pozivanje odgovarajuće rutine u
                                  skladu sa komandom;
                                  acc:=tim1;
                                end
                       abnorm:  ioerror;  end
                   end
           abnorm: ioerror;                end
       end
   until acc   tim1;
until forever
end.
```

Slika 2. Globalni tok zbivanja u monitoru VMDP

tične zone u kôd monitora. Kritična zona realizirana je isključenjem prekidnog sistema (disable interrupt) prije pristupa sistemskim tabelama, i ponovnim uključenjem (enable interrupt) nakon pristupa. Takvo rješenje omogućeno je činjenicom da postojeća konfiguracija računara CDC 3170 ima jedan centralni procesor, da se oduzimanje procesora taskovima vrši i bez njihovog "pristanka" (time slice prekid), i time što je vrijeme boravka monitora VMDP u svojim kritičnim zonama vrlo kratko (nekoliko milisekundi).

Komunikacija sa monitorom VMDP započinje tako da se prekine automatizam u izmjeni stanja MONITOR mod 1 i MONITOR mod 2. To je moguće učiniti u vrijeme kada se monitor nalazi u stanju MONITOR mod 2. Tada (slika 2.) program vmdp inicira operaciju čitanja sa terminala i nastavlja rad konkurentno sa odvijanjem te operacije (parametar ready u naredbi read). Taj nastavak rada je izvršavanje makroinstrukcije pause(tim2), kojom se monitor VMDP stavlja u stanje čekanja na CPU u trajanju od tim2 mili sekundi. U tom vremenskom intervalu operator ima mogućnost da pritiskom na tipku SEND terminala CDC 211 završi iniciranu operaciju čitanja (status finish).
Ako se je to desilo, program vmdp unosi u treću liniju ekrana poruku STOPPED i inicira novu operaciju čitanja sa terminala, ali sada sa parametrom wait, tako da monitor čeka neograničeno dugo da se ta operacija čitanja završi.
Operator tada unosi u prvu liniju ekrana neku od prije navedenih komandi i pritiskom na SEND daje znak da je operacija čitanja realizirana.
U skladu sa vrstom komande, monitor mijenja uvjete ili način rada, a procesiranje se nas-

tavlja u stanju MONITOR mod 1. Ako se prva operacija čitanja terminala nije završila (operator u intervalu od tim2 milisekundi nije pritisnuo tipku SEND) vrijednost tim2 se kumulira u varijabli acc, a stanje MONITOR mod 2 se ponavlja. Kad vrijednost varijable acc postane jednaka ili veća od tim1, monitor VMDP prelazi u stanje MONITOR mod 1 i ciklus se ponavlja. Vrijeme tim2 dovoljno je veliko da se monitor nakon jednog ili dva pritiska na tipku SEND "uhvati" u pauzi i tako prekine njegov rad.

DUMP mod

Namjena je DUMP moda u radu monitora VMDP da se na ekranu prikažu i neki drugi interesantni podaci o radu sistema, a koji iz nekog razloga nisu prikazani u MONITOR modu. Prilikom rada u DUMP modu, monitor VMDP poziva u rad task sa imenom DUMP koji se nalazi na sistemskoj biblioteci na magnetskom disku. Zadatak je taska DUMP da u skladu sa svojom namjenom formira podatke u bufferu veličine ekrana terminala CDC 211, te da te podatke i kontrolu vrati monitoru VMDP. Nakon povratka kontrole, VMDP vrši transfer tih podataka na ekran terminala i čeka na novu komandu operatora, koji ili nastavlja rad u DUMP modu (ponovno komanda XXXX) ili komandom MONI prelazi u MONITOR mod.

Zbog toga što se task DUMP gradi i instalira na sistemsku biblioteku neovisno o monitoru VMDP, DUMP mod je pod kontrolom vlastitog instalacionog parametra koji uključuje ili isključuje DUMP mod u radu monitora VMDP. U vrijeme pisanja ovog rada task DUMP nije bio napravljen, pa je DUMP mod isključen u instaliranoj verziji monitora.

3. Praćenje i analiza dinamike rada sistema

Za uvid u dinamiku rada operativnog sistema i svakog aktivnog batch programa, od posebnog su značaja faktori napredovanja FN koji se računaju na slijedeći način:

$$FN(MASTER) = T_M/t_1$$

$$FN_i = T_i/t_1$$

gdje su

- T_M -vrijeme CPU koje su između dva uzastopna stanja MONITOR mod 1 potrošili taskovi operativnog sistema i egzekutiva EXEC
- T_i -vrijeme CPU koje su između dva uzastopna stanja MONITOR mod 1 potrošili taskovi i-tog batch programa
- t_1 -realno vrijeme između dva uzastopna stanja MONITOR mod 1

U svakom stanju MONITOR mod 1 mora biti

$$FN(MASTER) + \sum_i FN_i = 1$$

gdje indeks "i" ide po svim aktivnim batch programima. Na ekranu se faktori napredovanja iskazuju kao postotci, a njihove vrijednosti i međusobni odnos ovise o broju aktivnih batch programa, o njihovom sastavu (compute programi, IO programi), kao i o načinu na koji MASTER razrješava konflikte u zahtjevima za resursima sistema.

Suma $\sum_i FN_i$ predstavlja direktno iskoristivost CPU, a FN(MASTER) gubitak u iskoristivosti CPU zbog rada operativnog sistema i besposlenosti CPU.
Na primjer, ako u sistemu nema aktivnih batch programa, tada je FN(MASTER)=1oo%, a CPU troše jedino task besposlenosti i, naravno monitor VMDP.
Ako je aktivan jedan ili više "čistih" compute programa, tada je FN(MASTER)=4%, pa overhead zbog rada operativnog sistema potječe samo od rada monitora VMDP i rada egzekutive EXEC na obradi time slice prekida.
Ako je workload takav da ima i compute i IO programa tada se FN(MASTER) kreće od 1o% do 5o% ovisno o karakteristikama takve skupine aktivnih programa.

Posebno nepovoljan slučaj u iskoristivosti CPU javlja se onda kada operativni sistem konflikte u zahtjevima za memorijom razrješava čestom suspenzijom korisničkih taskova na magnetski disk. Gubitak iskoristivosti CPU tada može porasti i na 6o - 7o%, pa sistem radi vrlo neefikasno (trashing). Srećom, ovakve situacije su vrlo rijetke. Pored indikacije ovakvih stanja u radu sistema, monitor VMDP može pomoći operatoru da formira takav workload koji neće dovesti sistem u tako neefikasan rad.

Pored indikacije iskoristivosti CPU, faktori napredovanja pokazuju i zastoje u radu batch programa. Ako vrijednost FN aktivnog programa kroz duže vrijeme postane jednaka nuli, to je znak da task ili taskovi tog programa ne troše CPU nego čekaju na neki drugi resurs ili intervenciju operatora. Kada vrijednosti FN svih aktivnih batch programa postanu jednake nuli, a ni jedan od njih ne čeka neku intervenciju operatora, to je znak da je sistem pao u mrtvu petlju (deadlock).
Jedini taskovi koji u takvoj situaciji troše CPU su task besposlenosti (*IDL) i monitor VMDP. Komandom JOBT se za svaki batch program tada mogu naći imena taskova koji su u stanju čekanja na neki resurs, a komandom TASK dobiti podaci iz njihovih vektora stanja, odakle postaje vidljiv uzrok mrtve petlje.

4. Ugradnja monitora VMDP u operativni sistem MASTER

Monitor VMDP kodiran je u asemblerskom jeziku, a čini ga 17 programskih modula. Unutar računarskog sistema monitor VMDP čini jedinstvenu izvršnu cjelinu - task *MON. Task *MON ugrađen je u rezidentni dio operativnog sistema, a kao task ima posebne privilegije taskova operativnog sistema. Ono što razlikuje task *MON od ostalih standardnih taskova koji su u vlasništvu MASTER-a, je njegov prioritet prilikom dodjele CPU.
Posebna rutina egzekutive EXEC vrši dodjelu CPU taskovima kombinacijom discipline round robin (time slice) i algoritma koji je najbliži disciplini SPN (shortest process next) i gdje se CPU dodjeljuje tasku sa trenutno najvišim prioritetom.

Prioritet taskova operativnog sistema je nepromjenljiv i veći od granice koju mogu doseći obični korisnički taskovi. Prioritet korisničkih taskova se mijenja u vremenu i funkcija je više veličina od kojih je dominantna gustoća IO prekida koje generira task. Prioritet korisničkog taska raste sa gustoćom IO prekida.

Zbog svoje namjene nadgledanja rada sistema u što pravilnijim vremenskim razmacima, task *MON

mora dobiti CPU što je moguće brže od trenutka kada ga iz stanja mirovanja (pause u stanju MONITOR mod 2) posebna rutina egzekutive stavi u listu kandidata za CPU. To je postignuto time da mu je prilikom ugradnje u operativni sistem dodjeljen fiksni visoki prioritet, veći i od prioriteta kojeg imaju standardni rezidentni taskovi operativnog sistema MASTER. To ne ometa normalno natjecanje drugih taskova za CPU, jer *MON dobiva procesor relativno rijetko (default je 5oo ms), a vrijednost kvantuma (time slice) stavljena je na 3o ms, što omogućuje da u vrijeme pauze u radu monitora VMDP desetak drugih taskova dobije CPU.
Interval u kojem se ponavlja stanje MONITOR mod 2 ograničen je sa donje strane sa loo ms, kako bi se onemogućio overhead u radu CPU zbog nepažnje operatora prilikom upotrebe komande TIM2. Isto tako, vrijednost TIM1 ograničena je sa donje strane na looo ms.

Prilikom svog rada monitor VMDP troši resurse koji su normalno namjenjeni korisnicima. To su vrijeme CPU, centralna memorija i kanalni procesor. Prilikom izgradnje monitora VMDP trebalo je paziti da utrošak tih resursa bude što je moguće manji. Zbog toga su programski moduli pisani u asemblerskom jeziku, a pažnja je posvećena pisanju što kompaktinijeg kôda, i posebno, realizaciji što kraćih kritičnih zona u stanjima MONITOR mod 1 i MONITOR mod 2. Svi programski moduli, uključujući različite tabele i buffer ekrana zauzeli su zajedno prostor od 16oo riječi (24 bita) rezidentne memorije.

Gubitak (overhead) resursa zbog prisustva i rada monitora VMDP je 2.5% centralne memorije i oko 4% vremena rada CPU kod default vrijednosti TIM1 i TIM2.
Utrošak vremena rada kanalnog procesora nije izmjeren, ali je praćenje rada sistema pokazalo da rad monitora ne usporava zamjetno rad korisnika na drugim terminalima. Dakle, overhead u radu sistema zbog prisutnosti monitora VMDP je vrlo mali.
Za izradu monitora od ideje do implementacije, te pisanja dokumentacije za održavanje i uputstva za korištenje, utrošeno je 45 dana po čovjeku.

5. Zaključak

U toku nekoliko mjeseci rada monitora VMDP, pokazalo se da je njegova izgradnja bila opravdana u odnosu na motive i potrebe istaknute u uvodu ovog rada. VMDP je omogućio praćenje i detaljan uvid u rad računarskog sistema, olakšavajući time rad operatora i sistemskog osoblja. Sve neobične manifestacije ili zastoji u radu sistema, mogu se sada lako i brzo uočiti i analizirati, i pri tome uštedjeti vrijeme računara i sistemskog osoblja. Znatno je olakšano praćenje rada sistema od strane operatora, a time i njegova efikasnost pri posluživanju računara. Zamjetan je edukacioni efekt podataka koje daje monitor VMDP na suradnike kojima rad operativnog sistema nije bliži interes.
Kao mogući nastavak ovog rada očekuje se proširenje monitora VMDP za prikaz aktivnosti IO sistema.

6. Literatura

1. Library Generation and Maintenance Manual Pub.No. 6o4154oo, CDC Software Public. Div.

2. Hansen P.B.: Operating System Principles Pretince-Hall, 1973.

3. Svobodova L.: Computer Performance Measurement and Evaluation Methods: Analyses and Applications, Elsevier, New York 1976.

INFORMATICA 4/1982

ZASNOVA ZANESLJIVOSTI PROGRAMSKIH SISTEMOV

ADOLF ŽIŽEK

BETNAVSKA 58, 62000 MARIBOR

UDK: 681.3.519.718

Računalniških programov ni mogoče sestavljati brez napak. Formalne napake je sicer mogoče odkrivati, vsebinske pa se lahko pojavijo šele pri določenih vhodnih podatkih, po nedoločenem številu tekov programa. Podan je osnutek modela, ki obravnava programe kot naključne sisteme in podaja njihovo zanesljivost po analogiji z zanesljivostjo tehničnih izdelkov (hardwara).

Software can't be made free from fault in general. Formal software faults can be found by usual procedures. Faults as to content can appear unexpextedly at some input data. Reliability schema treating the last case on the hardware analogy is discussed.

SOFTWARE RELIABILITY SCHEMA

1. Uvod

Če poznamo matematične izraze, ki opisujejo obnašanje določenega stvarnega sistema, bi pričakovali, da bomo s pomočjo računalnika pravilno izračunali odvisnosti med sistemskimi količinami. Žal to ne drži vedno, zaradi odpovedi v računalniškem sistemu ali zaradi napak pri programiranju; z le-temi se bomo ukvarjali v nadaljevanju in sicer s takšnimi vsebinskimi, ki jih ni mogoče odkriti z običajnimi postopki, s katerimi odkrivamo formalne napake. V takšnih okoliščinah lahko po mnogih pravilnih izidih nepredvideno nastopi napačen izid, pri določenih vstopajočih podatkih.

Če obravnavamo napačni izid kot odpoved programskega sistema, lahko po analogiji s tehničnimi izdelki, ki tudi odpovedujejo, govorimo o <u>zanesljivosti programskega sistema</u>.

Pregled načinov, s katerimi lahko obravnavamo zanesljivost programskih sistemov, z navedbo ustrezne literature, je mogoče najti v [1].

2. Postavitev modela

Opišimo ta vprašanja še formalno. Bodi X množica vstopajočih podatkov, Y pa množica izidov. Sprogramirani matematični izraz določa preslikavo e: $X \rightarrow Y$, ki priredi vsakemu vstopajočemu podatku (lahko je tudi vektorski) točno en izid (tudi ta je lahko vektorski). Če zanemarimo odpovedi v računalniškem sistemu, preslikuje program vstopajoče podatke v pripadajoče izide s preslikavo f: $X \rightarrow Y$. Pri pravilnem programu se preslikavi e in f skladata, če pa je v programu napaka, se lahko obe preslikavi skladata le pri nekaterih podatkih, ali pa sploh ne. Bodi $A \subseteq X$ množica vseh vstopajočih podatkov, za katere se obe preslikavi skladata. Če tedaj vstavimo v računalnik podatke iz množice A, bodo izidi pravilni; v primeru, da vstavimo podatke iz množice $X \setminus A$, pa bodo izidi napačni. Verjetnost pravilnega izida je zato enaka verjetnosti, da smo vstavili v računalnik podatek iz množice A.

Formalizirajmo doslej povedano še s stališča verjetnostnih ved [3]. Recimo, da je $(X,\mathcal{S})$ merljivi prostor, $(\Omega,\mathcal{F},P)$ pa verjetnostni prostor. Naj bo A element σ-algebre $\mathcal{S}$. Tedaj je izbira vstopajočih podatkov opisana kot naključna spremenljivka $\xi : \Omega \rightarrow X$. Ker uporabljamo isti program ponavadi večkrat, pripada vsakemu teku programa naključna spremenljivka ξ_n. Množica vseh takih naključnih spremenljivk tvori naključni potek (stohastični proces) $\{\xi_n;\ n \in N\}$. Če je $G \subseteq N$ podmnožica tekov programa, je verjetnost, da v podmnožici G ne bo napake, enaka takoimenovani <u>uspešnosti</u> [4] naključnega poteka, $U(G,A;\xi)$. V primeru, da izberemo množico $G = \{1,2,\ldots,n\}$, se uspešnost sklada z <u>zdržljivostjo</u>

$$R(n) = U(\{1,2,\ldots,n\},A;\xi) \qquad (1)$$

kar je enako verjetnosti, da v prvih n tekih programa ne bo nastopila napaka.

Število tekov programa do prvega napačnega izida imenujemo po analogiji s tehničnimi izdelki <u>trpežnost</u> programa T, ki je naključna količina. Naključni troežnosti lahko priredimo verjetnostne funkcije, znane iz zanesljivosti tehničnih izdelkov: porazdelitveno funkcijo.

$$F(n) = P\{T \leq n\} \qquad (2)$$

zdržljivost

$$R(n) = P\{T > n\} \qquad (3)$$

in verjetnost, da nastopi napačen izid med tekoma n_0 ter n_1

$$P\{n_0 < T \leq n_1\} = \sum_{n_0}^{n_1} F(n) \qquad (4)$$

V nadaljevanju bomo obravnavali le primer, da izbira vstopajočih podatkov pri poljubnem teku programa ni odvisna od izbire podatkov v prejšnjih tekih. Če je p verjetnost, da bo izbran vstopajoči podatek iz množice podatkov A, je zdržljivost, po verjetnostnih zakonitostih, enaka

$$R(n) = p^n \qquad (5)$$

verjetnost, da nastopi napaka do vključno n-tega teka programa, pa

$$F(n) = 1 - R(n) = 1 - p^n \qquad (6)$$

R(n) je monotono padajoča funkcija, F(n) pa monotono rastoča.

Ker je po zanesljivostni teoriji srednja trpežnost (MTTF)

$$E[T] = \int_0^\infty R(t)\,dt \qquad (7)$$

je v našem, diskretnem primeru

$$E[T] = \sum_{n=1}^{\infty} R(n) \qquad (8)$$

od tod in iz (5) izhaja, zaradi lastnosti vrst,

$$E[T] = \frac{p}{1-p} \qquad (9)$$

Če je $q = 1 - p$ verjetnost napačnega izida, je srednja trpežnost (9) enaka tudi

$$E[T] = \frac{1}{q} - 1 \qquad (10)$$

Srednja trpežnost E[T] predstavlja srednje število tekov programa do prvega pojava napačnega izida.

Če tečejo v zanesljivostno neodvisnem, zaporednem programskem sistemu njegove sestavine (podprogrami) zaporedoma m_1-krat, m_2-krat, ..., m_u-krat, je verjetnost pravilnega izida v enem teku programskega sistema enaka

$$R(1) = p_1^{m_1} p_2^{m_2} \cdot \cdot p_u^{m_u} \qquad (11)$$

Ker velja v tem primeru

$$R(n) = (R(1))^n \qquad (12)$$

je zdržljivost tega programskega sistema

$$R(n) = \left(p_1^{m_1} p_2^{m_2} \cdots p_u^{m_u}\right)^n \qquad (13)$$

njegova srednja trpežnost pa, v skladu s (6),

$$E[T] = \frac{p_1^{m_1} p_2^{m_2} \cdots p_u^{m_u}}{1 - p_1^{m_1} p_2^{m_2} \cdots p_u^{m_u}} \qquad (14)$$

3. Zanesljivostni preskusi

V skladu z verjetnostnimi načini poteka testiranje verjetnosti napake, oziroma določanje verjetnostne porazdelitve napake F(n), tako: Vzamemo dovolj veliko množico programov, ki imajo vsi isto vrednost parametrov: izkušenost programerja, zahtevnost področja, dolžina programa ipd., pustimo teči programe pri naključno izbranih vstopajočih podatkih ter pri vsakem od programov ugotavljamo zaporedni tek, pri katerem je prišlo do napake. Zdaj izračunamo verjetnosti $q_1, q_2, \ldots q_n, \ldots$, pri čemer je q_n verjetnost, da je nastopila napaka pri n-tem teku programa. Nato dobimo, v skladu z lastnostmi diskretne verjetnostne porazdelitve porazdelitveno funkcijo naključne trpežnosti kot

$$F(n) = \sum_{i=1}^{n} q_i \qquad (15)$$

ter pripadajočo zdržljivost kot

$$R(n) = 1 - F(n) = 1 - \sum_{i=1}^{n} q_i \qquad (16)$$

Verjetnost pravilnega izida pa dobimo s pomočjo (5) kot

$$p = \sqrt[n]{R(n)} = \sqrt[n]{1 - \sum_{i=1}^{n} q_i} \qquad (17)$$

Če velja domneva o neodvisnosti naključnega poteka $\{\xi_n : n \in N\}$, se smejo vrednosti za p, izračunane s poljubnim številom n, razlikovati le neznatno. Če je to razlikovanje znatno, domneva o neodvisnosti naključnega poteka ne velja. Vendar je tudi v tem primeru smiselno poiskati in uporabljati srednjo vrednost za p, saj je računski postopek v primeru, ko je zdržljivost R(n) potenčna funkcija, dokaj enostaven. Podobno pot poenostavitve so ubrali tudi na področju neelektronskih tehničnih sistemov, kjer uporabljajo (zaradi enostavnosti pri ocenitvi zdržljivosti) eksponentni potek zdržljivosti $R(t) = \exp(-\lambda t)$, kljub temu, da so dejanski poteki bolj zapleteni.

4. Sklep

Če se zgledujemo po elektronskih sistemih, bi bilo mogoče tudi pri programskih sistemih poenostaviti zanesljivostne računske postopke na tak način, da bi v posebnih priročnikih zbrali verjetnosti pravilnih izidov za posamezne tipe programov. Parametri, ki vplivajo na to verjetnost, so predvsem izkušenost programerjev, zahtevnost področja, ki ga obravnava program, in dolžina programa. Ti parametri so analogni obremenitvam pri elektronskih elementih [4].

Če uspemo na podlagi meritev ugotoviti verjetnosti pravilnih izidov in jih zbrati v priročniku, je nadaljnji postopek enostaven [4]: Ocenimo parametre sestavin programa, iz omenjenega priročnika poiščemo ustrezne verjetnosti, ocenimo število tekov posameznih podprogramov v enem računskem postopku, nato pa uporabimo formulo (13), oziroma (14).

LITERATURA

[1] A. Pavić: Teorija pouzdanosti i kvaliteta programskih sistema, Elektrotehnika, 24(1981)4, 111-114

[2] T. Thayer et al.: Software reliability, North Holland Publishing Co., 1978

[3] A. Žižek: Matematički model opće pouzdanosti, Elektrotehnika, 24(1981)3, 191-194

[4] A. Žižek, L. Gyergyek: Zanesljivost nakljućno obremenjenih sistemov, Elektrotehniški vestnik, 48(1981)5, 289-294

INFORMATICNI CRNI HUMOR

INFORMATIČNI CRNI HUMOR

INFORMATICNI CRNI HUMOR

****
Kaj vse lahko zraste na domačih tleh
****

Na računalniško smer visokošolskega študija se je vpisalo veliko novih študentov. Domala desetkrat več kot pred nekaj leti! Začnejo z osnovami računalništva in informatike. In kakšno znanje lahko pričakujejo?

Iz Objav o študiju na Fakulteti za elektrotehniko (1982-83) povzemamo vsebino predmeta:

1053 Osnove računalništva in informatike - 1. semester 3-2-0

Temeljne funkcije informacijskih strojev. Predstava informacij v računalništvu. Bistvene pridobitve posameznih generacij računalnika. Značilnosti sistema adresiranja. Značilnosti sistema krmiljenja. Mesto programiranega jezika v jezikovni okolici. Značilnosti in razdelitev sodobnih operacijskih sistemov. Vhodne in izhodne možnosti računalniških sistemov. Vrednotenje računalniške tehnike. Informacijski sistem. Upravljavski sistem. Vloga informacijske tehnike in tehnologije v sodobni družbi.

Kakšno znanje bo študent dobil v okviru tega predmeta? Namesto strojev za obdelavo podatkov ga čakajo informacijski stroji (to je verjetno naslednja ali kasnejša računalniška generacija). Informacije kot da izvajajo nekakšno predstavo v računalništvu, se razkazujejo in medseboj igrajo pred številnim občinstvom. Nato nastopi borba med generacijami računalnika, ki prinese nekakšne pridobitve, verjetno za uporabnike. Sistemi začnejo nato adresirati in krmiliti. Vendar se pojavi mesto programirnega jezika v jezikovni okolici. Ta jezik očitno stremi za vlogo, ki naj bi jo imel v uporabniškem okolju. Operacijske sisteme nekaj razdeljuje, možnosti kot vseej so pa tudi vhodne in izhodne. Tehnika, kot je računalniška, se lahko vrednoti. Informacijski in upravljavski sistem ima naposled tehnološko in družbeno vlogo.

Humorna interpretacija izvirnega teksta se tu vsiljuje predvsem zaradi njegovih pomenskih nejasnosti, da ne rečem zaradi pomanjkljivega strokovnega izražanja. Besedni pašaluk izvirnega teksta pa me sili k razmišljanju predvsem tudi zaradi neobičajne vsebinske zgradbe predmeta, ki je preprosto izven podobnih pedagoških standardov.

IFIP-UNESCOvo priporočilo (glej "Univerzitetni pouk računalništva I", Informatica 2/1980, str. 32), v katerem je podoben osnovni predmet "Uvod v informatiko I, II", ima tole konkretno vsebino:

organizacija računalnikov, programiranje, razvoj algoritmov, osnove strukturiranega programiranja, preizkušanje programov in odpravljanje napak, procesiranje nizov, iskanje in sortiranje, strukture podatkov, rekurzija.

Ali je primerjava domačega in mednarodno priporočenega predmeta za dežele v razvoju sploh mogoča? Trdim, da so možnosti za primerjavo neznatne, da je doma oblikovani predmet s pedagoškega vidika celotnega predmetnika didaktično in vsebinsko skrpucalo, da se v njem mešajo strokovni elementi z elementi nekakšne dnevne pragmatike in morda s prepričanjem, da računalništvo ni stroka in da je za našega študenta (prihodnjo delovno generacijo) dobro vse, kar se nekompetentno in neodgovorno nakljućno izbere. In takšno lekcijo neresnosti naj damo številnim novim študentom že ob samem vstopu v visokošolski študij?

V osnovnem predmetu, kot je koncipiran doma, želimo dati študentu nekaj več, kot je metodološko znanje, dajemo mu nekakšno enciklopedijo naših (strokovno subkulturnih) predstav, na samem začetku mu damo vedeti, da je študij šele začetek strokovnih (nestrokovnih) nejasnosti, ki se bodo med študijem nadaljevale in poglabljale. S tem učimo prihodnjo generacijo tiste nejasnosti, ki je preteklim generacijam onemogočila tehnološki, strokovni in naposled tudi poslovni prodor na zahtevnem delovnem področju.

Ob primeru osnovnega učnega predmeta se postavlja vprašanje, kako je s celotnim predmetnikom za smer računalništva, kakšni kvalitativni standardi so bili upoštevani (če so sploh bili kakšni), ali ni predmetnik nastal kot nakljućna zbirka predmetov neustreznega in predvsem nesodobnega znanja predavateljev, ali so se pri tem sploh postavila vprašanja za dvig znanja predavateljev, za konkretno in tehnološko ustrezno izvajanje pripadajočih vaj? Kaj so k oblikovanju programa lahko in smeli prispevati proizvajalci in drugi zunanji subjekti, kjer je strokovno znanje na višji ravnini kot npr. na fakulteti? K temu kompleksu vprašanj sodi vsekakor tudi vprašanje, ali so bili v redni učni proces pritegnjeni tisti zunanji predavatelji, ki določene discipline poklicno, metodološko in praktično obvladajo.

Vprašanje ustreznega visokošolskega študija postaja vsebolj problem upoštevanja določenih kakovostnih standardov (IFIP-UNESCO, ACM itd.) in pridobitve kakovostnih učiteljev (izkušenih, opremljenih z metodološkim znanjem predmeta), ki zmorejo izvajati poučevanje predmeta skladno z dovolj zahtevnim (kakovostnim) učnim programom. Tehnološki in poslovni prodor brez doslednega kakovostnega pristopa k visokošolskemu študiju računalništva v prihodnje ne bo mogoč.

A. P. Železnikar

INFORMATICNI CRNI HUMOR

INFORMATICNI CRNI HUMOR

INFORMATICNI CRNI HUMOR

JEDAN ALGORITAM SAŽIMANJA SRPSKOHRVATSKIH REČI

DRAGAN MIHAJLOVIĆ,
DANILO OBRADOVIĆ

FAKULTET TEHNIČKIH NAUKA
INSTITUT ZA MERNU TEHNIKU I UPRAVLJANJE
NOVI SAD

UDK: 681.3.06:808.61

Razmatra se algoritam sažimanja imenica i prideva iz srpskohrvatskog jezika. Svrha algoritma je sažimanje različitih oblika jedne imenice ili prideva u jedan isti sažeti oblik i sažimanje oblika različitih imenica ili prideva u različite sažete oblike. Daju se pravila za sažimanje sa prikazom dobijenih eksperimentalnih rezultata. Navode se mogućnosti primene algoritma.

AN ALGORITHM FOR SERBOCROATION WORDS COMPRESSION: The paper descusses an algorithm for the compression of nouns and odverbs in Serbocroation language. The purpose of the algorithm proposed is to compress different forms of a noun (or adverb) into unigue compressed form, as well as to compress forms of different nouns (or adverbs) into different compressed forms. Compression rules, as well as experimental results obtained are also presented the field of the possible application of the presented algorithm is also given.

1. UVOD

Problem sažimanja je nastao kao potreba jedne vrste automatizovanih sistema za memorisanje i pretraživanje informacija koji radi na osnovu prirodnog jezika u kojem se koristi sažimanje leksičkih jedinica teksta (kako dokumenata tako i upita). Smisaoni sadržaj dokumenata i upita nose imenice i pridevi. Zbog toga se problem sažimanja razmatra nad rečima srpskohrvatskog jezika (SH-jezik) koje su imenice ili pridevi. Problem sažimanja imenica i prideva razmatra se u novim granama lingvistike (matematička, strukturno-primenjena, inžinjerska, računarska) koje su nastale kao posledica sve većeg razvoja informacione tehnologije i upravljanja koji sve više primenjuju automatsku obradu tekstova.

Problem sažimanja imenica i prideva razmatran je u nekim stranim jezicima /1,2/. Primenljivost dobijenih rezultata ohrabruje i zahteva dodatna istraživanja na usavršavanju postavljenih rešenja. Kako se kod nas poslednjih godina uvodi sve veći broj automatizovanih sistema za memorisanje i pretraživanje informacija rešavanje problema sažimanja imenica i prideva u SH-jeziku ima kako teoretski tako i praktični značaj.

Pod sažimanjem imenica i prideva podrazumeva se neki algoritamski postupak koji oblik jedne imenice ili prideva transformiše u neki drugi oblik (sažeti oblik) tako da oblici različitih imenica ili prideva imaju različite sažete oblike a svi oblici jedne imenice ili prideva imaju isti sažeti oblik. Oblik imenice ili prideva je realizacija imenice ili prideva u nekom iskazu. Algoritam je dobio ime sažimanje zato što se njime oblik imenice ili prideva transformiše u oblik koji je po broju slova u najvećem broju slučajeva manji od početnog oblika.

2. POSTAVKA PROBLEMA

Problem sažimanja reči (imenica i prideva) može se predstaviti na sledeći način:

Neka je data reč $R=r_1r_2,\ldots,r_n$ gde je r_i slovo iz SH-alfabeta. Reč R želimo da sažmemo (transformišemo) u reč $S=s_1s_2,\ldots,s_m$ gde je s_j slovo iz SH-alfabeta, decimalna brojka ili specijalni znak. Označimo sa R'_k $k=1,2,\ldots,p$ sve oblike reči R' (za imenice i prideve to su padežni oblici). Sažimanjem bilo kog oblika R'_k treba se dobiti oblik S'. Neispunjenje ovog uslova dovodi do sinonimije sažimanja. Ako je R'_k oblik reči R' a R''_k oblik reči R" sažimanjem R'_k treba dobiti S', a sažimanjem R''_k treba dobiti

S". Homonimija sažimanja nastaje kada je S'=S".

Sažimanjem reči treba izbeći i sinonimiju i homonimiju. U sistemima za memorisanje i pretraživanje informacija sinonimija i homonimija sažimanja reči dovode do smanjenja potpunosti i tačnosti dobijanja informacija. Smanjenje i sinonimije i homonimije sažimanja su protivrečni zahtevi pa ih je teško izbeći.

3. OPIS ALGORITMA SAŽIMANJA

Algoritam sažimanja treba da ispuni sledeće opšte zahteve:

a) jednaka maksimalna dužina sažetih oblika,

b) dobijanje različitih sažetih oblika od različitih imenica ili prideva (razlikovanje imenica ili prideva),

c) sažimanje svih oblika jedne imenice ili prideva u jedan isti oblik.

Zahtev a) može se ostvariti dosta jednostavno. Ako se statistički utvrdi maksimalna dužina oblika imenica i prideva u iskazima onda se za maksimalnu dužinu sažetog oblika može uzeti ta veličina. Od interesa je da zbog efikasnog korišćenja računarskog sistema maksimalna dužina sažetog oblika bude što manja. Za maksimalnu dužinu sažetog oblika odabrano je 14 slova.

Zahtev b) može se ostvariti relativno lako. Naime, ako je sažeti oblik jednak samom obliku imenice ili prideva ovaj zahtev bi bio ispunjen veoma jednostavno. Medjutim, zahtev b) treba posmatrati povezano sa zahtevom c). Posmatrajući zahtev c) potrebno je pronaći postupak koji će od oblika imenice ili prideva u sažeti oblik preneti ono što je zajedničko za sve oblike jedne imenice ili prideva. Analizom nastajanja oblika imenica i prideva u iskazima zaključuje se da se promena imenica i prideva ne sastoji u dopisivanju nastavka na osnovu. S obzirom da je kao ulazni podatak uzet samo oblik imenice ili prideva to osnovu za dobijanje rešenja čini pretpostavka da se najveći broj imenica i prideva menja tako da se na osnovu dopiše nastavak.

Analizom promene imenica i prideva /3,4/ za algoritam sažimanja utvrdjene su sledeće važne činjenice:

1. U nastavcima imenica i prideva osim samoglasnika upotrebljavaju se i suglasnici M,G,H i J.

2. U nastavcima imenica najčešće se pojavljuje 'OV' i 'EV' (kao infiksi).

3. Česta pojava prelaska K u C (palatalizacija).

4. Česta pojava nepostojanog A.

Na bazi utvrdjenih činjenica o promeni imenica i prideva predlažu se sledeća pravila za sažimanje oblika imenica i prideva:

(P1) Identifikovanje na kraju dela reči od četvrtog slova i izbacivanje segmenta sledećeg oblika:

$v^n(n>0)$ $v\in(A,O,I,E,U,M,G,H,J)$.

Ukoliko su slova M, G, H, ili J četvrto slovo reči onda ne pripadaju navedenom segmentu.

(P2) Identifikovanje na kraju reči i izbacivanje segmenta oblika 'OV' i 'EV'.

(P3) Ukoliko je poslednje slovo reči C zameniti ga sa K.

(P4) Počinjući od četvrtog slova reči izbaciti samoglasnik A.

(P5) Za sažeti oblik uzeti najviše četrnaest slova.

Navedena pravila primenjuju se u navedenom redosledu. Grupe slova NJ, LJ i DJ bez obzira da li predstavljaju jedan ili dva glasa kodirane su na sledeći način: NJ→$, LJ→% i DJ→&.

Primeri sažimanja:

ZAK%UČAK P4→ZAK%UČK P5→ZAK%UČK,
ZAK%UČKA P1→ZAK%UČK P5→ZAK%UČK,
ZAK%UČKU P1→ZAK%UČK P5→ZAK%UČK,
ZAK%UČKOM P1→ZAK%UČK P5→ZAK%UČK,
ZAK%UČCI P1→ZAK%UCC P3→ZAK%UČK P5→ZAK%UČK,

CRN P5→CRN,
CRNOG P1→CRN P5→CRN,
CRNIH P1→CRN P5→CRN,

HLEB P5→HLEB,
HLEBOVIMA P1→HLEBOV P2→HLEB P5→HLEB.

4. OCENA EFEKTIVNOSTI ALGORITMA

Prema postavljenom algoritmu sažimanja napisan je program na COBOL-u i izvršeno testiranje.

Iz jednog skupa deskriptora koji se koriste za indeksiranje dokumenata i upita iz više oblasti (društveno-politička, pravna, ekonomska, tehnika, poljoprivreda itd.) izdvojeno je 1600 deskriptora (imenica u nominativu jednine ili množine) od po jedne reči. Iz istog skupa izdvojeno je 600 prideva. U navedenom skupu od 2200 imenica i prideva homonimija sažimanja pojavila se u sledećim slučajevima:

dnevnici→DNEVNIK,
dnevnice→DNEVNIK;
luk→LUK, luka→LUK;
narkoman→NARKOMN,

narkomanija→NARKOMN;
'obveznici→OBVEZNIK,
obveznica→OBVEZNIK;
prostor→PROSTOR,
prostorije→PROSTOR;
radio→RAD, radovi→RAD;
sud→SUD, sudije→SUD;
šećerane→ŠEĆERN,
šećerna→ŠEĆERN;
žena→ŽEN, ženeva→ŽEN.

Homonimiju sažimanja izazivaju i drugi oblici navedenih imenica i prideva (napr. dnevnika, dnevniku, dnevnikom, šećeranama itd.). Navedene reči treba uvrstiti u skup reči za čije sažimanje treba pronaći posebna pravila sažimanja. Potpuno eleminisanje i sinonimije i homonimije bez dodatnih ulaznih podataka ne može se eliminisati (napr. oblik luka može nastati kao genitiv jednine od imenice luk ili može da predstavlja nominativ jednine imenice luka).

Radi provere sinonimije sažimanja izvršena je obrada 6000 oblika imenica i prideva, pri čemu su za imenice i prideve uzeti samo karakteristični padežni oblici. Sinonimija sažimanja pojavila se u sledećim slučajevima:

gra&anin→GRA&NIN, gra&ani→GRA&N;
k$iga→K$IG, k$izi→K$IZ;
predlog→PREDL, predlozi→PREDLOZ;
nalog→NAL, nalozi→NALOZ;
zadruga→ZADR, zadruzi→ZADRUZ;
pruga→PRUG, pruzi→PRUZ;
prt%ag→PRT%, prt%azi→PRT%Z;
drvo→DRV, drveta→DRVET.

Homonijima sažimanja pojavila se u sledećim slučajevima:

Bečej→BEČ, Beč→BEČ;
delo→DEL, delovi→DEL;
spomenik→SPOMENIK,
spomenica→SPOMENIK;
voda→VOD, vodovi→VOD;
kosa→KOS, Kosovo→KOS;
list→LIST, lista→LIST;
topole→TOPOL, topologija→TOPOL;
elektroenergetika→ELEKTROENERGET,
elektroenergetski→ELEKTROENERGET.

Iz poslednjeg primera se vidi da je on nastao kao posledica maksimalne dužine sažetog oblika od 14 slova.

Pored nabrojanih slučajeva homonimije došlo je i do njenog pojavljivanja u desetak slučajeva kada su u pitanju lična imena ili prezimena (napr. Petković - PETKOVIĆ, Petaković - PETKOVIĆ; Mileva - MIL, Miloje - MIL itd.). Kako se imena lica u većini slučajeva pojavljuju u tekstovima sa punim imenom i prezimenom to se do pojave homonimije u sistemima za memorisanje i pretraživanje informacija može desiti u slučajevima da istovremeno postoji homonimija i imena i prezimena.

Posebna pravila za sažimanje oblika imenica ili prideva koji izazivaju homonimiju ili sinonimiju sažimanja mogu se definisati za svaki oblik reči ili mogu biti opštija. Na primer, ako oblik reči počinje sa 'ženev', tada se ne primenjuje pravilo P2. Sažimanjem svih oblika imenice ženeva dobija se ŽENEV, te nestaje homonimija sažimanja sa oblicima imenice žena - ŽEN.

5. ZAKLJUČAK

Predloženi algoritam sažimanja imenica i prideva SH-jezika pored dobrih rezultata ima i odredjenih nedostataka koji su iskazani kroz sinonimiju i homonimiju sažimanja. Potpuno eleminisanje sinonimije i homonimije sažimanja je nemoguće.

Primenom predloženog algoritma pruža se mogućnost automatskog pronalaženja svih pojavljivanja oblika jedne imenice ili prideva u nekom skupu tekstova na SH-jeziku. Kako se uz svako pojavljivanje nekog oblika nalaze reference tekstova ili delova tekstova u kojima se taj oblik pojavio omogućeno je izdvajanje svih tekstova koji sadeže zadatu imenicu ili pridev. Ovakvom tehnikom realizovan je sistem za komunikaciju sa bazom dokumenata na SH-jeziku.

Dobijeni rezultati s obzirom da se homonimija i sinonimija sažimanja pojavljuje u manje od 0,5% u posmatranom skupu deskriptora, a da bi sigurno ovaj procenat bio znatno manji da su deskriptori grupisani po oblastima podstiče na usavršavanje predloženog algoritma sažimanja imenica i prideva SH-jezika i dalja istraživanja njegove primene.

6. LITERATURA

1. V.P.Zaharov, P.G.Mordovčenko, L.V.Saharnij "SOVERŠENSTVOVANIJE LINGVISTIČESKOVA OBESPEČENIJA V IPS 'BEZTEZAURUSNOVA' TIPA", NTI ser. 2. br.6. pp. 14-19, 1980.
2. I.JU.Vesler "ALGORITM SŽATIJA RUSKIH SLOV UCITIVAJUŠCIJ DLINU SLOV", Strukturna i matematičeskaja lingvistika, br. 8. Kijev, 1980.
3. M.Stevanović SAVREMENI SRPSKOHRVATSKI JEZIK, Beograd, Naučna knjiga, 1970.
4. D.Vitas "GENERISANJE IMENIČKIH OBLIKA U SRPSKOHRVATSKOM JEZIKU", Informatika, br. 3. pp. 34-39, 1980.

KORIŠTENJE SUSTAVA BROJEVA REZIDUA U OBRADI SIGNALA RAČUNALIMA

DRAGAN GAMBERGER

INŠTITUT R. BOŠKOVIĆ, ZAGREB

UDK: 681.3:519.6

Sustav brojeva rezidua (engl. Residue Number System) je način prikaza cijelih brojeva koji omogućuje dekompoziciju operacija zbrajanja, oduzimanja i množenja na više manjih, međusobno neovisnih cjelina. U radu su definirani sustavi brojeva rezidua i razmotreni algoritmi konverzije binarno prikazanog broja u sustav brojeva rezidua i obratno te posebno mogućnosti direktne A/D i D/A konverzije u i iz sustava brojeva rezidua. Ukazano je na primjenljivost algoritma promjenljive baze i specijalnih D/A mreža u slučajevima programske realizacije konverzije za jedno i višeprocesorske sisteme.

RESIDUE NUMBER SYSTEM APPLICATION IN COMPUTER SIGNAL PROCESSING: Residue Number System is the way of integer number representation allowing decomposition of addition, subtraction and multiplication operations in the few smaller, mutually independent parts. In this paper residue number systems are defined and the conversion algorithms for the binary coded number as well as the possibilities of the direct A/D and D/A conversions to and from residue number system are analysed. The applicability of the variable base algorithm and the specially built D/A networks for the software conversions in single - and multiprocessor systems is pointed out.

1. UVOD

Sustav brojeva rezidua je način prikaza brojeva sa skupom manjih brojeva (rezidua) tako da svaki cijeli binarni ili dekadski broj unutar nekih granica ima jednoznačni prikaz u tom skupu. Aritmetičke operacije zbrajanja , oduzimanja i množenja moguće je vršiti između brojeva prikazanih u sustavu brojeva rezidua tako da se operacije izvrše između odgovarajućih elemenata iz skupa brojeva rezidua. Primjena sustava brojeva rezidua omogućuje tako dekompoziciju aritmetičkih operacija sa velikim brojevima na nekoliko paralelno izvedivih aritmetičkih operacija sa manjim brojevima. Ovo svojstvo od posebne je važnosti u digitalnim sistemima za obradu signala u realnom vremenu pogotovo u primjenama koje zahtijevaju potpunu točnost rezultata. Do sada su poznate primjene u digitalnom filtriranju /1/,/2/, a bilo je i pokušaja izgradnje vrlo brzih aritmetičkih procesora za velika računala na osnovu sustava brojeva rezidua. U računarskim sistemima sa više istovremeno aktivnih procesora primjena ovog brojevnog sustava omogućuje da svaki od procesora obrađuje samo brojeve jedne od baza te da se postigne paralelizam u izvođenju aritmetičkih operacija. Pravilan izbor baza omogucuje da i računala sa ograničenom dužinom riječi i brojem internih registara uključena u paralelu realiziraju po potrebi veliki sistem za obradu digitalnih signala.

U radu su definirani sustavi brojeva rezidua a zatim detaljno rezrađeni algoritmi za pretvaranje binarno kodiranog broja u sustav brojeva rezidua i obratno. Algoritmi su ilustrirani sklopovskim realizacijama ali općenito vrijedi da su isti primjenljivi i u slučaju programskih konverzija. Posebna pažnja posvećena je direktnim metodama pretvaranja analognih signala u digitalni prikaz u sustavu brojeva rezidua i obratno te su razrađene metode prikladne za primjene sa računalima.

2. SUSTAV BROJEVA REZIDUA

Sustav brojeva rezidua je sustav koji se formira na osnovu skupa P, $p_i \in P$ i=1,2,..N prirodnih međusobno prim brojeva

$$n.z.d.(p_i,p_j) = 1 \qquad (1)$$

gdje n.z.d. označava funkciju najvećeg zajedničkog djeljitelja. Cijeli broj B prikazuje se u sustavu brojeva rezidua sa uređenom N-torkom brojeva

$$B = <b_1,b_2,..,b_N> \qquad (2)$$

u kojoj su brojevi $b_1,b_2,..,b_N$ određeni jednadžbama

$$B = k_i \cdot p_i + b_i \qquad (3)$$

pri čemu su k_i cijeli brojevi takvi da je za svaki $i = 1,2,..,N$ zadovoljeno

$$0 \leq b_i \leq p_i - 1 \qquad (4)$$

U daljnjem tekstu koristiti će se oznaka

$$b_i = B \bmod p_i \qquad (5)$$

kojom je naglašeno da je b_i reziduum (ostatak) broja B pozitivan broj manji od p_i koji ostaje nakon zbrajanja ili oduzimanja potrebnog višekratnika broja p_i broju B .

Broj $B = <b_1,b_2,..,b_N>$ jednoznačno je prikazan i određen brojevima $b_1,b_2,..,b_N$ u brojevnom sustavu rezidua ukoliko je zadovoljen uvjet

$$0 \leq B \leq R-1 \qquad (6)$$

gdje je R

$$R = p_1 \cdot p_2 \cdot \ldots \cdot p_N \qquad (7)$$

Dokaz:

Pretpostavimo da postoje dva broja B i B'

$$0 \leq B = <b_1,b_2,..,b_N> \leq R-1$$
$$0 \leq B' = <b_1',b_2',..,b_N'> \leq R-1$$

takva da vrijedi $B \neq B'$, $b_1 = b_1'$, $b_2 = b_2'$, $b_N = b_N'$. Apsolutna vrijednost njihove razlike d trebala bi biti

$$d = |B - B'| \leq R-1$$

Iz

$$B = k_i \cdot p_i + b_i$$
$$B' = k_i' \cdot p_i' + b_i'$$

i $B \neq B'$, $b_i = b_i'$ slijedi da je

$$k_i \neq k_i'$$

i da razlika brojeva B i B' mora biti višekratnik od p_i.

Kako su brojevi $p_1,p_2,..p_N$ međusobno prim to je najmanji mogući d

$$d = p_1 \cdot p_2 \cdot \ldots \cdot p_N = R$$

što je u suprotnosti sa pretpostavkom.

Ako je $b_1 = B_1 \bmod p$, $b_2 = B_2 \bmod p$ $b_3 = B_3 \bmod p$ i $B_3 = B_1 * B_2$ gdje * označava funkciju zbrajanja, oduzimanja ili množenja tada vrijedi

$$b_3 = (b_1 * b_2) \bmod p \qquad (8)$$

Dokaz za zbrajanje:

$$B_3 = B_1 + B_2$$
$$k_3 \cdot p + b_3 = (k_1 + k_2) \cdot p + b_1 + b_2$$
$$b_1 + b_2 = k_{12} \cdot p + (b_1 + b_2) \bmod p$$
$$k_3 \cdot p + b_3 = (k_1 + k_2 + k_{12}) \cdot p + (b_1 + b_2) \bmod p$$
$$b_3 = (b_1 + b_2) \bmod p$$

Ako je $B_1 = <b_1^1,b_2^1,..,b_N^1>$, $B_2 = <b_1^2,b_2^2,..,b_N^2>$ $B_3 = <b_1^3,b_2^3,..b_N^3>$ te $B_3 = B_1 * B_2$ prema (8) vrijedi

$$<b_1^3,b_2^3,..,b_N^3> = <b_1^1 * b_1^2 \ , \ b_2^1 * b_2^2 \ , \ldots , b_N^1 * b_N^2> \qquad (9)$$

Ako je zadovoljeno

$$W \leq B_1 \ , \ B_2 \ , \ B_3 \leq R-1+W \ ,$$

gdje je W proizvoljan cijeli broj, iz (6) i (9) je vidljivo da se operacije zbrajanja, oduzimanja i množenja brojeva mogu jednoznačno izvesti tako da se te operacije izvedu nad odgovarajućim elementima u brojevnom sustavu rezidua. Primjena sustava brojeva rezidua omogućuje tako dekompoziciju aritmetičkih operacija na nekoliko međusobno neovisnih i paralelno izvedivih cjelina.

3. KONVERZIJA U SUSTAV BROJEVA REZIDUA

Binarno ili dekadski kodiran broj B moguće je pretvoriti u broj $b = B \bmod p$ prema (5) uzastopnim dodavanjem ili oduzimanjem broja p broju B dok ostatak ne zadovolji

jednadžbu (4). Ovaj algoritam je jednostavan ali dugotrajan za $B \gg p$. Postoji mogućnost da se u početku algoritma pokuša sa većim višekratnicima od p i da se zatim veličina višekratnika postepeno smanjuje. Vrijeme izvođenja algoritma može se time znatno skratiti ali zbog svoje složenosti ovakav način je primjenljiv samo za programske realizacije.

Za binarno prikazani broj B brži način izračunavanja broja $b = B \bmod p$ je pomoću tablice u kojoj su vrijednosti

$$m_j = 2^j \bmod p \qquad (10)$$

za sve $j = 0,1,2,\ldots,L$ tako da vrijedi

$$B_{max} = 2^{L+1}-1 \qquad (11)$$

Ako broj B prikažemo kao

$$B = \sum_{j=0}^{L} B_j \cdot 2^j \qquad B_j = 0,1 \qquad (12)$$

vrijedi

$$b = B \bmod p = \left(\sum_{j=0}^{L} B_j \cdot m_j \right) \bmod p \qquad (13)$$

Dokaz:

$$b = B \bmod p = \left(\sum_{j=0}^{L} \left((B_j) \bmod p \cdot (2^j) \bmod p \right) \bmod p \right) \bmod p$$

$B_j = B_j \bmod p$ jer je $B_j = 0,1$

$2^j \bmod p = m_j$ prema (10)

$(B_j \cdot m_j) \bmod p = B_j \cdot m_j$ te slijedi jednadžba (13).

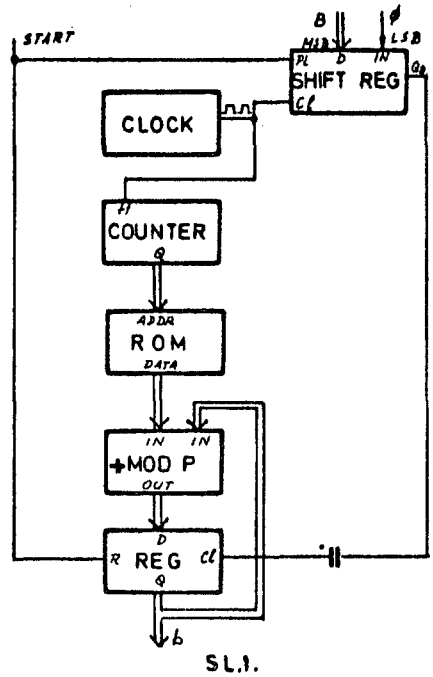
Iz (13) slijedi da se broj b dobije zbrajanjem u modulo p zbrajalu onih članova tablice koji odgovaraju jedinicama u binarnom prikazu broja B. Slika 1 prikazuje realizaciju ovakvog kodera. Prikazani ROM (ispisna memorija, engl. read only memory) treba imati L + 1 lokaciju dužine

$$q = \lceil \log_2 p \rceil \text{ bita}$$

gdje a označava "najmanji cijeli broj veći ili jednak a". U j-toj lokaciji memorije zapamćena je vrijednost m_j .

Ukoliko broj B^j pretstavlja pozitivan ili negativan broj u kodu komplement od 2 što znači

ako $B_L=0$ tada $B \geq 0$ i $|B| = B$



Sl.1.

ako $E_L=1$ tada $B<0$ i $|B| = 2^{L+1}-B$

i želi se dobiti broj $b = B \bmod p$ u odgovarajućem komplementarnom kodu u kojem je

$$-b = p - |b| \qquad (14)$$

na primjer $-1 = p - 1$ vrijedi jednadžba

$$b = \left(B_L(p - m_L) \bmod p + \sum_{j=0}^{L-1} B_j \cdot m_j \right) \bmod p \qquad (15)$$

Dokaz:

Ako je $B_L = 0$ ova jednadžba vrijedi prema (13).

Ako je $B_L = 1$ broj B je negativan a njegova apsolutna vrijednost je

$$|B| = 2^L - B'$$

gdje je B' binarni broj B bez najznačajnijeg bita koji označava predznak broja. Apsolutna vrijednost broja B mod p jednaka je

$$|b| = |B| \bmod p = 2^L \bmod p - B' \bmod p$$
$$= m_L - \left(\sum_{j=0}^{L-1} B_j \cdot m_j \right) \bmod p$$

Prema (14) stvarna vrijednost od b jednaka je

$$b = p - |b| = \left(p - m_L + \sum_{j=0}^{L-1} B_j \cdot m_j \right) \bmod p$$

čime je dokazana tvrdnja (15)

Sklopovska i programska realizacija kodera modulo p za komplementarni kod potpuno je ista ranije opisanoj osim što je u L-toj lokaciji ROM-a umjesto vrijednosti m_L upisana vrijednost $p - m_L$.

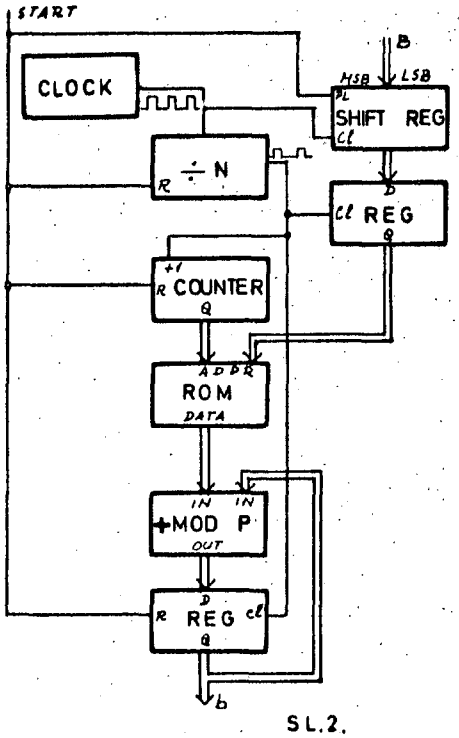
Algoritam kodiranja moguće je ubrzati pribrajanjem modulo p vrijednosti m_j za više binarnih mjesta broja B istovremeno. Ovakva realizacija zahtijeva osim djeljitelja sa N i registra duljine N bita i veći broj lokacija memorije čiji broj je određen izrazom

$$2^N \cdot \left(\left\lceil \frac{L+1}{N} \right\rceil - 1\right) + 2^{L+1-N \cdot \left(\left\lceil \frac{L+1}{N} \right\rceil - 1\right)}$$

gdje je N broj bita koji se istovremeno kodiraju. Ako je $L+1 = k \cdot N$, potreban broj lokacija memorije je

$$\frac{L+1}{N} \cdot 2^N$$

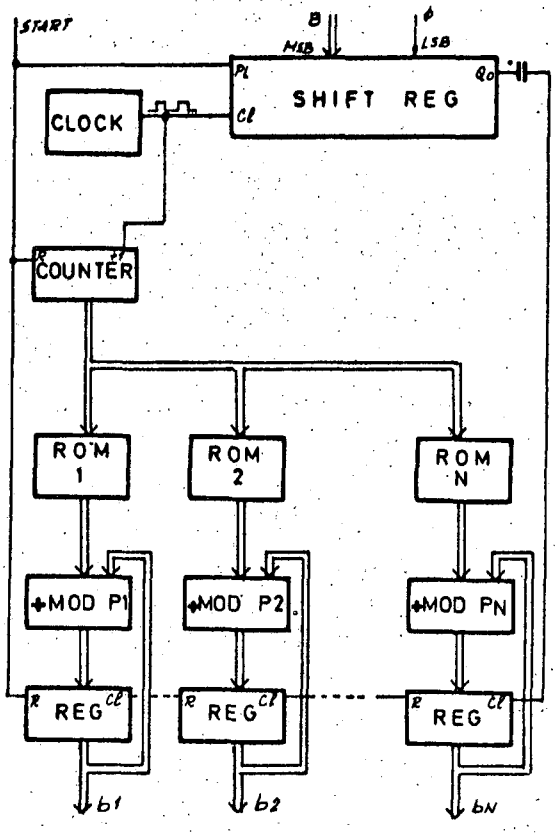
Slika 2 prikazuje izvedbu ovakvog kodera, a tablica 1 sadržaj ROM-a za slučaj $L = 7$, $N = 3$.



SL.2.

I u ovom slučaju ako se radi o komplementarno kodiranim brojevima u ROM se umjesto m_L upisuje svuda vrijednost $p - m_L$.

Slika 3 prikazuje pretvorbu binarno kodiranog broja u sustav brojeva rezidua



SL.3.

$<b_1, b_2, .., b_N>$. Za svaki element skupa P , $p_1, p_2, .., p_N \in P$ potrebno je realizirati poseban koder pri čemu oni mogu koristiti isti izvor takt impulsa, brojač i posmačni registar. Prikazan je jednostavniji sklop sa sukcesivnom akumulacijom za svaki bit posebno.

TABLICA 1

LOKACIJA	SADRŽAJ
0	0
1	m_0
2	m_1
3	$(m_0 + m_1) \bmod p$
4	m_2
5	$(m_2 + m_0) \bmod p$
6	$(m_2 + m_1) \bmod p$
7	$(m_2 + m_1 + m_0) \bmod p$
8	0
9	m_3
10	m_4
11	$(m_4 + m_3) \bmod p$

LOKACIJA	SADRŽAJ
12	m_5
13	$(m_5 + m_3) \bmod p$
14	$(m_5 + m_4) \bmod p$
15	$(m_5 + m_4 + m_3) \bmod p$
16	0
17	m_6
18	m_7
19	$(m_7 + m_6) \bmod p$

4. KONVERZIJA IZ SUSTAVA BROJEVA REZIDUA

Dekodiranje binarnog ili dekadskog broja B iz prikaza u sustavu brojeva rezidua $\langle b_1, b_2, .., b_N \rangle$ moguće je jednoznačno izvesti ako je zadovoljen uvjet $0 \leq B \leq R-1$ izračunavanjem izraza

$$B = \left(\sum_{i=1}^{N} b_i \cdot a_i \cdot r_i \right) \bmod R \qquad (16)$$

gdje r_i označava $r_i = \frac{R}{p_i}$ a a_i se izračunava iz jednadžbe

$$(a_i \cdot r_i) \bmod p_i = 1 \qquad (17)$$

(tzv. Kineski teorem ostatka, engl. Chinese remainder theorem).

Dokaz :

Prema Euklidovom algoritmu postoje svi brojevi $a_1, a_2, .., a_N$ jer su $p_1, p_2, .. p_N$ međusobno prim brojevi i mogu se njime izračunati /5/.

Jednoznačnost prikaza broja iz intervala $0 \leq B \leq R-1$ u sustavu brojeva rezidua vać je ranije dokazana (6),(7). Jednadžba (16) zadovoljava sve uvjete $b_1 = B \bmod p_1$ do $b_N = B \bmod p_N$ jer su modulo p_i svi sumandi osim i-tog u jednadžbi (16) jednaki nula (r_j je višekratnik od p_i za $i \neq j$) a u i-tom sumandu je prema (17) produkt $a_i \cdot r_i$ jednak 1 pa vrijedi $B \bmod p_i = b_i$.

Pri izvođenju dekodiranja prema jednadžbi (16) preporučljivo je prvo izvesti množenja $(b_i \cdot a_i) \bmod p$ jer zahtijevaju znatno kraću duljinu brojeva od ostalih operacija modulo R. Ova množenja je dopušteno izvesti modulo p_i jer su i b_i i a_i veličine modulo p_i . Uz

$$b_i' = (b_i \cdot a_i) \bmod p_i \qquad (18)$$

jednadžba (16) se može napisati

$$B = \left(\sum_{i=1}^{N} b_i' \cdot r_i \right) \bmod R \qquad (19)$$

Pretpostavimo da su svi brojevi b_i' jednako dugački a ako nisu učinimo ih takvima dodajući im početne nule. Svaki broj b_i' moguće je prikazati kao

$$b_i' = \sum_{k=0}^{q} b_{ik}' \cdot 2^k \qquad b_{ik}' = 0,1 \qquad (20)$$

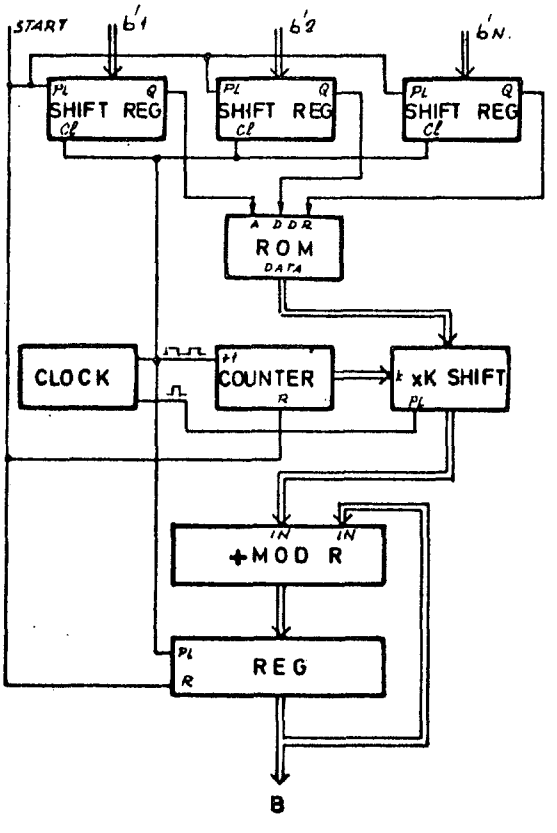
i

$$B = \left(\sum_{k=0}^{q} 2^k \sum_{i=1}^{N} b_{ik}' \cdot r_i \right) \bmod R \qquad (21)$$

Ista kombinacija brojeva $(b_{1k}', b_{2k}', .., b_{Nk}')$ doprinosi ukupnoj sumi jednaku veličinu ali pomnoženu sa 2^k. Radi veće brzine izvođenja algoritma prikladno je za svih 2^N mogućih kombinacija brojeva $(b_{1k}', b_{2k}', .., b_{Nk}')$ u tablici zapamtiti sume

$$\left(\sum_{i=1}^{N} b_{ik}' \cdot r_i \right) \bmod R \ .$$

Broj B se tada određuje u zbrajalu modulo R zbrajanjem q+1 pravilno šiftiranih parcijalnih suma iz tablice pohranjenje u ROM-u (slika 4).



SL.4.

Za dekodiranje broja iz sustava brojeva rezidua moguće je koristiti i algoritam promjenljive baze (engl. mixed-radix conversion). Broj B $0 \leq B \leq R-1$ može se prikazati sa

$$B = b_1 + b_{12}\cdot p_1 + b_{123}\cdot p_1\cdot p_2 + \\ + b_{12..N}\cdot p_1\cdot p_2\cdot\cdots\cdot p_{N-1} \qquad (22)$$

gdje za B $0 \leq B < R-1$ postoje brojevi $b_{12},\ldots, b_{12\,N}$ takvi da za sve k $1 \leq k \leq N$ vrijedi istovremeno

$$0 \leq b_{12\cdot k} \leq p_k - 1$$

Dokaz :

Ako je $b_{12.k} \geq p_k$ za $k < N$ on se može svesti u granice $0 \leq b_{12.k} \leq p_k - 1$ izračunavši ga modulo p_k a povečavši broj $b_{12.k+1}$ za broj

$$\left\lceil \frac{b_{12\cdot k}}{p_k} \right\rceil - 1$$

a da vrijednost prikazanog broja B ostane nepromijenjena. Ovaj postupak nije moguće izvesti jedino za $b_{12.N}$ ali ako je on veći ili jednak p_N tada je $B \geq R$ što je suprotno pretpostavci. Analogan postupak slijedi za $b_{12.k} < 0$.

Veličine $b_{12}, b_{123},\ldots, b_{12..N}$ se određuje iz prikaza broja B u sustavu brojeva rezidua $B = < b_1, b_2,\ldots, b_N >$ jednadžbama

$$b_{12} = \left(\frac{b_2 - b_1}{p_1 - p_2} \right) \bmod p_2$$

$$b_{123} = \left(\frac{b_{13} - b_{12}}{p_2 - p_3} \right) \bmod p_3$$

$$b_{1234} = \left(\frac{b_{124} - b_{123}}{p_3 - p_4} \right) \bmod p_4$$

$$\text{općenito } b_{12.ijk} = \left(\frac{b_{12.ik} - b_{12.ij}}{p_j - p_k} \right) \bmod p \qquad (23)$$

gdje je $N \geq k = j+1 = i+2 \geq 2$

Dokaz :

Prema (22) broj B se može prikazati kao

$$B = b_1 + .. + b_{12.ij}\cdot p_1\cdot p_2\cdot\cdots\cdot p_i + \\ + b_{12.ijk}\cdot p_1\cdot p_2\cdot\cdots\cdot p_i\cdot p_j + .. + \\ + b_{12.N}\cdot p_1\cdot p_2\cdot\cdots\cdot p_{N-1}$$

ali i kao

$$B = b_1 + .. + b_{12.ik}\cdot p_1\cdot p_2\cdot\cdots\cdot p_i + \\ + b_{12.ikj}\cdot p_1\cdot p_2\cdot\cdots\cdot p_i\cdot p_k + \\ + .. + b_{12.N}\cdot p_1\cdot p_2\cdot\cdots\cdot p_{N-1}$$

ako pretpostavimo da su p_j i p_k u skupu P zamijenili mjesta. Izjednačavanjem slijedi

$$b_{12.ij}\cdot p_1\cdot p_2\cdot\cdots\cdot p_i + b_{12.ijk}\cdot p_1\cdot p_2\cdot\cdots\cdot p_i\cdot p_j = \\ = b_{12.ik}\cdot p_1\cdot p_2\cdot\cdots\cdot p_i + b_{12.ikj}\cdot p_1\cdot p_2\cdot\cdots\cdot p_i\cdot p_k$$

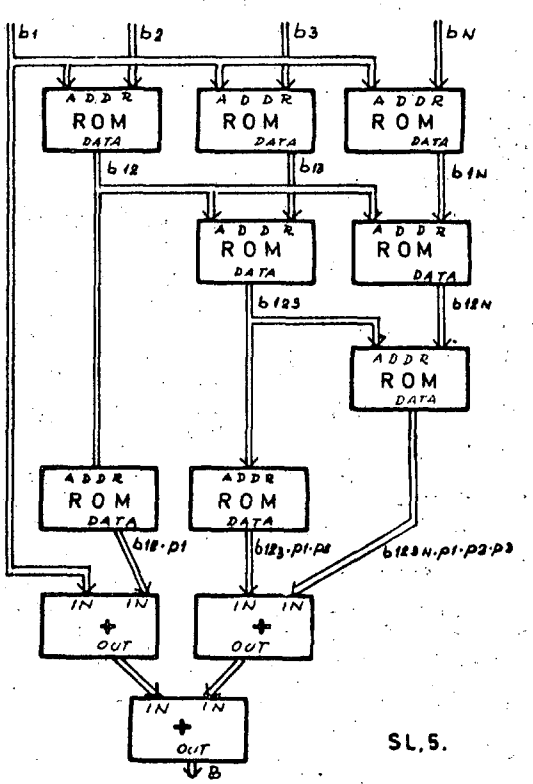
jer općenito vrijedi $b_{12.xy.z} = b_{12.yx.z}$

$$b_{12.ijk}\cdot p_j - b_{12.ikj}\cdot p_k = b_{12.ik} - b_{12.ij}$$

$$b_{12.ijk}\,(p_j - p_k) - b_{12.ikj}\cdot p_k = \\ b_{12.ik} - b_{12.ij}$$

$$b_{12.ijk} = \frac{b_{12.ik} - b_{12.ij} + b_{12.ikj}\cdot p_k}{p_j - p_k} \qquad (24)$$

Kako za svaki $b_{12.k}$ vrijedi $0 \leq b_{12.k} \leq p_k - 1$ jednadžbu (24) dozvoljeno je izračunati modulo p_k te slijedi (23).



SL.5.

Slika 5 prikazuje realizaciju dekodera metodom promjenljive baze sa ukupno

$$\frac{(N+2)\cdot(N-1)}{2} - 1$$

ROM-ova od kojih

$$\frac{N(N-1)}{2}$$

služe za dekodiranje brojeva $b_{12},..,b_{12..N}$ i trebaju imati svaki najviše po 2^{2q} riječi duljine q bita gdje je

$$q = \lceil \log_2 p_{max} \rceil$$

Ako se vrši dekodiranje broja koji može biti pozitivan ili negativan, a prikazan je u komplementarnom obliku broj B mora biti u granicama $[-w,w]$ gdje je

$$w = \frac{R-1}{2} \qquad (25)$$

Ako se dekodiranjem sa opisanim algoritmima dobije vrijednost B tada je

$$B = B^* \quad \text{ako je} \quad B^* \leq w$$

$$B = B^* + C_s \quad \text{ako je} \quad B^* \geq w \qquad (26)$$

gdje je $C_s = 2^s - R$ a s najmanji pozitivan cijeli broj takav da je $2^s \geq R$.

Dokaz :

Prvi dio tvrdnje (26) za pozitivne brojeve slijedi direktno iz opisanih algoritama.

U binarnom komplementarnom obliku se negativan broj B prikazuje sa vrijednošću

$$B = 2^s - |B|$$

gdje je s kako proizlazi iz (25) najmanji cijeli broj tako da vrijedi

$$2^s \geq 2w + 1$$

a w apsolutna vrijednost najvećeg broja koji se želi prikazati.

U sustavu brojeva rezidua negativan broj B prikazan je sa

$$\langle p_1-b_1, p_2-b_2, \ldots, p_N-b_N \rangle \quad \text{gdje je}$$

$$b_i = |B| \bmod p_i$$

Prema (16)

$$B^* = \left(\sum_{i=1}^{N} (p_i - b_i)\cdot a_i \cdot r_i\right) \bmod R$$

$$B^* = \left(\left(\sum_{i=1}^{N} p_i \cdot a_i \cdot r_i\right) \bmod R - \left(\sum_{i=1}^{N} b_i \cdot a_i \cdot r_i\right) \bmod R\right) \bmod R$$

Prva suma je jednaka nuli jer za svaki i vrijedi

$$(p_i \cdot r_i) \bmod R = \left(p_i \cdot \frac{R}{p_i}\right) \bmod R = 0$$

pa je

$$B^* = R - \left(\sum_{i=1}^{N} b_i \cdot a_i \cdot r_i\right) \bmod R = R - |B|$$

te uvrštavanjem slijedi $B = 2^s - R + B^*$

5. DIREKTNA A/D I D/A KONVERZIJA ZA SUSTAV BROJEVA REZIDUA

Direktnu konverziju analognog signala u digitalni broj prikazan u sustavu brojeva rezidua i obratno moguće je vršiti direktno primjenom metode promjenljive baze i posebno izgrađene D/A mreže u kojoj se veličine elemenata koji u kombinacijama određuju analogni signal međusobno odnose kao brojevi

$$1,2,4,..,2^{q_1-1},p_1,2\cdot p_1,..,2^{q_2-1}\cdot p_1 ,$$

$$p_1\cdot p_2,..,2^{q_N-1}\cdot p_1\cdot\ldots\cdot p_{N-1}$$

gdje su sa $q_1,q_2,..,q_N$ označeni najmanji cijeli brojevi tako da vrijedi

$$2^{q_1} \geq p_i$$

Nakon izvršene A/D konverzije sukcesivnom aproksimacijom uz korištenje definirane mreže dobiva se binarni broj dužine Q bita

$$Q = q_1 + q_2 + .. + q_N \qquad (27)$$

čijih q_1 najmanje značajnih bita odgovara veličini b_1, slijedećih q_2 odgovara veličini b_{12} i tako redom do veličine $b_{123..N}$. Iz tih veličina veličine $b_1,b_2,..,b_N$ koje pretstavljaju digitalni prikaz analogne veličine u sustavu brojeva rezidua dobiju se jednadžbama

$$b_2 = (b_1 + b_{12}\cdot p_1) \bmod p_2$$

$$b_3 = (b_1 + b_{12}\cdot p_1 + b_{123}\cdot p_1\cdot p_2) \bmod p_3$$

$$b_i = (b_1 + b_{12}\cdot p_1 + b_{123}\cdot p_1\cdot p_2 + .. + b_{12..i-1}\cdot p_1\cdot\ldots\cdot p_{i-1}) \bmod p_i \qquad (28)$$

za $2 \leq i \leq N$.

Dokaz :

$$b_1 = B \bmod p_1$$

$$b_i = (b_1 + b_{12}\cdot p_1 + b_{123}\cdot p_1\cdot p_2 + \dots + b_{12..N}\cdot p_1\cdot\dots\cdot p_{N-1}) \bmod p_i$$

Svi pribrojnici u toj jednadžbi koji sadrže p_i kao faktor su jednaki 0 modulo p_i te slijedi jednadžba (28) .

Za broj Q vrijedi

$$q_R \leq Q \leq q_R + N \qquad (29)$$

gdje je q_R najmanji cijeli broj tako da vrijedi

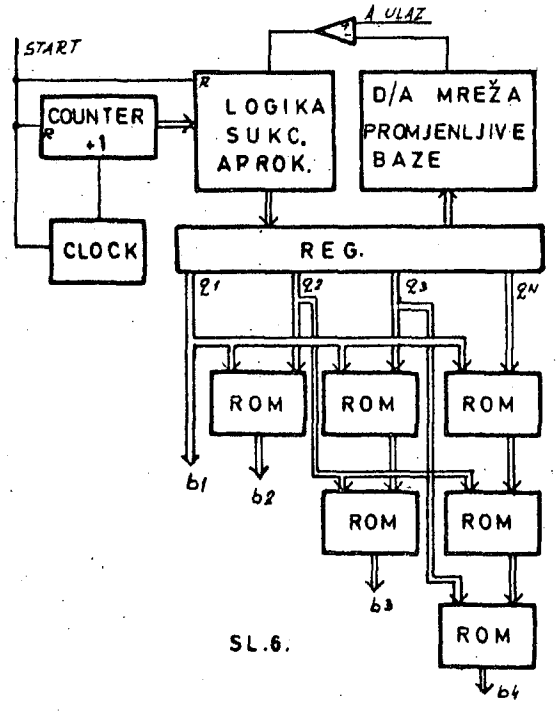
$$2^{q_R} \geq R \ .$$

Iz jednadžbe (29) se vidi da je za sukcesivnu aproksimaciju upotrebom D/A mreže u sustavu promjenljive baze općenito potrebno više stupnjeva aproksimacije nego upotrebom standardne binarne mreže i da veličina razlike ovisi o broju rezidua kojima se prikazuje broj.

Sve operacije unutar zagrade u jednadžbi (28) dozvoljeno je izvesti modulo p_i

$$b_i = (b_1 + b_{12}\cdot p_1 \bmod p_i + b_{123}\cdot(p_1\cdot p_2)\bmod p_i + \dots + b_{12..i}\cdot(p_1\cdot p_2\cdot\dots\cdot p_{i-1})\bmod p_i \qquad (30)$$

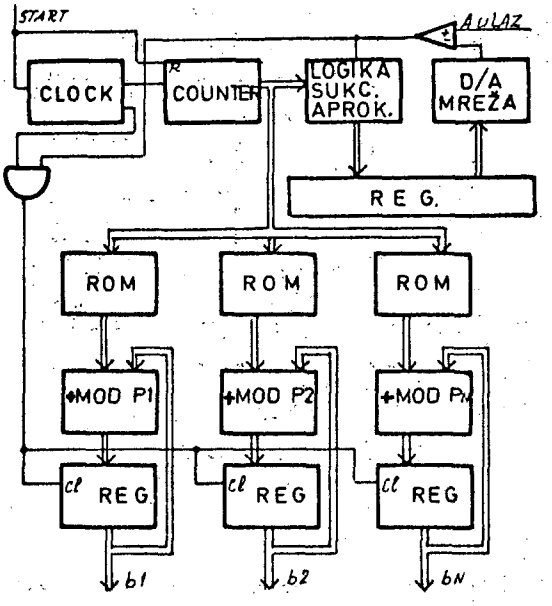
te slijedi realizacija prikazana slikom 6.



SL.6.

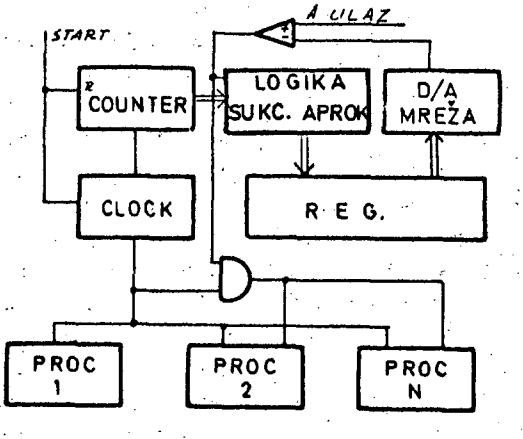
U slučaju velikog N potreban je velik broj ROM-ova i primjenljivost ove metode je ograničena na programske realizacije. Prikladnim izborom veličina $p_1, p_2, \dots, p_N$ tako da veličine $p_1 \bmod p_2$, $(p_1\cdot p_2)\bmod p_3$, do $(p_1\cdot p_2\cdot\dots\cdot p_{N-1}) \bmod p_N$ budu mali brojevi moguće je u programskoj realizaciji množenje sa njima svesti na jednostavne logičke operacije.

U slučaju A/D konverzije standardnom binarnom mrežom, konverziju digitalnog broja u sustav brojeva rezidua moguće je izdvojiti istovremeno sa A/D konverzijom i tako povećati brzinu konverzije a smanjiti broj potrebnih sklopova (sl. 7).



SL.7.

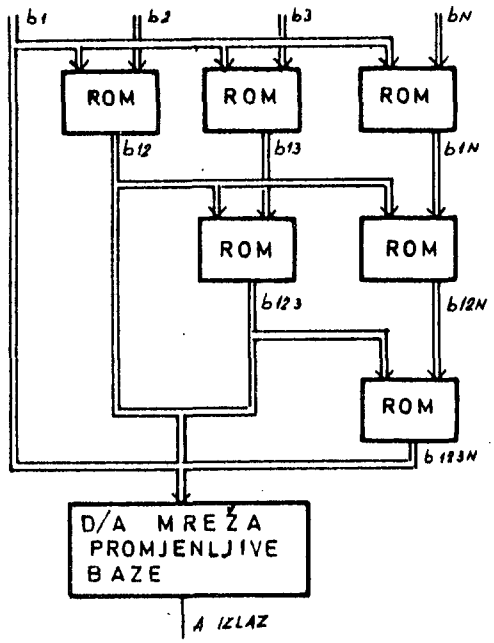
Ukoliko konverziju u sustav brojeva rezidua obavlja računalo moguće je i čitavu digitalnu logiku sukcesivne aproksimacije izvesti programski.



SL.8.

Slika 8 prikazuje način izvođenja direktne A/D konverzije u višeprocesorskom sistemu u kojem svaki od procesora vrši konverziju i računanje sa jednim od rezidua.

Primjena D/A mreže za sustav promjenljive baze od posebne je koristi u direktnoj D/A konverziji iz sustava brojeva rezidua u analognu veličinu primjenom algoritma promjenljive baze (sl. 9).



SL.9.

Primjenom specijalne D/A mreže povećana je brzina konverzije i smanjen broj potrebnih elemenata. I ovaj način konverzije pogodniji je za primjenu u računarskim sistemima nego za sklopovsku realizaciju zbog potrebe za većim brojem tablica.

ZAKLJUČAK

Mogućnost dekompozicije aritmetičkih operacija je od prvorazredne važnosti u brzoj obradi digitalnih signala te iz toga proizlazi važnost primjene sustava brojeva rezidua. Pri njegovoj primjeni javlja se potreba za konverzijom binarnog broja u sustav brojeva rezidua i obratno, što je u radu posebno detaljno analizirano. Prikazani algoritmi pretstavljaju osnovu sklopovskih i programskih rješenja. U radu nisu analizirane aritmetičke operacije među reziduima ali vrijedi da je zbrajanje, oduzimanje i množenje vrlo lako izvedivo posebno računalima a da su operacije dijeljenja, uspoređivanja i otkrivanja prelјeva vrlo teško izvedive unutar samog sustava te da to pretstavlja glavno ograničenje područja primjene sustava brojeva rezidua.

LITERATURA :

1. W. Jenkins : A Highly Efficient Residue Combinatorial Architecture for Digital Filters, Proceedings of the IEEE, Vol. 66, No 6, June 1978, pp700-702.

2. W. Jenkins, B. Leon : The Use of Residue Number System in the Design of Finite Impulse Response Digital Filters, IEEE Trans. of Circuits and Systems, Vol. CAS 24, April 1977, pp 191-200.

3. F. Taylor : A VLSI Residue Arithmetic Multiplier, IEEE Trans. on Computers, Vol. C 31, 1982, pp 540-546.

4. M. Sonderstrand, C. Vernia : A High-Speed Low-Cost Modulo P_i Multiplier with RNS Arithmetic Applications, Proceedings of the IEEE, Vol. 68, No 4, April 1980, pp 529-532.

5. E. R. Berlekamp : Algebraic Coding Theory, McGraw-Hill, 1968.

6. F. Taylor, C. Huang : An Autoscale Residue Multiplier, IEEE Trans. on Computers, Vol. C-31, No 4, April 1982, pp 321-325.

PROGRAMSKI SIMULATOR – KORAK K RAZVOJU MIKRORAČUNALNIŠKIH APLIKACIJSKIH PROGRAMOV

D. MILJAN,
J. ŠILC,
P. KOLBEZEN

UDK: 681.3.06:519.682

INSTITUT „JOŽEF STEFAN", LJUBLJANA

Analiza cene razvoja računalniškega sistema kaže, da večji del cene odpade na razvoj programske podpore, zato je izredno pomembno, da ima razvijalec na voljo učinkovite pripomočke za razvoj programske podpore. V članku so nakazani nekateri primeri tovrstnih pripomočkov. Natančneje je opisan pri nas razvit programski simulator za razvoj aplikacij z domačim integriranim mikroračunalnikom Iskra EMZ 1001.

SOFTWARE SIMULATOR - STEP IN DEVELOPMENT OF MICROCOMPUTER APPLICATION PROGRAM. Trends in computer system development costs show quite clearly that the major proportion of these costs must be attributed to software design and implementation. Some microcomputer software development aids available to programmer are introduced. Software is required both to execute a particular application on a system and to assist in the development of new software. Example of software simulator for Iskra EMZ 1001 single chip microcomputer is given.

UVOD

Računalniške programe smo običajno razvijali bodisi na računalniku za katerega smo program pisali, ali na njemu podobnem, če je bila njegova konfiguracija takšna, da je omogočala uporabo potrebne opreme (programske in materialne) za razvoj aplikacijskih programov. Pod razvojno programsko opremo mislimo: urejevalec teksta (editor), ustrezen prevajalnik in povezovalnik (linker). Pri razvojni materialni opremi je osnovna zahteva dovolj velik pomnilnik in možnost komunikacije človek-stroj s pomočjo CRT-ja ali teleprinterja.

Pri 'večjih' računalnikih običajno ni tovrstnih problemov, pri mikroračunalnikih pa smo pogosto soočeni s pomankanjem pomnilniškega prostora in revnejšo periferno opremo za razvoj aplikacijskih programov. Ti problemi postanejo akutni pri integriranih mikroračunalnikih (single chip computer), pri katerih rešujemo problem razvoja aplikacijskih programov s pomočjo t.i. razvojnih sistemov. V tem prispevku se bomo omejili na področje mikroračunalnikov. Razlike v načinu razvijanja aplikativnih programov pri mikroračunalnikih glede na 'večje' računalnike izhajajo iz naslednjih ugotovitev :

- mikroprocesorje velikokrat uporabljamo za posebne namene v sklopu računalnikov na eni tiskani plošči (single board computer) ali v sklopu posebno načrtovanih vezij za posamezne aplikacije. Takšne minimalne računalniške konfiguracije običajno niso primerne za razvoj aplikativnih programov,
- načrtovalec mikroračunalniških sistemov lahko v odvisnosti od tipa aplikacije izbira med različnimi tipi mikroprocesorjev. Možnost, da bi za vsak od izbranih mikroprocesorjev imel svoj razvojni sistem z vso programsko in materialno podporo za razvoj aplikativnih programov, izključujejo ekonomski razlogi.

Kot je omenjeno, poznamo več tipov razvojne opreme in postopkov za razvoj aplikativnih programov za mikroračunalnike. Nekatere rešitve nudijo proizvajalci računalniške opreme, nekatere pa razvijajo uporabniki mikroprocesorjev sami.

Razvojno opremo in postopke lahko razdelimo v nekaj kategorij :

i. Mikroračunalnik z osnovnim monitorjem v ROM pomnilniku.

Takšen sistem daje uporabniku minimalne možnosti za vpisovanje in testiranje aplikacijskega programa preko terminala. Takšen monitor je lahko koristen pri razvoju in testiranju manjših programov (do največ 100 računalniških besed) in je praktično neuporaben pri večjih aplikacijah. Kljub temu, takšno opremo ne smemo kar tako izključiti kot eno od možnih razvojnih pripomočkov že zaradi zelo široke uporabe mikroprocesorjev, pri kateri včasih rešujemo programske probleme tudi s takšnim monitorjem.

ii. Mikroračunalnik opremljen z zbirnikom in urejevalcem teksta.

Programa lahko stalno zasedata del ROM pomnilnika mikroračunalnika ali se po potrebi prečitata iz papirnega traku v RAM pomnilnik. Tak sistem omogoča pisanje in popravljanje aplikativnih programov v zbirnem jeziku in shranjevanje le-teh na papirni trak. Čeprav je to korak naprej od samega monitorja, še zdaleč ni omogočeno učinkovito aplikativno razvojno delo večjih dimenzij. Dodatno nerodnost predstavlja prepisovanje zbirnika in urejevalca texta iz papirnega traku v RAM pomnilnik

mikroračunalnika. Manjšo, a ne tako bistveno izboljšavo tega razvojnega sistema predstavlja nadomestitev sistema s papirnim trakom z sistemom z digitalno kaseto.

iii. Razvojni sistem na bazi uporabljenega mikroprocesorja.

Tak sistem, opremljen z gibkimi diski, omogoča pisanje, popravljanje in urejevanje aplikacijskega programa. Nadalje omogoča še zbiranje in prevajanje ter shranjevanje izvirnih in prevedenih programov. Takšen sistem predstavlja že relativno močno orodje pri razvoju aplikacij, ima pa eno večjo pomanjkljivost (gledano s stališča uporabnika) in to je, vezanost uporabnika na en tip ali eno družino mikroprocesorjev, za katero je razvojni sistem zgrajen. Pri uporabnikih, ki želijo (ali so prisiljeni) izbirati med različnimi mikroprocesorji za različne aplikacije, takšen razvojni sistem ne more biti trajna rešitev.

iv. Razvojni sistem na bazi poljubnega mikroprocesorja.

Takšen sistem (z gibkimi diski) vsebuje poleg urejevalca teksta običajno še nekaj različnih zbirnikov ali prevajalnikov za posamezne tipe mikroprocesorjev katere podpira. Omogočene so vse faze razvoja aplikacijskih programov do faze testiranja. Za testiranje je potrebno objektno kodo aplikacijskega programa prenesti na realno mikroračunalniško aplikacijo. Ta problem lahko rešujemo z dodatno opremo (programsko in materialno), ki je različna in posebna za posamezne mikroprocesorje. To so t.i. emulatorji. Emulator se priključi na razvojni sistem ter omogoča vpis, testiranje in izvajanje aplikativnega programa na aplikativnem sistemu. Čeprav nas takšen razvojni sistem ne omejuje (znotraj končne množice mikroprocesorjev, ki jih podpira) pri izbiri mikroprocesorja in je v veliko primerih zelo dobra rešitev, je za testiranje programov potrebna dodatna nabava že omenjenih emulatorjev. Ti stroški (ki so lahko pomembni) se bodo ponovili pri vsaki aplikaciji z novim tipom mikroprocesorja.

v. Razvojni sistem na 'večjem' računalniku.

Večji računalnik, ki je poleg standardne opreme ustrezno opremljen z križnimi zbirniki, prevajalniki in simulatorji za različne mikroprocesorje je zelo dobra rešitev za tiste uporabnike, ki v svoji opremi že imajo večji računalnik ali lahko razpolagajo z delom kapacitet njim dosegljivega večjega računalnika (time sharing terminal,..). Pri tem odpadejo vsa večja začetna vlaganja v razvojne sisteme. Tudi pri teh sistemih lahko omenimo nekaj problemov, ki se pojavljajo . Prvi problem je pomanjkanje t.i. križnih programskih pripomočkov. Proizvajalci mikroprocesorjev namreč močneje podpirajo razvoj lastnih razvojnih sistemov za lastne (in sorodne) družine mikroprocesorjev v oblikah, ki so opisane v prejšnjih odstavkih in manj podpirajo razvoj križnih programov za večje računalnike. Za takšne programe se bolj zavzemajo proizvajalci mikroprocesorjev, ki ne ponujajo lastne razvojne sisteme ker na ta način širijo uporabnost svojih proizvodov. Iz tega izvira drugi problem in ta je, velika raznolikost v načinu in tehniki uporabe križnih programov, ki so izdelani pri različnih proizvajalcih.

Posebni problemi nastanejo, ko se uporabnik odloči za uporabo (razlogi so lahko tehnične lastnosti, dosegljivost, cena,..) mikroprocesorja ali mikroračunalnika za katerega proizvajalec ponuja zelo skromno aplikativno razvojno opremo, ali pa le-te sploh ne nudi. V tem primeru preostane samo še razvoj lastnih pripomočkov. Odločimo se lahko za eno od prej opisanih kategorij razvojnega sistema. Odločitev bo odvisna od opreme s katero razpolagamo, časa, ki je na voljo za razvoj nove opreme ter od števila razvijalcev.

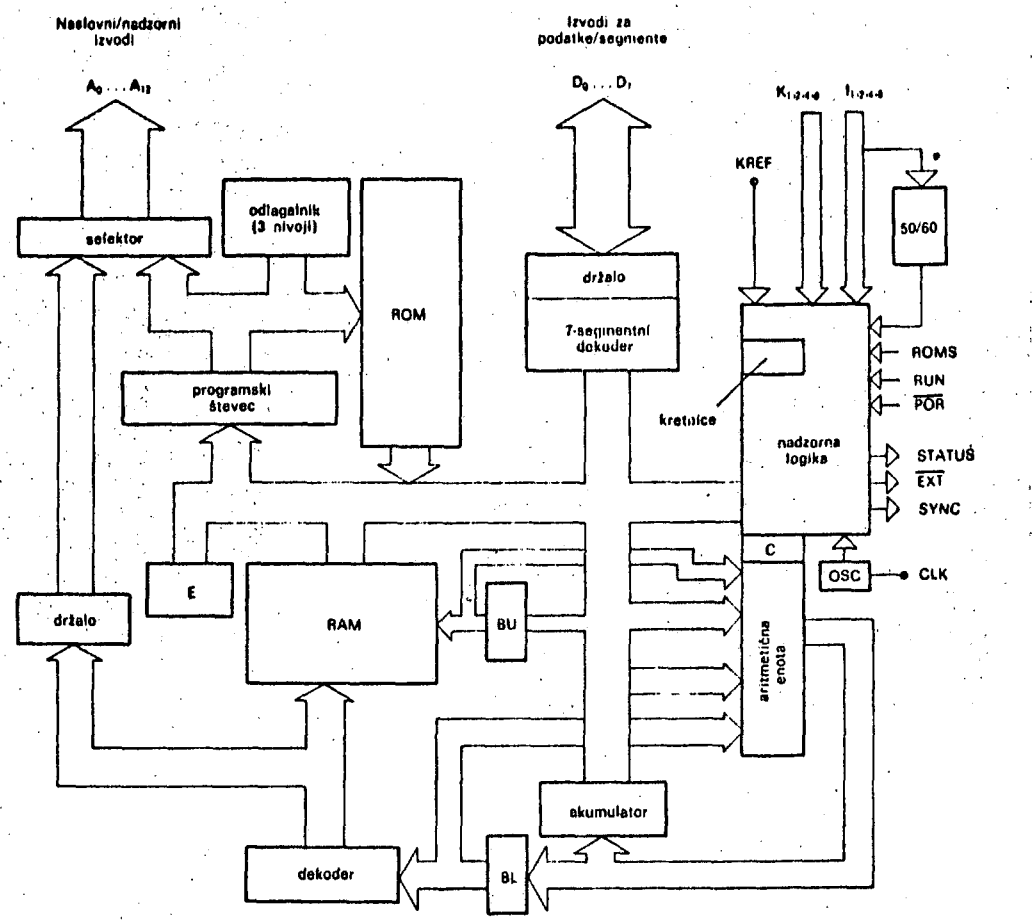
V tem prispevku bomo prikazali primer, ko se uporabnik (razvijalec aplikativnih mikroračunalniških sistemov) odloči za razvoj mikroračunalniškega sistema pri katerem je osnovni element sistema domači integrirani mikroračunalnik Iskra EMZ 1001 za katerega proizvajalec ne ponuja zadosti bogato aplikativno razvojno opremo.

INTEGRIRANI MIKRORAČUNALNIK Iskra EMZ 1001

Mikroračunalnik Iskra EMZ 1001 [4],[5] z veliko gostoto integriranih elementov prinaša vse prednosti krmiljenja z mikroračunalnikom pri minimalnih stroških opreme. Obdelava podatkov v mikroračunalniku EMZ 1001 poteka preko 4 bitnih besed. Nadzorni del in vhodno-izhodna konstrukcija sta 8 bitno zasnovana. Povezava z zunanjim svetom poteka preko več specijaliziranih vodil. Vodilo A ima 13 bitov, ki služijo samo kot izhodi. Vodilo D je 8 bitno in prenaša podatke v dveh smereh, lahko pa je tudi v t.i. nevtralnem stanju. Vodili I in K sta 4 bitni in služita samo kot vhoda. Pomnilnik RAM je razdeljen na 4 strani, od katerih ima vsaka 16 4-bitnih besed. Pomnilnik RAM se naslavlja z registri BL in BU. Naslovni prostor programskega ROM pomnilnika je razdeljen na 8 pomnilniških bank. Vsaka banka vsebuje 1K 8-bitnih besed. Banka '0' je tisti del pomnilnika, ki je vključen na integrirani ploščici, ostale banke pa po želji dodajamo od zunaj. Omogočena je izbira aktivnega ROM pomnilnika (samo notranji, samo zunanji ali notranji in zunanji pomnilnik obenem) ter izbira načina delovanja (statični, multiplexni in testni). Na osnovi prikazanih lastnosti mikroračunalnika EMZ 1001 (slika 1) se lahko prepričamo, da je ta uporabljiv v številnih manjših aplikacijah. Najbolj smiselne so rešitve v katerih zadošča število razpoložljivih vodil za neposredno krmiljenje sistemske okolice.

STANDARDNA APLIKACIJSKA RAZVOJNA OPREMA

Iskra Mikroelektronika, proizvajalec mikroračunalnika EMZ 1001, ponuja kot pomoč pri razvoju aplikacij s tem integriranim mikroračunalnikom t.i. statični emulator. Statični emulator izkorišča možnost multipleksnega delovanja mikroračunalnika EMZ 1001 in služi za preverjanje in izvajanje aplikativnih programov tako, kot da bi bil program že zapisan v notranjem ROM pomnilniku. Emulator vsebuje mikroračunalnik EMZ 1001, PROM pomnilnik z uporabnikovim programom, pomnilni vmesnik in 40-žilni kabel s priključkom v obliki mikroračunalnika. Za preverjanje programa spojimo priključek emulatorja s podnožjem, v katerem bo po končanem razvoju mikroračunalnik EMZ 1001 z notranjim uporabniškim programom.



Slika 1.

Poleg statičnega emulatorja, kot samostojne enote in z možnostjo priključka na mikroračunalnik Iskra Data, ponuja Iskra Mikroelektronika, na osnovi primerno podanih specifikacij pogodbeni prevzem izdelave programov in prototipov za posamezne uporabnike.

LASTNI RAZVOJNI SISTEM

Za večje število obsežnejših aplikacij s tem mikroračunalnikom potrebuje uporabnik več programske in materialne razvojne opreme. Narekuje se razvoj lastnega razvojnega sistema. Ta naj bi čim več pripomogel k učinkovitemu in hitremu razvoju aplikacij pri čim boljši uporabi že obstoječe opreme in z minimalnimi zaščetnimi vlaganji.

Slika 2. prikazuje primer takšnega razvojnega sistema pri katerem smo uporabili že obstoječi računalnik LSI 11/23 in mikroračunalnik na bazi mikroprocesorja Intel 8080. Postopek razvoja aplikativnih programov je pri tem sistemu razdeljen na dve etapi :

- Delo na računalniku LSI 11/23, ki je poleg standardne opreme (učinkovit urejevalec teksta, prevajalniki, gibki diski, winchester disk, tiskalnik, CRT-prikazovalnik,..) opremljen še s križnim zbirnikom za zbirni jezik [2] in s programskim simulatorjem mikroračunalnika EMZ 1001. V tej etapi se aplikativni program piše, testira in popravlja, dokumentira in shranjuje. Po končani prvi fazi, se lahko pri predpostavki, da je bil problem dovolj dobro definiran, razvoj aplikativnega programa tudi konča.

- Objektni kod aplikacijskega programa, ki je dobljen na računalniku, lahko dodatno testiramo na realnem okolju s pomočjo emulatorja [1] ter ga dodatno prilagodimo morebitnim materijalnim (hardwarskim) posebnostim aplikacije.

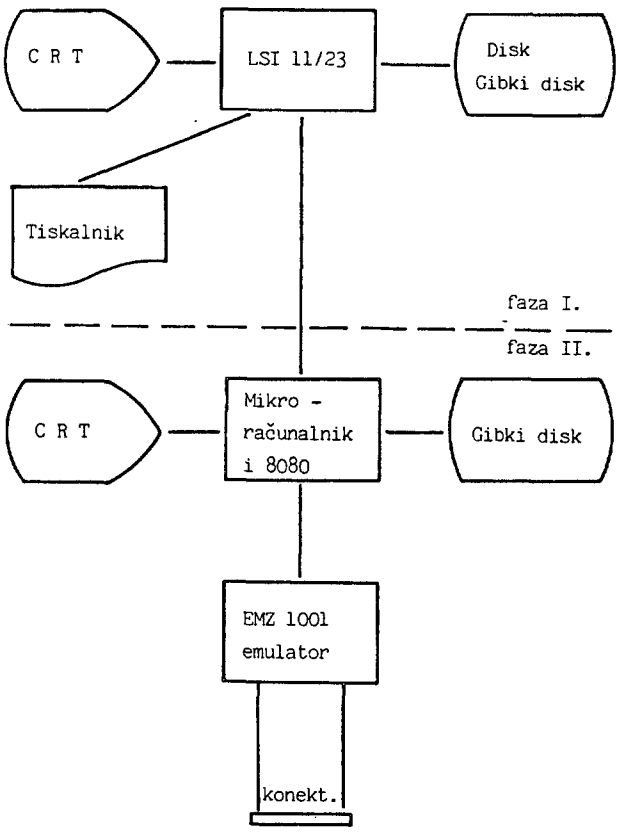
Omenjeni križni zbirnik za mikroračunalnik EMZ 1001 in emulator sta že predstavljena v IJS Delovnih poročilih [1] in [2]. V tem prispevku bomo predstavili še tretji razvojni pripomoček - programski simulator mikroračunalnika EMZ 1001.

PROGRAMSKI SIMULATOR

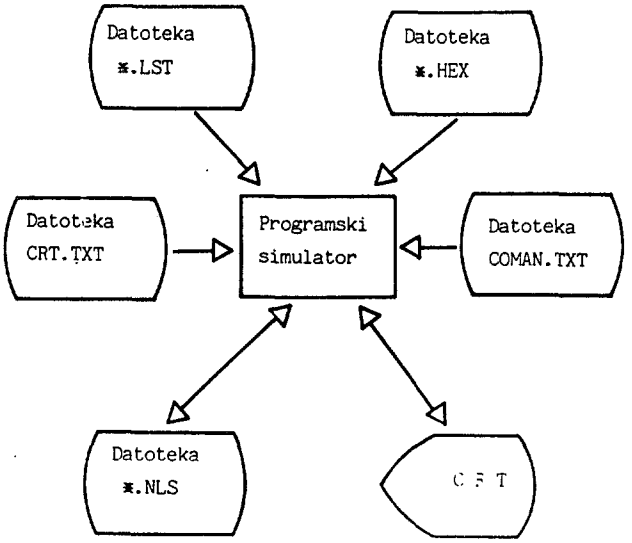
Programski simulator je programski paket, ki, v našem primeru, simulira delovanje mikroračunalnika in pri tem omogoča *uporabniku :*

- vpogled v notranjo strukturo registrov in pomnilnika mikroračunalnika pri vsakem koraku izvajanja aplikativnega programa,
- nadzor nad pretokom vhodno/izhodnih podatkov,
- prekinitev izvajanja pri poljubnih stanjih podatkovnega in adresnega vodila,
- koračno izvajanje programa
- izvajanje posameznih delov (modulov) aplikativnega programa,
- merjanje računalniškega časa,
- diagnosticiranje nekaterih napak.

Programski simulatorji so lahko napisani v kakšnem od 'splošnih' programskih jezikov, ali v posebnih simulacijskih jezikih in po za simulatorje posebno zgrajenih metodah (ISPS language [10], MicroSim [8]). Takšni posebni jeziki in sistemi omogočajo simulacijo mikroračunalniških sistemov in v nekaterih primerih tudi simulacijo multi mikroračunalniških sistemov. Simulator mikroračunalnika EMZ 1001 je napisan v programskem jeziku Pascal, teče na računalniku LSI 11/23 in je namenjen samo simulaciji izvajanja instrukcij tega mikroračunalnika.



Slika 2.

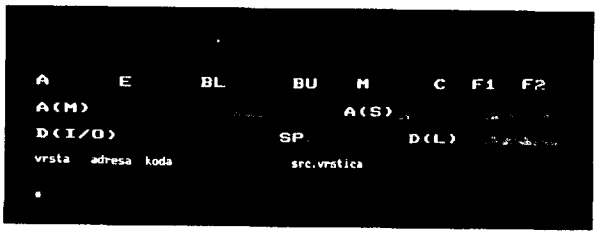


Slika 3.

Vhodni datoteki za 'programski simulator (slika 3.) sta datoteki (*.LST in *.HEX), ki jih je zgradil križni zbirnik. Pri inicijalizaciji bo simulator detoteko *.LST preuredil (odstranjene so prazne in komentarske vrstice) v datoteko z imenom *.NLS.

Datoteka CRT.TXT vsebuje podatke za izpis slike (monitorja) na zgornji polovici CRT-prikazovalnika, kot je razvidno iz slike 4. Monitor bo v času emuliranja stalno na prikazovalniku. Pri vsakem koraku simuliranja programa se vrednosti v okvirjih korigirajo, kar omogoča uporabniku stalen pregled nad vsebinami registrov, vhodno/izhodnimi vodili in računalniškim časom.

Za učinkovito uporabo simulatorja je predpisan enostaven vhodni jezik s katerim uporabnik nadzoruje delovanje simulatorja, vpisuje prekinitvene točke, kontrolira izpisovanje na CRT in podobno. V Tabeli 1. je prikazan nabor ukazov vhodnega jezika s krajšimi pojasnili pomenov le-teh. Podoben tekst kot v Tabeli 1. vsebuje datoteka COMAN.TXT, ki jo uporabnik dobi izpisano na CRT ko uporabi ukaz 'HELP



Slika 4.

Ukazi EMZ-simulatorja :

a) Kontrola delovanja mR EMZ :

HA [LT]
ustavitev delovanja mR

RU [N] [aaaaa]
izvajanje programa pri osmiški adresi 'aaaaa', če adresa ni podana, se začne izvajanje pri adresi 0

RE [SE]
reset mR (inicijalizacija)

b) Testne prekinitve izvajanja :

BAA aaaaa
T.p. pri osmiški adresi 'aaaaa'

BAD aaaaa
T.p. pri osmiškem podatku 'aaaaa' na A-linijah

BDI ddd[O][D][H]
T.p. pri instrukciji 'ddd'

BDD ddd[O][D][H]
T.p. pri V/I podatku 'ddd'

BDL ddd[O][D][H]
T.p. pri zadržanem podatku 'ddd'

S [TEP] [ddd[O][D][H]]
T.p. po 'ddd' korakih (instrukcijah), če število korakov ni podano, izvede 1 korak, nadaljnje enokoračno izvajanje sproži tipka 'presledek'

NS
prepoved koračnega izvajanja programa

c) Kontrola izpisa na CRT :

TR
izpisovanje tekočih vrstic programa

NT
brez ispisovanja tekočih vrstic programa

M [EMO] [ddd[O][D][H]]
izpis 'ddd' vrstice ali celotne vsebine notranjega RAM-pomnilnika mR

TI
resetiranje števca časa

d) Pomožni ukazi :

RA [M]
vpis novih vrednosti v notranji RAM-pomnilnik mR

NE [W]
vpis novih vrednosti v notranje registre mR

CO [MA]
izpis vseh trenutno aktivnih ukazov

CL [EA]
briše vse ukaze (začetno stanje simulatorja)

HE [LP]
izpis liste vseh ukazov za EMZ-simulator

O [UT]
izhod iz EMZ-simulatorja

Tabela 1.

LASTNOSTI PROGRAMSKEGA SIMULATORJA MIKRORAČUNALNIKA EMZ 1001

1. Pregled nad notranjimi registri

Kot je razvidno iz slike 4, so na CRT prikazovalniku stalno (v binarnem zapisu) prikazane vrednosti vseh notranjih registrov in sicer :

- akumulatorja (A)
- splošnega registra (E)
- naslovnih registrov (BL) in (BU)
- kretnic (flags) (C),(F1) in (F2)
- trenutno naslovljene RAM besede (M)
- globine vgnezdenosti podprogramov (stack pointer) (SP)

Nadalje lahko na monitorju opazujemo stanje podatkovnih vodil:

- Vodilo A.
  Prikazani sta vrednosti v t.i. 'slave' (A(S)) in 'master' (A(M)) 13-bitnih registrih. Pomožni (slave) register služi za pripravo izhodnih podatkov, glavni (master) register pa zadržuje podatke, ki so prisotni na zunanjem A-vodilu.

- Vodilo D.
  Prikazani sta vrednosti podatkovnega vodila pri prehodu vhodnih in izhodnih podatkov med mikroračunalnikom in okolico (D(I/O)) ter vrednost zadržanih (latched) podatkov na D-vodilu (D(L)).

Poleg pregledovanja stanja registrov in vodil, lahko s pomočjo ukaza 'NEW' vsem prikazanim elementom priredimo poljubno novo vrednost. Ta možnost se bo pokazala za zelo uporabno pri pripravi začetnih pogojev za izvajanje posameznih podprogramov.

2. Pregled nad RAM pomnilnikom

Omenjeno je že, da se vrednost trenutno naslovljene besede RAM pomnilnika stalno prikazuje na 'monitorju'. Pregled celotnega ali poljubnega dela RAM pomnilnika je omogočen z ukazom 'MEMO'. Primer izpisa vseh besed RAM pomnilnika je prikazan na sliki 5. Tako kot pri registrih, uporabnik lahko tudi v RAM pomnilniku spreminja vrednosti na posameznih lokacijah s pomočjo ukaza 'RAM'.

3. Nadzor nad vhodnimi podatki

Pri programskem simuliranju ni omogočen priključek posebnih zunanjih vhodno/izhodnih enot, kot je načrtano za realen sistem, zato bo simulator, ko pričakuje vhoden podatek prepustil uporabniku odločitev o vrednosti le-tega. Takšni primeri se pojavijo, ko se simulira EMZ instrukcija 'INP' in ko simulirani program pregleduje K in I vhode mikroračunalnika. Možnost podajanja poljubnih vhodnih stanj, omogoča uporabniku učinkovitejše testiranje tistega dela aplikacijskega programa, ki obdeluje vhodne podatke.

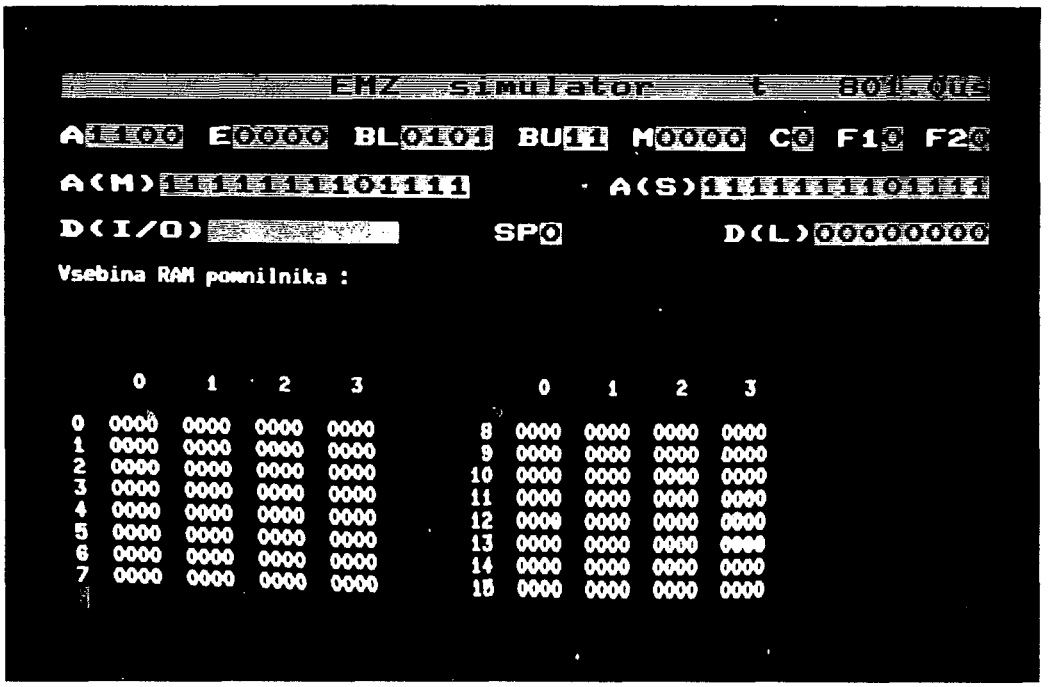
4. Testne prekinitve

Iz Tabele 1. je razvidno, da je testna prekinitev omogočena pri poljubni vrednosti na A in D vodilih, in sicer :

- pri instrukciji na poljubni adresi (ukaz 'BAA'),
- pri poljubnem podatku na zunanjem A-vodilu (ukaz 'BAD'),
- pri poljubni instrukciji (ukaz 'BDI'),
- pri poljubnem vhodno/izhodnem podatku (ukaz 'BDD'),
- pri poljubni vrednosti zadržanega podatka na D-vodilu (ukaz 'BDL').

6. Računalniški čas

Na CRT prikazovalniku v t.i. monitorju poleg zgoraj opisanih podatkov uporabnik lahko stalno opazuje čas (v μ sek.) izvajanja simuliranega programa (slike 4,5,6). Računalniški čas se resetira z ukazom 'TZ'. Na ta način lahko uporabnik dobi podatke o času izvajanja celotnega ali posameznih delov (podprogramov) simuliranega programa, poišče najkrajši in najdaljši čas izvajanja posameznih ciklov znotraj programa in podobno.



Slika 5.

Pri vsaki testni prekinitvi se na CRT-ju izpiše vrstica izvirnega programa (source line) pri kateri se je zgodila prekinitev. Na ta način se potek simulacije programa lažje spremlja tudi na 'listingu' aplikacijskega programa. Posebno kritične dele programa lahko uporabnik simulira po korakih s pomočjo ukaza 'STEP'. Isti ukaz omogoča testno prekinitev po poljubnem številu izvedenih instrukcij simuliranega programa.

5. Simulacija programskih modulov

Ukaz 'RUN' (Tabela 1.) omogoča nastavitev poljubne vrednosti programskega števca (znotraj naslovnega polja ROM pomnilnika) kar omogoča simuliranje posameznih delov simuliranega programa. V primeru simuliranja posameznega programskega modula (podprograma) je običajno potrebno nastaviti nekatere začetne pogoje kot so: začetne vrednosti v določenih registrih in na določenih lokacijah RAM pomnilnika, vrednost števca globine vgnezdenosti modula in računalniški čas. Na ta način lahko opazujemo (testiramo) delovanje posameznih modulov aplikacijskega programa, ki jih v končni fazi razvoja združimo v skupni program.

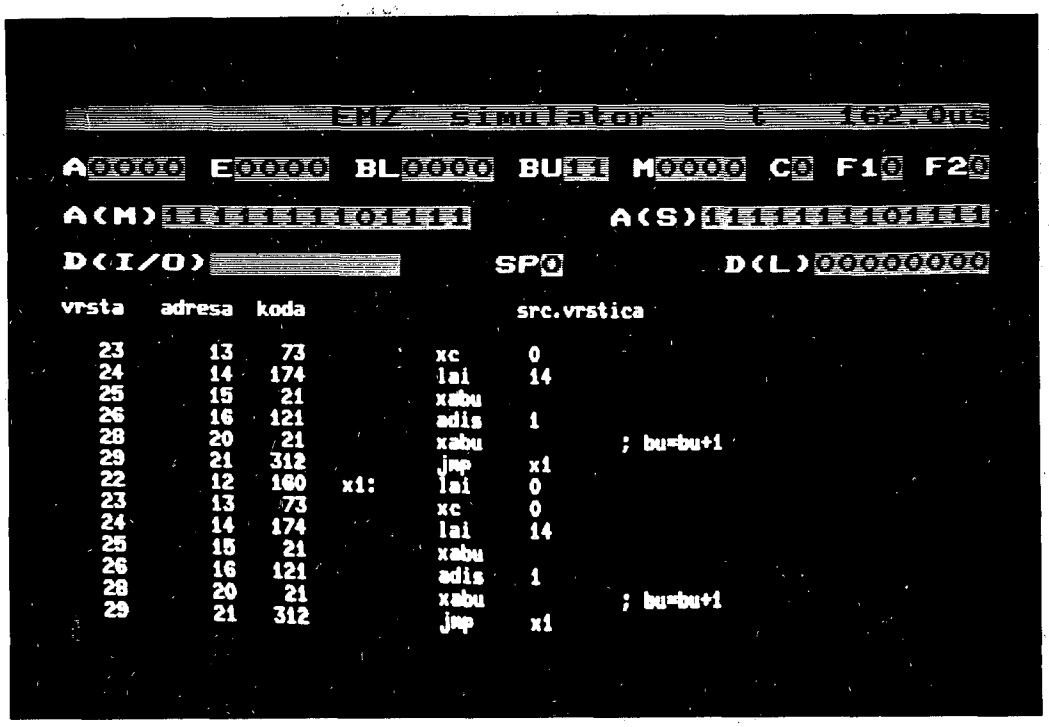
7. Diagnostika

Simulator opozarja uporabnika (z izpisom na CRT) na odvečne 'PP'- instrukcije in na preskakovanje več, v zaporedju programiranih 'LAI' in 'LB_' instrukcij. Simulacija se ustavi, če se program začne z 'LAI' ali 'LB' instrukcijo, če pride do prekoračitve globine vgnezdenja podprogramov (največja globina je 3) in če je prekoračeno naslovno področje ROM pomnilnika.

Simulacija se ustavi tudi, če pri simuliranju 'INP' instrukcije D-vodilo ni bilo predhodno nastavljeno v nevtralno (Tri-state) stanje in če pri odčitavanju I ali K vhodov le-ti niso bili predhodno pravilno naslovljeni.

Pri vseh opozorilih in vstavitvah se na CRT izpiše kratek opis napake in trenutna vrednost programskega števca, pri kateri je ugotovljena napaka.

Poleg opisanih možnosti simulatorja, zaradi lažjega sledenje simulacije, lahko uporabnik z ukazi 'TR' in 'NT' (trace / notrace) kontrolira izpisovanje vrstic simuliranega programa. Primer takšnega izpisa lahko vidimo na sliki 6.



Slika 6.

ZAKLJUČEK

Razvojni sistem, ki je predstavljen v tem prispevku je prestal začetna testiranja, popravke in izboljšave ter pokazal pozitivne rezultate. Trud in čas, ki sta bila vložena v izdelavo opisane programske in materialne opreme se izplačata že po nekaj aplikacijah. Čas razvoja aplikativnih programov se občutno zmanjša na račun iskanja boljših tehničnih in programerskih rešitev. Ugotovili smo že, da je sedanja rešitev posredovanja vhodnih podatkov, ko le-te vpisuje uporabnik preko CRT-ja neprimerna pri aplikacijah z velikim številom vhodnih podatkov. V takšnih primerih je boljše če bi vhodne podatke imeli vnaprej pripravljene in shranjena na posebni vhodni datoteki. Nadaljnja uporaba razvojnega sistema bo z novimi aplikacijami verjetno prinesla zahteve po dodatnih izboljšavah tako materijalne kot programske opreme.

LITERATURA

[1] Materialna in programska podpora za razvoj prototipov z integriranim mikroračunalnikom Iskra EMZ 1001 ;
D.Miljan, P.Reinhardt, P.Kolbezen ;
IJS Delovno poročilo Dp-2407,
Ljubljana, december 1981.

[2] Križni zbirnik na bazi makro ekspanzije za mikroračunalnik EMZ 1001 ;
P.Reihardt, R.Reinhardt ;
IJS Delovno poročilo Dp-2406,
Ljubljana, december 1981.

[3] Projektiranje z integriranimi računalniki ;
Davor Miljan ;
Časopis Informatica 4/1980, str.29-35.

[4] Mikroračunalnik EMZ 1001,
katalog Iskra - Industrija elementov za elektroniko - Mikroelektronika,
Izdala: Iskra Commerce 09. 78.

[5] Integrirani mikroračunalnik Iskra EMZ 1001 ;
Dušan Raič ;
Časopis Informatica 1/1979, str.12-23.

[6] Projektiranje z integriranimi mikroračunalniki ;
Davor Miljan ;
IJS Delovno poročilo Dp-2146,
Ljubljana, december 1980.

[7] Microprocessors and software design tools;
Keith D. Baker ;
Microprocessors and microsystems,
Vol 3, No.2, March 79, page 87 - 93.

[8] MicroSim - a new approach to program development ;
David Cosserat ;
Microprocessors and microsystems,
Vol 3, No.2, March 79, page 95 - 98.

[9] Designing with single chip microcomputers
Markus Moser ;
Microprocessors and microsystems,
Vol 3, No.3, April 79, page 135 - 139.

[10] New generation of microsystem simulators
F.W van Linden ;
Microprocessors and microsystems,
Vol 4, No.1, jan/feb 80, page 5 - 9.

INFORMATICA 4/1982

MATEMATIČNI MODELI MEHURČNIH PREKLOPNIH VEZIJ
1. DEL

P. KOLBEZEN,
B. MIHOVILOVIĆ,
J. ŠILC

UDK: 681.327.6

INSTITUT „JOŽEF STEFAN", LJUBLJANA

Problemi načrtovanja sistemov v najsodobnejših tehnologijah se vse bolj nanašajo na strukturo in modularnost sistemov. Članek preučuje uporabo matematičnih modelov, na katerih je mogoče obravnavati probleme logičnega načrtovanja v tehnologiji magnetnih mehurčkov. V drugem delu članka so definirani novi problemi minimizacije in razredi funkcij v zvezi z ohranjevanjem mehurčkov in lastnostmi funkcijskih izhodov.

MATHEMATICAL MODELS OF MAGNETIC BUBBLE LOGIC. In current technologies problems of structure and modularity in design have become increasingly important. In this paper it is considered the use of models to treat logic design problems in the technology of magnetic bubbles. In the second part of the paper new minimization problems and associated classes of functions are defined with respect to bubble preservation and function fanout properties.

1. UVOD

Večina problemov načrtovanja digitalnih sistemov se je v preteklosti nanašala na minimizacijo njihovih komponent. Načrtovanje v dandanašnjih tehnologijah pa vse bolj in bolj zadeva probleme, ki se nanašajo na strukturo in modularnost sistemov. Tovrstni problemi bodo brez dvoma še bolj prisotni v prihodnjih tehnologijah. Pri uvajanju nove tehnologije je za poenoten pristop k reševanju problematike načrtovanja potrebno najpreje izdelati model, ki natančno odraža fizikalne značilnosti tehnologije, in na katerem je mogoče probleme tudi matematično obravnavati. Mnogo problemov se pogosto rešuje na takšnem matematičnem modelu, ki je podoben modelom prejšnjih tehnologij.

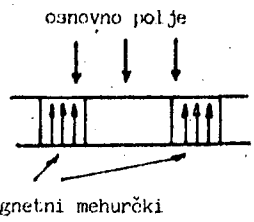
V tem delu bomo proučevali modele, na katerih je mogoče obravnavati probleme logičnega načrtovanja v eni od tehnologij, ki kaže potencialno uporabnost v bodočih digitalnih sistemih. To je tehnologija magnetnih mehurčkov. Videli bomo, da je mogoče ne le modelirati logiko magnetnih mehurčkov, ampak jo tudi obravnavati z vidika logičnega načrtovanja. Če hkrati upoštevamo tudi časovne parametre, so problemi podobni problemom načrtovanja asinhronih sistemov, čeprav jim niso povsem enaki. Definirali bomo nove probleme minimizacije in razrede funkcijskih izhodov.

Videti je, da bodo magnetni mahurčni pomnilniki širše uporabljivi v tistih bodočih digitalnih sistemih, v katerih bodo potrebni večji pomnilniki ali majhna poraba. V takšnih napravah je zaželjeno, da je sicer kompleksni vmesnik med mehurčnim pomnilnikom in ostalim delom naprave, ki je grajen v drugačni tehnologiji, čim manjši in preprostejši. To pa je tudi eden od zanimivejših problemov načrtovanja logike, ki uporablja magnetne mehurčke.

Magnetne mahurčke lahko uporabljamo tako za pomnenje informacij, kot tudi za izvajanje logičnih funkcij. Čeprav je hitrost delovanja mehurčnih sistemov za večkratni velikostni razred nižja od hitrosti sodobne polprevodniške tehnologije in s tem tudi njihova uporaba nekoliko omejena, so mehurčni sistemi zanimivi tam, kjer je pomembna zlasti robustnost, majhna poraba energije, zanesljivost in čim manjše vzdrževanje.

2. FIZIKALNE OSNOVE

Nekatere vrste magnetnih materialov, tako kot enojni kristalni magnetni oksidi in ortoferiti, imajo lastnost, da se lažje magnetizirajo v eni smeri kot v drugi. V tanki ploščici takšnega materiala se pri ustreznih pogojih z magnetnim poljem, ki je pravokotno usmerjeno na ploščico v smeri lažje magnetizacije, ustvarijo stabilne magnetne domene. Te domene imajo smer magnetizacije, ki je nasprotna smeri zunanjega magnetnega polja. Pod vplivom polja primerne jakosti v smeri lažje magnetizacije postanejo magnetne domene celindrične oblike. Takšne domene imenujemo magnetne mehurčke. Slika 1 kaže presek nad ortoferitno ploščico, zunanjim poljem in polarizacijo mahurčkov.



Slika 1. Presek ortoferitne ploščice

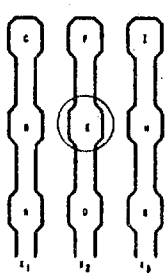
Premer mehurčkov je nekaj do več sto mikronov. Razlike so odvisne od izbire magnetnega materiala.

Velikost in obliko magnetnih domen določa jakost zunanjega polja. Če polje upada, premer mehurčka narašča in končno postane domena necilindrična. Nasprotno pa z naraščanjem zunanjega polja upada premer mehurčka in pri premočnem polju le-ta izgine.

Namestitev mehurčkov v ploščici je lahko omejena na končno množico možnih pozicij. Prisotnost ali odsotnost mehurčka na določeni lokaciji moremo interpretirati kot prisotnost logične vrednosti "1" ali "0" spremenljivke, ki je določena z lokacijo. Mehurčki lahko potujejo, podobno kot delci, od ene lokacije do druge. Poleg tega odbijajo drug drugega, kar je mogoče izkoriščati za izvrševanje logičnih funkcij. S tem pa se odpirajo magnetnim mehurčkom dodatne možnosti v aplikacijah procesiranja podatkov.

3. PROPAGACIJE MEHURČKOV

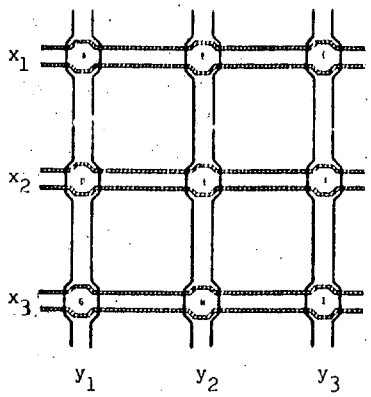
Poznamo več metod, ki jih uporabljamo za premike mehurčkov znotraj ploščic: na sliki 2 vidimo eno od možnih metod, ki je osnovana na prevodni tokovni zanki. Ta ustvarja lokalna magnetna polja. Iz slike vidimo, da so zanke zaključene le na enem koncu. Tako je olajšana njihova izdelava, osnovana na tehniki tankoplastnega filma.
Mehurčki se bodo premaknili iz ene zanke v drugo, sosednjo zanko pod vplivom tokovnega impulza v sosednji zanki v smeri, ki v prvi zmanjša vpliv zunanjega polja.



Slika 2. Propagacija s prevodno zanko

Na sliki 2 se bo mehurček pod tokovnim impulzom skozi zanko X_3 premaknil iz lokacije E v H. Podobno se zaradi tokovnega impulza v zanki X_1 premakne mehurček iz E v B. Tok v zanki pritegne mehurček iz sosednjih zank. Da to preprečimo v eni od sosednjih zank, moramo s tokovnim impulzom skozi izbrano zanko ustvariti primerno "zaviralno polje", ki prepreči, da jo mehurček zapusti.

Z drugo množico zank, ki je pravokotna na prvo množico (slika 3), moremo mehurček premakniti iz poljubne pozicije v katerokoli drugo od štirih sosednjih pozicij. S primernim zaporedjem nadaljnih impulzov pa v katerokoli drugo pozicijo. Mehurček iz pozicije A na sliki 3 premaknemo v pozicije H z eno od naslednjih impulznih sekvenc: Y_2,Y_3,X_2 ali Y_2,X_2,Y_3. To metodo imenujemo metodo kontroliranega dvodimenzionalnega premikanja magnetnih mehurčkov.



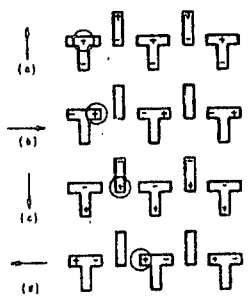
Slika 3. Propagacija s prevodno zanko v dveh dimenzijah

Pri drugi metodi propagacije mehurčkov je uporabljena permalojna prevleka in v njeni ravnini rotirajoče magnetno polje /1/. Permalojni vzorci na ploščici vsebujejo mehurčke. Na sliki 4 vidimo vzorec, ki se pogosto uporablja. Sestavljajo ga T-ji in palice (I-ji). Propagacijo mehurčkov takšnega tipa imenujemo T-paličasto ali TI-propagacijo.

Pri TI-propagaciji mora obstajati osnovno magnetno polje, ki drži mehurčke v stabilnem stanju. Drugo polje, ki se vrti v ravnini materiala, pravokotno na osnovno polje, pa rabi za gibanje mehurčkov. Zaradi visoke permeabilnosti permalojnega nanosa ustvarja magnetno ravninsko polje magnetne pole v permalojni prevleki. Pri določeni smeri ravninskega magnetnega polja dosežejo mehurčki energetsko šibko pozicijo mirovanja. Rotirajoče magnetno polje v ravnini pa tedaj povzroči premik mehurčkov. Te smatramo za nekakšne "magnetne naboje", ki poskušajo doseči pozitivne pole permalojnih magnetov. Slike 4a do 4d kažejo gibanje mehurčkov med enim obratom magnetnega polja. V nasprotno smer rotirajoče polje pa povzroči pomik mehurčka v smeri, ki je nasprotna

prejšnji smeri. Propagacija mehurčka, ki izkorišča permalojne prevleke, kot na primer zgoraj obravnavana metoda TI-propagacije, se zaradi preproste kontrole najpogosteje uporablja.

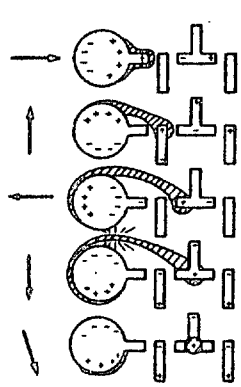
V primeru propagacije, ki izkorišča prevodna vezja, si morajo tokovni impulzi zank slediti v točni sekvenci. To omogoča gibanje mehurčka od ene lokacije do druge. Zahtevana sekvenca pa ni potrebna v primeru TI-propagacije. V primeru TI-organizacije na sliki 4 lahko uporabimo pomikalni register. Mehurček se bo pomaknil od enega T-vzorca do naslednjega v času enega cikla rotirajočega magnetnega polja.



Slika 4. TI-propagacija

3.2. Generiranje, destruiranje in detektiranje mehurčkov

Permalojna vezja se lahko uporabljajo tudi za generiranje in destruiranje mehurčkov /1,2/. Generacija mehurčka nastane s cepitvijo enega mehurčka v dva, kar nazorno kaže slika 5. Velik permalojni disk na začetku TI-propagacijske poti ima mehurček, ki predstavlja vir novega mehurčka. Pod vplivom rotirajočega magnetnega polja se ta mehurček, ki je priklenjen na TI-vzorec, raztegne in končno razdeli v dva dela. Nato zavzameta oba mehurčka premer, ki je določen z jakostjo osnovnega magnetnega polja. Prvi mehurček ostane na permalojnem disku, drugi pa se za tem širi vzdolž TI-propagacijske poti. Destruiranje mehurčka povzroči obraten proces, ki se konča z združitvijo "ponornega" mehurčka z velikim diskom na koncu TI-propagacijske poti.



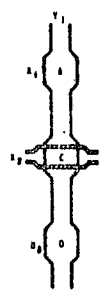
Slika 5. Generiranje mehurčka

Pri povezovanju magnetnega mehurčnega pomnilnika ali logike s konvencionalnimi vezji moramo detektirati prisotnost mehurčkov na določenih lokacijah in generirati ustrezne električne signale. V ta namen se uporablja več metod. Znane so: magnetooptična metoda detekcije, detekcija, ki izkorišča Hallov efekt, in metoda odtipavanja z magnetno rezistenčnimi napravami /1/.

3.3. Logične operacije

Z magnetnimi mehurčki je mogoče realizirati tudi osnovne logične funkcije. Obstoječe realizacije takšnih funkcij so osnovane na magnetostatičnem odboju med mehurčki, ki se nahajajo na sosednjih lokacijah. Mehurčke je mogoče prenesti na željene lokacije z rabo ene od zgoraj obravnavanih propagacijskih shem. V obeh obravnavanih shemah je pomik mehurčka pri vsakokratnem tokovnem impulzu ali rotaciji polja konstanten. Pri izvajanju logičnih operacij morajo biti mehurčki krmiljeni na poseben sinhroni način.

Slika 6 kaže metodo realizacije nekaterih preprostih logičnih funkcij dveh spremenljivk s prevodnimi zankami. Najpreje se mehurčka, ki predstavljata spremenljivki x_1 in x_2, razširita na lokaciji A in B. Mehurček bo prisoten na poziciji A, če in samo če bo $x_1 = 1$, in na poziciji B, če in samo če bo $x_3 = 1$. Predpostavljamo, da v začetku mehurček ni prisoten na poziciji C. Če steče skozi X_2 tokovni impulz, se bosta mehurčka odbijala drug od drugega. Torej se bo v C pojavil mehurček le tedaj, če se mehurček v času tokovnega impulza skozi X_2 nahaja v eni izmed pozicij A ali B. Ker predstavlja pozicija C logično funkcijo vsote po modulu 2 $(x_1 \oplus x_3)$. Na pozicijah A in B se mehurčka obdržita, če in samo če se v začetku hkrati nahajata na obeh pozicijah. Zato pozicija A in hkrati tudi B predstavljata funkcijo logičnega produkta $(x_1 . x_3)$.

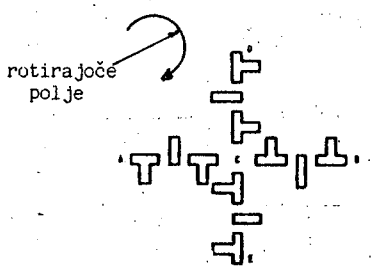


Slika 6. Realizacija logičnih funkcij pri tokovnih zankah

Z različno geometrijo propagacijskih permalojnih vzorcev ali z različnimi materiali propagacijskih vzorcev je mogoče zagotoviti, da se mehurček postavi na pozicijo C pod vplivom tokovnega impulza v X_2 tudi, če se mehurčka hkrati nahajata na obeh pozicijah A in B. V tem primeru predstavlja pozicija C funkcijo $x_1 + x_3$.

Možna pa je tudi sekvenca, ki omogoča realizacijo funkcije $x_1.x_3$ na poziciji C, na naslednji način: s tokovnim impulzom skozi X_2, se na poziciji C pojavi mehurček, če je le-ta prisoten na poziciji A ali B ali na obeh pozicijah hkrati. Če se v naslednjem koraku pojavi tokovni impulz skozi X_2 ustrezne jakosti in nasprotne smeri, se mehurček v C (če obstaja) uniči. Ponovni impuls v X_2 pa lahko pomakne mehurček, ki je ostal na poziciji A ali na poziciji B, v pozicijo C. Po treh tokovnih impulzih se mehurček nahaja na obeh pozicijah A in B.

Logične funkcije je možno realizirati tudi s TI-propagacijo. Takšno realizacijo prikazuje slika 7. Z rotirajočim poljem v ravnini, se mehurčka v A (ki predstavlja spremenljivko x_1) in/ali v B (ki predstavlja spremenljivko x_2) pomikata v C, od te pozicije pa dalje proti D ali E. S primerno namestitvijo vzorcev T in I na križišču dveh poti je mogoče doseči, da se mehurček pomika po eni sami poti, po tako imenovani "lažji poti", če le ni v neposredni bližini drugega mehurčka.

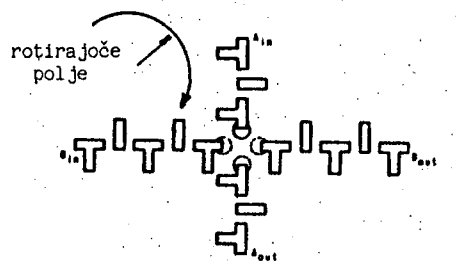


Slika 7. Realizacija logičnih funkcij pri TI-propagaciji

Odbojni vpliv sosednjih mehurčkov povzroči, da začne mehurček potovati po drugi poti, tako imenovani "težji poti". Na sliki 7 naj bo pot do pozicije D "lažja pot", medtem ko naj bo pot iz obeh pozicij A in B "težja pot".

Če se mehurček najpreje nahaja ali na poziciji A ali na poziciji B, bo proti D potoval po lažji poti. Če pa se mehurček hkrati nahaja na obeh pozicijah A in B, bosta oba mehurčka dosegla C istočasno. Eden od mehurčkov bo odbit v C, potoval proti E po "težji poti", medtem ko bo drugi mehurček potoval proti D po "lažji poti". Na ta način predstavlja lokacija D funkcijo $x_1 + x_2$, lokacija E pa funkcijo $x_1.x_2$. Na podoben način je mogoče realizirati tudi druge osnovne logične funkcije s primerno geometrično ureditvijo permalojnih vzorcev. Z opisanimi elementi, ki so standardnih velikosti in struktur, je možno graditi polja elementov poljubnih velikosti. Takšna polja omogočajo realizacijo različnih logičnih funkcij. Pri tem je vsak osnovni element v interakciji s sosednjimi elementi.

Slika 8 kaže mehurčno realizacijo dveh vodnikov, ki križata drug drugega. Vezje mora biti inicializirano s kroženjem mehurčka na sečišču obeh poti, kar je tudi razvidno iz slike 8. Mehurček, ki bo pripotoval od vhoda A_{in}, bo odbil krožeči mehurček navzdol po poti k izhodu A_{out}, sam pa bo začel krožiti na sečišču obeh poti. Enako se dogaja, če mehurček pripotuje od vhoda B_{in}. Kateri mehurček bo ostal v sečišču, če mehurčka potujeta hkrati po obeh poteh, je odvisno od relativnih časovnih situacij mehurčkov na poteh A in B, zanesljivo pa oba mehurčka pripotujeta do izhodov A_{out} in B_{out}.



Slika 8. TI-realizacija prekrižanih poti

4. MATEMATIČNI MODEL MEHURČNIH INTERAKCIJ

Lastnosti, ki se nanašajo na logiko magnetnih mehurčkov, bomo študirali na matematičnem modelu mehurčnih interakcij. Model bomo zgradili na osnovi nekaterih značilnih fizikalnih lastnostih mehurčkov, ki smo jih obravnavali v gornjih razdelkih.

Predpostavljamo, da ja V množica vrhov. Ti naj predstavljajo množico vseh možnih položajev mehurčkov. Predpostavljamo, da je lahko katerikoli par vrhov iz V v medsebojni interakciji. V praksi sta lahko v interakciji samo dva sosednja mehurčka. Mi pa v tem poglavju predpostavljamo, da takšna omejitev ne obstaja. Na ta način se lažje dokopljemo do nekaterih predhodnih rezultatov, ki so neodvisni od omenjene omejitve. Najprej obravnavamo najpreprostejšo operacijo, ki omogoča prenos mehurčka od vrha A k vrhu B. Pri tem naj se gibanje posameznih mehurčkov izvaja po propagacijski metodi, ki je osnovana na prevodnih zankah. Metodo smo že predhodno obravnavali v poglavju 2.

Mehurček se pomakne od A k B pod vplivom primernega magnetnega polja v B in zaviralnega polja na vseh sosednjih lokacijah, razen na lokaciji A. Mehurček se giblje od A k B, če in samo če je prisoten v A, v B pa ne. Označimo takšno interakcijo med A in B z e = (A,B). Imenujemo jo "mehurčni prenos" (bubble transfer /3/). Kot smo že zgoraj omenili, predpostavljamo, da sta v našem matematičnem modelu A in B lahko poljubna vrhova. V praksi pa se izkaže, da sta takšna vrhova lahko le sosednja vrhova. Ekvivalentno temu lahko tudi predposta-

vljamo, da so v našem matematičnem modelu vsi vrhovi iz V sosednji vrhovi poljubno izbranemu vrhu iz V.

Naj bo $X \subseteq V$ množica vrhov, ki v začetku vsebujejo mehurček, in X^e množica vrhov, ki bodo vsebovali mehurček po mehurčnem prenosu e = (A,B). Potem

$$X^e = (X - \{A\}) \cup \{B\} \quad \text{če } A \in X \text{ in } B \notin X$$

in drugače

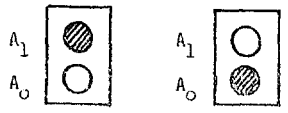
$$X^e = X .$$

Vsako takšno interakcijo bomo smatrali kot "instrukcijo" in sekvenco takšnih instrukcij kot "program" /3/.

Naj bosta a in b binarni spremenljivki, ki predstavljata začetna pogoja na lokacijah A in B (to je a = 1, če je na lokaciji A mehurček prisoten, in A = 0, če mehurček ni prisoten). Nadalje naj bosta a' in b' binarni spremenljivki, ki predstavljata pogoja na teh lokacijah po interakciji e = (A,B), in sicer

$$a' = a.b$$
$$b' = a + b$$

Tak mehurčni prenos uporabljamo za izvajanje logičnih funkcij "IN" in "ALI". Da dobimo operacijo komplementiranja, moramo pri odsotnosti mehurčka mehurček generirati in obratno: pri prisotnosti mehurčka mehurček odstraniti; v obeh primerih z enako sekvenco instrukcij prenosa. To pa ne moremo izvesti samo z mehurčnim prenosom. Spremenljivko moramo določiti z dvema mehurčnima lokacijama, namesto ene same. Predstavitev takšne spremenljivke kaže slika 9. Logična vrednost spremenljivke a je določena s prisotnostjo oz. odsotnostjo mehurčka na lokacijah A_1 in A_2. Prisotnost mehurčka na lokaciji A_1 in ne na lokaciji A_2, predstavlja a = 1, ter prisotnost mehurčka na lokaciji A_o, in ne na lokaciji A_1, predstavlja a = 0. Na ta način sta obe logični vrednosti določeni s prisotnostjo mehurčkov. Pri takšni določitvi spremenljivke je namreč nepotrebno, da pri odsotnosti mehurčka v poziciji A_1 zagotovimo prisotnost mehurčka v A_o in obratno. Vidimo, da je za n dvojiških spremenljivk potrebnih 2n lokacij.



Slika 9. Predstavitev spremenljivke z dvema lokacijama

Naj bo 2^n možnih kombinacij n spremenljivk zakodiranih v kodu "k od m". To je kod, ki zagotavlja natančno k izhodov m spremenljivk v "1" pri poljubni vhodni kombinaciji. Lahko bi dokazali:

1., da je pri takšnem načinu kodiranja mogoče realizirati poljubno logično funkcijo brez funkcije negacije, in

2., da je predstavitev 2^n vhodnih kombinacij potrebnih le m lokacij, če sta m in k takšna da velja

$$\binom{m}{k} \geq 2^n$$

Predpostavimo, da je vsaka spremenljivka določena z dvema lokacijama in da na lokaciji B ni mehurčka. Potem moremo spremenljivko a komplementirati s programom

$P = e_1 e_2 e_3$, kjer je
$e_1 = (A_1,B)$; $e_2 = (A_o,A_1)$ in $e_3 = (B,A_o)$
Po izvršenem programu je vsebina na obeh lokacijah dana z enačbama

$$a_1^P = a_o \quad \text{in} \quad a_o^P = a_1$$

kjer predstavlja a_i vsebino na lokaciji A_i, i = 0,1.

Ker smo dokazali, da moremo logične operacije IN, ALI in NE realizirati le z mehurčnimi prenosi, smemo zaključiti, da moremo z njimi izračunati tudi katerokoli logično funkcijo. Klasične metode realizacije pa upoštevajo tudi izhodni indeks. Do tu nismo pokazali, da lahko ta indeks upoštevamo tudi v našem matematičnem modelu mehurčnih interakcij. Vedeti moramo namreč, da bi v nasprotnem primeru samo z mehurčnimi prenosi ne mogli realizirati katerokoli logično funkcijo. V dokaz temu potrebujemo nekaj predhodnih ugotovitev.

Naj bosta $X \subseteq V$ in $Y \subseteq V$ množici lokacij, ki vsebujejo mehurčke. Število lokacij, ki so skupne množicama X in Y, to je $|X \cap Y|$, bomo imenovali prekrivanje množic X in Y.
Lahko dokažemo naslednje: Če e = (A,B) in sta X, $Y \subseteq V$, množici, ki v začetku vsebujeta mehurčke, velja $|X^e \cap Y^e| \geq |X \cap Y|$. Dokaz najdemo v /5/.

Nadalje lahko zapišemo trditev o nezmanjševanju prekritja: za vse programe $P = e_1 e_2 \ldots, e_n$ in vse X, $Y \subseteq V$ velja $|X^P \cap Y^P| \geq |X \cap Y|$. Dokaz te trditve sledi direktno iz trditve 1.

Naj bo V_1, $V_2 \subset V$, $V_1 \cap V_2 = \emptyset$ in konfiguracija mehurčkov taka, da je $X_1 \subseteq V_1$ in $X_2 \subseteq V_2$. X_2 je kopija od X_2, če obstaja enolična preslikava med elementi iz X_1 in elementi iz X_2. Program P, ki ga je mogoče ponovno uporabiti (ponovno uporabljiv program P), je takšen program P, da za nek V_1, $V_2 \subset V$ z mehurčno konfiguracijo X_1 in V_1, P generira kopijo X_1 v V_2 tako, da se X_1 v V_1 ohrani. Lahko ugotovimo, da takšen ponovno uporabljiv program P, $P = e_1 e_2 \ldots e_n$ ne obstaja. Dokaz za gornjo trditev najdemo v /5/.

Iz trditve 3 lahko zaključimo, da izhodni indeks ne moremo realizirati samo z instrukcijami mehurčnega prenosa, in zato tudi ne vseh kombinacijskih funkcij. Lahko pa realiziramo operacije IN, ALI in NE, če uporabimo kodiranje, ki je ponazorjeno s sliko 9 /3/. Vendar so fizikalno možni tudi drugi tipi interakcij, ki jih dodamo

našemu modelu, ne da bi prišli v nasprotje z realnostjo. Izhodni indeks implementiramo, če instrukcijem mehurčnih prenosov dodamo še tako imenovano "instrukcijo mehurčne podvojitve". Označimo jo z $e = (A,B)_s$ in naj velja:

$X^e = X \cup \{B\}$, če $A \in X$, $B \notin X$

$X^e = X$, če $A \notin X$ ali $B \in X$

Funkciji, ki sta po tej interakciji predstavljeni na lokacijah A in B, sta

$a' = a$

$b' = a + b$

Če v začetku B ne poseduje mehurčka, se po interakciji pojavi na izhodu spremenljivke a. Inicializacijo omogočimo z dvema dodatnima interakcijama: "generiranje mehurčka" in "destruiranje (anhilacija) mehurčka". Instrukcijo generiranja mehurčka označimo z $e = (1,A)$. Velja

$X^e = X \cup \{A\}$

Instrukcijo destruiranja mehurčka označimo z $e = (0,A)$. Velja

$X^e = X - \{A\}$

Na sliki 11 je prikazana razpredelnica vseh štirih tipov interakcij:

Ime	Opis	Lokacije z rezultirajočimi mehurčki
Prenos	$e = (A,B)$	$X^e = \begin{cases} (X - \{A\}) \cup \{B\}, & \text{če } A \in X,\ B \notin X \\ X, & \text{če } A \notin X \text{ ali } B \in X \end{cases}$
Podvojitev	$e = (A,B)_s$	$X^e = \begin{cases} X \cup \{B\}, & \text{če } A \in X \\ X, & \text{če } A \notin X \end{cases}$
Anhilacija	$e = (0,A)$	$X^e = X - \{A\}$
Generacija	$e = (1,A)$	$X^e = X \quad A$

Slika 11. Štirje tipi operacij nad mehurčki

Lahko bi dokazali, da z rabo vseh štirih tipov interakcij in zgoraj omenjenega kodiranja ne moremo še realizirati vseh kombinacij funkcij. Izjemoma pa, če obstajajo nekatere lokacije, na katere se lahko mehurčki začasno premaknejo. Velja /5/ naslednja trditev:

Kombinacijske funkcije so neizračunljive z vsemi štirimi tipi instrukcij (prenosa, podvojitve, anhilacije in generacije), če so informacije prisotne na vseh lokacijeh v V.

Gornja trditev velja tudi v primeru rabe kodiranja na sliki 9, kjer V vsebuje 2n lokacij z vrednostmi n spremenljivk. Program, ki bi uporabljal interakcije opisanih štirih tipov in omogočal komplementa vseh spremenljivk, ne obstaja. Pri tem se moramo zavedati, da smo pri preje izvajanem programu komplementiranja binarnih spremenljivk potrebovali tudi dodatne lokacije. Velja naslednja trditev:

Vse kombinatorne logične funkcije so izračunljive, če uporabljamo štiri tipe instrukcij, primerno kodiranje vhodov in, če je začasno potrebno dovolj veliko število lokacij. Dokaz najdemo v delu /5/.

5. ZAKLJUČEK

Pokazali smo, kako moremo logični element vsote po modulu 2 in element IN dveh spremenljivk realizirati z interakcijami magnetnih mehurčkov. Nadalje smo oblikovali matematični model takšnih interakcij. Čeprav zgoraj omenjena elementa še ne predstavljata polne množice osnovnih elementov, ki bi omogočali realizacijo poljubne kombinacijske funkcije, je vendar nekatere od teh mogoče realizirati z omenjenimi interakcijami. Pogoj obstaja le v možnosti generiranja mehurčkov (konstanta 1) in v možnosti realizacije izhodnega indeksa. Indeks moremo realizirati z interakcijo med lokacijama s konstanto 1 in spremenljivko x, če le-ta realizira izhoda x in $\bar{x}$. S ponavljanjem takih postopkov dobimo želeno število izhodov x. Pri tem pa ne potrebujemo nobenega kodiranja, saj so hkrati neposredno dosegljivi tudi komplementi $\bar{x}$ od x.

Medtem ko smo ugotavljali pogoje, ki so potrebni za realizacijo logičnih funkcij z mehurčnimi preklopnimi vezji, nismo obravnavali tudi število insturkcij, ki je potrebno za takšno realizacijo. Dodati pa moramo, da minimizacija števila instrukcij predstavlja poseben problem za učinkovito realizacijo logičnih funkcij.

6. LITERATURA

1) P. Kolbezen, R. Trobec, J. Šile, B. Miholović: Mehurčni pomnilniki, IJS Ljubljana, Raziskovalna študija, številka pogodbe: 03-BR-PK-1226/81, junij 81

2) Šile: Magnetni mehurčki v digitalni tehniki, magistrski delo, Univerza Edvarda Kardelja, Fakulteta za elektrotehniko, Junij 1982

3) R. L. Graham: A Mathematical Study of a Model of Magnetic Domain Interactions, BSTJ, vol.49, pp.1627-1644, October 1970

4) R. C. Minnick: A System of Magnetic Bubble Logic, IEEE Trans. on Computers, vol. C-24, pp.217-218. February 1975

5) P. Kolbezen: Optimizacijski problemi mehurčne logike, IJS Delovno poročilo Dp- 2976, Ljubljana, december 82

INFORMATICA 4/1982

SINOPTIKA MIKRORAČUNALNIŠKO VODENEGA SISTEMA

ROMAN TROBEC,
IZTOK LESJAK,
MATJAŽ ŠUBELJ

INSTITUT JOŽEF STEFAN, LJUBLJANA

UDK: 681.327.12

Opisan je enostaven način prikaza sinoptične sheme industrijskega procesa na grafičnem ekranu.

THE CONTROL PANEL OF μP CONTROL SYSTEM. A simple method of representing industrial sistem's scheme via graphic CRT(s) is described.

UVOD

V članku smo opisali klasično realizacijo sinoptične sheme. Posebej smo analizirali njene slabosti in nato predlagali preprosto realizacijo sinoptične sheme s CRT ekranom. Opisali smo programsko opremo za risanje sinoptike in analizirali karakteristike tako realizirane sheme. Sistem je bil realiziran na več procesorski hierarhiski mreži in služi za prikazovanje trenutnih stanj po posameznih dislociranih objektih.

1. SINOPTIKA INDUSTRIJSKEGA SISTEMA

Za komunikacijo operaterja s sistemom je potrebna funkcionalna shema, na kateri so obrisi sistema z vsemi podatki in komandami, ki se zahtevajo za normalno delovanje in upravljanje.

- Shema je lahko popolnoma neodvisna od računalniškega sistema, v primerih, ko je z računalnikom realizirana le dokumentacija delovanja. Običajno je shema v takih primerih vir podatkov za računalniški sistem.

- V realnih procesih so praviloma potrebne tudi komande, kar zahteva, da se del računalniških izhodov veže na sinoptično shemo.

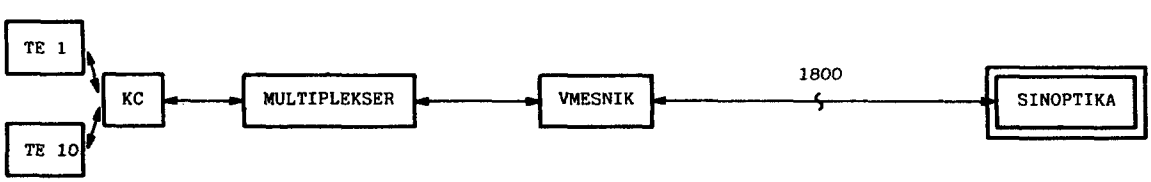
- Pravi pomen dobi sinoptika v sistemih z oddaljenimi točkami upravljanja in izvori podatkov, ob istočasni zahtevi po upravljanju iz centralnega mesta. Primeri takih sistemov so : vodovodni sistemi in sistemi kanalizacije, elektroenergetski sistemi in podobni kompleksni industrijski procesi.

Izkaže pa se, da velikost sinoptične sheme hitro preraste smiselne meje. Klasična shema dobiva nesprejemljive dimenzije, postane velik porabnik električne energije in objekt stalnega vzdrževanja. Zaradi teh razlogov predstavlja velik procent celotne investicije avtomatizacije industrijskega procesa.

Preverimo naša razmišljanja na praktičnem primeru upravljanja in nadzora nad srednje velikim vodovodnim sistemom, ki ima na zaključenem hidrografskem področju deset podsistemov, ki sodelujejo pri proizvodnji in usmerjanju pitne vode. Tipični podsistemi so: izviri, rezervoarji, zasunske komore, vodnjaki, črpalne postaje in podobno. Vsak zaključen podsistem je združen v terminalno postajo, ki jo upravlja sistem za prenos podatkov v centralno mesto upravljanja (komandni center) in prenos komand iz komandnega centra do izbranega terminala.

Povprečno je potrebno za vsak terminal prenašati 30 DA - NE signalov splošnega pomena, 10 alarmnih signalov, 10 komand in 10 meritev, od katerih potrebuje vsaka 8 bitov informacije. Torej je za povprečni terminal potrebno 180 bitov informacij.

Za pregled nad celotnim sistemom potrebujemo v komandnem centru 1800 bitov informacije. Če hočemo to množico podatkov predstaviti na sinoptični shemi mora imeti računalniški sistem v centru 1800 izhodnih linij, torej skoraj isto število povezovalnih kablov in ustreznih signalnih elementov (slika 1.).



Slika 1.

- Če hočemo imeti pregledno shemo, potrebujemo za naš primer shemo z dimenzijo najmanj 6 x 3 m, pri čemer je en bit informacije predstavljen s površino 10x10 cm.

- Sinoptična shema, bi morala biti realizirana z LED - display-i in diodami. Povprečna poraba električne energije na bit informacije je 10 mA/bit. Za celotni sistem bi se v tem primeru trošila moč približno 200 W. (Realizacija s klasičnimi elementi bi zahtevala bistveno večjo porabo moči).

- Zaradi velikih dimenzij nastopi problem pregleda nad celotno shemo.

- Realizacija sheme potrebuje drugačne profile stro-kovnjakov kot realizacija sistema za zbiranje in prikaz podatkov, kar pomeni izgubo časa ob montaži zaradi izmenjave informacij in napak v dokumentaciji na eni in drugi strani.

- Elementi sinoptične sheme niso preveč zanesljivi, zato je potrebno sinoptično shemo neprestano vzdrževati.

2. PREDSTAVITEV SINOPTIKE NA CRT - EKRANU

Veliki večini pomanjkljivosti, ki jih ima opisana klasična izvedba sinoptične sheme, se lahko izognemo, če uporabimo za prikaze stanja grafični ekran. Za prikaz običajne sinoptike ustreza ekran, ki omogoča:

- izpis nabora ASCII znakov,
- risanje ravnih in vogalnih črt,
- izpis znakov z reverznim ozadjem (reverse),
- utripanje znaka (blink),
- periodično spreminjanje svetlosti znaka (bold),
- relativni premik kursorja,
- absolutni premik kursorja,
- shranitev trenutnega položaja kursorja (save),
- restavriranje položaja kursorja (unsave).

Vsem naštetim zahtevam ustreza npr. CRT ekran domače proizvodnje "Kopa 1000".

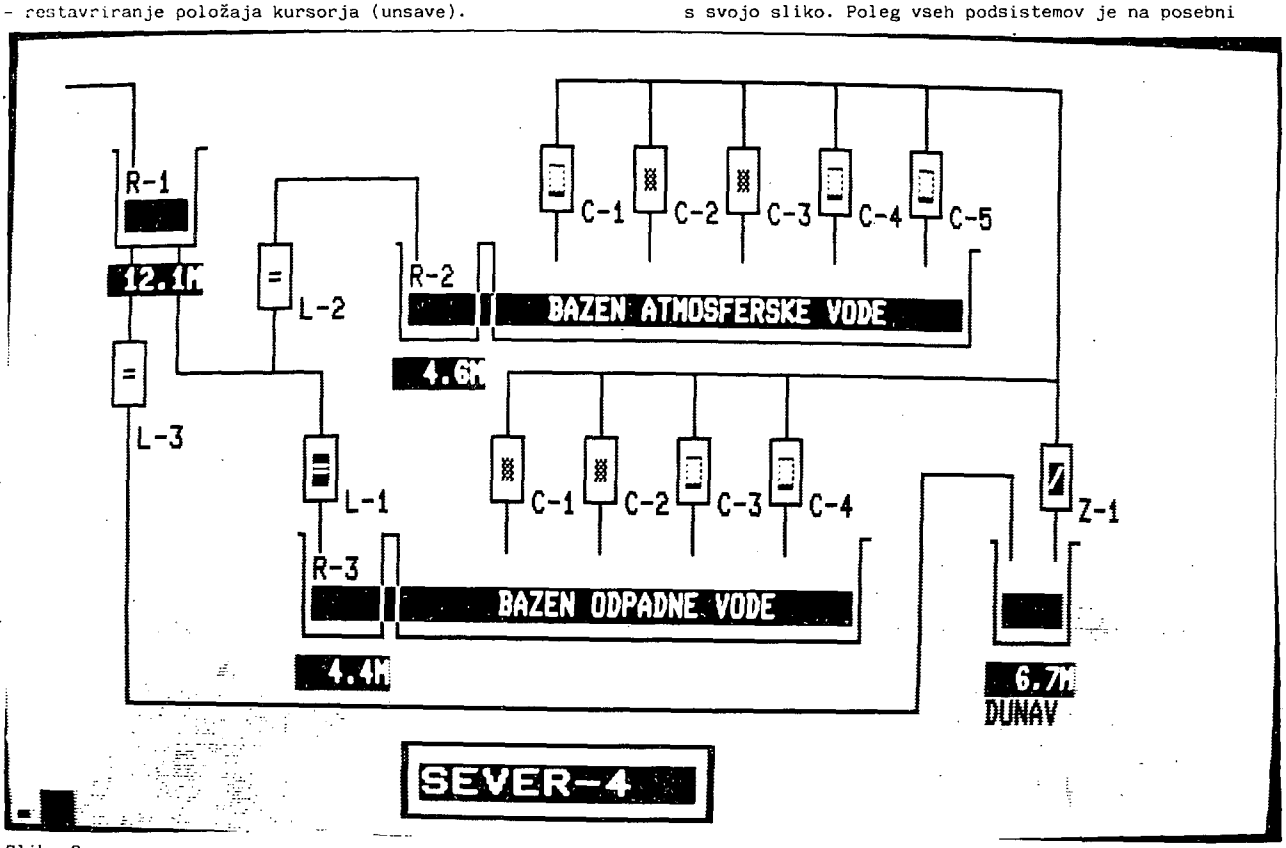
Shema sistema, ki predstavlja nespremenljivo ogrodje je shranjena v obliki tabel v pomnilniku mikroračunalniškega sistema.
Izpis signalov DA - NE je realiziran z reverznim ozadjem. Alarmnim signalom pa je dodano še utripanje (blink) in povečanje intenzitete (bold).
Meritve in številčne vrednosti (stanje števcev) so predstavljene z ustreznimi števili in enotami.

V tabelah so shranjeni tudi vsi naslovi, oznake elementov sistema in komentarji.

Primer, ki smo ga obravnavali v prejšnjem poglavju, je možno predstaviti s štirimi ekrani. Razdalja gledanja se zmanjša za 10 krat. Osnovni element meri sedaj le 1 cm x 1 cm, celotna shema pa dobi dimenzije 60 cm x 30 cm. Zorni koti ostanejo enaki kot pri klasični izvedbi, zato je tudi preglednost nad shemo enaka, če ne še boljša.

Končna realizacija dopušča tri možnosti:

- štiri fizične enote, na katerih se hkrati riše po 1/4 slike. Slabost take realizacije je, ker potrebujemo več izhodnih kanalov oziroma več procesorjev, prednost pa je v tem, da je vedno prisotna celotna slika.
- fizično je prisoten le en ekran, ki predstavlja "okno" za pogled na celotno shemo. Premikanje "okna" je lahko izvedeno zvezno (en stolpec, ena vrstica), kar zahteva kompliciran programsko opremo za mikroračunalniški sistem.
Vendar je enostavneje premikati "okno" za cel ekran (80 stolpcev, 20 vrstic), kar še vedno zadošča zahtevam po dobrem pregledu nad sistemom.
- Obstaja še tretja možnost, pri kateri je vsak terminal oziroma funkcionalno zaključeni podsistem predstavljen s svojo sliko. Poleg vseh podsistemov je na posebni



Slika 2.

sliki predstavljen celotni sistem, toda le z najvažnejšimi podatki. Opisani pristop je udoben in eleganten, zahteva pa precej pomnilniškega prostora za shranjevanje vseh slik. V našem primeru bi bilo to približno 11 K-bytov EPROM pomnilnika.

Pri vseh treh možnostih so prvi dve vrstici na ekranu rezervirani za tekoče izpise vseh alarmnih sprememb v sistemu, ki pomagajo usmerjati operaterjevo pozornost na določene detajle v celotnem sistemu. Na Sliki 2. je prikazan primer realizacije sinoptične sheme za majhen terminal v sistemu kanalizacije Novi Sad.

Podajmo še pregled prednosti, ki jih nudi predstavitev sinoptike na grafičnem ekranu:
- bistveno manjše dimenzije (10x) ob isti preglednosti,
- manjša poraba električne energije,
- minimalni materialni stroški,
- zmanjšanje kompleksnosti računalniškega sistema (niso potrebni izhodni kanali),
- možnost sprememb na shemi,
- celotni sistem za prenos in prikaz podatkov je produkt enega profila strokovnjakov,
- enostavna montaža,
- možnost dialoga operater - sistem.

Vse navedene točke pomenijo zmanjšanje investicije, skrajšanje časa izdelave in povečanje zanesljivosti delovanja sistema.

3. OPIS PROGRAMSKEGA PAKETA ZA RISANJE SINOPTIKE

Ob zasnovi programskega paketa smo analizirali več različnih možnosti realizacije. Kriteriji, ki so nas vodili pri analizi, so bili predvsem:

- risanje ne sme zavzemati preveč procesnega časa,
- programski paket ne sme biti preobsežen,
- isti program naj bo uporaben za risanje poljubnih slik,
- enostaven postopek širjenja paketa,
- možnosti implementacije v že delujočih sistemih,
- možnost samostojnega delovanja ali paralelno s sinoptiko.

Program za risanje sinoptike, smo zaradi navedenih vzorkov zasnovali tako, da iz tabel slik ekranov čita tekoči znak in glede na njegovo značilnost izvrši eno izmed procedur. Istočasno označi tekočo proceduro s status registrom.

Za naše zahteve so v prvi obliki zadoščale naslednje procedure:

- Oddaja znaka (ODZN)

Procedura je lahko samostojna (če gre za izpis običajnega znaka) ali pa zaključuje katero koli drugo proceduro s tem, da omogoči oddajo znaka.

- Izpis niza (IZNIZ)

Izpiše posamezne znake kontrolnih ali podatkovnih nizov poljubne dolžine. Kontrolni nizi so namenjeni upravljanju grafičnega terminala, podatkovni nizi pa so znaki in sporočila operaterju.

- Relativni pomik kursorja (RELPOM)

Premakne kursor na novo pozicijo. Ta procedura se uporablja zaradi hitrejšega risanja slik, ker se prazna mesta ne izpisujejo.

Poleg opisanih "sistemskih" procedur smo realizirali še "procesne" procedure, ki so potrebne za predstavitev spremenljivih delov na sliki:

- meritev (M)

Procedura generira štirištevilčno vrednost meritve v petih možnih formatih (xxxx, 0.xxx, x.xxx, xx.xx, xxx.x), s tem, da se izpuščajo začetne ničle, ki so nepotrebne. Podatkovni niz se zaključi z znakom za konec niza.

- signal (ONOFOK)

Procedura ugotovi trenutno stanje elementa procesa in izbere eno izmed možnosti (vključen, izključen, vključen in pokvarjen, izključen in pokvarjen) in glede na to izpiše ustrezni niz (normal, reverse, blink, bold).

3.1. STRUKTURA PROGRAMA ZA RISANJE SINOPTIKE

Program je zasnovan tako, da dopušča razširitve in spremembe, poleg tega pa je možno risati poljubne slike brez omejitev (seveda v okviru zmožnosti grafičnega ekrana).

Karakteristike različnih shem so v tabelah, ki se jih realizira na močnem računalniškem sistemu s pomočjo interpreterja sinoptičnih shem.

```
PROGRAM: RISANJE SINOPTIKE

  BEGIN
    WHILE       ZAHTEVA ZA RISANJE DO
      BEGIN
        PREGLED ZAHTEV ZA RISANJE
        REPEAT
          DEKODIRANJE ZNAKA
          CASE ZNAK OF:
          ' ':      RELPOM
          ' ':      M
          '&':      ONOFOK
          ' ':      IZNIZ
          ELSE:     odzn
        END
      UNTIL KONEC EKRANA
    END
  END.
```

3.2. STRUKTURA TABEL

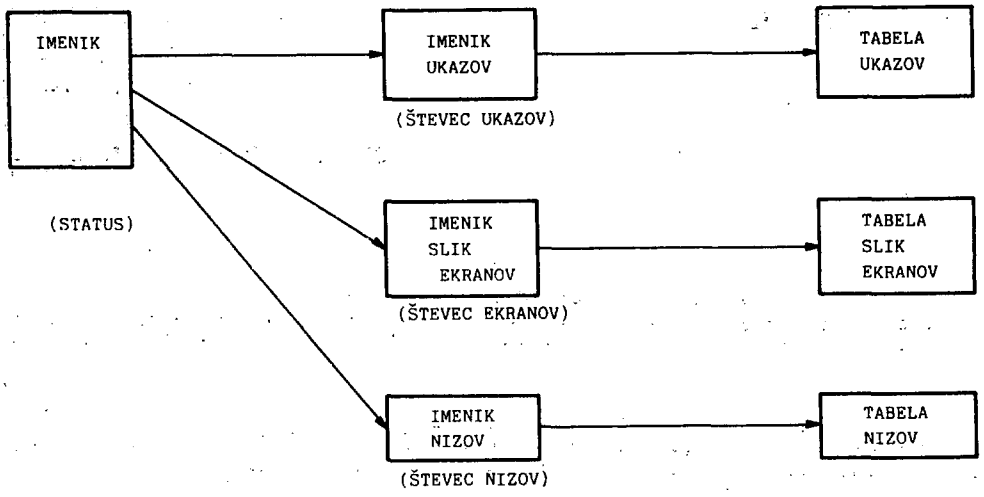
Programski paket ima dostop do podatkov preko splošnega imenika, ki vsebuje začetne naslove posameznih tabel. Uporabljeno je absolutno in relativno adresiranje. Podatki so razdeljeni v tri skupine:

- slike ekranov,
  vsebujejo vse znake (vključno s kontrolnimi), ki sestavljajo slike.

- tabele ukazov,
  vsebujejo naslove spremenljivih vrednosti (procesnih podatkov) v pomnilniku tipa RAM. Pri signalih in okvarah je dodana še maska, ki omogoča bitno predstavitev podatkov, pri meritvah pa je dodana informacija o formatu izpisa.

- tabela nizov,
  vsebuje vse kontrolne in upravljalne nize za grafični ekran. Primeri nizov so: normal, blink, bold, grafic mode, reset, save itd.

Struktura tabel je podana na Sliki 3. (izrazi v oklepajih pomenijo relativno adreso imenikov).



Slika 3.

Edini absolutni naslov, na katerega se nanaša progam za risanje sinoptike, je začetni naslov imenika. Vse drugo adresiranje je relativno, kar zagotavlja veliko fleksibilnost pri risanju različnih slik.

Avtomatsko tvorjenje tabel je dokaj zahteven postopek, ki smo ga realizirali z interpreterjem na miniračunalniku. Interpreter mora omogočati dialog z načrtovalcem sistema in ročno ali avtomatsko tvorjenje tabel ukazov, nizov in slik ekranov.

4. ZAKLJUČEK

Ugotovili smo, da nudi realizacija sinoptične sheme z grafičnim ekranom številne prednosti pred klasično izvedbo, zato je ekonomsko veliko bolj upravičena.

Pri načrtovanju je potrebno paziti na zanesljivost (pasivna ali aktivna rezerva) terminala, ki je v opisani izvedbi kritično mesto upravljanja in nadzorovanja sistema.

5. LITERATURA

(1) A.B.Anue: Status and Trends in Man-Machine Communication, Real - Time data handling and process control, North - Holland Publishing Company, pp 145-151, Brussels and Luxemburg 1980

(2) Dokumentacija sistema za zbiranje podatkov o Kanalizaciji Novi Sad, IJS 1981.

ŠESTO REPUBLIŠKO TEKMOVANJE SREDNJEŠOLCEV S PODROČJA RAČUNALNIŠTVA 1981/82

I. TVRDY,
M. MARTINEC

UDK: 681.3:371.27

INSTITUT JOŽEF STEFAN, LJUBLJANA

Članek podaja poročilo o šestem slovenskem republiškem tekmovanju srednješolcev s področja računalništva. Tekmovanje je organiziralo Slovensko društvo Informatika 10. aprila 1982. V članku so podane vse naloge in rešitve nalog s tekmovanja ter pregled rezultatov.

SIXTH SLOVENE COMPUTER SCIENCE CONTEST FOR HIGH-SCHOOL STUDENTS. This article gives a report on the Sixth Computer Science Contest, organized by Informatika, Slovene Society for Informatics. The article includes the complete set of problems with their solutions and a short overwiev of contest results.

I. Uvod

Ena od rednih dejavnosti Slovenskega društva Informatika je tudi popularizacija računalništva med srednješolsko mladino. Komisija za popularizacijo računalništva je zato skupaj z Institutom Jožef Stefan, Fakulteto za elektrotehniko in Fakulteto za naravoslovje in tehnologijo organizirala že šesto republiško tekmovanje srednješolcev s področja računalništva. Tekmovanja, ki je bilo 10. aprila 1982 na Fakulteti za elektrotehniko v Ljubljani, se je udeležilo 66 tekmovalcev po enem in 35 tekmovalcev po dveh letih pouka računalništva iz 21 srednjih šol.

Tekmovalna komisija je ob izbiranju nalog upoštevala neizenačenost učnih programov, različno znanje dijakov in različne možnosti za delo na samem računalniku. Zato je poskušala izbrati naloge, ki ne zahtevajo posebnega znanja, niti ne temeljijo preveč na praktičnih izkušnjah učencev. Naloge poskušajo pokriti najvažnejše učne smotre, zato zahtevajo od tekmovalca predvsem razvit smisel za algoritmično razmišljanje.

Tekmovalci obeh skupin so imeli na voljo po dve uri in pol časa za reševanje nalog, pri tem pa so smeli uporabljati poljubno literaturo. Uradni programski jeziki tekmovanja so bili pascal, fortran in basic.

II. Naloge za učence po enem letu pouka računalništva

1. Imamo dani dve zaporedji števil med 0 in 9. Opiši algoritem, ki določi:
 - število istoležnih, med seboj enakih elementov v obeh zaporedjih in ga shrani v spremenljivko z imenom črni;
 - število preostalih, med seboj enakih elementov v obeh zaporedjih (ki niso istoležni) in ga shrani v spremenljivko z imenom beli.

POZOR: vsak element upoštevaj pri štetju le enkrat!

Primeri:

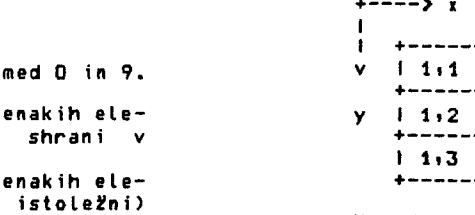
zaporedje 1	zaporedje 2	črni	beli
1234567890	1231111111	3	0
1112333333	0111177373	4	1
1234567890	0000000123	0	4

OPOMBA: naloga izhaja iz igre MEMO (števila 0 do 9 pomenijo barve).

2. V tovarni pijač stoji na koncu tekočega traku robot, ki mora zlagati steklenice v zaboje. Robot ima tipalo, ki mu pove, ali se na traku nahaja steklenica ali ne. V vsakem zaboju je prostora za 12 steklenic. Koordinate predalov v zaboju so označene na sliki. Napiši program za vodenje robota, ki polni zaboje s steklenicami! Predpostaviš lahko, da je na začetku že podstavljen prazen zaboj.

Program ima na voljo naslednje podprograme:

Tipalo	- funkcija ima vrednost 1, če je na traku steklenica, in 0, če je ni;
Primi	- robot vzame steklenico s traku;
Spusti(x,y)	- odloži steklenico v zaboj na mesto (x,y);
NovZaboj	- odstrani zaboj in podstavi novega, praznega.



Koordinate predalov v zaboju.

3. Večina majhnih računalnikov pozna le cela števila med -32768 in 32767. Na takem računalniku bi radi računali z dvajsetmestnimi celimi števili. Kako naj predstavimo tako število v računalniku, če želimo le seštevati in odštevati? Opiši algoritma za seštevanje in odštevanje dveh takih števil! Ne pozabi na predznake!

4. Napiši program, ki izračuna vsa srečna števila, ki so manjša od 1000. Srečna števila so tista, ki preživijo naslednji postopek:

N naravnih števil postavimo v vrsto. 1 izpustimo in začnemo z 2. Iz vrste vzamemo (prečrtamo - vržemo proč) vsako drugo število. Naslednje število, ki stoji za 2 je 3. Iz vrste vzamemo zato vsako tretje število. Naslednje preostalo število, ki stoji za 3 je 7... Za 7 je naslednje preostalo število 9, nato 13,... Postopek ponavljamo, dokler lahko odvzamemo še kakšno število.

Števila štejemo vedno od začetka vrste, to je od 1 dalje!

Prvih nekaj korakov:

```
1 2 3 4  5  6  7  8  9 10 11 12 13 14 15 16 17
  ...                    18 19 20 21 22 23
1 3 5 7  9 11 13 15 17 19 21 23
  ...          (v prejšnji vrsti črtan vsak 2.)
1 3 7 9 13 15 19 21
  ...          (v prejšnji vrsti črtan vsak 3.)
1 3 7 9 13 15 21
  ...          (v prejšnji vrsti črtan vsak 7.)
1 3 7 9 13 15 21    (ker je v prejšnji vrstici
      manj kot 9 števil, postopek zaključimo)
```

III. Naloge za učence po dveh letih pouka računalništva

1. Napiši program, ki prebere dve naravni števili in izpiše njun kvocient na m decimalk natančno. m naj bo tretji prebrani podatek. Upoštevaj, da je m lahko "poljubno" velik.

2. Sestavi postopek, ki naključno izbere 1000 Slovencev. Na razpolago imaš datoteko z imeni, priimki in naslovi vseh Slovencev (podatek iz popisa prebivalcev 31. 3. 1981: vseh Slovencev je 1 891 864) ter generator naključnih števil - funkcijo Slučaj, 0 <= Slučaj < 1. Za podatke o vseh Slovencih v spominu ni prostora. Postopek naj preišče datoteko le enkrat!

3. Radi bi napravili preproste elektronske orgle. Na razpolago imamo 60 klavirskih tipk s stikali, 5 oscilatorjev (generatorjev tona), ki jim moremo nastaviti frekvenco (višino tona), ter računalnik, ki naj krmili oscilatorje in odčitava klaviaturo. Zaradi omejenega števila oscilatorjev more hkrati zveneti največ pet tonov.

Programu so na voljo naslednji podprogrami:

```
tipka (številka)
 funkcija ima vrednost 1, če je tipka "številka" (med 1 in 60) pritisnjena, in 0, če ni.

frekv (oscilator, številka)
 podprogram nastavi "oscilator" (med 1 in 5) tako, da zveni s tonom "številka" (med 1 in 60), oziroma da utihne, če je "številka" 0.
```

a) možno je, da je pritisnjenih več kot pet tipk hkrati. V tem primeru naj utihnejo toni, ki že najdlje zvenijo, kljub temu da so ustrezne tipke še vedno pritisnjene. Drugače povedano: vsaka tipka mora takoj in vsaj za nekaj časa povzročiti zven svojega tona.

b) poenostavi algoritem tako, da ne upoštevaš pritiskov na več kot pet tipk hkrati.

OPOMBA: Napiši program, ki upravlja z elektroniko naših orgel po algoritmu a). Če imaš težave, lahko uporabiš algoritem b), ki ti prinese manj točk.

4. Ugotovi, kaj počne naslednji podprogram. Preskusi ga najprej na primeru! Razloži način delovanja (algoritem)!

```
const n = 10;
type  tabela = array [1..n] of integer;

procedure KajNaredim
 (var a: tabela; n: integer);
  var
   i, j, x: integer;
   ok : Boolean;
  begin
   for i := 2 to n do
    begin
     x := a[i];
     j := i - 1;
     ok := true;
     while ok do
      if j < 1 then
       ok := false
      else if a[j] <= x then
       ok := false
      else
       begin
        a[j+1] := a[j];
        j := j - 1
       end;
     a[j+1] := x
    end
  end { KajNaredim };
```

```
      SUBROUTINE KAJ (A, N)
C
      INTEGER N, A(N)
      INTEGER I, J, X
C
C
      DO 50 I = 2, N
        X = A(I)
        J = I - 1
10      CONTINUE
          IF (J.LT.1) GOTO 30
          IF (A(J).LE.X) GOTO 30
          A(J+1) = A(J)
          J = J - 1
          GOTO 10
30      CONTINUE
        A(J+1) = X
50    CONTINUE
      RETURN
      END
```

IV. Rezultati prvouvrščenih tekmovalcev v vsaki skupini

PO ENEM LETU POUKA RAČUNALNIŠTVA

nagrada	št. točk	tekmovalec šola
I.	97	Matjaž Kovačec Gimnazija M. Zidanška, Maribor
I.	95	Sandi Kodrič Gimnazija Bežigrad, Ljubljana
I.	94	Saša Puoko ŠC Brežice
II.	88	Borut Zalar Gimnazija, Novo mesto
II.	90	Tomaž Gornik Gimnazija M. Zidanška, Maribor
II.	90	Roman Šoper Gimnazija Novo mesto
III.	86	Tomaž Slivnik OŠ Prežihov Voranc, Ljubljana
III.	84	Boris Orehek Elektrotehniška šola, Ljubljana
III.	83	Pavel Ilija Elektrotehniška šola, Ljubljana

PO DVEH LETIH POUKA RAČUNALNIŠTVA

nagrada	št. točk	tekmovalec šola
I.	98	Ivan Pepelnjak Gimnazija Bežigrad, Ljubljana
I.	90	Tomaž Šebašek Računalniški krožek Iskra, Kranj
I.	90	Matjaž Kaufman Gimnazija Bežigrad, Ljubljana
II.	88	Aram Karalič Gimnazija Šentvid
II.	86	Igor Kukavica Gimnazija Bežigrad, Ljubljana
II.	82	Mitja Bensa Gimanzija Nova Gorica
III.	80	Alen Varšek Gimnazija Vič, Ljubljana
III.	73	Boštjan Berčič Računalniški krožek Iskra, Kranj
III.	73	Robert Bakula Gimnazija Bežigrad, Ljubljana

V. Rešitve nalog za učence po enem letu pouka računalništva

1. namesto algoritma podajamo kar program:

```
program IgraMemo;
 type
  tabela = array [1..100] of integer;
 var
  niz1, niz2: tabela;
  i, n, črni, beli: integer;

 procedure Memo (niz1, niz2: tabela;
                 var črni, beli: integer);
  var
   šeNe1, šeNe2: array [1..100] of Boolean;
   i, j: integer;
  begin { Memo }
   črni := 0; beli := 0;
   for i := 1 to n do
    begin
     šeNe1[i] := true; šeNe2[i] := true
    end;
   for i := 1 to n do
    if niz1[i] = niz2[i] then
     begin
      črni := črni + 1;
      šeNe1[i] := false; šeNe2[i] := false
     end;
   for i := 1 to n do
    for j := 1 to n do
     if šeNe1[i] and šeNe2[j] then
      if niz1[i] = niz2[j] then
       begin
        beli := beli + 1;
        šeNe1[i] := false;
        šeNe2[j] := false
       end
  end  { Memo };

begin  { IgraMemo }
 read (n);
 for i := 1 to n do read (niz1[i]);
 for i := 1 to n do read (niz2[i]);
 Memo (niz1, niz2, črni, beli);
 writeln ('   črni: ', črni:2,
          '   beli: ', beli:2);
end  { IgraMemo }.
```

2.
```
program Fructal;
 var x,y: integer;
 function Tipalo: integer; external;
 procedure Primi; external;
 procedure Spusti (x,y: integer); external;
 procedure NovZaboj; external;


 begin { Fructal }
   repeat
     for y := 1 to 3 do
       for x := 1 to 4 do
         begin
           repeat until Tipalo = 1;
           Primi; Spusti (x,y)
         end;
     NovZaboj
   until false
 end { Fructal }.
```

```
C      PROGRAM FRUCTAL
       INTEGER X, Y, STEKL
C
10     CONTINUE
       DO 40 Y = 1, 3
         DO 30 X = 1, 4
20         CONTINUE
           CALL TIPALO (STEKL)
           IF (STEKL .NE.  1) GOTO 20
           CALL PRIMI
           CALL SPUSTI (X,Y)
30       CONTINUE
40     CONTINUE
       CALL NOVZABOJ
       GOTO 10
       STOP
       END
```

3. Dolgo število moremo predstaviti v Pascalu na primer takole:

```
type
  DolgoŠtevilo =
    record
      predznak: (plus, minus);
      cifre   : array [1..20] of 0..9
    end;
```

Algoritem za seštevanje delimo v:
a) seštevanje enako predznačenih števil. V tem primeru je predznak enak kot pri obeh sumandih, cifre pa določimo takole:

```
prenos := 0;
ponovi za vseh 20 cifer, začenši pri enicah:
  a := vsota istoležnih cifer seštevancev +
       + prenos;
  istoležna cifra vsote := a mod 10;
  prenos := a div 10.
```

b) seštevanje različno predznačenih števil. V tem primeru je predznak tak, kot pri absolutno večjem od obeh seštevancev (če sta enaka, naj bo predznak vsote plus). Absolutno večjega določimo po naslednjem algoritmu:

```
poišči prvi med seboj različni istoležni
cifri (začenši na 20. mestu);
večje število je tisto, ki ima na tem mestu
večjo cifro;
```

Cifre vsote v tem primeru določimo takole:

```
posojilo := 0;
ponovi za vseh 20 cifer, začenši pri enicah:
  a := 10 + cifra absolutno večjega števila -
       - (istoležna cifra absolutno manjšega
       števila + posojilo);
  istoležna cifra vsote := a mod 10;
  posojilo := 1 - (a div 10).
```

Algoritem za odštevanje je naslednji: številu, ki ga odštevamo, obrnemo predznak in uporabimo postopek za seštevanje.

4.

```
program Srečna;
  const max = 1001;
  var
    stev: array [1..max] of integer;
    n, i, j, k, korak, indeks: integer;

  procedure Izpis;
    var i: integer;
    begin
      i := 1;
      while i < n do
        begin write (i:4); i := stev[i] end;
      writeln
    end;

  begin { Srečna }
    read (n);
    for i := 1 to n do stev[i] := i + 1;
    stev[n] := n; j := 1;
    korak := 2; indeks := 1;
    while j > 0 do
      begin
        j := 0; i := 1;
        while i < n do
          begin
            for k := 1 to korak - 2 do
              i := stev[i];
            stev[i] := stev[stev[i]];
            i := stev[i];
            if i < n then j := j + 1
          end;
        korak := stev[indeks]; indeks := korak
      end;
    Izpis
  end { Srečna }.
```

```
C       PROGRAM SREČNA
        INTEGER STEV (1001)
        INTEGER N, I, J, K, KORAK, INDEKS
C
        READ (2, 100) N
100     FORMAT (I5)
C
        DO 10 I = 1, N
          STEV(I) = I + 1
10      CONTINUE
        STEV(N) = N
        J = 1
        KORAK = 2
        INDEKS = 1
20      IF (J .LE. 0) GOTO 60
          J = 0
          I = 1
30        IF (I .GE. N) GOTO 50
            DO 40 K = 1, KORAK - 2
              I = STEV(I)
40          CONTINUE
            STEV(I) = STEV(STEV(I))
            I = STEV(I)
            IF (I .LT. N) J = J + 1
            GOTO 30
50        CONTINUE
          KORAK = STEV(INDEKS)
          INDEKS = KORAK
          GOTO 20
60      CONTINUE
C
C       SAMO ŠE IZPIS
C
        I = 1
70      IF (I .GE. N) GOTO 80
          WRITE (3, 110) STEV(I)
110       FORMAT (I5)
          I = STEV(I)
          GOTO 70
80      CONTINUE
        STOP
        END
```

VI. Rešitve nalog za učence po dveh letih pouka računalništva

1.

```
program Divide;
  var x, y, m, i: integer;
  begin
    read (x, y, m);
    write (x div y, '.');
    x := x mod y * 10;
    for i := 1 to m do
      begin
        write (x div y:1);
        x := x mod y * 10;
      end;
    writeln;
  end { Divide }.
```

2. Postopek razdelimo v dva dela. Imamo tabelo tab s 1001 elementom.

```
a) število naključno izbranih števil := 0;
   dokler je število naključno izbranih števil
   manjše od 1000 delaj
     izračunaj (Slučaj * številoSlovencev + 1)
       in mu odreži decimalke;
     če novo število še ni v tabeli tab potem
       ga vstavi na konec tabele tab;
   uredi tabelo tab s postopkom za urejanje
   tabele (glej npr. nalogo 4);
   { zanemarimo verjetnost, da imamo tako slab
     generator naključnih števil, da s tem po-
     stopkom v končnem času ne bi dobili 1000
     med seboj različnih naravnih števil }
```

```
b) številozapisov := 0;
   indeks := 0; tab[indeks] := 0;
   naredi tisočkrat;
     indeks := indeks + 1;
     naredi (tab[indeks]-tab[indeks-1]) - krat
       preberi zapis z datoteke Slovencev;
     izpiši zapis.
```

```
3. program Orgle;

 const max = 60;
 var
   i: integer;
   t: integer;      { zaporedna številka tona }
   pritisnjen: array [1..max] of Boolean;
                    { ali je tipka pritisnjena }
   oscilator: array [1..max] of 0..5;
                    { številka oscilatorja, ki
                      trenutno pripada tipki }
   ton: array [1..5] of 0..max;
                    { številka tipke, kateri
                  oscilator trenutno pripada }
   cas: array [1..5] of integer;
                  { t v trenutku, ko je oscilator
                    začel generirati ta ton }


 function Tipka (številka:integer): integer;
   external;
 procedure Frekv (oscilator,številka:integer);
   external;


 procedure Izklopi (številka: integer);
   var gen: 0..5;
   begin
     pritisnjen[številka] := false;
     gen := oscilator[številka];
     Frekv (gen,0); ton[gen] := 0;
     oscilator[številka] := 0
   end { Izklopi };


 procedure Vklopi (številka: integer);
   var
     i,gen,min: integer;
     najden: Boolean;
   begin
     pritisnjen[številka] := true;
     najden := false; gen := 1;
     while not najden and (gen <= 5) do
       if ton[gen] = 0 then najden := true
       else gen := gen + 1;
     if not najden then
       begin
         min := cas[1]; gen := 1;
         for i := 2 to 5 do
           if cas[i] < min then
             begin min := cas[i]; gen := i end
       end;
     Frekv (gen,številka);
     ton[gen] := številka;
     cas[gen] := t;
     t := t + 1;      { zanemarimo prekoračitev
                    obsega celih števil pri t }
     oscilator[številka] := gen
   end { Vklopi };
```

```
 begin { Orgle }
   t := 0;
   for i := 1 to max do
     begin
       pritisnjen[i] := false;
       oscilator[i] := 0
     end;
   for i := 1 to 5 do
     begin
       ton[i] := 0; cas[i] := 0; Frekv (i,0)
     end;
   repeat
     for i := 1 to max do
       if  (Tipka(i)=1) and not pritisnjen[i]
         then Vklopi(i)
       else if (Tipka(i)=0) and pritisnjen[i]
         then Izklopi(i)
   until false
 end { Orgle }.
```

4. Podprogram preuredi dano tabelo števil po velikosti. Prvi element tabele pusti na svojem mestu. Ta predstavlja do sedaj urejeni del tabele. Nato po vrsti jemlje števila iz tabele in jih vriva na pravo mesto v dotedaj že urejeni del tabele. Zato imenujemo ta način sortiranja urejanje z vrivanjem (insertion sorting).

VII. Zaključek

Tekmovanje je finančno podprlo precej organizacij združenega dela, ki so zainteresirane za razvoj slovenskega računalništva. Vsak tekmovalec je prejel v spomin na to tekmovanje nekaj novih slovenskih knjig s področja računalništva ter Bilten tekmovanja. Prav prijetno je presenetila Iskra Delta, ki je nagrajenim tekmovalcem omogočila dodatno brezplačno usposabljanje v njenih šolskih centrih.

Tako kot ostala dosedanja tekmovanja je bilo tudi letošnje na visokem nivoju tako organizacijsko kot tudi vsebinsko. Število tekmovalcev se v zadnjih treh letih ni bistveno spremenilo, prav tako ni opazna sprememba v kvaliteti znanja tekmovalcev. Še vedno se pojavljajo posamezne šole, kjer sta kvaliteta in obseg pouka računalništva kritična, kar je verjetno posledica nerešenih kadrovskih težav in še vedno neurejenega dostopa do računalnika za praktične vaje.

Precej sprememb si obetamo v prihodnjih nekaj letih, ko bodo dosedanje učne načrte zamenjali z novimi. Z nastopom usmerjenega izobraževanja se bodo razlike v znanju med tekmovalci še precej povečale, saj bodo imeli nekateri tekmovalci pouk računalništva vključen v redni pouk, drugi pa se bodo z njim ukvarjali le v okviru interesnih dejavnosti. Kako bomo organizacijsko in vsebinsko izvedli naslednja tekmovanja je ta hip še odprto vprašanje, vsekakor pa bo naslednje, že sedmo republiško tekmovanje srednješolcev s področja računalništva aprila ali maja 1983.

NOVICE IN ZANIMIVOSTI

****
Diskovni krmilniki (Western Digital)
****

Podjetje Western Digital je dalo na tržišče krmilnik na tiskani plošči za vinčestrske in upogljive diske za OEM ceno $ 195. To znižanje cene za tovrstne krmilnike je desetkratno glede na cene pred letom in več. Oznaka nove krmilne plošče je WD 1002, njena velikost pa je 12X15 cm. Novi krmilnik je zgrajen na novih integriranih vezjih (krmilnikih) za vinčestrski (WD 1010) in upogljivi disk (WD 279X); na plošči sta še vezje za popravljanje napak (WD 1014) in vmesni upravljalnik (WD 1015). V letu 1983 je predvidena možnost dobave novih, manjših in sodobnejših krmilnih plošč z oznakama WD 1003 in WD 1004. Naslov proizvajalca in dobavitelja je: Western Digital Corp., Components Group, 2445 McCabe Way, Irvine, CA 92714, USA.

A.P.Železnikar

****
čedalje več dvoprocesorskih mikroračunalnikov
****

Vrsta ameriških in japonskih proizvajalcev je začela proizvajati ali je najavila proizvodnjo dvoprocesorskih mikroračunalnikov: 8- in 16-bitnega. Gre za preposto in tržno upravičen kombinacijo, ki naj bi pokrila uporabo operacijskega sistema CP/M-80 in možnosti uporabe 16-bitnih operacijskih sistemov, ki so uporabljivi na IBMovem osebnem računalniku IBM PC (CP/M-86). Dvojica mikroprocesorjev Z80 in 8088 (v poštev prihaja tudi 8086) sestavlja mikroračunalnike DECa (Rainbow), Fujitsa (Micro 16) in še vrste drugih proizvajalcev. Mikro 16 ima npr. operacijske sisteme CP/M-80, CP/M-86 in MS-DOS, 128k RAMa s kontrolo parnosti, upogljive diske, barvno grafiko (640X200 delčkov), vinčestrski disk za 10M zlogov in paket, ki omogoča vključevanje v mrežo. V prihodnjem letu bo bo CP/M2.2 zamenjan s CP/M3.0 (multitasking).

A.P.Železnikar

****
Non-Von,
paralelni mikroprocesorski sistem
****

Non-Von je okrajšava za ne-von-Neumanov računalnik, pri katerem se programi ne bodo več izvajali zaporedno (po von Neumannovem principu), marveč paralelno. Gre torej za novo arhitekturo računalnikov oziroma procesorjev oziroma mikroprocesorjev.

Pred letom dni sta japonska vlada in industrija napovedali začetek novega projekta tkim. pete računalniške generacije; ta projekt sta podprli s 500 milijoni dolarjev v razdobju 10 let. Novi računalniki japonske proizvodnje naj ne bi več procesirali le podatkov temveč prevsem znanje.

Japonski izziv je že povzročil nasprotno akcijo v ZDA. ARPA (Advanced Research Projects Agency) pri ministrstvu za obrambo ZDA je sprožila alarm in predlagala posebna proračunska sredstva za nov računalniški projekt. S tem projektom naj bi ZDA poskušale izvesti protiudar z razvojno strategijo, ki je podobna japonski (za peto računalniško generacijo). ARPA je podobno kot Japonci (v letu 1981) organizirala sestanek specialistov za superračunalnike in za paralelno računalniško arhitekturo. Ugotovitev tega sestanka je, da je mogoče preseči zmogljivosti današnjih računalnikov za več kot velikostni razred le, če se izbere tkim. masivni paralelizem. Za kaj pri tem gre?

ARPA je že ugotovila pomembnost paralelnih računalniških arhitektur še posebej v povezavi z visoko stopnjo integracije vezij. Paralelna arhitektura zahteva hkratno delovanje velikega števila procesorjev pri reševanju problemov (pri izvajanju programov), kar je v principu drugače (kako drugače, še ni povsem jasno), kot je reševanje problemov z enim samim procesorjem (von Neumanov princip v tradicionalnih računalnikih). Pri tem strokovnjaki verjamejo, da bodo s paralelnimi stroji dosežene neprimerno večje zmogljivosti (inteligenca), kot bi ti bilo mogoče s stalnim izpopolnjevanjem polprevodniške tehnologije (bolj "gosta" integrirana vezja).

S paralelnimi sistemi naj bi se uresničile določene inteligenčne funkcije in na tej predpostavki temeljijo japonski podsistemi pete generacije za "upravljanje baze znanja", "stroju za sklepanje" in "inferotemporalnemu korteksnemu stroju". Stroji nove generacije bodo tako procesirali predvsem znanje, ne pa samo numerične in druge "podatke".

Za potrebe ARPA so na Kolumbijski univerzi že začeli razvijati non-von prototip (prof. David Elliot Shaw, vrednost projekta 2,5 M$) ne-von-Neumannovskega računalnika.

Non-von bo sestavljen iz približno milijon mikroprocesorjev, ki bodo razvrščeni v binarno drevo. Ta sistem bo procesiral navadne komercialne podatke in tiste numerične naloge, ki so bile doslej rezervirane za superračunalnike. Uporabljeni mikroprocesorji bodo enostavnejši od 8080A. Načrti za ta vezja so praktično gotovi in njihova fabrikacija na zahodni obali bo kmalu stekla.

Projekt non-von je deležen velike pozornosti industrije; IBM je podaril projektu 600 k$, DEC pa je dal znaten popust pri nabavi VAX opreme.

A.P.Železnikar

****
Pregled prodaje in proizodnje
mikroprocesorskih izdelkov
****

Pregled prodaje osebnih računalnikov za trileto razdobje na ameriškem tržišču je prikazan v naslednji tabeli:

Podjetje	1981/82 %	1982/83 %
Apple	39	28
Tandy	18	12
IBM	13	24
HP	7	6
DEC	2	6
Commodore	2	1
ostali	19	23

Desni stolpec je seveda napoved. Za proizvodnjo mikroprocesorskih integriranih vezij pa valjajo

in naj bi veljala naslednja razmerja (ZDA):

Podjetje	1981/82 %	1982/83 %
Intel	44,6	41,3
Motorola	21,7	24,1
Zilog	18,0	18,2
RCA	2,9	3,3
NS	1,4	2,2
Rockwell	1,4	1,9
ostali	10,0	9,0

A. P. Železnikar

****
Mikroračunalniki v letu 1983
****

IBM bo kmalu začel s proizvodnjo svojega osebnega računalnika (PC) izven ZDA in bo v letu 1983 proizvedel več kot milijon teh računalnikov. Leto 1983 bo značilno po masovni proizvodnji prenosnih (osebnih) računalnikov. Po poti, ki jo je pokazal Osborne, bodo šli predvsem tudi Radio Shack, Apple, IBM in mogoče tudi DEC (če bo sposoben dovolj hitro reagirati). CMOS tehnologija bo pri teh računalnikih v ospredju, še posebej CMOS RAM z baterijskim vzdrževanjem podatkov (mehurčni pomnilniki se ne bodo uveljavili).

Tržišče osebnih računalnikov se bo diferenciralo in nastopila bo dominacija treh do štirih proizvajalcev. IBM bo najbrž prevzel vodstvo (pred Applom in Tandyjem). Značilna delitev bo tale: računalniki za masovno potrošnjo (60 do 100$), domači računalniki za procesiranje teksta (1000 do 2000$), mali poslovni računalniki (3000 do 5000$), delovne postaje v lokalnih mrežah (5000 do 8000$) in večuporabniški sistemi (7000 do 10000$). Tretjino svetovne proizvodnje mikroračunalnikov naj bi prevzeli Japonci.

Mikropogljivi in mikrovinčestrski diski bodo vsebolj zamenjevali ustrezne mininaprave. Pojavil pa se bo tudi operacijski sistem UNIX za 8-bitne sisteme (Z80). Motorola bo začela prodajati procesor 68008 z reduciranim vodilom (8-bitno podatkovno vodilo) in z 1M zložnim pomnilnim prostorom (v 48-nožičnem ohišju). Nadalje bo začela dobavljati virtualni procesor z oznako 68010. Polna 32-bitna različica procesorja 68000 bo imela oznako 68020.

IBM bo poiskusil uporabiti procesor 68000 za implementacijo funkcije sistema 370. V tem primeru bo potrebno povečati dosedanji prostor za mikrokod v procesorju 68000 za faktor 4.

Japonci bodo še naprej kupovali sistemsko in uporabniško programsko opremo za mikroračunalnike od ameriških podjetij. To velja tako za operacijska sistema MS-DOS in CP/M, za prevajalnike in aplikativne programe. Intel pa je že znižal ceno za svoj matematični procesor 8087 od sedanjih 320 na 150$.

A. P. Železnikar

****
Alagićeve knjige doma in v svetu
****

Suad Alagić, profesor sarajevske elektrotehnične fakultete, oddelek za informatiko, sodi prav gotovo med naše najuspešnejše pisce na področju računalniških znanosti. Prva izdaja njegove knjige Principi programiranja je razprodana (založba Svjetlost, Sarajevo) in druga, dopolnjena izdaja se pripravlja za tisk.

Knjiga The Design od Well-Structured and Correct Programs (Springer, New York), ki jo je napisal skupaj s prof. Arbibom je prevedena v japonščino, prevod v ruščino pa je v teku.

Alagić pripravlja za tisk tudi rokopis knjige Relacione baze podataka, ki bo brzkone izšla pri založbi Svjetlost v Sarajevu. Alagićevemu prizadevanju za izdajanje učbenikov oziroma znanstvenih monografij s področja programiranja in organizacije relacijskih podatkovnih baz gre brez dvoma vse priznanje, saj s to svojo dejavnostjo bistveno prispeva k dvigu kvalitete na področju uporabe računalniških sistemov in računalniškega izobraževanja.

A. P. Železnikar

****
IBM ne bo šel tako daleč
****

IBM je odločno zanikal govorice, ki prihajajo iz Japnske, da se bo pridružil japonskemu projektu pete računalniške generacije. IBM izjavlja, da ni sprejel obveznosti za pomoč pri ambicioznem desetletnem japonskem projektu.

A. P. Železnikar

****
Zapad mora ostati previden
****

Zahod lahko zamudi vlak za peto generacijo, če bo preveč poudarjal pomen materialne tehnologije. Nevarnost na japonski izziv s peto računalniško genracijo je zasidrana v preveliki pozornosti za tehnologijo, ki ne upošteva transformacije tehnološke pobude (japonske) v ekonomske ugodnosti. Japonski projekt je visoko inovativen in zvišuje hkrati japonske ekonomske komponente. Pobuda je torej predvsem ekonomska in ne samo tehnološka.

Tehnologija se bo razvijala naprej kot doslej, toda poudarek je tu na ekonomski rasti, ki je za japonsko gospodarstvo značilna tudi v razdobju nazadovanja britanske in ameriške ekonomije. Japonci namreč že imajo gospodarsko infrastrukturo v MITI (Ministry for International Trade and Industry) in v NTT (Nippon Telephone and Telegraph), skozi katero se bo tehnologija filtrirala v ekonomsko sfero. Problem je v tem, da Zapad nima primerljive infrastrukture.

A. P. Železnikar

****
Videotex in teletext: razlika
****

Kakšna je razlika med videotexom in teletextom? Oba sistema imata relativno enostaven dostop do podatkov v oddaljenih bazah podatkov. Ta dostop je omogočen prek tipkovnice ali tipkala. Oba sistema imata enostavno iskalno strategijo. Uporabnik včita numerično kodirani indeks, potem pa odda kod, ki označuje želeno informacijo.

Bistvena omejitev obeh sistemov je, da nista bila predvidena za zapleteno iskanje in razpoznavanje velike količine podatkov v dovolj kratkem času. V videotex sistemu se uporabljajo numerično kodirana vprašanja, pri tem se uporablja vmesnik oziroma dekodirnik in izbirno telefonsko vezje za hitrost 75 bit/s. Podatki se potem vrnejo s hitrostjo 300 ali 1200 bit/s prek istega vezja. Po prehodu prek kodirnika se tekst prikaže na zaslonu uporabnika.

Teletext deluje nekoliko drugače. Uporabnik je tu povezan s televizijskim kanalom in ne s telefonsko linijo. Vprašanje se shrani v kodirniku sprejemnika. Teletext sistem pošilja podatke v obliki TV okvirjev, pri čemer so ti okvirji oštevilčeni. Okvirji se oddajajo nepretrgoma v ponavljajočem zaporedju. Številke okvirjev se primerjajo z ono v kodirniku. Pri ujemanju se ustrezni okvir izvleče iz zaporedja in prikaže na zaslonu.

Glavna prednost videotexa je, da je interaktiven in je tako primeren za transakcijske storitve; ko uporabnik dobi podatke o proizvodu ali storitvi, lahko naroči nakup. Videotex razpolaga tudi z večjo podatkovno bazo. Okvirji in njihovo število so sestavni del TV programa in dejansko ne predstavljajo svoboden kanal.

Velika prednost teletexta je njegova nizka cena. V Veliki Britaniji, kjer so televizijski sprejemniki izposojeni, znaša dodatek za teletext storitve manj kot $4/mesec. Vezje za teletext pa ima ceno od $ 100 do $ 150.

Tkim. prestel vidoetex vmesniki imajo ceno $ 300. Kabelska televizija omogoča uporabo celotne kabelske kapacitete in tako odpadejo omejitve za velikost podatkovne baze. V Franciji se videotex terminali dajejo zastonj, da bi se tako oblikovalo čim bolj masovno tržišče za tovrstne storitve. Digital Equipment Corp. uporablja hišno videotex mrežo že od pomladi 1982.

A.P.Železnikar

****
Nasprotniki jezika Ada
****

Tudi Ada (programirni jezik, ki ga opisujemo v tej številki) ima nasprotnike. Predvsem se pojavlja strah, da bo Ada spodrinila šolski in poučevalni standard, to je jezik Pascal. Ta bojazen je upravičena in kaj se bo zgodilo z množico učiteljev, ki se bodo zopet morali učiti novega jezika?

Jezik Ada je težavnejši (za učenje) od Pascala. Nudi pa nekatere nove koncepte, ki niso brez pomena. Toda v nekaj letih bo interaktivni pristop k računalniku tako ali drugače bistveno spremenjen, še posebej, če se bo uveljavila tkim. peta računalniška generacija.

Adi očitajo, da je tako ali drugače izpeljan iz Pascala, vendar bi lahko tudi Pascalu očitali, da je v bistvu le redukcija če že ne omejitev jezika Algol 60. Tudi za uporabo jezika Algol 60 je bilo potrebnega več programirnega znanja kot za Pascal. Res pa je, da bo jezik Ada povzročil velike dodatne stroške oziroma investicije, ki bi jih brez njega ne bilo. Tu gre za živo delo na prevajalnikih in za vgraditev določenih mehanizmov v arhitekturo mikroprocesorjev. Vsak napredek pač nekaj stane in razvoj programirnih jezikov se prav gotovo ne more ustaviti pri jeziku Pascal.

A.P.Železnikar

****
Izvedenski sistemi so specializirani
in ne vedo vsega
****

Z japonskim načrtom pete računalniške generacije se pojavljata tudi pojma izvedenskega sistema in sistema s pravili (pravilnostnega sistema). Računalniški uporabniki se sprašujejo, kaj je novega pri teh sistemih, saj že obstoječi sistemi opravljajo vrsto izvedenskih nalog hitro in zanesljivo ter vsebujejo vrsto vgrajenih pravil in algoritmov.

To je res, vendar imata pojma "izvedenski" in "pravilnostni" v okviru terminologije umetne inteligence (UI) poseben pomen. Pojem izvedenskega sistema je opredelil med drugimi tudi D. Michie kot "sistem, katerega cilj je prepričljivo opravljanje nalog svetovalca, ki izkazuje človekovo strokovno znanje na določenem področju s samorazlago sklepanja na zahtevo." Ta pojem ne pove ničesar o tem, kako naj bi bil tak sistem zgrajen, poudarja pa tkim. samorazlago. Izvedenski sistemi so specialisti le na določenem področju in se jim bržkone ne more prisojati prožna in vseobsegajoča inteligenca.

Praktično pa ima večina sistemov, ki so bili opisani kot izvedenski sistemi, le različice enega samega pristopa, in sicer produkcijskega. Zaradi tega prihaja večkrat do pojava, da se izvedenski sistemi zamenjujejo s produkcijskimi in obratno.

Produkcijski sistemi so podobni pravilnostnim sistemom. Pojem produkcije je privzet iz logike (jezikovno pravilo) in nima ničesar skupnega s proizvodnjo. Oglejmo si primer, ki ponazarja razliko med navadnimi metodami reševanja problemov v UI in pravilnostnimi metodami: to je igra šahovske končnice. Običajne metode šahovskih programov navajajo mogoče poteze, protipoteze itd. pri določenem stanju igre in ocenjujejo možne izide takih ali drugačnih potez. Program bo vselej izbral eno samo potezo, in sicer tisto, kateri je zagotovljen največji dobiček (največja verjetnost zmage). To je tkim. maksimin strategija, ljubljenka šahovskih teoretikov.

Načelno bi računalnik lahko oblikoval z analizo in ocenjevanjem prednosti vseh možnih prihodnjih potez in protipotez tkim. nepremagljivost. Vendar to praktično ni mogoče zaradi kombinatorične narave šahovske igre in časovnih omejitev in računalnik lahko izčrpno analizira le nekaj potez vnaprej. Dobra možnost za opravljanje te naloge z računalnikom obstaja v tkim. srednji igri. V končnicah je ta sposobnost gledanja vnaprej povsem nezadostna, ker relativno prazna šahovnica omogoči igralcu široko analizo položaja in uporabo kateregakoli pravila izmed številnih pravil. Uporaba določenih pravil je lahko rezultat osebnega izkustva, semiintuitivnega sklepanja in drugih izkušenj (npr. branja šahovskih priročnikov). Vzemimo npr. tole pravilo:

> Tekač in trdnjavski kmet ne moreta zmagati, če braneči kralj lahko zasede polje v kotu (na črti trdnjavskega kmeta), ki ni pod kontrolo nasprotnega tekača.

Z uporabo tega pravila začne igralec takoj pomikati kralja v varno področje. Računalnik bi moral tu predvideti dovolj veliko število potez (svojih in nasprotnikovih): tu pa se izkaže, da optimizacija z metodami UI ne ustreza, ker ji manjka preje opisana izkustvenost.

Pravilnostni pristop pomeni izognitev neustreznemu sklepanju na področju UI, ki izhaja iz širokega brskanja oziroma tipanja. Pristaši tega pristopa poudarjajo, da je verjetno bližji človekovemu načinu mišljenja. Poglejmo, kako se ta pravila vključujejo v tkim. produkcijski sistem. Kaj je produkcija?

Produkcijo lahko opišemo kot dvojico (pogoj, akcija) oblike:

```
IF pogoj THEN OPRAVI akcija
```

Komponenti produkcije imenujemo često leva (pogoj) in desna (akcija) stran. Produkcijski sistem je sestavljen iz zbirke produkcij oziroma pravil. Druga sestavina produkcijskega sistema

je podatkovna baza trenutno veljavnih dejstev in v to bazo se dostopa z namenom, da se ugotovi, ali dana produkcija ustreza, tj. ali se ujema njen pogoj. Stanje podatkovne baze se bo vobče spreminjalo kot posledica delovanja produkcij. (V jezikih delujejo produkcije nad stavčnimi oblikami - besedami-.)

Tretja sestavina produkcijskega sistema je način uporabe produkcij. Ta odloča, katere produkcije so umestne, izbere eno izmed njih in jo uporabi (izvrši, izvede). Ta postopek se potem ponovi vselej z novimi stanji.

Za razumevanje delovanja produkcijskega sistema je pomembno, da krmiljenje (uporaba produkcij) ne poteka po neki vnaprej dani listi produkcij. Produkcijski sistem tako ni preprosto lista IF...THEN... stavkov. Delovanje produkcijskega sistema si lahko zamislimo kot množico produkcij, ki čakajo, da bodo uporabljene. Kakor hitro je začetna množica dejstev znanih, se začne ocenjevanje levih produkcijskih delov in izberejo se ustrezne produkcije (za potencialno uporabo). Dejanje izvršitve lahko upošteva več faktov (dejstev) in izvrši se lahko več produkcij vse dotlej, dokler ni bilo deducirano vse, kar bi deduciral človeški izvedenec.

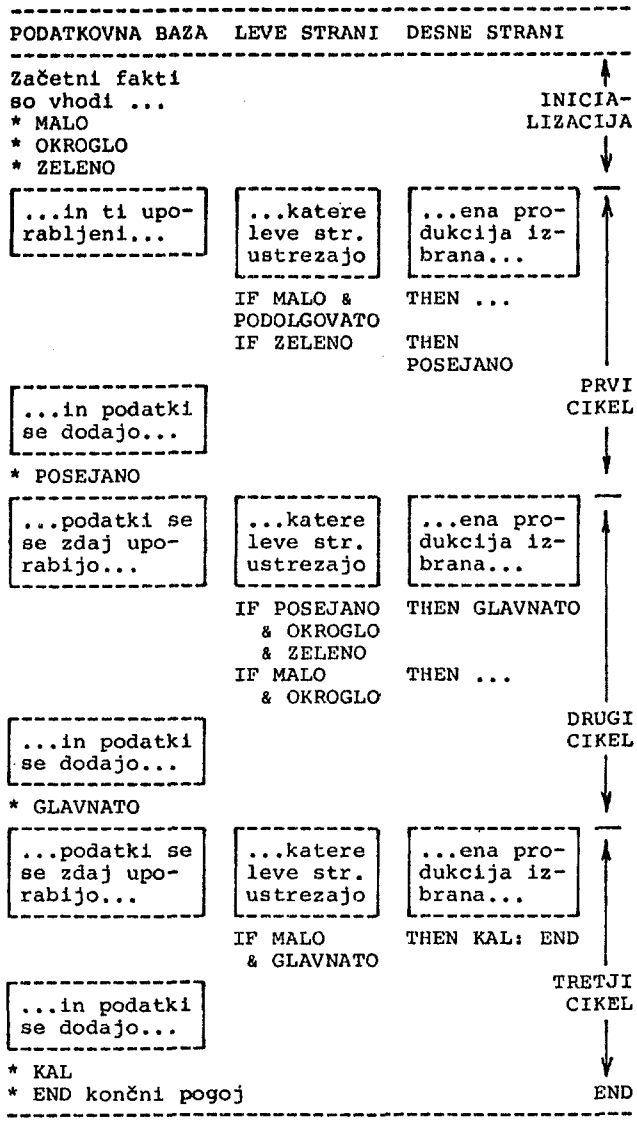
Pravkar opisani postopek je neke vrste "dedukcijski plaz". Trivialni primer takega postopka je dan na sliki.

Vitalni vidik produkcijskega sistema je, da mu je mogoče posredovati atome strokovnega znanja v malih količinah brez popravljanja logike uporabe (izvrševanja pravil). Ta princip je v nasprotju z običajnim pristopom, ko vstavljamo novo znanje v računalniški sistem in moramo to vstavitev izvršiti v pravi del programa ter popraviti še obdajajočo okolico. Drugo vprašanje, ki se pojavi, je, kako sodeluje sistem z uporabnikom med izvrševanjem. Ali zahteva vrednosti za določeno podatkovno množico in ali dinamično spreminja svoja vprašanja glede na rezultat prejšnjega vprašanja? Ali daje pojasnila na zahteve, da bi zadostil Michiejevi opredelitvi izvedenskega sistema?

Druga množica sprememb se nanaša na problem, kako sistem uspešno obvladuje nedoločnost (negotovost, dvomljivost, spremenljivost, nepreračunljivost). Pri tem imamo vrsto področij strokovnega znanja, kjer je načelo IF x THEN y boljše od načela IF x THEN VERJETNOST(y) IS z. Tu je potrebno obrniti delovanje sistema in uporabiti veriženje nazaj. Namesto, da začnemo s fakti in poskušamo doseči sklep, želimo vedeti, kateri fakti naj bi veljali, da bi dosegli dani (hipotetični) sklep.

Oglejmo si dva izvedenska sistema z različnima področjema uporabe. Prvi primer poudarjajo zlasti zagovorniki UI in ponazarja splošen tip problema, ki bi ga lahko imenovali diagnostični ali taksonomični (sistematizacijski, klasifikacijski). Izvedenec je Mycin, razvit je bil v Stanfordu, identificira pa bakterije v krvnih in urinskih vzorcih in svetuje možna antibiotična zdravljenja. Deluje na osnovi vodenja dialoga (vprašanja in odgovori) z uporabnikom, dokler ni sposoben postaviti diagnoze in napisati recept.

Značilnost tega sistema je, da uporablja verjetnost in prikaže listo bakterijskih kandidatov skupaj z ocenami (torej ne samo en sam odgovor). Zaradi nezanesljive narave tega področja je to vsekakor potrebno. Druga značilnost sistema je, da si pomaga z veriženjem nazaj. To je predvsem zaradi tega, ker tudi človeški izvedenci tako postopajo, ko izločajo sumljive predpostavke že na začetku dialoga brez postavljanja vprašanj z upoštevanjem simptomov in



Slika. Delovanje enostavnega produkcijskega sistema. Ta trivialni produkcijski sistem prikazuje uporabo strokovnega znanja s področja zelenjadarstva. Sistem začne delovati le z nekaj začetnimi fakti in se gradi z dodatnimi fakti po načelu "preizkusi, izberi in uporabi" v posameznih ciklih. Izbira enega samega pravila za uporabo ni vselej tako naključna kot je na tej sliki.

preiskovalnih rezultatov. Tretja lastnost tega sistema je, da lahko uporabnik vprašuje o verigi sklepanja, torej lahko dobi pojasnila na zahtevo. Te tri lastnosti so glavni vzrok, da je Mycin izredno priljubljena tema za pisce na področju UI.

Drugi primer zajema povsem drugo področje in je zanimiv tudi za navadnega bralca. Tkim. R1 produkcijski sistem sta razvila Carnegie-Mellon univerza in DEC, predstavlja pa izvedenca za DECove računalniške konfiguracije VAX. Vhodni podatki (večkrat nepopolni) predstavljajo tkim. BAX listo sestavnih delov, ki jo R1 pretvori v konkretno delujoči VAX sistem.

Sistem R1 odloči, katere enote (sestavine) imajo funkcijsko zvezo, kam naj se enote namestijo v ohišjih (kabinetih), določi pa tudi relativni

položaj ohišja in pripadajoče kable. Čeprav bi pričakovali, da je ta problem mogoče reševati z navadnimi metodami obdelave podatkov, je DEC po večletnih izkušnjah ugotovil, da je uspešna pot z obdelavo podatkov težavna in da opravlja sistem R1 to nalogo bolj uspešno.

Kateri so torej bistveni problemi izvedenskih sistemov? Prvi problem so kadrovski viri: ni veliko analitikov in programerjev, ki bi začeli izdelovati izvedenski sistem brez dodatnega študija in motivacije. Drugi problem se pojavlja pri prenosu idej na vodstvo brez posebnega dvigovanja pričakovanj: predloženi izvedenski sistem naj bi se tudi "prodajal," tako da ga ne bi občutili le kot investicijo, marveč kot veliko upanje prihodnjega uporabnika. Tretji problem je programska oprema, ki mora biti napisana v programirnem jeziku, za katerega je vodstvo obdelave podatkov le redko slišalo (Lisp, Prolog) in ta jezik navadno ni mogoče uporabljati na obstoječih instalacijah.

Tudi zajetje pravil predstavlja dodatno oviro. Ker mora imeti računalniški izvedenski sistem pravila človeških izvedencev, morajo biti ti izvedenci usposobljeni za prenos teh pravil v sistem. Industrija doslej ni pogosto uporabljala izvedenskih sistemov, in sicer predvsem zaradi pomanjkanja znanja in obveščenosti; potrebe po takih inovacijah pa vsekakor so in usposobljenih izvajalcev praktično še ni.

A.P.Železnikar

****
Kaj je večuporabniški sistem ?
****

Uvod

Čeprav se bo zdel ta naslov marsikomu odveč, se v praksi največkrat srečujemo s popolnim nepoznavanjem pojma "večuporabniški sistem," in to pri "strokovnjakih," ki so nekako srečno pristali na področju računalništva, kjer ribarijo in zastrupljajo strokovno ozračje.

Večuporabniški (mikro)sistem je posebna struktura, s posebnimi lastnostmi. Ni moč razglasiti za večuporabniški sistem nekaj, kar ne izpolnjuje določenih pogojev (standardov). Sodobni večuporabniški sistem združuje lastnosti enoprocesorskega večuporabniškega sistema z lastnostmi slabo ali dobro povezane mreže. Večuporabniško oklje je lahko organizirano na več načinov.

Klasičen večuporabniški sistem je enoprocesorski, ima velik pomnilnik in ustrezno število V/I vrat. Uporabniki sistema si delijo procesorski čas (timesharing). Čeprav je zamisel enostavna, je uresničitev takega sistema težavna: časovnik povzroči vsakih nekaj mikrosekund prekinitev, katere namen je odvzem procesorja trenutnemu uporabniku in njegova dodelitev drugemu uporabniku. V dvouporabniškem sistemu se procesor izmenoma dodeljuje enemu in drugemu uporabniku. Pri več uporabnikih se procesor dodeljuje krožno in vsakič opravi nalogo za posameznega uporabnika. Sistem lahko dodeljuje procesor tudi neenakomerno posameznim uporabnikom, upoštevaje uporabniško prednost.

Programska oprema za klasičen večuporabniški sistem je večinoma napisana za en sam procesor. Operacijski sistem je tako tesno povezan s krmiljenjem sistemskih virov. Učinkovitost takega sistema je pretežno odvisna od materialne zgradbe sistema. Ta zgradba lahko povsem ustreza enouporabniškem sistemu, v večuporabniškem pa postane neustrezna. Zgornja meja za današnje mikroračunalniške enoprocesorske (16-bitne) večuporabniške sisteme je pri štirih uporabnikih. Ta meja je dosežena tedaj, ko se pri določeni aplikaciji izdatno zmanjša hitrost ali zmogljivost, ko dodamo še enega uporabnika. Pri neustrezni materialni opremi se to lahko zgodi že pri dvouporabniškem sistemu. Ta meja je tedaj odvisna od računalno intenzivnega okolja, tj. od uporabnostnega področja.

Večuporabniški sistem lahko dobimo tudi z mrežno povezavo več mikroračunalnikov. V mreži je mikroračunalnik v vozlišču in ima sam določeno stopnjo inteligence (samostojnosti). Kombinacija terminala, mikroračunalnika in sekundarnega pomnilnika v vozlišču se imenuje delovna postaja in ta zmore opravljati normalne računalniške naloge, sprejema in pošilja pa lahko tudi mrežna sporočila.

Porazdeljena obdelava

Mrežna obdelava je le ena oblika porazdeljena obdelave. Pojem porazdeljene obdelave zahteva porazdelitev procesorjev v računalnem okolju. V primeru mreže so procesorji medsebojno tudi oddaljeni in niso medseboj tesno (intenzivno) povezani. Tako oddaljeni procesor v mreži ne more krmiliti neposrednih operacij drugega procesorja. Procesorji so namreč v mreži na določen način medsebojno ločeni (funkcijsko izolirani). Takšen sistem imenujemo slabo povezan.

Druga oblika porazdeljene obdelave temelji na povezavi več procesorjev v eni sami napravi (računalniku). V tem primeru krmili glavni (mojstrski) procesor akcije pomožnih (pomočniških) procesorjev. V enouporabniškem okolju bi se deli računalne naloge razdelili na pozamezne procesorje in vsak izmed njih bi svoj del opravil hkrati (istočasno) z drugimi procesorji. Tako bi se izvršitev naloge znatno pospešila. Takšen proces obdelave bi imenovali paralelna obdelava, ker smo uporabili več procesorjev za izvršitev ene same naloge in so procesorji delovali paralelno (časovno in prostorsko).

Veliki računalniki delujejo paralelno in dosegajo tako veliko podatkovno pretočnost. Te metode se uporabljajo tudi v integriranih vezjih. Procesor 8086 ima dva notranja procesorja: eden opravlja operacije z vodilom, drugi pa dekodira ukaze in jih izvršuje. Ta konstrukcija povzroči merljivi učinek pri celotni zmogljivosti. Procesor 80286 ima že štiri notranje procesorje, ki si delijo naloge. Učinek v povečani zmogljivosti je izreden. V tej notranji mreži imamo tesno povezavo med procesorji: mojster vodi svoje pomočnike.

Paralelna obdelava v mikroračunalniških sistemih povečuje pretočnost večuporabniških sistemov tako, da prirejuje procesor in neodvisen pomnilnik posameznemu uporabniku. Tako postane največja sistemska zmogljivost zelo visoka in je omejena samo še s hitrostjo množičnih pomnilnih naprav (sekundarnih pomnilnikov). Taki sistemi delujejo kot mreže, v katerih so procesorji neodvisni; v nekaterih sistemih so slabo, v drugih pa dobro medsebojno povezani.

Materialna večuporabniška arhitektura

Zniževanje zmogljivosti v večuporabniškem sistemu je mogoče preprečiti na več načinov. Z uporabo hitrega pomnilnika (RAM), visoko zmogljivih diskovnih krmilnikov in kompleksnih hitrih centralnih procesorjev je to prav gotovo mogoče. Materialna zgradba mikroračunalnika mora tedaj ustrezati večuporabniški arhitekturi. Kako je mogoče povečevati zmogljivost večuporabniškega sistema ?

Hitri procesor in pomnilnik oblikujeta prvi pogoj. Diskovni krmilniki morajo biti podprti z neposrednim pomnilniškim dostopom (DMA), da se

poveča sistemska pretočnost. V/I kanali morajo biti kar najbolj učinkovito prilagojeni večuporabniškemu okolju. Standarizirano vodilo (npr. IEEE 696/S-100) in določena modularnost sta nujna, da bi bilo mogče pokriti široko področje večuporabniških rešitev.

Kot že povedano, lahko čas hitrega centralnega procesorja delimo med uporabnike. Večkrat je smiselno uporabiti modul z dvema procesorjema (npr. 8085/8088, Z80A/8088, Z80A/8086, Z800/80186, Z800/80286), tako da je moč izvajati 8- in 16-bitne programe (MP/M-80, MP/M-86). Pri tem se npr. 8-bitne aplikacije lahko izvajajo kot opravila (tasks). S tako procesorsko enoto je mogoče obvladati v odvisnosti od aplikacije tudi 10 in več uporabnikov. Opisani sistem pa je še vedno klasičen in ne upošteva razvoja računalniške in procesorske arhitekture in njenih možnosti v zadnjih letih.

Multipleksni kanali

Podjetje IBM je že pred leti razvilo poseben tip podatkovnega kanala, ki ga je imenovalo multipleksorski (multipleksni) kanal. Ta kanal je v bistvu ločen, mali računalnik, namenjen povečanju hitrosti V/I operacij. Ta kanal krmili podatkovni pretok med sistemskim pomnilnikom (RAMom) in zunanjim okoljem. Kanal, ki opravlja storitve za eno samo V/I napravo (npr. za terminal), se imenuje izbirni kanal. Multipleksni kanal opravlja storitve za več V/I naprav s prenosom podatkov za različne naprave, ki jih krmili.

Multipleksni kanal za določeno vodilo mora vsebovati hitri procesor (npr. 8085 ali Z80B s taktno frekvenco 6 MHz), 16k zlogov RAMa, 8k-zložni ROM ali EPROM, hitri prekinitveni krmilnik (npr. 8259A) in takoimenovani vmesnik za začasni mojstrski dostop (ZMD), za povezavo z glavnim sistemskim vodilom. Ta multipleksni kanal (mikroprocesorski podsistem) deluje paralelno z glavno procesno enoto na sistemsko vodilo. Kanal ima tu funkcijo gosta (pomočnika), glavna procesorska enota pa funkcijo gostitelja (mojstra). ZMD omogoči kanalu komunikacijo s poljubno pomnilno lokacijo ali V/I vrati na sistemskem vodilu, prekinitveni krmilnik pa pošilja naslovne vektorje na lokalno vodilo. Tudi gostitelj lahko vzbudi pozornost gosta, gost pa lahko pri gostitelju sproži prekinitev.

Namen multipleksnega kanala je razbremenitev gostiteljskega procesorja s prevzemom opravil, sproženih s sistemskimi prekinitvami. Kaj nastopi v normalnem sistemu, ko V/I modul sproži prekinitev zaradi znaka, ki čaka v terminalu? Centralni procesor je izvajal opravilo za trenutnega uporabnika, ko je drugi uporabnik pritisnil na tipko. V/I modul je sprejel znak iz terminala in sprožil eno izmed vektorskih prekinitev, tj. izvajanje določene (storitvene) rutine. Centralni procesor mora prenehati s trenutnim opravilom, rešiti mora celotno podatkovno stanje v sklad in skočiti v izvajanje storitvene rutine. Ta rutina prebere znak iz V/I modula in ga vstavi v vmesnik. Storitvena rutina lahko znak preizkusi, da ugotovi, ali je ta znak krmilni znak, kot je npr. pomik nazaj ali pomik valja. Ko je bila sprejeta določena vrstica (na koncu te vrstice je pomik valja), se lahko postavi posebna zastavica, tako da določeno opravilo ve, da je vhod pripravljen za obdelavo. Nastaviti se morajo tudi kazalci v vmesnik skupaj s statusnim zlogom, ki vsebuje število shranjenih znakov v vmesniku. Potem storitvena rutina vrne krmiljenje rutini, ki naloži stanje ob prekinitvi in nadaljuje z izvajanjem prekinjenega opravila.

Ta primer obdelave enega samega znaka lahko zahteva več sto procesorskih izvajalnih ciklov. Ta čas je bil tako odvzet prvotnemu trenutnemu opravilu in to opravilo je bilo s tem upočasnjeno.

Oglejmo si ta postopek z multipleksnim kanalom: pojavi naj se podobna prekinitev, na katero pa bo odgovoril kanalni procesor namesto gostiteljskega, tako da gostiteljski procesor izvaja trenutno opravilo nemoteno. Kanal odvzame ob prekinitvi le en cikel vodila, da včita znak iz V/I modula. Kanal preizkusi znak in potem ustrezno ukrepa. Nastavijo se vmesniški kazalci in znak se lahko vpiše v vmesnik gostiteljevega pomnilnega prostora (pri tem se zopet odvzame le en cikel vodila) ali pa v vmesnik kanala. Zopet se lahko postavi zastavica, če se je pojavil znak pomika valja (konca vrstice), pri tem se odvzame en cikel vodila.

V primeru kanala imamo paralelno obdelavo prekinitve s kanalom in gostiteljskim procesorjem, pri čemer je kanal odvzel le nekaj ciklov vodila pri izvajanju opravila namesto nekaj sto ciklov v nekanalnem primeru. Sedaj je jasno, da lahko multipleksni kanal občutno poveča pretočnost večuporabniškega sistema. Kanal lahko seveda opravlja še vrsto drugih opravil v sistemu (npr. izpisovanje na tiskalnik) in v istem sistemu se lahko uporablja več multipleksnih kanalov.

Pomočniki in mojstri

Multipleksni kanal je tkim. osprednji/končni (mejni, front-end) računalnik, ki pohitri delovanje enoprocesorskega večuporabniškega sistema; toda v vrsti primerov tudi povečanje operativne hitrosti ne zadostuje. V nekaterih primerih je dodelitev posebnega procesorja vsakemu uporabniku edina taka rešitev, ki lahko zagotovi zahtevano pretočnost, hkrati pa je koristno obdržati prednosti tesnejše povezave več procesorjev. Oglejmo si sistem, ki izpolnjuje te zahteve.

Znano je, da obstajajo vodila (npr. IEEE 696/S-100), ki omogočajo delovanje in povezavo več procesorjev. Vsak S-100 sistem pa mora imeti mojstrski procesor, ki krmili delovanje sistema kot celote. Ta način se imenuje sistem z stalnim (permanentnim) mojstrom. V večini primerov opravlja to nalogo centralni procesni modul, ki ga poznamo tudi v navadnih mikrosistemih. Sistem pa lahko ima tudi več začasnih mojstrov (npr. 16), ki zahtevajo krmiljenje vodila od stalnega mojstra. Prednostni sistem odloča o tem, kateremu od začasnih mojstrov bo dodeljeno krmiljenje vodila. Postopek zahtevanja in dobivanja krmiljenja vodila in pripadajoče izvajanje ciklov vodila z začasnimi mojstri se imenuje "začasni mojstrski dostop" (ZMD). ZMD se razlikuje od DMA (neposrednega pomnilniškega dostopa) v tem, da lahko začasni mojster dostopa v pomnilnik ali pa opravlja V/I. Tako se npr. diskovni krmilnik (poseben V/I modul) lahko uporabi kot pravi začasni mojster v skladu z zahtevami standarda IEEE 696. Ta modul zahteva uporabo vodila od stalnega mojstra ob upoštevanju prednosti, ki jo predpisuje standard.

Pomnilniški in V/I moduli na vodilu pa so znani kot pomočniki (slaves), ker so podrejeni zahtevam in storilnostnim mehanizmom mojstrov - stalnega in začasnih. Vsak mojster (stalni ali začasni) lahko komunicira prek vodila s pomočniki. Povezovalno vezje na vodilo je veliko bolj zapleteno za mojstra, kot je za pomočnika.

Pojavi se torej lahko tudi potreba za uvedbo enega procesorja na enega uporabnika: v tem primeru bomo te dodatne procesorje imenovali pomočniški (pomožni procesorji) zaradi dveh razlogov. V vsakem primeru bomo imeli močni mojstrski procesor, ki bo nadzoroval delovanje sistema. Pomožni procesorji pa bodo npr. lahko delovali kot pomočniki skladno s standardom

IEEE 696 (ne kot začasni mojstri).

Obstaja vrsta razlogov, da vzamemo za pomožne procesorje pomočnike (slave) namesto trenutnih mojstrov. Povezovalno vezje na vodilo je pri pomočniku manj zapleteno (zavzame manj prostora na plošči) kot pri začasnem mojstru. Pri obravnavi posebnosti pomočnikov bo to postalo bolj jasno.

Začasni mojster ima dostop v pomnilnik in na V/I lokacije glavnega vodila. Če vzamemo začasne mojstre namesto pomočnikov, lahko en kanal neodvisno pokvari delovanje drugega kanala in povzroči tako porušenje kanala ali celo sistema. Zaščita uporabnika pred porušitvijo njegovega kanala prek drugega kanala oziroma pred porušitvijo sistema je bistvena. Pokazali bomo, kako se ta zaščita doseže z uporabo pomočnikov v kanalih.

Druga zahteva za uporabo pomočnikov namesto začasnih mojstrov se lahko pojavi zaradi omejitve števila začasnih mojstrov, ki je pri standardu IEEE 696 omejeno s 16. To število je lahko preseženo, če dodelimo vsakemu uporabniku svoj kanal. Število začasnih mojstrov pa seveda ne pomeni števila možnih uporabnikov: teh je lahko veliko več. Tako bi naj bil diskovni krmilnik vselej konstruiran kot začasni mojster, saj predstavlja npr. vinčestrski disk skupni sistemski vir (v tem je ravno smisel večuporabniškega sistema). Ker je takih skupnih virov lahko še več, bi omejitev na 16 začasnih mojstrov lahko znatno znižala število možnih uporabnikov.

Naposled se pojavi tudi vprašanje programske opreme za tako koncipirani večuporabniški sistem. Vodenje sistema z večjim številom začasnih mojstrov je veliko bolj težavna naloga kot razvoj programa za vodenje enega samega, močnega centralnega procesorja, ki uporablja medprocesorske komunikacije.

Oglejmo si tale primer! V sistemu želimo imeti 16- in 8-bitne procesorje, plošča z mojstrskim procesorjem pa je npr. lahko različnih tipov, kot so 8086/8087, 68000, 16032, 80286, da naštejemo le najmarkatnejše. Osprednji/končni procesor (pomočnik) naj bo dvoprocesorski, npr. tipa 8085/8088, tako da je mogoča njegova 8- in 16-bitna uporaba. Nadalje imejmo npr. tudi pomočnika tipa Z80B, ki ga uporabimo v 8-bitnem vozlišču za funkcijo en procesor na enega uporabnika. Ta modul imenujmo kratko PPE-Z (pomožna procesna enota, Z za Z80). Ta modul naj ima procesor Z80B (takt 6 MHz), 192k zlogov dinamičnega RAMa (DRAM), dvoje serijskih vrat tipa RS-232C, ena pozornostna vrata za zbuditev pozornosti s strani gostitelja (glavnega mojstra), sistem za sprožitev prekinitve s strani PPE-Z pri mojstru, 2k zlogov EPROMa z začetnim programom in 4k zlogov hitrega, statičnega, dvovratnega pomnilnika (RAM) za komunikacijo med glavnim vodilom in PPE-Z.

Podorbnosti modula PPE-Z

Procesor Z80B in 64k zlogov DRAMa oblikujejo izvajalski stroj za 8-bitna opravila. Dvoje serijskih vrat omogoča uporabo terminala in tiskalnika za enega uporabnika. Terminal in tiskalnik sta tako lokalni napravi, ki ne obremenjujeta glavnega vodila in dejansko povečujeta zmogljivost tega vodila.

Pojasnimo tudi uporabo dvovratnega RAMa (tj. pomnilnik, ki je dostopen prek dveh vrat). V ta pomnilnik lahko dostopata dva procesorja. Eden izmed njiju je Z80B, drugi pa je procesor kateregakoli mojstra (stalnega ali začasnega). Dvovratni pomnilnik se v PPE-Z uporablja za prenos podatkov v ali iz zunanjega sistema. Dvovratni pomnilnik je zamejen v intervalih po 4k zlogov v 16M-zložnem naslovnem prostoru sistemskega vodila (S-100). Interno lahko dvovratni RAM prekrije poljubni 8k-zložni odsek DRAMa ali EPROMa. Diskovni krmilnik (začasni mojster) in stalni mojster (gostitelj) lahko prenašata podatke neposredno v dvovratni RAM, kar povečuje pretočnost.

Modul PPE-Z lahko sproži prekinitev v gostiteljskem sistemu in ta lahko opozori PPE-Z prek pozornostnih vrat.

Močni pomočniki

Zaradi potreb uporabnika se lahko pojavi zahteva, da mora biti tudi pomočnik močnejši, npr. 16-bitni. Sistem potrebuje v določeni točki močnejše vozlišče. Za tako vozlišče lahko npr. izberemo procesorski tip 8086/8087, ki je močan v matematičnih operacijah. Procesor 8087 ima številne registre, dolge 80 bitov. Ti registri se morajo shranjevati v sklad ob prekinitvah in nove vrednosti se morajo nalagati v registre, če različni uporabniki uporabljajo ta procesor, ki se nahaja npr. samo na modulu stalnega mojstra. Zato je včasih bolj smotrno, da ima določen uporabnik procesor 8087 na svojem modulu.

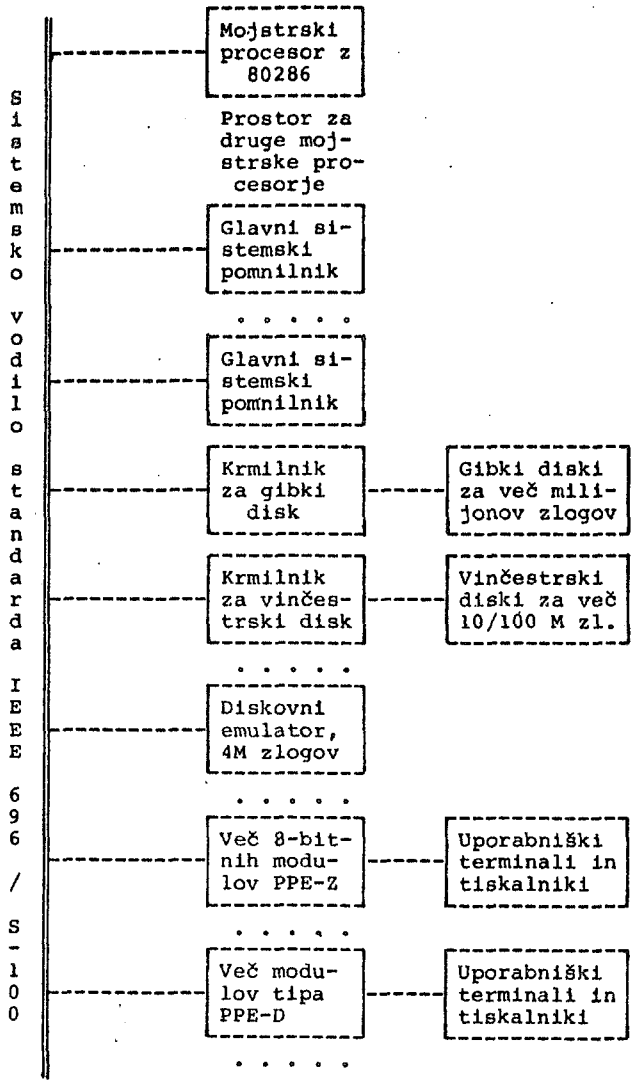
Ker se večkrat pojavi potreba za uporabo 8- in 16-bitne programske opreme, lahko uvedemo modul z dvema procesorjema in z matematičnim procesorjem, ki deluje kot visokozmogljivo pomočniško vozlišče. Tako imamo modul, ki ga označimo kot PPE-D (pomočniška procesna enota z dvojnim procesorjem) in ta vsebuje procesorje Z80B (6 MHz), 8088 (8 MHz) in podstavek za procesor 8087. Ima pa še 192k zlogov DRAMa (16-bitni programi so navadno bolj obsežni), dvovratni pomnilnik, EPROM in dvoje serijskih vrat (podobnost s PPE-Z!. Ta modul predstavlja tako veliko računalno moč, ki je dana uporabniku v večuporabniškem sistemu.

Moč in uporabniška zaščita: 80286

Vsak od preje navedenih modulov se lahko uporablja kot mojstrski modul (tipi 8085/8088, 8086/8087, 68000, 16032), ki upravlja sistem z več pomočniškimi procesorji oziroma moduli, katerih število je omejeno le z razsežnostjo kabineta. K tem, naštetim modulom pa želimo imeti še eno procesorsko ploščo (modul), ki naj bi bila osnova večuporabniškemu mikroračunalniškemu sistemu z veliko močjo (zmogljivostjo).

Procesorska plošča z Intelovim 16-bitnim super mikroprocesorjem 80286 je za to funkcijo izredno pripravna (glej sliko večuporabniškega mikrosistema). Procesor 80286 lahko naslovi 16M zlogov RAMa (v 1G-zložnem virtualnem pomnilnem prostoru), ima vgrajen mehanizem za popolno pomnilniško preslikavo in zaščito in lahko preklaplja med posameznimi opravili zelo hitro. Procesor 80286 lahko preklaplja iz enega opravila na drugo v 17 do 22 mikrosekundah; Motorolin procesor 68000 porabi 150 μs, njegova izboljšava 68010 pa 110 μs. Procesor 80286 bo izvajal kod, napisan za 8086/8088, vendar 4-krat hitreje, kot se ta kod izvaja na 8086 pri enaki taktni frekvenci. Procesor 80286 uporablja taktno frekvenco 10 MHz, dodamo pa mu lahko tudi matematični procesor 80287 in vezje, ki omogoča uporabo 8- ali 16-bitnega pomnilnika.

Najpomembnejša lastnost procesorja 80286 je prav gotovo vgrajeni mehanizem za zaščito pomnilnika. V enoprocesorskih večuporabniških sistemih je namreč izredno težko, če ne nemogoče, doseči tako stopnjo zanesljivost sistema, da določen uporabnik ne more porušiti sistema drugega uporabnika ali pa celotnega sistema. Ker v enoprocesorskem sistemu en procesor vse sam opravi, ima tudi dostop do vsake točke. Tako lahko doseže vsak bit v pomnilnih območjih



Slika. Tipična večuporabniška sistemska konfiguracija, ki uporablja stalnega mojstra tipa 80286 (ali po potrebi druge stalne mojstre tipa 8086/8087, 68000, 16032), več modulov glavnega sistemskega pomnilnika (DRAM), krmilnike za gibke in vinčestrske diske (začasni mojstri), hitre diskovne emulatorje za več milijonov zlogov (začasni mojstri ali pomočniki) ter različne tipe pomočniških (front-end) procesorjev (po načelu en procesor za enega uporabnika).

vseh uporabnikov. Procesor 80286 pa zagotavlja visoko stopnjo meduporabniške zaščite in posamezni uporabnik lahko poruši le svoj ali pa celoten sistem.

Prednost uporabe pomočniških procesorjev (ne mojstrskih) je v tem, da lahko le mojster neposredno krmili komuniciranje v sistemu. Mojstrski procesor (stalni mojster) tudi nikoli ne izvaja programa za določenega uporabnika; njegova naloga je izvajanje operacijskega sistema. V enoprocesorskem sistemu izvaja procesor uporabniške programe in operacijski sistem. V večprocesorskem sistemu je relativno lahko mogoče omejiti sistemsko preobsežne učinke posameznih pomočniških procesorjev. Seveda pa lahko zahteven uporabnik poruši še vedno svojega pomočnika, ne more pa vplivati na porušenje drugih pomočnikov; torej po taki porušitvi sistem še vedno deluje.

Povzetek bistvenega

Prikazana zamisel večuporabniške arhitekture združuje najboljše s področja mrežnih in enoprocesorskih sistemov. Ta arhitektura upošteva mrežo visoko zmogljivih pomočniških procesorjev, ki imajo najboljše lastnosti slabo in dobro povezanih mrež. Ta mreža je organizirana z uporabo enega samega procesorja visoke zmogljivosti.

Ker predloženi večuporabniški sistem temelji na standardu vodila IEEE 696/S-100, je mogoče mešati, prilagojevati in kombinirati sestavine v tem sistemu; to velja tudi za izbiro in modifikacijo različnih operacijskih sistemov (od CP/M-86 do UNIXa). Sistem lahko najprej konfiguriramo kot enouporabniški sistem (z uporabo ustreznega modula), kasneje pa ga dogradimo z mojstrskim modulom (80286) in dodamo še druge pomočniške module, da tako povečamo pretočnost. Za nadaljno povečanje pretočnosti lahko dodamo še modul za emulacijo diska z večmilijonskim (zložnim) pomnilnikom (glej sliko). Ko je pretočnost tega sistema kritična, dodamo še dodatne pomočniške procesorje (module) po načelu, vsakemu uporabniku svoj procesor; tistim uporabnikom, ki imajo velike zahteve, priredimo modul PPE-D (8- in 16-bitni procesor na modulu), ostalim pa damo module PPE-Z. Tako dosežemo maksimalno pretočnost sistema, ki je ne moremo več povečevati.

Večuporabniški sistemi v mreži

Opisane večuporabniške sisteme lahko povežemo v mrežo. Trenutno je na razpolago več mrežnih shem oziroma operacijskih sistemov. Pri tem se uporabljajo tudi sinhroni serijski kanali. Uporaba kombinacije 80286 in PPE-D kot vozlišča v mreži je očitno dovolj močna. Ta prispevek naj bi oblikoval podlago za arhitekturo večuporabniškega sistema, ki je sam zgrajen po mrežnem načelu, hkrati pa naj bi bil kot sistem povezan v širšo mrežo več- in enouporabniških sistemov.

A.P.Železnikar

****

Posebne novice

****

Pojavljajo se spekulacije, da bo podjetje IBM uvedlo novo, dopolnjeno različico svojega osebnega računalnika, v kateri bo uporabilo mikroprocesor Intel 80186 ali 80188. Ta dva procesorja bi lahko znatno izboljšala sistemske zmogljivosti in znižala število integriranih vezij za približno 20 ob znižanju cene osnovne enote pod $ 1000. Ker je Intel dal na tržišče tudi procesor 80286 (izboljšana različica procesorja 8086), katerega zmogljivost je enaka ali boljša od zmogljivosti Motorole 68000, se predpostavlja, da bo IBM verjetno uporabil ta procesor. Ta modifikacija bi lahko zboljšala sedanjo zmogljivost osebnega računalnika (PC) za faktor 6 pri dvojni ceni. IBM hoče svojemu osebnemu računalniku dodati možnost obdelave več opravil (multitasking).

Podjetje Compupro Systems (Box 2355, Oakland Airport, CA 94614) je za svoje osebne računalnike predstavilo modul z mikroprocesorjem NS 16032 in modul z 80286. Oba modula delujeta s taktno frekvenco 10 MHz in sta prilagojena na vodilo IEEE 696/S-100. Hkrati je podjetje Digital Research začelo z razvojem različice operacijskega sistema CP/M-86 za procesor 80286 in različice operaciskega sistema UNIX za procesorja 80286 in 16032. Compupro je izdelal prevajalnik za FORTH in makrozbirnik procesorja 16032. DEC je znižal ceno za svoj VT-100:CP/M iz $2400 na $1300.

A.P.Železnikar

VSEBINA LETNIKA 1982

navodilo za pripravo članka

Avtorje prosimo, da pošljejo uredništvu naslov in kratek povzetek članka ter navedejo približen obseg članka (število strani A 4 formata). Uredništvo bo nato poslalo avtorjem ustrezno število formularjev z navodilom.

Članek tipkajte na priložene dvokolonske formularje. Če potrebujete dodatne formularje, lahko uporabite bel papir istih dimenzij. Pri tem pa se morate držati predpisanega formata, vendar pa ga ne vrišite na papir.

Bodite natančni pri tipkanju in temeljiti pri korigiranju. Vaš članek bo s foto postopkom pomanjšan in pripravljen za tisk brez kakršnihkoli dodatnik korektur.

Uporabljajte kvaliteten pisalni stroj. Če le tekst dopušča uporabljajte enojni presledek. Črni trak je obvezen.

Članek tipkajte v prostor obrobljen z modrimi črtami. Tipkajte do črt - ne preko njih. Odstavek ločite z dvojnim presledkom in brez zamikanja prve vrstice novega odstavka.

Prva stran članka :
a) v sredino zgornjega okvira na prvi strani napišite naslov članka z velikimi črkami;
b) v sredino pod naslov članka napišite imena avtorjev, ime podjetja, mesto, državo;
c) na označenem mestu čez oba stolpca napišite povzetek članka v jeziku, v katerem je napisan članek. Povzetek naj ne bo daljši od 10 vrst.
d) če članek ni v angleščini, ampak v katerem od jugoslovanskih jezikov izpustite 2 cm in napišite povzetek tudi v angleščini. Pred povzetkom napišite angleški naslov članka z velikimi črkami. Povzetek naj ne bo daljši od 10 vrst. Če je članek v tujem jeziku napišite povzetek tudi v enem od jugoslovanskih jezikov;
e) izpustite 2 cm in pričnite v levo kolono pisati članek.

Druga in naslednje strani članka:
Kot je označeno na formularju začnite tipkati tekst druge in naslednjih strani v zgornjem levem kotu.

Naslovi poglavij:
naslove ločuje od ostalega teksta dvojni presledek.

Če nekaterih znakov ne morete vpisati s strojem jih čitljivo vpišite s črnim črnilom ali svinčnikom. Ne uporabljajte modrega črnila, ker se z njim napisani znaki ne bodo preslikali.

Ilustracije morajo biti ostre, jasne in črno bele. Če jih vključite v tekst, se morajo skladati s predpisanim formatom. Lahko pa jih vstavite tudi na konec članka, vendar morajo v tem primeru ostati v mejah skupnega dvokolonskega formata. Vse ilustracije morate (nalepiti) vstaviti sami na ustrezno mesto.

Napake pri tipkanju se lahko popravljajo s korekcijsko folijo ali belim tušem. Napačne besede, stavke ali odstavke pa lahko ponovno natipkate na neprozoren papir in ga pazljivo nalepite na mesto napake.

V zgornjem desnem kotu izven modro označenega roba oštevilčite strani članka s svinčnikom, tako da jih je mogoče zbrisati.