DataFlow mašine

Osnovna razlika između DataFlow i ostalih mašina je način pokretanja izvršavanja instrukcija (operacija). U daljem tekstu će se termini instrukcija i operacija koristiti kao sinonimi. Pretežno će se koristiti termin operacija, a na mestima gde je uobičajeno da se koristi termin instrukcija, koristiće se taj termin. Kontrolom pokretani procesori imaju kontrolnu jedinicu koja upravlja svim ostalim delovima procesora, a do pojave koncepta zavisnosti po podacima, svi procesori su bili zasnovani na započinjanju operacija na osnovu toka kontrole. Izvršavanje operacija se kod upravljanja kontrolom obavlja redosledom kojim su instrukcije poređane u memoriji, dok se ne dođe do instrukcije skoka. Na tom mestu se proverava da li treba da se nastavi izvršavanje instrukcija po redosledu u memoriji ili treba da se skoči na neko drugo mesto u programskom kodu. Ako se desio skok, nakon skoka se izvršavanje nastavlja opet redosledom u memoriji, bar do sledeće instrukcije skoka, gde se eventualno skače. Taj redosled instrukcija posledica je redosled koji su na neki način definisali programer i razvojni alat (najčešće programski prevodilac), npr. u višem programskom jeziku.

Da bi se u hardveru lakše pratio taj redosled, u kontrolom pokretanim mašinama je uveden registar/brojač koji sadrži adresu instrukcije koja se trenutno izvršava ili adresu instrukcije koja će naredna da se izvrši, ako nema skoka. Kada dođe do skoka, u registar/brojač se upisuje adresa instrukcije na koju se doskočilo i do sledećeg izvršenog skoka on funkcioniše kao brojač. Zbog različite dužine instrukcija, to nije običan brojač, već brojač sa preskokom, gde se kod svake instrukcije, na osnovu kôda instrukcije, određuje koliki je preskok do početka naredne instrukcije. Ovaj registar/brojač se zove programski brojač ili pokazivač na instrukcije i celokupno izvršavanje programa je oslonjeno na njegovu vrednost.

Kada su početkom sedamdesetih godina prošlog veka analizirali paralelizam, istraživači su ovladali pojmovima zavisnosti operacija po podacima i po kontroli. Tada su zaključili da je potrebno potpuno promeniti logiku izdavanja operacija na izvršavanje. Redosled izdavanja, po tom konceptu, je da pokretanje izvršavanja instrukcija (operacija) treba zasnivati na razdvajanju svih operacija na spremne za izvršavanje po podacima i na one koje nisu spremne po podacima. Spremne operacije su one koje mogu da se izvršavaju, jer nisu zavisne ni po podacima ni po kontroli od operacija koje nisu završene. One koje nisu spremne treba da čekaju da budu izračunati svi podaci-ulazi zbog kojih nisu spremne (podaci i flegovi). Takođe, nespremne operacije moraju da ažuriraju stanje svih svojih ulaza, kako bi na kraju, kada su svi ulazi spremni, prešle u spremne operacije.

Na bazi ove ideje, krenulo se u razvoj podacima pokretanih mašina (DataFlow) koje će funkcionisati tako što operacija prelazi u skup spremnih operacija u hardveru, kada su spremni ulazni podaci, i pod uslovom da ima slobodnih izvršnih jedinica, odmah kreće u izvršavanje. Kada se operacija izvrši, mora se nekako proslediti rezultat (ili flag) do svih operacija koje čekaju na taj rezultat (ili flag). Neke od tih operacija mogu da postanu spremne, ako su čekale samo još na taj argument (ili flag) da bi postale spremne za izvršavanje (postaju *data ready*). Takve operacije bi tada unutar mašine postajale novi članovi skupa spremnih operacija.

U ovom konceptu, postoji pokretanje operacija (instrukcija) na osnovu toga što imaju sve argumente (podacima pokretane), a ne na osnovu programskog brojača! Redosled operacija (instrukcija) u memoriji, odnosno lokacija na kojoj se nalazi, postaje nebitna za tok izvršavanja i služi isključivo za jedinstveno referenciranje operacije.

Osim toga, mora jednoznačno da bude određen podatak ili flag na koji čekaju operacije, a to se može postići principom da se u svaku lokaciju u kojoj se čuvaju podaci, upis radi samo jednom, od strane samo jedne instrukcije. To se zove uklanjanje bočnih efekata (side effects) kod izvršavanja operacija, a time se uklanjaju i antizavisnosti i izlazne zavisnosti. To uklanjanje se olakšava korišćenjem jezika za funkcionalno programiranje, jer je pravilo o upisu samo jednom u svaku promenljivu sastavni deo semantike programskog jezika. Kada se podacima pokreće izvršavanje, nije potreban nikakav programski brojač, već programski prevodilac treba da izgeneriše grafove zavisnosti po podacima i po kontroli i da ih pretoči u oblik pogodan za izvršavanje na DataFlow mašini. Očigledno je da reprezentacija u DataFlow mašini za zavisnosti po podacima i zavisnosti po kontroli kod operacija mora da bude takva, da se jako brzo radi ažuriranje stanja, kako za vrednosti, tako i za stanja operanada (da li su spremni) i za samu operaciju (da li postaje spremna).

Programski jezici za ovakve mašine treba da budu prilagođeni za lako generisanje grafa zavisnosti po podacima i izbegavanje svih slučajeva kada u vreme prevođenja ne može da se definiše da li ima ili nema zavisnosti po podacima. Osim toga, treba da se eliminišu sve antizavisnosti i izlazne zavisnosti, da one ne bi usporavale izvršavanje operacija. Postoji više načina da se to postigne. Osnovno je pravilo da se samo jednom može dodeljivati vrednost promenljivoj (*single-assignment rule)*.

Ako se koriste standardni programski jezici, neophodno je da kompajler dovede kôd do međuforme (intermediate form) u kojoj važi da je ispunjeno da se pridruživanje vrednosti radi samo jednom (*static single assignment form - SSA*). U toj formi se generišu instance promenljive sa leve strane izraza u koju bi se inače više puta upisivalo. Te instance su različito obeležene i numerisane. Prevodilac treba i da poveže te nove instance sa upotrebom odgovarajuće instance promenljive sa desne strane izraza. Posebno složen problem predstavlja generisanje takvih instanci za delove koda koji moraju više puta da se izvršavaju (petlje, procedure, reentrantni delovi koda, …).

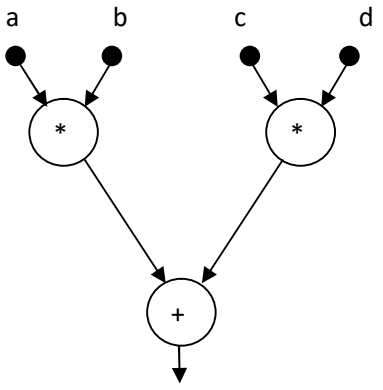
Generisanje SSA forme se može mnogo lakše postići korišćenjem jezika za funkcionalno programiranje za koje važi pravilo da se samo jednom može dodeljivati vrednost promenljivoj (*single-assignment rule)*. Promenljiva se može pojaviti samo jednom na levoj strani izraza unutar programa u kome je aktivna. Kompajler kontroliše ispunjenost forme SSA, odnosno da nema bočnih efekata. Po ostalim osobinama, ovi programski jezici liče na PASCAL jer se izbegava korišćenje pokazivača. Primeri takvih jezika su: VAL, Id i LUCID

Kako se DDG graf formira u vreme prevođenja, neophodno je u mašini iskoristiti sve raspoložive informacije koje se mogu odrediti u vreme prevođenja. Pre svega, na osnovu grafa se zna kojim operacijama neka operacija treba da prosledi svoj rezultat. Pretraživanje grafa bi bilo suviše sporo u vreme izvršavanja, pa je usvojeno da svaka operacija u vreme prevođenja dobije dodatnu informaciju kojim sve operacijama treba da se prosledi njen rezultat. Očigledno je da se i operacije moraju

jedinstveno označiti. Bazična ideja svih realizacija DataFlow mašina je da se za jedinstveno obeležavanje operacija koristi adresa lokacije u memoriji gde se nalazi torka koja opisuje operaciju.

Na osnovu te ideje, formirane su prvo statičke DataFlow mašine. Prvo se postavilo pitanje kako da se graf zavisnosti po podacima nekako pretoči u torke koje će opisivati pojedine operacije. Dakle, već za realizaciju kôda bazičnog bloka je neophodno da se napravi kompleksna instrukcija koja ima jedan ključan problem: u njenom kodu treba nekako da postoji referenciranje svih operacija koje su zavisne po podacima od nje. To predstavlja problem zato što je, u opštem slučaju, broj operacija koje koriste rezultat neke operacije promenljiv. Kako sve operacije nisu komutativne, neophodno je i definisati ulaz sledeće operacije (ulaz aritmetičko logičke jedinice) na koji dolazi rezultat prethodne operacije od koje je operacija zavisna. To je jedan od razloga zašto se moraju formirati mali paketi-torke za podatke. Različite torke se moraju formirati za prosleđivanje istog rezultata operacija do svih drugih zavisnih operacija. Ti mali paketi se nazivaju data tokeni, a u osnovi predstavlja torku sa svim neophodnim dodatnim informacijama koje prate podatak.

Operacije sa jedan ili dva ulaza za podatke, a koje se ne odnose na kontrolne zavisnosti, nazivaju se primitivne operacije. Ako želimo da predstavimo bazični blok, potrebne su nam samo primitivne operacije. Čvorove grafa zavisnosti po podacima zamenjujemo sa primitivnim operacijama, a grane grafa se zamenjuju granama između primitivnih operacija. Neke operacije inicijalno imaju spremne podatke (data ready), definisane u vreme prevođenja, i od trenutka započinjanja programa važe pravila za DataFlow aktivaciju. U DataFlow orijentisanom grafu, aktivacija (firing rule) koji važi je - operacija se može izvršavati kada je omogućena (enabled), a to je kada su svi njeni ulazni data tokeni prisutni. Kada se završi neka operacija, data token rezultata se prosleđuje do svih operacija koje su zavisile po podacima od rezultata završene operacije. Očigledno je da svaka primitivna operacija mora lokalno da ima evidenciju da li je dobila neki ulazni argument i naravno mesto gde pamti vrednosti ulaznih argumenata.



Sl. DF1 DataFlow graf za bazični blok koji izračunava a*b + c*d

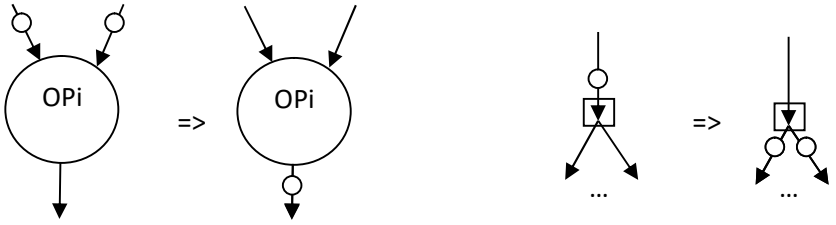
Izvršavanjem primitivnih operacija se data tokeni pojavljuju na izlazu i uklanjaju sa ulaza primitivnih operacija. Na primeru sa slike, sabiranje mora da čeka dok se oba množenja ne završe i pošalju tokene na ulaze primitivne operacije za sabiranje. U DataFlow mašinama uz svaku operaciju treba da postoji deo koji određuje kada operacija treba da se pošalje na izvršavanje. Taj deo treba da

funkcioniše po sledećoj jednostavnoj logici: Izdaj na izvršavanje operaciju (primitivnu operaciju) kojoj su na sve ulaze stigli data tokeni.

DataFlow simboli i forma operacija za bazični blok

Razmotrimo prvo najjednostavniji slučaj bazičnog bloka. U tom slučaju je aciklički graf zavisnosti po podacima sa pravim zavisnostima po podacima jedini koji ograničava paralelno izvršavanje. Operacije (instrukcije) moraju da budu predstavljene u nekom obliku koji podržava podacima pokretano izvršavanje. Kako se podaci ne pamte, već direktno prosleđuju između operacija, operacije moraju da obezbede lokalne lokacije za vrednosti svojih argumenata, ali i bite koji pokazuju da li je trenutna vrednost u tim lokacijama validna (što istovremeno znači i da li su *data ready*). Dakle, dodatni bit uz svaki argument pokazuje istovremeno i validnost argumenta i da li je podatak stigao. Naravno da operacija mora da ima i polje za kôd operacije. U hardveru mašine mora da postoji u nekom obliku i informacija kome se distribuira rezultat operacije kada se izračuna. Ona je u većini statičkih DataFlow mašina bila pridružena operaciji. Ta informacija teorijski sadrži neograničeni broj primalaca, mada je realna veličina mali prirodan broj, tipično manji od 5 operacija primalaca, a posledica je statističkih osobina grafova zavisnosti po podacima.

Dakle operacija mora da bude torka (template) koja ima sledeći oblik: <oper, argL, pL, argD, pD, dest1, L/R1, dest2, L/R2, ….. >. Oper označava polje za kôd operacije (šta treba da uradi sa argumentima), polje argL **vrednost** prvog argumenta ukoliko je upisan. Polje pL (prisutnost levog argumenta) označava da li je primljen argL, odnosno da li je data ready. ArgR i pR imaju analogna značenja za desni argument i potrebni su samo ako operacija zahteva dva argumenta. Zanimljivo je da su bile usvojene oznake L i R za ulaze operacije primaoca rezultata, sa značenjem levi i desni port operacije. Ako operacija zahteva jedan argument, polje za drugi argument (R) se ne koristi. Ako su svi argumenti prisutni, u slučaju bazičnog bloka, operacija može da ide na izvršavanje. Ovaj oblik torke za operaciju je nepogodan za implementaciji u mašini zbog različitog broja destinacija za različite operacije.



Primitivna operacija Fork

Sl. DF2. Simboli potrebni za opisivanje toka izvršavanja bazičnog bloka na DataFlow mašini

Podaci se, kako je već rečeno, prosleđuju takođe u torkama za podatke – data tokenima, a oni kod statičkih DataFlow mašina imaju sledeći oblik: <ip, p, v >. Polje ip označava adresu - pokazivač na operaciju kojoj je poslat taj data token (operaciju), a p označava port te operacije (L ili R, levi ili desni port operacije). Pokazivač na operaciju je u većini realizacija statičkih DataFlow mašina ustvari adresa u memoriji u kojoj se čuva torka destinacione operacije. Port je uveden da bi svaka operacija bila

jednoznačno određena po pitanju argumenata. Kako sve operacije nisu komutativne, moralo se preko porta tačno definisati koji je prvi, a koji drugi argument. Zbog analogije sa hardverskim jedinicama je uveden naziv port. U torci za podatke mora naravno da postoji i sama vrednost podatka, a to je v (value). Zato što se ovakve torke kreću kroz mašinu između operacija, na osnovu direktnih pravih zavisnosti po podacima, uveden je pojam data token-a.

Da bi se predstavile različite operacije u DataFlow mašinama uvedeni su simboli za operacije. Za bazični blok su uvedena dva simbola (operatora) za predstavljanje programa i njegovog toka izvršavanja preko kretanja data tokena (DataFlow). Oni su uvedeni preko primitivne operacije i Fork operacije.

Primitivna operacija prikazuje samo zavisnosti po podacima i zanemaruje kontrolne zavisnosti. Simboli za primitivnu operaciju se umeću na mesto operacija bazičnog bloka, a grane umesto grana zavisnosti po podacima u grafovima zavisnosti po podacima. Jedina razlika je u kružićima koji predstavljaju data tokene. Implikacijom na Sl. DF2, u slučaju primitivne operacije je pokazano da tek ako se pojave data tokeni na oba ulaza operacije (oba argumenta postala su data ready), tada se operacija aktivira i data token ispaljuje na izlaz, a uklanjaju se data tokeni sa ulaza. Fork simbol prikazuje kako se distribuira token sa ulaza, kada se pojavi, na sve izlaze, što predstavlja distribuiranje rezultata operacije – data tokena do svih zavisnih operacija. Fork simbol omogućava da operacije sadrže referenciranje samo dela operacija zavisnih po podacima od njenog rezultata, a deo distribucije može da radi Fork operacija. Ideja ovakvog koncepta je da se može pratiti tok izvršavanja na DataFlow mašini, praćenjem mesta svih data tokena. Inicijalno, za sve operacije slobodne na vrhu bazičnog bloka se smatra da postoje data tokeni na svim njihovim ulazima.

Ovakvom predstavom se ujedno pokazuje asinhronost izvršavanja operacija, isključivo pokretanih podacima (data tokeni). Koliko će vremena proći od trenutka kada su svi ulazi primitivne operacije prisutni do trenutka kada će se rezultat pojaviti kao data token na izlazu zbir je sledećih kašnjenja:

a. Vremena koliko je potrebno da se detektuje da su svi ulazi primitivne operacije prisutni
b. Vremena potrebnog da se operacija prebaci u skup spremnih primitivnih operacija u mašini
c. Vremena čekanja na funkcionalnu jedinicu da se oslobodi (može da bude 0)
d. Vremena potrebnog funkcionalnoj jedinici od trenutka kada je primila argumente do trenutka kada je izgenerisala rezultat

Trajanje fork operacije odgovara trajanju distribucije rezultata operacije do ulaza odgovarajućih zavisnih primitivnih operacija. U principu ta vremena mogu da budu različita za različite operacije po podacima. U odnosu na grafove zavisnosti po podacima bazičnog bloka, jedine promene u ovoj notaciji su uvođenje data tokena i eksplicitne fork operacije.

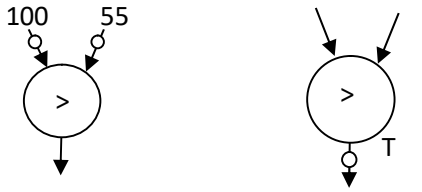
Simboli uvedeni zbog kontrolnih zavisnosti

Primitivne operacije na koje neće uticati kontrolne zavisnosti će se javiti tipično samo u prvom, a eventualno i u poslednjem i još možda u vrlo malom broju bazičnih blokova nekog programa. Dakle, većina operacija će biti zavisna i od kontrole. Naravno da nije moguće raditi sve moguće varijante izvršavanja programa, izbegavajući zavisnost po kontroli, jer bi došlo brzo do eksponencijalne eksplozije

varijanti potencijalno korektnih kodova, a tek na kraju bi se morale odbaciti sve varijante osim jedne, pa bi se za veoma veliki procenat operacija morali poništiti svi njihovi efekti. To bi dovelo do zasićenja mašine bilo kog stepena paralelizma ogromnim brojem nepotrebnih operacija. Dakle, kod DataFlow mašina se **moraju poštovati kontrolne zavisnosti i** moraju se kontrolne zavisnosti na neki način prilagoditi DataFlow konceptu.

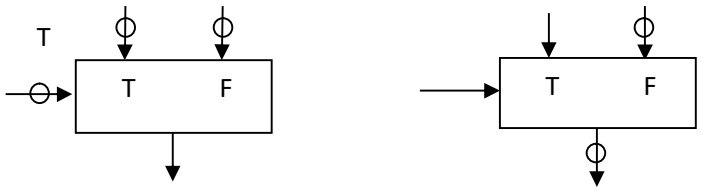
Simbol za relacioni operator

Relacioni operator ima ulaze koji u programskom kodu tipično nisu boolean promenljive, a rezultat uvek mora da bude boolean. Ovi operatori su potrebni npr. zbog ispitivanja uslova kod uslovnih grananja, uključujući i uslove izlaska iz petlje u programskom kodu. Cilj je da relacionim operatorima generisani tokeni sa boolean promenljivom postanu osnov za formiranje predikata zavisnosti po kontroli. Zato su uvedeni simboli koji omogućavaju uslovne operacije.



Sl. DF3. Simbol za relacioni operator , npr. >

Da bi se u DataFlow logici pokretanja operacija implicitno uvele zavisnosti po kontroli, uvedeni su novi simboli (operatori). Sledeća dva simbola su osnovni:



a. Pre izbacivanja data tokena b. Posle izbacivanja data tokena

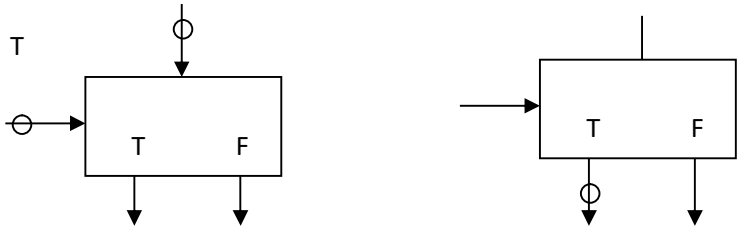
Sl. DF4 Simbol za Merge

Merge ima tri ulazna porta od kojih je jedan kontrolni koji prima kontrolne tokene od relacionih operatora, a dva su za tokene podataka. Dva ulazna porta za podatke simbola Merge se zovu data portovi, a namenjeni su za proizvoljne data tokene sa skalarnim podatkom, a treći ulaz je kontrolni i njegov data token mora da ima boolean tip podatka (kontrolni token). Ako je prisutan token za boolean podatak na kontrolnom ulazu i prisutan data token na ulazu za podatke koji odgovara boolean vrednosti u tokenu na kontrolnom ulazu (T ili F), dolazi do izbacivanja data tokena sa "izabranog" ulaza na izlaz. Ovde treba uočiti da je neophodno prisustvo samo odgovarajućeg ulaznog tokena za podatke i kontrolnog tokena na kontrolnom ulazu, a nije neophodno da oba data tokena sa ulaznih portova za podatke budu prisutni, da bi došlo do izbacivanja izlaznog data tokena. Naravno, sa kontrolnog i

“izabranog” ulaznog porta za podatke se, nakon prosleđivanja ulaznog data tokena na izlaz, uklanjaju tokeni.

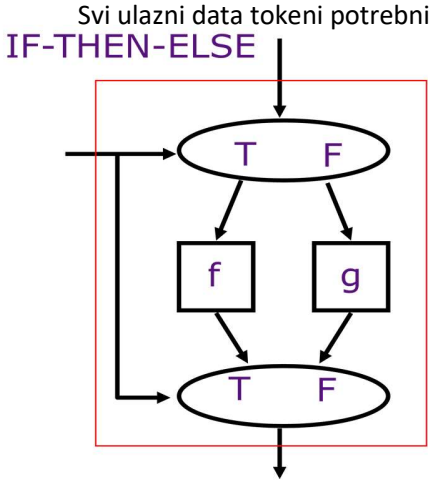
U primeru sa Sl. DF4. je prikazan slučaj kada data token na kontrolnom portu ima vrednost boolean promenljive na true vrednosti. Tada je bitno samo da li je data token prisutan na T (true) ulazu merge simbola I, ukoliko jeste, taj token se prebacuje na izlazni port. To je prikazano na delu slike b. Potrebno je uočiti da na F ulazu nema promene i da data token, ako postoji, ostaje prisutan na tom portu. Treba napomenuti da postoji i merge bez kontrolnog ulaznog porta, koji jednostavno propušta sve data tokene koji mu dolaze na ulazne portove.

Switch simbol ima jedan ulazni port za podatke, a dva izlazna porta za podatke, pored kontrolnog porta.



Sl. DF5. Switch simbol

Switch simbol može da prosleđuje ulazni token samo na jedan od izlaznih portova. Moraju da budu prisutni i kontrolni token na kontrolnom portu i data token na ulazu, da bi se izbacio izlazni data token. Izlazni data token se izbacuje ili na T ili na F izlaz, zavisno od boolean vrednosti u tokenu na kontrolnom ulazu. U primeru sa Sl. DF5. je pokazan slučaj kada je boolean vrednost u tokenu na kontrolnom portu true. Zanimljivo je zašto se ovaj simbol zove još i Select. Ako se data token pojavi na nekom izlazu, onda će moći da se izvršavaju samo operacije koje dobijaju data token sa tog izlaza. One operacije koje očekuju data token sa porta Switch simbola koji nije selektovan neće moći da prime data token sa argumentom, pa neće moći ni da se izvršavaju po data driven principu. Zato ovaj simbol ima semantiku izbora operacija koje se izvršavaju i predstavlja osnovni mehanizam za realizaciju ekvivalenta uslovnih skokova u DataFlow konceptu. U “granama” se i dalje nalaze samo primitivne operacije!



Switch dovodi do konverzije kontrolne zavisnosti u pojavu data tokena na ulazu operacija iz samo jedne grane. Switch se radi za **sve argumente** operacija iz jedne grane. Merge nije morao da bude sa select-om, jer grana kojoj nisu propušteni data tokeni nije ni mogla da generiše data tokene na ulazu merge simbola.

Kontrolni token sa flagom – boolean vrednošću upravlja distribucijom ulaznih argumenata (data token) ka delovima kôda f i g preko Switch operatora. Distribucija data tokena rezultata ili THEN ili ELSE grane se obavlja preko merge simbola.

Sl. DF6. IF-THEN-ELSE struktura realizovana pomoću select i merge simbola

U daljem tekstu je dat jedan primer izračunavanja koji pokazuje kako treba da se transformiše algoritam da bi bio pogodan za izvršavanje na podacima pokretanim mašinama. Kao primer je uzet algoritam izračunavanja faktorijela, koji se najčešće piše u rekurzivnom obliku:

long fact(n)

{ if(n == 0) return 1; else return n * fact(n-1); }

Drugi oblik, napisan u obliku koji nema rekurzivno pozivanje funkcija i prilagođen DataFlow notaciji je:

Fact(n)

inloop := false; I :=n; factor:=1;

WHILE I>1

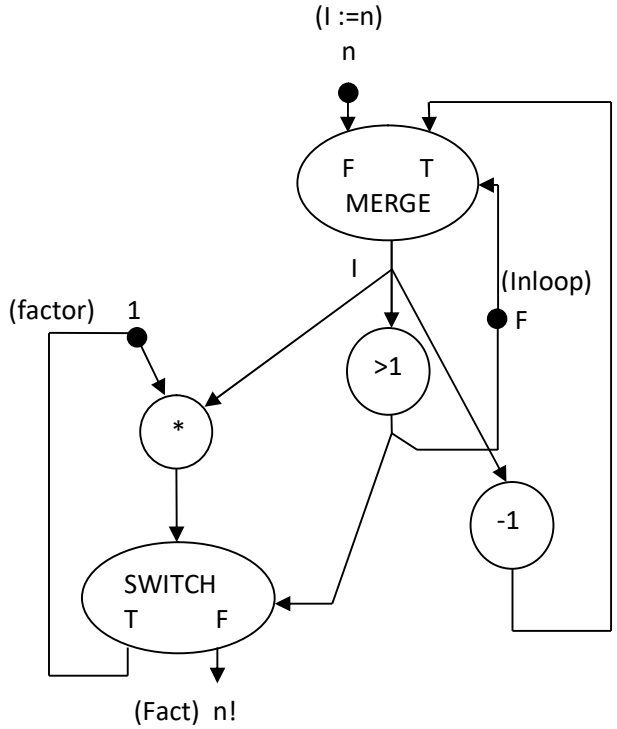
factor:= factor*I; I:=I-1; inloop := (I>1);

ENDWHILE

Return Fact:= factor;

ENDFact;

Svaki novi upis u promenljive factor i I, zahteva nove lokacije u DataFlow mašini. To su novi data tokeni za svaku iteraciju (a u DataFlow nema iteracija ☺)! Ova DOACCROSS petlja ima sledeći DataFlow oblik



Na Sl. DF7. je prikazan DataFlow dijagram korišćenjem navedenih simbola za slučaj izračunavanja faktorijela.

Taj dijagram ima na sebi Inicijalne data tokene sa vrednostima koje su inicijalizovane kao od strane kompajlera. Ta tri data tokena upravo odgovaraju delu koda: inloop := false; I :=n; factor:=1;.

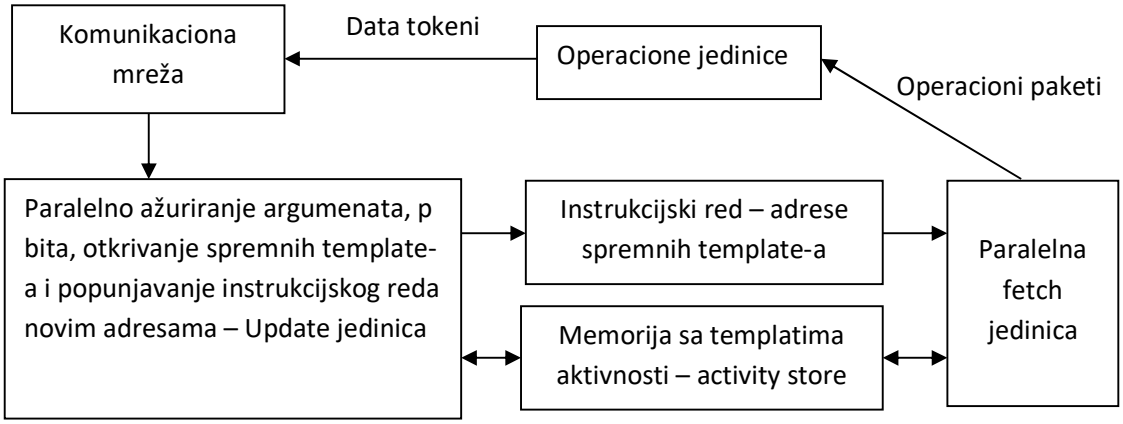
Inicijalno se, preko jednog od merge data ulaznih portova na osnovu pravila funkcionisanja merge operatora izbacuje vrednost n. Ta vrednost se prima na fork operatoru i generisani data tokeni se prosleđuju po jednom ulaznom portu: primitivne operacije koja dekrementira, množačke primitivne operacije i relacione operacije. Kada data tokeni pristignu na oba porta množača, po pravilima za primitivnu operaciju generiše se data token rezultata na izlazu i uklanjaju se data tokeni na ulazu. Token prosleđen na jedini port relacione operacije poredi vrednost iz data tokena da li je veća od 1. Ako jeste, šalje se kontrolni token sa boolean vrednošću true, koji preko drugog fork operatora stiže do kontrolnih ulaza switch i merge operatora. To odgovara dodeljivanju boolean promenljivoj inloop vrednosti unutar iteracija u polaznom programskom kodu. Ti tokeni obezbeđuju da se sve vreme izvršavanja iteracija na jedan ulazni port množača vraća prethodni proizvod i da se na jedan ulaz sabirača vraća prethodni zbir. To odgovara promenljivama factor i I u programskom kodu. Switch na kraju izbacuje rezultat kada I postane jednako 1, tako što izbacuje data token kao rezultat izračunavanja faktorijela na F izlaz.

Jedan deo originalnog programskog koda vezan za promenljivu inloop := (I >1) je bio napisan u obliku da odgovara DataFlow dijagramima. Razlog za to je što inicijalno treba da postoji samo jedan kontrolni token na kontrolnom ulazu merge simbola koja odgovaraju inloop := false. Međutim, drugi kontrolni ulaz (za switch simbol) koji dolazi sa izlaza istog fork simbola nema inicijalno kontrolni token. Zato se za uslov petlje nije mogao staviti inloop. Tok izvršavanja ovog programa se može pratiti preko data tokena koji se izbacuju prema pravilima za svaki simbol. Navedeni primer nema puno paralelizma, jer je u pitanju DoAccross petlja, ali je kroz taj primer objašnjeno izvršavanje programske petlje preko prosleđivanja data tokena kroz DataFlow graf.

Statičke DataFlow mašine

Statičke DataFlow mašine se zasnivaju na ideji da postoje ranije navedene torke <oper, argL, pL, argD, pD, dest1, L/R1, dest2, L/R2, ….. > koje odgovaraju statičkim operacijama, a smeštene su u memoriju koja se naziva *activity store*. Naziv potiče od toga što su čvorovi statičkog DataFlow grafa predstavljeni sa templatima aktivnosti (activity templates), odnosno navedenim torkama. *Instrukcijski red* sadrži pokazivače (memorijske adrese) template-a aktivnosti koji su spremni za izvršavanje, odnosno pokazivače na data ready skup operacija. *Fetch jedinica* ima ulogu da spremne template operacija iz *activity store* pošalje kao operacione pakete do operacionih (izvršnih jedinica). Ona praktično bira odgovarajuću izvršnu jedinicu na osnovu kôda operacije, bira među operacionim jedinicama slobodnu, ako postoji, i predaje template sa spemnim argumentima, kao operacioni paket, na izvršavanje kada nađe slobodnu operacionu jedinicu. Pritom ona i ažurira p bite u activity store – vraća ih na vrednosti nije prisutno, kada pošalje operacioni paket. *Operacione jedinice* nakon izvršavanja operacije generišu data tokene sa rezultatom svog izvršavanja posla definisanog u operacionom paketu. Data tokeni rezultata se preko komunikacione mreže paralelno dostavljaju *Update jedinici,* koja na osnovu destinacija definisanih u data tokenima prosleđuje rezultate do activity template-a destinacionih operacija i u njima se obavlja paralelan upis argumenata i ažuriranje P (presence), odnosno data ready bita. Pritom *Update jedinica* se

za svaki ažurirani activity template proverava da li su svi njegovi argumenti data ready i ako jesu, u instrukcijskom redu se dodaju pokazivači na nove data ready template aktivnosti (operacije).



Sl. DF8. Statička DataFlow mašina, konceptualni model

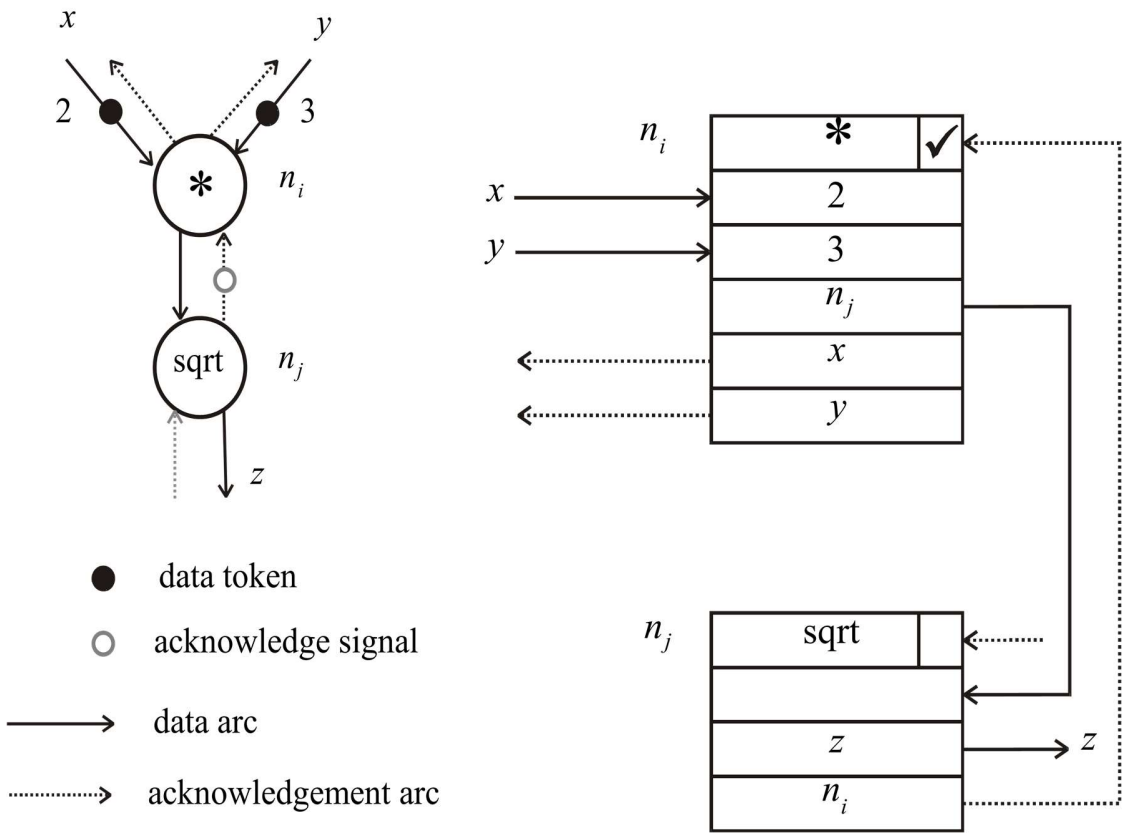
Osnovni nedostatak statičkih DataFlow mašina potiče od činjenice da su podaci vezani statički za operaciju, pa problem predstavljaju delovi koda koji se ponavljaju. To su pre svega petlje i procedure koje se pozivaju. Problem će biti ilustrovan sa DoAll petljama. Pretpostavimo zbog jednostavnosti da Doall petlja nema zavisnosti preko granica iteracija. Tada su slobodni skupovi operacija za sve iteracije odjednom data ready, ako se posmatra samo petlja. Kako su mašine podacima pokretane, u delovima koda pre petlji, ti spremni argumenti za iteracije petlji se izračunavaju i iteracije petlje mogu da krenu čim su raspoloživi. Dakle, za istu statičku operaciju iz petlje se može pojaviti poplava argumenata iz različitih iteracija – različitog konteksta. Pritom, statička DataFlow mašina nema mogućnost da uparuje levi i desni argument operacije, tako da moraju da budu iz iste iteracije. Na osnovu pokretanja podacima, bez restrikcija, moglo bi da se dobije mešanje argumenata za operacije iz razlicitih iteracija i samim tim nekorektno izvršavanje.

Osim toga, ako bi se dopustilo da se obavljanje operacije radi samo na osnovu dostupnosti argumenata (ulaznih data tokena), operacija bi mogla da se izvrši, ali update jedinica ne bi mogla da smesti rezultat – data token u activity store. To se događa: ako neka od operacija u activity store-u, koja treba da primi taj token, već ima bar jedan (bilo koji) argument iz neke druge iteracije prisutan, ili ako su oba argumenta već prisutna, ali se čeka *Fetch jedinica* da pošalje operacioni paket na izvršavanje. Rešenje sa privremenim pamćenjem takvog rezultata u *Update jedinici* bi užasno zakomplikovalo realizaciju statičke DataFlow mašine, upravo u njenom najkompleksnijem delu.

Zato je nađeno rešenje za prevazilaženje oba problema. Modifikovano je pravilo pokretanja izvršavanja operacija. Da bi se operacija pokrenula, potreban uslov je da budu na raspolaganju svi argumenti, ali je potreban uslov i da sve operacije koje treba da prime data tokene te operacije moraju da budu u takvom

stanju da odgovarajući port koji treba da primi token mora da bude slobodan. Kao rezultat, značajno je smanjen paralelizam, a operacije se nalaze u nekoj vrsti asinhronog pipeline-a po prstenu statičke DataFlow mašine na Sl. DF8.

Model operacije je izmenjen kao na primeru sa Sl. DF9.



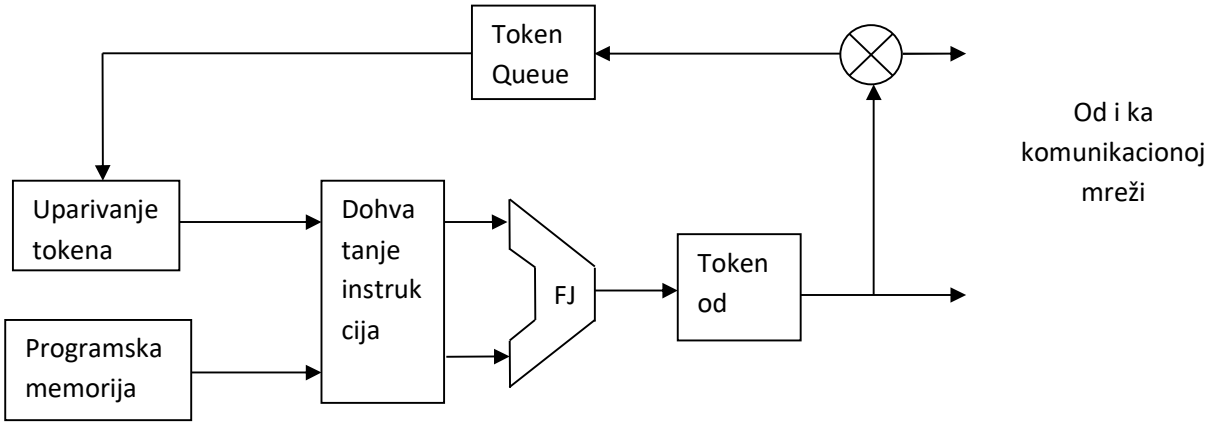
Sl. DF9. Primer izmene logike pokretanja, uvođenjem signala potvrde (acknowledge)

Kako ista pravila važe i za relacione operacije i kontrolne tokene, praktično se uvodi redosled iteracija, uparuju argumenti iz istih iteracija, i time izbegava nagomilavanje data tokena u update jedinici.

Cena je smanjen paralelizam, i potreba za praćenjem acknowledge signala svih operacija zavisnih od neke operacije, da bi se utvrdilo da li Fetch jedinica može da dohvati operaciju, da bi je poslala na izvršavanje. Ovo pravilo sa acknowledge signalom važi i za *Switch* i *Merge* operatore. Smanjeni paralelizam dovodi do toga da operacije iz samo malog broja iteracija mogu da budu preklopljene.

Dinamičke DataFlow mašine.

Dinamičke DataFlow mašine su razdvojile operacije od logike otkrivanja kada je neka instanca operacije spremna za izvršavanje. To je omogućilo da postoji više instanci izvršavanja nad kojima se vrši uparivanje argumenata za istu operaciju. Pritom je pridruženo dodatno polje za data token nazvan *frame*. Vrednost polja *frame* je ista za iste iteracije, pozive procedura, … U dinamičkoj DataFlow mašini se kod uparivanja tokena postavlja uslov da data tokeni budu sa istom vrednošću *frame*-a, da bi se krenulo u izvršavanje i generisao izlazni token. To uparivanje kod petlji znači da se paralelno može raditi uparivanje ulaza za istu operaciju iz različitih iteracija, čime se postiže veći paralelizam. Naravno, broj bita kojim je označen *frame* ograničava koliko će se paralelnih instanci uparivanja i izračunavanja operacije može istovremeno da postoji. To praktično znači koliko će različitih iteracija moći da se izvršava istovremeno. Preko broja *frame*-ova je ograničen paralelizam, ali ne moraju više da postoje signali potvrde - acknowledge kao kod statičkih DataFlow mašina. Mogući paralelizam je veći, ali je deo za uparivanje tokena kojim se vezuju podaci iz istog konteksta kompleksniji. Model jednog procesorskog elementa za dinamičku DataFlow mašinu je prikazan na Slici DF10.



Sl. DF10. Procesni element dinamičke DataFlow mašine

Svaki data token ima integralni tag koji čine: Adresa instrukcije i porta za koju je namenjen i kontekstna informacija – *frame* pointer. Kako data tokeni koji dolaze na granu DataFlow grafa imaju jedinstven tag, više njih se može naći na istoj grani, pa se svaka grana kao torba koja može da sadrži proizvoljan (ograničen veličinom polja za *frame* pointer) broj tokena sa različitim integralnim tagovima. Odatle sledi pravilo za dinamičke DataFlow mašine kada je omogućeno izvršavanje čvora: **Čvor je omogućen kada su tokeni sa identičnim integralnim tagovima istovremeno svi prisutni na svim ulaznim granama**.

Zbog potrebe da se ograniči paralelizam, jer su limiti pojedinih delova mašina konačni, mora se nekako uspostaviti mehanizam ograničavanja. Kako je već veličina *frame* pointera ograničena, mora se u mašini sprečiti generisanje novih tokena kada broj iteracija može da bude veći od broja raspoloživih različitih *frame* pointera. Zato je neophodno rešiti problem alokacije i dealokacije *frame* pointera i uspostaviti pravilo o kočenju generisanja novih tokena kada je dostignut limit paralelizma.

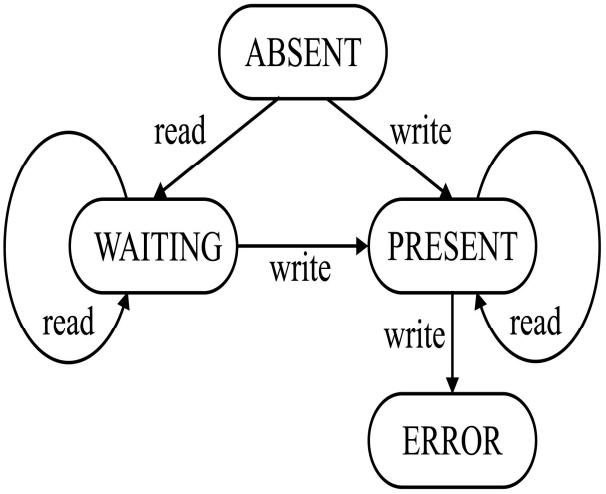
Strukturiranje podataka

Za strukturiranje podataka se koriste i strukture. Osnovna ideja je da postoji mogućnost strukturiranja podataka, a da za svaki pojedinačni element strukture važe pravila koja odgovaraju konceptu pokretanja izvršavanja po podacima. Strukturiranje se radi preko I struktura. Elementi I strukture odgovaraju elementima nizova, ali se ponašaju kao čuvari data tokena koji mogu da se proslede svima kojima je data token potreban. Zbog toga za svaki element I-strukture postoje stanja definisana na sledeći način:

- *prisutan*: element može da se čita, ali ne da se upisuje,
- *odsutan*: pokušaj čitanja mora da bude odložen, ali je dozvoljen jedini upis,
- *čekanje*: bar jedan zahtev za čitanje postoji za element I-strukture koji je na čekanju.

Kada se pokuša čitanje I strukture, ako je element prisutan, odmah se čita. Ako je bilo zahteva za čitanje, a element je bio u stanju čekanje, nakon što je element postao *definisan* (inicijalizovan, dodeljena vrednost; može se desiti samo jednom), sva *odložena čitanja*, koja se čuvaju u asocijativnom redu, prestaju da budu odložena i sve operacije koje su čekale na element pročitaju upisanu vrednost.

Na Sl. DF11. je prikazan dijagram za prelaze između stanja elementa I strukture sa anglosaksonskom terminologijom



Status svakog elementa I-strukture može biti:

present: može se čitati,

absent: zahtev za čitanje se mora zakasniti, a upis je dozvoljen,

waiting: postoji bar jedan read zahtev koji je zakašnjen.

Sl. DF11. Dijagram za prelaze između stanja elementa I strukture

Osnovne operacije koje mogu da se sprovode nad elementima I strukture su definisane na sledeći način:

Alociranje (allocate): rezerviše definisani broj elemenata za novu I-strukturu. Nakon alokacije se samo jednom može raditi upis.

Čitanje (I-fetch, read): čita sadržaj određenog elementa I-strukture (ili zakašnjava čitanje)

I-store: upisuje vrednost u element I-strukture, a ako element nije prazan javlja grešku.

Kao posledica usvojene logike I struktura, *I-fetch* instrukcije se ponašaju kao memorijske operacije sa razdeljenim fazama: zahtev za čitanje elementa je u principu nezavisan u vremenu od vremena odgovora (čeka se da element postane data ready).

Prednosti i mane DataFlow mašina

Prednosti DataFlow mašina je što, bar konceptualno, koriste sav raspoloživi paralelizam i teže da jedino ograničenje budu grafovi zavisnosti po podacima sa pravim zavisnostima određeni u vreme prevođenja. U tom kontekstu, one su vrlo dobre mašine za korišćenje iregularnog paralelizma (koji nije vezan isključivo za petlje). U praktičnoj implementaciji se taj paralelizam ograničava, zbog velikog broja resursa potrebnih za korišćenje iregularnog paralelizma.

Broj mana je daleko veći i zato su ove mašine naišle samo na ograničenu primenu. Pre svega, DataFlow graf se formira na bazi zavisnosti po podacima i kontroli koje su određene u vreme prevođenja. Kao posledica, na osnovu ranijih poglavlja ove knjige, nekada se dodaju zavisnosti i kad ih u vreme izvršavanja nema. Osim toga, veom je težak debugging, jer nema preciznog stanja, a praćenje svih razloga zašto je neka operacija započeta u nekom trenutku je praktično nemoguće. Poseban problem predstavljaju prekidi i obrada izuzetaka. Za njih je problem definisati u kom trenutku uopšte nastaju u kontekstu izvršavanja, a dodatnu teškoću predstavlja očuvanje stanja pravilnog izvršavanja na mašini nakon prekida ili obrade izuzetaka. Primena dinamičkih struktura podataka je teška u čistom DataFlow modelu, a I strukture su pokušaj da se preko mehanizama alokacije umanji značaj te mane. Pokazalo se da nije moguće realizovati obradu kada postoji preveliki paralelizam, a želi se pokretanje izvršavanja po podacima, bez drugih ograničenja. Zato je i bilo potrebno uvesti kontrolu paralelizma.

Kod dinamičkih DataFlow mašina, koje su jedini realni kandidati za širu primenu, ogroman posao vezan je za uparivanje tagova. Tu su potrebne velike asocijativne memorije za ogroman broj instrukcija i tokena i kombinacija sa interkonekcionim mrežama. Pritom postoji znatno veća potreba za zauzećem memorije za tokene podataka, jer su oni nekoliko puta veći nego sam podatak i još se generišu kopije (data tokeni) za sve operacije kojima je taj podatak potreban. Instrukcijski ciklus je neefikasan jer postoji veliko kašnjenje u sprovođenju data tokena između zavisnih instrukcija. Kod kontrolom pokretanih mašina se koristi lokalnost u memoriji, dok kod podacima pokretanih mašina to nije slučaj. Jezici za programiranje izbegavaju pokazivače, pa se programeri moraju navići na nov stil programiranja. Spekulativno izvršavanje po kontroli praktično ne postoji, jer se spekulativnost radi za cele blokove, a ne za pojedinačne operacije. Pritom spekulativnost znači izvršavanje obe grane, pa pokušaj agresivne spekulativnosti dovodi do ogromnog rasipanja resursa i malog procenta korisnih operacija. Realizacija

uslovnih grananja i petlji dovodi do trošenja dodatnih ciklusa zbog dodatih select i merge operatora za grananja. Operacije nad vrstama i kolonama niza se uvek svode na nezavisne operacije nad elementima I-struktura, što otežava pisanje programa. Sama konstrukcija ovakvih mašina onemogućava uspostavu dugačkih pipeline-a instrukcija međusobno povezanih funkcionalnih jedinica.

Čist DataFlow koncept je našao primenu u ugrađenim procesnim jedinicama gde se moraju veoma brzo obavljati operacije u petljama. Jedan takav primer je hardver sa umapiranim DataFlow grafom algoritma npr. brza Furijeova transformacija – FFT. Slični algoritmi su realizovani po DataFlow konceptu u ruterima, grafičkim procesorima, ...

Jedan od mogućih pravaca je da će u budućnosti unutrašnje petlje DoAll tipa da se kompajliraju pomoću silikonskih kompajlera i da se ugrade u programabilnu logiku sa idejom da se svakog ciklusa započinje jedna nova iteracija (kao hardverizovan software pipelining). Ovo je vrlo atraktivna ideja, ali je glavni problem kako upariti tako brzu logiku sa ostatkom procesora. Jedno od mogućih rešenja je uvođenje velikog broja vektorskih registara koji su veoma brzi.

Dominantan pravac razvoja današnjih procesora je kombinovanje dobrih strana iz DataFlow i control flow koncepata. Ideja je dinamički odrediti za deo instrukcija DDG i sa tim izabranim delom raditi kao ugrađenom DataFlow mašinom sa ograničenim brojem operacija. Izbor se radi na osnovu prediktora grananja, čime se uvodi spekulativnost po kontroli. U narednim poglavljima će biti opisana takva rešenja.