VLIW – EPIC mašine

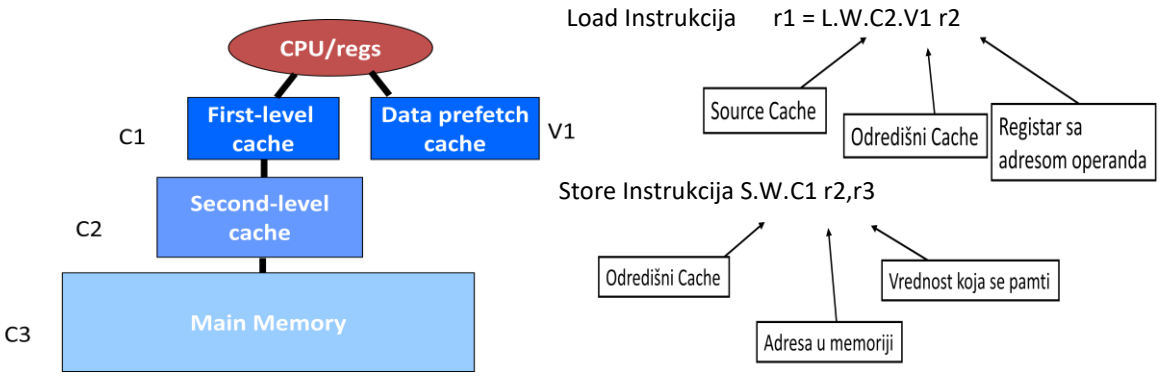
Very Long Instruction Word ili VLIW CPU arhitektura ostvaruje instrukcijski nivo paralelizma tako što postoji kontrola toka kao kod tradicionalne von Neuman arhitekture, ali se započinjanjem izvršavanja grupa operacija svakog ciklusa upravlja iz veoma široke instrukcijske reči. U poljima te široke reči su definisane operacije koje treba da se izvrše, a mogu da započnu izvršavanje u paraleli. Odluku da mogu da se izvršavaju u paraleli donosi kompajler na osnovu zauzeća resursa mašine u svim ciklusima tih operacija i zavisnosti po podacima definisanih u vreme prevođenja. Nakon ciklusa započinjanja operacija, kontrolni signali potrebni za završetak tih operacija prate faze izvršavanja kroz protočne stepene u ciklusima izvršavanja tih operacija. Svaka široka reč sadrži i polje za uslovni skok, a odlučivanje o skoku se radi na bazi izabranog flega sa izabrane funkcionalne jedinice. Postoji po nekoliko protočnih izvršnih jedinica (integer ALU, FP ALU, FP i integer množača, adresnih generatora za memorije), ali često i veći broj drugih jedinica, pre svega multiportnih registarskih memorija, magistrala, pa i memorija.

Broj izvršnih jedinica je direktno (eksplicitno) vidljiv u instrukcijskoj reči preko kontrolnih polja kojima se upravlja tim jedinicama. Da bi se paralelno upravljalo operacijama koje započinju u istom ciklusu, mora da postoji instrukcijska reč širine bar 128 bita, a postojale su implementacije sa 256 i više bita. Tako može da se započne preko 8 operacija po ciklusu, tako da je ukupan paralelizam, kada se uzme u obzir i protočna obrada raspoloživi paralelizam u mašini je tipično veći od onoga u grafu zavisnosti po podacima i kontroli.

Osnovni koncept VLIW procesora je da se u vreme prevođenja radi optimizacija kojom se želi najkraće vreme izvršavanja programa. To sprovodi kompajler na osnovu analize zavisnosti po podacima i raspoloživosti resursa mašine. Grupisanje operacija koje započinju u istom ciklusu se kompletno planira u vreme prevođenja. Otuda se za VLIW koristio i termin grupno-sekvencijalne mašine, jer se definišu grupe operacija koje započinju izvršavanje u istom ciklusu, a zadržava se tradicionalna kontrola toka. Kod VLIW, u okviru kompajlera mora se kod formiranja instrukcija obezbediti da raspored operacija poštuje zavisnosti po podacima, za svaki potencijalni dinamički trag. Osnovne metode paralelizacije se zasnivaju na algoritmima koji su ranije prikazani u ovoj knjizi (List Scheduling, Trace Scheduling i razmotavanje petlji i softverska protočnost petlji). Umesto Trace Schedulinga se često koriste razne modifikacije koje uglavnom služe za pojednostavljivanje procesa optimizacije.

VLIW mašine su zasnivanjem celokupne optimizacije kôda i planiranja operacija koje mogu da započnu istog ciklusa dovele do toga da se kôd može veoma paralelno izvršavati sa hardverom koji nije preterano kompleksan. Izbegavanjem analize zavisnosti po podacima i zavisnosti po kontroli u vreme izvršavanja je postignuta ta relativna jednostavnost hardvera. Kontrola toka kod Trace Scheduling-a zasnivala se na činjenici da se ceo kôd formira na bazi predikcije grananja u vreme prevođenja, odnosno na statički definisanim predikcijama grananja (profiling ili heuristike). Međutim, te statičke predikcije samo doprinose boljoj optimizaciji kôda. U osnovnoj verziji VLIW mašina, jedino su se operacije mogle obavljati spekulativno po kontroli, ako nisu generisale izuzetke i to ako su rađena preimenovanja, kada rezultat nije mrtav u drugoj grani. Celokupno planiranje se radilo uz pretpostavke o konfiguraciji VLIW mašine i egzaktnu definisanost šta svaka operacija radi u svakom ciklusu. To praktično znači da samo promena broja protočnih stepeni u nekoj izvršnoj jedinici znači da se mora ponovo prevoditi sav kôd.

U vreme prevođenja, kod prvih VLIW arhitektura, se podrazumevalo da kompajler planira koje operacije počinju istovremeno, koje funkcionalne jedinice će tačno da rade svaku operaciju i u kom ciklusu će svaka od tih reči da započne izvršavanje za svaki mogući dinamički trag. To je značilo i eksplicitno trajanje memorijskih operacija, da bi se kompletno planiranje moglo raditi u vreme prevođenja. Kao posledica, celokupna memorijska hijerarhija je bila vidljiva programeru i on je morao eksplicitno da radi load i store operacije za svaku od memorija iz hijerarhije. Kada želi da smanji broj ciklusa dohvatanja kod load operacije, programer je morao da radi preload – prebacivanje iz sporije memorije u bržu – koja odgovara keš memoriji. To je ilustrovano na Sl. VLIW1, a premeštanje podataka između memorija je bilo eksplicitno i zauzimalo je polja za operacije VLIW reči.



Sl. VLIW1 eksplicitni preload podataka i store u definisani deo memorije u hijerarhiji

Dakle, programeri su morali da definišu celokupno premeštanje podataka između delova memorije u hijerarhiji. U osnovi nije postojala keš memorija, već samo memorije različite brzine koje su na raspolaganju programeru, mada su to zvali keš memorijama. Iako je to značilo da se u vreme prevođenja moglo tačno znati vreme izvršavanja za svaki dinamički trag, što je poželjno za real-time primene, ovo je predstavljalo veliki nedostatak, jer je opterećivalo programera.

Dve osnovne mane su onemogućavale širu primenu bazičnih VLIW mašina. Prva je bila nepostojanje prave keš memorije, a druga je nekompatibilnost sa postojećim kôdom RISC ili CISC procesora, pa se nije mogao koristiti ranije razvijen softverski kôd. Uvođenjem dinamičkih VLIW mašina je omogućeno da se uvedu prave keš memorije. Kompajler je i dalje planirao koje sve operacije mogu da se započnu istovremeno i tačno na kojoj funkcionalnoj jedinici se obavlja operacija, ali je vreme starta ostavljeno hardveru u vreme izvršavanja. To je omogućilo da se tokom prevođenja predvidi broj ciklusa odziva memorijske hijerarhije kada se radi load, a hardver bi određivao da li je podatak stigao u predviđeno vreme ili ne. Na bazi tog stizanja, hardver bi puštao izvršavanje VLIW reči sa svim operacijama ili bi zakašnjavao do trenutka stizanja podatka.

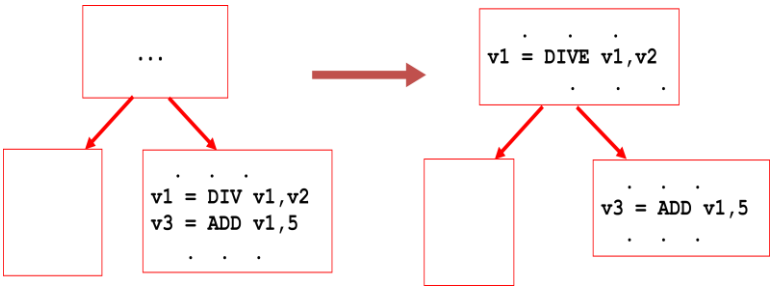
Mana nekompatibilnosti sa postojećim mašinama je razrešena na taj način što je za svaku operaciju unutar VLIW reči definisano da se koriste isti kôdovi operacija kao što su instrukcije neke RISC mašine. To je dovelo i do toga da polja VLIW reči ne moraju da automatski korespondiraju određenoj funkcionalnoj

jedinici u mašini, već se to može dinamički raditi i hardver može donositi odluke. Kada je uspostavljena ova vrsta kompatibilnosti, VLIW mašine su preimenovane u EPIC mašine. One su zadržale i osobinu dinamičkih VLIW mašina da hardver odlučuje vreme starta celokupne VLIW reči u vreme izvršavanja.

Današnji EPIC procesor tipično je RISC-baziran i ima više od 4 glavne izvršne jedinice. Program se tipično prvo prevodi kao za RISC, a tek zatim VLIW prevodilac radi izmenu redosleda i grupisanje u tokove koji nemaju međusobne zavisnosti po podacima. Zatim se takvi tokovi pakuju u susedne velike instrukcijske reči. Takav način generisanja kôda omogućava kompatibilnost kôda i mogućnost da se generišu različita prepakivanja za VLIW sa različitim nivoom paralelizma, a da se ujedno obezbedi kompatibilnost prema RISC mašinama.

U EPIC procesorima se nametnula ideja da se obezbedi spekulativnost po kontroli i kada operacija koja treba da se izvrši spekulativno može da generiše izuzetak. To je obezbeđeno uvođenjem operacija sa dodatkom na naziv željene spekulativne operacije, kojim se označava da je operacija spekulativna.

Prevodilac obeležava spekulativne instrukcije, npr. sa E dodatim na naziv operacije. Takve operacije npr. DIVE ili ADDE, su specifične po tome što se ne podiže odmah izuzetak, ako se dogodi. Kako su operacije spekulativne, ako se desi izuzetak tokom spekulativne operacije, samo se setuje dodatni bit za spekulativni izuzetak u registru rezultata, da zabeleži taj spekulativni izuzetak. Rezultat takve spekulativne operacije nije validan, ako je setovan dodatni bit za spekulativni izuzetak, ali i operacije koje koriste takav rezultat kao argument mogu biti spekulativne (sa E dodatim na naziv operacije). Zato se bit za spekulativni izuzetak jednostavno propagira od strane spekulativnih operacija (argument sa setovanim bitom => rezultat sa setovanim bitom). Tek kada se desi da operacija koja nije spekulativna dobije kao argument, odnosno pristupi registru sa postavljenim (setovanim) bitom za spekulativni izuzetak, tek tada se podiže izuzetak. Ta operacija je, samim tim što nije spekulativna, u originalnom bazičnom bloku, pa je mesto generisanja izuzetka vrlo blizu nekom mestu gde bi se on generisao u sekvencijalnom kôdu.



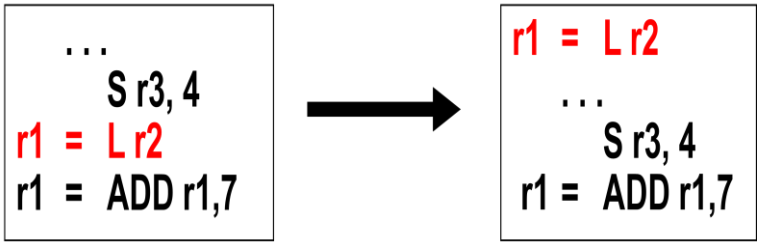
Ako se desi deljenje sa 0, izuzetak se podiže na mestu operacije ADD, gde je v1 operand (propagacija exception bita), a operacija kojoj je v1 operand nije spekulativna

Sl. VLIW2. primer generisanja izuzetka kod spekulativnog deljenja sa 0

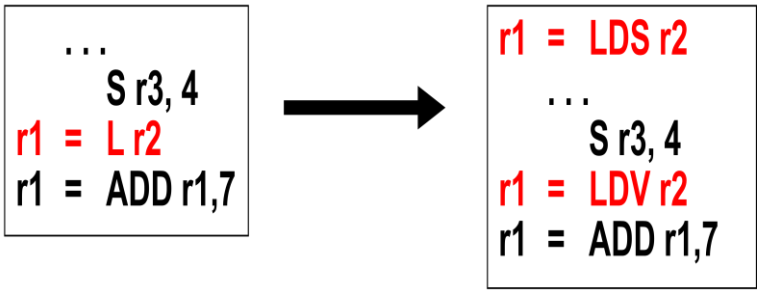
Mesto generisanja izuzetka je, dakle, bazični blok iz koga je izvučena spekulativna operacija ili bazični blok posle tog bazičnog bloka, ali veoma blizu, na dinamičkom tragu. Zbog potencijalne agresivnosti u paralelizaciji kôda u vreme prevođenja, ovo odlaganje exception-a je veom važno, da bi se pravilno radila obrada exception-a. Izuzetak nije potpuno precizan, na nivou operacije, jer nije ni mogao da se generiše na nivou operacije. Ipak, uvek se generiše, ako je neophodno da se generiše, a nikada se ne događa da dođe do toga da se nepotrebno generiše.

U EPIC procesorima su je i dalje ostao problem nemogućnosti da se u nekim slučajevima odredi zavisnost po podacima u vreme prevođenja. To se posebno odnosi na zavisnosti preko memorije, kada u vreme izvršavanja, sadržaji nekih registara određuju kojoj se lokaciji pristupa, a taj njihov sadržaj nije bilo moguće znati u vreme prevođenja. Tada svaki load mora da sačeka da se završe sve prethodne store operacije na nepoznatu lokaciju u vreme prevođenja. Pritom je mala verovatnoća da store i kasniji load po originalnom redosledu pristupaju istoj lokaciji. Nametnula se ideja spekulativnosti po podacima, kako bi se ubrzalo izvršavanje u ovakvim slučajevima, uvođenjem pretpostavke u vreme prevođenja da zavisnost ne postoji! Naravno, morao se naći način da se uradi oporavak kada se desilo da je store, koji je originalno bio pre load-a, upisivao u istu lokaciju iz koga je planiran load. Oporavak je uveden tako što su uvedene nove instrukcije za spekulativni load. Dakle, spekulativni load se raspoređuje pre store operacije, a kada se planira obavljanje store operacije, mora se uraditi provera pretpostavke iz vremena prevođenja da nije postojala zavisnost po podacima preko memorije. To se radi sa operacijom load verify.

Na Sl. VLIW3. je ilustrovano kako se raspoređuje par operacija spekulativnog load-a i load verify-a u odnosu na store operaciju. Naravno, oni su deo velike reči sa puno operacija. Zbog velikog kašnjenja kod load-a, željena optimizacija je da se load obavi pre store operacije. Međutim, ova optimizacija u vreme prevođenja nije validna ako load i store referenciraju istu lokaciju, tj. r2 i r3 sadrže istu adresu. U vreme prevođenja se međutim ne može odrediti da li su iste adrese. Način da se to uradi je određivanje da li postoji zavisnost u vreme izvršavanja. To se zove *run-time memory disambiguation.* Kod njega se pokušava što ranije dohvatiti podatak iz memorije, iako je neki store, po originalnom redosledu pre tog load-a, mogao da upisuje u tu lokaciju u vreme izvršavanja.



Ukoliko bi r2 i r3 imali iste vrednosti, ovo bi bio nekorektan kôd. Tada bi postojala zavisnost operacije Load od operacije Store, pa bi ovakav premešten Load uzimao pogrešnu vrednost



Spekulativni load LDS čita vrednost sa lokacije adresirane sa r2. Ukoliko bi r2 i r3 imali različite vrednosti, ovo bi bio korektan kôd, a load koji dugo traje je urađen unapred. Ako su r2 i r3 jednaki, blokiraju se pipeline-i i LDV radi ponovni Load, kako bi se pokupila vrednost zapamćena sa S (Store) operacijom i kada ta vrednost stigne do registra r1, otkoče se pipeline-i.

Sl. VLIW3. Primer korišćenja spekulativnog Load-a i Load verify-a

Na ovom mestu će kompajler da uradi sledeće:

Ubaciće dve specijalne instrukcije u zamenu za jednu load instrukciju:

r1 = LDS r2 - spekulativni load pre store instrukcije, pretpostavljajući da ne postoji zavisnost po podacima. Ovo je u osnovi spekulacija po podacima, jer se pretpostavlja da store neće promeniti taj podatak. U planiranom izvršavanju, započinje spekulativni load kao normalnu operaciju i pamti vrednost u posebno mesto (tabelu) za pamćenje vrednosti iz memorijske lokacije. Kada se započne izvršavanje store operacije i poznata je adresa na koju se pamti, u narednoj širokoj reči mora da postoji operacija **r1 = LDV r2** - load verify. Ona proverava da li je možda bilo pamćenja u tu lokaciju od kada je LDS započeta. Ako se to dogodilo, koči se pipeline i započinjanje izvršavanja celokupnih sledećih reči sa operacijma, a započinje se novi load za istu lokaciju se izdaje. U suprotnom je LDV operacija ekvivalentna sa no-op (pogodili smo u spekulaciji da nije bilo zavisnosti po podacima). Tada se izvršavanje nastavlja punom brzinom prema planiranom rasporedu u vreme prevođenja. Ovakvo kočenje pipeline-a, kada postoji zavisnost po podacima preko memorije, je moguće zbog principa rada dinamičkih VLIW mašina.

Predikacija

Osnovna ideja predikacije – predikatskog izvršavanja je da se doda boolean uslov – predikat uz instrukciju. To dodavanje se može raditi kada se izdaje instrukcija na izvršavanje, prilikom samog izvršavanja, ili kada se na neki način radi commit instrukcije. Trenutak u kome se proverava vrednost predikata je zavisan od implementacije. Ako se prilikom provere utvrdi da je predikat istinit, čuva se rezultat instrukcije. Ako se pak utvrdi da je predikat lažan, poništavaju se svi efekti operacije.

Kada se samo kod nekih kôdova operacije može izvršavati predikacija, govori se o parcijalnoj predikaciji. U nekim implementacijama je moguća predikacija za sve kôdove operacija i tada se govori o punoj predikaciji.

Predikacija donosi značajne koristi u realizaciji EPIC mašina. Pre svega, može se preklopiti izvršavanje operacija iz više bazičnih blokova bez ikakve eksplozije kôda, ako se ne uračuna izračunavanje bita predikacije. Uvođenjem predikacije se radi konverzija grananja u predikaciju, tako da se redukuje učestalost instrukcija grananja. Samim tim se redukuju svi problemi koji mogu nastati ako je pogrešno predviđeno grananje u vreme prevođenja. Samim tim se redukuje prosečan broj grananja koje treba sprovesti po ciklusu, tako da se može povećati broj operacija u VLIW/EPIC instrukcijskoj reči. Smanjuje se i broj dinamičkih tragova izvršavanja, što kod nekih kategorija mašina donosi i uštede u hardveru. Predikacija omogućava formiranje superblokova, koji se sastoje od velikog broja bazičnih blokova koji se paralelizuju zajedno, tehnikom sličnom List Scheduling-u, uz definisanje svih predikata.

Mane predikacije su vezane za povećano dohvatanje operacija, što odgovara povećanju zauzeća široke instrukcijske reči operacijama koje nisu bile potrebne za dinamički trag. Osim toga, povećava se i zauzeće registara rezultatima koji će biti poništeni. Ako se testira validnost predikata u vreme kada se radi commit operacije, povećava se i zauzimanje funkcionalnih jedinica. Spekulativno izvršavanje podrazumeva testiranje predikata baš kada se radi commit. Pomeranje operacija preko grananja na gore povećava kompleksnost obrade izuzetaka i kao što je prikazano ranije u ovom poglavlju, morale su da se uvode novi kôdovi operacija. Kod nekih EPIC mašina, sve operacije imaju predikat, pa su proširene, tako da traže veći broj bita u širokoj instrukcijskoj reči, a to opet ima za posledicu potrebu da instrukcijski keš ima širu reč.

Najšire je prihvaćen koncept da svaka operacija ima samo jedan bit za predikat. Da bi se formirao taj predikat, moguće je raditi logičke operacije nad prethodnim predikatima. Ako se poveže takva logika sa prikazom kontrolnih zavisnosti preko regionskih čvorova, taj jedan bit predikata u potpunosti odgovara predikatu regionskog čvora. U grafovima zavisnosti po kontroli, regionski čvorovi su često prikazivani kao zavisni od prethodno izračunatih predikata regionskih čvorova. Tako se i u mašinama sa predikacijom formiraju predikati na osnovu jednostavnih operacija između ranije određenih predikata nekog drugog regionskog čvora i novoizračunatih rezultata komparacija. Za većinu ili sve operacije se može uraditi predikacija. Zanimljiva je situacija sa predikatom operacije posle spoja dve grane. Ako se za sve operacije radi predikacija, onda se preko OR operacije nad predikatima grana koje se spajaju pravi predikat grane posle spoja. Međutim, ako bi se selila operacija preko grananja na gore, kompajler bi kopirao operaciju u obe grane i dodelio kopijama predikate dolaznih grana.

U osnovnoj formi predikacije, dodatni operand koji omogućava predikaciju je jednobitni predikatski registar. Primer takve operacije je

r2 = ADD r1,r3 <u>if p2</u>

Ako predikatski registar sadrži vrednost 0, operacija se ne izvršava ili ne komituje, a ako je 1, izvršava se ili se komituje (zavisno od trenutka testiranja predikata). Vrednosti predikatskih registara se tipično postavljaju sa “compare-to-predicate” operacijama, koje imaju oblik:

p1 = CMPP<= r4,r5

Značenje ovog izraza je p1 je 1 (true) ako je r4<=r5. Ovakva predikacija podrazumeva da je unapred određen predikat. Ona se može koristiti da bi se radila if-konverzija, čime se grananja pretvaraju u boolean operacije i predikacije sa jednim bitom. Drugi, mnogo značajniji način, korišćenja je pravljenje

hiperblokova prilikom paralelizacije kôda, a paralelizacija je jednostavnija nego kod Trace Scheduling-a. Hiperblokovi su delovi kôda koji imaju jedan ulaz i više izlaza.

Navedimo primer sa IF konverzijom gde postoji puno grananja i teško je uraditi paralelizaciju zbog kontrolnih zavisnosti:

```
if (a < b)
  c = a;
else
  c = b;
if (d < e)
  f = d;
else
  f = e;
```

Sl. VLIW4. Kod sa puno grana koji je potrebno paralelizovati IF konverzijom

Ovaj kôd se može predstaviti sa dve EPIC instrukcije u kojoj prva radi 4 “compare-to-predicate” operacije određivanja predikata u jednoj reči sa prvom reči, a u drugoj reči se rade 4 operacije sa različitim predikatima koje se izvršavaju IF konverzijom:

P1 = CMPP.< a,b	P2 = CMPP.>= a,b	P3 = CMPP.< d,e	P4 = CMPP.>= d,e
c = a if p1	c = b if p2	f = d if p3	f = e if p4

Veoma je često da su predikati komplementarni, pa su uvedene “compare-to-predicate” operacije koje obezbeđuju simultano dva izlaza CMPP instrukcije:

- p1,p2 = CMPP.W.<.UN.UC r1,r2
- U označava bezuslovno, N znači normalno, a C označava komplement

Sa samo dve dvoizlazne CMPP instrukcije i generisani kôd za primer sa Sl. VLIW4. može da bude:

p1,p2 = CMPP.W.<.UN.UC a,b	p3,p4 = CMPP.W.<.UN.UC d,e

c = a if p1	c = b if p2	f = d if p3	f = e if p4

.

Sa kompleksnijim compare-to-predicate operacijama, dobija se redukcija visine kontrolnih zavisnosti. Svi navedeni primeri su do sada podrazumevali prvo određivanje predikata, pa tek onda izvršavanje operacija sa IF konverzijom koja podrazumeva korišćenje izračunatih predikata. Da bi se postiglo jednostavno formiranje kompleksnijih predikata, uvedene su kompleksnije “compare-to-predicate” operacije.

Osnovna izmena koja omogućava kompleksnije predikate je uvođenje zaštićenih “compare-to-predicate” operacija. One se mogu predstaviti sledećim kôdom:

p1,p2 = CMPP.<cond>.<D1-action>.<D2-action>(r1,r2) if p3

p3 predstavlja zaštitu, a uslov (cond) relaciju između r1 i r2 (npr. "<="). Rezultat komparacije se koristi u kombinaciji sa predikatskim ulazom (p3) i destinacionom akcijom <Dx-action> da bi se odredile destinacione predikatske vrednosti p1 i p2.

Adestinacione akcije za svaki predikat mogu da budu bezuslovne U ili uslovne C i da generišu normalnu boolean vrednost N ili komplement C. Bezuslovne akcije zanemaruju predikat zaštite p3 u formiranju p1 i p2. Osim toga akcije mogu da sadrže i logičke operacije. Osnovna ideja zaštite kod uslovne operacije je da se propagira false vrednost predikata p1 i p2, ako je zaštita p3 određena kao false. To znači propagaciju poništavanja svih efekata operacija koje se izvršavaju pod predikatima p1 i p2, ako je p3 false (0). To je preduslov za predikaciju kada se predikat formira na bazi uslova, a logički je zavisan od prethodno izračunatog predikata nekog regionskog čvora.

Označavanja destinacionih akcija su: unconditionally set (UN or UC), conditionally set (CN or CC), wired-OR (ON or OC), ili wired-AND (AN or AC). Prvi karakter (U, C, O or A) određuje akciju koja se sprovodi sa odgovarajućim destinacionim predikatima, a drugi karakter (N or C) određuje da li se rezultat komparacije koristi u "normalnom modu" (N), ili u "komplementarnom modu" (C). wired-AND i wired-OR su između p3 i uslova (cond) relacije između r1 i r2.

Danas je najpoznatiji VLIW procesor Intel - HP Itanium IA-64 EPIC procesor. Osim toga, gotovo svi procesori za digitalnu obradu signala su VLIW.