

Elektrotehnički fakultet Univerziteta u Beogradu
Katedra za računarsku tehniku i informatiku

- diplomski rad -

Ontologija arhitekture i organizacije računara

Mentor:
Doc. dr Boško Nikolić

Kandidat:
Dragana Erić 01/14

Beograd, 2009.

UVOD.....	3
1. SEMANTIČKI <i>WEB</i> I ONTOLOGIJE.....	4
1.1 Vizija Semantičkog <i>Web</i> -a.....	4
1.2 Softverski agenti	5
1.3 Arhitektura Semantičkog <i>Web</i> -a.....	7
1.3.1 <i>Unicode</i> i <i>URI</i>	7
1.3.2 <i>XML</i> i <i>XML Schema</i>	8
1.3.3 <i>RDF</i> i <i>RDF Schema</i>	9
1.3.4 Ontologije i <i>OWL</i>	11
1.3.5 Nerealizovani deo steka Semantičkog <i>Web</i> -a.....	18
1.4 Projekti Semantičkog <i>Web</i> -a.....	19
2. ARHITEKTURA I ORGANIZACIJA RAČUNARA.....	22
2.1 Osnove arhitekture računara	22
2.2 Aritmetika računara	26
2.3 Organizacija i arhitektura memorijskog sistema.....	30
2.4 Interfejs i komunikacija	36
2.5 <i>Device</i> podsistemi	40
2.6 Dizajn procesorskog sistema.....	44
2.7 Organizacija procesora.....	46
2.8 Performanse	52
2.9 Modeli distribuiranih sistema	55
2.10 Poboljšanja performansi.....	60
3. ONTOLOGIJA ARHITEKTURE I ORGANIZACIJE RAČUNARA.....	64
ZAKLJUČAK.....	72
Literatura.....	73

UVOD

Trenutni *World Wide Web* je velika biblioteka povezanih dokumenata sa mnogo informacija i znanja. Mašine obično služe samo da dostave i prezentuju sadržaje dokumenata, dok ljudi moraju sami da povežu izvore relevantnih informacija i da ih interpretiraju. Semantički *Web* predstavlja pokušaj poboljšanja sadašnjeg *Web*-a koji bi omogućio mašinsko procesiranje, interpretaciju i povezivanje informacija.

Tekst je podeljen na tri dela. Prvi deo nazvan „Semantički *Web* i ontologije“ se sastoji iz opisa vizije Semantičkog *Web*-a, uloge koju imaju softverski agenti u toj viziji; zatim su date osnove arhitekture Semantičkog *Web*-a (*Unicode* i *URI*, *XML* i *XML Schema*, *RDF* i *RDF Schema*, *Ontologije* i *OWL*, i nerealizovani deo arhitekture) i na kraju su navedeni neki primeri projekata zasnovanih na tehnologijama Semantičkog *Web*-a (*FOAF*, *LOD*, projekti iz polja zdravstva i bioloških nauka).

Drugi deo „Arhitektura i organizacija računara“ je podeljena u sledeće oblasti čiji su koncepti ukratko opisani: Osnove arhitekture računara, Aritmetika računara, Arhitektura i organizacija memorijskog sistema, Interfejs i komunikacija, *Device* podsistemi, Dizajn procesorskog sistema, Organizacija procesora, Performanse, Modeli distribuiranih sistema i Poboljšanja performansi.

U trećem delu „Ontologija arhitekture i organizacije računara“ opisana je jedna moguća ontologija iz domena arhitekture i organizacije računara.

Na kraju je dat zaključak i spisak korišćene literature.

1. SEMANTIČKI WEB I ONTOLOGIJE

Tehnološki razvoj poslednjih nekoliko decenija nam je omogućio pristup velikoj količini informacija kojima se ne može efikasno upravljati. Neophodne su tehnike za filtriranje, izdvajanje i sumiranje potrebnih važnih informacija. Za postizanje zajedničkog opšteg pogleda na sve podatke i za razumevanje njihovih relacija, nužna je integracija informacija. Deljenje podataka i poslovne logike između različitih aplikacija čini suštinu integracije aplikacija. Efektna implementacija menadžmenta i integracije informacija kao i integracije aplikacija, zahteva semantičko opisivanje podataka i procesa tj. povezivanje sa mašinski-obradivim opisima njihovog značenja. Ovo predstavlja osnovnu ideju iza Semantičkog Web-a - kreiranje i upotreba semantičkih metapodataka.

1.1 Vizija Semantičkog Web-a

Tradicionalni Web se zasniva na tri osnovna gradivna bloka:

- globalna identifikacija resursa pomoću URI identifikatora (*Uniform Resource Identifier*),
- standardni klijent-server protokol za objavljivanje globalno dostupnih podataka – HTTP (*HyperText Transfer Protocol*) i
- jednostavan jezik za kreiranje hipertekst dokumenata – HTML (*HyperText Markup Language*).

Osnovne karakteristike HTML-a su povezivanje dokumenata ili delova dokumenata, strukturiranje teksta (naslovi, liste, tabele), formatiranje teksta, umetanje grafike, kreiranje jednostavnih korisničkih interfejsa (radio dugmad, tekstualna polja itd.). HTML je koristan za vizuelnu reprezentaciju i direktno ljudsko procesiranje – čitanje, pretraživanje, popunjavanje formi. Problem sa većinom podataka na Web-u je što su podaci u HTML fajlovima korisni samo u određenim kontekstima, dok ih je teško koristiti na globalnom nivou. Količina informacija na Web-u je stvorila veliku potrebu za automatskim procesiranjem Web sadržaja, a HTML ne omogućava mašinsko razumevanje dokumenata.

Tim Berners-Lee, izumitelj WWW, URI, HTTP i HTML-a, je u članku objavljenom 2001. godine izneo viziju Semantičkog Web-a u kojem bi računari bili sposobni da analiziraju sve Web podatke (sadržaje, linkove, transakcije) i da preuzmu obavljanje zadataka koje su dotad korisnici morali ručno da rade. Glavni cilj Semantičkog Web-a, kao ekstenzije tradicionalnog Web-a, je da on postane efikasniji za svoje korisnike. Povećanje efikasnosti se sastoji iz automatizacije stvari koje se inače teško obavljaju: pronalaženje, upoređivanje i povezivanje sadržaja, izvođenje zaključaka na osnovu informacija iz više resursa itd. Ideja iza Semantičkog Web-a je da se Web učini što inteligentnijim, da pored čuvanja i upravljanja podacima, pomaže ljudima u koordinaciji njihovih dnevnih aktivnosti. Tehnologije neophodne za ostvarenje ove vizije su: zajednički jezik za predstavljanje podataka i relacija među njima, kojeg razumeju sve vrste softverskih agenata, ontologije koje prevode informacije iz različitih baza podataka u zajedničke termine, i pravila koja omogućavaju agentima da zaključuju o informacijama opisanim datim terminima. Format podataka, ontologije i softver za

zaključivanje bi delovali kao jedna velika aplikacija na *Web-u* koja analizira sirove podatke iz baza.

U ovom novom *Web-u*, informacije će imati dobro definisano značenje, kao rezultat upotrebe semantičkih *markup* jezika. Koncept mašinski čitljivih dokumenata ne podrazumeva neku vrstu veštačke inteligencije, već će definisana semantika informacija omogućiti računarima da rešavaju probleme sekvencijalnim procesiranjem operacija. Semantički *Web* nije jedan entitet, već kolekcija dokumenata, baza podataka i miliona objekata čiju je bogatu semantiku potrebno opisati.

Semantički *Web* se može posmatrati kao kolekcija međusobno povezanih izvora informacija kojim treba upravljati sistemom za upravljanje baza podataka. Izazovi koji se postavljaju su modeliranje baza podataka, integracija heterogenih informacija i izvora informacija, postavljanje upita i pristupanje podacima tehnikama indeksiranja.

Prilikom kreiranja Semantičkog *Web-a*, mnoge karakteristike tradicionalnog *Web-a* treba zadržati - jednostavni protokoli, jednostavna sintaksa i pristup, opipljiva korist. Jedna od najboljih karakteristika *Web-a* je što znači različite stvari različitim ljudima. Za neke će definišuća osobina Semantičkog *Web-a* biti lakoća s kojom PDA uređaj, laptop, desktop računar i auto komuniciraju međusobno; za druge je to automatizacija odluka i lakoća nalaženja odgovora na pitanja, dok je za ostale to sposobnost procenjivanja pouzdanosti dokumenata na *Web-u*.

Posvećen tim ljudi u *Web* konzorcijumu (W3C) radi na standardizaciji, unapređivanju i proširivanju sistema, jezika, alata. Ne možemo reći da će razvoj Semantičkog *Web-a* stati kada budemo imali sistem sa mašinski čitljivim *Web* stranicama. Kako se pojavljuju nove tehnologije, Semantički *Web* će nastaviti da se širi. Skoro svi servisi će se obavljati putem *Web-a*, uključujući dnevne aktivnosti ljudi. Semantički *Web* će kao sledeća generacija sadašnjeg *Web-a* proširiti njegovu snagu dodavanjem mašinski čitljivih informacija i automatizovanih servisa.

1.2 Softverski agenti

Tehnološki napredak zajedno sa konzistentnim smanjenjem cena hardvera ubrzava automatizaciju društva kao celine. Uporedo s ovim, suočeni smo sa brzim rastom kompleksnosti procesa integracije, upravljanja i rada računarskih sistema. Današnji distribuirani sistemi traže aplikacije koje pored rešavanja zahteva za servisima, moraju da predviđaju i prilagođavaju se da bi odgovorili na potrebe korisnika.

Softverski agent je softverski entitet dizajniran da interaguje sa okolinom – korisnicima i drugim aplikacijama. Za definisanje agenta se koristi skup atributa koji ga karakteriše. On zavisi od tipa agenta, odnosno od zadataka koje agent treba da izvršava. Međutim u neophodne osobine agenata se svrstavaju interaktivnost, autonomnost i adaptivnost. Autonomnost je stepen nezavisnosti od spoljne kontrole koju agent ima. Interaktivnost je sposobnost agenta da komunicira sa njegovom okolinom i sa drugim entitetima-agentima. Interakcija među agentima može biti jednostavna razmena poruka ili složena koordinacija heterogenih agenata koja zahteva monitoring, procenu, interpretaciju, planiranje i pregovaranje.

Softverski agenti će imati važnu ulogu u građenju Semantičkog *Web-a*. Dobijaće određene zadatke i preference od korisnika, zatim će tražiti informacije iz raznih *Web*

izvora (pri tom će komunicirati sa drugim agentima), pronađene informacije će onda porediti sa zahtevima korisnika, i izabrane adekvatne podatke će kao odgovor vraćati korisniku. Agenti različitih tipova, kao npr. brokeri, preuzimaju zadatke koje bi inače ljudi sami radili, izvršavaju ih u ime korisnika i pregovaraju najbolje uslove. Na osnovu dostupnih informacija sprovode akcije i donose odluke, npr. rezervišu prevoz, avionsku kartu i hotel za putovanje, ažuriraju medicinski zapis ili daju jedan prilagođen odgovor na određeno pitanje bez da korisnik traži informaciju u listi rezultata. Primer jednog scenarija je da osoba treba da zakaže pregled kod lekara. Ograničenja koja se uzimaju u obzir su tesan raspored te osobe, doktorove kvalifikacije, geografska lokacija i drugi faktori. Rešenje se može postići i pomoću tradicionalnog Web-a, ali uz značajni napor te osobe i utrošak vremena na posećivanje raznih sajtova. Semantički Web rasterećuje korisnika od ovakvih glomaznih poslova jer se agenti zadužuju za sprovođenje pretrage i pregovaranje sa različitim stranama (u ovom slučaju sa pacijentovim rasporedom tj. agendom, doktorom, medicinskim katalogom). Iako svaka strana kodira svoje informacije na različit način, zbog semantičkog sloja oni će moći da interaguju i razmenjuju podatke. Zato je predstavljanje semantike i upravljanje semantikama aplikacija esencijalno za Semantički Web.

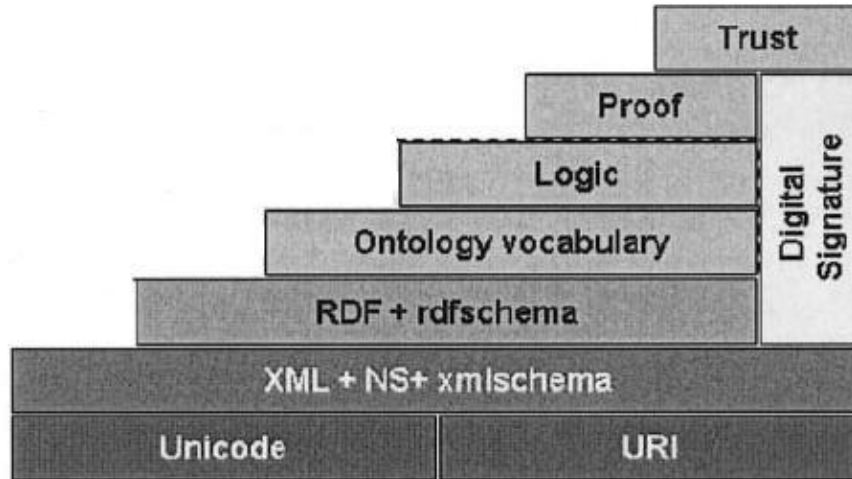
Da bi ispunili njihove zadatke, softverski agenti moraju da koriste metapodatke i ontologije uopšte. Uloga ontologija je da posluže kao okvir koji će omogućiti deljenje informacija između agenata. Iako bi komunikacija među agentima na Web-u bila znatno olakšana usvajanjem jedne referentne ontologije, u budućnosti se predviđa da će svaki Web sajt, organizacija ili biznis na Web-u imati svoju ontologiju. Pošto će različiti agenti imati odvojene ontologije, mogućnost integracije ontologija u jednu deljenu reprezentaciju će biti ključno za obezbeđivanje njihove komunikacije (semantička međuoperabilnost).

Većina agenata mora da komunicira međusobno kako bi koordinisali njihove aktivnosti, zajednički planirali akcije, pregovarali, održavali deljeno stanje ili menjali planove na osnovu akcija drugih agenata. Interakcija se odvija na više nivoa. U osnovi mora biti zajednička infrastruktura agenata da bi mogli komunicirati. Pošto se komunikacija odvija razmenom poruka, moraju znati protumačiti primljene poruke, rečnik korišćen u porukama (ontologiju). Takođe će možda morati da objasne svoje rezonovanje drugim agentima slanjem logičkih formula. Agent treba da ima sposobnost da dokaže poreklo, da ima ovlašćenje da radi u ime nekog korisnika (npr. plaćanje usluge). Deo nadležnosti će možda trebati da prosledi drugim agentima – još uvek je nerešen problem kako to uraditi bezbedno i pouzdano.

Najjednostavniji mogući agenti koji nemaju fleksibilno inteligentno ponašanje i koji ne sarađuju sa drugim agentima se već godinama koriste. Takozvani *spider* agenti obavljaju jedan jedini zadatak – prikupljaju određenu vrstu informacija iz raznih Web resursa. FIPA konzorcijum (*Foundation for Intelligent Physical Agents*) podržava rad na ontologijama, semantici, servisima i sigurnosti, a za cilj ima pravljenje standarda za međuoperabilnost heterogenih softverskih agenata. FIPA pruža obiman skup specifikacija za infrastrukturu agenata. Specifikacije su u različitim stadijumima razvoja i uglavnom su prilično abstraktne, npr. *ACL Message Structure* specifikacija definiše strukturu i sadržaj poruka ali nezavisno od njenog formata.

1.3 Arhitektura Semantičkog Web-a

Semantički *Web* identifikuje skup tehnologija, alata i standarda koji formiraju osnovne gradivne blokove infrastrukture. *Web* konzorcijum (*World Wide Web Consortium*, W3C) je izneo veoma jasnu slojevitcu arhitekturu Semantičkog *Web*-a. Sastavljena je iz niza standarda organizovanih u strukturu koja izražava njihove međusobne relacije; svaki sloj koristi sposobnosti slojeva ispod njega. Stek takođe pokazuje kako je Semantički *Web* proširenje klasičnog *Web*-a a ne njegova zamena.



Donji slojevi arhitekture (URI, *Unicode* i XML) obezbeđuju sintaksnu međuoperabilnost. Povrh XML sloja je RDF i RDF *Schema* za međuoperabilnost podataka. Digitalni potpis (*Digital Signature*) se pruža vertikalno duž steka čime se ističe njegova široka korisnost – dodaje nivo sigurnosti *Web*-u koji dotad nije postojao. Sloj ontoloških jezika prevazilazi ograničenja RDF(S) sloja ispod njega. Slojevi na vrhu steka (*Logic*, *Proof* i *Trust*) još uvek nisu značajno istraženi i ne zna se kako će biti implementirani.

1.3.1 *Unicode* i URI

U osnovi arhitekture Semantičkog *Web*-a su *Unicode* i URI standardi.

Unicode (*Universal Character Set*) je standard za kodiranje internacionalnih skupova karaktera koji obezbeđuje jedinstven broj za svaki karakter, nezavisno od jezika, programa ili platforme. Pre njega su postojali mnogi različiti sistemi kodiranja što je otežavalo manipulaciju podacima. Računari su morali da podržavaju više kodiranja, a uvek je postojala opasnost od konflikata kada se isti broj koristi za različite karaktere ili različiti brojevi za jedan isti karakter.

Za identifikovanje apstraktnih ili fizičkih resursa na *Web*-u se koriste formatirani stringovi - URI identifikatori. Bilo šta može da ima URI i za sve što ima URI se kaže da je na *Web*-u. Jedna forma URI identifikatora je URL (*Uniform Resource Locator*) koji identifikuje resurs reprezentacijom njegovog mehanizma pristupa. URL govori računaru gde da nađe određeni resurs (npr. adresa *Web* stranice). Pošto je globalna računarska mreža prevelika da bi bila kontrolisana od strane neke organizacije, URI identifikatori su

decentralizovani. Ne postoji osoba ili organizacija koja nadgleda pravljenje i upotrebu URI-ja. Zato više URI identifikatora može da se odnosi na potpuno isti resurs, ali se to ne može utvrditi. Nikad ne možemo sa sigurnošću reći šta dati URI predstavlja. URI identifikatore mogu da definišu individue ili zajednice i organizacije za standarde. Na primer, relacija „is a“ je generalno veoma korisna pa je W3C konzorcijum objavio standardni URI za njeno predstavljanje.

Internacionalna varijanta URI-ja je IRI (*Internationalized Resource Identifier*) koji dozvoljava upotrebu *Unicode* karaktera u identifikatoru i za kog je definisano mapiranje u URI.

1.3.2 XML i XML Schema

Prvi korak prema sledećoj generaciji *Web*-a predstavlja uvođenje XML jezika (*eXtensible Markup Language*) i familije povezanih standarda. Predložen je kao rešenje za probleme integracije podataka jer omogućava njihovo fleksibilno kodiranje i prikaz. XML podržava zajedničku reprezentaciju dokumenata što omogućava da različiti sistemi interpretiraju dokument na isti način. Različite heterogene aplikacije i baze podataka rade zajedno koristeći XML. On obezbeđuje elementarnu sintaksu za strukturu sadržaja unutar dokumenata. Opšti *markup* jezik kreiran za *Web* je ubrzo postao široko prihvaćen format za slanje i razmenu različitih strukturiranih podataka. XML dozvoljava aplikacionom dizajneru da sam osmisli svoj format i da zatim piše dokument u tom formatu. XML formati imaju *markup* oznake koje poboljšavaju značenje sadržaja i koje programi mogu da čitaju i interpretiraju.

Delovi sadržaja se obeležavaju tagovima. Ime taga predstavlja labelu za tekst između početnog i završnog taga (cela struktura se zove element). Na taj način računar ima određenu vrstu informacije o dokumentu. Više informacija pruža dodavanje atributa sa imenom i vrednošću. Komponente XML dokumenta su prolog, koreni element (telo dokumenta) i opcioni završni deo sa komentarima i informacijama kako da se procesira dokument. Prolog može da ima deklaraciju o datoj verziji XML-a i DTD (*document type definition*) koji specificira korišćene elemente, podelemente i attribute. DTD je osnovni mehanizam za opisivanje strukture podataka u XML dokumentu; za dokument se kaže da je validan ako je baziran na DTD-u.

Ograničenja DTD-a se prevazilaze drugim mehanizmom za definisanje XML gramatike, a to je XML *Schema*. Napredne karakteristike koje ona podržava su definisanje osnovnih tipova podataka (brojevi, stringovi, datumi..) i definisanje tipova atributa i elemenata (nasleđivanje dozvoljava restrikcije i ekstenzije). Međutim XML *Schema* funkcioniše na čisto sintaksnom nivou u definisanju strukture XML podataka, određene stvari su izvan njene ekspresivnosti. Pravo nasleđivanje na semantičkom nivou nije podržano, kao ni kompleksna upoređivanja elemenata ili atributa.

Iste reči koje se koriste u različitim *markup* jezicima mogu imati različita značenja. Da bi se sprečila zabuna, XML rečnici se jedinstveno identifikuju. Posebnu ulogu u XML dokumentu ima *xmlns* atribut koji definiše prostor imena (*namespace*) za sve elemente iz njegovog opsega. Prostor imena rešava konflikt (problem heterogenosti) tako što

identifikuje deo *Web* prostora iz kog se izvode URI-ji za sve tagove i attribute iz opsega. XML dozvoljava skraćivanje ili upotrebu podrazumevanog prostora imena da bi se izbeglo zamorno pisanje cele URI oznake.

Tehnologije koje se koriste za povezivanje XML dokumenata su *Xlink*, *Xpointer* i *Xpath*. Podržani su jednostavni linkovi kao u HTML-u (jedna veza između dva resursa) i prošireni linkovi koji povezuju više resursa (npr. veza rečnika sa prostorom imena). Tehnike za postavljanje upita u bazi XML dokumenata su *Xquery* i *XMLSQL* koji može da dohvata vrednosti elemenata i atributa u zavisnosti od nekog uslova.

Glavna kritika XML-a je što je smišljen samo za reprezentovanje strukture dokumenata, što dozvoljava samo sintaksnu međuoperabilnost. XML ne prenosi značenje podataka iz dokumenta; ne može se prepoznati semantika iz određenog domena. Značenje tagova se ne može opisati XML-om. Drugi problem je što XML ima slab model podataka koji ne obuhvata relacije. Za prevazilaženje semantičkih problema sa XML-om, razvijen je RDF „semantički XML“. Iako je XML samo standard za formatiranje podataka, pripada skupu tehnologija koje čine osnovu Semantičkog *Web*-a.

1.3.3 RDF i RDF *Schema*

Semantički *Web* dozvoljava uspostavljanje relacija između različitih informacija, bilo da je objekat dokument, fotografija, tag, finansijska transakcija, rezultat eksperimenta ili apstraktni koncept. Sveopšti jezik za predstavljanje podataka zvani RDF (*Resource Description Framework*) imenuje svaki objekat kao i relacije među njima na način koji omogućava da računari automatski razmenjuju informacije. *Web* konzorcijum je na XML-u razvio RDF jezik kako bi standardizovao definisanje i upotrebu metapodataka. To je prvi jezik napravljen specijalno za Semantički *Web* koji dodaje mašinski čitljive metapodatke postojećim podacima na *Web*-u. RDF opisuje sadržaj dokumenta kao i relacije između raznih entiteta u njemu. Sa RDF-om se mogu dodati dokumentu unapred definisane primitive za izražavanje semantike podataka bez pravljenja pretpostavki o strukturi dokumenta. Omogućava integraciju i međuoperabilnost podataka pa aplikacije mogu lakše da sarađuju i komuniciraju međusobno. RDF pruža mogućnost pravljenja iskaza koje mašine „razumeju“. Računar ne može zaista da razume iskaze ali može da ih procesira na takav način da izgleda kao da ih razume. Važan RDF princip je da bilo ko može reći bilo šta o bilo čemu, dve osobe mogu na primer da govore kontradiktorne stvari. RDF je idealan za objavljivanje baza podataka na *Web* - inteligentni programi kombinuju podatke i odgovaraju na upite koji zahtevaju pristup više baza istovremeno.

Na prvi pogled RDF je veoma sličan XML-u, ali detaljnija analiza otkriva da su oni konceptualno različiti. Ako bi informacije iz RDF modela predstavili XML-om, računar više ne bi mogao da raspozna semantičku strukturu. RDF komplementira XML - dok XML pruža sintaksu i notacije, RDF obezbeđuje semantičke informacije na standardizovan način. W3C je napravio XML serijalizaciju RDF-a. Zvanična RDF specifikacija definiše XML reprezentaciju RDF-a. RDF/XML se smatra standardnim formatom za razmenu RDF-a na Semantičkom *Web*-u iako postoje i drugi formati koji su

manje komplikovani i lakši za upotrebu, npr. *Notation3*. RDF je moćniji od XML-a jer pored toga što koristi XML sintaksu, on opisuje semantiku.

Osnovni model podataka RDF-a je veoma jednostavan. RDF iskaz liči na prostu rečenicu s tim što su umesto reči URI identifikatori; ima formu trojke <subjekat, predikat, objekat>. RDF trojka obeležava konekciju između dva resursa (predikat povezuje subjekat sa objektom).



Predikat se zove i properti jer opisuje resurs subjekat sa vrednošću properti-resursom objektom. Objekat jedne trojke može imati funkciju subjekta druge trojke što rezultuje označenim usmerenim grafom – subjekti i objekti korespondiraju čvorovima, a predikati usmerenim granama grafa. Napredniji koncept u RDF-u je *container* model sa tri tipa objekata sadržalaca: *bag* (neuređena lista resursa ili literala koja označava da properti ima više vrednosti čiji redosled nije bitan), *sequence* (lista uređenih resursa, redosled je bitan) i *alternative* (lista resursa koji predstavljaju alternativne vrednosti properti). RDF pruža podršku za opisivanje i komentarisanje RDF iskaza, npr. „iskaz A je netačan“.

RDF ne obezbeđuje mehanizam za opisivanje properti, ni za opisivanje relacija između properti i drugih resursa. To je uloga RDF *Schema* (RDFS) kao semantičkog proširenja RDF-a. RDFS je jednostavan ontološki jezik koji proširuje RDF sa osnovnim primitivama ontološkog modelovanja kao što su klase, instance, properti, hijerarhije klase, hijerarhije i ograničenja properti. Zvanični naziv je *RDF Vocabulary Description Language* jer se koristi za opisivanje RDF rečnika ali je termin *Schema* zadržan zbog istorijskih razloga. RDFS služi za pravljenje objektnog modela iz kog se referenciraju konkretni podaci i koji govori šta stvari zaista znače. Sadrži interpretacije raznih termina upotrebljenih u RDF rečnicima. Obezbeđuje mehanizme za opisivanje grupa povezanih resursa (klase) i relacija među njima.

Koncept RDFS klase je sličan konceptu klase u objektno-orjentisanim programskim jezicima. Klase su strukture sličnih stvari koje se mogu hijerarhijski organizovati. Resursi se definišu kao instance klase ili potklase. RDFS properti se može posmatrati kao atribut klase, to je takođe resurs identifikovan URI-jem. Da bi se ograničila upotreba properti, specificira se njegov domen i opseg navođenjem resursa - označava koji tip resursa može imati ulogu subjekta a koji ulogu objekta u relaciji. Domensko ograničenje specificira koje klase mogu imati dati properti, dok ograničenje opsega limitira vrednosti properti.

GRDDL (*Gleaning Resource Descriptions from Dialects of Languages*) dozvoljava objavljivanje podataka u tradicionalnim formatima (kao što su HTML i XML) i specificira kako da se ti podaci prevedu u RDF. To je metod koji omogućava dobijanje RDF trojki iz XML dokumenata i drugih formata. RDFa (*RDF-in-attributes*) standard proširuje XHTML generalizovanjem atributa meta i link elemenata tako da oni dodaju metapodatke bilo kojim elementima. Definisano je jednostavno mapiranje za izdvajanje RDF trojki. SPARQL (*Simple Protocol and RDF Query Language*) je jezik upita za RDF

podatke na Semantičkom *Web*-u, nalik SQL-u sa formalno definisanim značenjem. Omogućava da aplikacije traže specifične informacije u okviru RDF podataka. Osnovna ideja se sastoji u korišćenju grafičkih *triple* obrazaca koji imaju nepovezane simbole. Procesor SPARQL upita pretražuje RDF graf kako bi našao trojke koje odgovaraju navedenim obrascima. On povezuje nepovezane simbole sa korespondirajućim delovima trojki i selektuje podgrafove RDF grafa. SPARQL nije samo jezik upita već i protokol za pristup RDF podacima.

RDF(S), kombinacija RDF i RDF *Schema*, nije veoma izražajna u poređenju sa drugim ontološkim jezicima. Ne može da izrazi ekvivalentnost koncepata, razdvojenost klasa (*disjointedness*), kardinalnost i specijalne karakteristike svojstva kao što su tranzitivnost, simetričnost, inverznost itd. Ne može da deklariše restrikcije opsega koje važe samo za neke klase. Takođe, nije moguće praviti novu klasu koristeći presek, uniju, komplement drugih klasa ili enumeraciju. Izražajna ograničenja RDF(S)-a su bila glavna motivacija za razvijanje ekspresivnijih jezika Semantičkog *Web*-a. Prepoznavanje ograničenja RDF *Schema*-e je dovelo do razvoja OWL-a.

1.3.4 Ontologije i OWL

Na RDF sloju leži sloj ontologija sa dobro zasnovanim konstrukcijama za detaljnije opisivanje objekata i njihovih relacija, koje prevazilazi opise RDF šeme. Ontologije su vitalne za razvoj Semantičkog *Web*-a - čine osnovu za osposobljavanje programa da rasuđuju o različitim domenima i okruženjima. Obezbeđuju eksplicitnu reprezentaciju semantike što omogućava mašinsko razumevanje informacija pomoću linkova između informacionih resursa i termina ontologije. Simbole ontologije (koncepte i relacije) mogu da interpretiraju i ljudi i mašine. Pošto je značenje simbola formalno definisano, računar može da rezonuje i izvodi iste zaključke kao ljudi. Kombinacija ontologija i *Web*-a ima potencijal za prevazilaženje mnogih problema koji se tiču deljenja i ponovne upotrebe znanja i integracije informacija. Nepromenljivi (*hard-coded*) sistemi koji razumeju određene termine će vrlo verovatno zastareti ili će biti od ograničene koristi posle uvođenja novih termina jer ih neće znati procesirati. Pored toga, upotreba URI identifikatora je beskorisna ako se ne opiše njihovo značenje. Zato su uvedene šeme i ontologije za opisivanje značenja i veza između termina. Program koji razume ontologije će znati kako da novodefinisane termine konvertuje u već poznate i da onda procesira iskaze koje pre nije mogao interpretirati.

Termin ontologija potiče iz velike oblasti filozofije - metafizike i označava veštinu opisivanja vrsta entiteta i relacija među njima. Kao tehnički termin, ontologija je „formalna eksplicitna specifikacija deljene konceptualizacije“ tj. slično formalnoj specifikaciji programa, to je opis koncepata i relacija koje mogu postojati u zajednici agenata. Zajedničke komponente većine ontologija su: individue (instance, objekti), klase (koncepti, tipovi objekata, kolekcije), svojstvu (atributi, karakteristike individua ili klasa), relacije (načini povezivanja klasa i individua), restrikcije (formalno izraženi opisi onoga što mora biti ispunjeno da bi tvđenje važilo). Ontologije mogu biti veoma kompleksne sa hiljadama pojmova ali i veoma jednostavne; variraju u količini izraženih detalja i u opsegu upotrebe. U zavisnosti od nivoa opštosti znanja razlikuju se tri tipa: opšte (obuhvataju domenski-nezavisno znanje), domenske (obuhvataju znanje iz

specifičnog domena) i aplikacione (sadrže znanje potrebno za datu aplikaciju, npr. ontologija koja predstavlja strukturu određenog *Web* sajta). Takođe, postoje sledeći nivoi detalja ontologija: tezaurus (rečnik sinonima), neformalna hijerarhija (nema strogog nasleđivanja - instanca potklase ne mora biti istovremeno i instanca natklase), formalna hijerarhija (strogo nasleđivanje), okviri (klasa ili okvir sadrži proprijetije koje nasleđuju potklase i instance), ograničenja vrednosti proprijetija, opšta logička ograničenja (vrednosti se ograničavaju logičkim ili matematičkim formulama koristeći vrednosti drugih proprijetija) i ekspresivna logička ograničenja (izražajni jezik ontološki jezik dozvoljavaju ograničenja logike prvog reda među pojmovima i detaljnije relacije). Granice između nivoa ekspresivnosti nisu uvek jasne.

U računarskoj nauci, ontologije su najpre usvojene u oblasti veštačke inteligencije da bi olakšale deljenje i ponovnu upotrebu znanja. Danas se koriste u oblastima integracije informacija, kooperativnih informacionih sistema, softverskog inženjerstva baziranog na agentima. Mogu se predstavljati pomoću semantičkih mreža, okvira, objektnih modela, XML-a, RDF-a itd. Interesovanje za ontologije se javilo još 1980-tih godina sa sistemima za reprezentaciju znanja KL-ONE i CLASSIC. Početkom 1990-tih godina KIF (*Knowledge Interchange Format*) je postao standardni jezik za ontološko modelovanje i koristio ga je značajni alat za razvijanje ontologija – *Ontolingua*. Sistemi kao što su KL-ONE, CLASSIC, LOOM, su imali svoj sopstveni ontološki jezik dok je *Ontolingua* sistem mogao da prevodi ontologije između raznih jezika koristeći KIF standard. Krajem 1990-tih je počela primena ontologija na WWW. *Ontobroker* i njegov naslednik *On2broker* su označavali HTML dokumenta sa referencama na ontologije, pored toga su ih koristili za formulisanje upita i izvođenje odgovora. OIL (*Ontology Inference Layer*) jezik je rezultat IST *OntoKnowledge* projekta namenjenog za podršku elektronske trgovine i oblasti upravljanja znanja. U otprilike isto vreme kad je OIL razvijan, DARPA (*Defense Advanced Research Projects Agency*) američka državna organizacija za istraživanja je sponzorirala DAML program (*DARPA Agent Markup Language*). Cilj ovog programa je bio razvoj *markup* jezika za agente kako bi mogli razumeti i procesirati informacije na Web-u. Primarni rezultat projekta, DAML jezik, je spojen sa OIL-om u DAML+OIL jezik. Reformatirana verzija DAML+OIL-a je poslužila kao osnova za nastanak OWL (*Web Ontology Language*) jezika koji je od 2004. godine standard za definisanje ontologija kompatibilnih sa RDF tehnologijom.

OWL je najistaknutiji *markup* jezik za publikovanje i deljenje podataka koristeći ontologije na Internetu. Služi za definisanje *Web* ontologija koje mogu sadržati opise klase, njihovih instanci i relacija među njima. Jedna od prednosti OWL ontologija je dostupnost alata koji mogu da rasuđuju i izvedu logičke zaključke tj. činjenice koje nisu doslovno prisutne u ontologiji, već su određene semantikom. Ovi alati obezbeđuju generičku podršku, koja ne zavisi od posebnog domena. OWL je dizajniran za maksimalnu kompatibilnost sa RDF(S). Predstavlja vokabularnu ekstenziju RDF-a sa bogatijom semantikom; dodeljuje dodatna značenja određenim RDF konstrukcijama. OWL ontologija je RDF graf, odnosno skup RDF trojki. Elementi OWL-a (klase, instance i proprijetiji) se definišu kao RDF resursi i identifikuju pomoću URI-ja; koristi se većina primitiva RDFS modeliranja. Normativna sintaksa za razmenu je RDF/XML. XML serijalizacija je poželjna jer je XML postao široko prihvaćen i već su razvijeni brojni alati za njegovo procesiranje.

Definicija OWL-a je organizovana u tri podjezika sa različitom ekspresivnošću: *OWL Lite*, *OWL DL* i *OWL Full*:

- *OWL Lite* u poređenju sa RDFS-om dodaje lokalna ograničenja opsega, egzistencijalna i jednostavna ograničenja kardinalnosti (dozvoljene su samo vrednosti 0 i 1), koncept jednakosti i razne tipove propertija. Glavni nedostatak je ograničena ekspresivnost (koristi se za modeliranje tezaurusa i jednostavnijih ontologija), dok je prednost u tome što ga korisnici mogu lako razumeti, a projektanti mogu razviti jednostavnije alate za *OWL Lite* nego za njegove sledbenike.
- *OWL DL* nudi sve OWL konstrukcije ali pod određenim uslovima (npr. klasa ne može biti i individua ili properti; properti ne može biti i individua ili klasa). U odnosu na *OWL Lite*, dodaje punu podršku za disjunkciju, negaciju, enumeraciju, ograničenja kardinalnosti i vrednosti. Obezbeđuje maksimalnu ekspresivnost bez gubitka računske kompletnosti i odlučivosti – sva izračunavanja će se izvršiti u konačnom vremenu. *OWL DL* je dobio takvo ime zbog slaganja sa *description logics*, poljem istraživanja odlučivog fragmenta logike prvog reda.
- *OWL Full* je kompletan OWL jezik, ima maksimalnu ekspresivnost i sintaksnu slobodu RDF-a (npr. klasa se može tretirati i kao individua i kao kolekcija individua; *datatype* property može biti inverzno funkcionalan). Koristi sve OWL primitive, kao i proizvoljnu kombinaciju ovih primitiva sa RDF i RDF šemom. Jedan veliki problem je što ne garantuje izračunavanje; toliko je ekspresivan da je neodlučiv. Malo je verovatno da će bilo koji softver za zaključivanje moći da podrži sve karakteristike *OWL Full* jezika.



Svaki od ovih podjezika predstavlja ekstenziju jednostavnijeg prethodnika i u onome što se može legalno izraziti i u onome što se može validno zaključiti. Svaka *OWL Lite* ontologija ili zaključak je validna *OWL DL* ontologija, odnosno zaključak. Takođe, svaka *OWL DL* ontologija ili zaključak je validna *OWL Full* ontologija, odnosno zaključak. Obrnuto ne važi. *OWL Full* se može smatrati ekstenzijom RDF-a, a *OWL Lite* i *OWL DL* ekstenzijom ograničenog pogleda na RDF. Zato je svaki OWL dokument i RDF dokument i svaki RDF dokument je *OWL Full* dokument. Međutim samo su neki RDF dokumenti legalni *OWL Lite* ili *OWL DL* dokumenti (RDF ne postavlja ograničenja kako resursi mogu biti povezani međusobno) pa se mora obratiti pažnja prilikom migriranja RDF dokumenta u OWL.

Struktura OWL ontologije

Standardna inicijalna komponenta OWL ontologije je skup XML deklaracija prostora imena u okviru početnog **rdf:RDF** taga. Prostor imena (*namespace*) koji služi za

nedvosmisleno identifikovanje pojmova iz ontologije, se deklarirše pomoću XML atributa **xmlns**, kratkog prefiksa i vrednosti koja mora biti URI. Na primer

```
xmlns:owl =http://www.w3.org/2002/07/owl#
```

služi za uvođenje OWL rečnika. Termini sa prefiksom **owl:** označavaju pojmove iz prostora imena čiji je URI <http://www.w3.org/2002/07/owl#>. Pošto OWL koristi konstrukcije iz RDF, RDFS i XML *Schema*, na sličan način se uvode RDF i RDFS rečnici (**rdf:**, **rdfs:**), kao i XML *Schema* tipovi podataka (**xsd:**). Pored ovih konvencionalnih deklaracija, navodi se deklaracija date ontologije (i ontologija kao resurs ima svoj jedinstveni identifikator) i opciono deklaracija prefiksa uvezene ontologije. Korišćenje kratkih prefiksa umesto cele URI oznake čini ontologiju mnogo čitljivijom. Imena bez prefiksa pripadaju podrazumevanom prostoru imena, koji se takođe deklarirše (kao prostor imena date ontologije). Skraćenice se mogu definisati i pomoću **ENTITY** elementa u **DOCTYPE**, radi korišćenja u sklopu vrednosti atributa.

Posle uvodnog **rdf:RDF** taga sa *namespace* deklaracijama, obično sledi **owl:Ontology** tag sa metapodacima o dokumentu (*annotation* properti): komentar o datoj ontologiji **rdfs:comment**, string koji označava veziju tekuće ontologije **owl:versionInfo**, referenca na raniju verziju **owl:priorVersion**. Prethodna verzija ontologije može da sadrži iskaze koji protivreče tekućoj verziji. Sa **owl:backwardCompatibleWith** i **owl:incompatibleWith** tagovima se ukazuje na kompatibilnost ili njen nedostatak između verzija ontologije. Kako je Semantički Web inherentno distribuiran, OWL mora dozvoliti prikupljanje informacija iz distribuiranih izvora. Zato on omogućava povezivanje ontologija, uključujući eksplicitno uvođenje celog skupa iskaza iz druge ontologije, publikovane na Web-u i identifikovane URI-jem, pomoću **owl:imports** taga. Importovane informacije se ne mogu izmeniti, niti izbrisati, već im se samo mogu dodati novi podaci. Dodatni tagovi iz ovog dela pripadaju *Dublin Core* (DC) ontologiji, npr. **title**, **creator**, **date**, **description**.

Sledeći podelementi **rdf:RDF** taga čine kôd same ontologije.

Svaka definisana klasa je tipa **owl:Class**. Definicija se sastoji iz uvođenja imena klase i opcione liste restrikcija. **rdfs:subClassOf** je fundamentalni konstruktor koji povezuje specifičniju klasu sa generalnijom klasom. Na primer

```
<owl:Class rdf:ID="Wine">  
  <rdfs:subClassOf rdf:resource="#food;PotableLiquid"/>  
</owl:Class/>
```

definiše klasu Wine kao podklasom PotableLiquid klase iz uvezene ontologije čiji je definisani prefiks food. Definicije mogu biti inkrementalne i distribuirane – dozvoljava se ekstenzija uvezene definicije i inkrementalna konstrukcija veće ontologije. Pomoću skupovnih operatora (komplementa, unije ili preseka) i enumeracije se definišu kompleksnije klase. Element **owl:complementOf** se primenjuje na jednu klasu i označava sve individue iz domena koje ne pripadaju navedenoj klasi.

```
<owl:Class rdf:ID="NonConsumableThing">  
  <owl:complementOf rdf:resource="#ConsumableThing"/>
```

```
</owl:Class>
```

Element **owl:unionOf** kada se primeni na dve klase A i B, kreira novu klasu čiji su članovi sve instance iz klase A ili iz klase B.

```
<owl:Class rdf:ID="Fruit">
  <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#SweetFruit"/>
    <owl:Class rdf:about="#NonSweetFruit"/>
  </owl:unionOf>
</owl:Class>
```

Na sličan način, element **owl:intersectionOf** primenjen na klase A i B precizno označava individue koje istovremeno pripadaju i klasi A i klasi B. Klasa se može specificirati i iscrpnim navođenjem svih njenih članova koristeći **owl:oneOf** konstrukciju. Individue moraju biti prethodno definisane.

```
<owl:Class rdf:ID="WineColor">
  <owl:oneOf rdf:parseType="Collection">
    <WineColor rdf:about="#White"/>
    <WineColor rdf:about="#Rose"/>
    <WineColor rdf:about="#Red"/>
  </owl:oneOf>
</owl:Class>
```

Za OWL klase se pretpostavlja da se preklapaju, ako se ne kaže suprotno. Razdvojenost klase se izražava **owl:disjointWith** konstruktorom. On garantuje da individua koja je član jedne klase ne može biti istovremeno i član druge specificirane klase. Za izražavanje međusobne razdvojenosti skupa koji ima više od dve klase, neophodno je napisati *disjoint* iskaz za svaki par.

```
<owl:Class rdf:ID="NonSweetFruit">
  <owl:disjointWith rdf:resource="#SweetFruit" />
</owl:Class>
```

U slučaju spajanja dve nezavisno razvijene ontologije, korisno je naznačiti da je neka klasa iz jedne ontologije ekvivalentna klasi iz druge ontologije. Onda se može rezonovati na osnovu njihove kombinacije. Iskaz **owl:equivalentClass** definiše da dve klase imaju tačno isti skup instanci.

```
<owl:Class rdf:ID="Wine">
  <owl:equivalentClass rdf:resource="&vin;Wine"/>
</owl:Class>
```

Svaka individua je član **owl:Thing** klase, zato je svaka korisnički definisana klasa implicitno podklasa od **owl:Thing**. Individua se uvodi njenim deklarisanjem kao članom neke klase.

```
<Region rdf:ID="CentralCoastRegion"/>
```

OWL ne koristi *Unique Name Assumption* – samo zato što su dva imena različita, ne znači da se odnose na različite individue. Za dve instance se može reći da su identične

pomoću **owl:sameAs** konstrukcije, što je korisno prilikom ujedinjavanja ontologija sa identičnim individuama. Suprotan efekat ima **owl:differentFrom**. Koristi se u slučajevima kad je bitno osigurati različite identitete.

```
<WineColor rdf:ID="Red">
  <owl:differentFrom rdf:resource="#White"/>
</WineColor>
```

Pogodniji mehanizam za definisanje skupa međusobno distinktnih individua je **owl:AllDifferent**:

```
<owl:AllDifferent>
  <owl:distinctMembers rdf:parseType="Collection">
    <WineColor rdf:about="#Red"/>
    <WineColor rdf:about="#White"/>
    <WineColor rdf:about="#Rose"/>
  </owl:distinctMembers>
</owl:AllDifferent>
```

Pomoću klasa i instanci se definišu samo hijerarhije (*taxonomy*). Propertiji omogućavaju iskazivanje opštih činjenica o članovima klase i specifičnih činjenica o instancama. Dele se na **owl:ObjectProperty** - relacije između instanci, **owl:DatatypeProperty** - povezuju instance sa RDF literalima ili XML *Schema* tipovima podataka (svi OWL alati za rasuđivanje podržavaju **xsd:integer** i **xsd:string** tipove). OWL dozvoljava označavanje klasa, instanci, propertija i same ontologije različitim metapodacima pomoću **owl:AnnotationProperty**. Predefinisani OWL *annotation* propertiji su već pomenuti **rdfs:comment** i **owl:versionInfo**, zatim **rdfs:label** (slično komentaru ne doprinosi logičkoj interpretaciji ontologije; pruža alternativna imena, npr. podršku za više jezika) **rdfs:seeAlso**, **rdfs:isDefinedBy** (referenca na ontologiju koja definiše dati element), i propertiji koji se upotrebljavaju samo za označavanje ontologija: **owl:priorVersion**, **owl:backwardCompatibleWith**, **owl:incompatibleWith**.

Domen i opseg propertija tj. tip instanci ili tip podataka koji mogu biti u relaciji se specificiraju pomoću **rdfs:domain** odnosno **rdfs:range**.

```
<owl:ObjectProperty rdf:ID="hasWineDescriptor">
  <rdfs:domain rdf:resource="#Wine"/>
  <rdfs:range rdf:resource="#WineDescriptor"/>
</owl:ObjectProperty>

<owl:DatatypeProperty rdf:ID="yearValue">
  <rdfs:domain rdf:resource="#VintageYear"/>
  <rdfs:range rdf:resource="#xsd:positiveInteger"/>
</owl:DatatypeProperty>
```

Properti se definiše kao specijalizacija postojećeg propertija, slično klasama, pomoću **rdfs:subPropertyOf** elementa. Kreiranjem hijerarhija, softver za rasuđivanje može da zaključi da ako je neka individua povezana sa drugom pomoću podpropertija, onda je takođe povezana i pomoću nadpropertija.


```

<owl:ObjectProperty rdf:ID="hasColor">
  <rdfs:subPropertyOf rdf:resource="#hasWineDescriptor"/>
</owl:ObjectProperty>

```

Indivdui se može dodati properti sa specifičnom vrednošću.

```

<VintageYear rdf:ID="Year1998">
  <yearValue
    rdf:datatype="&xsd;positiveInteger">1998</yearValue>
</VintageYear>

```

Definisanje karakteristika propertija obezbeđuje veću ekspresivnost podataka – alati generišu moćne zaključke. Ako properti **P** ima karakteristiku **owl:SymmetricProperty**, onda za bilo koje x i y važi $P(x,y) \Leftrightarrow P(y,x)$. Ako **P** ima karakteristiku **owl:TransitiveProperty**, za bilo koje x , y i z važi $P(x,y) \wedge P(y,z) \Rightarrow P(x,z)$. Ako **P** ima karakteristiku **owl:FunctionalProperty**, za svako x , y i z važi $P(x,y) \wedge P(x,z) \Rightarrow y=z$, odnosno funkcionalan properti može imati najviše jednu vrednost za svaku instancu. Ako **P** ima karakteristiku **owl:InverseFunctionalProperty**, za svako x , y i z važi $P(y,x) \wedge P(z,x) \Rightarrow y=z$ odnosno članovi opsega mogu poslužiti kao jedinstveni identifikatori za članove domena. Ako je **P1 owl:inverseOf P2**, za svako x i y važi $P1(x,y) \Leftrightarrow P2(y,x)$, tj. definiše se relacija inverznosti između propertija **P1** i **P2**.

```

<owl:ObjectProperty rdf:ID="producesWine">
  <rdf:type rdf:resource="#owl:InverseFunctionalProperty"/>
  <owl:inverseOf rdf:resource="#hasMaker"/>
</owl:ObjectProperty>

```

Dosadašnja ograničenja propertija predstavljaju globalni mehanizam (primenjuju se na sve instance), dok su properti restrikcije lokalne za datu definiciju klase i ograničavaju opseg propertija. **owl:allValuesFrom** restrikcija ne garantuje postojanje relacije preko datog propertija, ali ako postoji mora biti ka elementima specificiranog opsega. **owl:someValuesFrom** restrikcija opisuje skup individua koje su u barem jednoj relaciji preko datog propertija sa elementima specificiranog opsega (mogu postojati i relacije ka drugim elementima).

```

<owl:Class rdf:ID="TouristGolfPlayer">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Tourist"/>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#plays"/>
      <owl:someValuesFrom rdf:resource="#Golf"/>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>

```

U navedenom primeru, restrikcijom se definiše anonimna klasa stvari koje ispunjavaju datu restrikciju (da su povezane barem sa jednom instancom tipa Golf preko plays propertija). Članovi TouristGolfPlayer klase su oni članovi Tourist klase koji su preko plays propertija povezani sa bar jednom instancom klase Golf. Ali individua

TouristGolfPlayer klase može istovremeno da se bavi i drugim sportovima. **owl:hasValue** specificira klasu individua koje preko datog svojstva imaju barem jednu vezu ka navedenoj vrednosti (instanci ili podatku). Postojanje drugih relacija preko datog svojstva nije ograničeno. Ograničenja kardinalnosti opisuju klasu instanci koje imaju najmanje (**owl:minCardinality**), najviše (**owl:maxCardinality**) ili tačno određen (**owl:cardinality**) broj veza preko datog svojstva. Kombinacijom najmanje i najveće kardinalnosti se može definisati i interval.

```
<owl:Class rdf:ID="Vintage">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasVintageYear"/>
      <owl:cardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

1.3.5 Nerealizovani deo steka Semantičkog Web-a

Slojevi na vrhu steka Semantičkog Web-a sadrže tehnologije koje još uvek nisu standardizovane ili sadrže samo ideje koje tek treba implementirati.

Dok ontologije mogu da izraze znanje o klasama, hijerarhiji klasa, svojstvima itd., pravila imaju komplementarnu ekspresivnost – mogu izraziti znanje u formi „ako A, onda B“. RIF (*Rule Interchange Format*) je predložena komponenta za Semantički Web koju W3C trenutno razvija kao mogući format za razmenu različitih *rule* jezika i *inference* mašina. RIF pravila opisuju kako da se izvode nove informacije iz ontologija. Mašina za zaključivanje je softverski program koji deluje na jednom nivou iznad ontologija tako što pregleda različite ontologije da bi našao nove relacije između termina.

Sloj logike (*Logic*) treba da obezbedi interoperabilni jezik za opisivanje skupa dedukcija koje se izvode iz kolekcija podataka. Računar se prepušta da sam zaključuje koristeći navedene logičke principe.

Sloj dokaza (*Proof*) podrazumeva da računar koristeći logička pravila rezonuje i izvodi dokaz da je neko tvrđenje tačno. Verifikovani dokazi bi se onda prosleđivali kao nove činjenice u sistemu.

Ako bilo ko može da kaže bilo šta (RDF princip), pitanje je kako verovati takvom sistemu. Digitalni potpis treba da obezbedi dokaz da je određena osoba napisala ili se slaže sa nekim dokumentom. Sloj *Digital Signature* omogućava da se sadržaj sloja označi sa pouzdanim poreklom.

U *Trust* sloju korisnik ima važnu ulogu jer on odlučuje da li nekom izvoru informacija treba verovati ili ne. Možemo podesiti kojim potpisima i u kojoj meri će da veruje naš računar. Na primer verujemo svojim prijateljima i to kažemo programu. Svako od naših prijatelja opet veruje nekoj drugoj grupi ljudi itd. Relacije se šire i formiraju „*Web of Trust*“.

1.4 Projekti Semantičkog Web-a

Kako sve više grupa ljudi iz određenog polja ili vokacije razvija taksonomije tj. šeme za predstavljanje informacija iz njihovog domena, alati Semantičkog Web-a (SW-a) dozvoljavaju povezivanje ovih šema i prevođenje termina i time se postepeno povećava broj ljudi i zajednica čiji Web softveri mogu automatski da razumeju jedan drugog. Možda najvidljiviji primeri, iako ograničeni u opsegu, jestu tagging sistemi prisutni na socijalnim sajtovima kao što su *MySpace* i *Flickr*. U ovim šemama, ljudi selektuju zajedničke termine da bi opisali informacije koje nalaze ili postavljaju na neki sajt, što za uzvrat omogućava Web programima da grubo razumeju i nalaze označene informacije. Ali tagovi iz jednog sistema se ne mogu koristiti u drugom, pa se ovakvi sistemi ne mogu srazmerno uvećavati da analiziraju sve informacije na Web-u.

Web konzorcijum je već objavio jezike i tehnologije SW-a potrebne za prelazak ovih granica i velike kompanije ih eksploatišu. *Ordnance Survey*, engleska nacionalna agencija za mapiranje, koristi interno SW da bi preciznije i jeftinije generisala geografske mape. MITRE korporacija primenjuje alate SW-a da bi američkoj vojsci olakšala interpretaciju pravila za kretanje konvoja. *Vodafone Live!*, multimedijalni portal za pristup ring tonovima, igricama i mobilnim aplikacijama, je izgrađen na formatima SW-a koji omogućavaju mnogo brže preuzimanje sadržaja na mobilne telefone korisnika nego ranije. *Joost* koji besplatno stavlja televiziju na Web, takođe koristi SW softver za upravljanje rasporedima i programskim vodičima.

FOAF projekat (*Friend Of A Friend*) započet 2000. godine se smatra prvom socijalnom aplikacijom SW-a jer kombinuje RDF tehnologiju sa zanimanjima socijalnog Web-a. Ilustruje jedan način za povezivanje distribuiranih podataka preko Web-a. To je decentralizovani semantički sistem društvenog umrežavanja; dozvoljava grupama ljudi da opišu mreže bez potrebe za centralizovanom bazom podataka. Ideja je da se dele informacije o osobama i njihovim poznanicima. FOAF zato obezbeđuje standardni SW rečnik za izražavanje osnovnih činjenica (imena, godišta, lokacija i poslova ljudi kao i relacije među njima) pomoću RDF trojki. Svaki profil ima jedinstveni identifikator: npr. e-mail adresu, Jabber ID, URI weblog-a osobe itd. Identifikujuće informacije mogu biti u normalnoj čitljivoj formi ili su šifrirane ako se ne žele izneti lični podaci. FOAF fajl jedne osobe može sadržati linkove ka FOAF stranicama drugih ljudi njegovih poznanika. Pomoću linkova se formira mapa ili graf o tome ko poznaje koga, što računari koriste za dobijanje raznih informacija npr. spisak svih ljudi koji žive u Evropi, ili spisak osoba koje neka dva prijatelja poznaju.

LOD projekat (*Linking Open Data*) je pokušaj kreiranja javno dostupnih, međusobno povezanih RDF podataka na Web-u. RDF linkovi omogućavaju navigaciju od jedne informacione jedinice u nekom izvoru podataka ka srodnim podacima iz drugih izvora. *DBpedia* projekat je pokušaj inteligentnog povezivanja informacija iz sedam miliona *Wikipedia* članaka. RDF se koristi kao fleksibilni model podataka za reprezentaciju i objavljivanje strukturiranih podataka izdvojenih iz *Wikipedia*-e na Web-u. Ovaj projekat treba da omogući SW agentima detaljne pretrage, zaključivanje i postavljanje sofisticiranih upita nad sadržajem *Wikipedia*-e koji dotad nisu bili izvodljivi. Od novembra 2008. godine, *DBpedia* ima skup podataka sastavljen iz 274 miliona RDF trojki i povezan na RDF nivou sa raznim drugim *Open Data* skupovima podataka (*Freebase*, *OpenCyc*, *Eurostat*, *US Census*...).

Još uvek se radi na implementaciji vizije agenata koji automatizuju dnevne zadatke ljudi. Ali najviše napretka se dešava u poljima bioloških nauka i zdravstva. Istraživači iz ovih disciplina su napravili studije realnih sistema koji ukazuju na potencijale SW-a.

Tradicionalni model za lekove je da jedna doza svima odgovara. Danas, međutim, se bolje razumevanje biologije kombinuje sa alatima koji mogu predvideti najefektniju dozu za datu osobu. Izazov predstavlja integracija raznih skupova podataka (medicinski zapisi o svakoj osobi, naučni izveštaji o testiranju lekova i njihovim neželjenim pratećim dejstvima..). Klasični alati za baze podataka se ne mogu nositi sa ovom kompleksnošću – samo održavanje je teško jer svaki put kad se novo znanje uključi u neki izvor podataka, drugi izvori povezani s njim se moraju redom reintegrirati. Naučnici u *Lilly* farmaceutskoj kompaniji koriste semantičke alate za prevođenje brojnih nekompatibilnih bioloških opisa u jedan zajednički fajl, čime se značajno povećava efikasnost u pronalaženju novih lekova, a industrija će okreće ka personalizovanoj medicini. Tradicionalni pristup otkrivanja genetskih uzroka bolesti je pretraga gena koji se različito ponašaju u normalnom i obolelom tkivu. Rezultat može biti na stotine gena koji potencijalno doprinose nekoj bolesti. Pred istraživačima je onda naporan zadatak prolaska kroz više baza podataka za svaki od izdvojenih gena, kako bi razaznali koji su najverovatniji krivci za poremećaj. Istraživački tim medicinskog centra *Cincinnati Children's Hospital* koji uključuje SW konsultanta, koristi tehnologije SW-a u nalaženju genetskih uzroka kardiovaskularnih bolesti. Najpre preuzimaju baze podataka sa relevantnim informacijama kao što su *Gene Ontology*, *MeSH*, *OMIM*, *Entrez Gene*. Pošto su baze različitog porekla i neusaglašenih formata, sve informacije se prevode u RDF formu i smeštaju u SW bazu; primenom *Jena* i *Protege* softvera se integriše znanje. Zatim se pomoću rangirajućeg algoritma definišu prioriteta stotine sumnjivih gena. Procenom rangiranih informacija i relacija gena prema simptomima obolenja, softver identifikuje nekoliko gena koji su jako povezani sa određenom bolesti. Istraživači potom razmatraju efekte mutacije tih gena radi dobijanja novih tretmana lečenja.

SAPPHIRE sistem za izveštavanje o javnom zdravstvu, razvijen na univerzitetu u Hjustonu (*University of Texas Health Science Center*), integriše široki opseg podataka iz naučne literature, lokalnih zdravstvenih centara, bolnica, kako bi unapredio detektovanje, analizu i reagovanje na opšte zdravstvene probleme u nastanku. Fleksibilnost ovog sistema dozvoljava njegovu efikasnu primenu u različitim kontekstima; informacije ujedinjene iz mnogih izvora se koriste u različite svrhe. Jedan primer je praćenje širenja influence: SAPPHIRE klasifikuje simptome pacijenata- groznicu, kašalj, bolno grlo- kao potencijalne slučajeve gripa i automatski ih prijavljuje Centru za kontrolu i prevenciju bolesti (*Center for Disease Control*). Generisanje izveštaja bi inače morale raditi medicinske sestre i proces bi duže trajao. Sistem je takođe bio uspešan u brzom identifikovanju izbijanja gastrointestinalnih i respiratornih obolenja kod evakuisanih lica posle uragana Katrine. Ovim se ilustruje važna karakteristika SW sistema: kada se jednom konfigurišu za neki opšti problem, mogu se lako prilagoditi za razne situacije iz date oblasti.

Klasični računarski sistemi podrške kliničkim odlukama (*Clinical Decision Support*, CDS) koje koriste lekari, su baze podataka sa poslednjim znanjem o zdravstvenim tretmanima. Svaka bolnica ili mreža lekara ima svoj prilagođeni CDS sistem; međusobno oni često nisu kompatibilni a to ne odgovara potrebama personalizovane medicine. Pored

toga, svaki put kad se desi neki pomak u dijagnozama ili kliničkim procedurama, administratorima treba mnogo vremena da ažuriraju i prerade njihove sisteme. Da bi popravila ovu situaciju, *Agfa HealthCare* kompanija je konstruisala CDS sistem baziran na SW tehnologijama. Kada se unese promena u neki deo sistema, zapisi koje treba promeniti u drugim delovima sistema ili u sistemu druge institucije se automatski ažuriraju. Implementiranjem ovakvih sistema u mreži zdravstvene zaštite, baze medicinskog znanja postaju inteligentnije, lakše i jeftinije za upotrebu.

2. ARHITEKTURA I ORGANIZACIJA RAČUNARA

Arhitektura računara je ključna komponenta računarskog inženjerstva. Predstavlja skicu i funkcionalni opis zahteva i implementacije za razne delove računara. Obuhvata konceptualno projektovanje svih velikih podsistema računarskog sistema – procesora, memorije i I/O sistema. Dizajn procesora počinje projektovanjem skupa instrukcija koje treba da izvršava, zatim obuhvata dizajn aritmetičkog i logičkog hardvera za obavljanje izračunavanja, dizajn skupa registara koji treba da sadrže operande operacije, dizajn upravljačke jedinice koja sprovodi izvršavanje instrukcija i konekcije (ili lokalne magistrale) koje omogućavaju komunikaciju ovih komponenata. Dizajn memorijskog sistema koristi čitav spektar komponenata sa različitim karakteristikama da bi formirao celokupni sistem koji ima neophodnu brzinu i kapacitet za izvršavanje namenjenih aplikacija uz pristupačnu cenu. Cilj dizajna I/O sistema je što brže i efikasnije unošenje programa i podataka u memoriju (i procesor) i dostavljanje rezultata izračunavanja korisniku (ili drugom računarskom sistemu).

Dok se arhitektura računara odnosi na atribute sistema koji su vidljivi programeru (imaju direktan uticaj na logičko izvršavanje programa), organizacija računara razmatra kako su sastavni delovi sistema povezani i kako funkcionišu međusobno da bi implementirali apstraktnu sliku tog računarskog sistema (arhitektonsku specifikaciju). Organizacioni atributi podrazumevaju detalje hardvera koji su transparentni za programera, npr. upravljački signali, interfejs između računara i perifernih uređaja, korišćena memorijska tehnologija.

2.1 Osnove arhitekture računara

Viši programski jezici skrivaju detalje arhitekture od programera, međutim njihovo poznavanje može pomoći u pisanju efikasnijih programa. Razumevanjem asemblerskog jezika (simboličke reprezentacije mašinskog jezika) se dobija uvid u rad procesora, i uopšte u arhitekturu sistema.

Dizajn skupa mašinskih instrukcija (koje procesor može da izvršava) je jedna od najkompleksnijih i najviše analiziranih faza računarskog dizajna jer utiče na mnoge aspekte računarskog sistema. Ipak fundamentalna pitanja dizajna skupa instrukcija ostaju sporna:

- repertoar operacija: koje operacije obezbediti i koje složenosti treba da budu,
- tipovi podataka: nad kojim različitim tipovima podataka treba izvršavati operacije
- format instrukcija: dužina instrukcija u bitima, broj adresa, veličina polja itd.
- registri: broj procesorskih registara koje mogu upotrebiti instrukcije i njihova namena
- adresiranja: načini za specificiranje adrese operanda.

Organizacija von Neumann-ove mašine

Računar von Neumann arhitekture ima jednu procesorsku jedinicu i jednu memoriju u kojoj se smeštaju i podaci i instrukcije. Procesor ne može da čita instrukciju i da čita/upisuje podatak u memoriju istovremeno jer se koristi ista magistala za prenos

instrukcija i podataka. Ovo je u suprotnosti sa *Harvard* arhitekturom koja ima posebnu memoriju za podatke i posebnu memoriju za instrukcije. Pored toga, dve memorije se mogu razlikovati po tehnologiji implementacije, širini memorijske reči, strukturi adrese. Procesori modernog dizajna imaju osobine oba tipa arhitekture. Keš memorija je podeljena na *instruction cache* i *data cache*, dok glavna memorija sadrži zajedno instrukcije i podatke.

Formati instrukcija

Instrukcije se mogu klasifikovati na osnovu broja operandata na:

- troadresne (format oblika op A, B, C ; 2 izvorišna i 1 odredišni operand),
- dvoadresne (format oblika op A, B ; 2 operanda, jedan je istovremeno i izvorišni i odredišni),
- jednoadresne (format oblika op A ; specificira se 1 izvorišni operand a 2. izvorišni i istovremeno odredišni operand je podrazumevano akumulator), i
- nulaadresne koje koriste stek operacije push i pop (operandi i odredište operacija su na steku; pristupa se pomoću SP registra procesora).

Faze izvršavanja instrukcije

Izvršavanje instrukcije se sastoji iz sledećih faza:

- faza čitanje instrukcije (iz operativne memorije sa mem. lokacije određene sadržajem PC registra se čita instrukcija i smešta u prihvatni registar IR u procesoru; PC se inkrementira)
- faza formiranje adrese i čitanje operanda (u zavisnosti od načina adresiranja specificiranog u instrukciji formiraju se adrese koje se zatim koriste za dobavljanje operandata u odgovarajuće registre procesora)
- faza izvršavanje operacije (u zavisnosti od polja instrukcije koje određuje tip operacije, nad vrednostima operandata se u aritmetičko-logičkoj jedinici realizuje odgovarajuća operacija; rezultat operacije se smešta na lokaciju specificiranu sadržajem instrukcije)
- faza opsluživanje prekida (sadržaji programskog brojača i programske statusne reči se stavljaju na stek; sadržaji registra broja ulaza i registra IVTP se sabiraju čime se dobija adresa u memoriji na kojoj se nalazi adresa prekidne rutine koja treba da se izvršava; adresa prekidne rutine se upisuje u PC)

Faza formiranje adrese i čitanje operanda se realizuje samo za adresne instrukcije. Faza opsluživanje prekida se realizuje samo ako je u toku izvršavanja prethodnih faza došlo do generisanja signala prekida.

Registri i registarski fajlovi

Registri su brze memorijske lokacije u procesoru koje služe prilikom izvršavanja operacija i drugih kalkulacija. Registri procesora se mogu podeliti na adresabilne

(programski dostupni) i interne (nisu programski dostupni). Mogu da imaju opštu namenu ili specifičnu namenu.

Registri neophodni za operacije čitanja i upisa u operativnu memoriju su: MAR –memory address registar i MDR – memory data registar. PC – program counter je registar koji sadrži adresu instrukcije koja sledeća treba da se izvršava. IR – instruction registar služi za čuvanje instrukcije prethodno pročitane iz memorije. PSW – program status word registar čuva informacije o statusu trenutnog izvršavanja. SP – stack pointer registar ukazuje na adresibilnu lokaciju na steku.

Različite arhitekture računara imaju različit skup registara. Skupovi se mogu razlikovati po broju registara, tipu registara kao i njihovim veličinama ili namenama. Pentium procesor ima tri grupe registara: registri opšte namene, segmentni registri, i instruction pointer(PC) i flag registar. MIPS procesor sadrži 32 registra opšte namene.

Registarski fajl je niz procesorskih registara u procesoru, koji su povezani tako da je moguć istovremeno upis podatka u selektovani registar i čitanje iz njega. Registar se selektuje pomoću njegove adrese tj. broja koji se dovodi na ulaz fajla.

Tipovi instrukcija i načini adresiranja

Tipovi standardnih instrukcija su:

- instrukcije prenosa (MOVE, LOAD, STORE, PUSH, POP, IN, OUT)
- aritmetičke instrukcije (ADD, SUBTRACT, MULTIPLY, DIVIDE, INCREMENT, DECREMENT)
- logičke instrukcije (AND, OR, XOR, NOT, SHIFT, ROTATE)
- instrukcije skoka (bezuslovni skok JUMP, uslovni skokovi BRANCH-IF-CONDITION, skok na potprogram JSR, povratak iz potprograma RTS, instrukcija prekida INT, povratak iz prekidne rutine RTI)
- instrukcije postavljanja indikatora u PSW (INTD, INTE, TRPD, TRPE, VARD, VARE, EDGD, EDGE)

Nestandardne instrukcije (ne postoje kod RISC računara) su:

- instrukcije nad celobrojnim veličinama promenljive dužine
- string instrukcije (MOVC, MOVTC, CMPC, LOCC, SKPC, MATCHC)
- decimalne instrukcije
- instrukcije kontrole petlji (ACB, AOB, BBC, BBS, CASE, SOB)

Načini adresiranja koji specificiraju kako se određuje lokaciju operanda su:

- neposredno adresiranje (vrednost operanda se nalazi u samoj instrukciji; nije zgodno kad treba promeniti vrednost operanda)
- registarsko direktno adresiranje (operand se nalazi u jednom od registara podataka ili registara opšte namene)
- registarsko indirektno adresiranje (u instrukciji se navodi ime jednog od adresnih registara ili registara opšte namene koji sadrži adresu mem. lokacije operanda)
- memorijsko direktno adresiranje (adresa memorijske lokacije operanda se navodi u instrukciji)

- memorijsko indirektno adresiranje (u instrukciji se navodi memorijska lokacija koja sadrži adresu lokacije operanda)
- bazno adresiranje sa pomerajem (operand se nalazi na mem. lokaciji čija adresa se dobija sabiranjem sadržaja jednog od baznih registara i pomeraja)
- indeksno adresiranje sa pomerajem (operand se nalazi na mem. lokaciji čija adresa se dobija sabiranjem sadržaja jednog od indeksnih registara i pomeraja)
- registarsko indirektno adresiranje sa pomerajem (operand se nalazi na mem. lokaciji čija adresa se dobija sabiranjem sadržaja jednog od registara opšte namene i pomeraja)
- bazno-indeksno adresiranje sa pomerajem (operand se nalazi na mem. lokaciji čija adresa se dobija sabiranjem sadržaja jednog od baznih registara, jednog od indeksnih registara i pomeraja. Umesto baznog i indeksnog registra mogu se specificirati dva registra opšte namene.)
- postdekrement adresiranje i preinkrement adresiranje (operand se nalazi u memoriji na adresi određenoj sadržajem jednog od adresnih registara ili registra opšte namene. Kod postdekrement adresiranja sadržaj specificiranog registra se smanji za 1 posle pristupa operandu, a kod preinkrement adresiranja sadržaj registra se poveća za 1 pre pristupa)
- relativno adresiranje (operand se nalazi u memoriji na adresi dobijenoj sabiranjem sadržaja programskog brojača i pomeraja)

Poziv potprograma i mehanizam povratka

Instrukcija skoka JSR (jump to subroutine) uzrokuje prelazak sa izvršavanja programa na izvršavanje potprograma. Vrednost programskog brojača se stavlja na stek, a zatim se u programski brojač upisuje adresa potprograma koji treba da se izvrši. Time se prelazi na izvršavanje potprograma. Na kraju potprograma uvek postoji instrukcija RTS (return from subroutine) kojom se restaurira vrednost programskog brojača sa steka u programski brojač što omogućava nastavak izvršavanja programa od mesta gde je prekinut pozivom potprograma.

Programiranje u asemblerskom jeziku

Programiranjem u asemblerskom jeziku se može dobiti manji i brži mašinski kod u odnosu na kod dobijen prevođenjem programa pisanih u jezicima višeg nivoa. Ovo je naročito bitno u slučaju *embedded* i *portable* aplikacija. Takođe, programer u asemblerskom jeziku može imati pristup delovima hardvera kojima ne bi mogao pristupiti korišćenjem jezika višeg nivoa.

Mašinski program čini kolekcija mašinskih instrukcija predstavljenih nizom nula i jedinica. Asemblerski program je simbolička (razumljivija) predstava mašinskog programa. Svaka linija asemblerskog programa se može podeliti u četiri polja: labela (simboličko ime za mem. adresu), mnemonik operacije, polje operanda i komentar za lakše razumevanje. Specijalni program-assembler konvertuje asemblerski program u odgovarajući mašinski program. U svom radu assembler koristi barem tri tabele (tabela mnemonika, tj. skraćenica koje se koriste umesto mašinskih instrukcija, tabela simbola, koji se koriste umesto brojeva, i tabela direktiva, pseudoinstrukcija tj. komandi namenjenih assembleru koje ne korespondiraju nekoj mašinskoj instrukciji –npr. kad assembler treba da rezerviše mem. prostor za neki podatak). U prvom prolazu kroz kod,

asembler generiše tabelu simbola i popunjava je njihovim binarnim vrednostima. Prilikom drugog prolaza, asembler koristi tabelu simbola kao i druge tabele da bi generisao objektni program. Ako postoji potreba za tim, linker povezuje module objektnih programa u jedan izvršni program. Loader zatim učitava izvršni program u operativnu memoriju i započinje njegovo izvršavanje.

Mehanizam prekida

Mehanizam prekida omogućava prekidanje u izvršavanju tekućeg programa i skok na prekidnu rutinu. Na kraju izvršavanja svake instrukcije u programu, proverava se da li se u toku njenog izvršavanja desio zahtev za prekid. Ako nije, procesor izvršava sledeću instrukciju programa, a ako se desio zahtev, počinje opsluživanje zahteva za prekid. Kontekst procesora –programski dostupni registri se čuvaju na steku da bi se po povratku iz prekidne rutine obezbedilo isto stanje kao da nije bilo prekida. Adresa prekidne rutine koja treba da se izvršava, se dobija sabiranjem broja ulaza u IV tabelu sa sadržajem IVTP registra (*interrupt vector table pointer*). Broj ulaza u IV tabelu može da bude fiksna vrednost, može biti određen adresnim delom instrukcije INT ili su to vrednosti koje kontroleri periferija šalju procesoru. U okviru opsluživanja zahteva se još u PSW registru brišu biti I i T (bit maskiranje svih maskirajućih prekida i bit prekid posle svake instrukcije) a L biti nivoa prioriteta tekućeg prekida se postavljaju u slučaju maskirajućeg prekida. U slučaju da stigne više zahteva za prekid, redosled njihovog prihvatanja je definisan prioritetima različitih vrsta prekida. Najviši prioritet ima prekid izazvan instrukcijom INT, slede procesorski prekidi zbog nekorektnosti u izvršavanju instrukcije, zatim spoljašnji nemaskirajući prekidi (generišu uređaji koji kontrolišu napajanje, rad memorije itd.), spoljašnji maskirajući prekidi (generišu kontroleri periferija) i prekid posle svake instrukcije (kada je zadat takav režim rada procesora; postavljen bit T registra PSW). Takođe upisivanjem odgovarajuće vrednosti u registar maske IMR mogu se selektivno maskirati maskirajući prekidi čime se zabranjuje njihovo opsluživanje. Posebnim instrukcijama se u I razred registra PSW mogu upisati 0 /1 čime se dozvoljava /zabranjuje opsluživanje svih maskirajućih prekida (bez obzira da li su maskirani IMR registrom). Kada procesor izvršava prekidnu rutinu može stići novi zahtev za prekid. On se može prihvatiti u prekidnoj rutini (gnežđenje prekida) ili po povratku u glavni program, što opet zavisi od prioriteta i maskiranja. Poslednja instrukcija u svakoj prekidnoj rutini je RTI koja restauriranjem konteksta procesora sa steka, omogućava povratak u glavni program i nastavak izvršavanja od mesta gde je bilo prekinuto izvršavanje.

2.2 Aritmetika računara

Dva glavna pitanja dizajna računarske aritmetike, koja važe i za celobrojnu aritmetiku i za aritmetiku veličina u pokretnom zarezu, su način reprezentacije brojeva i algoritmi korišćeni za osnovne aritmetičke operacije. Za razliku od klasične aritmetike, računarsku aritmetiku karakteriše limitirana preciznost. Ograničenje se ogleda prilikom izračunavanja brojeva koji su manji ili veći od predefinisanih granica opsega; ove

anomalije (zване *overflow* i *underflow*) uzrokuju izuzetke i prekide. Dodatni izazov predstavlja aproksimacija realnih brojeva u aritmetici veličina u pokretnom zarezu.

Reprezentacija celih brojeva

Celobrojna veličina bez znaka predstavljena pomoću binarne reči dužine n bitova

$(a_{n-1}a_{n-2}...a_1a_0)$ ima vrednost $\sum_{i=0}^{n-1} 2^i a_i$.

Celobrojne veličine sa znakom se mogu predstaviti na više načina:

- reprezentacija kao znak i veličina (najstariji bit se interpretira kao znak broja a preostali bitovi kao veličina; prilikom izvršavanja operacija se najstariji bit razmatra posebno)
- reprezentacija u prvom komplementu (negativni broj se predstavlja tako što se veličina napiše u binarnom obliku i onda svaki pojedinačni bit komplementira; u slučaju postojanja prenosa, neophodno je korigovati rezultat operacija)
- reprezentacija u drugom komplementu (negativni broj se predstavlja tako što se veličina napiše u binarnom obliku, svaki pojedinačni bit komplementira i sa rezultatom se sabere jedinica; svi bitovi imaju isti tretman i prenos se ignoriše prilikom operacija).

Algoritmi za uobičajene aritmetičke operacije

Navedeni algoritmi su za brojeve predstavljene u drugom komplementu.

Sabiranje: $s_i = a_i \oplus b_i \oplus c_{i-1}$, $c_i = a_i b_i + a_i c_{i-1} + b_i c_{i-1}$; gde su

a_i, b_i biti brojeva A i B sa pozicije i

c_i prenos sa pozicije i

s_i rezultat sabiranja bitova sa pozicije i .

Oduzimanje se izvršava na isti način kao sabiranje jer važi da je $B - A = B + \bar{A} + 1$.

Množenje:

Booth-ov algoritam

Proces se odvija u n koraka, gde je n broj bita množioca Q . U svakom koraku se posmatraju dva uzastopna bita množioca (počinje se sa bitima $q_0 q_{-1}$, $q_{-1} = 0$). Ako su te vrednosti 01, vrednost n -bitnog A registra (inicijalizovanog na 0) se sabira sa množenikom M i rezultat se upisuje u A. Ako su uzastopne vrednosti 10, vrši se oduzimanje $A \leftarrow A - M$. Ako su vrednosti 00 ili 11, ništa se ne preduzima. U sva četiri slučaja se dalje vrši aritmetičko pomeranje udesno konkatencije AQ.

Rezultat množenja $M \times Q$ je konkatencija AQ.

Algoritam se može unaprediti ako se posmatraju tri uzastopna bita množioca.

Deljenje:

Non-restoring algoritam

Korak #1: n puta raditi sledeće

- ako je sadržaj $A > 0$ (A je $(n+1)$ -bitni registar inicijalizovan na 0), treba pomeriti ulevo konkatenciju AQ (Q- količnik, inicijalno jednak X- deljeniku) i $A \leftarrow A - D$ (D- delilac); inače ASL(AQ) i $A \leftarrow A + D$

- ako je $A > 0$, postaviti $q_0 = 1$, inače $q_0 = 0$

Korak #2: ako je $A < 0$, uraditi $A \leftarrow A + D$.

Rezultat deljenja X/D je količnik Q i ostatak A .

Značaj opsega, preciznosti i tačnosti u računarskoj aritmetici

Preciznost označava broj bitova koji se koristi za predstavljanje vrednosti (npr. 8, 16, 32, 64). Dati broj bitova ograničava skup vrednosti koje se mogu tačno prikazati. Tako koristeći n bitova, mogu se predstaviti celi brojevi bez znaka u opsegu od 0 do $2^n - 1$, celi brojevi sa znakom u drugom komplementu u opsegu od -2^{n-1} do $2^{n-1} - 1$ itd. Rezultat operacija da bi bio tačan, mora biti u definisanom opsegu koji važi za operande inače se događa prekoračenje. Ograničen broj bitova često može prouzrokovati određeni gubitak u tačnosti (npr. greške prilikom zaokruživanja). Prednost reprezentacije veličina u pokretnom zarezu u odnosu na celobrojne veličine je mnogo veći opseg vrednosti ali *floating-point* operacije mogu dovesti do neočekivano nepreciznih rezultata.

Reprezentacija realnih brojeva

U skladu sa IEEE standardom, definicija realnog broja je:

$$R = (-1)^s 2^E (b_0 b_1 b_2 \dots b_{p-1}) \text{ gde je } s, b_i \in \{0, 1\}, E_{\min} \leq E \leq E_{\max}.$$

Reprezentacija veličine u pokretnom zarezu ima tri polja: znak s , eksponent e i razlomljeni deo-mantisa m . Da bi se olakšalo izvođenje operacija, veličine u pokretnom zarezu se uvek predstavljaju u normalizovanoj formi (krajnji levi bit mantise mora biti 1). Za veličine u pokretnom zarezu, IEEE definiše jednostruki, jednostruki prošireni, dvostruki i dvostruki prošireni format (*single-precision*, *single-precision extended*, *double-precision* i *double-precision extended*).

Karakteristike jednostrukog i dvostrukog formata su:

- fiksna dužina u bitovima (32 za jednostruki, 64 za dvostruki format)
- fiksne dužine polja ($e=8$, $m=23$ bita za jednostruki i $e=11$, $m=52$ bita za dvostruki)
- jedan skriveni bit koji se podrazumeva da je jedinica (mantisa je oblika $1.m$)
- koristi se pomeraj za predstavljanje i poz. i neg. eksponenata (+127 za jednostruki i +1023 za dvostruki format) pa važi $E = e - \text{pomeraj}$

Prošireni formati nemaju fiksnu dužinu ($e \geq 11, m \geq 32$ tj. ukupno barem 44 bita za jednostruki prošireni odnosno $e \geq 15, m \geq 64$ tj. ukupno barem 80 bitova za dvostruki prošireni format), nemaju definisan pomeraj i ne koriste skriveni bit.

Algoritmi za uobičajene operacije sa veličinama u pokretnom zarezu

Sabiranje/Oduzimanje:

1. Ako nisu isti, treba izjednačiti eksponente operanada (mantisu broja sa manjim eksponentom poravnati prema drugom operandu).
2. Izvršiti sabiranje/oduzimanje mantisa.
3. Normalizovati rezultat.

Množenje:

1. Rezultujući eksponent je zbir eksponenata operanada.
2. Izvršiti množenje mantisa.
3. Normalizovati rezultat.

Deljenje:

1. Rezultujući eksponent je razlika eksponenata deljenika i delioca.
 2. Izvršiti deljenje mantisa.
 3. Normalizovati rezultat.
- Kod svih operacija ako je potrebno, zaokružiti rezultat na kraju.

Konverzija između celih i realnih brojeva

Kada kompajler pretvara realni broj u celi broj, bilo implicitnom ili eksplicitnom konverzijom (*cast*), dešavaju se greške zaokruživanja ili skraćivanja. Kako se razlomljeni deo gubi zauvek, greške ponekad mogu da imaju katastrofalne posledice. Problemi nastaju i kad programer vrši konverziju između različitih *floating-point* tipova. Npr. pretvaranje *double* vrednosti u *float* vrednost će takođe uzrokovati grešku zaokruživanja. Pošto je konverzija mašinski zavisna, programer se ne može osloniti na njenu konzistentnost.

Aritmetika proizvoljne preciznosti

Za korisnike koji zahtevaju zagarantovanu tačnost u računarskoj aritmetici, a spremni su da prihvate veoma sporo izvršavanje, rešenje je aritmetika proizvoljno velike preciznosti. Naravno, ni u ovom slučaju preciznost ne može da bude beskonačna, ali je tipično ograničena jedino količinom raspoložive memorije u sistemu. Implementacija operacija je u softveru, umesto u hardveru mašine. Većina softverskih paketa ove namene smešta vrednosti u nizove cifara promenljive dužine, za razliku od standardne računarske aritmetike koja za to koristi fiksni broj bitova. Iako postoje različiti nivoi preciznosti, moguće je dizajnirati jedan jedini skup algoritama pogodan za sve. Algoritam množenja će npr. povećati preciznost odredišta koliko je potrebno da bi zadovoljio dodatnu preciznost rezultata operacije, dok bi se u standardnom sistemu odbacili suvišni bitovi ili bi se zaokružila vrednost rezultata.

Značajnu primenu ova aritmetika ima u kriptografskim sistemima koji koriste veoma velike cele brojeve koje napadači ni sa najboljim mašinama ne mogu da izračunaju u nekom konačnom vremenu.

Implementacija aritmetičke jedinice

Aritmetička jedinica je kombinaciona mreža koja realizuje neki podskup aritmetičkih operacija. Implementira se kao kompozicija logičkih elemenata (I, ILI, NE...) i standardnih kombinacionih modula (multiplekser, sabirač,...).

Za realizaciju operacije sabiranja/oduzimanja se najčešće koristi CRT (*carry-ripple through*) ili CLA (*carry-look-ahead*) sabirač. Sabirač za jedan razred se naziva potpuni sabirač (*full-adder* FA). Paralelni sabirač sa kaskadnim generisanjem prenosa tj. CRT sabirač se dobija kaskadnom vezom potpunih sabirača. Njegova glavna mana je očekivano veliko kašnjenje proporcionalno broju razreda. Kod CLA sabirača je razbijen lanac zavisnosti između prenosa pomoću podfunkcija *carry generate* i *carry propagate*.

2.3 Organizacija i arhitektura memorijskog sistema

Iako na izgled konceptualno jednostavna, računarska memorija ispoljava možda najveći opseg tipova, tehnologija, organizacija, performansi od svih delova računarskog sistema. Pošto ni jedna tehnologija ne predstavlja optimalan izbor za zadovoljenje svih zahteva (brzina, kapacitet, cena), tipični računarski sistem sadrži hijerarhiju memorijskih podsistema.

Za optimalno iskorišćenje resursa podaci se kompresuju pre upisa u memoriju. Pored uštede prostora, potrebno je manje vremena za prenos takvih kompaktnih podataka. Bez obzira na to koji je metod kodiranja upotrebljen, medijum za smeštanje podataka ne može biti potpuno bez greški (učestanost pojavljivanja je direktno srazmerna broju bita po kvadratnom milimetru). Većina modernih memorijskih sistema uključuje logiku za detektovanje i ispravljanje grešaka.

Hijerarhija memorijskog sistema

Još 1946. godine je primećeno da memorija treba da bude organizovana u hijerarhiju u kojoj bi veće i sporije (i jeftinije) memorije dopunjavale manje i brže (i skuplje) memorije. Cilj ovakvog dizajna je da performanse sistema budu kao da je sistem sastavljen samo od brzih jedinica memorije, a da cena takvog sistema bude uglavnom određena cenom sporije memorije.

Na vrhu hijerarhije su registri procesora (lokacije za čuvanje podataka u procesoru) kao najmanje, najbrže i najskuplje memorijske jedinice. Sledi keš memorija –mala, brza i skupa, zatim operativna memorija –veća i jeftinija od keša, relativno spora. Sledeće u hijerarhiji su mnogo veće, sporije i jeftinije magnetne memorije kao što je hard disk.

U datom trenutku, memorija na višem nivou sadrži podskup memorijskog prostora nižeg nivoa. Efikasnost hijerarhije je zasnovana na fenomenu lokaliteta da za dati period vremena, programi više puta referenciraju ograničeno područje u memoriji. Prostorni lokalitet je fenomen da kada se referencira adresa velika je verovatnoća da će uskoro biti referencirana neka adresa koja se nalazi blizu prethodno referencirane adrese (npr. sekvencijalno izvršavanje instrukcija u programu). Vremenski lokalitet izražava veliku verovatnoću da jednom referencirana adresa bude ponovo referencirana (npr. instrukcija u petlji). Kada procesor traži neki element, prvo se proverava da li se on nalazi u memoriji prvog nivoa. Ako se ne nalazi tu (u slučaju promašaja), pretražuje se memorija sledećeg nivoa. Ovaj process se ponavlja sve dok se ne nađe traženi element. Verovatnoća pogotka (pronalska traženog elementa) se zove hit ratio, dok je verovatnoća promašaja miss ratio. U slučaju sistema sa tri nivoa hijerarhije, prosečno memorijsko vreme pristupa je:

$t_{av} = t_1 + (1 - h_1)t_2 + (1 - h_2)t_3$, gde su h_1 i h_2 hit ratio za prvi odnosno drugi nivo, a t_1, t_2, t_3 vremena pristupa za nivoe jedan, dva i tri respektivno.

Kodiranje, kompresija podataka, integritet podataka

Integritet podataka se odnosi na konzistentnost i tačnost podataka.

Kompresija podataka (ili *source coding*) je proces kodiranja informacija sa manjim brojem bitova nego što bi imala nekodirana reprezentacija. Kompresija je korisna jer

pomaže da se smanji upotreba skupih resursa kao što je npr. prostor na disku. Dizajn podrazumeva kompromise između stepena kompresije, gubitka tačnosti podataka i računarskih resursa neophodnih za kompresiju i dekompresiju podataka.

Kompresija se sastoji iz transformacije niza simbola u niz kodova (kodiranje) na osnovu datog modela (skup podataka i pravila na osnovu kojih se donose odluke prilikom kodiranja). Da bi bila efikasna, rezultujući niz mora biti manji od polaznog.

Huffman-ovo kodiranje podrazumeva da simbol sa većom verovatnoćom pojavljivanja generiše kod sa manjim brojem bitova (obratno za simbol s manjom verovatnoćom pojavljivanja). Aritmetičko kodiranje ne generiše pojedinačni kod za svaki simbol već kod za ceo niz. Svaki simbol koji se dodaje u niz, inkrementalno menja izlazni kod.

Tehnike kompresije podataka se mogu podeliti na dva tipa: kompresije bez gubitaka (*lossless*) ili sa gubicima (*lossy*). Kompresije sa gubicima prihvataju određeni gubitak tačnosti podataka radi postizanja veće kompresije (može se podesiti nivo kvaliteta). Koristi se za kompresiju vizuelnih i audio podataka kod kojih se određeni gubitak kvaliteta može tolerisati zahvaljujući ograničenju ljudskog senzornog sistema. Primeri standarda su JPEG i MPEG za slike odnosno video. Međutim za izvršne programe ili tekstove, gubitak samo jednog bita može biti katastrofalan. Tada se koriste tehnike koje garantuju generisanje duplikata iz kojeg je moguće rekonstruisati originalne podatke. Kompresija bez gubitaka je moguća zahvaljujući statističkoj redundantnosti koju ima većina podataka. Obično koristi statističko modeliranje (simboli se kodiraju na osnovu verovatnoća njihovog pojavljivanja iz statičke tabele; varijanta je adaptivni model gde se uporedo sa kodiranjem ažuriraju statistike simbola) ili modeliranje zasnovano na rečniku (jednim kodom se zamenjuje niz simbola iz rečnika; varijanta je adaptivni rečnik).

Elektronske, magnetne i optičke tehnologije

Elektronska tj. poluprovodnička tehnologija koristi integrisana kola bazirana na tranzistorima i kondenzatorima. Neke od vrsta ove tehnologije su: DRAM (mem. ćelija se implementira sa jednim tranzistorom i jednim kondenzatorom), SRAM (mem. ćelija se implementira pomoću šest CMOS tranzistora), Flash (mem. ćelija se implementira pomoću FGMOS tranzistora). Operativna memorija se najčešće realizuje kao DRAM, keš memorija kao SRAM, a USB stick-ovi koriste FLASH tehnologiju.

Magnetna tehnologija koristi magnetizaciju za čuvanje informacija. Podatku se pristupa pomeranjem glave za upis/čitanje duž medijuma. Nekad se ova tehnologija koristila i za op. memoriju, flopi diskove i magnetne trake, a danas se koristi uglavnom za hard diskove.

Optička tehnologija tipično podrazumeva optičke diskove koji čuvaju informaciju u vidu deformiteta na površini. Za pristup se koristi laserska dioda koja osvetljava površinu diska, a zatim se posmatraju refleksije. Primeri optičkih diskova su: CD, DVD, UDO.

Operativna memorija, organizacija, njene karakteristike i performanse

Operativna memorija (*main memory*) je nepostojana (*volatile* –gubi podatke kad prestane napajanje) memorija prvog nivoa čijem sadržaju se pristupa pomoću memorijske adrese. Sastoji se od ćelija grupisanih u memorijske lokacije. Svaka ćelija je povezana sa dve data linije. Adresne linije na ulazu dekodera služe za selektovanje jedne lokacije;

selektuju se sve ćelije te lokacije što omogućava upis ili čitanje njihovog sadržaja preko data linija. Različite organizacije memorija istog kapaciteta mogu uzrokovati različit broj neophodnih pinova, a broj pinova značajno utiče na cenu čipa. Povećanje broja bitova po memorijskoj lokaciji dovodi do povećanja broja pinova u integrisanom kolu. Uobičajena praksa koja rešava ovaj problem je da se adresne linije podele na linije koje adresiraju vrste i posebne linije koje adresiraju kolone. Dodatno, obe vrste adresnih linija se mogu transportovati preko istih pinova (*time-multiplexing*). Tada moraju postojati dve kontrolne linije \overline{RAS} (row address strobe) i \overline{CAS} (column address strobe) koje ukazuju kad su koje adresne linije validne.

Operativna memorija se realizuje pomoću DRAM čipova, što znači da se svaka ćelija sastoji iz jednog tranzistora i kondenzatora; da je tip pristupa *random access* (bilo kom podatku pristupa za približno isto vreme); da je vreme ciklusa veće od vremena pristupa (zbog neophodnog osvežavanja i upisa prilikom čitanja). Vreme pristupa je mnogo kraće (oko milion puta) nego kod hard diska realizovanog magnetnom tehnologijom. Još kraće vreme pristupa je kod keš memorija (realizovanih sa SRAM)- nije potrebno povremeno osvežavanje ćelija. Shodno performansama, cena operativne memorije je manja od cene keša a veća od cene hard diska.

Kašnjenje, vreme ciklusa, propusna moć i preklapanje pristupa modulima

Kašnjenje (*latency*) se definiše kao vremenski interval koji protekne od generisanja zahteva za informacijom do pristupa toj informaciji.

Vreme ciklusa (*cycle time*) se definiše kao vreme koje mora da protekne između dva uzastopna obraćanja memoriji. Kod nekih memorija je vreme ciklusa isto kao vreme pristupa, a kod nekih je vreme ciklusa veće.

Propusna moć (*bandwidth* ili *throughput*) izražava broj bita kojima se može pristupiti u jednoj sekundi. Tehnike za povećanje propusne moći glavne memorije su veća širina mem. reči ili preklapanje pristupa memorijskim modulima (*interleaving*). Ako se memorija ne sastoji iz jednog, već iz više mem. modula, procesor i U/I uređaji mogu istovremeno da pristupaju memoriji ako su adrese koje traže iz različitih modula. Pri tome magistrala mora da bude realizovana sa podeljenim ciklusima (gazda drži magistralu zauzetu samo onoliko vremena koliko treba za prenos potrebnih informacija da bi se realizovao pristup memoriji).

Keš memorije (preslikavanje adresa, tehnike zamene blokova i tehnike upisa)

Kada procesor zatraži neki element, prvo se proverava da li se on nalazi u keš memoriji. Ako se podatak ne nalazi u kešu, treba ga doneti iz operativne memorije u keš. Ali zbog fenomena prostornog lokaliteta, iz operativne memorije se dovlači čitav blok podataka, koji između ostalog sadrži traženi element. Dobro bi bilo da se blok prenese u jednom pristupu memoriji (tehnika preklapanja pristupa mem. modulima ovo omogućava ako se elementi bloka nalaze u različitim modulima tj. ako su uzastopne adrese smeštene u različitim modulima).

Pretpostavimo da je operativna memorija podeljena na blokove označene sukcesivnim brojevima (tag). Prilikom donošenja bloka u keš, sadržaj bloka se pamti u Data memoriji a oznaka bloka na odgovarajućoj lokaciji u Tag memoriji. Procesor upućuje zahtev za

dobijanje nekog podatka navođenjem adrese tog podatka. U MMU jedinici (*memory management unit*) se u zavisnosti od organizacije keša, adresa traženog podatka interpretira na različite načine i sprovode koraci odgovarajućeg protokola.

U slučaju keša sa direktnim preslikavanjem (blokovi se smeštaju na fiksnu lokaciju određenu njihovim tagom):

1. *Block* polje adrese direktno određuje lokaciju u Data memoriji gde treba da se nalazi blok sa traženim podatkom.
2. *Tag* polje adrese se poredi sa sadržajem odgovarajuće lokacije u Tag memoriji. Ako postoji poklapanje sadržaja, znači da se blok sa traženim elementom zaista nalazi u Data memoriji.
3. Elementu unutar bloka se pristupa koristeći *Word* polje adrese.

U slučaju keša sa asocijativnim preslikavanjem (blokovi se smeštaju na bilo koju lokaciju):

1. *Tag* polje adrese se paralelno poredi sa sadržajem svih lokacija u Tag memoriji (realizovana kao asocijativna memorija). Ako postoji poklapanje sa sadržajem neke lokacije, znači da se blok sa traženim elementom nalazi u Data memoriji na korespondirajućoj lokaciji.
2. Elementu unutar bloka se pristupa koristeći *Word* polje adrese.

Prednost keša sa asocijativnim preslikavanjem je dobro iskorišćenje prostora jer nema ograničenja gde može da se smesti dolazeći blok. Prednost keša sa direktnim preslikavanjem je što se ne troši vreme na pretragu tagova. Prednosti obe organizacije su iskorišćene kod keša sa set-asocijativnim preslikavanjem (keš je podeljen na setove; blokovi se smeštaju na bilo koju lokaciju u fiksno određenom setu):

1. *Set* polje adrese direktno određuje set-skup lokacija u Data memoriji gde treba da se nalazi blok sa traženim podatkom.
2. *Tag* polje adrese se paralelno poredi sa sadržajem lokacija u Tag memoriji koje korespondiraju datom setu. Ako postoji poklapanje sa sadržajem neke lokacije, znači da se blok sa traženim elementom nalazi u Data memoriji na odgovarajućoj lokaciji.
3. Elementu unutar bloka se pristupa koristeći *Word* polje adrese.

Poslednji korak kod svih organizacija je da se u slučaju promašaja, blok sa traženim elementom dovlači u keš i onda pristupa elementu.

Ako je keš memorija puna i treba doneti blok iz op. memorije, jedan od blokova iz keša se vraća u op. memoriju. Da bi se odredilo koji blok treba zameniti, primenjuje se jedna od sledećih tehnika zamene blokova u keš memoriji (ne koristi se kod direktnog preslikavanja jer je blok za zamenu trivijalno određen):

- tehnika slučajnog izbora (random selection), slučajno bira blok pomoću generatora slučajnih brojeva

- FIFO tehnika (first-in-first-out) zamenjuje blok koji je najduže boravio u kešu

- LRU tehnika (least recently used) zamenjuje blok kome je najmanje pristupano od kako se nalazi u kešu.

Tehnike upisa u kešu se mogu podeliti na tehnike upisa prilikom pogotka i tehnike upisa prilikom promašaja. Zamena blokova je potrebna samo kod tehnike *write-back* (i to samo blokova u koje je izvršen upis; dirty bit vodi evidenciju o tome). Alternativna tehnika upisa prilikom pogotka - *write-through* znači da se operacija upisa vrši

istovremeno u bloku u operativnoj memoriji i u njegovoj kopiji u kešu, pa nije potrebno naknadno ažuriranje sadržaja op. memorije.

Tehnike upisa prilikom promašaja su: *write-allocate* (blok se dovlači iz op. memorije i onda vrši upis u njegovu kopiju u kešu) i *write-no-allocate* (upis se vrši u blok u op. memoriji).

Sistemi virtuelne memorije

Virtuelna memorija je tehnika za proširenje očigledno ograničene fizičke memorije. Delovi programa koji se trenutno izvršavaju su učitani u operativnu memoriju, dok se ostali delovi nalaze na disku. Ako procesor zahteva podatak iz dela programa koji je na disku, onda se taj deo programa mora učitati u op. memoriju. Niz memorijskih reči koji se prenosi između diska i op. memorije se zove stranica (niz fiksne veličine) ili segment (niz promenljive veličine). Adrese koje generiše procesor su virtuelne adrese (iz virtuelnog adresnog prostora) koje treba prevesti u odgovarajuće realne adrese (adrese iz operativne memorije). Informacije potrebne za preslikavanje se nalaze u tabeli stranica i/ili tabeli segmenata.

Virtuelna memorija stranične organizacije ima virtuelni adresni prostor podeljen na stranice koje se po potrebi smeštaju u blokove (iste veličine kao stranice) op. memorije. Tabela stranica za svaku stranicu ima poseban ulaz koji predstavlja deskriptor te stranice. Pomoću *Page* dela virtuelne adrese, pristupa se odgovarajućem ulazu koji sadrži broj bloka op. memorije gde je smeštena ta stranica. Konkatenacijom broja bloka i *Word* dela virtuelne adrese se dobija fizička adresa traženog podatka.

Kod virtuelne memorije segmentne organizacije, virtuelni adresni prostor je podeljen na segmente različite veličine koji se po potrebi smeštaju u delove op. memorije odgovarajuće veličine. Tabela segmenata ima poseban ulaz za svaki segment (deskriptor). Pomoću *Segment* dela virtuelne adrese, pristupa se odgovarajućem ulazu tabele koji sadrži početnu adresu dela op. memorije gde je smešten segment. Sabiranjem početne adrese segmenta i *Word* dela virtuelne adrese, dobija fizička adresa traženog podatka.

Kod virtuelne memorije segmentno-stranične organizacije, virtuelni adresni prostor je podeljen na segmente različite veličine koji se sastoje iz stranica fiksne veličine. Svaki proces ima jednu tabelu segmenata i onoliko tabela stranica koliko ima segmenata. Pomoću *Segment* dela virtuelne adrese, pristupa se odgovarajućem ulazu tabele segmanata koji sadrži početnu adresu odgovarajuće tabele stranica. Ona se sabira sa *Page* delom virtuelne adrese, čime se pristupa odgovarajućem ulazu tabele stranica. Dalje, kao kod stranične organizacije se dobija fizička adresa traženog podatka.

Tabele preslikavanja se nalaze u op. memoriji što znači da je potreban dodatni pristup op. memoriji da bi se odredila adresa podatka. Rešenje je da se mali delovi tabela preslikavanja kojima je najskorije pristupano nalaze u posebnom kešu TLB jedinici (translation look-aside buffer). Najpre se pristupa TLB jedinici - ako se nađe traženi deskriptor, formira se adresa podatka i nije potreban dodatan pristup op. memoriji. U slučaju promašaja, deskriptor se traži u tabeli preslikavanja u op. memoriji. Opet su moguća dva slučaja: ako postoji deskriptor, generiše se adresa i TLB ažurira sa nađenim deskriptorom. U slučaju promašaja, treba učitati traženu stranicu (ili segment) sa diska u op. memoriju i ažurirati tabelu preslikavanja. Ovo je zadatak operativnog sistema, koji pri tome oduzima procesor datom procesu i dodeljuje ga nekom drugom radnosposobnom

procesu. Ako je op. memorija puna, dolazeća stranica (segment) treba da zameni neku stranicu (segment) iz op. memorije. Da bi se odredilo koju stranicu treba prepisati (ili vratiti na disk ako je izmenjena) koristi se neka od tehnika zamene: tehnika slučajnog izbora, FIFO ili LRU tehnika. TLB jedinica može da ima organizaciju sa direktnim, asocijativnim ili set-asocijativnim preslikavanjem. U zavisnosti od redosleda povezivanja TLB jedinice i keša, keš može da sadrži virtuelne ili realne adrese.

Memorijske tehnologije DRAM, EPROM i FLASH

DRAM je vrsta random access memorije (RAM- bilo kom podatku pristupa za približno isto vreme) koja ima jednostavniju strukturu od SRAM memorije. Obično ima formu kvadratne matrice ćelija koje služe za smeštanje bitova. Ćelija se sastoji iz jednog tranzistora i kondenzatora. Pošto kondenzator vremenom curi, da se ne bi izgubila informacija, neophodno je povremeno osvežavanje (refreshing). Zato memorija realizovana sa DRAM nije povremeno raspoloživa. Takođe, iz istog razloga, svako čitanje mora da bude ispraćeno upisom tog istog podatka. Ćelija SRAM memorije zauzima više prostora na čipu (četiri tranzistora čuvaju podatak a druga dva kontrolišu pristup ćeliji) ali je ne treba osvežavati. Sa porastom kapaciteta DRAM čipova, rastao je i broj potrebnih adresnih linija a time i broj pinova što uvećava cenu čipa. Rešenje je bilo multipleksiranje adresa, što je umanjilo broj pinova. Kod DRAM-a cilj je što veći kapacitet pa tek onda brzina, a kod SRAM-a je obrnuto. Zato se DynamicRAM koristi za realizaciju operativne memorije, a StaticRAM keš memorije.

EPROM (Erasable Programmable Read-Only Memory) je vrsta ROM memorije (sačuvani podaci se ne mogu promeniti- lako ili brzo) kod koje se brisanje podataka vrši izlaganjem snažnoj ultraljubičastoj svetlosti. Za ponovni upis podataka je potreban napon veći od standardnog. Izdržljivost većine EPROM čipova prelazi hiljadu ciklusa brisanja i reprogramiranja.

FLASH memorija je moderni tip EEPROM memorije (Electrically Erasable Programmable ROM – brisanje se vrši elektronski i selektivno, za razliku od EPROM gde se briše čitav sadržaj) koji brže briše i upisuje od standardnog EEPROM-a, a manje košta. Njena izdržljivost prelazi milion ciklusa. To je postojana memorija (sadržaj se ne gubi prestankom električnog napajanja) koja se primarno koristi kod prenosivih uređaja, memorijskih kartica i USB stick-ova.

Pouzdanost memorijskih sistema;detektovanje greške, sistemi za ispravljanje greške

Kodovi za detektovanje i ispravljanje grešaka se često koriste da bi se poboljšala pouzdanost memorijskih sistema. *Parity* bit je najjednostavniji mehanizam za detekciju grešaka koji može dektovati samo neparan broj grešaka. Bit parnosti se dodaje da bi se osiguralo da broj bitova sa vrednošću jedan u datom skupu bude uvek paran ili neparan. Mnogi keševi za instrukcije sadrže *parity* zaštitu (u slučaju greške se stari sadržaj odbacuje i učitava ispravan iz operativne memorije).

Elektronske ili magnetne smetnje unutar računarskog sistema mogu prouzrokovati da neki bit spontano pređe u suprotno stanje. Greške u DRAM čipovima se često dešavaju kao posledica kosmičkog zračenja. Zato je kod servera i računara sa ozbiljnom namenom ova memorija strukturirana kao ECC memorija (*error-correcting memory*). Sistem je

tako dizajniran da susedni bitovi pripadaju različitim rečima. Zato jedan događaj uzrokuje samo jednu grešku u nekoj reči koja se može ispraviti kodom za korekciju greške u jednom bitu. Memorijski kontroleri najčešće koriste *Hamming* kod koji omogućava korekciju greške u jednom bitu i detekciju grešaka kod dva bita neke reči.

Hard diskovi koriste CRC (*cyclic redundancy check*) kod za detektovanje i *Reed-Solomon* kod za ispravljanje grešaka i premeštanje podataka sa pokvarenih sektora na slobodne sektore.

2.4 Interfejs i komunikacija

U računarskom sistemu različiti posistemi moraju da imaju interfejs ka drugom podsistemu (memorija i procesor trebaju da komuniciraju, takođe procesor i I/O uređaji). To se postiže magistralom – deljenim komunikacionim linkom sastavljenim iz skupa žica koje povezuju više podsistema. Njen osnovni nedostatak je što širina magistrale limitira maksimalni I/O protok. Projektovanje *bus* sistema koji može da odgovori na zahteve procesora i konektovanja velikog broja I/O uređaja predstavlja značajni izazov.

Periferni uređaji se ne konektuju direktno na sistemsku magistralu zbog sporijeg (ili bržeg) transfera podataka u odnosu na procesor ili memoriju. Često koriste drugačije formate i dužine reči u odnosu na dati računar. Takođe zbog postojanja raznih tipova perifera, bilo bi neopraktično ugraditi neopraktičnu logiku za njihovu kontrolu u procesor. Zato su neopraktični I/O moduli (kontroleri) čije su dve osnovne funkcije interfejs ka procesoru i memoriji i interfejs ka I/O uređaju.

Osnove I/O: *handshaking*, *buffering*

I/O (ulazno/izlazni) uređaji se razlikuju po tome kako smeštaju informacije i po brzini kojom šalju ili primaju podatke. Tako npr. tastatura šalje oko 10 karaktera (bita) u sekundi, a skener 200,000. Pošto postoji velika razlika u brzini kojom procesor i I/O uređaji procesiraju informacije, bila bi šteta da procesor čeka na podatke sa sporih uređaja. Zato postoje *data* registri u koje ulazna periferija smešta podatak i signalizira dostupnost podatka procesoru. Kada procesor preuzme podatak, periferija može da prenese sledeći podatak u registar. Slično, kod izlaza, procesor postavlja podatak u *data* registar. Kada izlazna periferija pokupi taj podatak, signaliziraće procesoru da može da nastavi sa prenosom sledećeg podatka. Ovaj jednostavan način komunikacije se zove I/O protokol.

I/O tehnike: programirani I/O, I/O generisanjem prekida, DMA

Kontroler perifera i procesor rade asinhrono pa mora da postoji mehanizam njihove sinhronizacije. Mora se osigurati da podatak koji procesor pošalje u *data* registar ne bude prepisan sledećim poslatim podatkom (zbog velike brzine procesora) ili da procesor ne pročita dva puta isti podatak u slučaju ulaza. Za to služi *ready* bit statusnog registra kontrolera perifera. Aktivna vrednost *ready* bita ukazuje da je *data* registar raspoloživ, a neaktivna da nije. Kod ulaza, kada se podatak sa ulazne perifera smesti u *data* registar, *ready* bit se setuje. Procesor koji povremeno čita statusni registar i proverava bit *ready*, će pročitati podatak iz *data* registra i resetovati *ready* bit. Slično, izlazna periferija uzima

podatak iz *data* registra tek kad procesor setuje *ready* bit, i posle preuzimanja resetuje taj bit. Kod programiranog ulaza/izlaza, prenos podataka između operativne memorije i kontrolera periferije se realizuje programskim putem, operacijama koje izvršava procesor. Ova tehnika je jednostavna ali spora, korisna je kad treba da se podaci pojedinačno prenose.

Kod I/O generisanjem prekida, procesor ne mora da gubi vreme na proveravanje *ready* bita, već dobija signal prekida od kontrolera periferije. Indikacija o tome da je *data* registar postao raspoloživ, se šalje procesoru generisanjem signala prekida intr. Procesor reaguje tako što prekida tekuću obradu, skače na odgovarajuću prekidnu rutinu u kojoj se obavlja transfer podatka između kontrolera periferije i op. memorije, a zatim se vraća na prekinuto izvršavanje. Izvršavanje se prekida onoliko puta koliko podataka treba preneti. Ako treba preneti veliki blok podataka, onda će procesor trošiti više vremena na ulazak i izlazak iz prekidne rutine nego na izvršavanje programa. Tada je pogodnija DMA tehnika.

Kontroler periferije sa direktnim pristupom memoriji (DMA) obavlja transfer velike količine podataka između periferije i memorije bez učešća procesora koji za to vreme može da izvršava neke druge operacije. Programskim putem se vrši inicijalizacija i startovanje kontrolera, dok sam kontroler vrši prenos podataka, vodi evidenciju o adresama podataka i broju podataka koje treba preneti pomoću adresnog registra i registra veličine bloka podataka. Da bi izvršio prenos podataka, kontroler mora da traži dozvolu da koristi magistralu (posle obavljenog prenosa se odriče tog prava). *Burst* bitom upravljačkog registra kontrolera se specifikira način prenosa: *burst* mod kada kontroler drži magistralu sve dok ne završi prenos svih podataka ili *single-cycle* mod kada kontroler predaje kontrolu nad magistralom posle prenosa jednog podatka. Signalom prekida kontroler obaveštava procesor da je završio prenos.

Prekidi: vektorisani, prioritet prekida, utrošak pri prekidu

Postoji mnogo načina za određivanje adrese prekidne rutine ali je najčešći putem vektor tabele IVT (*interrupt vector table*). To je oblast u operativnoj memoriji koja sadrži adrese prekidnih rutina. I/O uređaj može da signalizira procesoru spremnost za transfer podatka slanjem signala prekida intr. Kada procesor odgovori signalom potvrde *inta*, uređaj šalje broj ulaza u IVT tabelu procesoru (poseban registar kontrolera periferije sadrži broj ulaza dodeljen toj periferiji). Na osnovu broja ulaza i početne adrese tabele koji se nalazi u IVTP registru procesora, dolazi se do adrese prekidne rutine koja treba da se izvršava. Ulasku u prekidnu rutinu prethodi čuvanje konteksta procesora na steku i maskiranje prekida. Upisom adrese prekidne rutine u programski brojač počinje izvršavanje prekida. Na početku se mogu dozvoliti maskirajući prekidi pa je moguće gnežđenje prekida (zavisi od prioriteta prekida). Na kraju prekidne rutine se kontekst procesora restaurira sa steka. Kod I/O generisanjem prekida, procesor može da troši više vremena na ulazak i izlazak iz prekidne rutine nego na izvršavanje programa. Zato je za prenos većeg bloka podataka bolje koristiti neku drugu tehniku.

Postoje dva tipa organizacije *interrupt* linija između procesora i I/O uređaja. U jednoj organizaciji su I/O uređaji povezani u lanac kroz koji ide jedna linije potvrde (*grant line*, *inta*) od procesora kroz sve uređaje. Uređaj koji je najbliži procesoru (prvi uređaj u nizu) prvi proverava signal potvrde-ako je on bio generisao signal prekida, zadržaće za sebe

signal potvrde i započne komunikaciju s procesorom; inače će propustiti signal potvrde do sledećeg uređaja u nizu koji će opet na isti način postupiti. Na taj način su I/O uređajima dodeljeni prioriteti zavisno od fizičkog položaja u odnosu na procesor. Svi uređaji signal prekida šalju preko zajedničke *intr* linije. U drugoj organizaciji, svaki I/O uređaj ima svoju liniju po kojoj šalje signal prekida i svoju liniju po kojoj dobija signal potvrde. Sada prioritet uređaja ne zavisi od lokacije. Jedinica za opsluživanje prekida u procesoru (ISU) je zadužena za rešavanje problema prioriteta u slučaju više istovremenih zahteva za prekid.

Dizajn memorijskog sistema i interfejs

Razlikuju se dva mehanizma na osnovu kojih procesor adresira registre kontrolera periferije. Jedan mehanizam je memorijski preslikan I/O prostor što znači da se registrima pristupa kao da su memorijske lokacije. Nema potrebe za specijalnim I/O instrukcijama, već se registri kontrolera mogu adresirati u svim naredbama u kojima se adresiraju memorijske lokacije (npr. LOAD, STORE, MOV). Nedostatak je što se mora rezervisati deo memorijskog adresnog prostora za adresiranje I/O uređaja. Drugi mehanizam je razdvojen I/O prostor – registrima se dodeljuju posebne adrese izvan memorijskog adresnog prostora. Za pristup se koriste samo specijalne instrukcije (IN, OUT). Operativna memorija i I/O uređaji dele adresne linije i linije podataka iz procesora. Aktivna (neaktivna) vrednost signala M/IO označava da se na ABUS linijama nalazi adresa iz memorijskog (ulazno-izlaznog) adresnog prostora. Kombinaциона mreža za prepoznavanje ciklusa u kontroleru stalno proverava vrednost M/IO signala i sadržaj na adresnim linijama.

Magistrale: protokoli i arbitracije

Magistrala je uređena grupa linija koje služe za prenos sadržaja između modula računarskog sistema. Tok prenosa sadržaja se zove ciklus na magistrali. Modul koji započinje ciklus je gazda (*master*), a modul s kojim gazda realizuje ciklus je sluga (*slave*). U zavisnosti od toga koliko je magistrala zauzeta prilikom ciklusa, razlikuju se magistrale sa atomskim i magistrale sa podeljenim ciklusima.

Kod magistrala sa atomskim ciklusima, magistrala je zauzeta sve vreme dok se ne realizuje prenos podataka. Postoje ciklusi:

- ciklus čitanja (gazda postavlja adresu lokacije na adresne linije magistrale ABUS i signal čitanja na upravljačku liniju magistrale RDBUS; na signal čitanja reaguje samo modul koji sadrži datu lokaciju tako što čita podatak i postavlja taj sadržaj na linije podataka magistrale DBUS),
- ciklus upisa (gazda postavlja adresu lokacije na ABUS, podatak na DBUS i signal upisa na upravljačku liniju magistrale WRBUS.; na signal upisa reaguje samo modul koji sadrži datu lokaciju tako što upisuje podatak sa linija DBUS na tu lokaciju) i
- ciklus prihvatanja broja ulaza (procesor postavlja signal potvrde prekida *inta* na posebnu liniju koja ne pripada magistrali; odgovarajući ulazno/izlazni uređaj reaguje na taj signal tako što postavlja broj ulaza na DBUS linije sa kojih procesor onda očitava traženi podatak.

Kod magistrala sa podeljenim ciklusima, magistrala je zauzeta samo onoliko vremena koliko treba za prenos informacija neophodnih za realizaciju čitanja ili upisa. Dok traje čitanje/upis u nekom modulu, magistrala je slobodna pa je moguće ostvariti da više modula paralelno realizuje čitanje/upis. Postoje ciklusi:

- slanje zahteva za čitanje (gazda postavlja adresu na ABUS, svoj identifikator na DBUS (koji sluga koristi prilikom ciklusa vraćanja podatka) i signal čitanja RDBUS. Ako je sluga bio zauzet (tj. ciklus nije uspešno realizovan), on postavlja signal ACKBUS na neaktivnu vrednost. A ukoliko je slobodan, sluga upisuje sadržaj sa ABUS u svoj adresni registar, sadržaj sa DBUS u svoj registar identifikatora i postavlja ACKBUS na aktivnu vrednost.),

- slanje zahteva za upis (gazda postavlja adresu na ABUS, podatak na DBUS i signal WRBUS. Ukoliko je slobodan, odgovarajući modul reaguje tako što upisuje sadržaj sa ABUS u svoj adresni registar, sadržaj sa DBUS u svoj registar podataka i postavlja ACKBUS na aktivnu vrednost. Inače, postavlja ACKBUS na neaktivnu vrednost.),

- slanje zahteva za dobijanje broja ulaza (gazda šalje svoj identifikator na DBUS i postavlja signal *inta*. Ako je slobodan, sluga upisuje sadržaj sa DBUS u svoj registar identifikatora i postavlja ACKBUS na aktivnu vrednost. Inače, postavlja ACKBUS na neaktivnu vrednost.) i

- ciklus vraćanja podatka (gazda postavlja identifikator na ABUS, traženi podatak na DBUS i postavlja signal na upravljačku liniju DABUS. Modul koji prepozna sadržaj na ABUS postaje sluga. Ako je slobodan, sluga upisuje sadržaj sa DBUS u svoj registar podataka i postavlja ACKBUS na aktivnu vrednost. Inače, postavlja ACKBUS na neaktivnu vrednost.).

Magistrale mogu biti sinhronne ili asinhronne. Kod asinhronih magistrala, postavljanjem upravljačke linije završetka ciklusa (FCBUS) sluga signalizira gazdi da je traženi podatak raspoloživ ili da mu adresa i podatak sa magistrale više nisu potrebni. Ovo je potrebno kada moduli rade asinhrono-svaki na svoj signal takta pa trajanje ciklusa na magistrali nije fiksno. Kod sinhronih magistrala, moduli rade sinhrono-na isti signal takta (MCLK). Pošto je trajanje ciklusa fiksno, gazda po isteku određenog vremena pretpostavlja da je traženi podatak raspoloživ na DBUS ili da je sluga upisao podatak sa DBUS linija. Nema potrebe za postojanjem signala završetka ciklusa FCBUS.

Arbitracija je potrebna za rešavanje konflikta kada više modula istovremeno želi da realizuje ciklus na magistrali. Kod centralizovane arbitracije, poseban uređaj-arbitrator određuje ko može sledeći (od datih kandidata) da postane gazda. Moduli računara mogu biti povezani sa arbitratorom preko jedne zajedničke linije za slanje zahteva i jedne linije za primanje dozvole (*request* i *grant*) ili da svaki modul ima svoju *request* i *grant* liniju preko koje je povezan s arbitratorom. U prvom slučaju prioritet modula zavisi od njegovog položaja u nizu, dok u drugom slučaju prioriteti nisu fiksni već zavise od toga koji algoritam primenjuje arbitrator. Decentralizovana arbitracija znači da svaki modul ima jedinstveni arbitracioni broj koji služi za rešavanje konflikta. Modul koji zahteva dozvolu za korišćenje magistrale šalje ovaj broj ostalim modulima koji ga porede sa svojim brojem. Samo uređaj sa najvećim brojem ostaje i dobija dozvolu. Arbitracija može da bude sa praćenjem zahteva ili sa pamćenjem zahteva. Kod praćenja zahteva, svi moduli uvek učestvuju u arbitraciji što za posledicu ima da moduli nižeg prioriteta mogu dugo da čekaju. U slučaju sistema sa pamćenjem zahteva, u arbitraciji učestvuju samo

zahtevi upamćeni do tog trenutka u listi. Kad se svi oni opsluže, ponovo se pamte zahtevi u listu.

2.5 Device podsistemi

I/O (periferni) uređaji služe za interakciju računarskog sistema sa spoljašnjim svetom (npr. tastature, štampači) ili za smeštanje podataka (npr. disk jedinice). Iako su performanse procesora i memorije važni faktori u performansama sistema, optimalne performanse diska su presudne za propusnu moć sistema. Veliki broj interakcija korisnika sa računarom uključuje upravo neku vrstu čitanja ili upisivanja na disk. Pristup disku se dešava i prilikom svake *page fault* situacije prilikom korišćenja virtuelne memorije. U slučaju I/O sistema koji loše funkcioniše, usporava se izvršavanje procesa i nagomilavaju se neobavljeni poslovi u procesoru.

Organizacija i struktura hard diska i optičke memorije

Magnetni disk se sastoji iz niza metalnih diskova – ploča (*platters*) koje se okreću na osovini. Obe strane ploče su prekrivene magnetnim materijalom pa se informacije mogu smeštati na obe površine. Površina diska je podeljena na koncentrične krugove - trake (*tracks*) koje se onda dele na sektore – najmanje jedinice za čitanje ili upisivanje podataka, susedni sektori su sekvencijalno raspoređeni. Sve trake ili imaju isti broj sektora (manja gustina bitova na spoljnim, dužim stazama) ili je broj sektora veći na dužim trakama (*constant bit density*). Cilindar čine sve trake istog poluprečnika. Iznad svake površine se nalazi pokretna ruka sa glavom za čitanje/upis podataka. Disk kontroler izdaje komande koje upravljaju rukom tako što je pomeraju na odgovarajuću traku (*seek time*). Rotacijom diska se traženi sektor postavlja ispod magnetne glave za čitanje/upis (*rotation latency*). Sledeća komponenta pristupa disku je vreme potrebno za transfer bloka bitova (tipično veličine sektora) sa glave na disk – *transfer time*. Pre slanja na disk, podaci se prevode i kompresuju u format date disketne jedinice. Disk kontroler određuje gde ima slobodnog prostora na disku pregledanjem tabele popunjenosti diska (obično postoje dve kopije da bi se obezbedio integritet podataka). Posle pozicioniranja, kompresovane informacije se šalju u glavu za upis koja menja magnetna svojstva diska i tako smešta podatke. Napredak u metodama za kompresiju podataka je omogućio da više informacija stane u jedan sektor; veličina sektora se mogla smanjiti što znači i da se ostavlja manje neiskorišćenog prostora. Hard diskovi rotiraju tehnikom konstantne ugaone brzine (glava pokriva veće udaljenosti u jedinici vremena kod spoljnih traka) za razliku od CD diskova (*constant linear velocity*; glava pokriva iste distance u jedinici vremena kod svih traka; disk sporije rotira kad se ruka pomeri ka spoljnim trakama).

Optički disk je kružni disk koji kodira podatke u mikroskopskim neravninama na specijalnom materijalu, često aluminijumu. Podaci se smeštaju pomoću lasera na jednu od površina diska, a pristupa im se osvetljavanjem diska laserskom diodom u optičkom uređaju koji vrti disk promenljivom ugaonom brzinom. Postoje tri tipa dizajna: *read-only* (npr. CD i CD-ROM), *recordable* (npr. CD-R) i *re-recordable* (npr. CD-RW). Prva generacija optičkih uređaja je koristila infracrvenu lasersku glavu za čitanje. Kako je talasna dužina ograničavajući faktor za veću gustinu informacija, najveći kapacitet koji se

postizao infracrvenim laserom je bio 700MB. Druga generacija je koristila laser manje talasne dužine sa vidljivom svetlošću (obično crvena). To je omogućilo kapacitet od 4.7 GB u DVD formatu. Medijum koji je zamenio standardni DVD format je Blu-ray Disc sa kapacitetom do 50GB. Ime potiče od plavo-ljubičastog lasera kojim se čita disk; jedna od glavnih namena je za distribuiranje videa visoke definicije. Za potrebe prenosa i arhiviranja podataka - *backup*, optičke diskove postepeno zamenjuju manje, brže i pouzdanije *USB flash* jedinice.

Osnovni I/O kontroleri kao što je tastatura i miš

Tastatura je ulazni uređaj sa nizom dugmadi koji deluju kao elektronski prekidači. Kod *dome-switch* tastatura, pritiskom tastera pritišće se gumena kupola *dome* s čije donje strane postoji provodnički kontakt. On onda dotiče par provodničkih linija i premošćava procep između njih. Signal iz tog para linija postaje drugačiji što registruje čip i generiše *make* kod za odgovarajući pritisnut taster. Kad se taster otpusti, generiše se *break* kod. Starije tastature sa *buckling spring* mehanizmom su proizvodile zvuk prilikom pritiska tastera kao povratnu informaciju korisniku. Kada bi računar pratio svaki puls, registrovao bi više udara tastera umesto samo jednog jer taster odskače nekoliko puta pre nego se smiri nakon pritiska ili otpuštanja. Zadatak kontrolnog procesora je da agregira ove pokrete u jedan. Kontroler tastature je komponenta računara koja prima *make* i *break* kodove i šalje procesoru signal hardverskog prekida kad god se pritisne ili otpusti taster. Prekidna rutina obično smešta kodove u red da bi se obradili kasnije a procesor nastavlja s onim što je pre izvršavao. Dok većina tastera proizvodi slova, brojeve i znake u tekstualnom editoru ili nekom drugom programu, simultani pritisak više tastera zadaje akcije ili komande operativnom sistemu. Interpretacija tastera se može promeniti softverom (remapiranje dugmadi) i računar se obaveštava o novom značenju pritisnutog tastera. Bežične tastature zahtevaju kombinaciju jedinica predajnika i prijemnika; funkcionišu pomoću radio frekvencije ili infracrvenih signala. Ako koriste industrijski standard radio frekvencije *Bluetooth*, primopredajnik može biti ugrađen u računar.

Miš je pokazivački uređaj koji detektuje njegovo dvodimenzionalno relativno kretanje u odnosu na podupiruću površinu i prevodi ga u kretanje kursora na ekranu. Za razliku od mehaničkih miševa, optički miševi nemaju pokretnih delova već osvetljavaju podlogu po kojoj se kreću LED ili laserskom diodom. Senzor snima sukcesivne slike podloge a specijalni ugrađeni čip za procesiranje slika prevodi promene između frejmova u dvodimenzionalno kretanje.

RAID arhitekture

RAID je inicijalno označavao *Redundant Array of Inexpensive Disks* tehnologiju za postizanje visokog nivoa pouzdanosti skladištenja pomoću organizovanja jeftinih manje pouzdanih diskova u jedan niz. Danas RAID (*Redundant Array of Independent Disks*) označava razne dizajne (RAID 0, RAID 1, RAID 5...) čiji je glavni cilj povećanje pouzdanosti podataka ili poboljšanje input/output performansi. Njihova zajednička karakteristika je kombinovanje više fizičkih diskova u jednu logičku jedinicu (RAID niz) koristeći specijalni hardver ili softver. U slučaju hardverskog rešenja, operativni sistem nije svestan da postoji više diskova, dok je drugo rešenje implementirano u samom

operativnom sistemu. Tri osnovna koncepta u RAID-u su: *mirroring* (kopiranje istih podataka na više diskova; redundantnost smanjuje kapacitet ali se podaci ne gube posle kvara nekog od diskova), *striping* (distribuiranje podataka duž niza diskova omogućava brži pristup podacima) i *error correction* (upis redundantnih podataka omogućava detektovanje kvara i rešavanje problema – sistem nastavlja s radom dok se pokvareni disk zamenjuje novim i ponovo se formira niz; zapisani dodatni *parity* podaci zajedno sa preostalim podacima služe za rekonstrukciju izgubljenih podataka).

Video kontrola

Video kartica (*graphics card*, *video card*) je ekspanziona kartica sa funkcionalnošću generisanja i iznošenja slika na ekran. Neke kartice imaju dodatne mogućnosti – konektovanje više monitora, TV izlaz, TV tjuner adapter itd. Kod ranijih računara je video hardver bio integrisan na matičnoj ploči i zvao se video ili grafički kontroler. VGA (*Video Graphics Array*) je prva široko prihvaćena kartica čije je poboljšanje rezolucije i broja korišćenih boja dovelo do SVGA (Super VGA) standarda sa 2MB video memorije i 1024x768 rezolucije. *Voodoo* grafički čipovi su uveli 3D efekte (*Z-buffering*, *anti-aliasing*). Familija *GeForce* kartica je bila fokusirana na poboljšanja 3D algoritama, a uvođenjem DDR tehnologije je kapacitet video memorije povećan na 128MB kod *GeForce 4* modela. Od 2002. godine tržištom video kartica dominiraju ATI *Radeon* i *Nvidia GeForce* linije.

Video kartica se sastoji od štampane ploče na kojoj se nalaze sledeće komponente: GPU jedinica (*Graphics processing unit*, optimizovana za izvršavanje *floating-point* operacija koje su fundamentalne za 3D grafiku, glavni atributi su frekvencija takta i broj *pipeline* struktura koje prevode 3D sliku u 2D sliku), video BIOS (firmver koji sadrži program za upravljanje operacijama video kartice i koji obezbeđuje skup instrukcija za pristup kartice video hardveru) i video memorija (obično brza memorija sa više portova kao npr. VRAM, WRAM, SGRAM i od 2003. godine tipično DDR tehnologije). RAMDAC (*Random Access Memory Digital-to-Analog Converter*) je komponenta koja konvertuje digitalne signale u analogne da bi ih mogli koristiti CRT ekrani sa analognim ulazima. Današnji LCD i plazma ekrani rade u digitalnom domenu i ne zahtevaju upotrebu RAMDAC.

I/O performanse

Tradicionalne mere performansi, vreme odziva (*response time*) i propusna moć (*throughput*), se takođe primenjuju na I/O sistem. Vreme koje protekne od momenta kada korisnik ukuca komandu do trenutka kada se pojavi rezultat je mnogo bolja mera performansi od CPU vremena. Prema Amdalovom zakonu ako je razlika između CPU vremena i vremena odziva 10%, ubrzanje procesora 100 puta će poboljšati I/O samo 10 puta čime se gubi 90% potencijala. Interakcija korisnika sa računarom (*transaction time*) se može podeliti na tri dela: *entry time* – vreme potrebno za unos komande od strane korisnika, *system response time* – vreme koje protekne od zadavanja komande do prikazivanja rezultata i *think time* – vreme od prijema rezultata do početka unosa nove komande. Rezultati više istraživanja su pokazali da smanjenje vremena odziva redukuje *transaction time* više od samog smanjenja vremena odziva. Ovaj naizgled nemoguć

rezultat se objašnjava ljudskom prirodom – čoveku je potrebno manje vremena za razmišljanje ako dobije brži odgovor.

IOPS (*Input/Output Operations Per Second*) je uobičajeni benčmark za hard diskove i druge računarske medijume za čuvanje podataka. Rezultati veoma zavise od promenljivih koje tester unosi u program - balans operacija čitanja i upisa, obrasci direktnog ili sekvencijalnog pristupa, broj niti, veličina reda, veličina bloka podataka. Najčešće karakteristike performansi koje se mere su *total* IOPS (ukupni broj I/O operacija u sekundi), *random read* IOPS (prosečan broj direktnih čitanja u sekundi), *random write* IOPS (prosečan broj direktnih upisa u sekundi), *sequential read* IOPS (prosečan broj sekvencijalnih čitanja u sekundi) i *sequential write* IOPS (prosečan broj sekvencijalnih upisa u sekundi). *Random* IOPS brojevi najviše zavise od *seek* vremena, dok sekvencijalni IOPS brojevi tipično ukazuju na maksimalni protok za dati uređaj. Tradicionalni hard diskovi imaju otprilike iste IOPS vrednosti za čitanje i za upis, dok su USB *flash* jedinice mnogo sporije prilikom upisivanja nego prilikom čitanja.

SMART tehnologija i otkrivanje kvarova

Self-Monitoring, Analysis and Reporting Technology SMART je kontrolni sistem koji izveštava o raznim indikatorima pouzdanosti hard diskova. Služi da upozori korisnika na predstojeći kvar kako bi se preduzele preventivne akcije, kao što je kopiranje podataka na zamenski uređaj. Kvarovi hard diska se svrstavaju u dve kategorije: nepredvidljivi kvarovi koji se dešavaju iznenada bez upozorenja (npr. oštećenja elektronskih komponenti, iznenadni mehanički kvarovi), i predvidljivi koji se pojavljuju postepeno pa se mogu detektovati na vreme da se preduzmu preventivne akcije (npr. mehaničko habanje). Indikatori predstojećeg kvara mogu biti povećan nivo šuma, problemi prilikom čitanja ili upisivanja podataka, povećana proizvodnja toplote ili oštećeni sektori diska. Osnovna informacija koju SMART pruža je SMART status. Moguće su dve vrednosti – prag nije pređen (*threshold not exceeded, drive OK*) i prag pređen (*threshold exceeded, drive fail*, ukazuje na relativno veliku verovatnoću da disk ubuduće neće zadovoljavati svoju specifikaciju). Predstojeći predviđen kvar može da bude katastrofalan, ili da predstavlja nesposobnost upisa na određene sektore, ili da samo ukazuje na lošije performanse od deklarisanog minimuma. Više detalja o stanju diska se dobija uvidom u SMART atribut čije značenje i interpretacija zavisi od tipa proizvođača. Proizvođači diskova postavljaju granične vrednosti atributa koje se ne prelaze u normalnim uslovima. SMART nije uvek ispravno implementiran zbog nedostatka standarda za sve aspekte ove tehnologije.

Interfejs procesora ka mreži

Network interface controller (NIC, ili *network interface card*) je hardverska komponenta koja omogućava računarima da komuniciraju u mreži. Ima više vrsta NIC kartica: za *Ethernet* mrežu, *Token Ring*, *FDDI*, a razlikuju se i po brzini (maksimalne brzine transfera su tipično 10, 100 ili 1000 megabita u sekundi). Nekad su to bile ekspanzione kartice koje su se priključivale na magistralu, a danas zbog sveprisutnosti *Ethernet* mrežne tehnologije i niske cene njenog hardvera, mnogi proizvođači ugrađuju funkcionalnost *Ethernet* kartica direktno na matičnu ploču PC računara. U tom slučaju

posebna mrežna kartica više nije potrebna. Svaki *Ethernet* NIC ima jedinstven 48-bitni serijski broj upisan u ROM čip – MAC adresu za identifikovanje uređaja u mreži. Prvo 24-bitno polje MAC adrese OUI (*Organizationally Unique Identifier*) specificira proizvođača kartice. Mrežna kartica tipično ima priključak za kabl upredene parice i par LED dioda koje informišu korisnika o tome da li je računar konektovan na mrežu i da li se podaci trenutno prenose. Mrežni interfejs ne mora uvek da ima fizičku formu, npr. *loopback* interfejs je deo softvera koji simulira mrežni interfejs za potrebe testiranja.

2.6 Dizajn procesorskog sistema

Većina internih komponenata sistema (procesor, keš, memorija, ekspanzione kartice i hard disk) međusobno komunicira pomoću magistrala. Ona predstavlja kanal za protok informacija između dva ili više uređaja, obično ima pristupne tačke odnosno mesta na koje se uređaji mogu priključiti. *Chipset* koristi procesorsku magistralu za prenos podataka iz/u procesor. Keš magistrala (*backside bus*) je posvećena magistrala za pristup kešu drugog nivoa. Memorijska magistrala povezuje memorijski podsistem sa procesorskom magistralom i *chipset*-om. I/O magistrale (*expansion bus*) spajaju periferne uređaje sa ostalim komponentama računara. Sistemski *chipset* kontroliše i obezbeđuje ispravnu komunikaciju svih delova sistema.

CPU interfejs: signal takta, upravljačke, adresne i magistrale podataka

Rad većine procesora je usklađen sa sinhronizacionim signalom takta. Njegova perioda se određuje prema maksimalnom vremenu potrebnom za prenos električnih signala u kolima procesora. Povećanjem frekvencije signala takta, postalo je sve teže izbeći njegovo kašnjenje kroz čitavu jedinicu. Problem predstavlja i veće rasipanje toplote što zahteva efikasnija rešenja za hlađenje procesora. Dizajn nekih procesora podrazumeva da određeni delovi kao što je ALU jedinica rade asinhrono. Manje uobičajeno rešenje je asinhroni procesor koji ne koristi globalni signal takta – pogodno za *embedded* računare zbog manje potrošnje energije i rasipanja toplote. Moderni PC računari tipično imaju četiri ili pet signala takta sa različitim frekvencijama (za delove sistema sa različitim brzinama rada).

Upravljačka magistrala prenosi komande procesora (signali upisivanja ili čitanja) i statusne signale uređaja dok magistrala podataka nosi podatke koje treba procesirati. Adresna magistrala služi da specificira fizičku adresu kada procesor ili DMA kontroler želi da očita ili upiše na neku memorijsku lokaciju. Širina adresne magistrale određuje veličinu memorije koju sistem može adresirati. Rani procesori su koristili jednu žicu za svaki bit adrese ali ovakav pristup je postao neprikladan kako se povećavala veličina memorije. Zato se u modernim procesorima adresa šalje iz dva dela (npr. za 32-bitnu adresu adresna magistrala ima 16 žica).

Dekodiranje adresa

Adresni dekoderi su osnovni gradivni blokovi za sisteme koji koriste magistralu. To su kola koja na ulazu imaju dva ili više bitova adresne magistrale, a na izlazu linije za

selekciju uređaja. Jedna varijanta je da postoji odvojeni dekodier u svakom uređaju, a druga je jedan dekodier sa N ulaznih bitova kojeg mogu koristiti 2^N uređaja.

Osnovni tipovi dekodiranja adresa su: iscrpno – 1:1 mapiranje adresa u memorijske lokacije; parcijalno – n:1 mapiranje n adresa u jednu memorijsku lokaciju (kada nije implementiran čitav adresni prostor) što omogućava programeru da referencira mem. lokaciju koristeći n različitih adresa; linearno – adresne linije se direktno koriste.

Osnovni paralelni i serijski interfejsi

Linkovi koji služe za komunikaciju između računara ili delova jednog računara mogu biti serijski (prenose jedan tok podataka, bit po bit) ili paralelni (prenose više tokova podataka – reči duž više kanala). Na prvi pogled bi izgledalo da je serijski link lošiji od paralelnog jer prenosi manje podataka u jednom trenutku. Međutim danas se serijskim vezama često postiže veća frekvencija slanja jer nema elektromagnetnih smetnji, efekta preslušavanja *crosstalk* (manji broj provodnika zauzima manje prostora pa su bolje izolovani) ili *clock skew*-a između kanala. Da bi se smanjio broj pinova a time i cena integrisanih kola, često se koriste serijske magistrale koje su jeftinije za implementaciju od paralelnih. Neke od serijskih magistrala su 1-Wire, I^2C (*Inter-Integrated circuit*), Serial ATA, USB (*Universal serial bus*, sa brzinom prenosa podataka 12 Mbps) i FireWire koji podržava velike brzine od 400 Mbps. Serijski interfejs PCI Express je zamenio paralelni AGP (*Accelerated graphics port*) kada performanse te magistrale više nisu mogle zadovoljiti potrebe sve bržih grafičkih kartica. Primeri paralelnih veza su: ISA (*Industry standard architecture*), konvencionalni PCI (*Peripheral component interconnect*), ATA (*Advanced technology attachment*, IDE, ATAPI, za povezivanje hard disk i CD-ROM uređaja). Pre pojave USB-a, paralelni interfejs se koristio za mnoge periferne uređaje – štampače, skenere, web kamere, eksterne modeme, hard disk i CD-ROM uređaje. Danas ga mnogi proizvođači računara smatraju zastarelim i zamenjuju USB ili Ethernet portovima.

Timers

Računarski sistemi obično imaju bar jedan timer. To su digitalni brojači koji ili inkrementiraju i nikad ne dostižu svoj limit ili dekrementiraju i prekidaju procesor kada dođu do nule. Rade sa fiksnom frekvencijom koja se može konfigurisati. Složeni tajmeri imaju logiku za poređenje njihove vrednosti sa nekom specifičnom vrednošću, čije poklapanje služi kao okidač za neku akciju. Operativni sistemi često koriste jedan hardverski tajmer da implementiraju više softverskih tajmera. U tom slučaju se on postavlja da istekne kada sledeći softverski tajmer treba da istekne. Po isteku, prekidna rutina ažurira hardverski brojač prema roku sledećeg softverskog brojača i okida akcije za softverski tajmer koji je upravo istekao.

Firmware

Firmware (upravljački softver) je termin koji se koristi za označavanje malih programa koji upravljaju radom elektronskih uređaja, npr. mobilnih telefona, prenosnih muzičkih uređaja, računarskih komponenata- hard diskova, tastatura, TFT monitora. Originalno je

predstavljao mikrokod koji definiše i implementira skup mašinskih instrukcija procesora. Razlikovao se od hardvera (samog procesora) i softvera (programa koji se izvršavaju u procesoru), međutim danas nema striktno granice između firmvera i softvera. Jednostavniji firmver je obično smešten u ROM memoriju dok kompleksniji često podrazumeva *flash* memoriju koja omogućava ažuriranja kao što su ispravljanje grešaka ili dodavanje novih funkcionalnosti.

2.7 Organizacija procesora

Za povezivanje registara, ALU jedinice i upravljačke jedinice (CU) procesora, obično se koristi jedna ili više internih magistrala. Upotreba zajedničkih putanja podataka pojednostavljuje prikaz međusobne povezanosti komponenata i štedi prostor što je posebno bitno kod mikroprocesora malih dimenzija. Različite tehnike implementacije upravljačke jedinice procesora se mogu grupisati u dve kategorije: ožičena i mikroprogramska realizacija. CU ožičene realizacije je u suštini kombinaciona mreža čiji se ulazni logički signali transformišu u upravljačke signale na izlazu. Projektovanje i testiranje ovakve mreže osnovnih logičkih elemenata je komplikovano; pored toga nije jednostavna ni promena dizajna radi dodavanja nove instrukcije. Alternativa je mikroprogramska implementacija CU čija je logika određena mikroprogramom.

Organizaciona unapređenja procesora mogu da poboljšaju performanse. Pored upotrebe *cache* memorije i skupa registara umesto jednog akumulatora, uobičajeni pristup je instrukcijski *pipeline*. Slično fabričkoj traci, koristi se činjenica da instrukcija sekvencijalno prolazi kroz različite faze obrade (npr. dohvaćanje instrukcije, dekodiranje, dohvaćanje operandi, izvršavanje i upisivanje rezultata) tako da svi stepeni *pipeline*-a mogu u jednom trenutku da obrađuju različite instrukcije. Grananja i zavisnosti među instrukcijama komplikuju dizajn i upotrebu *pipeline*-a.

Implementacija von Neumann-ove mašine

Mašina *von Neumann*-ove arhitekture se sastoji iz jedne memorije koja služi za skladištenje instrukcija i podataka, procesorske jedinice i ulazno-izlazne jedinice za komuniciranje sa korisnicima. Procesorska jedinica obuhvata kontrolnu jedinicu i ALU aritmetičko-logičku jedinicu zaduženu za izvršavanje elementarnih operacija.

Programski brojač služi za adresiranje i dohvaćanje mašinskih instrukcija iz memorije (pri tome se koriste i registri MAR i MBR). Pročitana instrukcija se smešta u prihvatni IR registar procesora. Sledi izračunavanje adresa i dohvaćanje operandi iz memorije. Zatim ALU izvršava traženu operaciju, rezultat se smešta na odredište, ispituje se da li je u međuvremenu stigao signal prekida, pa ako jeste prelazi se na program obrade prekida. Rad procesora čini ova sekvencija koraka koja se neprestano ponavlja.

Opisani metod rada se zove princip programskog upravljanja i na njemu je zasnovan rad svih tradicionalnih računara sa *von Neumann*-ovom arhitekturom. Prednost se sastoji u jednostavnosti dok je osnovni nedostatak sekvencijalnost u radu koja ima za posledicu ograničenje maksimalnih performansi računara.

Operaciona jedinica sa jednom ili sa više magistrala

Prenos podataka između registara procesora i ALU jedinice može biti organizovan na različite načine. Operacione jedinice procesora identične arhitekture mogu biti realizovane na različite načine. Najčešće realizacije su operaciona jedinica sa direktnim vezama, sa jednom, dve ili sa tri magistrale. Operaciona jedinica sa direktnim vezama nema lokalnu magistralu već su prekidačke mreže povezane direktno. Kod operacione jedinice sa dve magistrale, moguće je istovremeno preneti dva podatka do ulaza ALU jedinice. U slučaju operacione jedinice sa jednom magistralom, za to bi trebalo da prođe dva ciklusa signala takta. Kod organizacije sa dve magistrale, jedna magistrala može da služi za prenos podataka u registre (*in-bus*), dok je druga zadužena za prenos iz registara (*out-bus*). Organizacija sa tri magistrale (dve magistrale tipa *in-bus* i jedna *out-bus*) je najsloženija i najskuplja, ali zbog većeg broja magistrala se više podataka može preneti tokom jednog ciklusa signala takta.

Instruction set arhitektura - ISA

ISA je deo arhitekture računara koji specificira skup instrukcija, skup opkodova, formate instrukcija, načine adresiranja, registre, mehanizam prekida i sve ono što treba da se zna o računaru da bi mogli da se napišu programi koji će uvek davati identične rezultate bez obzira kako je računar realizovan. Mikroarhitektura (organizacija računara) je skup tehnika dizajna kojima se implementira tj. realizuje skup instrukcija. Drugim rečima, organizacija računara opisuje veze između delova sistema i njihovu međusobnu komunikaciju čime se implementira ISA. Računari različite mikroarhitekture mogu imati istu *instruction set* arhitekturu. Na osnovu lokacija operanada, ISA se može podeliti na Stek, Akumulator, Registar - Memorija, Registar - Registar i Memorija - Memorija arhitekturu. Na osnovu broja operanada, mašine mogu biti nulaadresne, jednoadresne, dvoadresne i troadresne. Na osnovu gustine koda (*code density*) razlikuju se RISC i CISC arhitekture.

Odnos između arhitekture i kompajlera

Razumevanje tehnologije kompajlera je bitno za efikasan dizajn i implementaciju seta instrukcija. Izbori u arhitekturi utiču na kvalitet generisanog koda i na složenost odgovarajućeg kompajlera. Današnji kompajleri se obično sastoje od dva do četiri prolaza koji postepeno transformišu reprezentacije višeg nivoa u reprezentacije nižeg nivoa, sve dok se ne dosegne set instrukcija na kraju. Najvažniji cilj u pisanju kompajlera je korektnost i brzina prevedenog koda. Jedna od najvažnijih optimizacija koje izvršavaju moderni kompajleri je alokacija registara (značajno za ubrzanje koda i za druge vrste optimizacija; npr. ako postoji više instanci izraza koji izračunavaju istu vrednost, onda se ta vrednost posle prvog izračunavanja čuva u registru i koristi umesto preostalih izračunavanja). Algoritam alokacije registara koji se često koristi – *graph coloring* radi kako treba samo kod arhitekture sa 16 ili više registara opšte namene.

Neke osobine seta instrukcija mogu služiti kao smernice pri pisanju kompajlera koji treba da generiše efikasan kod. Kad god to ima smisla, tri osnovne komponente instrukcije (operacija, tipovi podataka i načini adresiranja) trebaju biti ortogonalne odnosno

nezavisne (npr. da se sve instrukcije prenosa mogu kombinovati sa svim podržanim načinima adresiranja). Ovo pojednostavljuje generisanje koda. Kod Registar-Memorija arhitektura bitno je odlučivanje o tome koliko puta promenljiva treba da bude referisana pre nego se prebaci iz memorije u registar. Ovaj prag se teško izračunava i varira u zavisnosti od modela arhitekture.

Implementacija instrukcija

Implementaciju mašinske instrukcije čini niz mikrooperacija. Svaka mikrooperacija je povezana sa određenim skupom upravljačkih signala koji moraju biti aktivirani kako bi se izvršila data mikrooperacija. Zavisno od organizacije procesora, više mikrooperacija bi moglo da se izvršava u paraleli.

Npr. implementacija faze čitanja instrukcije kod operacione jedinice sa jednom magistralom može biti sledeća sekvenca mikrooperacija (znak ; razdvaja mikrooperacije koje se izvršavaju u paraleli):

$MAR \leftarrow (PC); A \leftarrow (PC)$

$MDR \leftarrow Mem[MAR]; PC \leftarrow (A) + 4$

$IR \leftarrow (MDR)$

Ili npr. implementacija faze izvršavanja operacije sabiranja sadržaja registara R_1 i R_2 i smeštanje rezultata operacije u registar R_0 :

$A \leftarrow (R_1)$

$B \leftarrow (R_2)$

$R_0 \leftarrow (A) + (B)$

Ožičena ili mikroprogramska realizacija upravljačke jedinice

Upravljačka jedinica upravlja tokom podataka unutar procesora i između procesora i drugih modula. Razlikuju se dve osnovne tehnike realizacije: ožičena i mikroprogramska. Ožičena realizacija je ekonomičnije rešenje za male upravljačke jedinice. Brža je od mikroprogramske realizacije, ali se teže prilagođava promenama. Kod mikroprogramske realizacije možemo lako dodati nove instrukcije bez menjanja hardvera.

Ožičena implementacija je direktna implementacija gde fiksna logička kola direktno korespondiraju *boolean* izrazima za generisanje upravljačkih signala. Upravljačka jedinica se sastoji iz brojača, dekodera, kombinacione mreže za generisanje nove vrednosti brojača (u slučaju da treba izvršiti skok; inače se vrednost brojača samo inkrementira) i kombinacione mreže za generisanje upravljačkih signala. Svakom koraku iz sekvence upravljačkih signala odgovara jedno dekodovano stanje brojača. Ova stanja služe za generisanje upravljačkih signala operacione jedinice (koji upravljaju izvršavanjem mikrooperacija), dok zajedno sa signalima logičkih uslova iz operacione jedinice služe za generisanje upravljačkih signala upravljačke jedinice (neophodnih za promenu vrednosti brojača prilikom skoka).

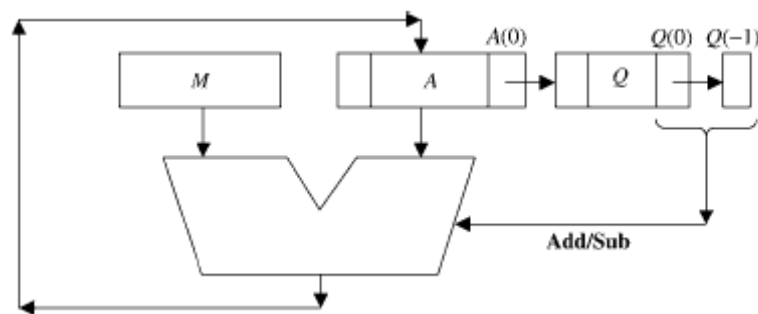
U mikroprogramskoj realizaciji se svakom koraku iz sekvence upravljačkih signala pridružuje jedna mikroinstrukcija (specificira jednu ili više mikrooperacija). Niz mikroinstrukcija čini mikroprogram koji se smešta u brzu mikroprogramsku memoriju (CM - *control memory, control store*). Drugi delovi upravljačke jedinice sa ovom

realizacijom su brojač (vrednost brojača predstavlja adresu u CM sa koje treba pročitati mikroinstrukciju), kombinaciona mreža za generisanje nove vrednosti brojača (u slučaju da treba izvršiti skok; inače se vrednost brojača samo inkrementira), prihvatni registar (služi za prihvatanje mikroinstrukcije očitane iz CM) i kombinaciona mreža za generisanje upravljačkih signala pomoću sadržaja prihvatnog registra mikroinstrukcije i signala logičkih uslova iz operacione jedinice. Mikroprogramska realizacija može biti sa jednim tipom mikroinstrukcija ili sa dva tipa mikroinstrukcija (operaciona za operacione korake - kodira upravljačke signale operacione jedinice i upravljačka za upravljačke korake - kodira upravljačke signale upravljačke jedinice). U slučaju sekvence upravljačkih signala sa spajanjem koraka mora biti realizacija samo sa jednim tipom mikroinstrukcija koji kodira sve upravljačke signale.

Postoji više načina kodiranja upravljačkih signala operacione jedinice u mikroinstrukciji. Horizontalni format mikroinstrukcija omogućava maksimalni paralelizam jer za svaki upravljački signal postoji poseban bit u mikroinstrukciji (vr. 1 tog bita označava da dati signal treba da bude aktivan, vr. 0 obrnuto). U slučaju vertikalnog formata, u jednom koraku je moguće izvršiti samo jednu mikrooperaciju jer mikroinstrukcija kodira kombinaciju upravljačkih signala neophodnu za realizaciju samo jedne mikrooperacije u jednom koraku. Prednost ove realizacije je kraća mikroinstrukcija jer se kodira kombinacija signala a ne svaki signal posebno. Pozitivni efekti horizontalnog i vertikalnog kodiranja su iskorišćeni kod mešovitog formata – signali su podeljeni u grupe na način da signali iz istog koraka sekvence upravljačkih signala budu svaki u različitoj grupi. Unutar grupe je vertikalno kodiranje, a na nivou grupa je horizontalno kodiranje.

Aritmetičke jedinice za množenje i deljenje

Aritmetička jedinica koja realizuje *Booth*-ov algoritam za množenje se implementira pomoću aritmetičke jedinice za sabiranje/oduzimanje, 3 registra (za smeštanje dva množioca i jedan pomoćni registar) i upravljačke logike koja generiše potrebne signale i koja može da realizuje aritmetičko pomeranje udesno ASR konkatencije sadržaja registara.



Na sličan način se implementira aritmetička jedinica za deljenje prema *non-restoring* algoritmu: aritmetička jedinica za sabiranje/oduzimanje, 3 registra (za smeštanje deljenika, delioca i jedan pomoćni registar) i upravljačka logika koja generiše potrebne signale i koja može da realizuje pomeranje ulevo konkatencije sadržaja registara.

Instrukcijski pipeline

U slučaju tradicionalnog sekvencijalnog izvršavanja instrukcija, izvršavanje jedne instrukcije ne počinje dok se ne završi prethodna instrukcija. Pošto izvršavanjem instrukcija prolazi redom kroz niz faza (npr. dohvatanje instrukcije, njeno dekodiranje, dohvatanje operanada, izvršavanje operacije i smeštanje rezultata), delovi procesora zaduženi za izvršavanje određenih faza će biti besposleni dok instrukcija prolazi kroz druge faze.

Kod instrukcijskog *pipeline*-a, funkcionalne jedinice procesora mogu simultano da izvršavaju odgovarajuće faze različitih instrukcija. Tako se npr. u *pipeline*-u sa 5 stepeni preklapa izvršavanje faza od maksimalno 5 instrukcija (svaka instrukcija u drugom stepenu). Princip je sličan fabričkoj traci. Paralelno izvršavanje delova više instrukcija u *pipeline*-u poboljšava performanse procesora u poređenju sa tradicionalnim izvršavanjem. Korišćenje keš-a u procesoru je omogućilo da se frekvencija rada procesora mnogo brže povećava od frekvencije rada glavne memorije. *Pipeline* kod procesora bez brze keš memorije ne bi imao mnogo smisla.

Trendovi u arhitekturi računara: CISC, RISC, VLIW

Po jednoj filozofiji treba što više uraditi u pojedinačnoj instrukciji, kako bi se za obavljanje istog posla koristio manji broj instrukcija. Povećanje kompleksnosti instrukcija i povećanje ukupnog broja načina adresiranja otežava postizanje visokih frekvencija signala takta, koje bi inače bile moguće. Mašine koje podržavaju ovu filozofiju se označavaju kao CISC računari (*complex intructions set computers*), npr. Intel Pentium, Macintosh PowerPC.

Istraživanja početkom 1980-ih godina su otkrila da kod tipičnih programa, 80% svih instrukcija čine dodele vrednosti, uslovna grananja i pozivi procedura. Zato se optimizacija arhitekture može postići smanjivanjem složenosti instrukcija i ubrzavanjem najčešće korišćenih instrukcija. Mašine koje podržavaju ovu filozofiju se označavaju kao RISC računari (*reduced intructions set computers*), npr. Sun SPARC, MIPS.

Kod VLIW (*very long instruction word*) arhitektura, jedna instrukcija specificira više operacija koje treba simultano izvršiti. Kompajler je zadužen za proveravanje svih zavisnosti i pravljenje fiksnog rasporeda u paralelnom izvršavanju operacija. Kod VLIW arhitektura se povećava složenost kompajlera, (umesto hardvera kao kod superskalar arhitektura gde je za raspoređivanje operacija zadužen hardver).

Instruction-level paralelizam (ILP)

Performanse najjednostavnijeg tipa procesora, koji izvršava jednu instrukciju u svakom trenutku, su skoro uvek subskalarne (protok je manji od jedne instrukcije u jedinici vremena). Radi postizanja boljih performansi (skalarnih i superskalarnih), razvijene su razne metodologije dizajna procesora, jedna od njih je *instruction-level parallelism* ILP kojom se povećava frekvencija izvršavanja instrukcija. ILP tehnike obuhvataju *pipeline*, *superscalar* arhitekture i VLIW arhitekture koje imaju ILP primenjen direktno u softveru. U najboljem slučaju kod pipeline arhitekture (izvršava delove instrukcija istovremeno) postiže se skalarni protok od jedne instrukcije u jedinici vremena. Kod superskalarnih

arhitektura umnožavanje funkcionalnih jedinica (npr. više *load* jedinica, *integer* jedinica, *floatin- point* jedinica..) omogućava da se više instrukcija potpuno paralelno izvršava (protok je veći od jedan). Svaka instrukcija kodira samo jednu operaciju pa mora postojati deo hardvera koji je zadužen za raspoređivanje operacija koje mogu istovremeno da se izvršavaju. U slučaju VLIW pristupa, kompajler je zadužen za analizu zavisnosti i grupisanje operacija koje mogu simultano da se izvršavaju u *very long instruction word*.

Pipeline hazardi

Pipeline hazardi su situacije kada instrukcija treba da ostane u nekom stepenu *pipeline*-a duže od predviđenog vremena. Ovo za posledicu ima da prethodni stepeni postaju besposleni a instrukcije kasne sa izvršavanjem, što utiče na performanse. Postoje tri vrste hazarda: strukturalni, upravljački i hazardi podataka.

Strukturalni hazard se javlja kad dve instrukcije u različitim stepenima treba da pristupe istom resursu. Npr. ako se koristi ista memorija za instrukcije i za podatke, očitavanje nove instrukcije ne bi moglo da se realizuje u istom taktu kada druga instrukcija (*load* ili *store*) pristupa toj memoriji – rešenje je postojanje posebne memorije za instrukcije i posebne za podatke. Za složenije operacije (sa veličinama u pokretnom zarezu ili množenje i deljenje celobrojnih veličina), treba više taktova. Da to ne bi kočilo *pipeline*, izvršni stepen se realizuje kao paralelna veza više funkcionalnih jedinica. Kada neka instrukcija dođe do izvršnog stepena, šalje se u odgovarajuću slobodnu jedinicu. Problem nastaje kada su sve jedinice zauzete pa instrukcija mora da čeka u prethodnom stepenu. Rešenje može predstavljati veći broj funkcionalnih jedinica.

Upravljački hazard nastaje kada od rezultata izvršavanja jedne instrukcije zavisi koja će biti sledeća instrukcija za dohvaćanje. Primer je izvršavanje instrukcije uslovnog skoka ili instrukcije koje menjaju vrednost programskog brojača jer je nova vrednost programskog brojača poznata tek na kraju izvršavanja.

Hazard podataka se javlja zbog izmenjenog redosleda pristupa podacima u odnosu na procesor bez *pipeline* organizacije. Hazardi podataka se mogu svrstati u tri grupe: RAW (*read after write*; instrukcija pokušava da pročita podatak pre nego što je prethodna instrukcija upisala novu vrednost u taj podatak), WAR (instrukcija pokušava da upiše vrednost u podatak pre nego što je prethodna instrukcija očitala staru vrednost tog podatka) i WAW (zbog izmenjenog redosleda upisa, u podatku će ostati zapamćena stara vrednost umesto nove). Još jedan slučaj izmenjenog redosleda pristupa može da se javi - RAR (*read after read*) ali on ne predstavlja hazard.

Umanjivanje efekata hazarda

Najjednostavnija metoda za obezbeđivanje korektnog izvršavanja instrukcija u *pipeline*-u je zaustavljanje *pipeline*-a onoliko vremena koliko je potrebno da se utvrdi nova vrednost programskog brojača ili potrebnog podatka. Implementira se umetanjem jedne ili više NOP instrukcija (*no operation*) koje obezbeđuju potrebno kašnjenje a ne utiču na stanje procesora. Iako je primamljiva metoda jer je jednostavna za realizaciju, značajno usporavanje *pipeline*-a se odražava na njegove performanse.

Za umanjivanje ili potpuno eliminisanje kašnjenja *pipeline*-a usled upravljačkih hazarda koriste se sledeće metode:

- upotreba posebnog hardvera u sklopu *fetch* jedinice koji je sposoban za prepoznavanje instrukcije skoka i izračunavanje ciljne adrese (umesto standardne obrade instrukcije u jedinici za izvršavanje). Ovim se redukuju upravljački hazardi ali se javljaju novi hazardi podataka (instrukcija uslovnog skoka mora da zna rezultat prethodne instrukcije prilikom odlučivanja o skoku).

- softverska metoda zakašnjen skok (*delayed branch*) kompajler treba da preuredi sekvencu koda tako što posle instrukcije uslovnog skoka stavlja onoliko instrukcija koliko bi inače trebalo zakasniti *pipeline* do utvrđivanja adrese skoka. Umeću se instrukcije koje inače prethode *branch* instrukciji i izvršavaju se bez obzira da li će biti skoka ili ne.

- predviđanje skoka (*branch prediction*) je u stvari metoda predviđanja sledeće instrukcije koju treba dohvatiti posle instrukcije uslovnog skoka. Ako je predikcija da će skok biti napravljen, izvršavanje se nastavlja od adrese skoka (alternativno ako je predviđanje da nema skoka, izvršavanje se nastavlja sekvencijalno). Tek na kraju izvršavanja *branch* instrukcije će se znati da li je predikcija bila ispravna (nastavlja se izvršavanje, nema izgubljenog vremena) ili pogrešna (neophodno je restaurirati status procesora koji je bio pre predviđanja i očistiti *pipeline* od pogrešnih instrukcija - *flush*).

Za umanjivanje ili potpuno eliminisanje kašnjenja *pipeline*-a usled hazarda podataka koriste se sledeće metode:

- prosleđivanje (*forwarding*) podrazumeva da se hardverski proverava da li je neko izvorište operanada date instrukcije odredište za rezultat izvršavanja neke od prethodnih instrukcija (hazard tipa RAW). Ako je to slučaj, onda se rezultat jednog stepena direktno prosleđuje na ulaz istog ili nekog drugog stepena *pipeline*-a.

- softverska metoda zakašnjeno punjenje (*delayed load*) kompajler treba da preuredi sekvencu koda tako da se izbegne da se neposredno iza *load* instrukcije nalazi instrukcija koja za izvorište ima registar koji je odredište prethodne *load* instrukcije.

2.8 Performanse

Procena performansi računarskog sistema može biti pravi izazov; tome doprinosi kompleksnost modernih softverskih sistema zajedno sa širokim opsegom tehnika koje koriste projektanti hardvera za poboljšanje performansi. Različitim tipovima aplikacija odgovaraju različite mere performansi, takođe različiti aspekti sistema mogu biti ključni u određivanju ukupnih performansi. Za korisnika desktop računara bitno je vreme potrebno za izvršenje određenog zadatka (*response time*), dok je za menadžera servera koji izvršavaju zadatke za više korisnika bitan protok (*throughput*) odnosno ukupna količina posla koja se odradi u određenom intervalu.

Mere performansi računara; frekvencija signala takta, MIPS, CPI, *benchmarks*

Signal takta je periodični signal koji služi za sinhronizaciju rada delova računara. Trajanje periode signala takta je određeno najvećim vremenom potrebnim za prenos nekog signala. Što je veća frekvencija signala takta, znači da se signali brže prenose.

Vreme potrebno za izvršenje posla od strane računara se često izražava u ciklusima signala takta, jer je smeštanje rezultata izvršavanja sinhronizovano sa rastućom (ili opadajućom) ivicom ovog signala.

Jedna mera performansi računara je CPI (*cycles per instruction*), prosečan broj ciklusa signala takta po instrukciji.

$$CPI = \frac{ClockCyclesForProgram}{InstructionCount}, \quad CPI = \frac{\sum_{i=1}^n CPI_i \times I_i}{InstructionCount},$$

gde je I_i broj pojavljivanja instrukcije tipa i u programu, a CPI_i je CPI za instrukciju tipa i .

MIPS, (*million instructions per second*) opisuje učestanost instrukcija generalno, a postoji i MFLOP, (*million floating-point instructions per second*) koji se definiše za podskup *floating-point* instrukcija.

$$MIPS = \frac{InstructionCount}{ExecutionTime \times 10^6} = \frac{ClockRate}{CPI \times 10^6}$$

$$MFLOPS = \frac{Floating - point InstructionCount}{ExecutionTime \times 10^6}$$

Benchmarks su standardizovani testovi koji se sastoje iz različitih programa, i služe za procenu efektivnih performansi računara. Najbolji tip programa za ovu namenu su realne aplikacije koje korisnici redovno upotrebljavaju.

Snaga i slabost mera performansi

CPI odražava organizaciju i *instruction set* arhitekturu procesora, dok broj instrukcija odražava *instruction set* arhitekturu i korišćenu kompajlersku tehnologiju. Zato je neophodno razmatrati obe vrednosti prilikom procene performansi računara.

MIPS se ne može koristiti za upoređivanje mašina sa različitim skupovima instrukcija.

Na performanse procesora značajno utiču performanse memorijske hijerarhije, što nije uključeno u MIPS izračunavanjima. *Benchmark* testovi omogućavaju poređenje performansi raznih podsistema u različitim arhitekturama.

Aritmetička, geometrijska, harmonijska sredina kao mera performansi

Najpopularniji način da se izraze performanse računara u odnosu na veće skupove programa (*benchmark suites*) su aritmetička, geometrijska i harmonijska sredina vremena izvršavanja.

Prosek vremena izvršavanja je aritmetička sredina:

$$ArithmeticMean = \frac{1}{n} \sum_{i=1}^n ExecutionTime_i ;$$

Prosečno normalizovano vreme izvršavanja se može izraziti kao aritmetička ili kao geometrijska sredina:

$$GeometricMean = \sqrt[n]{\prod_{i=1}^n ExecutionTimeRatio_i} ; \quad ExecutionTimeRatio_i \quad \text{je} \quad \text{vreme}$$

izvršavanja i -tog programa, normalizovano prema referentnoj mašini.

Ako se performansa izražava kao frekvencija, onda je prosek harmonijska sredina:

$$HarmonicMean = \frac{n}{\sum_{i=1}^n \frac{1}{ExecutionTime_i}} ; \quad ExecutionTime_i \quad \text{je} \quad \text{vreme izvršavanja } i\text{-tog}$$

programa, a n je ukupan broj programa u *benchmarks* skupu.

Uloga Amdahl-ovog zakona u računarskim performansama

George Amdahl je 1967. godine prepoznao vezu između komponenata i ukupne efikasnosti računarskog sistema. Svoja opažanja je izrazio formulom koja je sada poznata kao Amdahl-ov zakon i koja u suštini govori da ubrzanje sistema zavisi i od ubrzanja određene komponente i od toga koliko se ona koristi u sistemu.

Ubrzanje (*speedup*) je mera performansi računara posle poboljšanja u odnosu na originalne performanse:

$$SU = \frac{PerformansePoslePoboljšanja}{PerformansePrePoboljšanja} = \frac{VremeIzvršavanjaPrePoboljšanja}{VremeIzvršavanjaPoslePoboljšanja}$$

Amdahl-ov zakon služi za izračunavanje ubrzanja ukupnih performansi računara kada je primenjeno poboljšanje performansi moguće samo deo vremena:

$$SU \equiv \frac{1}{(1 - \Delta) + \frac{\Delta}{SU_{\Delta}}} , \quad SU_{\Delta} \quad \text{je} \quad \text{ubrzanje moguće za deo vremena } \Delta$$

Prethodna formula može biti generalizovana za slučaj kada je više nezavisnih poboljšanja primenjeno u različitim delovima vremena $\Delta_1, \Delta_2, \dots, \Delta_n$ što dovodi do ubrzanja $SU_{\Delta_1}, \dots, SU_{\Delta_n}$:

$$SU = \frac{1}{[1 - (\Delta_1 + \Delta_2 + \dots + \Delta_n)] + \frac{\Delta_1 + \Delta_2 + \dots + \Delta_n}{SU_{\Delta_1} + SU_{\Delta_2} + \dots + SU_{\Delta_n}}}$$

2.9 Modeli distribuiranih sistema

Distribuirani sistem predstavlja kolekciju pojedinačnih računarskih uređaja koji mogu da komuniciraju međusobno. Ova opšta definicija obuhvata širok opseg modernih računarskih sistema (npr. multiprocesori, klasteri, Internet). Oni tipično pružaju mogućnost deljenja resursa i podataka, poboljšavaju pristupačnost u slučaju otkaza nekih komponenata, dok ambiciozniji sistemi pokušavaju obezbediti unapređenje performansi rešavanjem potproblema u paraleli. Iako su veoma poželjni, sklapanje ovakvih sistema obuhvata teškoće zbog asinhronosti, otkaza, heterogenog hardvera i softvera, nedostatka pridržavanja standarda.

Klasifikacija modela računara: *Flynn's taxonomy, Handler's classification*

Za svrhu poređenja računara bez zalaženja u detalje, oni se klasifikuju na osnovu svega nekoliko karakteristika. Cilj klasifikovanja je da se arhitekture sa zajedničkim osobinama grupišu u jednu kategoriju i da kategorije budu disjunktne (određena arhitektura treba da pripada samo jednoj kategoriji). U praksi to nije lako postići jer neke mašine imaju sposobnost da se mogu rekonfigurisati ili da mogu raditi u nekoliko različitih modova.

Flynn-ova klasifikacija se zasniva na identifikovanju dva ortogonalna toka u računaru: tok (sekvenca) instrukcija koje izvršava računar i tok (prenos) podataka između memorije i procesorske jedinice. Svaki od ova dva toka može biti jednostruki ili višestruki, na osnovu čega se razlikuju četiri kategorije arhitekture:

- *SISD (single-instruction single-data streams)* računari sa jednim procesorom koji izvršava jedan tok instrukcija nad jednim tokom podataka, npr. *von Neumann* arhitektura. Izvršavanje instrukcija može biti preklapljeno upotrebom *pipeline* tehnike ili višestrukih funkcionalnih jedinica.

- *SIMD (single-instruction multiple-data streams)* kategorija arhitektura koje postižu *data level* paralelizam. Više procesorskih jedinica pod kontrolom jedne upravljačke jedinice izvršava istu instrukciju nad različitim podacima, npr. *vector* procesori.

- *MISD (multiple-instruction single-data streams)* više procesorskih jedinica izvršava različite operacije nad istim podacima. Nema mnogo primera ovog tipa arhitekture. *SIMD* i *MIMD* su mnogo zastupljenije od *MISD* arhitekture jer bolje koriste računarske resurse.

- *MIMD (multiple-instruction multiple-data streams)* sistem ima više procesora koji funkcionišu asinhrono i nezavisno (svaki izvršava svoj tok instrukcija nad svojim tokom podataka) npr. multiprocesori i multiračunari.

1977. godine *Handler* je predložio složenu notaciju za izražavanje paralelizma u računarima pomoću tri različita nivoa: kontrolna jedinica procesora CU, aritmetičko-logička jedinica ALU i *bit-level* kolo BLC. *Handler*-ova klasifikacija koristi tri para celih brojeva za opisivanje računara:

```
Računar = (k * k', d * d', w * w')
k = broj CUs
k' = broj CUs koji mogu biti pipelined
d = broj ALUs koje kontroliše svaka CU
d' = broj ALUs koji mogu biti pipelined
w = broj bitova u reči
w' = broj pipeline stepena
```

Primena operatora \vee označava da hardver može raditi u više modova, operator $+$ ukazuje da jedinice rade na nezavisnim tokovima podataka, a simbol \sim se koristi za označavanje opsega vrednosti. Ako drugi broj u paru ima vrednost jedan, može se izostaviti. Periferni procesori se takođe opisuju koristeći tri para brojeva, pre glavnog procesora.

Granularnost, nivoi paralelizma

Granularnost predstavlja odnos vremena izračunavanja prema vremenu komunikacije. *Fine-grained* paralelizam označava da su procesi mali, u smislu količine koda i vremena izvršavanja, a da se podaci često prenose između procesora. *Coarse-grained* paralelizam predstavlja suprotnost: podaci se ređe prenose posle dužeg izračunavanja. Što je manja granularnost, veća je sposobnost paralelizma ali su onda veći i troškovi zbog sinhronizacije i komunikacije.

Bit level parallelism

Ubrzanje u računarskoj arhitekturi se postiže dupliranjem veličine reči, tj. količine informacija koju procesor obradi za jedan ciklus signala takta. Na taj način se smanjuje broj instrukcija koje procesor mora da izvrši da bi se kompletirala operacija nad podacima čija je veličina veća od veličine reči. Istorijski, 4-bitne procesore su zamenili 8-bitni, zatim 16-bitni, 32-bitni i u skorije vreme 64-bitni.

Instruction level parallelism

Ako se programske instrukcije preurede i grupišu na odgovarajući način, moguće ih je izvršavati u paraleli i dobiti isti rezultat kao kod sekvencijalnog izvršavanja. Primeri implementacije su *pipeline* i *superscalar* arhitekture.

Data level parallelism

Ova vrsta paralelizma se koristi kod vektorskih (*array*) procesora: pomoću jedne instrukcije se izvršava operacija nad skupom podataka umesto nad samo jednim ili dva podatka kao kod skalarnih procesora. Poboljšanje performansi se ogleda u smanjenju vremena koje se troši na dekodiranje.

Thread (task) level parallelism

Vreme izvršavanja koda se smanjuje simultanim procesuiranjem niti (programa) na različitim procesorima kod multiprocesorskih sistema, ili u okviru jednog procesora kod *multithreading* arhitekture.

Multiprocesori i multiračunari; *tightly coupled* i *loosely coupled* arhitekture

Prema modelu komunikacije (*communication model*), MIMD mašine se dele na:

- *Shared memory* multiprocesori

Procesori u okviru jednog računara komuniciraju čitanjem i upisivanjem na lokacije u *shared* memoriji - deljenoj između procesora. U slučaju *centralized shared memory* arhitekture vreme pristupa glavnoj memoriji je uvek isto pa se ovi sistemi zovu i UMA (*uniform memory access*). *Distributed shared memory* arhitekture imaju deljenu memoriju fizički distribuiranu među procesorima tako da procesor poseduje lokalnu memoriju ali može da pristupi i memoriji drugog procesora. Zbog razlike u vremenu pristupa tim memorijama, sistem se zove i NUMA (*non-uniform memory access*).

Neki od primera multiprocesorskih tehnika su CMP (*chip-level multiprocessing*, smanjenje veličine tranzistora je omogućilo da se više procesora implementira na jednom

silikonskom čipu – *multicore* procesori) i SMP (*symmetric multiprocessing*, mali broj procesora, do 32, povezanih pomoću magistrale deli koherentan pogled na memorijski sistem).

- *Distributed memory* multiračunari

Procesori međusobno komuniciraju slanjem poruka (*message passing*) preko interkonekcijske mreže jer je memorija i fizički i logički distribuirana. Svaki procesor ima svoju lokalnu memoriju i svoj adresni prostor. Neki od primera su klasteri (grupa *loosely coupled* računara koji se nalaze blizu jedan drugog, povezani su mrežom, rade zajedno i smatraju se jednim računarom) i gridovi (računari koji međusobno komuniciraju preko Interneta se koriste za rešavanje zadataka; *middleware* softver između operativnog sistema i grid aplikacije upravlja mrežnim resursima i standardizuje interfejs).

Fizička veza (*physical connection* PC) u MIMD sistemima može biti bazirana na magistrali (*bus*, procesori su direktno povezani) ili na mreži (procesor je direktno povezan samo sa nekoliko drugih procesora) čija je topologija tipa *star*, *ring*, *tree*, *hypercube*, *mesh*...

Kod *loosely coupled* arhitektura, moduli nisu zavisni jedan od drugog, jednostavno se dodaju ili menjaju u sistemu (npr. računari povezani u mreži). Te osobine ne poseduju *tightly coupled* sistemi.

Procesi: niti, klijenti, serveri, softverski agenti

Proces predstavlja izvršavanje instance programa. Moguće je kreirati više procesa nad istim programom; svaki radi nad svojim podacima tj. ima svoj adresni prostor. Takođe, jedan proces može da se podeli na više niti koje izvršavaju deo koda procesa. Pošto sve niti jednog procesa dele njegove resurse i adresni prostor, promena konteksta između niti je mnogo brža nego između procesa.

Jedna vrsta distribuirane arhitekture je klijent-server arhitektura (npr. *web browser* i *web server*). Serveri (*service provider*) očekuju dolazeće zahteve klijenata, prihvataju ih, obrađuju i šalju traženu informaciju klijentu. Za razliku od servera, klijenti ne dele svoje resurse. Server može da opslužuje zahteve klijenata u paraleli kreiranjem posebne niti za svaki primljeni zahtev. U slučaju povećanog broja istovremenih klijentskih zahteva, server može postati preopterećen. Tada je bolje rešenje *peer-to-peer* arhitektura P2P jer se propusna moć u stvari povećava dodavanjem čvorova. Svaki host ima isti status, može da se ponaša i kao klijent i kao server. Resursi su distribuirani između više čvorova, pa ako se jedan pokvari, zahtev će i dalje moći da se opsluži.

Softverski agent je računarski program ovlašćen da obavlja određenu vrstu posla i koji odlučuje o akcijama koje treba sprovesti. Neke vrste agenata su: inteligentni softverski agent (ISA; koristi veštačku inteligenciju da bi izvršio zadatak, ima sposobnost učenja i zaključivanja), *mobile* agent (premešta svoj kod i stanje sa jedne mašine na drugu da bi došao do potrebnih podataka), *fuzzy* agent (inteligentni agent koji implementira *fuzzy* logiku), *multiple* agenti (distribuirani agenti koji nemaju sve podatke da bi mogli sami da izvrše zadatak pa sarađuju sa drugim agentima).

Clock sinhronizacija, Lamport i vektorske vremenske oznake

Clock sinhronizacija je problem zbog razlika koje postoje između generatora signala takta više računara. Čak i kada se inicijalno precizno postave, posle nekog vremena više neće biti sinhronizovani jer im se neznatno razlikuju frekvencije rada. Problem dolazi do izražaja u distribuiranim sistemima gde više računara mora da zna isto globalno vreme. U slučaju centralizovanog sistema rešenje je jednostavno, svi računari u mreži se sinhronizuju prema centralnom serveru (npr. *Cristian* ili *Berkeley* algoritam). Kod Interneta je najčešće rešenje NTP protokol (*Network Time Protocol*).

Koncept *logical clock* polazi od pretpostavke da nije važno apsolutno vreme kad su se desili događaji već je bitno da procesi znaju redosled njihovog dešavanja. Lamportov algoritam tako podrazumeva sledeće:

- proces ima svoj lokalni brojač *clock* C, koga inkrementira pre svakog lokalnog događaja; svakom događaju x se dodeljuje vremenska oznaka *timestamp* C(x) i ne može se desiti da dva događaja iz istog procesa imaju iste vremenske oznake; da bi se razlikovali događaji koji su se desili simultano u različitim procesima, ID procesa se pridružuje vremenskoj oznaci;
- kad šalje poruku, proces uključuje trenutnu vrednost svog brojača u poruku;
- kad prima poruku, proces postavlja vrednost svog brojača da bude veća od maksimuma primljene vrednosti i njegove vrednosti lokalnog brojača.

Kod *vector clock* mehanizma, svaki proces sadrži lokalnu kopiju vektorskog *clock*-a, koji predstavlja niz *logical clock*-ova (po jedan za svaki proces). Pravila su sledeća:

- kad se desi lokalni događaj, proces inkrementira svoj *logical clock* u kopiji *vector clock*
- pre slanja poruke (zajedno sa svojom kopijom *vector clock*), proces inkrementira svoj *logical clock*
- kad proces primi poruku, on inkrementira svoj *logical clock* a sve ostale elemente vektora postavlja da budu maksimum stare vrednosti i vrednosti dobijene u poruci.

Election algoritmi

Election algoritmi služe za izbor procesa koji će da bude koordinator. Nije bitno koji će proces iz grupe biti izabran jer su svi isti.

Bully algoritam bira proces sa najvećim identifikatorom:

- kada proces p detektuje da koordinator više ne odgovara na zahteve, on šalje *election* poruku svim procesima sa većim identifikatorima; ako niko ne odgovori p postaje koordinator;
- ako proces primi *election* poruku od procesa sa manjim identifikatorom, on zadržava selekciju i šalje OK poruku natrag;
- proces koji pobeđi šalje poruku svim ostalim da je on novi koordinator.

Ring algoritam se koristi kad su procesi raspoređeni u prstenu i svaki zna za svog sledbenika. Onaj proces koji primeti kvar, šalje *election* poruku sa svojim identifikatorom susedu odakle se poruka dalje prenosi po prstenu (aktivni procesi dodaju svoje identifikatore u poruku) sve dok ne dođe do procesa koji ju je generisao. Lista identifikatora još jednom kruži po prstenu u okviru poruke *coordinator* tipa, pri čemu svaki proces bira najveći broj da bude koordinator. Ako više procesa detektuje kvar, više poruka će kružiti ali će rezultat biti isti.

Algoritmi međusobnog isključivanja

U centralizovanom sistemu *mutual exclusion* se postiže monitorima, semaforima, specijalnim *test and set* instrukcijama. Kada resurs zahteva ekskluzivan pristup u distribuiranoj arhitekturi, procesi koji mu pristupaju moraju međusobno da koordiniraju. Za tu svrhu postoje algoritmi:

Central server algoritam

Jedan proces se izabere da bude koordinator. Svaki proces koji želi da uđe u kritičnu sekciju (deo koda koji pristupa zajedničkom resursu), šalje koordinatorskoj poruci zahteva. Ako nikog nema u kritičnoj sekciji, on odgovara porukom dozvole. Inače ne šalje poruku, i zahtevalac se blokira. Kada završi pristup kritičnoj sekciji, proces šalje *release* poruku koordinatorskoj koja onda može da pošalje dozvolu procesu koji je čekao. Loša strana ovog mehanizma je što je centralni server usko grlo sistema.

Distributed mutual exclusion algoritam

Proces koji želi da uđe u kritičnu sekciju, šalje svim ostalim procesima zahtev sa svojim identifikatorom, identifikatorom krit. sekcije i trenutnim vremenom. Zatim čeka dok svi ne daju dozvolu i onda ulazi u krit. sekciju. Proces koji dobija zahtev i nalazi se u krit. sekciji ne šalje odgovor a zahtev stavlja u lokalni red zahteva. Po izlasku iz krit. sekcije će svima iz reda poslati dozvole i isprazniti red. Ako je proces primalac zahteva takođe zainteresovan za ulazak u krit. sekciju (poslao je svoj zahtev), on upoređuje vremensku oznaku iz primljene poruke sa svojom oznakom koju je poslao. Raniji *timestamp* pobeđuje. Mana ovog rešenja je veliki promet poruka.

Token ring algoritam

U softveru se konstruiše logički prsten tako što se svakom procesu dodeljuje pozicija u prstenu i svaki proces mora da zna ko mu je sledbenik. Token oznaka kruži u prstenu i u svakom trenutku se nalazi samo kod jednog procesa. Proces koji u nekom trenutku dobije token, proverava da li želi da uđe u krit. sekciju. Ako ne želi, token prosleđuje svom sledbeniku; inače ulazi u krit. sekciju i tek po izlasku šalje token dalje.

Distribuirane transakcije, *concurrency control*

Distribuirana transakcija je transakcija koja ažurira distribuirane podatke na više računara. Kao i obična transakcija, mora da ima sve četiri ACID osobine (atomičnost – iako se sastoji od skupa operacija, mora da izgleda nedeljivo; konzistentnost – mora da očuva konzistentnost podataka; izolacija – svaka od konkurentnih transakcija se mora ponašati kao da se samo ona izvršava u sistemu; izdržljivost – ako se računar pokvari neposredno posle *commit*-a, ažuriranja će se ipak izvršiti po restartovanju).

XA standard za procesiranje distribuiranih transakcija koristi *two-phase commit* protokol koji obezbeđuje da svi resursi simultano komituju. Svaki *host* ima lokalni transakcioni menadžer zadužen za kreiranje i upravljanje transakcijama. Kada treba izvršiti transakciju distribuiranu između nekoliko računara, transakcioni menadžer šalje *prepare* poruke svim podređenim menadžerima (nadređeno-podređena relacija se formira kad stigne zahtev za datu transakciju) jer svaki učesnik mora biti pripremljen da izvrši potrebnu akciju. Ako su sve komponente spremne, superiorni menadžer komituje transakciju (šalje *commit* poruku), inače šalje *abort* poruku svim učesnicima. Pri tome on vodi evidenciju *log* da bi se u slučaju kvara i restartovanja mogle kompletirati nedovršene operacije.

Concurrency control obezbeđuje tačnost rezultata konkurentnih operacija, do kojih se dolazi što je brže moguće. Najvažniji cilj distribuirane kontrole konkurentnosti je *serializability* – ishod konkurentnih transakcija mora biti isti kao kad bi se sekvencijalno izvršavale, bez preklapanja. Najčešće *distributed concurrency control* tehnike su *strong strict two-phase locking* SS2PL i *distributed lock manager* DML.

2.10 Poboljšanja performansi

Razna arhitektonska unapređivanja mogu da poboljšaju performanse sistema. Ovde su razmatrane sledeće tehnike:

- superskalarni procesor - popravljja performanse multiplikacijom hardvera,
- predviđanje skoka - *pipeline* procesor unapred pregleda kôd i nagađa koja grana, odnosno koji skup instrukcija treba sledeći da se izvršava,
- *prefetching* - smanjuje vreme koje procesor provede čekajući da instrukcije budu donešene iz memorije; naredne instrukcije se unapred dovlače u *prefetch* red,
- *speculative execution* - unapred izvršavanje instrukcija koje možda nisu potrebne,
- *multithreading* - deljenje jednog procesora između više niti na način koji minimizira vreme potrebno za promenu konteksta,
- *short vector* skup instrukcija - ubrzava *floating-point* izračunavanja presudna za 3D performanse.

Superscalar arhitektura

Superskalarna arhitektura implementira *instruction-level* paralelizam pomoću redundantnih funkcionalnih jedinica unutar jednog procesora. Princip rada je sledeći: sekvencijalni niz instrukcija se transformiše u dinamički paralelni niz, kako bi se više instrukcija izvršavalo simultano. U toku faze dohvaćanja instrukcija, dohvata se više instrukcija istovremeno. Dekodiranje za svaku instrukciju obuhvata utvrđivanje operacije koju treba izvršiti, utvrđivanje lokacije operanada i lokacije za smeštanje rezultata. U toku sledeće, *issue* faze, poseban hardver, zadužen za raspoređivanje instrukcija na osnovu analize zavisnosti koje postoje u programu, identifikuje među datim instrukcijama one koje mogu početi izvršavanje i šalje ih u odgovarajuće slobodne funkcionalne jedinice. Na kraju izvršavanja se rezultati operacija skupljaju i preuređuju.

Branch prediction

Za postizanje visokih performansi u *pipeline* procesorima, ključno je predviđanje kako će se program ponašati kod instrukcija uslovnog skoka. Metode predviđanja se svrstavaju u statičke i dinamičke. U slučaju statičkih metoda, predviđanje za svaki skok je uvek isto za vreme kompletnog izvršavanja programa (uvek se predviđa da ima skoka – *branch taken* ili uvek se predviđa da nema skoka – *branch not taken*). Ako je predviđanje bilo dobro nastavlja se izvršavanje bez izgubljenog vremena; inače se mora isprati i restartovati *pipeline*. Kod dinamičkih metoda, predviđanje se menja u toku izvršavanja programa, na osnovu prethodnog ponašanja programa. Ako se koristi tehnika sa baferom predviđanja (*branch prediction buffer*), kada se u izvršavanju nađe na instrukciju skoka, proverava se zapis za tu instrukciju da se utvrdi da li se ona i pre

javljala, i ako jeste da se vidi da li je bilo skoka ili ne. Pomoću algoritma sa dva (ili sa četiri) stanja i ovih zapisa, odlučuje se da li treba izvršiti skok ili ne i pri tome se ažuriraju zapisi. Nedostatak ove tehnike je što se odlučivanje sprovodi tek kad se utvrdi da je reč o instrukciji grananja. Tehnika koja koristi asocijativni keš za predikciju skokova podrazumeva da se već u *fetch* stepenu, prilikom dohvatanja neke instrukcije, adresa te instrukcije vodi na ulaze keša za predikciju skokova gde se vrši provera da li se ta adresa nalazi u kešu. Ako se nalazi, u pitanju je instrukcija skoka koja je već bila izvršavana pa za nju postoji predviđanje o ishodu skoka i predviđena vrednost za programski brojač s koje treba očitati sledeću instrukciju. Ako se ne nalazi, instrukcija ili nije instrukcija grananja ili je u pitanju instrukcija grananja koja nije prethodno izvršavana. U stepenu izvršavanja se ažurira keš memorija i proverava ispravnost instrukcija koje su usledile iza instrukcije skoka.

Prefetching

Moderni procesori su mnogo brži od operativne memorije gde su smešteni programi, što znači da se programske instrukcije ne mogu čitati dovoljno brzo da bi procesor bio stalno zaposlen. Dodavanje keš memorije može obezbediti brži pristup potrebnim instrukcijama. U ovom slučaju problem predstavljaju situacije kada se tražena instrukcija ne nalazi u kešu (*miss rate*) pa procesor mora da čeka da ona bude dovučena iz glavne memorije što uzrokuje veliko kašnjenje u izvršavanju. Rešenje može da bude unapred dovlačenje (*prefetching*) instrukcija iz glavne memorije u keš, pre nego što su potrebne procesoru. Tako će se pročitanim instrukcijama, smeštenim u keš memoriji, moći brže pristupiti kada budu bile potrebne. Instrukcije se sekvencijalno unapred dovlače (jer se programi generalno izvršavaju sekvencijalno) ali *prefetching* može biti i deo metode predviđanja skokova (*branch prediction*) kada procesor pokušava da predvidi rezultat izvršavanja i unapred pročita pravu instrukciju.

Speculative execution

Ova metoda optimizuje performanse tako što omogućava procesoru da špekuliše o svojstvima instrukcija i da ranije započne izvršavanje drugih instrukcija koje zavise od datog predviđanja. Mora da postoji mehanizam za proveru da li je nagađanje bilo ispravno i mehanizam za poništavanje efekata nastalih kao posledica špekulativnog izvršavanja kada predviđanje nije bilo ispravno. Prednosti ranog izvršavanja moraju biti dovoljne da se kompenzuju situacije kada je predviđanje bilo pogrešno.

Primer su *pipeline* procesori koji mogu da predvide (*branch prediction*) ishod instrukcije uslovnog skoka i odmah započnu izvršavanje (*speculative execution*) instrukcija koje slede posle grananja. Rezultati ovakvog izvršavanja se baferišu. Ako se ispostavi da je nagađanje bilo tačno, onda se dozvoljava upis sadržaja bafera u registre ili u memoriju; inače se baferi ispiraju (*flush*) i nastavlja se sa izvršavanjem ispravne sekvence instrukcija. Špekulativno izvršavanje kod *pipeline* procesora je relativno jeftino jer bi inače stepeni ostali neiskorišćeni dok se ne izračuna rezultat instrukcije skoka.

Multithreading

Jedna od strategija za postizanje boljih performansi je *thread level parallelism* - izvršavanje više programa ili niti u paraleli. Ova vrsta paralelizma se postiže pomoću dve tehnike: *multiprocessing* (MP; svaki procesor u sistemu izvršava različitu nit) i *multithreading* (MT; niti dele resurse u okviru jednog procesora, replicira se samo deo hardvera - kao što je registarski fajl, programski brojač). Jedna vrsta MT-a je *block multithreading* koja podrazumeva da procesor ne čeka da potrebni podatak stigne iz glavne memorije da bi nastavio izvršavanje date niti, već se odmah prebacuje na izvršavanje druge spremne niti. Drugi tip je *simultaneous multithreading* gde se instrukcije različitih programa/niti izvršavaju simultano unutar jednog ciklusa signala takta, kod procesora superskalarne arhitekture. Prednost tehnike MT nad MP je repliciranje samo dela procesora umesto čitavog procesora. Mana je što je hardverska podrška za MT vidljivija softveru pa operativni sistem mora da doživi velike promene za podršku MT-a.

Scalability

Proširivost (*scalability*) je poželjna osobina sistema koja ukazuje na njegovu sposobnost da izađe na kraj sa rastućom količinom posla ili da se sistem može uvećati. Za sistem čije se performanse posle dodavanja hardvera povećavaju proporcionalno dodatom kapacitetu se kaže da je proširiv (*scalable*). Metode dodavanja resursa se mogu svrstati u dve kategorije: vertikalno skaliranje (*scale up*; dodavanje resursa jednom čvoru u sistemu, tipično znači dodavanje procesora ili memorije pojedinom računaru) i horizontalno skaliranje (*scale out*; dodavanje više čvorova u sistemu, npr. dodavanje računara u klaster). Kako cene računara opadaju a njihove performanse rastu, jeftini sistemi (npr. hiljadu malih računara konfigurisanih u klaster) se mogu koristiti za zahtevna izračunavanja za koje su se nekad koristili superračunari. Ali veći broj računara znači i veću složenost upravljanja, kompleksniji programski model, problem protoka i kašnjenja između čvorova.

Odnos između računarske arhitekture i multimedijalnih aplikacija, *short vector* skup instrukcija, *Streaming extensions*, *Altivec*

Prvi vektorski procesori su se koristili samo kod superračunara za naučna istraživanja i kriptografske aplikacije. Od 1990-tih godina podrška za *data* paralelizam je postala značajna i kod računara opšte namene jer se time optimizuje izvršavanje multimedijalnih aplikacija. Česta operacija kod mnogih tih aplikacija je dodavanje (oduzimanje) iste vrednosti velikom broju podataka, npr. menjanje nivoa osvetljenja slike. Poboljšanje koje se postiže vektorskim procesuiranjem je dohvaćanje grupe piksela iz glavne memorije što zahteva mnogo manje vremena nego kad se oni dohvataju pojedinačno. Operacija sabiranja (oduzimanja) se zatim izvršava nad svim dohvaćenim podacima istovremeno. Većina današnjih procesora opšte namene uključuje instrukcije za procesuiranje vektora sa dve, tri ili četiri dimenzije (za podršku 3D grafike), dok su superračunari iz 1980-tih godina izvršavali operacije nad dugačkim vektorima od nekoliko stotina ili hiljada

brojeva. Zato da bi se razlikovale od starijih, nove SIMD arhitekture se označavaju kao *short-vector* arhitekture.

Neke od ranih SIMD specifikacija, kao što je *Intel*-ov *MMX instruction set*, su obuhvatale samo instrukcije sa celim brojevima - problem je što mnoge aplikacije koje imaju koristi od DLP rade sa *floating-point* veličinama. Drugi problem je što MMX skup instrukcija deli registarski fajl sa *floating-point* stekom, pa procesor ne može da radi sa *floating-point* i vektorskim podacima u isto vreme (postoje skupe instrukcije za promenu konteksta). Naslednik MMX-a u *Intel*-u je *Streaming SIMD Extensions* (SSE), dodatak za x86 arhitekturu koji sadrži 70 novih instrukcija. Uvedene su *floating-point* instrukcije *scalar* i *packed* tipa, kao i osam 128-bitnih registara (XMM0 do XMM7) za smeštanje vektorskih podataka.

Mnoge mikroprocesorske kompanije su razvile *short vector multimedia* ekstenzije u SIMD stilu, a jedna od naprednijih je *Apple*-ov *AltiVec*, skup instrukcija za celobrojne veličine i veličine u pokretnom zarezu implementiran na *PowerPC* arhitekturi. Ovaj dodatak za razliku od SSE, uvodi poseban tip podataka - *RGB pixel*, kao i mnogo kompletnije i kompleksnije instrukcije nego SSE, a obezbeđuje i tridesetdva 128-bitna vektorska registra.

3. ONTOLOGIJA ARHITEKTURE I ORGANIZACIJE RAČUNARA

Osnovne koncepte domena čine klase u korenu hijerarhije:

- ArhitekturaMemorije,
- ArhitekturaProcesora,
- ArhitekturaRacunara,
- ArhitekturaUlaznoIzlaznogSistema,
- Bit,
- ElementArhitectureMemorije,
- ElementArhitectureProcesora,
- ElementArhitectureUISistema,
- InterfejsIKomunikacija,
- Modul,
- NivoParalelizma,
- OrganizacijaRacunara,
- PipelineHazard,
- Podatak,
- Program i
- Signal.

ArhitekturaMemorije je definisana kao potklasa klase *Thing* koja može da ima element arhitekture memorije tj. instance klase *ArhitekturaMemorije* ako učestvuju u nekoj relaciji preko *ima* svojstva, to mora biti sa instancama klase *ElementArhitectureMemorije*. Na sličan način su definisane i klase *ArhitekturaProcesora* i *ArhitekturaUlaznoIzlaznogSistema* kao skupovi instanci koje preko *ima* svojstva mogu biti u vezi jedino sa instancama klase *ElementArhitectureProcesora* i *ElementArhitectureUISistema* respektivno. Klasa *ArhitekturaRacunara* je definisana kao unija sledećih disjoint klasa:

- ArhitekturaMemorije
- ArhitekturaProcesora
- ArhitekturaUlaznoIzlaznogSistema.

Vrste odnosno potklase klase *ArhitekturaRacunara* su razdvojene klase *VonNeumannArhitektura* (arhitektura računara koja ima tačno jednu memoriju) i *HarvardArhitektura* (arhitektura računara koja ima najmanje dve memorije), zatim razdvojene klase *CISCArhitektura*, *RISCArhitektura* (može da ima samo standardne instrukcije) i *VLIWArhitektura*, i potklase *SISDArhitektura*, *SIMDArhitektura*, *MISDArhitektura*, *MIMDArhitektura* koje su takođe međusobno disjunktne kategorije. *MIMDArhitektura* se deli na *MultiprocesorskiSistem* i *MultiracunarskiSistem*. *MultiprocesorskiSistem* može biti *NUMASistem* ili *UMASistem*.

Bit je deo reči – instance klase *Bit* preko svojstva *jeDeo* mogu biti povezane samo sa instancama klase *MemorijskaRec*. Navedene su neke vrste bitova:

- BitProgramskeStatusneReci
- BitStatusnogRegistraKontrolera i
- BitUpravljackogRegistraKontrolera.

BitProgramskeStatusneReci može biti *BitStatusnogKaraktera* (C, In, IV, L0, L1, N, NN, U, V, Z) ili *BitUpravljackogKaraktera* (E, I, P, T). Jedna potklasa klase *BitStatusnogRegistraKontrolera* je *Ready* a potklase *BitUpravljackogRegistraKontrolera* - *Burst, Direction, Enable, Mem, Start* i *Ui* koji određuje režim rada kontrolera periferije.

ElementArhitektureMemorije mogu biti sledeći pojmovi:

- *AdresniProstor,*
- *Memorija,*
- *MemorijskaLokacija,*
- *MemorijskaRec,*
- *MemorijskaTehnologija,*
- *MeraPerformansiMemorije* i
- *Stek.*

AdresniProstor koji sadrži adrese iz memorije ima podvrste: *RealniAdresniProstor* (samo adrese iz *OperativnaMemorija*), *UlaznoIzlazniAdresniProstor* (adrese iz kontrolera periferije) i *VirtuelniAdresniProstor* (adrese iz *VirtuelnaMemorija*).

Memorijsku hijerarhiju čini *Memorija* u korenu, njene potklase su *PrimarnaMemorija* (direktno dostupna procesoru), *SekundarnaMemorija* i *TernarnaMemorija* (za arhiviranje velike količine podataka kojima se retko pristupa; potklase *BibliotekaTraka* i *OptickaBiblioteka*). *PrimarnaMemorija* se deli na *Kes*, *OperativnaMemorija* i *RegistarProcesora*. *Kes* se implementira pomoću SRAM tehnologije, ima organizaciju sa direktnim, asocijativnim ili setasocijativnim prelikavanjem, koristi neku tehniku upisa u kešu kao i tehniku zamene. Ima sledeće potklase:

- *KesDrugogNivoa,*
- *KesSaPodblokovima,*
- *KesZrtava,*
- *NeblokirajuciKes,*
- *TLBJedinica* i
- *VirtuelniKes.*

TLBJedinica je deo procesora i ubrzava preslikavanje adresa iz virtuelnog adresnog prostora. *RealniKes* sadrži adrese iz realnog adresnog prostora dok *VirtuelniKes* ima adrese iz virtuelnog adresnog prostora (definiše se kao komplement klase *RealniKes*). *OperativnaMemorija* je modul koji se implementira pomoću DRAM tehnologije i sadrži minimalno dve memorijske lokacije. *RegistarProcesora* ima dve nadklase – *PrimarnaMemorija* i *ElementArhitektureProcesora*. Može biti programski dostupni *AdresabilniRegistarProcesora* (*AdresniRegistar AR*, *Akumulator A*, *BazniRegistar BR*, *IndeksniRegistar XR*, *ProgramskaStatusnaRec PSW*, *ProgramskiBrojac PC*, *RegistarMaske IMR*, *RegistarOpsteNamene GPR*, *RegistarPodataka DR*, *UkazivacNaListuArgumenata AP*, *UkazivacNaOkvirSteka FP*, *UkazivacNaTabeluIVT IVTP*, *UkazivacNaVrhSteka SP*) ili programski nedostupni *InterniRegistarProcesora* (*AdresniRegistarMemorije MAR*, *InterniBafer*, *PrihvatniRegistarInstrukcije IR*, *PrihvatniRegistarPodatkaMemorije MDR*). *SekundarnaMemorija* obuhvata *HardDisk* (implementiran magnetnom tehnologijom), *OptickiDisk* (implementiran optičkom tehnologijom; ima potklase *CD* i *DVD*), *Traka* (implementirana magnetnom tehnologijom), *USBStick* (implementiran flash tehnologijom) i *VirtuelnaMemorija* koja

povećava realni adresni prostor i ima jednu od organizacija (straničnu, segmentnu ili segmentnostraničnu).

MemorijskaLokacija je element arhitekture memorije koji sadrži memorijsku reč. *MemorijskaRec* može biti ili *Podatak* ili *Instrukcija*, predstavlja uniju pomenutih koncepata. *MemorijskaTehnologija* se može podeliti na *MagnetnaTehnologija*, *OptickaTehnologija* i *PoluprovodnickaTehnologija* (*DRAM*, *Flash* i *SRAM*). *MeraPerformansiMemorije* obuhvata potklase *MeraPerformansiKesa* (opisuje performanse keša) i *MeraPerformansiOperativneMemorije* (opisuje performanse operativne memorije). *GubitakVremenaPriPromasaju* kao mera performansi keša se poboljšava neblokirajućim kešom, kešom drugog nivoa ili sa podblokovima. *ProcenatPromasaja* keša se poboljšava kešom žrtava, a *VremePristupaPriPogotku* virtuelnim kešom, organizacijom sa direktnim ili setasocijativnim preslikavanjem. *MeraPerformansiOperativneMemorije* su *PropusnaMoc*, *VelicinaAdresnogProstora* (poboljšava se virtuelnom memorijom) i *VremePristupa* (poboljšava se kešom ili preklapanjem pristupa memorijskim modulima). *Stek* struktura je deo jedne operativne memorije i može biti aritmetički ili upravljački.

ElementArhitektureProcesora mogu biti sledeći pojmovi:

- *AritmetickoLogickaJedinica*, *ALU*
- *FazaIzvršavanjaInstrukcije*,
- *FormatInstrukcije*,
- *Instrukcija*,
- *LokacijaOperanda*,
- *MehanizamPrekida*,
- *NacinAdresiranja*,
- *OperacionaJedinicaProcesora*,
- *PoljeInstrukcije*,
- *RegistarProcesora*,
- *RegistarskiFajl*,
- *TipPodataka* i
- *UpravljackaJedinicaProcesora*.

AritmetickoLogickaJedinica ALU je deo operacione jedinice procesora.

FazaIzvršavanjaInstrukcije može biti :

FazaCitanjaInstrukcije (proverava *SignalDuzineInstrukcije*),

FazaFormiranjaAdreseICitanjaOperanda (proverava *SignalNacinaAdresiranja*),

FazaIzvršavanjaOperacije (proverava *SignalOperacije*),

FazaOpsluzivanjaPrekida (proverava *SignalPrekida*).

FormatInstrukcije se deli na *PromenljiviFormat* (sadrži najmanje jedno polje instrukcije),

NulaadresniFormat, *JednoadresniFormat* (sadrži tačno dva polja instrukcije),

DvoadresniFormat (sadrži tačno tri polja instrukcije) i *TroadresniFormat* (sadrži tačno

četiri polja instrukcije). Instrukcije se mogu grupisati u standardne i nestandardne kojih

nema kod RISC arhitektura. Standardne instrukcije su *AritmetickaInstrukcija* (*ADD*,

DEC, *DIVS*, *DIVU*, *INC*, *MULS*, *MULU*, *SUB*), *InstrukcijaPomeranjaIliRotiranja* (*ASH*,

LSH, *ROT*, *ROTC*), *InstrukcijaPrenosa* (*IN*, *LOAD*, *MOVE*, *OUT*, *STORE*,

StekInstrukcija - *POP* i *PUSH*), *InstrukcijaSkoka* (*InstrukcijaUslovnogSkoka*, *INT*, *JMP*,

JSR, *RTI*, *RTS*), *LogickaInstrukcija* (*AND*, *NOT*, *OR*, *XOR*) i *MesovitaInstukcija* (*EDGD*,

EDGE, *INTD*, *INTE*, *NOP*, *TRPD*, *TRPE*, *VARD*, *VARE*).

LokacijaOperanda je unija klasa *MemorijskaLokacija* i *AdresabilniRegistarProcesora* i *NeposrednaVelicinaUInstrukciji*.

MehanizamPrekida obuhvata sledeće pojmove:

- *GnezdenjePrekida*,
- *MaskiranjePrekida*,
- *OpsluzivanjeZahtevaZaPrekid*,
- *PovratakIzPrekidneRutine*,
- *Prekid i*
- *PrihvatanjeZahtevaZaPrekid*.

GnezdenjePrekida prekida izvršavanje jedne prekidne rutine a izaziva izvršavanje druge prekidne rutine. *MaskiranjePrekida* zabranjuje prihvatanje maskirajućeg prekida (svih maskirajućih prekida pomoću bita *I*, ili selektivno pomoću registra maske *IMR*). *OpsluzivanjeZahtevaZaPrekid* obuhvata stavljanje na stek PSW-a i PC-a i utvrđivanje adrese prekidne rutine pomoću ukazivača na IVT tabelu. *PovratakIzPrekidneRutine* je deo prekidne rutine koji izaziva izvršavanje *RTI* instrukcije. *Prekid* prekida izvršavanje programa i izaziva izvršavanje prekidne rutine. *UnutrasnjiPrekid* generiše *Procesor*

- *PrekidZbogInstrukcijeINT*, prioritet 1,
- *PrekidZbogNekorektnostiUIzvršavanjuInstrukcije*, prioritet 2,
- *PrekidPosleSvakeInstrukcije*, prioritet 5,

za razliku od spoljašnjih prekida:

- *MaskirajuciPrekid*, prioritet 4,
- *NemaskirajuciPrekid*, prioritet 3.

MaskirajuciPrekid se selektivno maskira pomoću registra maske, a generiše ga *KontrolerPeriferije*. *NemaskirajuciPrekid* se ne može selektivno maskirati. *PrihvatanjeZahtevaZaPrekid* znači prelazak na *OpsluzivanjeZahtevaZaPrekid*.

NacinAdresiranja je *ElementArhitectureProcesora* koji se koristi za pristup lokaciji operanda (bazno adresiranje pomoću baznog registra specificira memorijsku lokaciju; bazno-indeksno pomoću baznog, indeksnog registra ili registra opšte namene specificira memorijsku lokaciju; indeksno sa pomerajem pomoću indeksnog registra i neposredne veličine u instrukciji specificira memorijsku lokaciju; memorijski indirektno adresiranje pomoću mem. lokacije specificira lokaciju operanda u memoriji; registarsko indirektno koristi adresni ili registar opšte namene za specificiranje memorijske lokacije operanda; memorijski direktno određuje lokaciju operanda u memoriji; neposredno određuje lokaciju operanda u polju instrukcije – *NeposrednaVelicinaUInstrukciji*; postdekrement i preinkrement koriste adresni ili registar opšte namene za određivanje mem. lokacije; registarsko direktno pomoću registra podataka ili registra opšte namene specificira lokaciju operanda u registru procesora; registarsko indirektno sa pomerajem koristi polje instrukcije *NeposrednaVelicinaUInstrukciji* i registar opšte namene za određivanje mem. lokacije operanda; relativno adresiranje pomoću programskog brojača specificira mem. lokaciju operanda).

OperacionaJedinicaProcesora je deo procesora koji izvršava instrukcije i generiše signale logičkog uslova. Ima određenu organizaciju i sadrži registre

UkazivacNaVrhSteka, PrihvatniRegistarInstrukcije, ProgramskaStatusnaRec, ProgramskiBrojac, AdresniRegistarMemorije, PrihvatniRegistarPodatakaMemorije.

PoljeInstrukcije se definiše kao deo instrukcije - *PoljeOperacijaITipPodataka, PoljeIzvorisniIOdredisniOperandi* ili *NeposrednaVelicinaUInstrukciji*.

RegistarskiFajl je deo operacione jedinice procesora koji sadrži barem dva registra.

TipPodataka se klasifikuje u *AlfaNumerickiNiz, CelobrojnaVelicina* (sa ili bez znaka, promenljive ili fiksne dužine), *NumerickiNiz* (pakovanog ili nepakovanog formata) i *VelicinaUPokretnomZarezu*.

UpravljackaJedinicaProcesora je deo procesora koji koristi signale logičkog uslova i generiše upravljačke signale operacione i upravljačke jedinice procesora. Ima organizaciju *OrganizacijaUpravljackeJediniceProcesora*.

U elemente arhitekture ulazno/izlaznog sistema se ubrajaju *KombinacionaMrezaZaPrepoznavanjeCiklusaCitanjaIliUpisa, KontrolerPeriferije, Periferija* i *ProgramskiDostupanRegistarKontrolera*. *KontrolerPeriferije* mora da sadrži kombinacionu mrežu za prepoznavanje ciklusa i registre *RegistarBrojaUlaza, RegistarPodatakaKontrolera, StatusniRegistar, UpravljackiRegistar*. Startuje se i zaustavlja postavljanjem bita *Start*. Kontroler može biti bez direktnog pristupa operativnoj memoriji (nonDMA, prenosi podatke između registra podataka kontrolera i periferije) i sa direktnim pristupom (DMA, mora da ima adresni registar i registar za veličinu bloka podataka; prenosi podatke između registra podataka kontrolera i memorije ili periferije ili nonDMA kontrolera). Potklase klase *Periferija* su *UlaznaPeriferija* (šalje podatke u kontroler periferije; tastatura, miš, skener) i *IzlaznaPeriferija* (prima podatke iz kontrolera periferije; monitor, zvučnici, štampač). *AdresniRegistarKontrolera, RegistarBrojaUlaza, RegistarPodatakaKontrolera, RegistarVelicineBlokaPodataka, StatusniRegistar* (između ostalih bitova mora da sadrži *Ready* bit) i *UpravljackiRegistar* (između ostalih bitova mora da sadrži *Start* bit) su specifičnije klase nadklase *ProgramskiDostupanRegistarKontrolera*.

InterfejsIKomunikacija obuhvata sledeće pojmove:

- *Arbitracija,*
- *CiklusNaMagistrali,*
- *Linija,*
- *Magistrala i*
- *UcesnikCiklusaNaMagistrali.*

Arbitracija može biti centralizovana i decentralizovana, sa pamćenjem ili sa praćenjem zahteva. Kao potklase klase *CiklusNaMagistrali* su navedene vrste ciklusa na magistrali:

- *CiklusCitanja,*
- *CiklusUpisa i*
- *CiklusPrihvatanjaBrojaUlaza*

koje postoje kod magistrale sa atomskim ciklusima i

- *CiklusSlanjaZahtevaZaCitanje,*
- *CiklusSlanjaZahtevaZaUpis,*
- *CiklusSlanjaZahtevaZaDobijanjeBrojaUlaza i*
- *CiklusVracanjaPodatka*

koje postoje kod magistrale sa podeljenim ciklusima.

Od linija su date linije magistrale i linije U/I uređaja (*ACKLinija*, *DATALinija*, *IBFLinija*, *OBFLinija*, *STARTLinija*, *STBLinija*). *LinijaMagistrale* se deli na *AdresnaLinija*, *LinijaPodataka* i *UpravljackaLinijaMagistrale* (*ACKBUS*, *DABUS*, *FCBUS*, *MIOBUS*, *RDBUS*, *WRBUS*). *Magistrala* može biti *LokalnaMagistrala* koja je deo procesora ili *SistemskaMagistrala* koja povezuje bar dva modula u sistemu i obavezno ima adresnu liniju, liniju podataka, i linije za prenos upravljačkih signala čitanja i upisa. Ako sadrži *FCBUS* liniju u pitanju je *AsinhronaMagistrala*, inače je *SinhronaMagistrala* koja koristi signal takta *MCKL*. *MagistralaSaAtomskimCiklusima* može da ima ciklus čitanja, upisa i prihvatanja broja ulaza, dok njena disjoint klasa *MagistralaSaPodeljenimCiklusima* može da ima ciklus slanja zahteva za čitanje, za upis ili za dobijanje broja ulaza i ciklus vraćanja podatka (mora da sadrži linije *DABUS* i *ACKBUS*). *UcesnikCiklusaNaMagistrali* je ili *Gazda* (započinje ciklus na magistrali) ili *Sluga* (prihvata ciklus na magistrali), a to može biti procesor, operativna memorija i ulazno-izlazni uređaj.

Moduli računara su procesor, operativna memorija, ulazno/izlazni uređaj i arbitrator uređaj (izvršava centralizovanu arbitraciju). *Procesor* može biti paralelni ako ima DLP (*VectorProcesor*), ILP (*PipelinedProcesor* -pipeline organizacija, *SuperscalarProcesor*) ili TLP (*MulticoreProcesor*, *MultithreadedProcesor*) nivo paralelizma, inače je serijski. *UIUredjaj* mora da ima periferiju i kontroler periferije.

U nivoe paralelizma se ubrajaju *DataLevelParalelizam*, *InstructionLevelParalelizam* i *ThreadLevelParalelizam*.

OrganizacijaRacunara je podeljena na *OrganizacijaMemorije*, *OrganizacijaProcesora* i *OrganizacijaUlaznoIzlaznogSistema*. *OrganizacijaMemorije* ima disjoint potklase: *OrganizacijaSaAsocijativnimPreslikavanjem*, *OrganizacijaSaDirektnimPreslikavanjem* i *OrganizacijaSaSetAsocijativnimPreslikavanjem*; takođe disjoint *SegmentnaOrganizacija*, *StranicnaOrganizacija* i *SegmentnoStranicnaOrganizacija*. Pored ovih, još postoje potklase *PreklapanjePristupaMemorijskimModulima* koje koristi magistralu sa podeljenim ciklusima i *RasporedAdresaPoMemModulima* (*MesovitRaspored*, *SusedneAdreseUIstomModulu*, *SusedneAdreseUSusednimModulima*), *TehnikaZamene* definisana nabravanjem njenih instanci (*LRU*, *FIFO*, *SlucajniIzbor*) i *TehnikaUpisaUKesu* (u slučaju pogotka može biti *UpisiSkroz* ili *UpisiNazad*; u slučaju promašaja je ili *NeDovlaciBlok* ili *DovuciBlok*). *OrganizacijaProcesora* se deli na organizaciju operacione jedinice i organizaciju upravljačke jedinice procesora. *OrganizacijaOperacioneJediniceProcesora* može biti sa jednom, dve, tri magistrale ili sa direktnim vezama, *PipelineOrganizacija* (*StatickiPipeline* –asinhroni i sinhroni koji koristi signal takta, *DinamickiPipeline*), *MetodZaResavanjePipelineHazarda* (za rešavanje upravljačkog hazarda služi *PredvidjanjeSkoka*, *UpotrebaPosebnogHardveraUFetchJedinici*, *ZakasnjenSkok*, za rešavanja hazarda podataka *Prosledjivanje* i *ZakasnjenoPunjenje*; dok se metoda *ZaustavljanjePipelinea* može koristiti za rešavanje svih vrsta hazarda). *Organizacija upravljačke jedinice procesora* može biti mikroprogramska (sa jednim ili dva tipa mikroinstrukcija; sa horizontalnim, vertikalnim ili mešovitim formatom mikroinstrukcija) ili ožičena. *OrganizacijaUlaznoIzlaznogSistema* ima podkategorije:

- *PovezivanjeKontroleraIPeriferije*, npr. *ParalelniInterfejs*
- *ProgramiraniUlazIzlaz* i
- *RezimRadaDMA*.

Programirani ulaz/izlaz može biti čitanjem statusnog registra (povremeno ili u petlji) generisanjem prekida. Režimi rada DMA kontrolera su paketski režim, režim prenosa ka višim ili nižim lokacijama, režim prenosa memorija-memorija i režim slanja iste vrednosti.

PipelineHazard postoji kod pipeline organizacije (*HazardPodataka* – RAW, WAR, WAW, zatim *StrukturalniHazard* i *UpravljackiHazard*).

Program (sadrži jednu ili više instrukcija) ima podkategorije potprograma, prekidne rutine (program za obradu prekida), program u asemblerskom jeziku (simbolička reprezentacija programa u mašinskom jeziku), program u mašinskom jeziku (objektni i izvršni program) i specijalni program (assembler – od programa u asemblerskom pravi program u mašinskom jeziku, povezič – od objektnog programa pravi izvršni program, punilac – učitava u operativnu memoriju izvršni program).

Od vrsta signala su navedeni *SignalLogickogUslova* (signal dužine instrukcije, načina adresiranja, operacije, prekida ili rezultata operacije), *SignalTakta* (*CLK* i *MCLK*) i upravljački signal operacione i upravljački signal upravljačke jedinice procesora.

Za pravljenje restrikcija prilikom definisanja klasa su korišćeni sledeći objektni propterti:

- generise,
- seImplementiraPomocu,
- saljePodatkeU,
- postojiKod,
- seStartujeIliZaustavljaPomocu,
- jeSimbolickaReprezentacija,
- prenosiSadrzajRegistraKontroleraPeriferijeU,
- prenosiSadrzajAkumulatoraU,
- resava,
- uzimaSaSteka,
- realizujePovratakIz,
- odObjektnogProgramaPravi,
- jeFormata,
- utvrdjujeAdresuPrekidneRutinePomocu,
- ubrzavaPreslikavanjeAdresaIz,
- ukazujeNaAdresibilnuLokacijuNa,
- opisujePerformanse,
- proverava,
- sePoboljsavaSa,
- primaPodatkeIz,
- povecava,
- programskimPutemIzaziva,
- zabranjujePrihvatanje,
- izvrsava,
- biraSledecegGazdu,
- izazivaIzvršavanje,
- realizujeSkokNa,
- prenosiSadrzajVrhaStekaU,
- odProgramaUAssemblerskomJezikuPravi,
- seKoristiZaPristup,
- znaciPrelazakNa,
- služiZaPrihvatanje,
- učitavaUOperativnuMemoriju.

jeDeo tranzitivni properti čiji je inverzni properti *sadrzi*. Podproperty *jeDeoOperativneMemorije* ima opseg *Memorija*. Properti *sadrzi* ima podpropertyje: *sadrziLokalnuMagistralu* (opseg je klasa *Magistrala*), *sadrziLiniju* (opseg je klasa *Linija*), *sadrziMemorijskuLokaciju* (opseg *MemorijskaLokacija*), *sadrziAdresuOd*, *sadrziAdreseIz*, *sadrziBit* (opseg *Bit*), *sadrziInstrukciju* (opseg *Instrukcija*), *sadrziPolje* (opseg *PoljeInstrukcije*) i *sadrziRegistar*. Funkcionalan properti *seSelektivnoMaskiraPomocu* ima domen *MaskirajuciPrekid* i opseg *RegistarMaske*. Properti *koristi* ima podpropertyje *koristiStek* (opseg *Stek*), *koristiMagistralu* (opseg *Magistrala*), *koristiRasporedAdresa* (opseg *RasporedAdresaPoMemModulima*), *koristiSignal* (opseg *Singal*), *koristiTehniku*, *koristiRegistar* (domen je unija klasa registarskog direktnog i indirektnog, baznog, indeksnog sa pomerajem, registarskog indirektnog sa pomerajem, baznoindeksnog, postdekrement, preinkrement, relativnog adresiranja i klase selektivnog maskiranja prekida, dok je opseg *AdresabilniRegistarProcesora*) i *koristiInstrukciju* (opseg *Instrukcija*). Domen propertyja *specificiraLokacijuOperanada* je *NacinAdresiranja*, a opseg je unija klasa *NeposrednaVelicinaUInstrukciji*, *MemorijskaLokacija* i *AdresabilniRegistarProcesora*. Properti *prenosiOperandU* ima za domen klasu *Instrukcija*, a *povezujeModule* ima domen *Magistrala*. Objektni properti *ima* sadrži podpropertyje *imaMemoriju* (opseg *Memorija*), *imaInstrukciju* (opseg *Instrukcija*) i *imaOrganizaciju*. *prihvataCiklus* ima domen *Sluga* a opseg *CiklusNaMagistrali*. Properti *zapocinjeCiklus* ima ograničen domen klasom *Gazda*, a opseg klasom *CiklusNaMagistrali*. Klasa *OpsluzivanjeZahtevaZaPrekid* je domen propertyja *stavljaNaStek*, dok je opseg *AdresabilniRegistarProcesora*. Properti *pristupaMemoriji* ima opseg *Memorija*. *KontrolerPeriferije* je domen za properti *prenosiPodatkeIzmedjuRegistraPodatakaKontroleraI*, a opseg je klasa *Periferija*. Za properti *prekidaIzvršavanje* je definisan domen *GnezdenjePrekida* i opseg *Prekid*. Klasa *Asembler* je domen za *odProgramaUAsemblerskomJezikuPravi*, a opseg vrednosti je *ProgramUMasinskomJeziku*. *Ui* klasa čini domen propertyja *odredjujeRezimRada*, dok je njegov opseg *KontrolerPeriferije*.

Od datatype propertyja postoje jedino funkcionalni properti *imaPrioritetPrekida* sa domenom *Prekid* i opsegom koga čini niz celih brojeva 1,2,3,4 i 5, i takođe funkcionalan properti *imaVelicinuUBitovima* sa domenom *MemorijskaRec* i opsegom vrednosti 4, 8, 16, 32 i 64.

Što se tiče annotation propertyja, korišćeni su comment i label properti za opisivanje odnosno navođenje alternativnog imena za dati pojam. Za opisivanje ontologije su upotrebljeni creator, date i title propertyji iz uvezene Dublin Core ontologije.

ZAKLJUČAK

Navedena ontologija predstavlja strukturirani rečnik sa mašinski-razumljivim definicijama osnovnih pojmova iz domena arhitekture i organizacije računara, i definicijama nekih od relacija koje postoje između tih pojmova. Eksperti iz domena ontologije mogu da je koriste za deljenje i označavanje informacija iz njihovih oblasti. Eksplicitne specifikacije domenskog znanja mogu biti pogodne za nove korisnike koji trebaju da nauče značenja osnovnih termina iz domena. Jasna tvrđenja ontologije se za razliku od tvrđenja iz programskog kôda mogu lako pronaći, razumeti i promeniti ako se izmeni znanje iz date oblasti.

Softverski agenti i neke aplikacije takođe mogu da imaju koristi od ove ontologije. Na primer ako bi više *Web* sajtova sa informacijama o računarima publikovali termine iz iste ontologije, elektronski agenti u potrazi za određenim podacima bi mogli izdvojiti i zatim agregirati relevantne informacije da bi odgovorili na postavljene upite. Neka aplikacija koja u osnovi ima datu ontologiju bi mogla da analizira listu računarskog inventara i da sugeriše koje kategorije PC komponenata treba proširiti.

Nadogradnja ontologije može da podrazumeva uvođenje podrške za različite jezike (mapiranje), definisanje novih koncepata ili detaljniji opis postojećih pojmova. Može se kreirati baza znanja definisanjem najspecifičnijih koncepata - instanci i popunjavanjem konkretnih vrednosti za njihove svojstva. Odlučivanje da li određeni koncept treba da bude klasa ili instanca zavisi od potencijalnih namena ontologije. U ovom slučaju je izabrano opštije rešenje pa su svi koncepti klase, izuzev par individua, i naravno svojstva. Zato se ova generalna ontologija može ponovno upotrebiti i iskoristiti za opisivanje specifičnijeg domena interesovanja.

Literatura

<http://www.w3.org>
<http://www.wikipedia.org>
http://www.fer.hr/_download/repository/Marin_Prcela_SW.pdf
<http://obitko.com/tutorials/ontologies-semantic-web/introduction.html>
<http://infomesh.net/2001/swintro/>
<http://logicerror.com/semanticWeb-long>
<http://www.xml.com/pub/a/2001/03/07/buildingsw.html>
<http://thefigtrees.net/lee/sw/sciam/semantic-web-in-action#single-page>
<http://owl.cs.manchester.ac.uk/tutorials/protegeowltutorial/>

Bhavani Thuraisingham „XML Databases and the Semantic Web”; [2002] Published by CRC Press LLC

Thomas B. Passin “Explorers Guide to the Semantic Web”; [2004] Published by Manning Publications Co.

Jorge Cardoso, Amit P. Sheth „Semantic Web Services, Processes and Applications”; [2006] Published by Springer Science+Business Media, LLC.

Dieter Fensel, Holger Lausen, Axel Polleres, Jod de Bruijn, Michael Stollberg, Dumitru Roman, John Domingue „Enabling Semantic Web Services”; [2007] Published by Springer-Verlag Berlin Heidelberg

K. K. Breitman, M. A. Casanova, W. Truszkowski “Semantic Web: Concepts, Technologies and Applications”; [2007] Published by Springer-Verlag London Limited

<http://www.pcguide.com/topic.html>
<http://www.stat.uchicago.edu/~thisted/Distribute/comparch.pdf>
<http://www.webopedia.com/>
<https://www.cs.tcd.ie/Jeremy.Jones/vivio/dlx/dlxtutorial.htm>
<http://www.gigaflop.demon.co.uk/comp/chapt7.htm>
<https://www.student.gsu.edu/~sdasari1/1/Presentation1.ppt>
<http://cs-www.cs.yale.edu/homes/arvind/cs425/lectureNotes/clocks-6.pdf>
[http://msdn.microsoft.com/en-us/library/ms681205\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms681205(VS.85).aspx)

Mostafa Abd-El-Barr, Hesham El-Rewini “Fundamentals Of Computer Organization And Architecture”; [2005] Published by John Wiley & Sons, Inc.

David A. Patterson, John L. Hennessy “Computer Organization And Design”; [2005] Published by Elsevier Inc.

William Stallings “Computer Organization And Architecture Designing for Performance” [2003] Published by Pearson Education Inc.

Linda Null, Julia Lobur “The Essentials of Computer Organization and Architecture”; [2003] Published by Jones and Bartlett Publishers

Dr Jozo J. Dujmović “Programski jezici i metode programiranja”; [2005] Izdavač Naučna knjiga, Beograd

Dr Jovan Đorđević “Priručnik iz arhitekture i organizacije računara”; [1999] Izdavač Beopres, Beograd