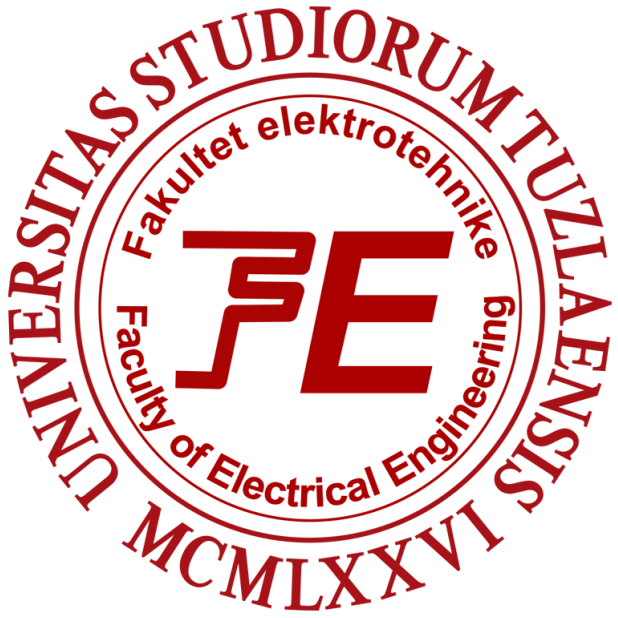
Univerzitet u Tuzli
Fakultet elektrotehnike



ARHITEKTURA RAČUNARA

JEDNOCIKLUSNA CPU IMPLEMENTACIJA

Kešetović Amar

Sadržaj

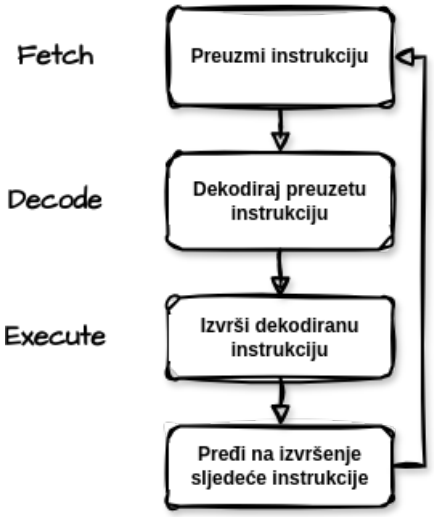
Slike

Predgovor

Ova skripta fokusirat će se na razmatranje centralne procesorske jedinice sa stanovišta njene interne jednociklusne implementacije te upoznavanjem sa osnovama rada procesora. Materijal prezentiran u ovoj skripti prati program predmeta Arhitektura računara na Fakultetu elektrotehnike u Tuzli. Za praćenje materijala koji slijedi u nastavku podrazumijeva se poznavanje i razumijevanje MIPS32 arhitekture te MIPS assemblya.

0 Uvod

Prije svega potrebno je prisjetiti se nekih osnova i postavki od kojih kreće bilo koja procesorska arhitekture koje će poslužiti za dalja razmatranja i početke same implementacije. Rečeno je kako svaki procesor, bez obzira na svoju arhitekturu, od trenutka napajanja izvršava instrukcije u petlji koja se sastoji od nekoliko koraka koji mogu biti prikazani dijagramom ispod.



Slika 1: Izvršavanje instrukcije

Napomena 0.1 Instrukcija

Instrukcija predstavlja jednostavnu operaciju, obično aritmetičku ili logičku. Instrukcija suštinski jeste niz nula i jedinica zapisanih na način specificiran za svaku procesorsku arhitekturu tako da ih ona može razumjeti i prevesti koje opisuju radnju koju računar treba da obavi. Arhitektura skupa instrukcija (ISA) definira sve instrukcije koje pojedini procesor može da izvrši. Očigledno, procesor predstavlja jednu od najbitnijih komponenti svakog računara jer se na njemu izvršava svaka instrukcija to jeste naredba data računaru.

Napomena 0.2 Startup

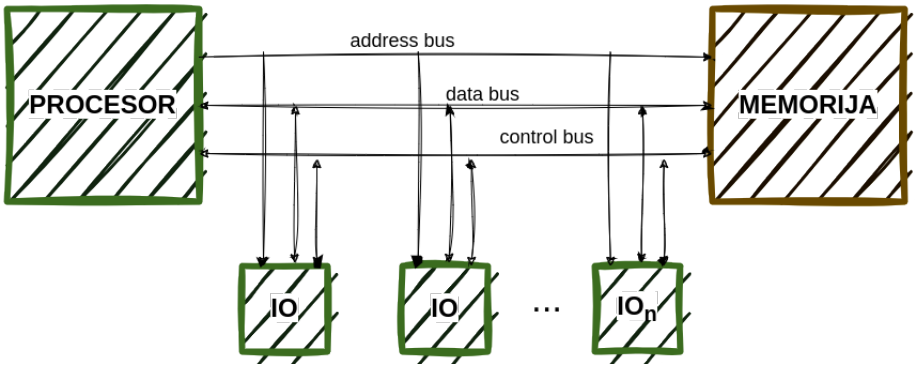
Od momenta dobivanja napajanja $pc registar postavlja se na fiksnu adresu koja predstavlja lokaciju prve instrukcije koju izvršava centralna procesorska jedinica. Procesor u svakom trenutku dok je računar upaljen izvršava neku instrukciju. Detaljnije o samom boot-u operativnog sistema i pomenutoj prvoj instrukciji pričat ćemo na predmetu Operativni sistemi.

Ponovimo i tipičnu **load-store** arhitekturu koja CPU prikazuje kao kutiju unutar koje se nalaze aritmetičko-logička jedinica u direktnoj vezi sa registar fajlom. Registar fajl nije ništa drugo do skup registara koji se nalazi direktno na CPU i predstaljvaju super brzu memoriju. U slučaju MIPS32 arhitekture registar fajl jeste skup 32 registra od kojih svaki može da spremi informaciju od 32 bita. ALU predstavlja uređaj koji pomoću osnovnih logičkih kola implementira sve operacije definiranih instrukcija datih putem ISA specifične procesorske arhitekture.

Napomena 0.3 Računar

Važno je napomenuti kako računar na najnižem nivou nije ništa više od posebno moćnog i brzog kalkulatora koji vrši aritmetičke i logičke operacije. Računar vrši milijarde kalkulacija svake sekunde na takav način da izvrši ono što mu je naređeno, prikaže sliku, pusti zvuk, pokrene igricu itd. Svaki od tih zadataka naravno nije jednake kompleksnosti i zahtijevnosti sa stanovišta računara.

Isto tako, prisjetit ćemo se i pojednostavljenog dijagrama računara.



Slika 2: Pojednostavljeni dijagram računara

Na slici su prikazane dvije glavne komponente računara, koje računar i čine računarom. Procesor kao mozak računarskog sistema koji izvršava svaku instrukciju i naređuje hardveru šta da radi, te memorija koja procesoru dostavlja instrukcije u binarnom formatu kao i podatke nad kojima treba da se operira. Sa slike se vidi kako se komunikacija između ove dvije komponente odvija putem *magistrala* ili **bus**-ova koji predstavljaju skup paralelnih žica kojima "putuju" biti.

Ovaj dijagram možemo objasniti jednostavnim primjerom pristupanja memoriji u svrhu čitanja neke sekvence bita. U tu svrhu, procesor će na adresni bus (unidirekcioni) postaviti bite na vrijednost koja čini adresu sa koje želi da čita u memoriji, dok će na kontrolnom busu (bidirekcioni) postaviti sekvencu bita koji će "reći" da procesor pristupa memoriji u svrhu čitanja podataka. Na data bus-u (bidirekcioni) naći će se informacija (biti) pročitana sa date adrese i biti proslijeđena procesoru na dalju obradu. Analogno tome, može se posmatrati bilo koja slična interakcija procesora i memorije. IO na slici predstavljaju bilo koje ulazno-izlazne uređaje koji se mogu naći u računarskom sistemu a s kojima se komunikacija odvija pomoću procesa memorijskog mapiranja pri čemu svaki uređaj dobiva adresu, a procesor komunicira s tim uređajem putem asocirane adrese.

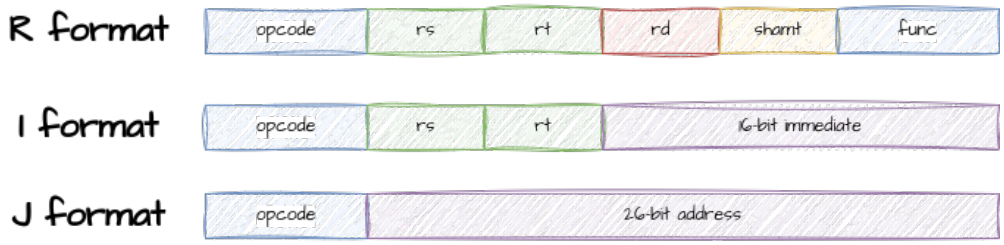
0.1 MIPS instrukcije

U svrhu naše implementacije procesora, zgodno je prisjetiti se formata instrukcija definiranih u MIPS ISA. Svaka instrukcija na MIPS platformi kodira se unutar 32-bita odnosno jedne riječi - **word**.

Napomena 0.4 Word

Word ne predstavlja definitivnu veličinu, nego se razlikuje od procesora do procesora. Word za MIPS procesor ne mora biti isti kao što je word za neku drugu arhitekturu(na primjer 16-bitni računar nikada ne može imati word = 32 bita). Word predstavlja nativnu jedinicu podatka za procesor. Također, "32-bitni" atribut u imenu procesora nam ne govori ništa o širini sabirnica tog procesora nego upravo o nativnoj sposobnosti da obrađuje tu količinu bita.

Instrukcije na MIPS platformi kodiraju se u tri formata: J, I, R. J format instrukcija koristi se za kodiranje bezuslovnih skokova, I format koristi se kod instrukcija koje uključuju numeričke konstante, te također za branch, load i store instrukcije. Za sve ostale instrukcije slobodno možemo tvrditi da su R formata. Svaka instrukcija u zavisnosti od svog tipa dijeli se na polja, od kojih svako polje predstavlja segment informacije koji daje upute procesoru koju operaciju da izvrši.



Slika 3: Format MIPS instrukcija

Prisjetimo se sljedećih činjenica:

- opcode - 6-bitno opcode polje
- rs, rt, rd - 5-bitna polja koja identificiraju izvorne i destinacijske registre instrukcije
- shamt - 5-bitno polje koje određuje broj bita za shift operacije od 0-31
- 16-bit immediate - numerička konstanta, ili relativna pozicija skoka branch instrukcija
- 26-bit immediate - apsolutna destinacijska adresa jump instrukcija
- func - 6 bitno polje koje se koristi za određivanje funkcije R tipa instrukcije

Kod R formata instrukcija, opcode polje je uvijek niz od 6 nula, a zajedno sa func poljem identifikuje operaciju koju izvršava instrukcija. Polja rs i rt predstavljaju redom prva dva izvorna registra za čitanje dok rd predstavlja destinacijski registar za pisanje. Polje shamt je nula za slučaj R instrukcija koje nisu shift operacije.

I format instrukcija ima opcode koji unikatno identificira operaciju. Svi opcode osim 0100xx(rad sa HI i LO registrima) i 00001x(j i jal) predstavlja opcode za I instrukcije. Polje rs se ponaša kao prvi izvorni registar za sve instrukcije osim load i store za koje je bazni odnosno adresni registar. Polje rt je destinacijski registar za sve I instrukcije osim za store i branch kod kojih je to drugi source registar.

J format instrukcije je format za kodiranje jump instrukcija, a totalna adresa skoka formira se uzimanjem 26 bita iz instrukcije, shiftanjem istih za 2 bita ulijevo (ekvivalentno množenju sa 4, što je u skladu sa činjenicom da vrijednost u $pc registru mora biti dijeljiva sa 4 u svakom trenutku) te preuzimanjem gornja 4 bita od trenutne vrijednosti programskog brojača. Zahtjev da je vrijednost $pc registra djeljijva sa 4, *koji inače čuva adresu instrukcije koja se trenutno izvršava/instrukcije na adresi 4 bajta iza trenutne*, proizilazi iz činjenice da je svaka instrukcija na MIPS platformi kodirana unutar 32-bita, odnosno 4 bajta, a kako $pc sadrži adresu prvog bajta od kojeg počinje instrukcija očigledno ta ista mora biti djeljiva sa 4.

Svaka instrukcija izvršava se unutar nekoliko koraka odnosno faza, pri čemu neke instrukcije tokom svog izvršenja u nekim fazama ne obavljaju nikakvu akciju tjst. miruju, dok su druge aktivne u svakoj od faza. Podjela izvršenja instrukcije na pojedine faze proizilazi iz neefikasnosti i kompleksnosti izvršenja instrukcije unutar jednog hardverskog bloka.

1 Komponente računara. CPU. Datapath.

Na slici 2 prikazan je jednostavni dijagram računara sa osnovnim komponentama. Centralna procesorka jedinica predstavlja srce računara i u njoj se izvršavaju sve instrukcije definirane u ISA specifikaciji. CPU je aktivna komponenta računara koja obrađuje sve podatke i donosi sve odluke. Sastoji se od **datapatha** odnosno staze podataka koja suštinski predstavlja podijelu na prethodno pomenute faze i dio je procesora u kojem se odrađuju sve operacije nad podacima. Drugi dio CPU-a jeste **control** koji određuje koju operaciju u datom trenutku treba da izvrši datapath. Kao što je prethodno naglašeno datapath dijeli se u komponente, koje su međusobno povezane i od kojih svaka izvršava pojedinu fazu u toku izvršenja instrukcije. Datapath faze:

1. Faza 1 - instruction fetch
2. Faza 2 - instruction decode
3. Faza 3 - ALU

4. Faza 4 - memory access
5. Faza 5 - register write

Faza 1 datapath-a predstavlja **dohvatanje instrukcije**, bez obzira na tip instrukcije, 32 bita koja opisuju sljedeću instrukciju moraju biti učitana iz memorije, u ovoj fazi također **inkrementira se** i programski brojač tj $pc = $pc + 4.

Faza 2 datapath-a jeste faza u kojoj se dekodira dohvaćena instrukcija, prvenstveno opcode polje koje dalje diktira dužinu i sadržaj ostalih polja u instrukcijji.Također u ovoj fazi dolazi i do **čitanja registara** iz registar fajla, i to **dva registra u slučaju R tipa** instrukcije, **jednog registra** za I format instrukcije, i **bez čitanja** registara za J tip instrukcije.

Faza 3 podrazumijeva aktiviranje aritmetičko-logičke jedinice u kojoj se obavlja većina instrukcija, aritmetičke, logičke, poređenja, i slično. Instrukcije za pristup memoriji koriste ovu komponentu za **sračunavanje adrese**, kao naprimjer lw kod koje je potrebno sabrati 16-bitnu konstantu sa adresom koja se nalazi u nekom registru. J instrukcija **ne koristi** ALU.

Faza 4 je faza pristupa memoriji i većina instrukcija ovu fazu **ne koristi**, preskaču je ili ostaju neaktivne u ovoj fazi. Instrukcije za pristup memoriji jedine su aktivne u ovoj fazi datapath-a.

Faza 5 je faza u kojoj dolazi do **pisanja u registar fajl**, većina instrukcija u ovoj fazi zapisuje rezultat svog izvršenja u destinacijski registar, instrukcije store, branch i jump ne pišu u registar fajl te ovu fazu preskaču ili ostaju neaktivne u njoj.

2 Logička kola. Kombinatorna i sekvencijalna kola.

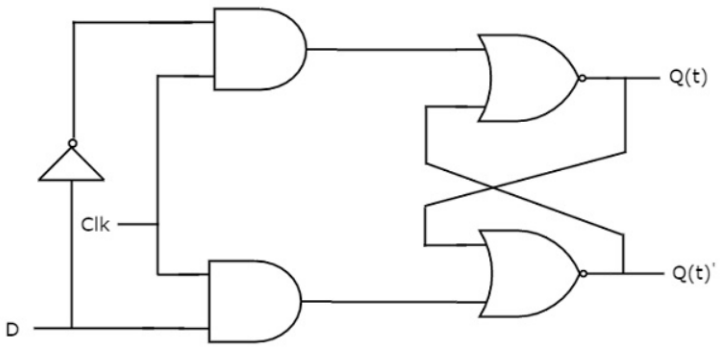
Logička kola su osnovna jedinica građe računara, čitav računar izgrađen je na osnovu **logic gate**-ova koji su zasnovani na matematici i Boolovoj algebri. Sama logička kola izrađena su od tranzistora kao osnovnog elektroničnog elementa. Spajanjem više logičkih kola u smislenu vezu dobivamo kompleksnija kola koja se uopšteno mogu podijeliti na kombinatorna kola i sekvencijalna kola.

Odlika kombinatornih kola jeste da izlaz u svakom trenutku zavisi samo od ulaza, te da isti ulaz uvijek proizvodi isti izlaz. To nije slučaj kod sekvencijalnih kola, kod kojih izlaz u nekom trenutku zavisi od kombinacije ulaza i stanja kola. Posljedica toga je da isti ulaz može prouzrokovati različite izlaze, te ulaz može mijenjati i interno stanje kola.

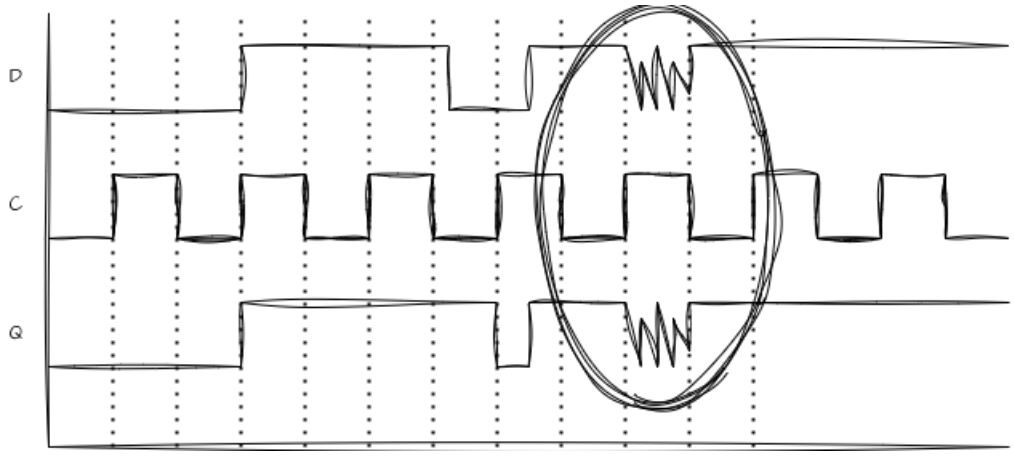
2.1 D-Latch

D-Latch je jedno od najčešće korišćenih logičkih kola za pohranu podataka u digitalnim sistemima. Ima dva ulaza od kojih jedan predstavlja podatak koji se treba zapisati označen sa D i clock signal C na osnovu kojeg se određuje trenutak promjene stanja. Ovakav latch ima i dva izlaza, od kojih jedan predstavlja vrijednost internog stanja Q, a drugi je njegov komplement odnosno uvijek suprotan njemu.

Princip rada ovakvog latcha je u sljedećem: za vrijeme trajanja clock signala na vrijednosti 1, interno stanje Q postavlja se na vrijednost koja je na ulazu D u tom trenutku, s druge strane dok je clock signal jednak nuli, kolo zadržava svoje interno stanje i zanemaruje ulaz D.



Slika 4: D-Latch kolo



Slika 5: Vremenski dijagram D-latcha

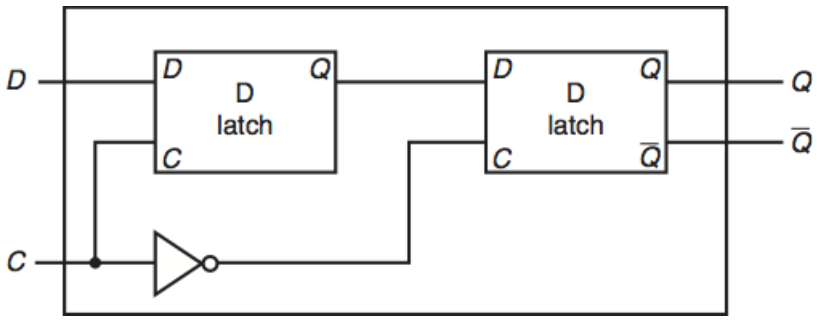
Na slici iznad prikazan je i glavni problem D-latcha, a to je što dozvoljava promjenu internog stanja za cjelokupno trajanje clock signala vrijednosti 1, pri čemu dolazi do nestabilnog stanja kola koje je zaokruženo na slici gore, kao i umanjuje kontrolu koja nam je potrebna za preciznu kontrolu procesa unutar računara.

2.2 D flip-flop

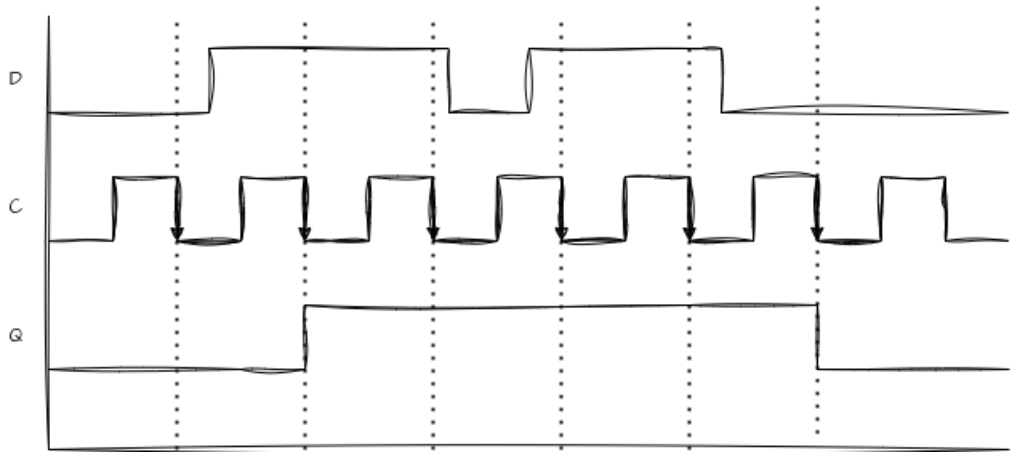
Razlika između D latcha i D flip-flopa jeste u tome što D flip-flop dozvoljava promjenu internog stanja samo na opadajućuj (**falling-edge**) ili rastućoj (**rising-edge**) ivici clock signala, i upotreba njega kao takvog je učestalija. Principijelno je isti kao i D latch, a implementira se kao kaskadna veza dva D latcha. U ovisnosti od načina uspostavljanja te veze ovisi i na koju ivicu clock signala će isti okidati. Također, moguće su i izvedbe flip flopova koji okidaju na obje ivice, ali njihova primjena nam trenutno nije od značaja. U našim razmatranjima posmatrat ćemo D flip flopove kao i ostale komponente sa pretpostavkom da sve okidaju na **opadajuću ivicu** clock signala.

Napomena 2.1 Clock metoda

Clockovi (satovi, timeri) su potrebni u kolima sekvencijalne logike da bi kontrolisali kada se stanje određenog sekvencijalnog elementa može mijenjati. Ovo se radi kako bismo osigurali tačnost podataka koji se na primjer pišu u registar fajl. Mi ćemo u našim razmatranjima pretpostavljati da svi elementi sa stanjima "okidaju" na opadajuću ivicu clock signala, te da se sve promjene stanja dešavaju na istoj ivici. Bitno je napomenuti da za vrijeme trajanja perioda clock signala podaci moraju biti validni i stabilni prije nego što dođe do okidanja, odnosno prije nego clock signal dođe do opadajuće ivice.



Slika 6: D flip-flop



Slika 7: Vremenski dijagram D flip-flopa

Na slici iznad prikazano je prethodno opisano ponašanje kola, pretpostavljajući da flip-flop radi u režimu okidanja na opadajuću ivicu.

3 Datapath komponente

Datapath komponente možemo podijeliti u dvije grupe, one koje predstavljaju komponente kombinatorne logike, prethodno definisane kao one kod kojih je izlaz potpuno definisan ulazom, i komponente sa stanjima prethodno naznačene kao sekvencijalna kola.
Kombinatorni elementi koji se susreću u datapath-u procesora:

- Aritmetičko-logička jedinica - vrši sve aritmetičke i logičke operacije
- Sabirači (adder) - half adder, full adder, vrše aritmetičku operaciju sabiranja nad bitima
- Extenderi - u našem slučaju ekstenderi 16-bitnog immediate na 32 bita koliko je veličina informacije koju pohranjuju registri MIPS32 arhitekture, princip rada je tako da uzmemo najjaču žicu odnostno 15.ti bit (brojeći od nule) i tu vrijednost ponovimo na svih gornjih 16 žica dobivajući 32-bitni signal
- **Multiplexeri** - kombinatorno kolo koje ima 2^n ulaza, i samo jedan izlaz koji se bira na osnovu n selekcionih ulaza, koristi se odabir samo jednog izlaza od mnogo mogućih ulaza na osnovu nekog kontrolnog signala, interna struktura ovog elementa nam nije pretjerano bitna ali moguće je potražiti na internetu

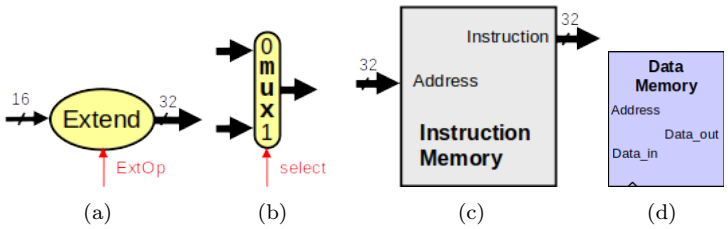
Elementi sa stanjima koje ćemo uzeti u razmatranje su:

- Memorija za instrukcije (instruction memory) - za pohranu instrukcija, read-only memorija ovdje se stavlja .text sekcija našeg programa
- Memorija za podatke (data memory) - za pohranu podataka, read-write memorija koja će nam biti stack, heap i data region (.data, .bss, .rodata sekcije)
- Registar fajl
- PC registar - programski brojač

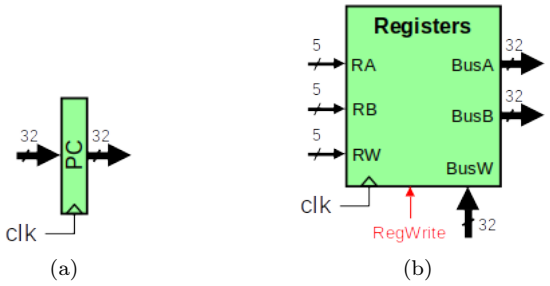
U nastavku posmatrat ćemo svaku od komponenti pojedinačno, a zatim svaku od datapath faza i tako nadograđivati dok ne upotpunimo jednociklusnu implementaciju procesora. Također posebno ćemo razmatrati datapath R instrukcije, zatim posebno I instrukcija i tako nadograđivati do kombinovanog upotpunjenog datapatha našeg CPU-a. U detalje interne implementacije pojedinih komponenti kao što su instruction memory, data memory i slično nećemo ulaziti jer je dovoljno uzeti ih kao gotove komponente za svrhe ovog predmeta. Na sljedećim stranicama koristit ćemo oznake za elemente datapath-a prikazane ispod.

Napomena 3.1 Control

Na ilustracijama u nastavku crvenom bojom bit će prikazani **kontrolni signali** koji naređuju komponentama kako da se ponašaju u određenom trenutku, koju operaciju da rade, **dozvoljavaju pisanje** i slično.



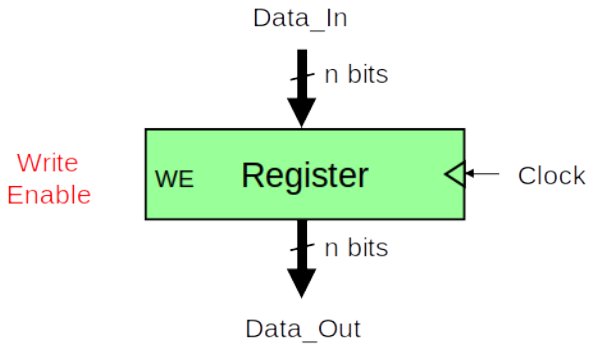
Slika 8: (a) Extender (b) Multiplexer (c) Instruction memory (d) Data memory



Slika 9: (a) PC registar (b) Registar fajl

3.1 Registar

Prvo ćemo pogledati registar kao komponentu koja ima stanje. Registar po svojoj strukturi sličan je D flip-flopu, u suštini predstavlja paralelnu vezu D flip flopova (32 za slučaj MIPS procesora), stoga je lako zaključiti da registar možemo posmatrati kao komponentu do koje se dovodi 32 žice od kojih je svaka spojena na ulaz D jednog od pojedinačnih flip flopova unutar registra i predstavlja ulaz u registar, dok također na izlazu imamo 32 žice (32 bita) koja su izvučena iz Q izlaza pojedinačnih flip-flopova.



Slika 10: Registar

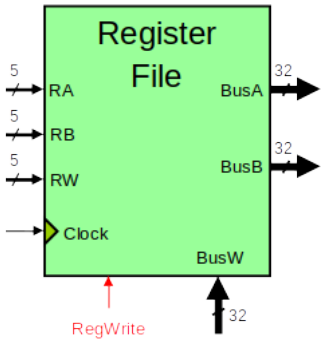
Bitno je naglasiti da clock signal predstavlja samo signal koji određuje vremenske momente u kojima je *moguća* promjena stanja flip-flopova, ali to ne znači nužno da će doći do **promjene stanja**. Da bismo kontrolisali kada želimo da signal sa ulaza flip-flopa recimo propagira na izlaz potreban je enable signal, koji je prikazan kao **write enable** na slici iznad, dakle iako je clock signal u nekom momentu na opadajućoj ivici, odnosno clock pada sa vrijednosti 1 na 0, to ne znači nužno da će nam signal propagirati kroz flip-flop jer u tom momentu write enable može biti setovan na 0. Ukoliko je write enable postavljen na 1, u momentu dolaska na opadajuću ivicu clock signala, dolazi do propagiranja ulaza i promjene stanja internih flipflopova unutar registra čime se mijenja i Data_Out na vrijednost Data_In.
Kombinacijom registara, dolazimo do prve kompleksnije komponente u implementaciji našeg procesora, registar fajla.

Napomena 3.2 Oznake

Crne strelice na ilustracijama koje su prekerižene crticom i imaju labelu koja je broj, možemo posmatrati kao skup žica čiji je broj jednak broju labele, i predstavljaju neki signal.

3.2 Registar fajl

Sa registar fajlom u opštem slučaju želimo da možemo raditi tri stvari, čitati prvi registar (registar source), čitati drugi registar(registar target), te pisati u neki treći registar. Unutar registar fajla na MIPS platformi nalaze se 32 registra, odnosno 31 jer $0 registar nije stvarni registar nego se implementira kao konstantna vrijednost 0, od kojih svaki može čuvati 32 bita informacije, to jeste svaki registar ima 32 žice na ulazu, i 32 žice na izlazu. Svi registri unutar fajla paralelno su vezani, te također imaju isti zajednički clock signal i operiraju sa njim.



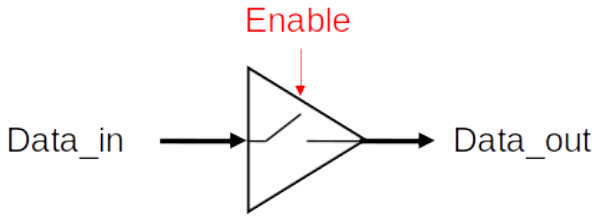
Slika 11: Registar fajl

RA i RB na slici iznad predstavljaju 5-bitne signale koji korespondiraju sa indexom registra iz kojeg želimo da čitamo(RA = **polju rs** iz instrukcije, RB = **polju rt** iz instrukcije). Dakle na ove inpute dovodit ćemo signale (bite) iz instrukcije kojima ćemo reći koje registre želimo da pročitamo. Na BusA pojavit će se 32 bita pročitana iz registra određenog signalom dovedenim na RA, a na BusB pojavit će se 32 bita porčitana iz registra izabranog putem signala dovedenog na RB. BusW predstavlja ulaz u registar fajl na kojem će se naći 32 bita podatka koji će biti upisani u registar odabran signalom koji je doveden na RW ukoliko je instrukcija koja se izvršava neka koja vrši fazu 5 odnosno piše u registre, te će u toku izvršenja te instrukcije na RegWrite pin nekako biti doveden signal koji će omogućiti pisanje u registar fajl.

Ovdje je bitno naglasiti da je čitanje stanja registara moguće u bilo kom vremenskom trenutku, i predstavlja asinhronu operaciju neovisnu od clock signala, dok je pisanje u registar fajl sinhrono i vezano za opadajuću ivicu clock signala te je moguće samo u određenim vremenskim trenucima. Ovakav princip rada omogućava pisanje instrukcija kao što su `addu $t0, $t0, $t1`, gdje za vrijeme jednog ciklusa clock signala možemo i čitati iz jednog registra i pisati u isti registar. Štaviše prilikom čitanja registara, možemo ih posmatrati kao kombinatorna kola, te njihovu vrijednost njihovo stanje možemo uvijek "pogledati" tjst. pročitati, dok je za pisanje potrebno ispuniti određene uslove. Kada ulaz na RA i RB postane validan i stabilan, nakon određenog **vremena pristupa** (access time) signali na izlazu BusA i BusB također će postati validni, ovi izlazi kasnije idu na ulaz ALU i prolaze kroz sve ostale datapath faze koje određena instrukcija koristi, da bi na kraju signal bio vraćen u registar fajl na BusW, gdje će na opadajućoj ivici clock signala ukoliko je RegWrite postavljen na 1 doći do pisanja novog stanja u registar koji je određen sa 5 bita ekstraktovanih iz rd polja R tipa instrukcije. Na sljedećim stranicama ćemo

analizirati kako interno 5 bita dovedenih na RA mogu "aktivirati" čitanje samo jednog registra i na koji način se samo izlazni signal tog registra dovodi na BusA i odgovoriti na pitanja slična ovom. Prije dublje analize internog izgleda registar fajla, potrebno je donekle se upoznati sa elementima kao što su tri-state buffer i dekoder.

3.3 Tri-State Buffer



Slika 12: Tri-State Buffer

Tri-State buffer predstavlja uređaj koji ima dva ulazna signala odnosno inputa, te jedan output. Ova komponenta omogućava nam da veći broj izvornih signala kontroliše jedan bus, naprimjer izlazi iz 32 registra registar fajla spojeni su na jedan BusA, pomoću ove komponente kontrolisat ćemo koji skup od 32 žice (čitaj koji registar i njegova vrijednost) će propagirati na BusA. Ukoliko je Enable pin jednak jedan, omogućavamo propagiranje signala i output nam je jednak inputu, u suprotnom slučaju output nam ima stanje velike impedanse, odnosno output je "isključen". Ove komponente mogu se koristiti za implementaciju multipleksera. Očigledno je kako ćemo moći na svaki izlaz iz svakog pojedinog registra postavljati jedan tri state buffer, a ostatkom logike kontrolisati da samo jedan tri-state buffer bude **enabled** i da se 32 bita tog registra kod kojeg je vezani buffer enabled nađu na BusA.

3.4 Decoder

Decoder je element koji na osnovu nekog n broja ulaza, formira izlaz čiji je broj bita jednak 2^n. Tako recimo za 5 bita dovedenih na ulazu, na izlazu iz dekodera imat ćemo 32 žice, odnosno 32 bita čiju vrijednost diktira željena logika. Ovakvi elementi mogu se kreirati korištenjem Karnaughovih mapa rađenih na Osnovama računarstva, ali za potrebe ovog predmeta nije potrebno poznavanje tog materijala, nego će se koristiti Logisim alat koji će ovaj posao odraditi za nas.

Napomena 3.3 Logisim

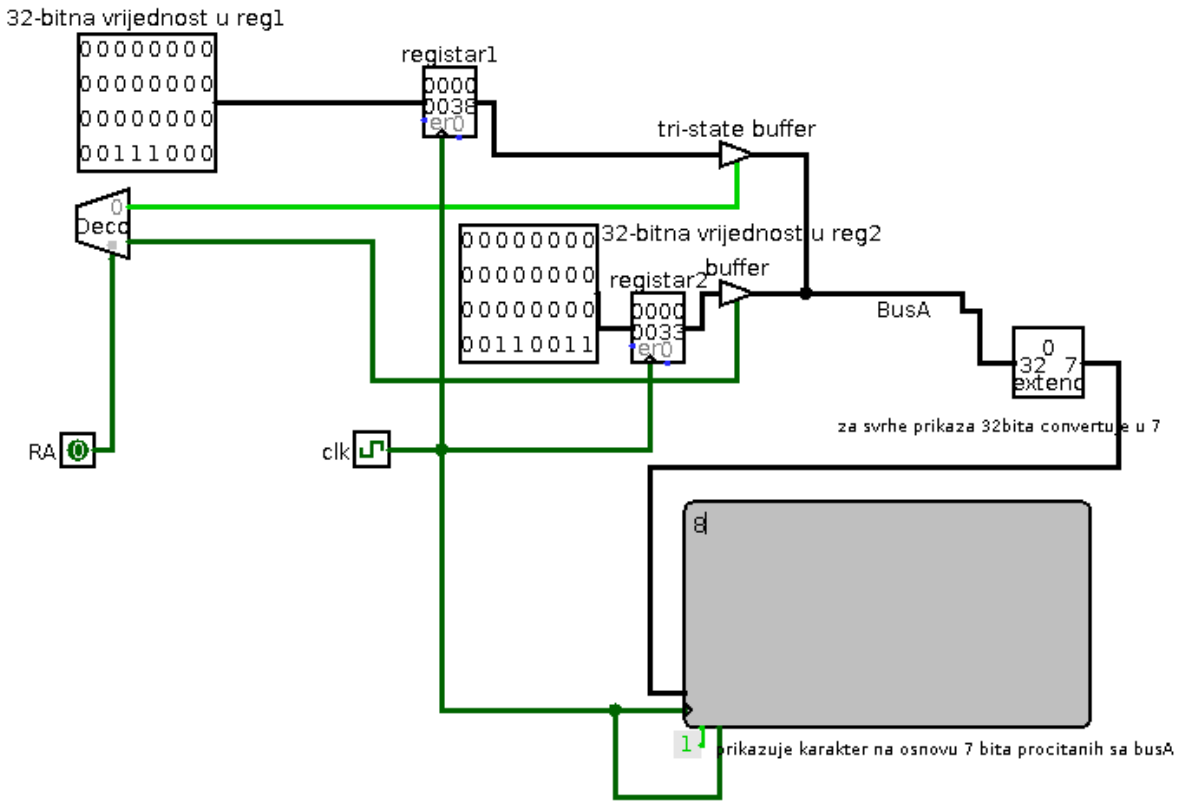
Logisim na debian distribucijama Linuxa moguće je instalirati korištenjem sljedeće komande u terminalu: `sudo apt-get update && sudo apt-get install logisim`.

Nakon čega se aplikacija može pokrenuti kucanjem `logisim` komande u terminalu ili iz menua za aplikacije.
Također unutar docker kontejnera može se pokrenuti prekonfigurisani logisim. Nakon pokretanja kontejnera potrebno je unijeti sljedeću komandu:
`cd /opt/logisim && java -jar logisim-evolution.jar`.

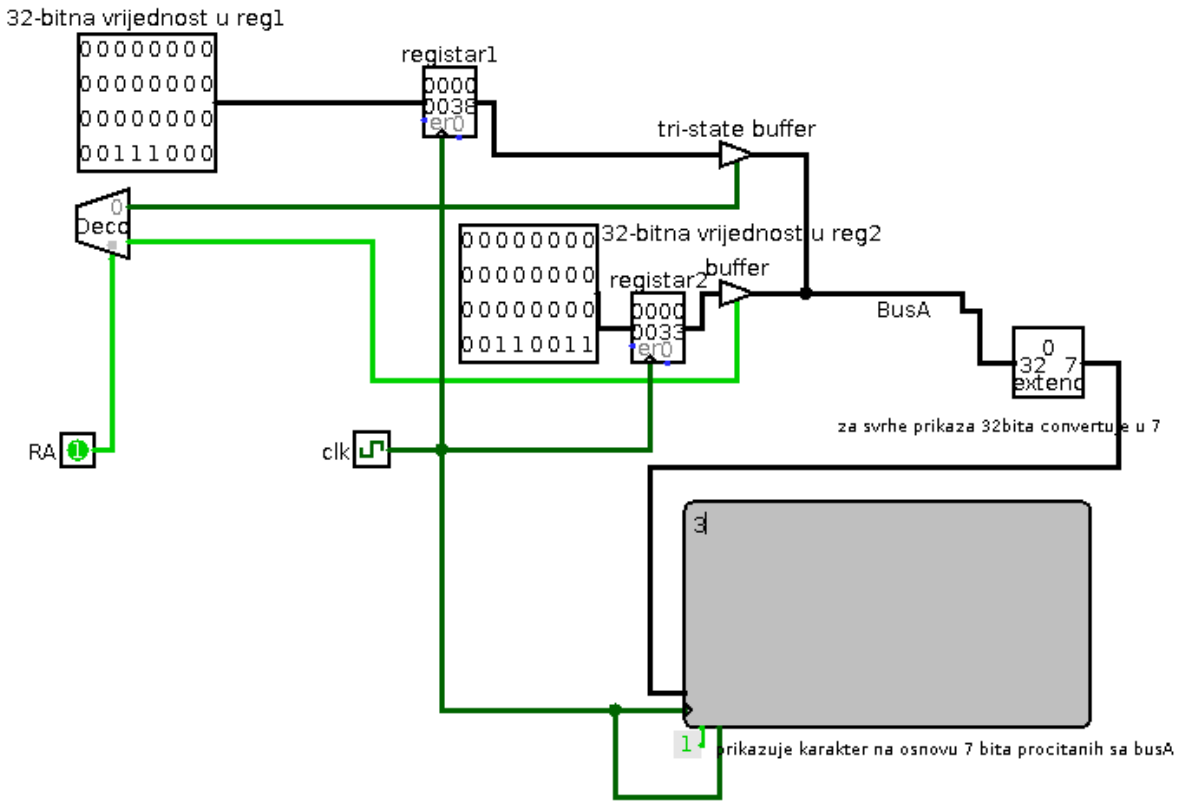
Tako recimo u našem slučaju dovedenih 5 bita na RA ulaz registar fajla bit će proslijeđeni u interni dekoder, koji će recimo registar broj 5 (ulaz `00101` na RA ulazu registar fajla) pretvoriti u sekvencu od 32 bita koja je jednaka `00000000000000000000000000100000`. Ovu činjenicu možemo iskoristi tako što će ovakav signal, odnosno svaka pojedinačna žica biti proslijeđena tri-state bufferima razmatranim u prethodnoj subsekciji čime će samo tri-state buffer koji korespondira sa registrom broj 5 biti enablean, dok će tri-state bufferima na izlazima svih drugih registara biti doveden signal 0, i tako će jedino vrijednost registra 5 biti propuštena na BusA izlaz registar fajla.

Funkciju dekodera možemo prikazati na jednostavnom primjeru posmatrajući registar fajl koji ima samo dva registra te će ovaj primjer poslužiti i za razumijevanje procesa koji se odvija u kompletnom registar fajlu. Simulacija je urađena koristeći Logisim.

Kako smo već rekli izlazi svih registara unutar registar fajla, dovedeni su na jedan isti bus (u slučaju na slici BusA), te nam je potrebna logika kojom ćemo izdvojiti samo signal registra 1 ili registra 2 u ovisnosti od vrijednosti dovedene na RA ulaz. Rekli smo kako ćemo tu logiku implementirati kombinacijom



dekodera i tri-state buffera. Na gornjoj slici stanje koje se trenutno odvija u kolu je sljedeće: na ulazu RA dovedena je `0` koja je u ovom slučaju indeks koji korespondira sa registrom 1. Dekoder u tom slučaju prvi izlaz postavlja na 1 a sve ostale na nula (u slučaju sa dva registra, drugi izlaz na 0) a taj izlaz odveden je na **enable** pin tri-state buffera koji je postavljen na izlazu iz registra 1. Vrijednost registra 2 u ovom slučaju dolazi na tri-state buffer koji je postavljen u vezu sa registrom 2 ali njegov enable pin nije uključen jer dekoder na njegov pin dovodi signal jednak nuli. Tako nam vrijednost iz registra 2 neće propagirati dalje a na BusA pojavit će se 32 bita iz registra 1. Ti biti u svrhu primjera ulaze u bit-extender koji 32 bita svodi na 7 bita tako što uzima najslabijih 7 i ispisuje ASCII vrijednost na sivi element u donjem desnom ćošku što je u ovom slučaju karakter 8 jer se na BusA našlo 7 donjih bita iz registra 1 `0111000`.

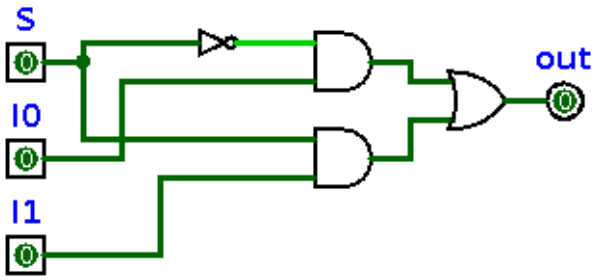


Slika 13: Primjer rada dekodera i tri-state buffera

Analogno tome, u slučaju kada na RA dovedemo `1` to je indeks koji označava registar 2. Dekoder prvi izlaz postavlja na nulu i vodi je na buffer izlaza prvog registra, a drugi izlaz postavlja na jedinicu i dovodi je na enable pin buffera na izlazu iz drugog registra. Ovime se dozvoljava propagiranje samo vrijednosti iz registra 2 na BusA, i ostatak procesa je isti. Na ekranu se ispisuje karakter 3 koji je ASCII vrijednost koja korespondira sa donjih 7 bita registra 2 `0110011`. Žice na izlazu iz buffera koji nije **enabled** možemo usporediti i reći da se ponašaju kao otpojen provodnik. Tako je omogućeno da se na BusA postavi samo signal koji je odabran kombinacijom logike dekodera i tri-state buffera. Pomoću buffera postavljenih na izlaze svakog od n registara i uz upotrebu dekodera koji dozvoljava da samo jedan od n buffera bude aktivan, na BusA postavit će se vrijednost samo 32 žice registra koji čitamo a koji biramo ulazom RA koji ulazi u dekoder. Nakon ove analize trebalo bi biti jednostavnije razumjeti šemu koja slijedi u narednoj sekciji.

3.5 Multiplexer

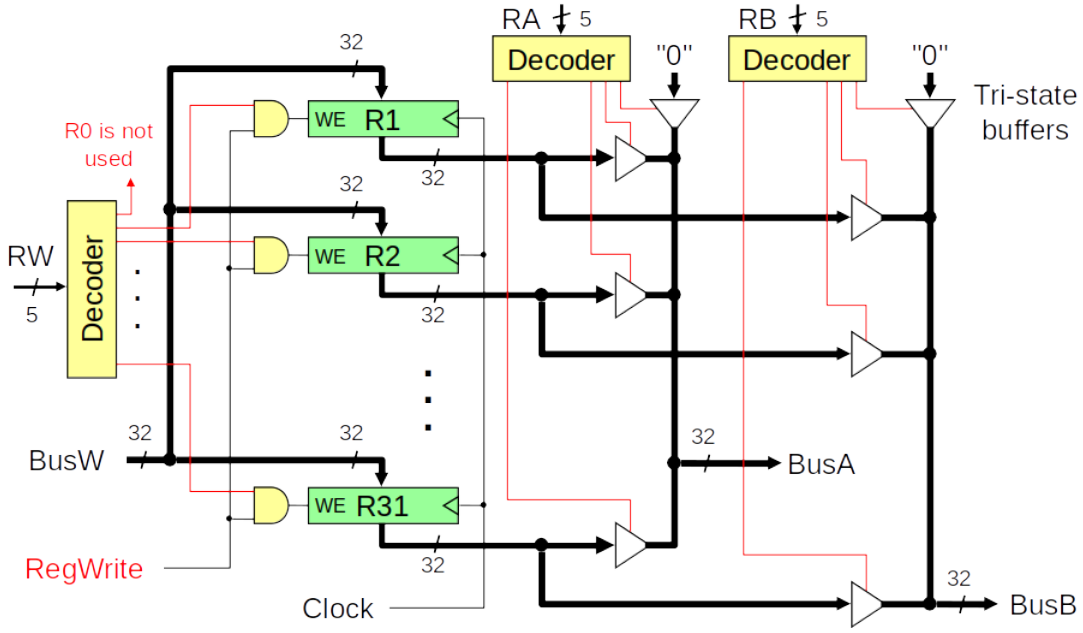
Multiplexer predstavlja uređaj kombinatorne logike, koji ima 2^n ulaza, n selekcionih signala i jedan izlaz. Multiplexer se još naziva i data selektorom. Na osnovu dovedenih selekcionih signala odnosno bita, samo jedan od ulaza multipleksera prosljeđuje se na njegov izlaz, i on kao takav poslužit će nam u implementaciji dosta komponenata u datapath-u našeg procesora gdje trebamo kontrolirati koji signal treba da bude proslijeđen nekoj komponenti.



Slika 14: Implementacija mux2u1

Princip rada multiplexera prikazanog na slici iznad je sljedeći, dovođenjem S bita 0, propušta se signal I0, a dovođenjem bita 1 na input S, na izlazu out pojavit će se signal I1. Ova logika može biti primijenjena na bilo koji multiplexer tipa mux 4u1, 16u1 itd.

4 Registar fajl. Interna implementacija.



Slika 15: Registar fajl interno

Na slici iznad prikazan je pogled unutar registar fajla MIPS procesora. Prva od stvari koju treba uočiti jeste ta da registar $0 nije stvarni registar nego da je on implementiran putem konstantne vrijednosti 0.

Sljedeće što možemo uočiti jeste da je registar fajl baš ono što smo prethodno naglasili, ništa drugo do skup registara u kombinaciji sa ostalim logičkim elementima od kojih su nam najbitniji dekoderi i tri-state bufferi čije smo principe rada i opis dali u prethodnim sekcijama.

Ovu sliku opisat ćemo krećući se od lijeva ka desno i komentarišući svaki dio ovog kompleksnog kola.

Prije svega prvo što vidimo sa lijeve strane jeste dekoder kojem je ulaz RW, 5-bitno polje koje indeksira registar u koji će se pisati odnosno destinacijski registar to jest polje **rd** dekodirano iz instrukcije koja se trenutno izvršava. Pretpostavit ćemo da svi registri čuvaju neku vrijednost i imaju neko stanje. Izlaz dekodera jeste 32-bitni signal, odnosno 32 žice od kojih je svaka spojena na po jedno AND kolo koje se nalazi na ulazu u WE(write enable) pin svakog od pojedinačnih registara. Žica R0 dekodera ostat će da "visi u zraku" s obzirom da nemamo harvderski implementiran registar $zero što smo prethodno napomenuli.

Drugi ulaz u AND kola koja se nalaze na ulazu svakog od registara unutar registar fajla jeste jedan isti **RegWrite** signal koji predstavlja kontrolni signal(označen je crvenom bojom, prethodno smo naglasili da ćemo to smatrati kontrolom) koji ukoliko je setovan na `1` omogućava pisanje u registar.

Očigledno, kako dekoder dozvoljava samo jednom od 32 bita na izlazu da bude 1, da će samo na jednom registru kod kojeg će biti dovedena jedinica sa dekodera biti omogućeno pisanje u isti uz prethodno postavljen kontrolni signal RegWrite na jedan. Ovim je završena logika koja se tiče izbora jednog registra od mogućih 31 za pisanje.

Recimo da smo na RW doveli bite 00001, i postavili RegWrite na 1, dekoder će to prevesti u sekvencu bita `00000000000000000000000000000010`. Ove nule i jedinice sa dekodera bit će svaka pojedinačno odvedena svakom od pojedinačnih registara, pri čemu će jedino kod registra R1 kojem i želimo da pristupimo AND koji se na ulazu registra imati izlaz 1 jer će samo tu i RegWrite ulaz i drugi ulaz koji dolazi sa dekodera biti 1. Tako će se aktivirati WE pin registra R1 i u njemu će biti omogućeno pisanje.

Također sa lijeve strane možemo vidjeti i BusW koji je bus putem kojeg će se dovoditi podatak koji se treba upisati u neki od registara. Ovaj bus dolazi kao rezultat izvršenja ostalih komponenata datapath-a koje ćemo razmatrati kasnije, za sada neka on dolazi "odnekud".

Priimijetimo i da je na svaki od registara u registar fajlu doveden isti clock signal koji će diktirati kada će se dogoditi pisanje u registar, to jest na njegovoj opadajućoj ivici uz pretpostavku da je u datom periodu RegWrite bio jedan i na nekom od registara WE pin također bio aktivan. Na slici uočavamo i da je iz svakog registra prema dole izvučena linija od 32 bita, koja se dalje grana i odvodi na BusA te isto tako i na BusB. Prisjetimo se da je čitanje asinhrona operacija, to jest na ovim izvučenim linijama uvijek će se nalaziti trenutna vrijednost u registru neovisno od clock signala.

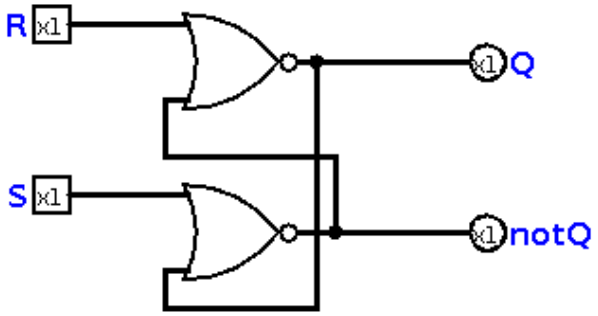
Dva dekodera koja se nalazi na vrhu slike vrše istu funkciju kao i opisani dekoder kod RW. Ova dva dekodera na osnovu dovedenih bita na RA i RB ulazima registar fajla na isti način biraju dva source registra koja želimo da čitamo. Uočavamo da su na izlazima iz registara i na svakoj grani prema BusA i BusB gdje se signali spajaju postavljeni prethodno pomenuti tri-state bufferi. Gornji dekoderi u ovom slučaju u kombinaciji sa ovim bufferima(za razliku od AND kola u lijevoj strani slike), dozvoljavaju propagiranje izlaza samo jednog od registara na buseve. Tako recimo ukoliko je na RA dovedena sekvenca 00001, na RB sekvenca 00010, dekoderi će analogno postaviti da buffer koji vodi ka busu A na registru 1 jedini bude enabled i propustiti vrijednost registra 1 na BusA, analogno tome vrijednost registra 2 bit će propuštena na BusB. Bufferi svih ostalih registara bit će postavljeni na 0 u skladu sa izlazom dekodera i njihove vrijednosti neće se pojaviti na busevima.

Također bufferi kod kojih je zapisano "0" koriste se za postavljanje vrijednosti 0 na BusA ili BusB, ukoliko je RA ili RB `00000`. Dekoder će poslati enable signal na buffere čiji je input konstantna vrijednost 0, i ista takva će se pojaviti na busu.

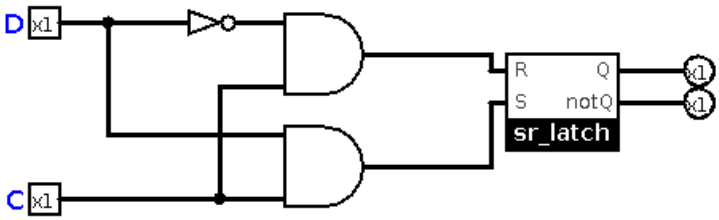
Posmatrajmo liniju assemly koda `addu $1, $2, $3`. Ova instrukcija prevedena u binarni format R glasi `000000 00010 00011 00001 100001`. Proces koji će se dogoditi u registar fajlu je sljedeći. Pet bita iz polja **rs** bit će dovedeno na **RA** ulaz registar fajla gdje će dekoder prevesti to u sekvencu bita `00000000000000000000000000000100`. Neka se u registru 2 nalazi neka vrijednost bita. Svi biti sa izlaza dekodera bit će dovedeni svakom od buffera koji su preko registara povezani na BusA. Vrijednost registra nalazi se na ulazu buffera i čeka da bude propuštena. Jedinica sa dekodera bit će dovedena na buffer registra 2 spojen sa BusA čime dolazi do propagiranja te vrijednosti na sami bus. Isto će se dogoditi i na RB, pet bita iz polja rt dovode se dekoderu, dekoder prevodi to u sekvencu bita i aktivira buffer na izlazu registra 3 koji je spojen sa BusB i dolazi do pojave vrijednosti registra 3 na busu B. Pet bita iz polja **rd** naći će se na ulazu dekodera kod RW a on će zajedno uz kontrolni signal RegWrite aktivirati **write enable** pin na registru 1 u koji pišemo te će dolaskom opadajuće ivice clock signala doći do pisanja vrijednosti u registar i time je završeno izvršenje instrukcije **addu**.

4.1 Od NILI kola do registar fajla

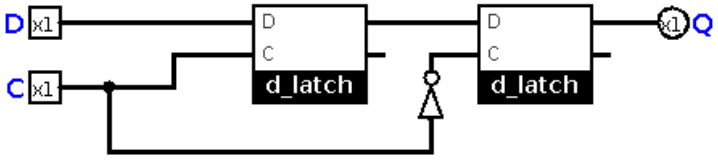
Sva kompleksna kola, svaki element kao što je dekoder, multiplexer, demultiplexer, koder, shifter, adder nastao je sintezom osnovnih logičkih kola (AND, OR, NAND, NOR, XOR,...). U ovoj podsekciji bit će prikazan put od osnovnih logičkih kola do registar fajla koji sadrži četiri 4-bitna registra.



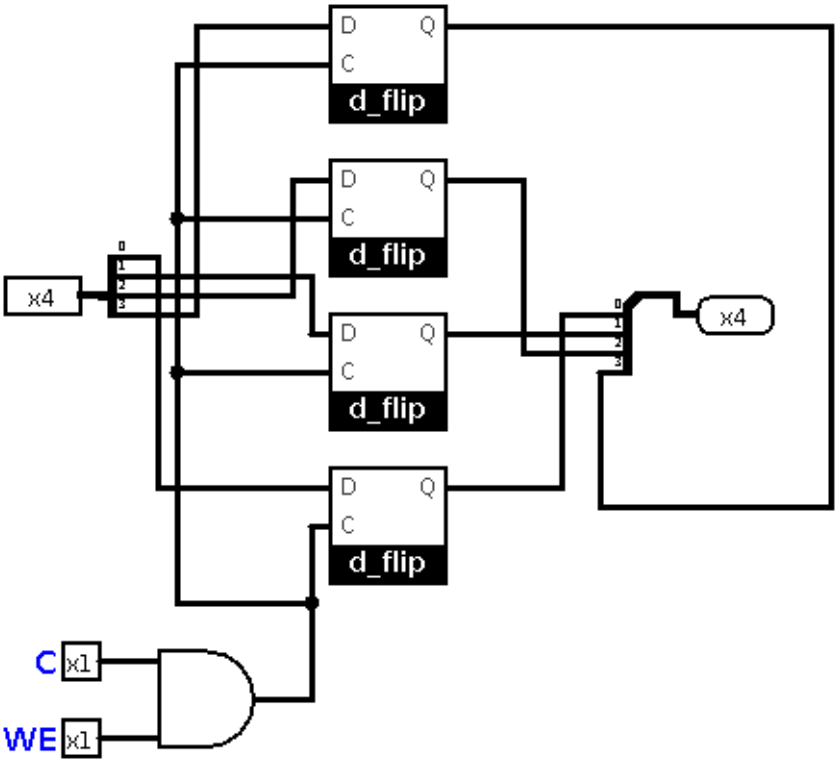
Slika 16: SR Latch



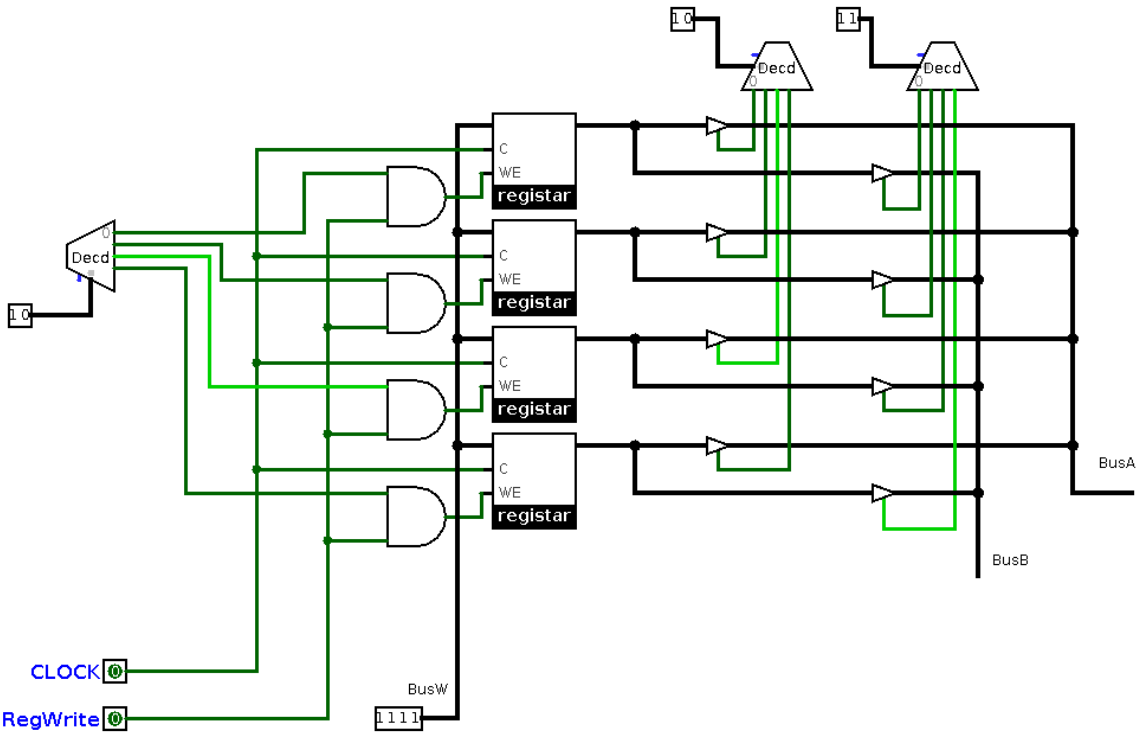
Slika 17: D Latch



Slika 18: D flip-flop



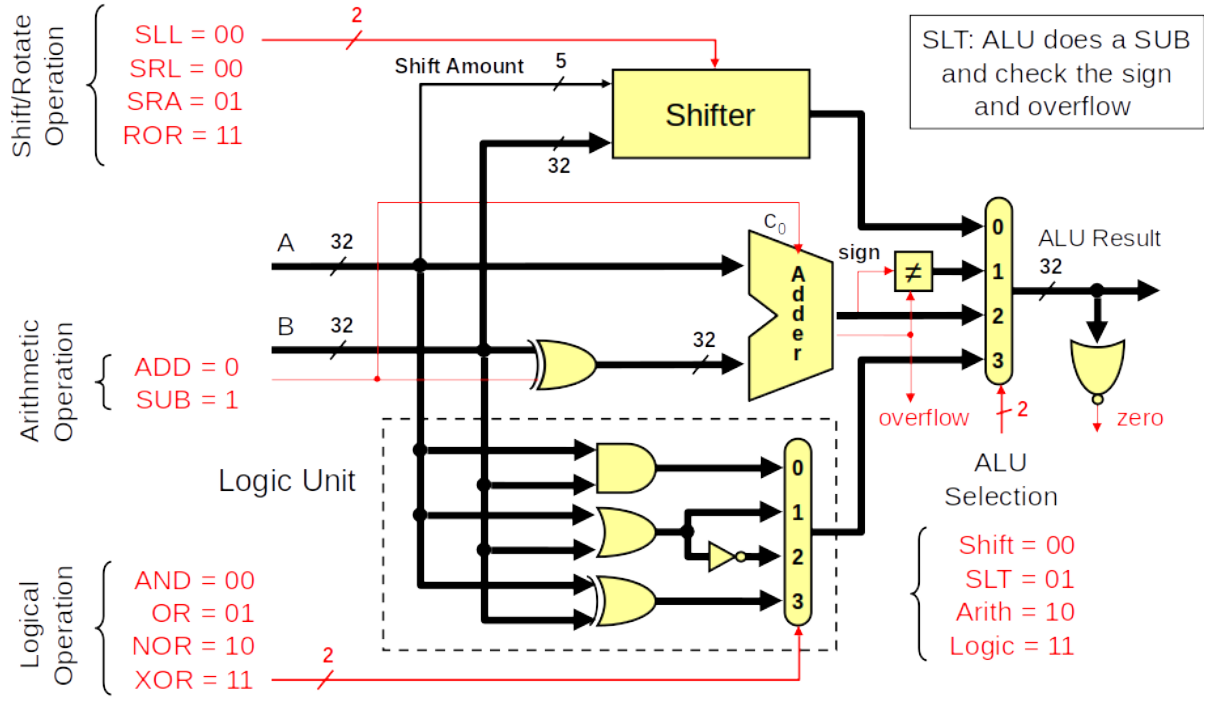
Slika 19: 4-bitni registar



Slika 20: Registar fajl 4 registra

Dakle krenuvši od dva NILI kola došli smo sve do jednostavnog registar fajla koji sadrži četiri registra koja mogu čuvati svaki po 4 bita informacije. Postupak će biti površno objašnjen jer predstavlja gradivo osnova računarstva a ne arhitekture računara. Prvo smo kreirali SR Latch koji predstavlja najosnovniji latch u digitalnoj logici. Zatim kombinovanjem SR Latcha u adekvatno kolo formirali smo D Latch čije smo ponašanje opisivali u uvodnim dijelovima. Konektovanjem dva D Latcha na specifičan način dobili smo D flip-flop koji okida na negativnu ivicu clock signala. Da smo negirali clock ulaz u prvi D Latch, a drugi ostavili regularnim, dobili bismo D Latch koji okida na pozitivnu ivicu clock signala. Kako smo već rekli registar je ništa drugo do paralelna veza D flip-flopova i to smo i uradili, AND kolo koje se nalazi u kolu služi kako bi omogućilo zapisivanje 4 bita na ulazu lijevo samo kada je clock signal jednak 1 i istovremeno WE pin je postavljen na jedinicu. Dalje primjenom opisane interne strukture registar fajla, vezali smo pojedinačne registre, oformili logiku sa dekoderima i bufferima i dobili jednostavni registar fajl prikazan na slici 20.

5 ALU Implementacija



Slika 21: ALU primjer implementacije

Slika iznad prikazuje primjer interne implementacije artimetičko-logičke jedinice. Analizirajmo sliku kao i dosad. Na linijama A i B bit će dovedena ukupno 64 bita, 32-bitni podatak na liniji A i 32 bitni podatak na liniji B. Prvo što možemo primijetiti na slici jeste da će se na podacima iz A i B paralelno svaki put raditi sve moguće operacije koje ALU radi, a putem multiplexera ćemo odabirati koji od rezultata ćemo proslijediti na izlaz AL jedinice na **ALU Result**. Na dnu možemo primijetiti da imamo poseban box koji predstavlja logičku jedinicu ili **logic unit**. A i B će se dovesti na svako od kola redom AND, OR, XOR, rezultati tih operacija nad bitima bit će dovedeni na ulaze multiplexera na dnu, redom 0, 1, 2, 3, pri čemu je na 2 doveden negirani izlaz OR kola odnosno tehnički NOR logička operacija. Ovisno od 2 kontrolna bita na multiplexeru logičke jedinice odlučuje se koji će se rezultat proslijediti dalje u kolu, odnosno na krajnje desni multiplexer na njegov ulaz 3.
U sredini slike vidimo adder, sabirač u koji ulaze ulazi A i B, pri tome ulaz B će ući kao originalno B ili njegov komplement odnosno invertovano B. Kontrolni signal od jednog bita kontroliše ulaz u XOR kolo na liniji B, koji u slučaju oduzimanja biva postavljen na jedan, time dolazi do operacije B XOR 1 čiji je rezultat komplement od B, a sabiranje A i komplementa od B je efektivno isto kao i oduzimanje A - B. A će se uvijek proslijediti onakvo kakvo je dovedeno na ulaz, direktno u adder. Adder je implementiran tako da čuva carry bit, što predstavlja overflow crveni signal koji izlazi iz addera, a on se javlja prilikom overflowa odnosno prekoračenja pri sabiranju.
Na vrhu nalazi se shifter koji izvršava shift operacije. Na osnovu dva kontrolna bita naređuje se shifteru koju operaciju da izvrši, pri tome iz signala B uzima se 5 najslabijih bita koji ulaze u shifter i predstavljaju iznos shift operacije, odnosno koliko bita se treba shiftati, a kao drugi ulaz u shifter jeste kompletnih 32 bita signala A. Izlaz iz shiftera proslijeđen je na krajnji multiplexer na njegov ulaz 0.

Napomena 5.1

Na slici iznad postoji greška, sa busa B pročitat će se najmanjih 5 bita i koristiti se kao shift amount, a 32 bita kompletnog signala A će predstavljati bite koji se trebaju shiftati!!

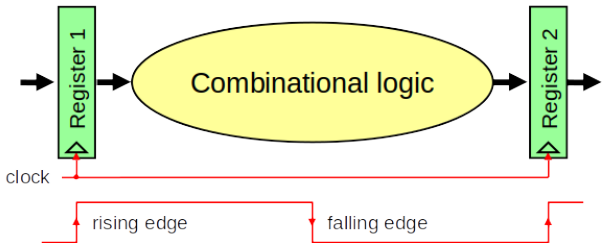
Na osnovu dva kontrolna bita **ALU selection multiplexeru** naređuje se koji od rezultata operacija prije da finalno proslijedi kao rezultat koji izlazi iz kompletne ALU jedinice, te pri tome i poseban **zero** signal se odvaja koji je uvijek 0 ako je ALU result neki broj različit od 0, ili 1 ako je ALU result jednak nuli. Tako naprimjer ukoliko želimo da saberemo ulaze A i B, na adder ćemo dovesti kontrolni bit 0, na ALU selection multiplexer ćemo dovesti 10 kako bismo propustili rezultat adderra od svih mogućih

rezultata, na izlazu ALU result dobit ćemo rezultat sabiranja ulaza A i B. Ukoliko bismo željeli izvršiti logičko nili nad bitima na multiplexer logičke jedinice doveli bismo kontrolne bite 10, a na ALU selection multiplexer kontrolne signale(selekcione) 11, tada će se na izlazu ALU pojaviti rezultat logičke operacije. Više o kontrolnim signalima, i kontroli govorit ćemo kasnije. Također mala komponenta koja ima simbol nije jednako nacrtan na sebi služi za izvršavanje slt instrukcije tjst. set less than, princip rada je takav da za slt instrukciju ALU vrši oduzimanje i provjerava predznak i overflow odnosno prekoračenje.

6 R Datapath

Kao što je već rečeno krenut ćemo sa formiranjem dijela po dijela datapath-a i na kraju ga kombinovati u finalnu implementaciju jednociklusnog procesora. Prije svega krenut ćemo sa kreiranjem datapath-a za R instrukcije, njegove implementacije i njegovog opisa. Komponente kao što su registar fajl koje ćemo koristiti za datapath R tipa instrukcija već smo analizirali, isto tako opisali smo i aritmetičko-loičku jedinicu te nam ostaje asemblirati još kopmonente koje će izvršavati preostale faze datapath-a za R format instrukcija.

6.1 Clock metodologija



Slika 22: Clock metodologija

Clock metodologija već je pomenuta u prethodnim poglavljima stoga ćemo u ovoj sekciji samo reći još par o rečenica o istoj. Dakle clockovi su nam potrebni kako bismo kontrolisali promjene stanja unutar elemenata sekvencijalne logike, imajući potpunu kontrolu nad trenucima u kojima će dolaziti do pisanja i čitanja stanja elemenata. Rekli smo također da ćemo u našim primjerima pretpostavljati da se sve promjene internih stanja dešavati na istoj ivici clock signala, u našem slučaju opadajućoj. Isto tako rečeno je kako podaci moraju biti validni i stabilni prije dolaska ivice clock signala kako bismo osigurali pravilno funkcionisanje. Sistem sa okidanjem na ivicu signala odnosno **edge-triggered** metodologija omogućava da u **jednom ciklusu** istovremeno mogu biti pročitana dva registra, te da se **istovremeno** može pisati u jedan registar (u našem slučaju). To je moguće zahvaljujući činjenici da čitanje registara predstavlja asinhronu operaciju, unutar recimo registar fajla promjenom inputa dekodera skoro **momentalno** na BusA dobivamo vrijednost registra kojeg smo izabrali signalom RA i proslijedili dekoderu, dok se isto dešava i sa BusB, zato i kažemo da za vrijeme čitanja registara možemo ih posmatrati kao kombinatorna kola. Svakako, kako su sva kola digitalne logike izrađena na principima elektronike i tranzistori čine osnovne elemente izgradnje, propagiranje signala kojeg želimo čitati nije stvarno "momentalno" nego zahtijeva određen mali vremenski period, upravo to vrijeme je prethodno označeno kao **access time**. Stoga je lako zaključiti kako naš ciklus odnosno period našeg clock signala mora biti dovoljno dug da se signali registara koji se čitaju mogu pročitati, isti signal može validirati i biti stabilan na busevima, prije nego naiđe opadajuća ivica clock signala koja će omogućiti da se podaci sa buseva zapišu u registar fajl. Suštinski period clocka mora biti dovoljno dug da se zadovolji vrijeme potrebno za put podataka kroz čitav datapath. Sve datapath faze jednociklusne implementacije CPU-a odvijaju se unutar jednog ciklusa clock signala, odakle je ovakva implementacija i dobila ime.

Napomena 6.1

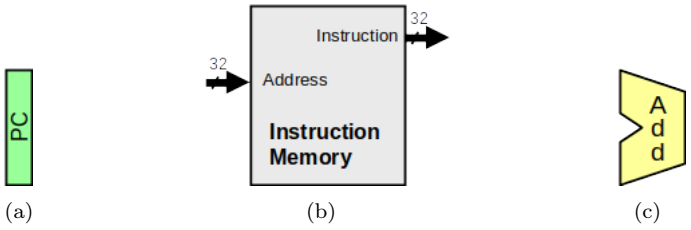
Da ne bi došlo do zabune, na slici iznad prikazan je primjer logike koja okida na pozitivnu odnosno rastuću ivicu clock signala.

6.2 Instruction fetch faza

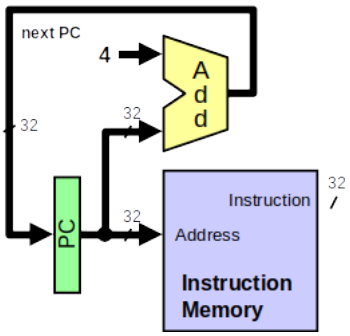
Ako se prisjetimo podjele datapath-a na faze, razumljivo je da će prva komponenata datapath-a koju ćemo implementirati biti ona za izvršavanje prve faze, odnosno faze u kojoj dolazi do pristupa isntruction memoriji i dohvatanja instrukcije. Također u ovoj fazi dolazi i do inkrementiranja programskog brojača za 4 (podjestimo se da su instrukcije u MIPS-u uvijek kodirane u 4 bajta = 32 bita, stoga je inkrement jednak 4).

Dakle ovo bi trebalo biti lagano. Sve što nam treba za ovu komponentu datapath-a jeste jedan registar kojeg nazivamo **$pc** registrom (programski brojač), logika kojom ćemo inkrementovati isti te instrunction memory komponenta u čiju internu strukturu u sklopu ovog predmeta nećemo ulaziti. Dovoljno je znati da je instruction memorija **read-only** memorija u koju će biti učitana `.code / .text` sekcija našeg programa i gdje će se nekako učitavati sve instrukcije koje naš program izvršava.

Još jednom naglašavamo kako je vrijednost u $pc registru uvijek djeljiva sa 4, i tu činjenicu iskoristit ćemo za kreiranje dvije varijante **fetch** datapath-a. Također za logiku inkrementiranja vrijednosti u registru koristit ćemo kombinatorno kolo **adder**. Ovdje vidimo i kako $pc registar zaista nije u sklopu registar fajla nego predstavlja zasebnu komponentu.



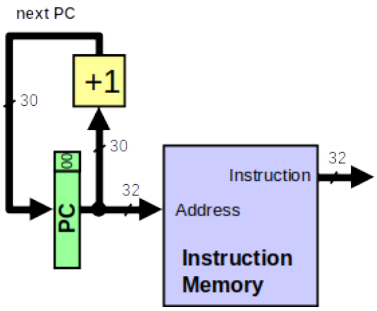
Slika 23: (a) PC registar (b) Instruction memory (c) Adder



Slika 24: Varijanta 1 fetch datapath-a

U varijanti jedan datapath komponente koja će izvršavati prvu fazu staze podataka postavili smo da sa izlaza PC registra uzimamo svih 32 bita koji se nalaze u njemu. Nakon toga tu 32-bitnu sekvencu granamo i prosljeđujemo dalje kao ulazni signal za **Address** ulaz u instrunction memory. Drugu granu ovog signala proslijedimo na jedan od ulaza addera, dok je drugi ulaz istog addera konstantna vrijednost 4 čime smo efektivno uradili inkrement vrijednosti $pc registra za 4. Tada izlaz iz addera proslijeđujemo nazad na ulaz u $pc registar u kojem će nakon udara opadajuće ivice clock signala biti zapisana adresa sljedeće instrukcije koju treba izvršiti u narednom ciklusu clock signala.

Instruction memory prihvata 32-bitni ulaz koji predstavlja adresu sa koje treba da dohvati 32 bita instrukcije koja se treba izvršiti u trenutnom ciklusu clock signala, interno je dohvata i na izlazu iz instruction memorije dobivamo 32 bita koja predstavljaju MIPS instrukciju zakodiranu u nekom od korespondirajućih formata. Kuda ćemo dalje taj signal odvesti vidjet ćemo kasnije nadogradnjom ostalih komponenti datapath-a.



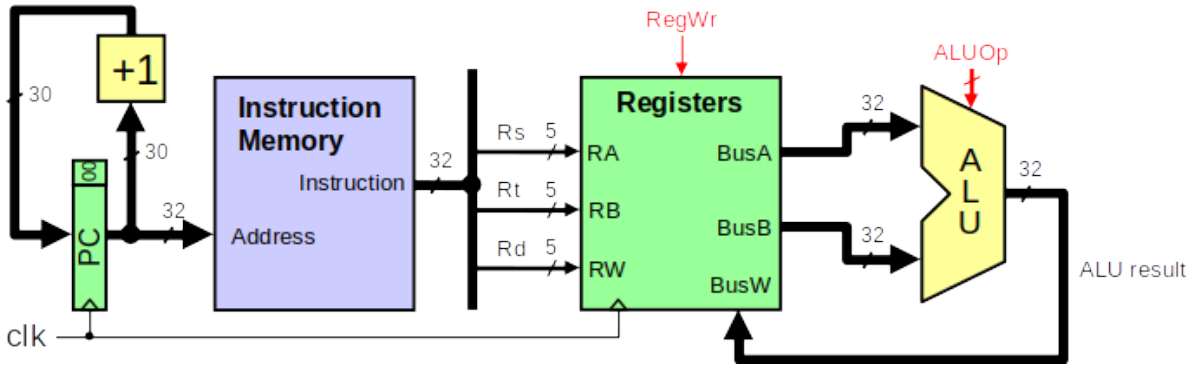
Slika 25: Varijanta 2 fetch datapath-a

Varijanta 2 implementacije prve faze datapath-a razlikuje se samo u tome što iskorištava činjenicu da je adresa svake instrukcije, pa tako i vrijednost pc registra uvijek djeljiva sa 4. Tako ćemo u ovome slučaju svih 32 bita iz registra prosljeđivati samo ka instrunction memoriji, a za inkrement vrijednosti dovoljno je proslijediti samo gornjih najjačih 30 bita iz registra $pc i sabrati ih sa jedan. Efektivno ovo je kao da smo uradili right shift za 2 bita nad registrom pc. Nakon sabiranja sa jedinicom na ulaz u $pc registar vraća se 30 bita koji će se zapisati u gornjih 30 bita registra nakon udara opadajuće ivice clock signala. Proces pristupa instruction memoriji je isti u obje varijante.

Ovime je završena implementacija prve komponente datapath-a za R tip instrukcija, ali i za sve ostale instrukcije, jer svaka od instrukcija koristi i aktivna je u fazi 1 što je i logično.

6.3 Datapath 2, 3, 5

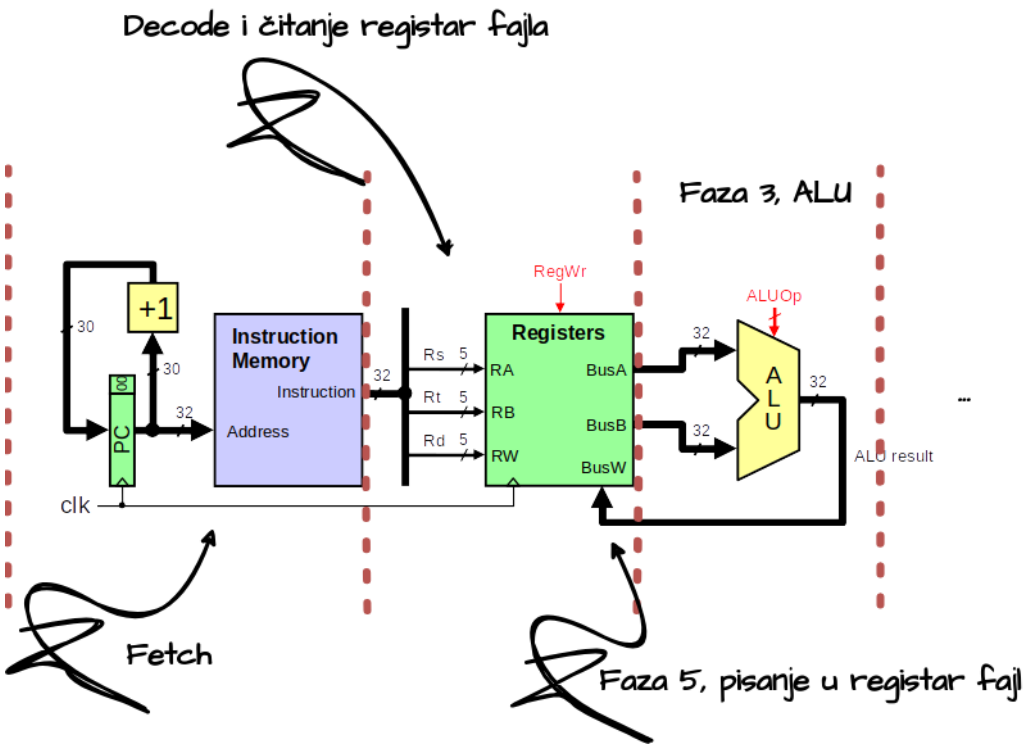
Objasnili smo da se u fazi 2 staze podataka prvenstveno dekodira informacija koja je **fetchirana** u prethodnoj fazi, a osim toga dolazi i do čitanja registara, kako ćemo prvo razmatrati R format instrukcija uvijek ćemo čitati 2 registra iz registar fajla, te ćemo također uvijek i pisati u registar fajl ali o tome malo kasnije.



Slika 26: Izgled R datapath-a

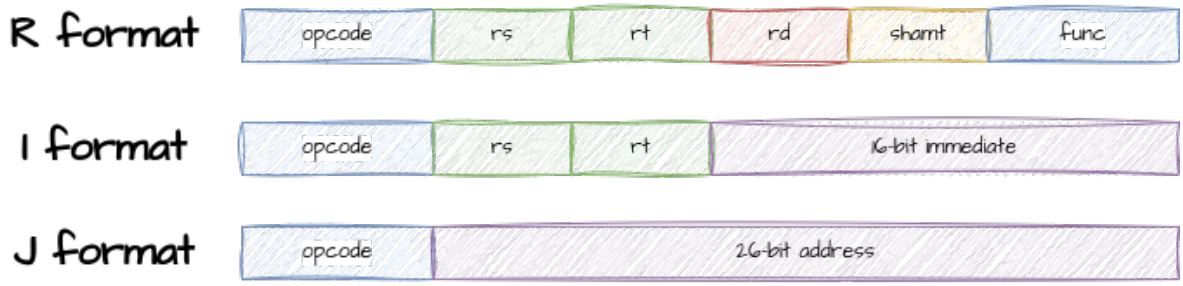
Na slici je efektivno prikazan izgled MIPS procesora koji zna da izvršava samo instrukcije R tipa. Sada je jasno kako izlaze odnosno **BusA** i **BusB** iz prethodno opisanog registar fajla vodimo na ulaz u aritmetičko-logičku jedinicu, a rezultat odnosno izlaz ALU komponente (koja se formira neovisno i čiju smo implementaciju prethodno analizirali) će predstavljati ulaz odnosno podatak koji dovodimo na **BusW** registar fajla koji treba upisati u neki od registara. Kako se kod R tipa instrukcija ne vrši pristup memoriji odnosno R instrukcije neaktivne su u fazi 4 datapath-a, tako u ovom posebnom slučaju datapath-a koji može izvršavati samo R tip instrukcija nemamo komponente **data memory**.

Gornju sliku možemo podijeliti na faze, a koje su prikazane na slici ispod.



Slika 27: R datapath podjela na faze

Podsjetimo se formata MIPS instrukcija.



Odavdje je lako primjetiti kako će sa izlaznog signala instruction memorije koji predstavlja MIPS instrukciju kodiranu u 32 bita, 5 bita iz polja **rs** biti proslijeđeno na ulaz **RA** registar fajla (obrada tih 5 bita interno u registar fajlu je prethodno objašnjena), 5 bita iz polja **rd** biti će proslijeđeno na RB, a 5 bita iz rd polja koje predstavlja registar destinacije odnosno registar u koji pišemo na RW ulaz registar fajla. Za R tip instrukcije opcode i func polje u kombinaciji određuju operaciju koju treba izvršiti. Biti koji korespondiraju sa ovim poljima bit će proslijeđeni komponentama koje za sada nisu prikazane na slici a kojima ćemo se baviti kasnije, za sada je bitno zapamtiti kako se 32 bita razbija na grupe i pojedine grupe bita koje korespondiraju sa poljima R formata instrukcije se ekstraktuju i prosljeđuju odgovarajućim komponentama.

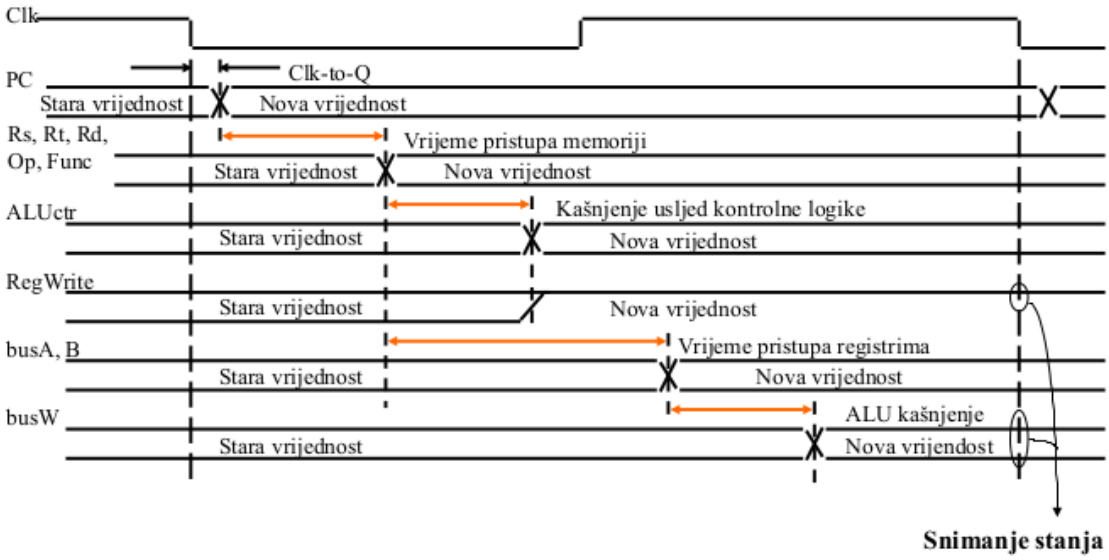
Na slici možemo uočiti i dva kontrolna signala **RegWr** i **ALUOp**, za sada o njima možemo reći da ALUOp signal nastaje od **func** polja instrukcije, a RegWr se koristi da se omogući pisanje rezultata ALU jedinice u registar fajl i to u registar odabran na osnovu 5 bita "isčupanih" iz 32 bitne instrukcije koji su dovedeni na RW ulaz. Bitno je naglasiti kako isti clock signal ažurira stanja i registar fajla i $pc registra što je prethodno i kratko spomenuto.

Na primjeru instrukcije `addu $0, $1, $2` objasnit ćemo šta se dešava u R datapath-u. Prije svega prolazi se kroz instruction fetch fazu i iz instruction memorije sa neke adrese recimo 0x0 dohvata se 32 bita koja predstavljaju pomenutu instrukciju zapisanu u 32 bita, a na ulazu u pc registar postavlja se adresa sljedeće instrukcije koja čeka na pisanje do kraja ciklusa. Iz 5 bita polja rs te instrukcije signal se odvodi na RA i predstavlja prvi izvorni registar iz kojeg se čita (u našoj instrukciji registar 1), sa 5 bita polja rt na polje RB registar fajla (registar 2) a na RW dovodi se 5 bita iz polja rd koji biraju registar za pisanje (registar 0 u našem slučaju). Nakon izvjesnog vremena čitanja na izlazima BusA i BusB pojavit će se vrijednosti pročitane iz ovih registara(1 i 2) i ti signali bit će proslijeđeni ALU jedinici. Na osnovu 6

bita func polja iz instrukcije koji će biti proslijeđeni nekom boksu ALU jedinici će biti naređeno da izvrši operaciju sabiranja bez detekcije prekoračenja. Nakon izvjesnog kašnjena usljed ALU logike na izlazu iz ALU pojavit će se rezultat sabiranja vrijednosti ova dva registra koji će biti doveden na ulaz registar fajla preko BusW. Nakon kraja ciklusa dolazi opadajuća ivica signala i rezultat se zapisuje u registar odabran poljem rd (registar 0 za ovu instrukciju) i prelazi se na izvršavanje sljedeće instrukcije. Principi rada unutar ALU objašnjeni su prethodno.

Nadogradnjom gornjih komponenti (registar fajla i ALU) na komponentu faze 1 za fetch instrukcije, efektivno smo dodali sve komponente koje se koriste u R datapath-u, registar fajl je komponenta u kojoj se odvija faza 2, aritmetičko-logička jedinica predstaljva komponentu koja obavlja fazu 3 datapath-a, dok proslijeđeni rezultat ALU koji se dovodi na BusW registar fajla jeste finalna faza 5 pisanja u registar. Prethodno je pomenuto kako R instrukcije nisu aktivne u fazi 4 odnosno ne pristupaju memoriji pa tu komponentu nismo ni uključili. Ovime je završen opis i implementacija posebnog datapath-a koji može izvršavati samo R tip instrukcija.

Preostaje nam još pokušati objasniti sliku ispod za bolje razumijevanje clock metodologije i procesa koji se odvivaju tokom izvršenja jedne instrukcije.



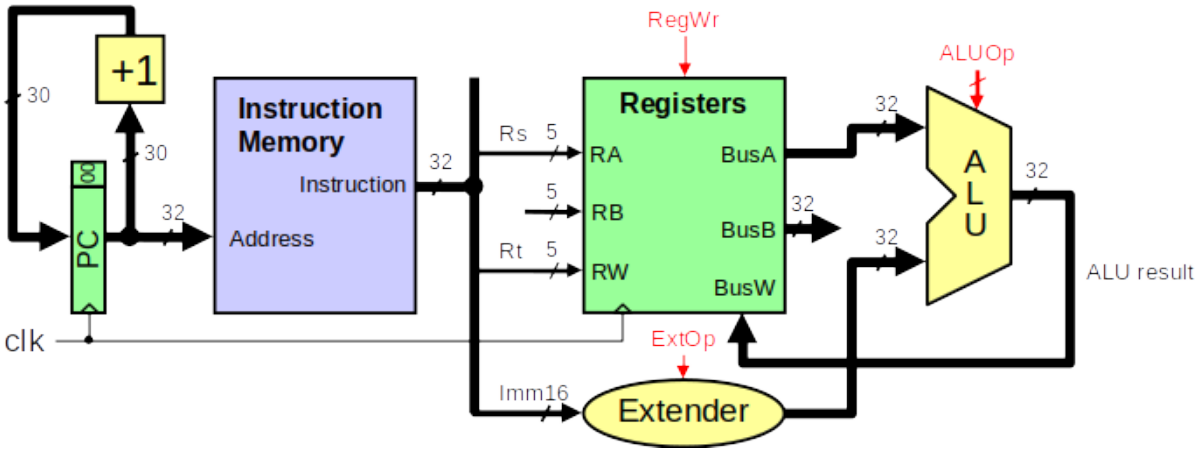
Slika 28: Vremenska analiza R datapath-a

Sliku 28 analizirat ćemo od gore ka dole, od nekog trenutka t ka nekom trenutku t + t0, gdje je t0 period clock signala odnosno jedan ciklus. Na vrhu nam se nalazi clock signal koji ćemo posmatrati od prve opadajuće ivice do one sljedeće, efektivno jedan ciklus. Nakon opadajuće ivice krajnje lijevo, završena je prethodna instrukcija i počinjemo za izvršavanjem sljedeće. Već znamo da na opadajućoj ivici clock signala dolazi do pisanja inkrementovane vrijednosti u pc registar, ta vrijednost koja se upiše u sljedećem momentu biva propagirana na ulaz instruction memorije i predstavlja adresu instrukcije koju trebamo izvršiti u tom ciklusu, dok se u međuvremenu vrši i inkrementovanje iste i ta vrijednost čeka sljedeću opadajuću ivicu da bi bila zapisana. Tako efektivno "nova vrijednost" pc registra predstavlja adresu koja se u trenutnom ciklusu nalazi u instruction memoriji i sa koje dohvatamo instrukciju. Clk-to-Q predstavlja izvjesno jako malo vrijeme koje prođe od dolaska opadajuće ivice clocka do pisanja vrijednosti u registar (u stvarnosti ništa nije momentalno, potrebno je vrijeme za promjenu stanja). Nakon toga potrebno je izvjesno vrijeme da bi se pristupilo podacima u instruction memoriji i da bi oni postali stabilni na izlazu iz instruction memorije. Sve dok taj signal nije stabilan rs, rt, rd, opcode, func polja koja se izvlače iz instrukcije imaju vrijednosti izvučene iz prethodno vršene instrukcije, a nakon proteklog vremena pristupa, nove vrijednosti ovih polja propagiraju na ulaze registar fajla RA, RB, RW. Kontrolni signali ALUctrl (=ALUOp) i RegW (=RegWrite) zadržavaju svoju staru vrijednost sve dok se ne ustabile svi prethodni signali te plus još izvjesno vrijeme koje se potroši usljed kašnjena kontrolne logike. Izlazi registar fajla, BusA i BusB zadržavaju prethodno stanje sve dok se ne izvrši pristup instruction memoriji i plus još vrijeme koje je potrebno da se pročitaju vrijednosti iz registar fajla i dok se ti signali ne ustabile, primijetimo kako ovi signali nisu ovisni o kontrolnoj logici i njeno kašnjenje ne utiče na njih. Preostaje još signal na BusW koji će imat staru vrijednost odnosno vrijednost koja je nastala kao proizvod izvršenja

neke prethodne instrukcije, sve dok se signali pročitani iz registar fajla ne proslijede ALU, te dok se izlaz ALU ne ustabili što na dijagramu predstavlja dio označen kao ALU kašnjenje. Nakon ovoga sva su stanja stabilna i preostali dio perioda clock signala čekamo na dolazak opadajuće ivice clock signala kako bi podatak bio upisan u registar fajl. Ovdje očigledno možemo zaključiti da smo period clock signala mogli smanjiti do na momenat kada je na BusW stabilna i validna nova vrijednost, međutim ostavlja se margina greške kako bi se osiguralo pravilno izvođenje ove instrukcije. Upravo ova činjenica da se period ostavlja duži nego potrebnim često se koristi u svrhe overclockinga, smanjena trajanja perioda clock signala procesora zarad bržih performansi, međutim ukoliko se to ne uradi kako treba problemi koji nastaju lako su vidljivi, smanjenjem perioda na premalu vrijednost narušit će se validnost i stabilnost podataka koji prolaze kroz CPU, te računar od momenta paljenja i prve instrukcije neće funkcionisati pravilno te tada kažemo da je računar *brickovan*.

7 I Datapath

Prirodno je sada nastaviti i oformiti poseban datapath koji će moći izvršavati samo I tip instrukcije, a nakon toga kombinacijom R datapath-a sa ovime koji slijedi dobit ćemo implementaciju koja može vršiti i jedan i drugi tip instrukcije.

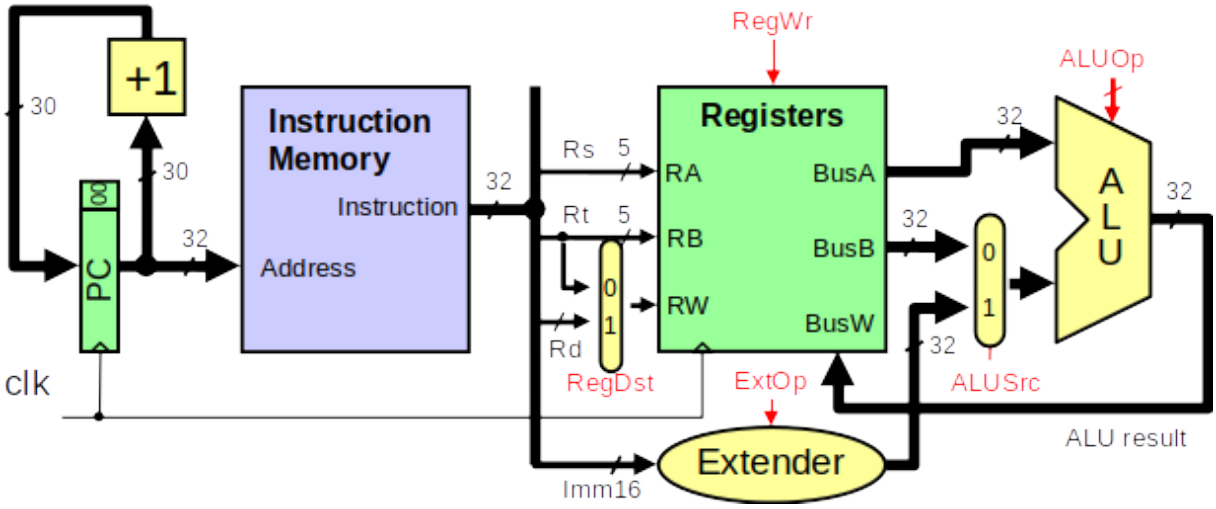


Slika 29: Izgled I datapath-a

Prvo ćemo primijetiti nekoliko razlika u odnosu na datapath koji smo asemblirali za R format instrukcija. Ono što znamo jeste da se I instrukcije kodiraju unutar opcode, rs, rt i immediate polja, dakle rd polje se neće koristiti, a za većinu instrukcija rt polje koristit će se kao destinacijski registar u koji rezultat izvršenja operacije treba biti zapisan, osim za slučaj store i branch operacija u kojima je rt izvor. Međutim za sada nećemo brinuti o store i branch instrukcijama te ćemo se fokusirati na tipične I instrukcije u kojima će nam RB polje "visiti u zraku" kao i vrijednost koja se nalazi na BusB, a 5 bita iz polja rt će biti 5 bita koji su dovedeni na ulaz RW registra i određuju registar u koji se piše. Također uočit ćemo kako je drugi ulaz za aritmetičko-logičku jedinicu umjesto BusB, za I tip instrukcija 32-bitna vrijednost nastala na osnovu proširivanja (extendiranja) donjih 16 bita dekodiranih iz instrukcije. Kod I formata instrukcija imamo i još jedan kontrolni signal **ExtOp** koji će kontrolirati način proširenja 16-bitne vrijednosti na 32 bita koji može biti ili **zero-extend**(svih gornjih 16 bita postavljaju se na 0) ili **sign-extend**(uzima se 16. ti bit, bit predznaka i u gornjih 16 bita se kopira njegova vrijednost). ExtOp za većinu aritmetičkih opracija biti će 1 odnosno uzimat ćemo vrijednost predznaka dok recimo za logičke instrukcije sa numeričkim konstantama kao što je recimo `ori` ExtOp kontrolni signal bit će 0. **ALUOp / ALUCtrl** za slučaj I format instrukcije dobiva se na osnovu bita ekstraktovanih iz opcode polja insturkcije. Ovime smo uočili sve razlike između R i I datapath-a. Ovdje se nema puno šta za reći osim razmotriti procese koji se dešavaju unutar I datapath-a pri izvršenju instrukcije na primjer `addi $0, $1, 1`. Naprije će se svakako proći kroz instruction fetch fazu i dohvatit će se 32 bita instrukcije sa adrese u instruction memoriji koja je dobivena iz registra $pc, te će se također inkrementovati vrijednost u $pc registru ali neće biti upisana do kraja ciklusa. Sada tih 32 bita se razbijaju u grupe odakle se 5 bita sa polja rs dovodi na RA ulaz u registar fajl čime se efektivno bira izvorni registar za instrukciju (u našem slučaju registar 1). 5 bita sa polja rt instrukcije bit će dovedeno na ulaz RW i efektivno će birati registar u koji treba zapisati rezultat izvršenja instrukcije (registar 0 za gornju instrukciju). Sada vidimo kako je BusB "otpojen" od ALU jedinice a drugi ulaz u ALU bit

će doveden iz donjih 16 bita instrukcije (u našem slučaju broj jedan `0000000000000001`) koji će prije ulaska u ALU biti adekvatno(ili zero ili sign-extend ovisno o instrukciji, za naš slučaj ExtOp je 1 i vrši se sign-extension) ekstendirani u komponenti koja se naziva **extender** na 32-bitnu vrijednost. ALUOp ili ALUCtrl kontrolni signal nastaje na osnovu najjačih 6 bita odnosno opcode polja I instrukcije i kontoliše operaciju koju će izvršiti ALU jedinica (sabiranje sa numeričkom konstantom sa detekcijom prekoračenja). Prethodno su se naravno na BusA i drugom ulazu ALU našli stabilni i validni signali nad kojima treba operirati, i to na BusA vrijednost pročitana iz registra određenim rs poljem, a na drugom ulazu u ALU 32 bita nastalih extendiranjem 16 donjih bita instrukcije. ALU rezultat nakon validiranje se prosljeđuje na BusW gdje će nakon dolaska opadajuće ivice clock signala biti upisan u registar unutar registar fajla određen rt poljem instrukcije (registar 0). Ovime je opisan i I datapath a sada prelazimo na kombinaciju R i I datapath-a u jedan.

8 Kombinovani datapath

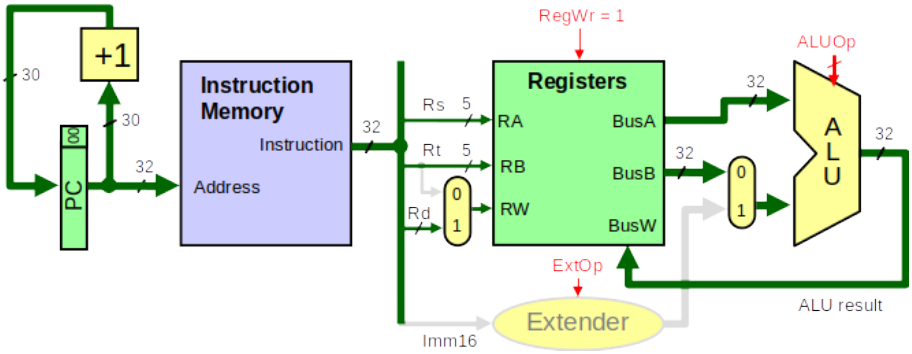


Slika 30: Izgled kombinovanog datapath-a

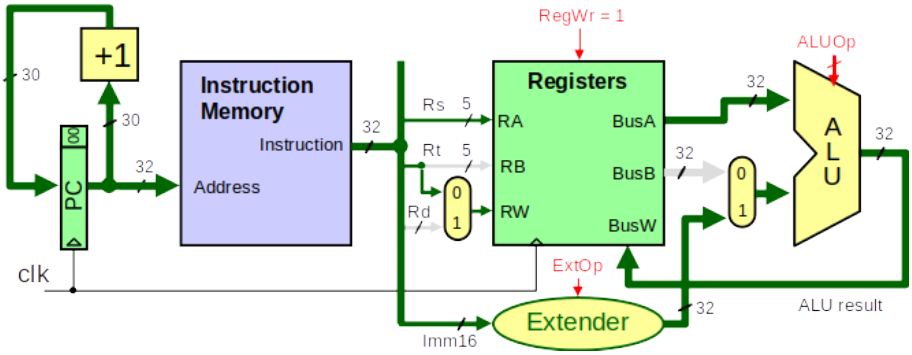
Na osnovu svega onoga što je rečeno trebalo bi biti lagano razumjeti sliku iznad.

Primijetili smo da za R tip instrukcija na RB uvijek dovodimo 5 bita sa polja rt, dok za RW polje ovisno o tipu instrukcije to može biti 5 bita ili sa **rt ili sa rd** polja. Kako mi moramo dinamički u toku izvršenja instrukcija na procesoru birati da li će se na RW naći 5 bita sa rt ili rd polja, logično je zaključiti da ćemo u tu svrhu koristiti multiplexer da na osnovu kontrolnog signala **RegDst** izaberemo koji signal ćemo propustiti prema RW. Već ovdje primjetimo kako signali za koje smo prethodno rekli da su "otpojeni" ili "vise u zraku" neće imati takvo ponašanje, nego će biti uvijek prisutni ali ih nećemo koristiti i njihova vrijednost nas neće interesovati, nego ćemo propuštati samo onaj koji nam je bitan za datu instrukciju. Tako će prema gornjoj slici za slučaj R instrukcije gdje na RW trebamo propustiti bite koji korespondiraju sa bitima na poziciji polja **rd** R instrukcije, RegDst kontrolni signal biti setovan na 1 i propustit će se upravo tih 5 bita. Za slučaj I tipa instrukcije RegDst će biti setovan na 0 a multiplexer će na RW propustiti signal koji korespondira sa 5-bita koji se nalaze na pozicijama 5 bita iz polja rt I tipa instrukcije. Dakle za slučaj I instrukcije na ulaz 1 multiplexera bit će proslijeđeno 5 bita koji su najjačih 5 bita od 16 donjih bita immediate polja iz I instrukcije, koji se nalaze na pozicijama koje bi odgovarale polju rd kada bi instrukcija bila R tipa, i ovo predstavlja **garbage** vrijednost koja nas ne zanima jer svejedno neće biti propuštena.

Nadalje, BusB će uvijek imati vrijednost koja je pročitana iz registra koji je odabran sa RB ulazom registar fajla, međutim primijetili smo kako za I tip instrukcija nećemo koristiti taj signal jer I instrukcije i ne čitaju dva registra. Međutim kako bismo kombinovali ova dva datapath-a i ovdje ćemo uvijek na RB proslijeđivati vrijednost polja rt što znači da ćemo tehnički kod I instrukcija istovremeno i čitati registar u koji treba da pišemo i njegova vrijednost će se nalaziti na BusB, ali kako za slučaj I instrukcije ne trebamo propagirati signal BusB u drugi input ALU-a tu ćemo postaviti još jedan multiplexer koji će ovisno o tipu instrukcije propustiti ili vrijednost BusB ili extendirani immediate a to će kontrolisati kontrolni signal **ALUSrc**. Tako za I tip instrukcija ALUSrc će biti setovan na 1, i multiplexer će propuštati signal dobiven ekstenzijom donjih 16 bita iz instrukcije.



Slika 31: Aktivni dio datapath-a za R instrukcije



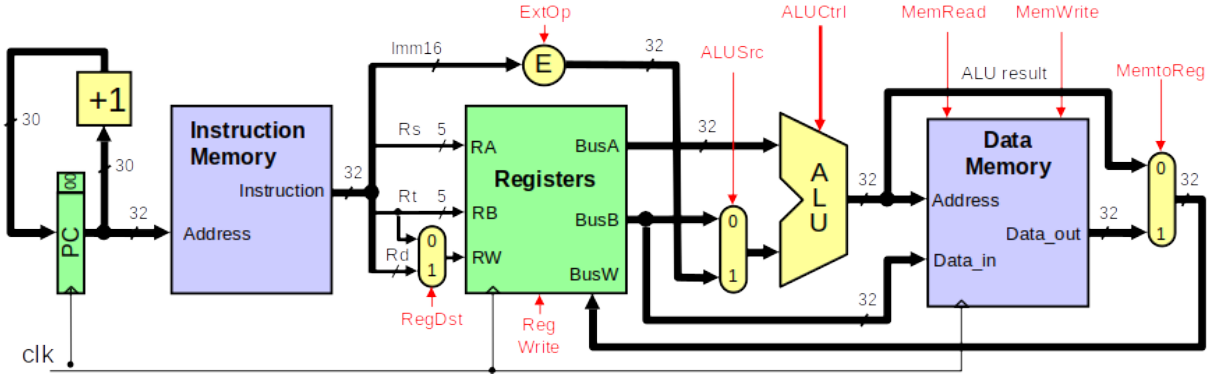
Slika 32: Aktivni dio datapath-a za I instrukcije

Na osnovu ovoga za sada možemo zaključiti kako ćemo uvijek neovisno o tipu instrukcije najjačih 6 bita instrukcije koji su opcode odvoditi negdje gore ka nekom sada nepoznatom boxu, sljedećih 5 bita svake instrukcije odvodit ćemo na RA ulaz registar fajla (rs polje), sljedećih 5 bita (rt) odvoditi uvijek na RB polje i istovremeno na multiplexer koji će odlučiti da li će ih propustiti na RW ovisno o tipu instrukcije, sljedećih 16 bita odvest ćemo uvijek na extender a zatim na drugi multiplexer na ulazu u ALU gdje će se birati da li izlaz BusB registar fajla ili extendovanu konstantu treba proslijediti ALU-u. Pri čemu prvih 5 najjačih bita (rd polje kod R) od tih 16 bit će izvedeno ka prvom multiplexeru kao drugi input gdje će u slučaju R tipa instrukcije biti proslijeđeni kao input za RW, a ako ne bit će proslijeđeno rt. Osjenčeni dijelovi na slici tehnički će uvijek imati neku vrijednost, ali ovisno o tipu instrukcije jednostavno će biti ignorisana i neće nas zanimati, to jest neće biti aktivni za određenu instrukciju. Ovime smo implementirali procesor koji kombinovano može izvršavati I i R tip instrukcija.

Pokazat ćemo još detaljno stanje ovakvog datapath-a pri izvršavanju recimo instrukcije R tipa `addu $0, $1, $2` koja je na primjer učitana na adresu 0x0 u instruction memory. Krenimo od fetch faze, u pc registru nalazit će se vrijednost 0x0 koja predstavlja adresu naše instrukcije. Ova vrijednost bit će preko izlaza pc registra proslijeđena na ulaz instruction memorije takva, a zatim će doći do njenog inkrementa i 0x4 će biti vrijednost koja će biti upisana u $pc registar nakon završetka trenutnog clock ciklusa. Instruction memorija će sa adrese 0x0 dohvatiti 32 bita koja predstavljaju kodiranu instrukciju i oni će izgledati ovako `000000 00001 00010 00000 00000 100001`. U našem kombinovanom datapath-u prvih 6 bita od gornjih 16 bit će ekstraktovano i odvedeno kontrolnoj jedinici o kojoj ćemo govoriti kasnije. Sljedećih 5 bita bit će dovedeno na ulaz RA registar fajla. 5 bita nakon toga bit će dovedeno istovremeno na RB i multiplexer kod RW. Sljedećih 5 bita bit će dovedeno na mupltiplexerov drugi ulaz kod RW. Skup svih donjih 16 bita instrukcije bit će doveden na Extender. U toku izvršavanja ove instrukcije ono što sigurno znamo jeste da je RegDest kontrolni signal 1 i da će mux propustiti 5 bita Rd signala na RW. Također kontrolni signal ALUSrc će biti postavljen na 0 i vrijednost BusB će biti proslijeđena kao ulaz u ALU, i zasigurno nam je RegWrite kontrolni signal postavljen na 1, a ALUOp na sekvencu koja diktira ALU da propusti rezultat sabiranja kao svoj izlaz. Pri ovome iako je R tip instrukcije na ulazu u extender naći će se sekvenca bita `00000 00000 100001` odnosno donjih šesnaest, a na izlazu 32 bita postojat će random signal odnosno vrijednost nastala na osnovu ExtOp kontrolnog signala koji nam nije bitan za R tip instrukcije, i ovaj signal neće biti propušten dalje ali će postojati. Na BusA nalazit će se vrijednost pročitana iz registra $1, na BusB vrijednost pročitana iz registra $2

koja će i biti propuštena u ALU. Na RA naći će se sekvenca bita `00001`, na RB `00010`. Na RW će biti propuštena sekvenca `00000`, dok će na multiplexer ulazu 0 biti `00010` a na ulazu 1 koji se i propušta u slučaju R biti `00000`. Rezultat iz ALU bit će odveden na BusW i uz kontrolni signal RegWrite i okidanje na opadajućoj ivici clock signala biti upisana vrijednost u registar 0, i u $pc će se tada naći adresa 0x4 za dohvatanje naredne instrukcije. Analogno se može analizirati primjer I instrukcije.

9 Datapath sa pristupom memoriji

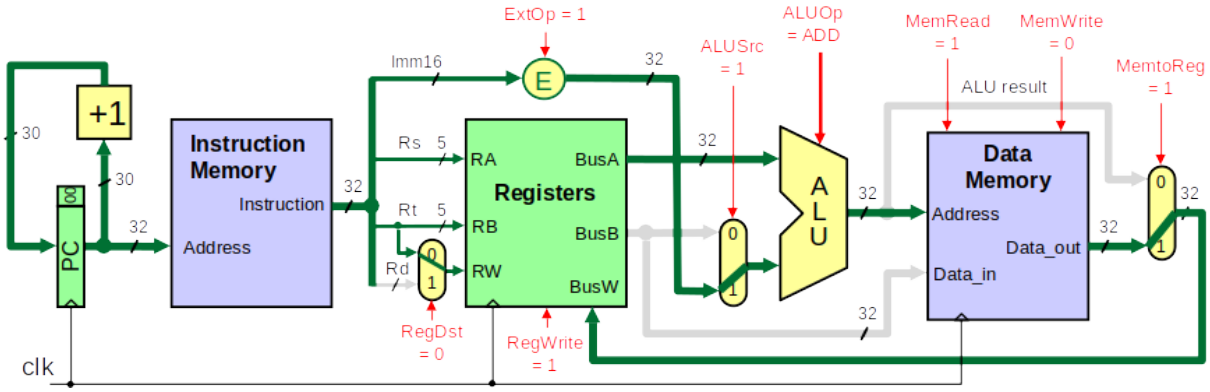


Slika 33: Datapath sa memorijom

Na slici iznad prikazan je prethodno oformljeni kombinovani data path za R i I tip instrukcije, na koji je nadovezana nova komponenta **data memory** zajedno sa novim međuvezama među komponentama.

Data memory je komponenta koja predstavlja **read and write** memoriju za podatke koju će koristiti **load i store** instrukcije koje ćemo omogućiti da naš procesor izvršava. Internu implementaciju data memorije kako smo rekli nećemo analizirati, a sada bi ovu sliku posle svega dosad trebalo biti lako opisati. Ono što znamo jeste da load i store instrukcije spadaju pod I tip instrukcija, te se za obje rs polje ponaša kao **bazni** / adresni registar, dok je rt destinacijski registar u koji se piše pročitana vrijednost loada, a **izvorni** registar za store iz kojeg se čitaju 32 bita koja trebaju biti zapisana u memoriju. Također za slučaj load ili store instrukcije u fazi 3 aktivna je aritmetičko-logička jedinica koja sračunava adresu sa koje se čita/piše u memoriju. Na osnovu ovoga očigledno je kako ćemo izlaz iz ALU prosljeđivati na **Address** ulaz data memorije, te isto tako i na multiplexer sa kontrolnim signalom MemtoReg. Ovaj multiplexer nam je potreban jer imamo dvije situacije do kojih može doći, na BusW možemo dovesti ili rezultat dobiven iz ALU jedinice, ili 32 bita pročitana iz memorije zavisno od instrukcije koja se izvršava. Isto tako kako podatak možemo pisati u memoriju u slučaju store instrukcije, a znamo da je izvorni registar iz kojeg čitamo 32 bita koja želimo storati na polju **rt** i pročitani podatak će se nalaziti na BusB, tako BusB prosljeđujemo dodatno i na **Data_in** ulaz u data memory komponentu. Izlaz **Data_out** iz data memorije prosljeđujemo na pomenuti multiplexer sa kontrolnim signalom MemtoReg. Uz ovo sve imamo i dva kontrolna signala koja su inputi u data memory koji kontroliraju da li će se moći pisati / čitati iz memorije. Pogledajmo sada interno stanja cjelokupne dosadašnje implementacije pri izvršenju load i store instrukcija.

9.1 Izvršavanje load instrukcije

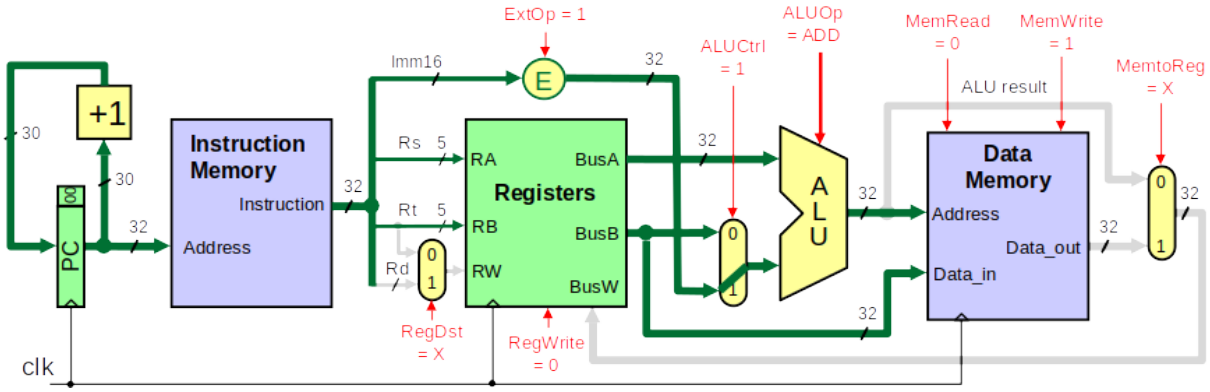


Slika 34: Izvršavanje load instrukcije

Pretpostavimo da je instrukcija koju procesor trenutno treba da izvrši `lw $0, 0($1)`. Prvi korak naravno jeste fetch instrukcije sa adrese iz **instruction memorije** na kojoj se nalazi 32 bita kodirane instrukcije, pri tome se inkrementuje vrijednost $pc registra i čeka kraj clock ciklusa za zapisivanje u registar. Dohvaćena instrukcija prolazi dalje i dolazi do raspodjele njenih bita kroz sve međuveze i komponente, odnosno dekodiranja na relevantna polja. Instukcija lw dohvatit će 32 bita iz **data memorije** i spremiti u registar $0 pri čemu će je dohvatiti sa adrese koja je pročitana iz registra $1 i sabrana sa **0**. Dakle pet bita (00001) ekstrahovanih u polje rs predstavljat će bazni registar koji će se naći na BusA i biti proslijeđen kao prvi ulaz u ALU. Za load instrukciju rt jeste destinacijski registar u koji treba zapisati pročitani podatak, pa nam je RegDst mux postavljen na 0 kako bi pet bita (00000) sa polja rt biralo registar u koji upisujemo. Donjih 16 bita iz instrukcije bit će proslijeđeno na extender pri čemu će ExtOp biti jednak jedan jer je numerička konstanta sa sračunavanje adrese signed vrijednost, a izlaz tog ekstendera dovest ćemo na ALUSrc mux gdje će ALUSrc bit postavljen na jedan kako bi kao drugi ulaz u ALU proslijedio upravo ovu konstantnu s kojom treba sabrati adresu pročitanu sa BusA iz registra određenog poljem rs. ALU jedinica vršit će kako je već rečeno operaciju sabiranja pa će kontrolni signal ALUOp biti setovan tako da kaže ALU da obavi sabiranje i proslijedi njega kao rezultat na izlazu. Sračunata adresa sa izlaza ALU proslijedit će se data memoriji koja će na **Data_out** izlaz postaviti podatak pročitan sa date adrese, te uz kontrolni signal MemtoReg postavljen na 1, taj podatak postaviti na BusW za pisanje u registar fajl u registar određen poljem rt. Dolaskom opadajuće ivice i kraja ciklusa ovaj podatak bit će zapisan u registar. Podsjetimo se na svim providnim linijama i dalje će se nalaziti biti koji su ili dekodirani iz instrukcije, ili pročitani iz registar fajla, rezultat proračunat u ALU samo što oni neće propagirati dalje pa su osjenčeni kako bi se naglasilo da ih zanemarujemo. Kontrolni signali za load instrukciju:

- RegDst = 0, bira rt polje kao ulaz za RW koji je registar za pisanje
- RegWrite = 1, omogućava pisanje u registar fajl jer load instrukcija vrši pisanje
- ExtOp = 1, postavlja ekstender tako da sign-extenda 16-bitnu konstantnu vrijednost jer je offset adrese signed numerička konstanta
- ALUSrc = 1, bira extendanu numeričku konstantu kao drugi ulaz ALU jer je instrukcija I tipa
- ALUOp = ADD, jer ALU vrši sabiranje u svrhu sračunavanja adrese
- MemRead = 1, omogućava čitanje iz memorije jer load upravo to radi
- MemWrite = 0, ne omogućava pisanje signala sa Data_in koji postoji i koji bi se upisao na sračunatu adresu na opadajućoj ivici clocka kada ne bi postojao ovaj kontrolni signal
- MemtoReg = 1, bira izlaz iz data memorije kao signal koji treba postaviti na BusW registar fajla

9.2 Izvršavanje store instrukcije



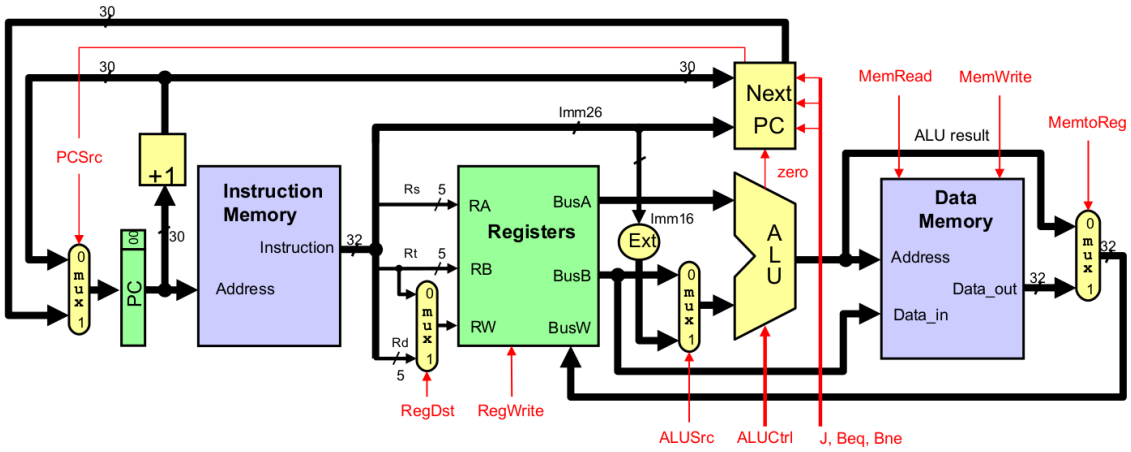
Slika 35: Izvršavanje store instrukcije

Izvršavanje store instrukcije jako je slično prethodno opisanom loadu. Ako pretpostavimo da je instrukcija koja se izvršava `sw $0, 0($1)` tada je stanje u CPU opisano u nastavku. Standardno se dohvata instrukcija i vrši se njeno dekodiranje. U slučaju **store** instrukcije polje rs je i dalje adresni registar, dok je polje rt u ovom slučaju izvorni registar iz kojeg treba pročitati podatak koji se zapisuje u memoriju. Dakle na osnovu polja rs čita se adresni registar u ovom slučaju registar $1 i pročitana 32 bita koja su adresa se prosljeđuju na ALU preko BusA. Drugi ulaz u ALU je opet sign-extendirana konstantna vrijednost dobivena od donjih 16 bita iz instrukcije. Kod **store** instrukcije rt je izvorni registar, u našem slučaju $0 čija se pročitana vrijednost dovede na BusB sa kojeg se dovodi na Data_in ulaz **data memorije**. Izlaz iz ALU se dovodi kao input u **Address** ulaz data memorije i predstavlja adresu u memoriji u koju će se zapisati podatak doveden sa BusB iz registra određenog poljem rt. Ovime je završeno sve što je trebalo završiti kroz datapath store instrukcije te sada preostaje sačekati opadajuću ivicu clock signala kojom će se podatak zapisati u memoriju.

- RegDst = x, vrijednost će postojati ali može biti bilo koja jer nam je nebitna s obzirom na činjenicu da store instrukcija ne vrši pisanje u registar fajl pa je nebitno šta će se naći kao ulaz u RW
- RegWrite = 0, omogućava da RegDst bude bilo šta jer ovaj kontronli signal onemogućava pisanje u registar fajl jer sam store ne vrši pisanje u registar fajl
- ExtOp = 1, kao i u slučaju load instrukcije
- ALUCtrl = 1, prosljeđuje sign-extendirani immediate kao drugi ulaz u ALU
- ALUOp = ADD, kao i u slučaju load instrukcije
- MemRead = 0, jer vršimo store instrukcija koja piše u memoriju a ne čita je
- MemWrite = 1, omogućava da se na opadajućoj ivici clock signala signal koji se nalazi na Data_in zapiše u memoriju na odgovarajuću adresu
- MemtoReg = x, jer nam nije bitno koji ćemo signal propagirati na BusW, jer uz RegWrite = 0 svejedno ništa neće moći biti zapisano u registar fajl

10 Datapath sa jump i branch instrukcijama

Za plan naše implementacije jedine preostale instrukcije su instrukcija bezuslovnog skoka **jump** i **branch** instrukcija poređenja. Datapath sa komponentama potrebnim za implementaciju ovih instrukcija prikazan je na slici ispod.

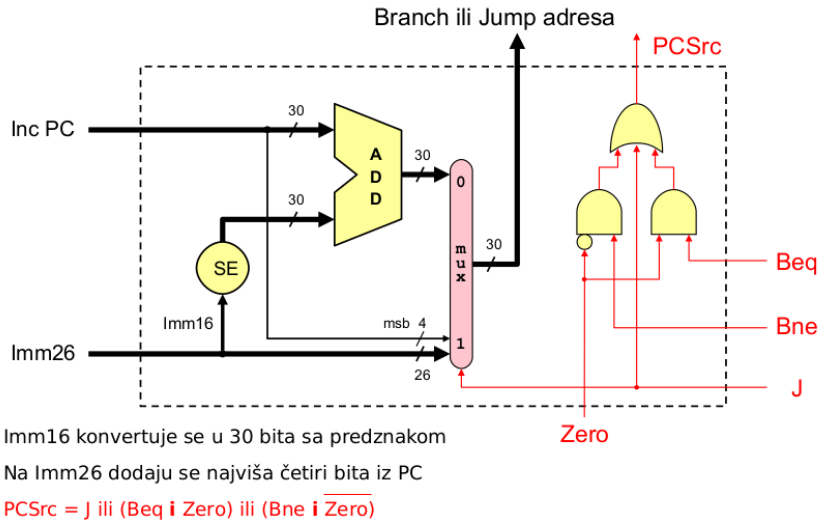


Slika 36: Datapath sa jump i branch

Na našu prethodnu implementaciju dodana je nova komponenta označena kao **NextPC** blok koji će služiti za proračun adrese koja se treba upisati u registar $pc na opadajućoj ivici signala, te tri dodatna kontrolna signala koja će kontrolisati upravo taj blok **J, Beq, Bne**. Od ova tri kontrolna signala samo jedan može biti jednak jedinici dok se izvršava neka instrukcija, a ako se ne vrši jump ili branch ova 3 signala su jednaka nuli. Ovdje vidimo i kako je **zero** kontrolni signal iz ALU proslijeđen u NextPC blok. Na ulaz u $pc registar postavljen je i multiplexer koji će birati koji ulazni signal da propagira u registar, te zajedno sa njim imamo i kontrolni signal **PCsrc** koji potiče od samog NextPC bloka. PCSrc kontrolni signal očigledno će biti setovan na jedinicu kada je instrukcija koja se izvršava jump ili branch jer tada nemamo standardni inkrement adrese u registru $pc. Zero kontrolni signal jeste uslov za grananje koji proizvodu ALU jedinica, a vrijednost **zero** signala je 0 ako je ALU rezultat različit od 0 a 1 ako je ALU rezultat jednak nuli. Za slučaj samog brancha ALU će vršiti operaciju oduzimanja, tako ukoliko imamo instrukciju beq sa dva registra čije su vrijednosti jednake, njihovim oduzimannjem ALU rezultat će biti 0, a zero kontrolni signal 1, što ćemo iskoristiti da kontolišemo NextPC blok. U suprotnom ako radimo bne instrukciju, ako dva registra koja poredimo nisu jednaka ALU rezultat bit će neka nenulta vrijednost, a zero će biti jednak 0, i ovu činjenicu ćemo iskoristiti za kasniji skok putem bloka NextPC.

Uočit ćemo i da se donjih 26 bita iz svake instrukcije dovodi na ulaz u NextPC kao i 30 bita **inkrementirane** vrijednosti pročitane iz $pc registra kao drugi ulaz. Izlaz ovog bloka doveden je na ulaz broj 1 multiplexera PCSrc. U nastavku pogledat ćemo internu implementaciju NextPC bloka i objasniti istu.

10.1 NextPC blok implementacija



Slika 37: NextPC blok implementacija

S opisom ove slike krenut ćemo od sabirača koji se koristi da sračuna relativni skok u slučaju **branch** instrukcija. Iz 26 donjih bita instrukcije, donjih 16 bita se odvodi do ekstendera unutar NextPC bloka koji iste sign-extenda na 30-bitnu signed vrijednost i ulazi u sabirač, a drugi ulaz sabirača je inkrementovana vrijednost registra $pc. Sabirač ovime efektivno sračunava adresu u kodu za slučaj branch instrukcija na principu relativnog pomaka koji je dat u donjih 16 bita branch instrukcije. Izlaz iz sabirača ulazi u ulaz broj 0 multiplexera.

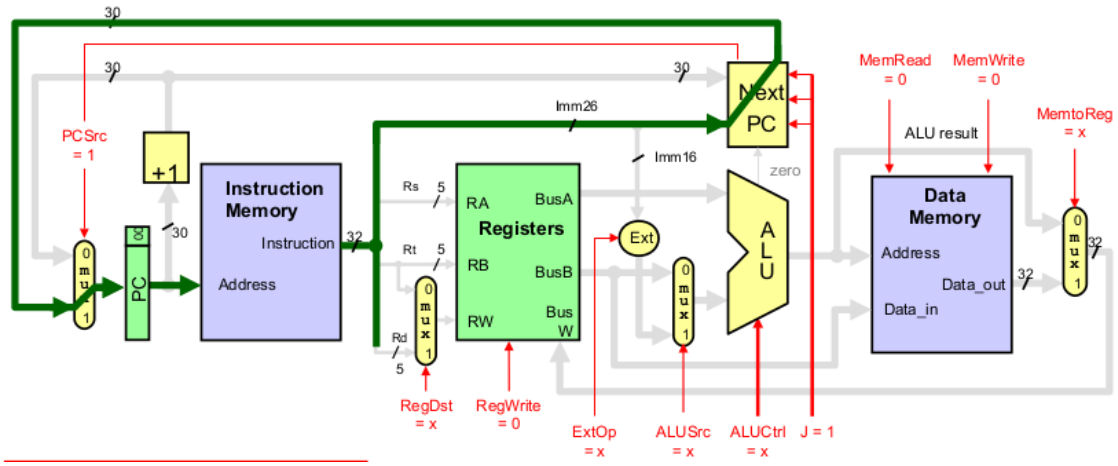
Donjih 26 bita instrukcije također je direktno proslijeđeno na multiplexerov ulaz broj 1, kao i 4 najjača bita iz IncPC vrijednosti što je posljedia prethodno objašnjenog načina formiranja apsolutne adrese jump instrukcije. Kontrolni signal koji kontroliše ovaj multiplexer je J, dakle ako je instrukcija koja se izvršava jump, signal **J** biti će postavljen na jedan i 30 bita dobivenih spajanjem najjača 4 iz IncPC signala sa donjih 26 bita dekodiranih iz instrukcije bit će postavljeno kao izlaz koji se vodi ka registru $pc. Ako je signal J jednak nuli, onda se adreda proračunata u sabiraču propagira prema registru. Vidimo da ćemo uvijek ovdje imati vrijednost koja će se odvoditi na multiplexer na ulazu u pc registar, ali za slučaj instrukcija koje nisu jump ili branch to će biti **garbage** vrijednost koju svejedno nećemo propagirati.

Desno je prikazana i logika koja kontroliše PCSrc kontrolni signal, koji kontroliše koja vrijednost će se pisati u $pc registar. Znamo da ako je PCSrc = 1 da ćemo u registar $pc upisivati adresu sračunatu u NextPC bloku. Analizirajmom sada kada je signal PCSrc jednak jedinici. Na osnovu slike desno vidimo da ukoliko je instrukcija **jump** (*bezuslovni* skok), J signal je uvijek jednak jedinici, PCSrc je definitivno jedinica i u registar ćemo upisati vrijednost adrese dobivene kombinacijom donjih 26 bita iz insturkcije i 4 najjača bita iz IncPC.

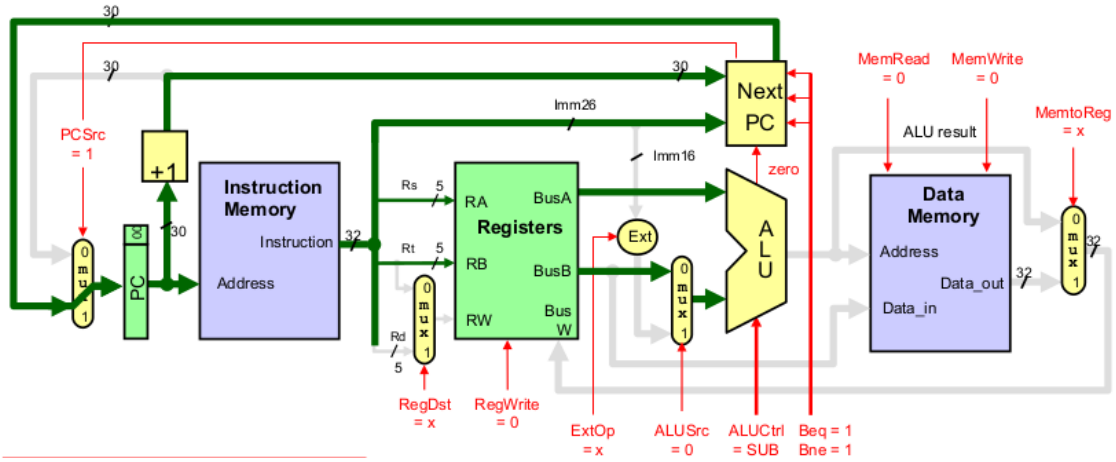
Kada je instrukcija koja se izvršava **beq** tada su kontrolni signali J i Bne jednaku nuli, a kontrolni signal Beq je jednak jedinici. Ako je ALU rezultat oduzimanjem vrijednosti registara koje se porede dobio rezultat 0, to znači da su registri jednaki, i rekli smo da je tada **zero** signal jednak jedinici. Očigledno s ovime ulazi u AND kolo Beq i Zero su jedinice, pa je i PCSrc jedinica. Pri tome kako je J signal jednak nuli, iz NextPC izaći će adresa proračunata u sabiraču jer je propušten signal sa ulaza broj 0 multiplexera. Ukolik bi rezultat ALU bio nenulta vrijednost, znači da branch ne trebamo obaviti što će Zero signal i učiniti jer će biti postavljen na nulu, izlaz AND kola će biti nula pa i sam PCSrc.

Preostala je još samo jedna situacija a to je kod izvršenja **bne** instrukcije. Vidimo da je Bne kontrolni signal ulaz u lijevo AND kolo, a drugi ulaz u to AND kolo jeste negirani **Zero**. ALU će poredeći vrijednosti dva registra oduzimanjem dobiti nenultu vrijednost ako vrijednosti registara nisu jednake i tada **trebamo** obaviti branch (branch on not equal). Zero će biti postavljen na 0 od strane ALU, ali kako za slučaj bne instrukcije mi tada trebamo obaviti grananje moramo negirati to zero. Tako da ukoliko dva registra nisu jednaka ulaz u AND kolo su jedinice pa je i PCSrc jednak jedinici, i analogno propuštena je vrijednost adrese sa ulaza broj 0 multiplexera. Ako bi rezultat ALU-a bio nula, registri bi bili jednaki i u slučaju bne instrukcije ne bismo trebali obaviti grananje, što će uz Zero postavljen na 1 i negiran na ulazu u AND kolo i učinio. Dakle ukoliko uslov za grananje nije ispunjen kod beq ili bne instrukcija, PCSrc postavlja se na nula, i sljedeća vrijednost registra $pc bit će ona inkrementovana za 4.

10.2 Izvršavanje jump i branch instrukcija



Slika 38: Izvršavanje jump instrukcije



Slika 39: Izvršavanje branch instrukcije

Sada nema potrebe detaljno opisivati slike iznad. Zelenim putanjama pokazani su aktivni signali koji će se koristiti, a osjenčeni su oni koji će postojati ali neće se koristiti ni u kom slučaju prilikom izvršenja.

Kod izvršenja jump instrukcije iz instruction memorije dohvati se 32 bita jump instrukcije, raspodijeli se na polja pri čemu nas interesuje samo donjih 26 bita koji ulaze u NextPC blok. Ovaj blok ztim odrađuje prethnodno opisanu magiju a uz kontrolni signal J i PCSrc i opadajuću ivicu clock signala postavlja u $pc registar apsolutnu adresu za skok do sljedeće instrukcije.

Kod izvršenja branch instrukcija inkrementovanih 30 bita trenutne vrijednosti $pc registra proslijeđeno je bloku NextPC. NextPC blok uzima i donjih 26 bita iz dekodirane instrukcije preuzete iz memorije od kojih interno koristi samo donjih 16. Na polju rs i ulazu RA naći će se index prvog registra koji se poredi, a na polju rt indeks drugog registra za poređenje. Na BusA i BusB pročitane vrijednosti ova dva registra proslijeđuju se ALU jedinici direktno i preko multiplexera uz kontrolni signal ALUSrc = 0, pri čemu ALU vrši oduzimanje a na osnovu tog oduzimanja daje i signal zero koji prosljeđuje NextPC bloku. NextPC blok uz zero signal, i Beq ili Bne odredit će da li treba doći do grananja, a ako ne PCSrc će postaviti na 0 i neće doći do zapisivanja sračunatih vrijednosti u $pc registar, nego one za 4 bajta inkrementovane.

Prokomentarisat ćemo i postavku kontrolnih signala.

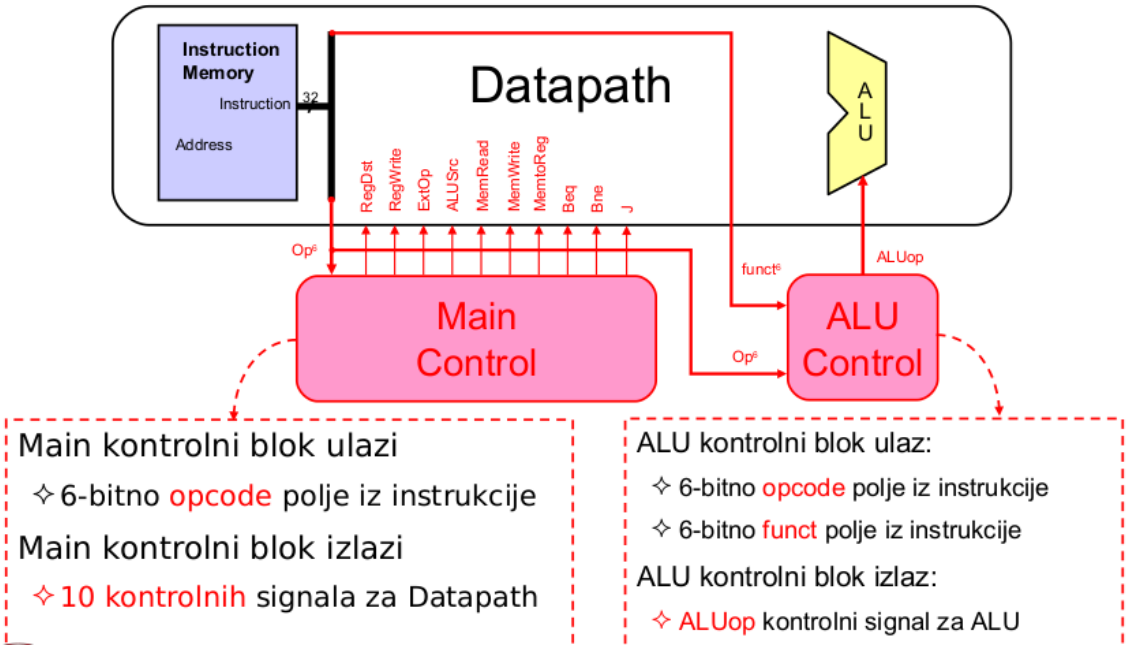
Za slučaj izvršenja **jump** instrukcije PCSrc definitivno mora biti jednak jednici jer je vrijednost adrese za sljedeću instrukciju definitivno dobivena iz NextPC bloka. Isto tako obavezno ne smijemo dopustiti pisanje u registar fajl, niti čitanje i pisanje iz data memorije pa ovi kontrolni signali moraju biti jendaki nuli. J kontrolni signal koji ulazi u NextPC bit će 1 jer se izvršava jump, a Beq i Bne bit će setovani na

nulu. Svi ostali kontrolni signali imaju vrijednost **x** odnosno mogu imati proizvoljnu vrijednost jer svi ti signali koji će se dobiti iz komponenti kao što su ALU, ekstdener, ALUSrc multiplexer itd. neće se ni koristiti pa nemamo potrebu da ih kontrolišemo na bilo koji način.

Za slučaj izvršenja **branch** instrukcije, PCSrc će biti jednak jedinici ako je zadovoljen uslov za grananje. Opet moramo zabraniti pisanje u registar fajl, i čitanje i pisanje u data memoriji pa su ovi kontrolni signali jednaki 0. Kod branch instrukcije ALU jedinica je aktivna jer vrši oduzimanje u svrhu poređenja pa je ALUCtrl setovan na vrijednost bita koja korespondira sa operacijom oduzimanja. Isto tako ALUSrc u ovom slučaju treba biti setovan na 0 kako bi drugi ulaz u ALU bila vrijednost pročitana sa BusB iz registra odabranog poljem rt branch instrukcije odnosno drugog izvnornog registra koji se poredi. U ovisnosti da li je riječ o beq ili bne instrukciji, jedan od signala Beq ili Bne će biti setovan na jedinicu. Svi ostali kontrolni signali mogu imati proizvoljnu vrijednost jer komponente koje kontrolišu ne utiču na rezultat izvršenja instrukcije.

11 Kontrola datapath-a

Do sada smo dosta spominjali kontrolne signale koji kontrolišu naš datapath, omogućavaju pisanja, čitanja, propuštaju odgovarajuće signale, naređuju komponentama kako tačno da obrade ulaz i slično. Lagano je zaključiti da ovu kontrolu određujemo iz same instrukcije koja se trenutno izvršava, te da ona diktira vrijednosti kontrolnih signala. Dakle, morat ćemo oformiti na neki način neku kontrolnu jedinicu koja će na osnovu dekodirane instrukcije postavljati kontrolne signale na odgovarajuće vrijednosti. Upravo ova kontrola, kontrolna jedinica / jedinice predstavljaju posljednji korak u finalizaciji naše *jednociklusne implementacije centralne procesorske jedinice.*



Slika 40: Kontrola datapath-a

Prethodno nismo spomeninjali odakle tačno dolaze kontrolni signali, ali moglo se naslutiti da će opcode i func polje iz dekodiranih instrukcija biti ono iz kojeg će se ekstrahovati način setovanja kontrolnih signala. Na slici iznad vidimo da ćemo imati dvije odvojene kontrolne jedinice, **main control** koja će kontrolirati većinu komponenti istovremeno, i **ALU control** jedinicu koja će kontrolirati ALU komponentu. Vidimo kako će se najjačig 6 bita, odnosno opcode polje instrukcije proslijeđivati kao ulaz u main control i to će joj biti jedini ulaz. Na osnovu ovog ulaza main kontrolni blok interno će odrediti postavku 10 kontrolnih signala. Izlaz iz main kontrolnog bloka dakle bit će 10 kontrolnih signala za datapath: RegDst, RegWrite, ExtOp, ALUSrc, MemRead, MemWrite, MemtoReg, Beq, Bne, J.

ALU kontrolni blok ima dva ulaza, a to je 6-bitno opcode polje te 6-bitno func polje iz dekodirane instrukcije. Na osnovu ovih bita ALU kontrolna jedinica formirat će sekvencu bita koja će predstavljati kontrolni signal ALUOp i naređivati ALU jedinici koju operaciju da uradi i koje signale da propusti na svoj izlaz. Za svrhe ove kontrolne jedinice potrebno je podsjetiti se prethodno analizirane ALU implementacije

sa slike 21 kako bi dizajn kontrolnih signala imao smisla.

11.1 Main kontrolni blok signali za dizajn

Op	Reg Dst	Reg Write	Ext Op	ALU Src	Beq	Bne	J	Mem Read	Mem Write	Mem toReg
R-tip	1 = Rd	1	x	0=BusB	0	0	0	0	0	0
addi	0 = Rt	1	1=sign	1=Imm	0	0	0	0	0	0
slti	0 = Rt	1	1=sign	1=Imm	0	0	0	0	0	0
andi	0 = Rt	1	0=zero	1=Imm	0	0	0	0	0	0
ori	0 = Rt	1	0=zero	1=Imm	0	0	0	0	0	0
xori	0 = Rt	1	0=zero	1=Imm	0	0	0	0	0	0
lw	0 = Rt	1	1=sign	1=Imm	0	0	0	1	0	1
sw	x	0	1=sign	1=Imm	0	0	0	0	1	x
beq	x	0	x	0=BusB	1	0	0	0	0	x
bne	x	0	x	0=BusB	0	1	0	0	0	x
j	x	0	x	x	0	0	1	0	0	x

Slika 41: Main kontrolni blok signali za dizajn

Napomena 11.1

Znak 'x' na gornjoj slici znači da nam vrijednost tog kontrolnog signala nije bitna jer nam za konkretnu instrukciju taj dio datapath-a nije relevantan. U suštini može biti bilo koja vrijednost (što će i biti jer ne možemo imat signal x) i koristi se za minimizaciju logike.

Kako smo kroz cijelu skriptu konstantno objašnjavali zašto pojedini signali trebaju biti setovani na određene vrijednosti, gornja slika bit će ostavljena bez komentara. Ova slika opisuje na koje vrijednosti treba da budu postavljeni određeni kontrolni signali zavisno od instrukcije.

Ovakva raspodjela dobiva se iz opcode polja instrukcije. Na primjer ako je ulaz u main kontrolni blok tokom izvršenja neke instrukcije bilo opcode polje od šest nula, znamo da se radi o R tipu instrukcije i na osnovu ove slike, a i svega ispričanog do sad, znamo koji output main kontrolnog bloka treba da bude. Za R tip instrukcija RegDst treba da bude postavljen na jedan jer na RW treba da bude propušten signal polja rd kao destinacijski registar kod R tipa. R instrukcije uvijek pišu u registar pa RegWrite za svaku treba da bude 1. ExtOp je x jer nam nije bitno šta ExtOp treba da bude. ALUSrc je postavljen na 0 jer kod R instrukcija ulaz u ALU treba da bude vrijednost pročitana sa BusB iz registra određenog sa RB. Branch i jump instrukcije nisu R tipa pa su kontrolni signali za ove jednaki nulama. Kod R instrukcija ne čitamo, ne pišemo i ne proslijeđujemo vrijednost iz memorije ka registar fajlu pa su ovi također jednaki nulama. Analogno tome opisali bismo i raspored kontrolnih signala za svaku sljedeću instrukciju na tabeli.

Ova slika predstavlja u stvari istinosnu tabelu na osnovu koje koristeći alat logisim i build circuit opciju (u logisimu možemo prekucati ovu tabelu a alat će nam na osnovu nje sam izraditi kompletno kolo) možemo kreirati kolo koje će biti naš main kontrolni blok.

11.2 ALU blok kontrolni signali za dizajn

Op	funct	ALUOp	4-bit Code
R-type	AND	AND	11 00
R-type	OR	OR	11 01
R-type	XOR	XOR	11 10
R-type	ADD	ADD	10 00
R-type	SUB	SUB	10 10
R-type	SLT	SLT	01 10
ADDI	X	ADD	10 00
SLTI	X	SLT	01 10
ANDI	X	AND	11 00
ORI	X	OR	11 01
XORI	X	XOR	11 10
LW	X	ADD	10 00
SW	X	ADD	10 00
BEQ	X	SUB	10 10
BNE	X	SUB	10 10
J	X	X	X

Slika 42: ALU blok kontrolni signali za dizajn

Napomena 11.2 Greška u materijalu

Gornja slika razlikuje se od one date na prezentacijama sa predavanja. Četverobitni kod treba da korespondira sa ALU implementacijom datom na početku u sekciji 5, što nije slučaj sa tabelom datom na predavanju. Ovdje je postavljena ispravljena tabela.

Gornja tabela predstavlja također istinosnu tabelu, gdje na osnovu opcode i func polja instrukcije kao ulaza, treba proizvesti 4-bitni kod kao output koji je **match** sa ALU implementacijom. Ovaj output predstavljat će kontrolni signal ALUOp koji će ulaziti u aritmetičko logičku jedinicu. Ukoliko se vratimo na sekciju 5 gdje smo prikazali internu implementaciju ALU bloka, poredeći 4-bitne kodove sa slikom ALU bloka možemo primjetiti vezu.

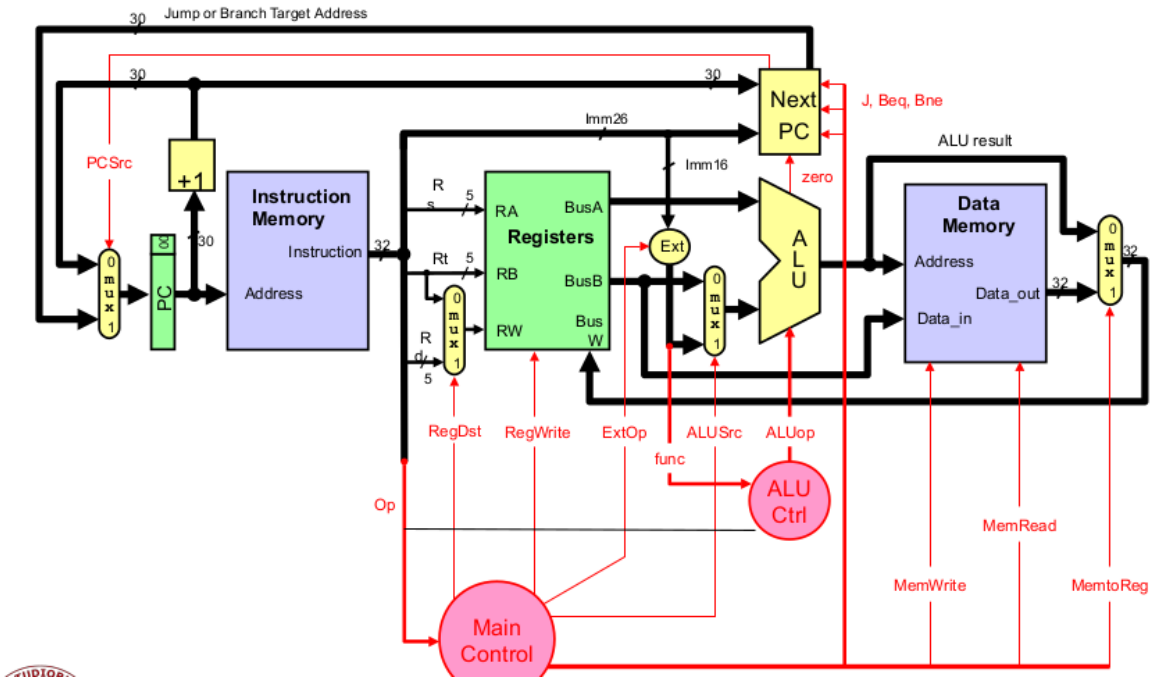
Gornja 2 bita koda predstavljaju ALU selection signal koji bira šta će biti izlaz iz ALU bloka (adder izlaz, shifter izlaz, logical unit izlaz ili slt izlaz), dok donja dva bita kontrolišu da li treba izvršiti logičku ili aritmetičku operaciju i konkrektno koju (add,sub, and, or, xor, nor). Za primjer instrukcije uzmimo `add` instrukciju. Ova instrukcija je R formata pa će opcode biti sve nule, main kontrolna jedinica će to očitati i postaviti kontronle signale koje kontroliše main na odgovarajuće vrijednosti. Iz func polja ove instrukcije izvući će se 6 bita `100000` i proslijediti ALU kontrolnom bloku. Kod instrukcije add trebamo izvršiti aritmetičku operaciju unutar ALU jedinice, i to sabiranja (ADD), a kao ALU rezultat trebamo proslijediti vrijednost sa ulaza 2 ALUSelection mulitplexera ako pratimo sliku 21. Dakle kako radimo ADD aritmetičku operaciju, prateći našu sliku ALU implementacije to će nam korespondirati sa bitima `00`. Kako rezultat ALU treba da bude rezultat aritmetičke operacije to prema našoj implementaciji korespondira sa bitima `10` koji će propagirati ulaz 2 multiplexera kao rezultat, što jeste signal dobiven iz addera (sabirača). Kako smo rekli da su nam gornja dva bita koda u stvari ALUSelection, a donja dva biraju aritmetičku/logičku operaciju to je naš 4-bitni kod ALUop za `add` instukciju `10 00`.

Ove 4-bitne kodove lako je izvući iz opcode i func polja specifične instrukcije. Naš procesor neće raditi shift operaciju pa gornji dio ALU zanemarujemo.

Napomena 11.3 Ostale instrukcije

Kao što vidimo naš procesor izvršavat će jako mali skup instrukcija, međutim naša implementacija treba da može fetchirati bilo koju sekvencu 32-bita koja predstavlja instrukciju. Ukoliko detektujemo da ta 32-bita ne predstavljaju instrukciju datu u gornjim tabelama, dovoljno je narediti procesor da uradi **nop** instrukciju, a za to je dovoljno postaviti kontrolne signale RegWrite, MemRead, MemWrite na 0 i tako neće doći do interne promjene stanja prilikom "izvršavanja" takvih instrukcija.

12 Jednociklusna implementacija CPU-a



Slika 43: Jednociklusna implementacija CPU-a

Za kraj detaljno objasnimo proces pri izvršenju instrukcije `add $1, $1, $1`.

Instrukcija kodirana u 32-bita -> `000000 00001 00001 00001 00000 100000`. Neka je ova instrukcija učitana na 0x0 adresi u **instruction memory** i neka je u $pc registru na početku clock ciklusa ove instrukcije vrijednost `0000 0000 0000 0000 0000 0000 0000 0000` koja korespondira sa tom adresom. Prvo što se dogodi jeste inkrement 30 bita iz pc registra.

Nova vrijednost je `00 0000 0000 0000 0000 0000 0000 0100`) i proslijeđuje se istovremeno na 0 ulaz muxa kod registra te u NextPC blok. Kako je instrukcija R tipa NextPC blok neće biti aktivan odnosno uz J, Beq, Bne = 0 PCSrc će biti 0 pa vrijednost dobivena u NextPC bloku neće biti propagirana u registar.

Instrukcija se dekodira i prosljeđuje na polja, kako je ovo R tip instrukcije RegDst signal dobiven iz Main Controla bit će postavljen na 1 jer će polje rd određivati registar za pisanje. Kako je naša instrukcija specifično napisana na poljima rs, rt, rd naći će se jednake vrijednosti bita `00001` ekstrahovane iz instrukcije. Interno u registar fajlu odabrat će se registar broj 1, te će biti aktivirani bufferi na izlazu ovog registra i u slučaju BusA i BusB izlaza pa će se upravo vrijednost pročitana iz ovog registra naći na oba bus-a. 6 najjačih bita opcodea bit će odvedeno main kontroli, a 6 najslabijih bita func polje bit će odvedeno ALU kontroli koja će generisati kontrolne signale. Donjih 26 bita će biti odvedeno cjelokupno NextPC bloku da izvrši proračun koji neće biti korišten, a donjih 16 od toga odvedeno na extender a zatim na ulaz broj 1 multiplexera ALUSrc koji za ovu instrukciju tu i ostaje i dalje se ne koristi. Donjih 6 od tih 16 bit će proslijeđeno ALU kontroli kao func polje kod R instrukcija. ALUSrc u ovome slučaju je jedan, jer kod add instrukcije drugi ulaz u ALU je vrijednost pročitana iz registra RB. Također vrijednost sa BusB bit će odvedena u Data_in ulaz data memorije ali će MemWrite biti

podešen na 0 za R tip instrukcije pa on tu ostaje i ne koristi se. ALU će obaviti sabiranje vrijednosti pri čemu će ALUop kontrolni signal dobiven iz ALU kontrolne jedinice biti `1000`. Interno unutar ALU jedinice postojat će svi signali ali će se koristi onaj koji se dobio kao rezultat sabirača što je određeno sa gornja dva bita ALUOp kontrolnog signala. Zero signal ovisi od vrijednosti koja se nalazi u registru 1 pa to nećemo komentarisati ali on neće imati učinak na NextPC jer se rezultat koji se dobije interno u ovom bloku neće zapisivati u $pc registar za slučaj `add` instrukcije. Izlaz iz ALU bit će proslijeđen na Address ulaz data memorije te istovremeno i na MemtoReg multiplexerov ulaz broj 0. Kako je ovo instrukcija koja ne pristupa memoriji MemWrite i MemRead bit će 0 i MemtoReg će biti postavljen na 0 jer se u add instrukciji ne pristupa memoriji i treba proslijediti ALU rezultat na BusW registar fajla. Tu će nakon okidanja opadajuće ivice clock signala u registar 1 biti upisana vrijednost koja je jednaka zbiru vrijednosti registra 1 sabrane sa same sobom, nakon toga kreće novi clock ciklus i izvršava se instrukcija koja se nalazi na adresi 0x4 u instruction memoriji.

Kontrolni signali za ovu instrukciju:

- RegDst = 1, jer rd polje treba biti na RW
- RegWrite = 1, jer R instrukcije zapisuju u registar fajl
- ExtOp = x, jer nas ne zanima šta će ekstender tačno uraditi jer njegov signal neće propagirati dalje
- ALUSrc = 0, jer drugi ulaz u ALU treba biti vrijednost registra biranog sa RB
- ALUop = 1000, ekstrahovano iz func polja, gornja dva 10 biraju ulaz broj 2 multiplexera da propagira kao ALU rezultat, donja dva biraju aritmetičku operaciju sabiranja unutar ALU
- J, Beq, Bne = 0, jer nije jump ili branch instrukcija
- PCSrc = 0, iz prethodna 3 signala prozilazi da će bit i on 0 jer ne trebamo rezultat NextPC upisati u registar $pc s obzirom da ne radimo jump ili branch instrukciju
- MemWrite, MemRead = 0, jer ne pristupamo memoriji
- MemtoReg = 0, jer vrijednost sa izlaza ALU treba propagirati na BusW registar fajla

Ovime smo došli do finalne forme jednociklusne implementacije naše centralne procesorske jedinice koja je bazirana na MIPS32 arhitekturi i MIPS ISA. Autor skripte ne garantuje za tačnost informacija datih u ovoj skripti.