4 DIZAJNIRANJE ARITMETIČKO-LOGIČKE JEDINICE

Aritmetičko-logička jedinica (ALU, odnosno engl. Arithmetic-Logic Unit) je dio centralne procesorske jedinice (CPU, odnosno eng. Central Processing Unit) računara, preciznije njenog Datapath-a. Prije nego započnemo dijzajniranje ALU, razmotrimo nekoliko činjenica neophodnih za njeno dizajniranje.

Reprezentacija negativnih brojnih veličina. U većini savremenih računarskih arhitektura, negativne brojne veličine reprezentiju se u dvojnom komplementu. Primijetimo da bi alternativu reprezentaciji u dvojnom komplementu mogla predstavljati "znak + apsolutna vrijednost" (eng. *sign + magnitude*) reprezemtacija. Medjutim, nekoliko je očitih prednosti reprezentacije u dvojnom komplementu, koje opredjeljuju njenu gotovo isključivu upotrebu:

- Iako bliža čovjekovom načinu zapisivanja negativnih dekadnih brojeva, znak + apsolutna vrijednost reprezentacija uključuje dvije nule: +0 i –0, dok reprezentacija u dvojnom komplementu uključuje samo jednu 0 i time omogućava zapisivanje jedne brojne veličine više u odnosu na atlternativnu reprezentaciju,
- Upotrebom reprezentacije negativnih brojeva u dvojnom komplementu dobija se nulta vrijednost prilikom sabiranja brojeva istih apsolutnih vrijednosti, a suprotnih znakova,
- Reprezentacija u dvojnom komplementu doprinosi redukovanju hardverske složenosti i ostalih performansi povezanih sa njom (gabariti, potrošnja energije, cijena, ...),
- Jednostavno i veoma jasno uočava se razlika u reprezentaciji pozitivnih i negativnih brojeva. Pozitivni brojevi u svom zapisu uključuju vodeću 0-u (MSB bit kod pozitivnih brojeva je 0-a), dok negativni brojevi u svom zapisu uključuju vodeću 1-cu (MSB bit kod negativnih brojeva je 1-ca). Shodno tome, hardware-u je neophodno da testira samo MSB (sign bit, odnosno znak bit) brojeva zapisanih u dvojnom komplementu u cilju detekcije da li je riječ o pozitivnom ili o negativnom broju.

Označene vs NE-označene brojne veličine. Sve do sada razmatrane instrukcije funkcionišu sa označenim brojnim veličinama. Medjutim, postavlja se pitanje funkcionisanja sa neoznačenim brojnim veličinama. Primijetimo najprije da se neoznačene brojne veličine upotrebljavaju u jednom značajnom segmentu funkcionisanja računara – kod ***adresne aritmetike*** (negativne adrese su nepostojeće i, shodno tome, rad sa negativnim adresama nema smisla). Uočimo razliku izmedju rada sa označenim i neoznačenim brojnim veličinama na primjeru poredjenja:

- Ukoliko radimo sa **označenim brojnim veličinama**, prvi/vodeći bit u binarnom zapisu broja upotrebljava se za reprezentaciju znaka broja (vodeća 1-ca označava negativnu brojnu vrijednost, dok vodeća 0-a označava pozitivnu brojnu vrijednost). Shodno tome, **brojevi čiji zapisi započinju binarnom 1-om uvijek su manji od brojeva čiji zapisi započinju binarnom 0-om**. Naime, brojevi čiji binarni zapis započinje 1-om predstavljaju negativne brojne vrijednosti, dok brojevi čiji binarni zapis započinje 0-om predstavljaju pozitivne brojne vrijednosti, a sve pozitivne brojne vrijednosti su veće od negativnih brojnih vrijednosti.
- Ukoliko radimo sa **neoznačenim brojnim veličinama**, prvi/vodeći bit u binarnom zapisu broja doprinosi njegovoj vrijednosti na isti način kao ostali bitovi upotrijebljeni u zapisu broja. Shodno tome, **brojevi čiji zapisi započinju binarnom 1-om uvijek su veći od brojeva čiji**

zapisi započinju binarnom 0-om, dok god je riječ o brojevima istoga znaka (neoznačenim brojevima).

Primijetimo, za istu kobinaciju 1-ca i 0-a kod oba operanda koja se porede, u slučaju označenih brojnih veličina može da važi odnos ***manji od***, dok u slučaju neoznačenih brojnih veličina može da označava odnos ***veći od***. Da bi se prevazišla ova, uslovno rečeno, kontradiktornost, pored već razmatranih instrukcija koje vrše poredjenja označenih brojnih veličina/operanada, uvode se ***unsigned oblici istih instrukcija*** koje takodje vrše poredjenja, ali neoznačenih brojnih veličina/operanada. Preciznije,

- Instrukcije *slt* i *slti* rade sa označenom brojnim veličinama,
- Unsigned oblici ovih instrukcija, *slt unsigned* i *slti unsigned*, odnosno, u simboličkoj formi, *sltu* i *sltiu* instrukcije upotrebljavaju se za poredjenje neoznačenih brojnih veličina. Notirajmo da ove instrukcije imaju istu sintaksu kao instrukcije *slt* i *slti* i da poredjenja vrše na potpuno isti način, samo veličine koje porede tretiraju neoznačenim brojnim veličinama, umjesto označenim brojnim veličinama (kao što to rade instrukcije *slt* i *slti*).

Kopiranje znaka broja (eng. sign extend). Prilikom implementacije data-transfer (*lw* i *sw*) i immediate instrukcija, kao i prilikom implementacije PC-relativnog adresiranja, sabira se 16-bitni sadržaj addess/immediate polja ovih instrukcija (poglavlja 3.2.2, 3.6 i slika 7 iz poglavlja 3.7) sa 32-bitnim sadržajem registra (registra koji se adresira rs poljem *lw*, *sw* i immediate instrukcija, odnosno PC registra). Sa druge strane, pouzdano sabiranje operanada može se izvršiti samo ukoliko su operandi jednakih dužina. Stoga je, prije izvršavanja željene operacije, dužine operanada neophodno izjednačiti i to dopunjavanjem 16-bitnog operanda do 32-bitne dužine, naravno ne mijenjajući time numeričku vrijednost operanda. Sada se postavlja pitanje: kako dopuniti 16-bitni operand do 32-bitne dužine, jednovremeno zadržavajući neizmijenjenu njegovu numeričku vrijednost?

Kod označenih brojeva/operanada, željena akcija postiže se kopiranjem znaka operanda. Pojasnimo ovaj navod:

- U slučaju pozitivnog 16-bitnog broja/operanda, njegov vodeći bit je 0, tako da popunjavanje nedostajućih bitova (od 16-bitne do 32-bitne dužine broja) sa 16 0-a sa lijeve strane 16-bitnog broja/operanda ne mijenja njegovu vrijednost (dodavanje proizvoljnog broja 0-a sa lijeve strane broja, ne mijenja vrijednost broja!!).
- U slučaju negativnog 16-bitnog broja/operanda, njegov vodeći bit je 1, a broj/operand je reprezentovan u dvojnom koplementu. Popunjavanjem nedostajućih bitova (od 16-bitne do 32-bitne dužine broja) sa 16 1-ca sa lijeve strane 16-bitnog broja/operanda dobija se broj sa 17 vodećih 1-ca. Primijetimo da je na ovaj način dobijeni 32-bitni broj i dalje negativan. U cilju tumačenja njegove vrijednosti, potrebno je pronaći najprije jedinični, a potom i dvojni komplement broja. Kako se jedinični komplement broja dobija zamjenom 0-a sa 1-ma i 1-ca sa 0-ma, a dvojni komplement – dodavanjem 1 na jedinični komplement broja, jednostavno se može žaključiti da se nakon pronalaženja dvojnog komplementa dobija 32-broj sa 16 vodećih 0-a. Drugim riječima, kopiranjem vodeće 1-ce (znaka) 16-bitnog negativnog broja, sa lijeve strane broja, do 32-bitne dužine ne mijenja se vrijednost broja (nakon komplementiranja, kopiranje znaka negativnog broja sa lijeve strane 16-bitnog broja do njegove 32-bitne dužine, odgovara dodavanju 16 0-a sa lijeve strane broja, čime se ne ne mijenja vrijednost broja!!).

NAPOMENA: Neoznačeni brojevi/operandi takodje mogu biti 16-bitni i može se zahtijevati njihovo produžavanje do 32-bitne dužine (ovo je slučaj, kao što ćemo vidjeti u nastavku, sa immediate oblicima logičkih instrukcija), naravno uz dodatni zahtijev da se ovom operacijom ne promijeni vrijednost ovih brojeva. Medjutim, neoznačeni broj/operand ne posjeduju znak, tako da se ne može kopirati znak broja, već se nedostajući bitovi (od 16-bitne do 32-bitne dužine broja) popunjavaju sa 16 0-a sa lijeve strane 16-bitnog broja/operanda. Ova operacija naziva se ***kopiranjem nule*** (eng. *zero extend*) i u literaturi ova operacija obično se naziva *zero extend*.

Prekoračenje dozvoljenog opsega riječi/registara (overflow). U MIPS arhitekturama, registri su 32-bitne dužine. Stoga, u njima se mogu smještati brojne veličine iz ograničenog opsega. Na primjer, u ovim registrima mogu biti smješteni označeni cijeli brojevi iz dekadnog opsega $[(-2^{31})_{(10)} \div (2^{31}-1)_{(10)}]$.

Tabela 1. Znaci operanada i znak rezultata dobijenih prilikom izvršavanja naznačenih aritmetičkih operacija i prilikom dolaska do prekoračenja dozvoljenog opsega registara.

Operacija	*Operand A*	*Operand B*	*Rezultat*
A + B	≥0 (znak/s bit=0)	≥0 (znak/s bit=0)	<0 (znak/s bit=1)
A + B	<0 (znak/s bit=1)	<0 (znak/s bit=1)	≥0 (znak/s bit=0)
A – B	≥0 (znak/s bit=0)	<0 (znak/s bit=1)	<0 (znak/s bit=1)
A – B	<0 (znak/s bit=1)	≥0 (znak/s bit=0)	≥0 (znak/s bit=0)

Izvršavanjem oderedjenih matematičkih operacija nad brojevima iz ovog opsega, može doći do situacije da odgovarajući rezultat ne pripada opsegu brojeva koji mogu biti zapisani u registrima iste dužine[1], već da je za zapisivanje njegove vrijednosti potreban dodatni bit (odnosno znak bit – MSB bit koji se upotrebljava za zapisivanje znaka označenih brojeva i koji ne doprinosi numeričkoj vrijednosti broja). Pojednostavljeno, dolazi do prenamjene znak bita – njegove upotrebe za zapisivanje bita koji doprinosi vrijednosti broja, što rezultira rezultatom neodgovarajućeg znaka, kao što je prikazano u Tabela 1. Ukoliko do prekoračenja dolazi sa donje strane dozvoljenog opsega registra, ovaj efekat se naziva *underflow*, a ukoliko do prekoračenja opsega dolazi sa gornje strane (kao u primjeru navedenom u fusnoti 1), efekat se naziva *overflow*. Ipak, najčešće se u literaturi oba pojma nazivaju istom riječju, i to *overflow*, pa ćemo i mi prekoračenje dozvoljenog opsega registara nazivati isključivo sa overlow.

Posljedica efekata overflow-a je greška, sa kojom računar ne smije nastaviti da funkcioniše. Cilj je detekvovati pojavu overfolw-a i nakon toga reagovati. Detekcija overflow-a može se izvršiti poredjenjem znak bitova operanada i dobijenog rezultata, kako je to predstavljeno u Tabeli 1.

Primijetimo da do overflow-a dolazi prilikom implementacije operacije sabiranja operanada istog znaka i dobijanjem rezultata suprotnog znaka od znakova operanada koji se sabiraju, kao i prilikom implementacije operacije oduzimanja, ali operanada suprotnih znakova i dobijanja rezultata čiji znak je suprotan od znaka umanjenika. Ipak, ovo razmatranje može se značajno pojednostaviti ukoliko se shvati da ono odgovora medjusobnom odnosu ulaznog i izlaznog prenosa na MSB rezultata. Preciznije, ukoliko ulazni prenos na MSB rezultata ($CarryIn_{MSB}$) i izlazni prenos sa MSB rezultata ($CarryOut_{MSB}$) imaju različite vrijednosti, konstatuje se da je došlo do overflow-a, a ukoliko je $CarryIn_{MSB}$=$CarryIn_{MSB}$, do overflow-a nije došlo. Drugim riječima, pojava overflow-a odgovara, i uobičajeno se detektuje, pronalaženjem logičke ekskluzivno-ILI operacije izmedju ulaznog i izlaznog prenosa na MSB bitu,

$$Overflow=CarryIn_{MSB}\oplus CarryOut_{MSB}. \quad (1)$$

Sada se postavlja pitanje: što računar radi nakon detekcije overlow-a?

U odgovoru na ovo pitanje, neophodno je naglasiti da je kod ranijih računara overflow uzrokovao prekid u funkcionisanju, odnosno njegovo resetovanje. Ipak, savremene arhitekture pokušavaju da prevazidju overflow i, nakon toga, da nastave sa normalnim funkcionisanjem. U tom cilju, odgovor MIPS arhitektura na overflow je neplanirani poziv *exception rutine/potprograma* odgovarauće detektovanom problemu. Drugim riječima, MIPS arhitekture overflow smatraju *izuzetkom* (eng. *exception*, koji odgovara pojmu *interrupt*–prekid, upotrebljavanom kod mnogih drugih savremenih računarskih arhitektura), koji se pozvanom rutinom/potprogramom nastoji prevazići.

Adresa instrukcije čije izvršavanje je uzrokovalo overflow (tzv. *interrupt adresa*, eng. *interrupt address*) čuva se u posebnom (sistemskom) registru, nazvanom EPC (*Exception PC* registar),

[1] Na primjer, sabiranjem označenih cijelih brojeva $(2^{31}-1)_{(10)}$ i $2_{(10)}$, koji oba mogu biti zapisani u 32-bitnim riječima/registrima, dobija se označeni cijeli broj/rezultat $(2^{31}+1)_{(10)}$, koji ne može biti zapisan u 32-bitnoj riječi – izlazi iz dozvoljenog opsega brojeva koji mogu biti zapisani u 32-bitnoj riječi.

u želji da se, nakon mogućeg prevazilaženja problema, vratimo na izvršavanje programa i to od iste instrukcije čijim izvršavanjem je došlo do overflow-a. Izvršavanje se, potom, prepušta exception rutini/potprogramu. Ova rutina nastoji prevazići neželjenu situaciju korekcijom koda instrukcije, čije izvršavanje uzrokuje overflow, i to korekcijom u unsigned oblik iste instrukcije. Na taj način, znak bit formalno se proglašava bitom koji doprinosi numeričkoj vrijednosti rezultata, opseg zapisivanih neoznačenih brojeva, shodno tome, suštinski postaje dvostruko širi u poredjenju sa slučajem pozitivnih označenih brojeva, a sve u nadi mogućeg prevazilaženja problema. Ipak, ne uspijeva se uvijek prevazići problem. Tada, prekinuti program nije moguće nastaviti i njegovo izvršavanje se prekida.

Ukoliko se problem uspije prevazići, računar se vraća na izvršavanje programa i to od instrukcije čije izvršavanje je uzrokovalo overflow. U tom cilju, interrupt adresu (adresu instrukcije koja je uzrokovala overflow), neophodno je premjestiti/iskopirati iz EPC registra u jedan od 32 procesorska registra ($0–$31) kako bi se na kraju exception rutine/potprograma, uz posredovanje *jr* instrukcije, računar mogao vratiti na izvršavanje prekinutog programa. Premještanje/kopiranje sadržaja EPC registra u neki od procesorskih registara obavlja se uz pomoć *mfc0* (*Move From Coprocessor Reg.*) instrukcije. Ipak, postavlja se pitanje da li sadržaj EPC registra može biti premješten/kopiran u bilo koji od procesorskih registara i, ukoliko ne može, u koji registre se njegov sadržaj može kopirati?

Prije nego odgovorimo na ovo pitanje, podsjetimo da prilikom pozivanja i izvršavanja rutine/procedure/potprograma moraju biti sačuvani (na stack-u) sadržaji svih registara koje upotrebljava pozivajuća procedura (u ovom slučaju program prekinut overflow-om), a čiji sadržaji se mijenjaju u pozvanoj proceduri/rutini/potprogramu. Ukoliko bi se sadržaj EPC registra mogao kopirati u bilo koji procesorski registar, ozbiljno bi se rizikovalo da bi čuvanjem i povratkom sadržaja registara sa stack-a adresa iskopirana iz EPC registra u proizvoljni procesorski registar bila prebrisana, te da bi bio onemogućen povratak na prekinuti program bez obzira što se prevazišao problem zbog kog je došlo do overflow-a. Uz to, ozbiljno bi se rizikovalo da prilikom povratka na izvršavanje prekinutog programa, interrupt adresa prebriše sadržaj registara koji nadalje treba da upotrebljava prekinuta procedura. Takodje, sve i ukoliko se obezbijedi neprebrisavanje sadržaja registra u kome bi bila sačuvana adresa puvratka (interrupt adresa) od strane povratnih sadržaja registara sa stack-a, moramo se obezbijediti i od upotrebe istog registra (sa sačuvanom interrupt adresom) od strane programa u medjuvremenu. Da bi se sve navedene nepoželjne mogućnosti prevazišle, registri $26 i $27 ostavljaju se na upotrebu[2] isključivo operativnom sistemu, tako da se sadržaj EPC registra kopira u jedan od dva registra. Drugim riječima, u slučaju exceptiona, u ovim registrima se ne može čuvati, niti kopirati (povratiti) sa stack-a sadržaj bilo kog registra upotrebljavanog od strane exception rutine/potprograma, niti oni mogu biti upotrebljavani od strane prekinutog programa ili exeception rutine u medjuvremenu, osim na kraju exception rutine – prilikom povratka na prekinuti program.

NAPOMENA: Overlow se detektuje samo kod označenih brojnih veličina. Kod neoznačenih brojeva, overlow se ne detektuje (pretpostavlja se da se overflow u slučaju neoznačenih brojeva ne dešava). Naravno, ne zato što ne postoje dovoljno veliki neoznačeni brojevi koji prevazilaze opseg registara ograničene dužine (kod MIPS arhitektura 32 bita), već zato što se neoznačena brojevi upotrebljavaju uglavnom kod adresiranja i adresne aritnetike, tako da, za razliku od prirodnih brojeva, oni imaju ograničene limite za memorijske čipove konačnih opsega (znaju se memorijski opsezi koji mogu biti adresirani i ne pretpostavlja se da bi se moglo poseći za adresiranjem izvan memorijskih opsega). Notirajmo, kod MIPS arhitekture, adresira se operativna memorija računara kapaciteta od 2^{32} byte-a.

Iz prethodno navedenih razloga, kod nekih instrukcija provjerava se da li je došlo do overflow-a ili ne, a kod unsigned oblika istih instrukcija se ne provjerava. Drugim riječima,

- Prilikom izvršavanja instrukcija *add*, *sub* i *addi* vrši se detekcija overflow-a,

[2] Compiler, odnosno programeri u assembleru, uzdržavaju se od upotrebe registara $26 i $27. Na taj način, omogućava se upotreba ovih registara od strane operativnog sistema, na isti način na koji se od compilera- i programera u assembleru rezerviše registar $1 za isključivu upotrebu od strane assembler-a (kao privremenog registra assembler-a).

- Prilikom izvršavanja unsigned oblika istih instrukcija, odnosno instrukcija *addu*, *subu* i *addiu* ne detektuje se overflow. Shodno tome, prilikom izvršavanja exception rutine, overlow nastao izvršavanjem operacija sabiranja i oduzimanja nad označenim operandima pokušava se prevazići zamjenom/korekcijom koda instrukcija *add*/*sub*/*addi* njihovim unsigned oblicima *addu*/*subu*/*addiu*.

Logičke operacije. Osim rada sa memorijskim ili registarskim riječima, veoma često se zahtijeva rad sa poljima bitova ili samo pojedinim bitovima. Izdvajanje polja bitova ili samo pojedinih bitova iz riječi obavlja se formiranjem maski i upotrebom logičkih operacija. Instrukcije koje zahtijevaju izvršavanje logičkih operacija pripadaju instrukcijama R-tipa. Najpoznatije su instrukcije *shift*-ovanja podataka u lijevu i u desnu stranu, logička AND i logička OR instrukcija.

Operacija logičkog shiftovanja. Operacijom shift-ovanja pomjera se sadržaj riječi za zadati broj bitova u lijevu ili u desnu stranu, dok se bitovi napušteni pomjeranjem popunjavaju 0-ma. Razlikuju se 2 instrukcije shift-ovanja:

- *sll* (eng. Shift Left Logical) instrukcija, kojom se sadržaj riječi/registra pomjera ulijevo,
- *srl* (eng. Shift Right Logical) instrukcija, kojom se sadržaj riječi/registra pomjera udesno.

Simbolički kod zapisivanja ovih instrukcija je sljedeći:

```
sll     $x, $y, 8       # $x←$y<<8
srl     $x, $y, 8       # $x←$y>>8
```

gdje je epilog/rezultat ovih instrukcija zapisan u komentarima svake od instrukcija, dok su << i >> simboličke oznake za pomjeranje u lijevu i u desnu stranu, respektivno. Kao što je već navedeno, *sll* i *srl* pripadaju R-tipu instrukcija, tako da instrukciji *sll* $x, $y, 8 odgovara sljedeći mašinski kod:

Polje:	op	rs	rt	rd	shamt	Funct
Br. bitova:	6	5	5	5	5	6
Sadržaj:	0	0	y	x	8	0, 2

Slika 1. Format mašinskog zapisivanja instrukcija *sll*/*srl*.

Primijetimo da se obilježje y operanda/registra, čiji sadržaj se pomjera u lijevu ili u desnu stranu, upisuje u polje rt, u rd polju se upisuje obilježje registra u koji se smješta rezultat shift-ovanja, dok se "količina" shift-ovanja/pomjeranja upisuje u polje shamt (eng. *shift amount*). Opcode (op) polje je 0, kao kod svih instrukcija R-tipa, dok se obilježjima $0_{(10)}$, odnosno $2_{(10)}$, zapisanim u funct polju, definišu operacije logičkog pomjeranja u lijevu i u desnu stranu, respektivno.

Logičke operacije AND i OR. Logičke AND i OR operacije izvršavaju se na pojedinačnim bitovima operanada nezavisno od izvršavanja na ostalim bitovima operanada. Stoga se kaže da AND i OR bit-by-bit operacije. Definišu se na sljegeći dobro poznat način:

```
0 AND 0 = 0          0 OR 0 = 0
0 AND 1 = 0          0 OR 1 = 1
1 AND 0 = 0          1 OR 0 = 1
1 AND 1 = 1          1 OR 1 = 1
```

Primijetimo da se logička AND operacija jednoznačno definiše za logičku 1-cu (samo se logička 1-ca jednoznačno dobija – kada su svi ulazi na nivou logičke 1-ce), dok se logička OR operacija jednoznačno definiše za logičku 0-u. Logičke AND i OR operacije hardware-ski se implementiraju logičkim I i ILI kolima. AND operacija se upotrebljava za izolovanje polja riječi. U tom cilju, najprije se kreira maska operand (bitovi ovog operanda koji odgovaraju poljima riječi koja trebaju da budu izolovana postavljaju se na 1, a ostali bitovi na 0), a nakon implementacije *and* instrukcije, odgovarajuća polja zadate riječi (drugog operanda) biće izolovana.

and i *or* instrukcije su takodje instrukcije R-tipa. Simbolički kod njihovog zapisivanja odgovara instrukcijama *add*/*sub*/*mult*/*slt*,

```
and/or          $x, $y, $z      # $x=$y&$z (za and) ($x=$y|$z (za or))
```

Epilog/rezultat izvršavanja ovih instrukcija zapisan je u komentaru, gdje je sa & označena simbolička oznaka operacije AND, a sa | simbolička oznaka operacije OR. Mašinski kod zapisivanja ovih istrukcija odgovara mašinskom kodu zapisivanja aritmetičkih instrukcija *add*/*sub*/*slt*,

Polje:	op	rs	rt	rd	shamt	funct
Br. bitova:	6	5	5	5	5	6
Sadržaj:	0	y	z	x	0	36, 37

Slika 2. Format mašinskog zapisivanja instrukcija *and*/*or*, gdje je funct=$36_{(10)}$ za *and* instrukciju, dok je funct=$37_{(10)}$ za *or* instrukciju.

Logičke immediate instrukcije *andi* i *ori*. Analogno aritmetičkim instrukcijama, logičke instrukcije takodje podržavaju rad sa trenutnim/immediate/konstantnim operandima, uz ograničenje da se konstantni operandi čuvaju u samoj instrukciji *andi*/*ori*. Uz to, assemblerski i mašinski kod *andi*/*ori* instrukcija apsolutno odgovara asemblerskom i mašinskom kodu zapisivanja aritmetičkih immediate instrukcija *addi*/*muli*/*slti* i, stoga, oni ovdje neće biti ponavljani.

VAŽNO ZAPAŽANJE: Logičke operacije su bit-by-bit, te stoga njihov rad sa označenim brojnim vrijednostima/konstantama nema smisla. Shodno tome, trenutni/immediate/konstantni operandi, koji se upotrebljavaju kod *andi*/*ori* instrukcija, tretiraju se neoznačenim cijelim brojevima i njihovo dopunjavanje do 32-bitne dužine vrši se kopiranjem nule (ranije razmatrana zero extend rutina kopiranja nule sa lijeve strane 16-bitne konstante do 32-bitne dužine). Pored toga, kreiranje velike konstante (veće od veličine 2^{16}), razmatrano ranije u poglavlju 3.6, obavlja se parom instrukcija *lui* i *ori*, umjesto para instrukcija *lui* i *addi*, kao što je to ranije sugerisano u 3.6. Naime, prilikom implementiranja instrukcije *ori*, 16-bitna konstanta dopunjava se 0-ma (sa svoje lijeve strane) do 32-bitne dužine, dok se prilikom implementiranja instrukcije *addi*, 16-bitna konstanta dopunjava (sa svoje lijeve strane) kopiranjem znaka broja za koji se, u tom slučaju, mora paziti da ne bude negativan (ukoliko se dopunjava negativni 16-bitni broj, on se dopunjava sa 16 1-a i time šteti viših 16 bitova velike konstante, koja je unaprijed kreirana implementiranjem naredbe *lui* – vidjeti poglavlje 3.6).

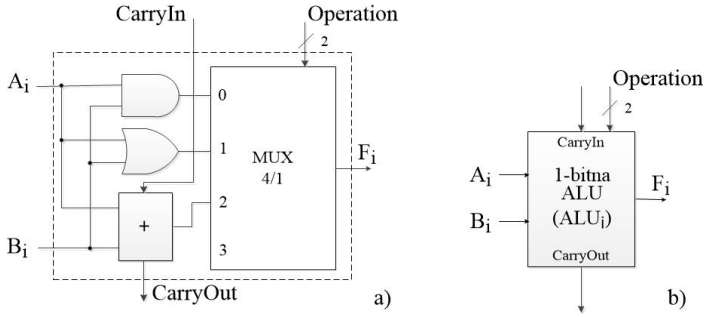
4.1 DIZAJNIRANJE ALU

Aritmetičko-logička jedinica (ALU) je komponenta računara (njegovog procesora, odnosno datapath-a) u kojoj se obavljaju aritmetičke i logičke operacije zahtijevane od strane izvršavanih instrukcija. Stoga, ALU predstavlja moć računara da izvrši zahtijevane operacije. ALU će u nastavku biti dizajnirana tako da obezbijedi izvršavanje operacija koje mogu biti zahtijevane od osnovnih aritmetičkih instrukcija *add* i *sub*, osnovnih logičkih instrukcija *and* i *or*, ali i od instrukcija koje razlikuju računar od običnog kalkulatora (instrukcija uslovnog skoka/grananja *beq*/*bne* i od instrukcije koja omogućava relativna poredjenja *slt*). Primijetimo da ove instrukcije predstavljaju reprezentativni skup instrukcija koje obezbjedjuju izvršavanja svih akcija na računaru.

Kako su u razmatranoj MIPS arhitekturi podaci/riječi/operandi nad kojima se izvršavaju operacije 32-bitni, potrebno je dizajnirati 32-bitna ALU. Ona će biti kreirana medjusobnim povezivanje 32 1-bitne ALU. Stoga će najprije biti dizajnirana 1-bitna ALU, a potom i 32-bitna ALU i to do posljednjeg detalja (upotrebom osnovnih logičkih kola).

4.1.1 1-bitna ALU

Prilikom kreiranja 1-bitne ALU, poći ćemo najprije od osnovnih logičkih operacija AND i OR i od operacije sabiranja. Logičke operacije AND i OR, kao bit-by-bit operacije, veoma je jednostavno implementirati uz pomoć jednog I i jednog ILI kola, pošto se ove operacije direktno implementiraju na svaki pojedinačni bit operanada. Na jedan od dva ulaza ovih kola dovodi se i-ti (i=0,...,31) bit jednog od operanada, a na njihove druge ulaze odgovarajući bit drugog operanda, slika 3. Sa druge strane, operacija sabiranja zahtijeva implementaciju potpunog 1-bitnog sabirača. Slično logičkim operacijama, ulazi potpunog 1-bitnog sabirača su odgovarajući bitovi operanada, ali i ulazni prenos (CarryIn) sa prethodnog stepena višebitnog sabirača (potpunog 1-bitnog sabirača sa prethodnog stepena operanada). Za razliku od logičkih I i ILI kola koji imaju izlaz za rezultat operacije koju



Slika 3. 1-bitna ALU koja omogućava obavljanje operacija AND, OR i 1-bitnog sabiranja. a) Logička struktura, b) Šematski prikaz.

obavljaju, potpuni 1-bitni sabirač ima dva izlaza – rezultujući izlaz S i izlazni prenos (CarryOut) za sljedeći stepen višebitnog sabirača, slika 3.

Kao funkcionalna jedinica, 1-bitna ALU jednovremeno izvršava sve one operacije koje izvršavaju njeni sastavni elementi, ali kao svoj rezultat prezentira rezultat samo jednog svog elementa. Drugim riječima, u istom trenutku logičko I kolo, logičko ILI kolo i sabirač obavljaju operacije kojima su namijenjeni (AND, OR i operaciju sabiranja, respektivno). Medjutim, 1-bitna ALU na izlazu, kao svoj rezultat prezentira rezultat samo jedne od ovih operacija. Izlazni multipleksor MUX 4/1, svojim selekcionim ulazima Operation, vrši izbor operacije koji će 1-bitna ALU prezentirati kao rezulta i to postavljanjem odgovarajućih Operation bitova (2 Operation bita za 3 ulaza multipleksora MUX 4/1),

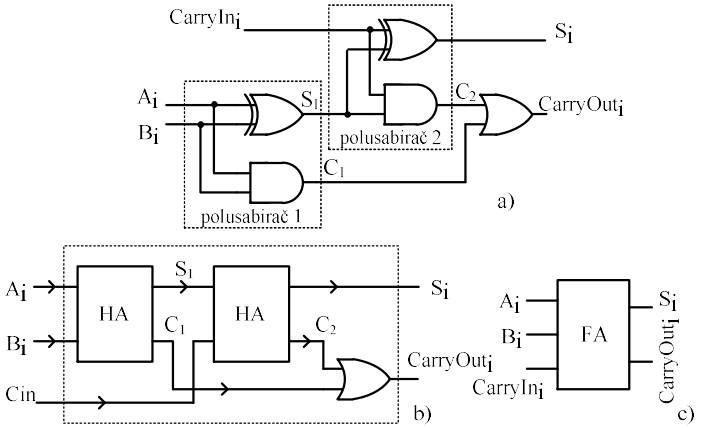
Tabela 2. Rezultat na izlazu jednobitne ALU zavisan od selekcionih Operation bitova izlaznog multipleksora MUX 4/1.

Operation	Funkcija 1-bitne ALU
00	$F_i = A_i \wedge B_i$
01	$F_i = A_i \vee B_i$
10	$F_i = A_i + B_i$

Primijetimo da se selekcioni ulazi izlaznog multipleksora MUX 4/1 nazivaju Operation, jer definišu operaciju koju 1-bitna ALU prezentira kao svoj rezultat, dok se izlaz ovog multipleksora označava sa F_i, jer predstavlja rezulujući izlaz (rezultat izvršene funkcije) 1-bitne ALU. Primijetimo takodje da su samo prva 3 ulaza izlaznog multipleksora MUX 4/1 upotrijebljena, te da se selektiraju samo ovi ulazi selekcionim Operation bitovima, a da u tabeli 2 nije predvidjeno selektiranje četvrtog ulaza ovog multipleksora (jednostavno nije upotrijebljen). Ipak, mi smo tek na početku dizajniranja ALU. Dodavanjem novih funkcija 1-bitnoj ALU, broj aktivnih ulaza izlaznog multipleksora MUX 4/1 će se shodno tome uvećavati.

Na početku razmatranja o ALU, naveli smo da će ona biti dizajnirana do nivoa osnovnih logičkih kola. Ipak, na slici 3, potpuni 1-bitni sabirač predstavljen je u blok-šematskoj reprezentaciji. Da ne bismo ostali na blok-šematskoj reprezentaciji potpunog 1-bitnog sabirača, dizajnirajmo i njega isključivo upotrebom logičkih kola. U tom cilju, podjimo od tabelarnog opisa njegovog funkcionisanja, Tabela 3. Iz funkcionalne tabele 3 neposredno slijede logički zapisi rezultujućih izlaza S_i i $CarryOut_i$ potpunog 1-bitnog sabirača:

$$S_i = \overline{A_i} \cdot \overline{B_i} \cdot CarryIn_i + \overline{A_i} \cdot B_i \cdot \overline{CarryIn_i} + A_i \cdot \overline{B_i} \cdot \overline{CarryIn_i} + A_i \cdot B_i \cdot CarryIn_i =$$
$$= \overline{A_i} \cdot (\overline{B_i} \cdot CarryIn_i + B_i \cdot \overline{CarryIn_i}) + A_i \cdot (\overline{B_i} \cdot \overline{CarryIn_i} + B_i \cdot CarryIn_i) =$$
$$= \overline{A_i} \cdot (B_i \oplus CarryIn_i) + A_i \cdot \overline{(B_i \oplus CarryIn_i)} = A_i \oplus B_i \oplus CarryIn_i$$



Slika 4. Potpuni 1-bitni sabirač (FA). a) Dizajniran upotrebom osnovnih logičkih kola, b) Pojednostavljena implementacija, c) Šematska oznaka. Na slici, HA označava 1-bitni polusabirač (1-bitni sabirač na nivou sabiranja isključivo odgovarajućih bitova ulaznih sabiraka, a bez uzimanja u obzir ulaznog prenosa CarryIn sa prethonog stepena).

Tabela 3. Funkcionalna tabela potpunog 1-bitnog sabirača.

A_i	B_i	$CarryIn_i$	S_i	$CarryOut_i$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

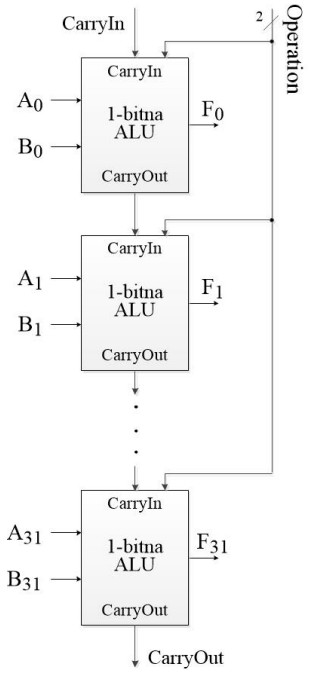
$$\begin{aligned} CarryOut_i &= \overline{A_i}\cdot B_i\cdot CarryIn_i + A_i\cdot\overline{B_i}\cdot CarryIn_i + A_i\cdot B_i\cdot\overline{CarryIn_i} + A_i\cdot B_i\cdot CarryIn_i = \\ &= (\overline{A_i}\cdot B_i + A_i\cdot\overline{B_i})\cdot CarryIn_i + A_i\cdot B_i\cdot(\overline{CarryIn_i} + CarryIn_i) = \\ &= (A_i\oplus B_i)\cdot CarryIn_i + A_i\cdot B_i. \end{aligned}$$

Implementacijom posljednja dva izraza upotrebom osnovnih logičkih kola dolazi se do realizacije 1-bitnog potpunog binarnog sabirača prikazanog na slici 4 a), čija je pojednostavljena implementacija (dobijena nakon uočavanja 1-bitnih polusabirača (HA)) prikazana na slici 4 b), dok je zamjenska šema, analogna onoj upotrijebljenoj na slici 3, prikazana na slici 4 c).

4.1.2 32-bitna ALU

Kombinovanjem 32 1-bitne ALU sa slike 3, odgovarajućim povezivanjem njihovih kontrolnih signala Operation, te njihovih ulaznih i izlaznih prenosa $CarryIn_i$ i $CarryOut_i$ (i=0,1,...,31) dobija se 32-bitna ALU prikazana na slici 5. Primijetimo da je 32-bitni sabirač, kao element 32-bitne ALU, realizovan na način da izlazni prenos jednog njegovog stepena ($CarryOut_i$, i=0,1,...,30) predstavlja ulazni prenos prvog sljedećeg višeg stepena ($CarryIn_i$, i=1,2,...,31), da ulazni prenos na LSB bitu ($CarryIn_0$) odgovara ulaznom prenosu 32-bitnog sabirača, CarryIn=$CarryIn_0$, a da izlazni prenos na MSB bitu ($CarryOut_{31}$) odgovara izlaznom prenosu 32-bitnog sabirača, CarryOut=$CarryOut_{31}$. Notirajmo da se ovaj sabirač u litelatiru naziva *ripple-carry adder*.

Podsjetimo, takodje, da Operation kontrolni signali upravljaju funkcionisanjem 1-bitnih ALU, Tabela 2. Oni stoga moraju biti zajednički kontrolni signali za sve upotrijebljene 1-bitne ALU, kao što je prikazano na slici 5. Na taj način obezbjedjuje se da sve upotrijebljene 1-bitne ALU kreiraju svoje



Slika 5. 32-bitna ALU dizajniranja upotrebom 32 1-bitne ALU sa slike 3.

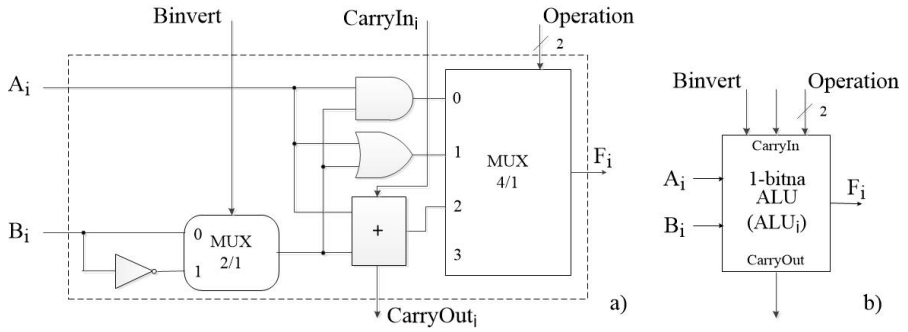
rezultujuće bitove nakon izvršavanje jedne te iste operacije (AND, OR ili operacije sabiranja), odnosno obezbjedjuje se funkcionisanje 32-bitne ALU kao jedinstvene cjeline.

4.1.3 Operacija ODUZIMANJA

Operacija oduzimanja može se izvršiti sabiranjem umanjenika sa negativnom vrijednosti umanjioca, odnosno, u binarnoj aritmetici, sabiranjem umanjenika sa dvojnim komplementom umanjioca. Sa druge strane, 1-bitna i 32-bitna ALU, predstavljene na slikama 3 i 5, već realizuju operacija sabiranja i potrebno je omogućiti da isti sistem, u pogodnom trenutku (prilikom izvršavanja operacije oduzimanja), izvrši sabiranje umanjenika i dvojnog komplementa umanjioca.

Drugim riječima, potrebno je omogućiti da dvojni komplement umanjioca, koji odgovara zbiru jediničnog komplementa (invertovanog zapisa) umanjioca sa 1-om dodatom na LSB bitu broja, može pristupiti na drugi ulaz 32-bitnog sabirača (dio 32-bitne ALU). U tom cilju, 1-bitnu ALU, sa slike 3 i upotrijebljenu u 32-bitnoj ALU na slici 5, neophodno je modifikovati tako da omogući invertovanje bitova umanjioca i njihovo dovodjenje na drugi ulaz 1-bitnih potpunih binarnih sabirača, te dodati 1-cu na potpuni 1-bitni sabirač koji se nalazi na LSB bitu (dio ALU_0 – 1-bitne ALU na LSB bitu).

- Invertovanje bitova umanjioca realizuje se dodavanjem invertora (najjednostavnijeg logičkog kola) u dizajn 1-bitne ALU, u granu kojom se pristupa drugom ulazu potpunog 1-bitnog sabirača, kako bi se, po potrebi (kada se izvršava operacija oduzimanja), na drugi ulaz potpunog 1-bitnog sabirača doveo invertovani bit umanjioca (operanda B). Medjutim, po potrebi (kada se izvršava operacija sabiranja), na drugi ulaz potpunog 1-bitnog sabirača treba dovesti odgovarajući originalni (neinvertovani) bit operanada B. Kako bi sistem mogao odabrati da li će se na drugi ulaz potpunog 1-bitnog sabirača dovesti originalni (neinvertovani) ili invertovani bit operanda B, na ovom ulazu potpunog 1-bitnog sabirača potrebno je dodati multipleksor 2/1 (sa jednim selekcionim ulazom Binvert), kao što je prikazano na slici 6 a). Selekcioni ulaz Binvert ovog multipleksora upravlja propuštanjem originalne (neinvertovane) ili invertovane vrijednosti odgovarajućeg bita operanda B na drugi ulaz potpunog 1-bitnog sabirača.
- Dodavanje 1-ce na LSB bitu može se obaviti njenim dovodjenjem na $CarryIn=CarryIn_0$ ulaz 32-bitne ALU.



Slika 6. Modifikovana 1-bitna ALU koja, pored operacija AND, OR i 1-bitnog sabiranja, obezbjedjuje i podršku izvršavanju buduće operacije oduzimanja (kod 32-bitne ALU). a) Logička struktura, b) Šematski prikaz.

1-bitna ALU, modifikovana na način da podrži realizaciju operacije oduzimanja, prikazana je na slici 6. Pretpostavimo da je ova 1-bitna ALU uključena u dizajn 32-bitne ALU sa slike 5. Primijetimo da se za Binvert=0, na drigi ulaz sabirača dovodi originalni (neinvertovani) bit operanda B, dok se za Binvert=1, na drugi ulaz ovog sabirača dovodi invertovani bit operanda B. Dakle, da bi se implementirala operacija oduzimanja, na druge ulaze potpunih 1-bitnih sabirača treba dovesti invertovane bitove operanda B (sa Binvert=1), selektovati ulaz 2 izlaznih multipleksora MUX 4/1 1-bitnih ALU (sa Operation=10), te postaviti ulazni prenos 32-bitne ALU na 1-cu, CarryIn=CarryIn$_0$=1. Da bi se implementirala operacija sabiranja, na druge ulaze potpunih 1-bitnih sabirača treba dovesti originalne (neinvertovane) bitove operanda B (sa Binvert=0), selektovati ulaz 2 izlaznih multipleksora MUX 4/1 1-bitnih ALU (sa Operation=10), te postaviti ulazni prenos 32-bitne ALU na 0-u, CarryIn=CarryIn$_0$=0.

Preciznije, razlikujemo sljedeća 2 slučaja:

1. Za Binvert=0, CarryIn=0, Operation=10,

   $F = A + B$

2. Za Binvert=1, CarryIn=1, Operation=10,

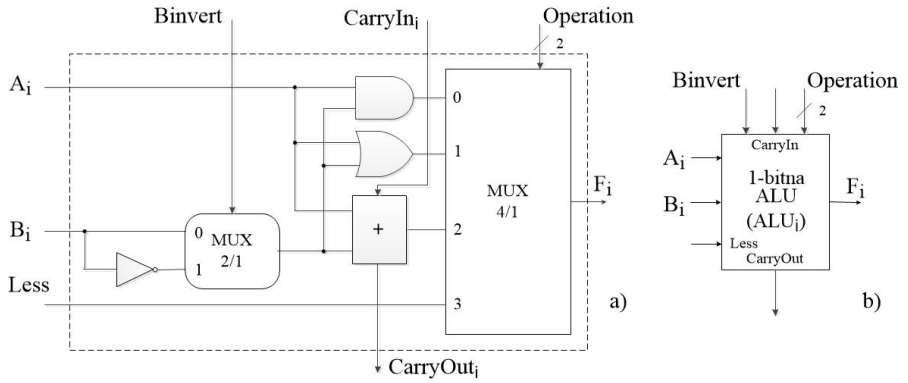
   $F = A + \overline{B} + 1 = A + (\overline{B} + 1) = A + (-B) = A - B$

   gdje je $\overline{B} + 1 =$ dvojni komplement $(B) = -B$.

NAPOMENA: Primijetimo da je na veoma jednostavan način, uz adekvatnu kontrolu, isto hardware-sko rješenje upotrijebljeno za implementaciju dvije aritmetičke operacije: sabiranja i oduzimanja. Jednostavnost dizajniranja hardware-a za implementaciju sabirača/oduzimača uz pomoć dvojnog komplementa predstavlja veoma moćan razlog upotrebe dvojnog komplementa kao univerzalnog standarda u integrisanju aritmetike računara.

4.2 32-BITNA ALU PRILAGODJENA POTREBAMA INSTRUKCIJA POREDJENJA SLT I USLOVNOG SKOKA/GRANANJA BEQ I BNE

32-bitna ALU predstavlja, kako je ranije rečeno, snagu ili moć računara koja obezbjedjuje izvršavanje svih operacija koje se instrukcijama mogu zahtijevati. Do sada dizajnirana ALU omogućava izvršavanje operacija AND, OR, sabiranja i oduzimanja, odnosno operacija čije izvršavanje redom zahtijevaju instrukcije *and*, *or*, *add* i *sub*. Primijetimo da ove instrukcije predstavljaju osnovne aritmetičke i logičke instrukcije koje mora biti sposoban da izvrši svaki računar. Medjutim, instrukcije uslovnog skoka/grananja *beq* i *bne*, kao i instrukcija *slt* omogućavaju ispitivanje uslova jednakosti/nejednakosti, odnosno relativnih odnosa zadatih operanada. Drugim riječima, ove instrukcije omogućavaju računaru da vrši izbore izmedju nekoliko mogućih opcija, tako da mogućnost njihove implementacije sušinski razlikuje računar od običnog kalkulatora. Stoga ih je veoma značajno implementirati.



Slika 7. Modifikovana 1-bitna ALU dizajnirana da, pored operacija zahtijevanih instrukcijama *and*, *or*, *add* i *sub*, dodatno omogući izvršavanje operacije zahtijevane instrukcijom *slt*. a) Logička struktura, b) Šematski prikaz.

4.2.1 Implementacija operacije zahtijevane *slt* instrukcijom

Instrukcijom *slt*, poglavlje 3.4, porede se sadtžaji operanada, te ukoliko je sadržaj prvonavedenog operanda ($y) manji od sadržaja drugonavedenog operanda ($z), u rezultujući registar ($x) upisuje se 1 (registar se *set*-uje); u suprotnom u rezultujući registar upisuje se 0,

$$slt \qquad \$x, \$y, \$z \qquad \#\ \$x = \begin{cases} 1, & \text{sadržaj } (\$y) < \text{sadržaj } (\$z) \\ 0, & \text{sadržaj } (\$y) \geq \text{sadržaj } (\$z) \end{cases}$$

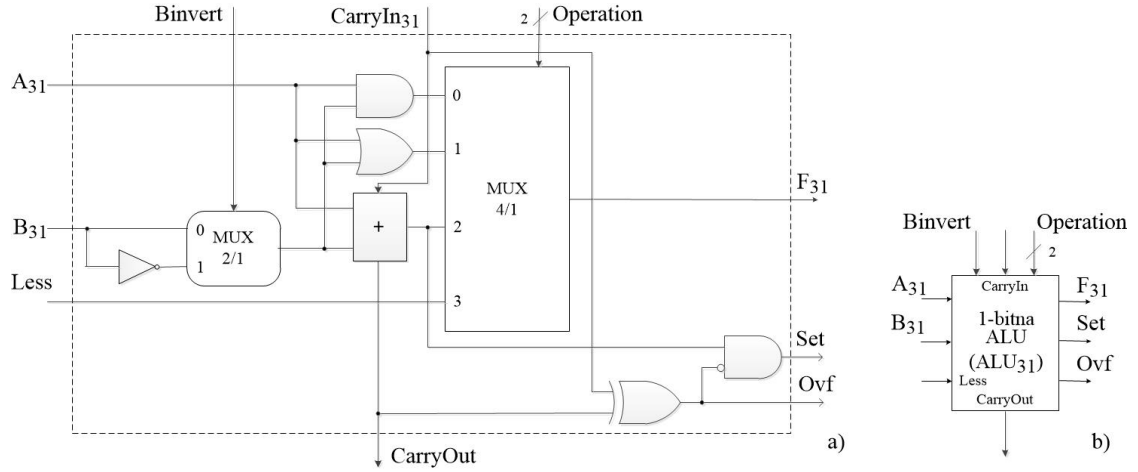
gdje se rezultujuća 1 u 32-bitnom registru $x upisuje sa 31 0-om u višim bitovima registra i 1 u njegovom LSB bitu, 00...01, dok se rezultujuća 0 upisuje sa 32 0-e, uključujući 0 u LSB bitu. Drugim riječima, nakon izvršavanja instrukcije *slt*, u svim bitovima rezultujućeg registra ($x) biće upisane 0-e, osim u LSB bitu (bitu 0) rezultujućeg registra u kome će biti upisano 1 ukoliko je sadržaj prvonavedenog operanda ($y) manji od sadržaja drugonavedenog operanda ($z), odnosno 0 ukoliko sadržaj prvonavedenog operanda ($y) nije manji (veći je ili jednak) od sadržaja drugonavedenog operanda ($z),

$$slt \qquad \$x, \$y, \$z \qquad \#\ \text{LSB}(\$x) = \begin{cases} 1, & \text{sadržaj } (\$y) < \text{sadržaj } (\$z) \\ 0, & \text{sadržaj } (\$y) \geq \text{sadržaj } (\$z) \end{cases} \tag{2}$$

Shodno navedenom, da bi se implementirala instrukcija *slt*, potrebno je modifikovati 1-bitnu ALU prezentiranu na slici 6. U tom cilju, na četvrti ulaz (do sada neupotrebljavani ulaz 3) izlaznog multipleksora MUX 4/1 dovodi se signal Less. Na ovaj ulaz dovodi se 0, Less=0, na svim višim 1-bitnim ALU (ALU_i, i=1, 2, ..., 30, 31), osim 1-bitne ALU koja odgovara LSB bitu (ALU_0). U cilju implementacije *slt* instrukcije, Less ulaz 1-bitne ALU koja odgovara LSB bitu (ALU_0) treba da uzme vrijednost 1 ukoliko je sadržaj prvonavedenog operanda ($y) manji od sadržaja drugonavedenog operanda ($z), odnosno treba da uzme vrijednost 0 ukoliko sadržaj prvonavedenog operanda ($y) nije manji (veći je ili jednak) od sadržaja drugonavedenog operanda ($z).

Medjutim, postavlja se pitanje kako na Less ulaz ALU_0 dovesti vrijednost koja odgovara medjusobnom odnosu operanada $y i $z i kako to hardware-ski implementirati?

Da bismo odgovorili na ovo pitanje, podsjetimo najprije da je ALU kombinaciona logika, čiji svi sastavni elementi obavljaju operacije kojima su namijenjeni, dok kontrolni bitovi ALU (Operation, Binvert, CarryIn) odlučuju koju od obavljenih operacija će ALU prezentirati kao svoj rezultat. Drugim riječima, tokom izvršavanja *slt* instrukcije, Operation kontrolni signali će biti postavljeni na 11, kako bi ALU na svom izlazu dala rezultat koji odgovara *slt* instrukciji. Medjutim, istovremeno će i ostale funkcionalne cjeline/elementi ALU obavljati svoje funkcije (AND, OR, te sabiranje ili oduzimanje zavisno od vrijednosti kontrolnih signala Binvert i CarryIn=$CarryIn_0$). Naravno, i rezultat ovih operacija može biti upotrijebljen ukoliko se obezbijedi izlaz sa kog bi on mogao biti uzet ili pročitan.



Slika 8. Modifikovana 1-bitna ALU na MSB bitu (ALU_{31}) dizajnirana da, pored operacija zahtijevanih instrukcijama *and*, *or*, *add*, *sub*, *slt*, dodatno omogući postavljanje Set izlaza i detekciju overlow-a (Ovf bit). a) Logička struktura, b) Šematski prikaz.

Shodno navedenom, postavljanjem Operation bitova na 11, rezultat ALU će odgovarati operaciji zahtijevanoj instrukcijom *slt*, ali istovremenim postavljanjem kontrolnih bitova Binvert i CarryIn=$CarryIn_0$ na 1, 32-bitni sabirač, kao dio 32-bitne ALU, će vršiti oduzimanje operanada dovedenih na njegove ulaze (to su sadržaji registara $y i $z kao i kod *slt* instrukcije), samo što rezultat operacije oduzimanja neće biti prezentiran kao rezulzat ALU.

Razmotrimo rezultat operacije oduzimanja operanada sadržanim u registrima $y i $z. Znak dobijenog rezultata biće negativan (znak bit, odnosno MSB=1) ukoliko je umanjenik manji od umanjioca (sadržaj registra $y manji od sadržaja registra $z), dok će znak dobijenog rezultata biti pozitivan (MSB=0) ukoliko umanjenik nije manji (veći je ili jednak) od umanjioca (sadržaj registra $y nije manji od sadržaja registra $z). Znak rezultata ove operacije (MSB bit) kreira naravno potpuni 1-bitni sabirač na MSB bitu (bitu 31),

$$\text{MSB(Res)} = \begin{cases} 1, & \text{sadržaj (\$y)} < \text{sadržaj (\$z)} \\ 0, & \text{sadržaj (\$y)} \geq \text{sadržaj (\$z)} \end{cases} \tag{3}$$

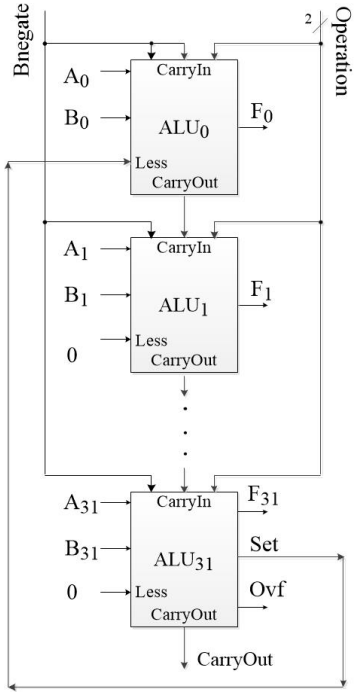
gdje je sa Res označen rezultat koji se dobija oduzimanjem sadržaja registara $y i $z, a koji se ne predstavlja se na izlazu ALU kao njen rezultat, već se dobija na izlazu 32-bitnog sabirača. Medjutim, uporedjivanjem jednačina (2) i (3), jednostavno se može zaključiti da rezultat dobijen na izlazu potpunog 1-bitnog sabirača na MSB biti (bitu 31), nakon oduzimanja oparanada dovedenih na ulaze ALU, odgovara rezultatu koji je potrebno kreirati na LSB bitu prilikom izvršavanja instrukcije *slt* nad istim operandima. To je ujedno odgovor na pitanje postavljeno nakon jednačine (2).

Medjutim, da bi se prethodno izvedeni zaključak mogao implementirati 1-bitnu ALU koja odgovara MSB bitu (ALU_{31}) treba modifikovati u odnosu na ostale 1-bitne ALU (ALU_i, i=0, 1, ..., 30). U tom cilju, izlaz potpunog 1-bitnog sabirača na MSB bitu treba napraviti izlazom ALU_{31} i to izlazom istog ranga kao njen rezultujući izlaz F_{31}, kao što je prikazano na slici 8. Ovaj izlaz nazivamo Set, jer će se on upotrebljavati za postavljanje (set-ovanje) Less ulaza ALU_0.

U medjuvremenu, kada već modifikujemo 1-bitnu ALU koja odgovara MSB bitu (ALU_{31}) u cilju implementacije instrukcije *slt*, iskoristimo priliku i kreirajmo logiku za detekciju overflow-a, koji se može dogoditi prilikom izvršavanja aritmetičkih operacija. Naime, podsjetimo da se overflow detektuje takodje na MSB bitu – ekskluzivnom ILI operacijom izmedju ulaznog i izlaznog prenosa na MSB bitu (pogledaj jednačinu (1)), odnosno u slučaju razmatrane 32-bitne ALU,

$$\text{Ovf}=\text{CarryIn}_{31}\oplus\text{CarryOut}_{31}=\text{CarryIn}_{31}\oplus\text{CarryOut}. \tag{4}$$

Primijetimo da se Set izlaz ALU_{31} uzima sa izlaza potpunog 1-bitnog sabirača sa MSB bita. Shodno tome, Set izlaz ima smisla postavljati samo ako nije došlo do overlow-a (ukoliko je Ovf=0),



Slika 9. Potpuna 32-bitna ALU dizajniranja u cilju obavljenja operacija zahtijevanih prilikom izvršavanju instrukcija *and*, *or*, *add*, *sub* i *slt*.

$\text{Set} = S_{31} \cdot \overline{\text{Ovf}}$, što je na slici 8 implementirano izlaznim I kolom sa komplementiranim ulazom Ovf. Sa S_{31}, u prethodnom izrazu, označen je izlaz potpunog 1-bitnog sabirača sa MSB bita (bita 31),.

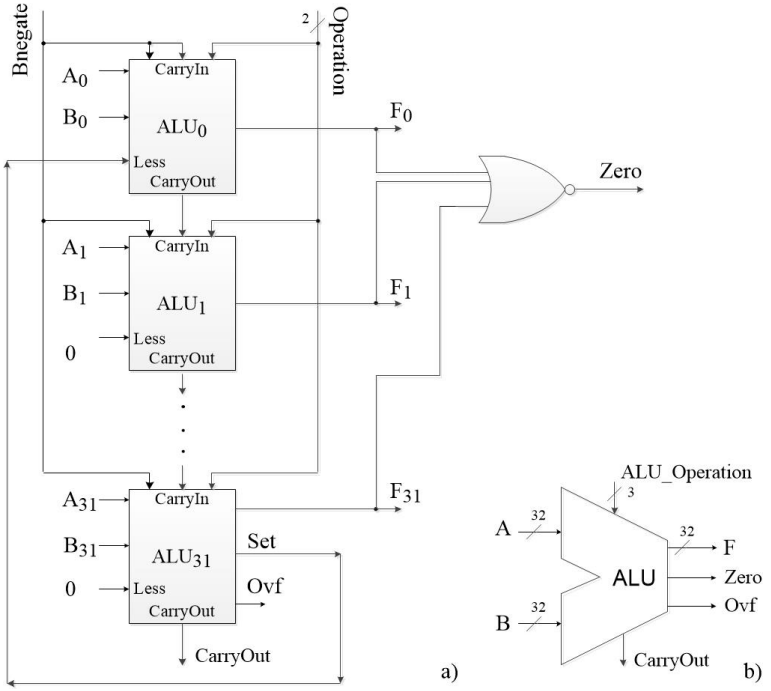
Potpuna 32-bitna ALU, dizajniranja medjusobnim povezivanjem 31 1-bitne ALU sa slike 7 (ALU_i, i=0, 1, ..., 30) i jedne modifikovane 1-bitne ALU sa slike 8 (na MSB bitu – ALU_{31}), prikazana je na slici 9. Namijenjena je izvršavanju operacije koje zahtijevaju instrukcije *and*, *or*, *add*, *sub* i *slt*. Kao svoje izlaze, 32-bitna ALU sa slike 9 generiše rezultat zahtijevane operacije F i overflow bit Ovf, kao indikator detektovanja prekoračenje dozvoljenog opsega registra (overflow-a).

NAPOMENA: Kad god 32-bitna ALU izvršava operaciju oduzimanja, bilo da njeno izvršavanje zahtijeva instrukcija *sub* ili u cilju implementiranja instrukcije *slt*, Binvert i CarryIn kontrolni bitovi uzimaju jediničnu vrijednost (Binvert=1 obezbjedjuje dovodjenje invertovanih vrijednosti bitova drugog operanda na ulaz sabirača, dok CarryIn=CarryIn_0=1 dopunjava jedinični komplement drugog operanda do njegovog dvojnog komplementa). Sa druge strane, prilikom izvršavanja ostalih operacija (AND, OR, sabiranje), kontrolni bitovi Binvert i CarryIn uzimaju nultu vrijednost. Drugim riječima, prilikom izvršavanja svih operacija, kontrolni signali Binvert i CarryIn uzimaju iste vrijednosti, odnosno predstavljaju jedan te isti kontrolni signal (Bnegate signal na slici 9).

4.2.2 Implementacija operacija zahtijevani instrukcijom uslovnog skoka/grananja *beq* i *bne*

Instrukcijama uslovnog skoka/grananja *beq* i *bne*, ispituje se jednakost (u slučaju instrukcije *beq*) i nejednakost (u slučaju instrukcije *bne*) zadatih operanada. Ukoliko je ispitivani uslov zadovoljen, ovim instrukcijama vrši se grananje/skok na instrukciju čije je obilježje/adresa zapisana u kodu instrukcije *beq*/*bne*. U suprotnom, sljedeća instrukcija koja će se izvršavati biće instrukcija koja je u memoriji računara zapisana odmah nakon instrukcije *beq*/*bne*.

U cilju implementacije instrukcija *beq* i *bne*, ALU treba da izvrši poredjenje operanada u odnosu na jednakost i nejednakost, respektivno (kasnije će ostatak datapath-a voditi računa o adresi instrukcije koja sljedeća treba da se izvrši – nakon instrukcije *beq*/*bne*). U tom smislu, jednostavnije je da ALU koju dizajniramo testira jednakost operanada. Naime, tada je razlika uporednih operanada A i B nula, A–B=0, odnosno svaka od 32 1-bitne ALU (ALU_i, i=0, 1, ..., 31) na svom rezultujućem izlazu generiše nulu, $F_0=F_1= ... =F_{31}=0$. Drugim riječima, ukoliko na logičko NILI (eng. NOR) kolo sa 32



Slika 10. Potpuna 32-bitna ALU dizajniranja u cilju obavljenja operacija zahtijevanih prilikom izvršavanju instrukcija *and*, *or*, *add*, *sub*, *slt*, *beq* i *bne*. a) Logička struktura, b) Standardni šematski prikaz ALU.

Tabela 4. Funcije 32-bitne ALU predstavljene u zavisnosti od kontrolnih signala ALU (ALU_Operation bitova).

Kontrolni signali ALU (ALU_Operation)		Funkcija ALU
Bnegate	Operation	
0	00	AND
0	01	OR
0	10	Add
1	10	Sub
1	11	Set-on-Less-Than

ulaza dovedemo rezultujuće izlaze svake od 32 1-bitne ALU, kao što je prikazano na slici 10 a), izlaz ovog kola (nazvan Zero) će biti 1 samo ukoliko je $F_0=F_1= ... =F_{31}=0$, odnosno ukoliko su operandi A i B jednaki. Jednovremeno, Zero bit je izlaz 32-bitne ALU istoga ranga kao rezulrujući 32-bit izlaz F, Ovf bit i CarryOut bit, slika 10 b). Zero bit detektuje jednakost operanada dovedenih na ulaze ALU, Ovf bit mogući overflow, dok je CarryOut bit izlazni prenos (tzv. pretek iz 32-bitne ALU). Pojednostavljeno, Zero=1 detektuje zadovoljenje uslova jednakosti operanada koji se ispituje prilikom izvršavanja instrukcije *beq*, dok Zero=0 detektuje zadovoljenje uslova nejednakosti operanada koji se ispituje prilikom izvršavanja instrukcije *bne*.

Postavlja se još jedno pitanje: što je sa izlazom Set sa 1-bitne ALU koja odgovara MSB bitu (ALU_{31}) i da li je on izlaz 32-bitne ALU? Odgovor na ovo pitanje je jednostavan. Set bit sa izlaza ALU_{31} postavlja Less ulaz ALU_0 i to je njegova namjena. Ovo se dogadja unutar 32-bitne ALU i ne predstavlja njen izlaz, kao što se može primijetiti na slici 10 b).

Na slici 10 b) data je standardna šematska oznaka ALU. Treba napomenuti da se veoma sličnom oznakom na šemama označavaju višebitni sabirači (32-bitni sabirači u našem smislu riječi), s' tom razlikom što se u slučaju sabirača na šematskoj oznaci stavlja znak + ili akronim ADD, dok je u

slučaju ALU u njenoj šematskoj oznaci jasno naglašava o čemu je riječ, kao što je prikazano na slici 10 b).

Funkcionisanje ALU i operacije koje je ona sposobna da izvršava zavisno od vrijednosti kontrolnih bitova ALU_Operation=(Bnegate,Operation(1,0)) sumirane su u tabeli 4. Prmijetimo da nijesu upotrijebljene 3 kombinacije kontrolnih bitova (Bnegate,Operation(1,0))=(0,1,1), (1,0,0), (1,0,1). Ova činjenica može biti upotrijebljena u cilju budućeg dodavanja novih operacija koje ALU treba da izvrši, a može biti upotrijebljena i za minimizaciju datapath-a (okruženja) sa kojim ALU neposredno komunicira.