5.7 NADOGRADNJA/REDIZAJNIRANJE ARHITEKTURE ZA MULTITAKTNU IMPLEMENTACIJU U CILJU REALIZACIJE IMMEDIATE INSTRUKCIJA

Arhitektura za multitaktni implementaciju, prikazana na slici 11, i njena glavna kontrolna jedinica, čija je logika dizajniranja zasnovana na mašini sa konačnim brojem stanja prikazana na slikama 18 i 19, obezbjedjuju implementaciju instrukcija R-tipa (*add*, *sub*, *and*, *or*, *slt*), *lw*, *sw*, *beq* i *j*. U ovom poglavlju razmatraćemo načine za nadogradnju/redizajniranje arhitekture sa slika 11, 18 i 19 u cilju implementacije najčešće upotrebljavanih immediate instrukcija *addi*, *slti*, *ori* i *andi*. Nadogradnju ćemo razmatrati pojedinačnim dodavanjem svake od navedenih immediate instrukcija. Nakon toga ćemo razmotriti mogućnost jednovremenog uključivanja sve 4 immediate instrukcije.

5.7.1 Prilagodjavanje arhitekture za multitaktnu implementaciju u cilju implementacije *addi* instrukcije

Prije nego predjemo na razmatranje mogućeg prilagodjavanja postojeće arhitekture za multitaktnu implementaciju u cilju uključivanja *addi* instrukcije, podsjetimo se simboličkog koda i mašinskog formata zapisa ove instrukcije. Njen simbolički kod (pogledaj poglavlje 3.6 i tabelu 1 iz poglavlja 5.1) je:

addi $x, $y, C # $x=$y+C

gdje se konstanta (konstantni operand) C čuva unutar instrukcije i zapisuje na njenom kraju, dok $x, x=1,...,31, označava registar u koji se upusuje rezultat sabiranja operanda sadržanog u registru $y, y=0,...,31, sa konstantom C, kao što je zapisano u komentaru gore navedene instrukcije.

Mašinski format zapisa instrukcije *addi* (pogledaj poglavlje 3.6 i tabelu 1 iz poglavlja 5.1) je:

Polje:	op	rs	rt	Immediate
Br. bitova:	6	5	5	16
Sadržaj:	8	y	x	C

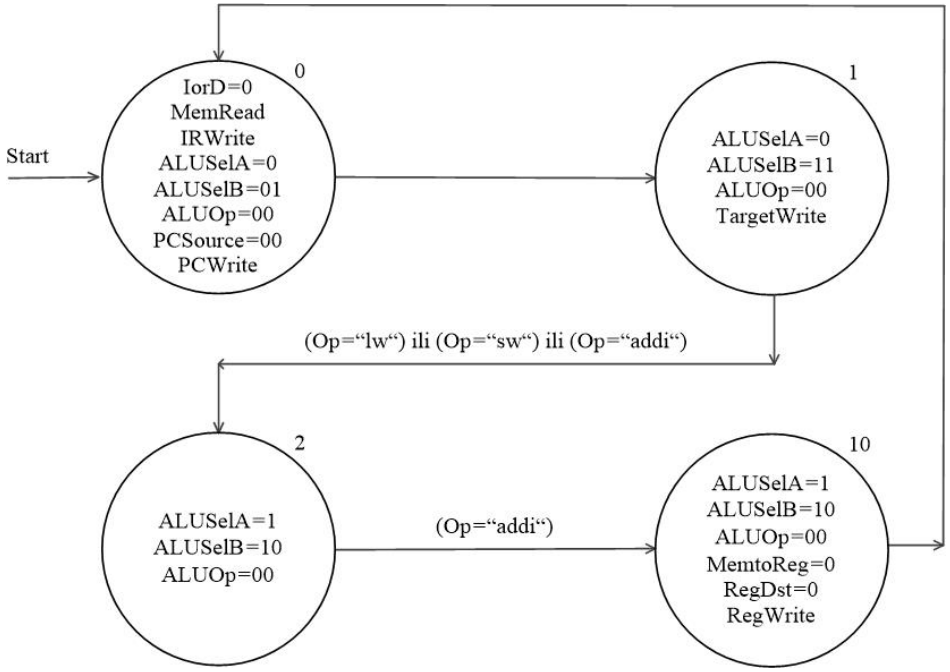
Slika 20. Mašinski format zapisa instrukcije *addi*.

Primijetimo da je, kod instrukcije *addi*, op=Instruction [31–26]=$8_{(10)}$=$001000_{(2)}$=0x8, gdje se oznakom 0x opisuje heksadekadni zapis (odnosno, 0x8≡$8_{(16)}$). U mašinskom formatu zapisa, konstanta C smještena je na najnižim bitovima instrukcije (u 16-bitnom Immediate=Instruction [15–0] polju), u rs=Instruction [25–21] polju instrukcije zapisano je obilježje (broj) registra u kome se nalazi operand ($y u simboličkoj formi), a u rt=Instruction [20–16] polju instrukcije – obilježje odredišnog registra (registar u kome se smješta rezultat sabiranja).

Drugim riječima, nakon uzimanja instrukcije, njenog dekodiranja i čitanja sadržaja registra označenog rs=IR [25–21] poljem instrukcije (u prva 2–zajednička–koraka/taktna intervala – stanja 0 i 1 sa slike 18), u cilju kompletiranja instrukcije *addi* potrebno je implementirati akciju:

Reg(IR [20–16])=Reg(IR [25–21]) + sign_extend(IR [15–0]). (14)

PRIMJEDBA: Taktni interval multitaktne implementacije, razmatrane u ovom materijalu, odredjen je vremenom pristupa Memory jedinice (pogledaj napomenu 1 u poglavlju 5.5). Medjutim, na isti način kao u slučaju instrukcija R-tipa, izvršavanje akcije (14) (i drugih pojedinačnih akcija povezanih sa aritmetičko-logičkim operacijama koje mogu biti zahtijevane od strane instrukcija R-tipa i od strane ostalih immediate instrukcija – pogledaj poglavlje 3.6) zahtijeva izračunavanje u ALU, te nakon toga pristup Registers jedinici u cilju upisa izračunatog rezultata (u slučaju immediate instrukcija, upis se vrši u registar označen rt=IR [20–16] poljem instrukcije). Odnosno, izvršavanje akcije (14) (i svih drugih akcija koje mogu biti zahtijevane instrukcijama R-tipa, sekcija 5.6.5, i immediate instrukcijama) zahtijeva vrijeme koje odgovara vremenu potrebnom za izračunavanje od strane ALU uvećanom za vrijeme pristupa Registers jedinici i koje, shodno tome, prevazilazi dužinu taktnog intervala odredjenog vremenom pristupa Memory jedinice (pogledaj napomene 3 i 4 iz sekcije 5.6.5). Drugim riječima, ukoliko bi dužina taktnog intervala bila dizajnirana tako da odgovara vremenu



Slika 21. Mašina sa konačnim brojem stanja za implemantaciju instrukcije *addi*.

zahtijevanom za izvršavanja akcije (14), ona bi bila veća od intervala odredjenog vremenom pristupa Memory jedinici (pogledaj primjer iz poglavlja 5.5), što bi impliciralo gubitke u vremenu izvršavanja koji se odnose na ostale korake, a time i gubitke u ukupnom vremenu izvršavanja pojedinačnih instrukcija. Stoga, taktni interval dizajnirane implementacije ostaje odredjen vremenom pristupa Memory jedinici, a akcija (14) izvršava se tokom 2 koraka/taktna intervala (III i IV).

Shodno navedenom, u III koraku/taktnom intervalu, izračunava se operacija zahtijevana akcijom (14), dok se u IV koraku/taktnom intervalu, izračunati rezultat upisuje u odredišni registar Reg(IR [20–16]),

1. U cilju izračunavanja operacije zahtijevane akcijom (14), na prvi ulaz ALU potrebno je dovesti sadržaj registra koji je označen rs=IR [25–21] poljem instrukcije (*ALUSelA*=1), na drugi ulaz ALU – 16-bitno immediate=IR [15–0] polje instrukcije produženo, kopiranjem znaka, do 32-bitne dužine (*ALUSelB*=10), te omogućiti ALU da izvrši operaciju sabiranja (*ALUOp*=00). Postavljanjem ovih kontrolnih signala kreira se stanje kojim se obezbjedjuje izvršavanje III koraka/taktnog intervala tokom implementacije *addi* instrukcije (stanje 2 na slici 21). Primijetimo da je ovo stanje identično stanju 2, koje je ranije kreirano u cilju izvršavanje instrukcija *lw*/*sw*, slika 18.

   NAPOMENA 1: Sabiranje koje se obavlja pod kontrolom funct polja pretpostavlja postavljanje bitova ALUOp=10 (pogledaj tabelu 7 iz sekcije 5.4.1). Medjutim, immediate instrukcije ne posjeduju funct polje (slika 20), tako da je ALUOp≠10 u slučaju immediate instrukcija, a operacija sabiranja zadaje se postavljanjem ALUOp=00, kao u slučaju instrukcija *lw*/*sw*, te u slučaju sabiranja u I i u II koraku izvršavanja instrukcija, kada takodje ne raspolažemo funct poljem.

   NAPOMENA 2: 16-bitno immediate=IR [15–0] polje instrukcije, u kome je zapisana konstanta C, produžava se, ***kopiranjem znaka***, do 32-bitne dužine da bi ostao nepromijenjen znak ove konstante, odnosno da bi se omogućio rad i sa negativnim konstantnim operandom, odnosno da bi se implementacijom *addi* instrukcije jednovremeno omogućilo izvršavanje opercije oduzimanja sa konstantnim operandom.

2. U cilju upisa izračunatog rezultata u odredišni registar (registar označen rt=IR [20–16] poljem instrukcije) potrebno je obezbijediti sljedeće:

2.1. Nepromjenljivost rezultata (iz prethodnog koraka/taktnog intervala) na izlazu ALU, zadržavanjem istih kontrolnih signala koji se odnose na funkcionisanje ALU, a postavljeni su u prethodnom koraku, odnosno *ALUSelA*=1, *ALUSelB*=10, ALUOp=00 (za razloge pogledaj napomenu 2 u sekciji 5.6.5 i paragraf koji se nalazi nakon napomene 2 u poglavlju 5.5.2).

2.2. Dovesti rezultat sa izlaza ALU na Write data ulaz Registers jedinice (*MemtoReg*=0), poljem rt=IR [20–16] instrukcije označiti/adresirati registar u koji je potrebno upisati dovedeni rezultat (*RegDst*=0), te, na koncu, dozvoliti upis dovedenog rezultata u označeni registar (*RegWrite*).

Postavljeni kontrolni signali formiraju novouvudeno (u odnosu na dijagram toka sa slike 18) stanje 10, slika 21, kojim se kompletira izvršavanje instrukcije *addi*.

Primijetimo da za implementaciju instrukcije *addi* nijesu potrebne izmjene/redizajniranje postojeće arhitekture sa slike 11 (datapath-a, ALU control ili glavne kontrolne jedinice, odnosno kreiranje bilo kog novog ili izmjena postojećeg kontrolnog signala), već samo uvodjenje novog stanja u funkcionisanje mašine sa konačnim brojem stanja kojom se implementira logika glavne kontrolne jedinice multitaktne implementacije.

Realizacija instrukcije *addi* uključuje 4 koraka/taktna intervala implementirana stanjima 0, 1, 2, 10, gdje su

- stanje 0 (čitanje/uzimanje instrukcije i njeno donošenje u proces izvršavanja) i stanje 1 (dekodiranje instrukcije i čitanje sadržaja označenih registara) – zajednička stanja za implementaciju svih instrukcija,
- stanje 2 – postojeće stanje upotrebljavano za implementaciju instrukcija *lw*/*sw*, dok je
- stanje 10 – novouvedeno stanje za implementaciju instrukcije *addi*.

NAPOMENA 3: Primijetimo razlike i sličnosti izmedju stanja 10 kojim se kompletira instrukcija *addi* i stanja 4 i 7 kojima se kompletiraju *lw* i instrukcije R-tipa (navedene instrukcije takodje zahtijevaju upis rezultata zahtijevane operacije nazad – u odredišni registar):

- Slično instrukcijama R-tipa, instrukcija *addi* zahtijeva upis **rezultata ALU** nazad u odredišni registar, tako da je *MemtoReg*=0 u stanjima 10 i 7. Suprotno tome, instrukcija *lw* zahtijeva upis **podatka pročitanog iz Memory jedinice** u odredišni registar, tako da je *MemtoReg*=1 u stanju 4,
- Slično instrukciji *lw*, odredišni registar, u koji se vrši povratni upis, označen je rt=IR [20–16] poljem instrukcije *addi*, tako da je *RegDst*=0 u stanjima 10 i 4. Suprotno tome, kod instrukcija R-tipa, odredišni registar označen je rd=IR [15–11] poljem instrukcije, tako da je *RegDst*=1 u stanju 7.

5.7.2 Nadogradnja/redizajniranje arhitekture za multitaktnu implementaciju u cilju implementacije *slti* instrukcije

Prije nego započnemo redizajniranje postojeće arhitekture za multitaktnu implementaciju u cilju uključivanja *slti* instrukcije, podsjetimo se simboličkog koda i mašinskog formata zapisa ove instrukcije. Njen simbolički kod (pogledaj poglavlje 3.6 i tabelu 1 iz poglavlja 5.1) je:

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

gdje se konstanta (konstantni operand) C čuva unutar instrukcije i zapisuje na njenom kraju, dok $x, x=1,...,31, označava registar u koji se upusuje rezultat set-on-less-than operacije (rezultat poredjenja operanda, sadržanog u registru $y, y=0,...,31, sa konstantom C u odnosu "manji od", kao što je zapisano u komentaru gore navedene instrukcije).

Mašinski format zapisa instrukcije *slti* (pogledaj poglavlje 3.6 i tabelu 1 iz poglavlja 5.1) je:

Polje:	op	rs	rt	Immediate
Br. bitova:	6	5	5	16
Sadržaj:	10	y	x	C

Slika 22. Mašinski format zapisa instrukcije *slti*.

Primijetimo da mašinski format zapisa *slti* instrukcije odgovara mašinskom formatu zapisa instrukcije *addi* (pogledaj sliku 20 iz sekcije 5.7.1), s tom razlikom što je op=Instruction [31–26]=$10_{(10)}$ =$001010_{(2)}$=0xA u slučaju *slti* instrukcije (kod *addi* instrukcije je op=Instruction [31–26]=$8_{(10)}$ =$001000_{(2)}$=0x8), gdje se oznakom 0x opisuje heksadekadni zapis. U mašinskom formatu zapisa *slti* instrukcije, konstanta C smještena je na najnižim bitovima instrukcije (u 16-bitnom immediate=Instruction [15–0] polju), u rs=Instruction [25–21] polju instrukcije zapisano je obilježje registra u kome se nalazi operand ($y u simboličkoj formi), a u rt=Instruction [20–16] polju instrukcije – obilježje odredišnog registra (registar u kome se smješta rezultat set-on-less-than operacije).

Drugim riječima, nakon uzimanja instrukcije, njenog dekodiranja i čitanja sadržaja registra označenog rs=IR [25–21] poljem instrukcije (u prva 2–zajednička–koraka/taktna intervala – stanja 0 i 1 sa slike 18), u cilju kompletiranja instrukcije *slti* potrebno je implementirati akciju:

$$\text{Reg(IR [20–16])}=\text{Reg(IR [25–21])}\ \underline{\text{set-on-less-than}}\ \text{sign\_extend(IR [15–0])}, \qquad (15)$$

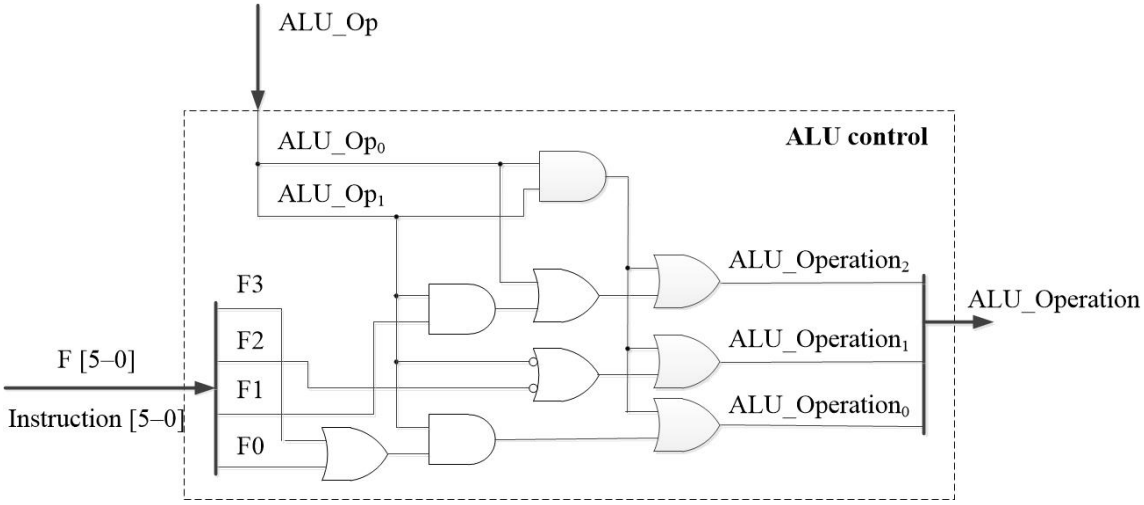
gdje je sa <u>set-on-less-than</u> simbolički zapisana set-on-less-than operacija (poredjenje u odnosu na "manji od") izmedju sadržaja registra označenog rs=IR [25–21] poljem instrukcije i 16-bitnog immediate=IR [15–0] polja instrukcije koje je produženo, kopiranjem znaka, do 32-bitne dužine.

Shodno činjenicama detaljno razmotrenim u primjedbi iz sekcije 5.7.1, akcija (15) zahtijeva izvršavanje u 2 koraka/taktna intervala (III i IV). U III koraku/taktnom intervalu, izračunava se operacija set-on-less-than, zahtijevana akcijom (15), dok se u IV koraku/taktnom intervalu, izračunava rezultat upisuje u odredišni registar Reg(IR [20–16]).

1. U cilju izračunavanja operacije zahtijevane akcijom (15), na prvi ulaz ALU potrebno je dovesti sadržaj registra koji je označen rs=IR [25–21] poljem instrukcije (*ALUSelA*=1), na drugi ulaz ALU – 16-bitno immediate=IR [15–0] polje instrukcije produženo, kopiranjem znaka, do 32-bitne dužine (*ALUSelB*=10), te omogućiti ALU da izvrši operaciju set-on-less-than. Medjutim, postavlja se pitanje kako omogućiti ALU da izvrši ovu operaciju?

   POJAŠNJENJE: ALU je dizajnirana za izvršavanje operacije set-on-less-than (vidji tabelu 5 iz sekcije 5.4.1), a izvršavanje ove operacije zadaje se postavljanjem kontrolnih signala $\text{ALU\_Operation}_{(2,1,0)}$=(1,1,1). Dakle, nepotrebne su izmjene u dizajnu ALU. Potrebno je samo natjerati ALU da, u cilju implementacije *slti* instrukcije, izvrši operaciju set-on-less-than. Medjutim, u slučaju set-on-less-than operacije, postavljanje ALU_Operation signala (od strane ALU control jedinice) suštinski se vrši pod kontrolom funct=Instruction [5–0] polja instrukcije. Naime, tada je ALUOp=10 (pogledaj tabele 7 i 8 u sekciji 5.4.1), ali ne samo za slučaj set-on-less-than operacije, već za svih 5 operacija zahtijevanih instrukcijama R-tipa, a sve one ne mogu jednoznačno biti definisane sa jednom vrijednošću ALUOp signala. Stoga se, u slučaju ALUOp=10, kontrola postavljanja ALU_Operation signala prepušta funct polju, kojim se jednoznačno definišu zahtijevane operacije ALU. Medjutim, immediate instrukcije (time i instrukcija *slti*) ne raspolažu funct poljem. Shodno tome, u slučaju *slti* immediate instrukcije, potrebno je pronaći alternativni način da se ALU zada da obavi operaciju set-on-less-than, a da to nije pod kontrolom funct polja.

   RJEŠENJE: ALU control jedinica postavlja ALU_Operation kontrolne signale na osnovu ALUOp bitova, koje generiše glavna kontrolna jedinica, i funct=Instruction [5–0] polja instrukcije, ali na osnovu funct polja samo kada je ALUOp=10. Kombinacija ALUOp=00 obezbjedjuje izvršavanje operacije sabiranja, kombinacija ALUOp=01 obezbjedjuje izvršavanje operacije oduzimanja, dok kombinacija ALUOp=10 prepušta kontrolu funkcionisanja ALU – funct polju instrukcije, kojim, ponovimo, ne raspolažu immediate instrukcije. Dakle, ne može se upotrijebiti niti jedna od do sada upotrebljavanih kombinacija ALUOp bitova u cilju kontrole izvršavanja set-on-less-than operacije za potrebe instrukcije



Slika 23. ALU control jedinica, redizajnirana u cilju generisanja ALU_Operation signala potrebnih za kontrolu izvršavanja operacija zahtijevanih instrukcijama iz tabele 7, ali i dodatom *slti* instrukcijom. Elementi, dodati u cilju redizajniranja (u odnosu na dizajn sa slike 7), istaknuti su sijenčenjem.

Tabela 14. Funkcionalna tabela ALU control jedinice sa uključenom operacijom koju zahtijeva izvršavanje instrukcije *slti*.

Instr.	ALUOp		Funct (Instruction [5–0])						ALU_Operation		
	1	0	F5	F4	F3	F2	F1	F0	2	1	0
lw, *sw*	0	0	×	×	×	×	×	×	0	1	0
beq	0	1	×	×	×	×	×	×	1	1	0
add	1	0	×	×	0	0	0	0	0	1	0
sub	1	0	×	×	0	0	1	0	1	1	0
and	1	0	×	×	0	1	0	0	0	0	0
or	1	0	×	×	0	1	0	1	0	0	1
slt	1	0	×	×	1	0	1	0	1	1	1
slti	1	1	×	×	×	×	×	×	1	1	1

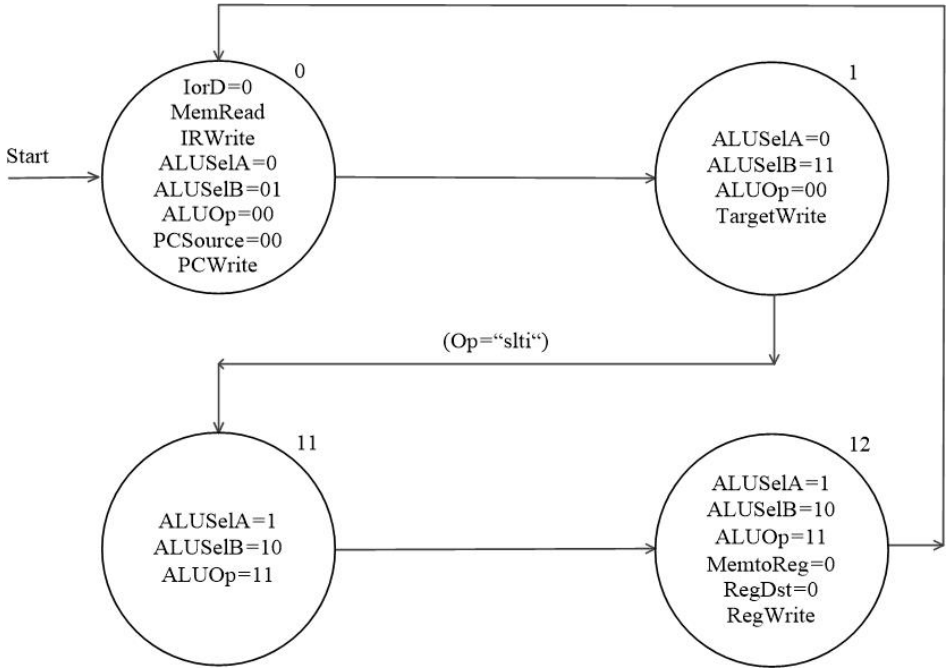
slti. Medjutim, do sada nije upotrijebljena kombinacija ALUOp=11. Upotrijebit ćemo je sada na način da se ALU zada da bezuslovno izvrši operaciju set-on-less-than kada je ALUOp=11, kao što je prikazano u tabeli 14. Drugim riječima, u cilju implementacije *slti* instrukcije potrebno je redizajnirati ALU control jedinicu iz sekcije 5.4.1 tako da obezbijedi generisanje $\text{ALU\_Operation}_{(2,1,0)}=(1,1,1)$ za ALUOp=11. Primijetimo da funkcionalna tabela 14, u poredjenju sa tabelama 7 i 8 iz sekcije 5.4.1, sadrži samo jedan dodatni red više (za kombinaciju bitova ALUOp=11 – red odvojen debljom isprekidanom linijom), tako da je:

$$\text{ALU\_Operation}_{(2,1,0)} = \text{ALU\_Operation}^{\text{staro}}_{(2,1,0)} + \text{ALUOp}_1 \wedge \text{ALUOp}_0$$

gdje kontrolni signali $\text{ALU\_Operation}^{\text{staro}}_{(2,1,0)}$ odgovaraju dizajnu ALU control jedinice sa slike 7 i već su izvedeni u sekciji 5.4.1 i dati izrazima (1)–(3). ALU control jedinica, redizajnirana u skladu sa prethodno izvedenim izrazima, prikazana je na slici 23.

Sumarno, postavljanjem kontrolnih signala *ALUSelA*=1, *ALUSelB*=10, ALUOp=11 kreira se novouvedeno (u odnosu na dijagram toka sa slike 18) stanje kojim se obezbjedjuje izvršavanje III koraka/taktnog intervala tokom implementacije *slti* instrukcije (stanje 11 na slici 24).

NAPOMENA: 16-bitno immediate=IR [15–0] polje instrukcije, u kome je zapisana konstanta C, produžava se, ***kopiranjem znaka***, do 32-bitne dužine da bi ostao nepromijenjen znak ove konstante, odnosno da bi se omogućio poredjenje u odnosu "manji od" i sa negativnim konstantnim operandom.



Slika 24. Mašina sa konačnim brojem stanja za implementaciju instrukcije *slti*.

2. U cilju upisa izračunatog rezultata u odredišni registar (registar označen rt=IR [20–16] poljem instrukcije) potrebno je obezbijediti sljedeće:

   2.1. Nepromjenljivost rezultata (iz prethodnog koraka/taktnog intervala) na izlazu ALU, zadržavanjem istih kontrolnih signala koji se odnose na funkcionisanje ALU, a postavljeni su u prethodnom koraku, odnosno *ALUSelA*=1, *ALUSelB*=10, ALUOp=11 (za razloge pogledaj napomenu 2 u sekciji 5.6.5 i paragraf koji se nalazi nakon napomene 2 u poglavlju 5.5.2).

   2.2. Dovesti rezultat sa izlaza ALU na Write data ulaz Registers jedinice (*MemtoReg*=0), poljem rt=IR [20–16] instrukcije označiti/adresirati registar u koji je potrebno upisati dovedeni rezultat (*RegDst*=0), te, na koncu, dozvoliti upis dovedenog rezultata u označeni registar (*RegWrite*).

   Postavljeni kontrolni signali formiraju novouvudeno (u odnosu na dijagram toka sa slike 18) stanje 12, slika 24, kojim se kompletira izvršavanje instrukcije *slti*.

Primijetimo da su za implementaciju instrukcije *slti* neophodne izmjene/redizajniranje postojeće arhitekture sa slike 11 u dijelu kontrole funkcionisanja ALU (ALU control jedinice – slika 23 i tabela 14), kao i uvodjenje novih stanja u funkcionisanje mašine sa konačnim brojem stanja kojom se implementira logika glavne kontrolne jedinice multitaktne implementacije.

5.7.3 Nadogradnja/redizajniranje arhitekture za multitaktnu implementaciju u cilju implementacije *ori* instrukcije

Podsjetimo se, naprije, simboličkog koda i mašinskog formata zapisa *ori* instrukcije. Njen simbolički kod (pogledaj poglavlje 3.6 i tabelu 1 iz poglavlja 5.1) je:

ori $x, $y, C # $x=$y | C

gdje se konstanta (konstantni operand) C čuva unutar instrukcije i zapisuje na njenom kraju, dok $x, x=1,...,31, označava registar u koji se upusuje rezultat logičke ILI (eng. OR) operacije izmedju operanda sadržanog u registru $y, y=0,...,31, i konstante C, kao što je zapisano u komentaru gore navedene instrukcije.

Mašinski format zapisa instrukcije *ori* (pogledaj poglavlje 3.6 i tabelu 1 iz poglavlja 5.1) je:

Polje:	op	rs	rt	Immediate
Br. bitova:	6	5	5	16
Sadržaj:	13	y	x	C

Slika 25. Mašinski format zapisa instrukcije *ori*.

Primijetimo da je, kod instrukcije *ori*, op=Instruction [31–26]=$13_{(10)}$=$001101_{(2)}$=0xD, gdje se oznakom 0x opisuje heksadekadni zapis. U mašinskom formatu zapisa, konstanta C smještena je na najnižim bitovima instrukcije (u 16-bitnom immediate=Instruction [15–0] polju), u rs=Instruction [25–21] polju instrukcije zapisano je obilježje registra u kome se nalazi operand ($y u simboličkoj formi), a u rt=Instruction [20–16] polju instrukcije – obilježje odredišnog registra (registar u kome se smješta rezultat logičke OR (ILI) operacije).

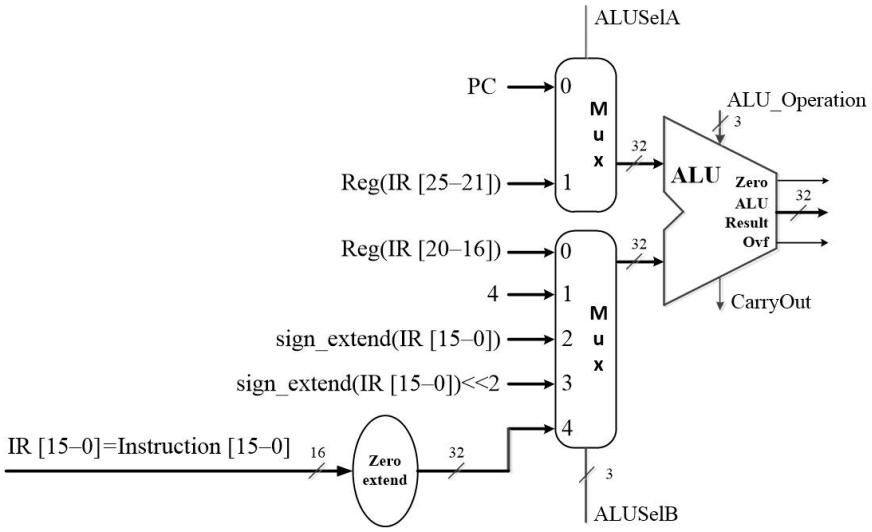
Drugim riječima, nakon uzimanja instrukcije, njenog dekodiranja i čitanja sadržaja registra označenog rs=IR [25–21] poljem instrukcije (u prva 2–zajednička–koraka/taktna intervala – stanja 0 i 1 sa slike 18), u cilju kompletiranja instrukcije *ori* potrebno je implementirati akciju:

$$\text{Reg(IR [20–16])}=\text{Reg(IR [25–21])}\ \underline{\text{OR}}\ \text{zero\_extend(IR [15–0])}, \tag{16}$$

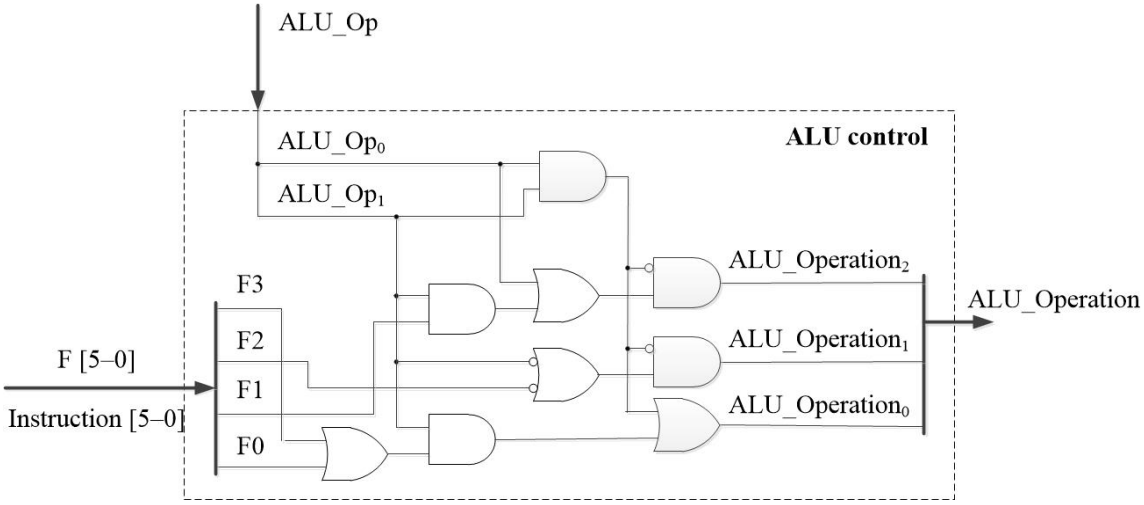
gdje je sa OR označena logička OR (ILI) operacija izmedju sadržaja registra označenog rs=IR [25–21] poljem instrukcije i 16-bitnog immediate=IR [15–0] polja instrukcije koje je produženo, kopiranjem nule, do 32-bitne dužine.

NAPOMENA 1: Logičke operacije su bit-by-bit, odnosno izvršavaju se nad svakim od bitova operanada pojedinačno i nezavisno od ostalih bitova. Drugim riječima, operandi logičkih operacija tretiraju se kao neoznačeni (unsigned) brojevi i njihovo produžavanje do odredjene dužine vrši se kopiranjem nule (eng. zero-extend – pogledaj "važno zapažanje" iz poglavlja 4), a implementira se zero-extend jedinicom.

POSLJEDICA: U cilju implementacije immediate instrukcija koje zahtijevaju izvršavanja logičkih operacija, neophodno je proširiti multipleksor na II ulazu ALU da bi se na njegov dodati ulaz doveo sadržaj 16-bitnog immediate=IR [15–0] polja instrukcije produžen, **kopiranjem nule (upotrebom Zero extend jedinice)**, do 32-bitne dužine, kao što je prikazano na slici 26. Shodno tome, prošireni multipleksor treba da ima 3 selekciona ulaza, na koje se dovodi 3-bitni *ALUSelB* kontrolni signal, te je sa ovom činjenicom (da je *ALUSelB* kontrolni signal multipleksora na II ulazu ALU – 3-bitni signal) potrebno uskladiti postavljanje ovog kontrolnog signala u prva 2–zajednička–koraka/taktna intervala. Primijetimo, uz to, da prva 4 ulaza ovog multipleksora ostaju nepromijenjena u odnosu na sliku 11, samo što se, u izmijenjenoj arhitekturi, adresiraju/selektiraju 3-bitnim kontrolnim signalom *ALUSelB*.



Slika 26. Izmjene arhitekture za multitaktnu implementaciju sa slike 11 na drugom ulazu ALU u cilju implementacije immediate instrukcija koje zahtijevaju izvršavanje logičkih operacija.



Slika 27. ALU control jedinica, redizajnirana u cilju generisanja ALU_Operation signala potrebnih za kontrolu izvršavanja operacija zahtijevanih instrukcijama iz tabele 7, ali i dodatom *ori* instrukcijom. Elementi, dodati u cilju redizajniranja (u odnosu na dizajn sa slike 7), istaknuti su sijenčenjem.

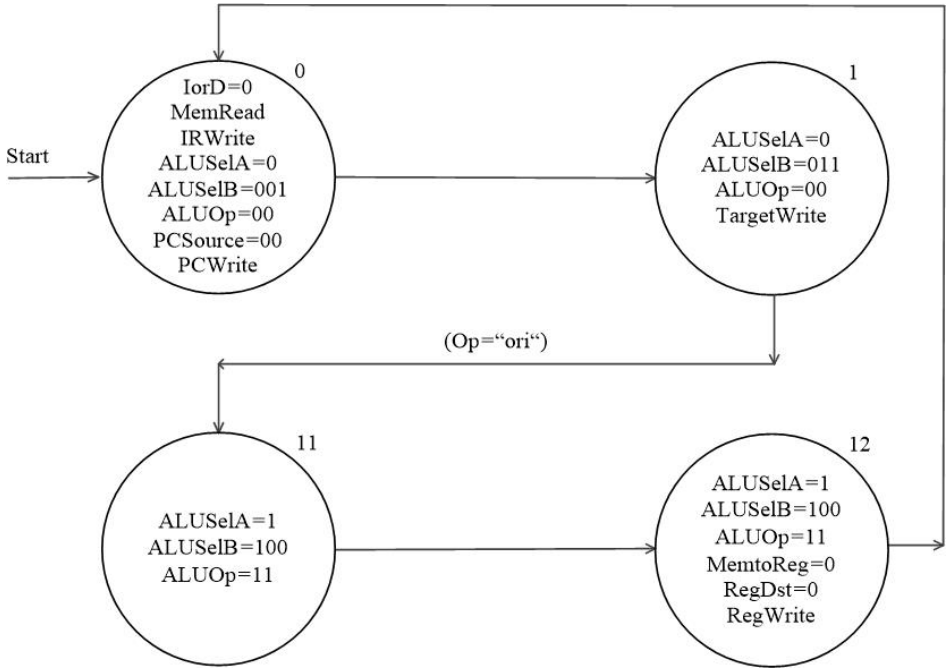
Tabela 15. Funkcionalna tabela ALU control jedinice sa uključenom operacijom koju zahtijeva izvršavanje instrukcije *ori*.

Instr.	ALUOp		Funct (Instruction [5–0])						ALU_Operation		
	1	0	F5	F4	F3	F2	F1	F0	2	1	0
lw, *sw*	0	0	×	×	×	×	×	×	0	1	0
beq	0	1	×	×	×	×	×	×	1	1	0
add	1	0	×	×	0	0	0	0	0	1	0
sub	1	0	×	×	0	0	1	0	1	1	0
and	1	0	×	×	0	1	0	0	0	0	0
or	1	0	×	×	0	1	0	1	0	0	1
slt	1	0	×	×	1	0	1	0	1	1	1
ori	1	1	×	×	×	×	×	×	0	0	1

Shodno činjenicama detaljno razmotrenim u primjedbi iz sekcije 5.7.1, akcija (16) zahtijeva izvršavanje u 2 koraka/taktna intervala (III i IV). U III koraku/taktnom intervalu, izračunava se logička OR (ILI) operacija, zahtijevana akcijom (16), dok se u IV koraku/taktnom intervalu, izračunati rezultat upisuje u odredišni registar Reg(IR [20–16]).

1. U cilju izračunavanja operacije zahtijevane akcijom (16), na prvi ulaz ALU potrebno je dovesti sadržaj registra koji je označen rs=IR [25–21] poljem instrukcije (*ALUSelA*=1), na drugi ulaz ALU – 16-bitno immediate=IR [15–0] polje instrukcije produženo, **kopiranjem nule**, do 32-bitne dužine (*ALUSelB*=100), te omogućiti ALU da izvrši operaciju OR (ILI).

   Primijetimo da je ALU dizajnirana za izvršavanje operacije OR/ILI (drugim riječima, nije je potrebno redizajnirati), ali da je, u aktuelnoj implementaciji ALU control jedinice iz sekcije 5.4.1, postavljanje ove oparacije kontrolisano kombinacijom bitova ALUOp=10, odnosno funct=IR [5–0] poljem instrukcije (pogledaj "pojašnjenje" iz sekcije 5.7.2), kojim ne raspolažu immediate instrukcije (time ni *ori* instrukcija). Drugim riječima, za izvršavanje akcije (16), potrebno je obezbijediti izvršavanje logičke OR/ILI operacija koje neće biti kontrolisano funct=IR [5–0] poljem. U tom cilju, upotrebljava se kombinacija bitova ALUOp=11, kao što je prikazano u tabeli 15, koja nije upotrijebljena prilikom dizajniranja ALU control jedinice u sekciji 5.4.1. (Naravno, ova kombinacija ALUOp bitova može biti upotrijebljena isključivo pod pretpostavkom da nije ranije upotrijebljena – prilikom implementacije instrukcije *slti*. Mogućnost redizajniranja postojeće arhitekture u cilju istovremene implementacije instrukcija *slti* i *ori* razmatrana je u napomeni 2 iz ove sekcije).



Slika 28. Mašina sa konačnim brojem stanja za implementaciju instrukcije *ori*. Primijetimo da je u stanjima 0 i 1 uskladjeno postavljanje *ALUSelB* kontrolnog signala sa 3-bitnom strukturom ovog signala koja je posljedica redizajniranja arhitekture prikazanog na slici 26.

U skladu sa upotrebom kombinacije ALUOp=11 u svrhu kontrole operacije zahtijevane *ori* instrukcijom, ALU control jedinica iz sekcije 5.4.1 mora biti redizajnirana tako da obezbijedi generisanje $\text{ALU\_Operation}_{(2,1,0)}=(0,0,1)$ za ALUOp=11. U tom cilju, primijetimo da funkcionalna tabela 15 ALU control jedinice u odnosu na tabele 7 i 8 iz sekcije 5.4.1 sadrži samo jedan dodatni red više (za kombinaciju bitova ALUOp=11 – red odvojen debljom isprekidanom linijom), tako da je:

$$\text{ALU\_Operation}_{(2,1)} = \text{ALU\_Operation}_{(2,1)}^{\text{staro}} \wedge \overline{\text{ALUOp}_1 \wedge \text{ALUOp}_0}$$

$$\text{ALU\_Operation}_0 = \text{ALU\_Operation}_0^{\text{staro}} + \text{ALUOp}_1 \wedge \text{ALUOp}_0$$

gdje kontrolni signali $\text{ALU\_Operation}_{(2,1,0)}^{\text{staro}}$ odgovaraju dizajnu ALU control jedinice sa slike 7 i već su izvedeni u sekciji 5.4.1 i dati izrazima (1)–(3). ALU control jedinica, redizajnirana u skladu sa prethodno izvedenim izrazima, prikazana je na slici 27.

Sumarno, postavljanjem kontrolnih signala *ALUSelA*=1, *ALUSelB*=100, ALUOp=11 kreira se novouvedeno (u odnosu na dijagram toka sa slike 18) stanje kojim se obezbjedjuje izvršavanje III koraka/taktnog intervala tokom implementacije *ori* instrukcije (stanje 11 na slici 28).

NAPOMENA 2: Redizajniranje arhitekure sa slike 11 u cilju pojedinačne implementacije instrukcija *slti* i *ori* izvršeno je uz kontrolu funkcionisanja ALU sa istom kombinacijom ALUOp signala (ALUOp=11). Naravno, ovo je moguće samo pod pretpostavkom pojedinačne implementacije svake od ovih instrukcija (kao da se ona druga instrukcija ne implementira). Redizajniranje arhitekture sa slike 11, ali u cilju jednovremene implementacije *slti* i *ori* instrukcija, koje obje zahtijevaju upotrebu dodatne kombinacije ALUOp bitova, zahtijevalo bi uvodjenje dodatnog ALUOp_2 bita, tako da bi se kontrola funkcionisanja ALU obavljala 3-bitnim ALUOp signalom. U tom slučaju bi, na primjer, kombinacija bitova ALUOp=011 mogla biti rezervisana za kontrolu bezuslovnog izvršavanja (od strane ALU) OR/ILI logičke operacije, a kombinacija ALUOp=100 – za kontrolu bezuslovnog izvršavanja set-on-less-than operacije. Naravno, kombinacije bitova ALUOp=000, ALUOp=001 i ALUOp=010 odgovarale bi ranije utvrdjenim namjenama (bezuslovnom sabiranju, bezuslovnom

oduzimanju i izvršavanju operacija zahtijevanih instrukcijama R-tipa – pogledaj poglavlje 5.4.1), dok bi kombinacije ALUOp=101, ALUOp=110 i ALUOp=111 mogle biti upotrijebljene za bezuslovno izvršavanje neke nove ili neke od preostalih operacija ALU koje su zahtijevane od strane instrukcija R-tipa (na primjer logičke AND operacije – sekcija 5.7.4).

2. U cilju upisa izračunatog rezultata u odredišni registar (registar označen rt=IR [20–16] poljem instrukcije) potrebno je obezbijediti sljedeće:

   2.1. Nepromjenljivost rezultata (iz prethodnog koraka/taktnog intervala) na izlazu ALU, zadržavanjem istih kontrolnih signala koji se odnose na funkcionisanje ALU, a postavljeni su u prethodnom koraku, odnosno *ALUSelA*=1, *ALUSelB*=100, ALUOp=11 (za razloge pogledaj napomenu 2 u sekciji 5.6.5 i paragraf koji se nalazi nakon napomene 2 u poglavlju 5.5.2).

   2.2. Dovesti rezultat sa izlaza ALU na Write data ulaz Registers jedinice (*MemtoReg*=0), poljem rt=IR [20–16] instrukcije označiti/adresirati registar u koji je potrebno upisati dovedeni rezultat (*RegDst*=0), te, na koncu, dozvoliti upis dovedenog rezultata u označeni registar (*RegWrite*).

   Postavljeni kontrolni signali formiraju novouvudeno (u odnosu na dijagram toka sa slike 18) stanje 12, slika 28, kojim se kompletira izvršavanje instrukcije *ori*.

Primijetimo da su za implementaciju instrukcije *ori* neophodne izmjene/redizajniranje postojeće arhitekture sa slike 11 i to

- proširivanje multipleksora koji se nalazi na drugom ulazu ALU (slika 26),
- uključivanje u dizajn Zero extend jedinice (slika 26) i
- izmjene u dijelu kontrole funkcionisanja ALU (ALU control jedinice – slika 27 i tabela 15),

kao i uvodjenje novih stanja u funkcionisanje mašine sa konačnim brojem stanja kojom se implementira logika glavne kontrolne jedinice multitaktne implementacije.

5.7.4 Nadogradnja/redizajniranje arhitekture za multitaktnu implementaciju u cilju implementacije *andi* instrukcije

Podsjetimo se, naprije, simboličkog koda i mašinskog formata zapisa *andi* instrukcije. Njen simbolički kod (pogledaj poglavlje 3.6 i tabelu 1 iz poglavlja 5.1) je:

andi $x, $y, C # $x=$y & C

gdje se konstanta (konstantni operand) C čuva unutar instrukcije i zapisuje na njenom kraju, dok $x, x=1,...,31, označava registar u koji se upusuje rezultat logičke I (eng. AND) operacije izmedju operanda sadržanog u registru $y, y=0,...,31, i konstante C, kao što je zapisano u komentaru gore navedene instrukcije.

Mašinski format zapisa instrukcije *andi* (pogledaj poglavlje 3.6 i tabelu 1 iz poglavlja 5.1) je:

Polje:	op	rs	rt	Immediate
Br. bitova:	6	5	5	16
Sadržaj:	12	y	x	C

Slika 29. Mašinski format zapisa instrukcije *andi*.

Primijetimo da je, kod instrukcije *andi*, op=Instruction [31–26]=$12_{(10)}$=$001100_{(2)}$=0xC, gdje se oznakom 0x opisuje heksadekadni zapis. U mašinskom formatu zapisa, konstanta C smještena je na najnižim bitovima instrukcije (u 16-bitnom immediate=Instruction [15–0] polju), u rs=Instruction [25–21] polju instrukcije zapisano je obilježje registra u kome se nalazi operand ($y u simboličkoj formi), a u rt=Instruction [20–16] polju instrukcije – obilježje odredišnog registra (registar u kome se smješta rezultat logičke AND operacije).

Drugim riječima, nakon uzimanja instrukcije, njenog dekodiranja i čitanja sadržaja registra označenog rs=IR [25–21] poljem instrukcije (u prva 2–zajednička–koraka/taktna intervala – stanja 0 i 1 sa slike 18), u cilju kompletiranja instrukcije *andi* potrebno je implementirati akciju:

$$\text{Reg(IR [20–16])}=\text{Reg(IR [25–21])}\ \underline{\text{AND}}\ \text{zero\_extend(IR [15–0])}, \qquad (17)$$

gdje je sa AND označena logička AND operacija izmedju sadržaja registra označenog rs=IR [25–21] poljem instrukcije i 16-bitnog immediate=IR [15–0] polja instrukcije koje je produženo, kopiranjem nule, do 32-bitne dužine.

NAPOMENA: Kao i OR operacija, AND je takodje logička bit-by-bit operacija (izvršava se nad svakim od bitova operanada pojedinačno i nezavisno od ostalih bitova). Drugim riječima, operandi se tretiraju kao neoznačeni (unsigned) brojevi i njihovo produžavanje do odredjene dužine vrši se kopiranjem nule (eng. zero-extend – pogledaj "važno zapažanje" iz poglavlja 4, kao i napomenu 1 iz sekcije 5.7.3), a implementira se zero-extend jedinicom.

POSLJEDICA: U cilju implementacije immediate instrukcija koje zahtijevaju izvršavanja logičkih operacija, time i *andi* instrukcije, neophodno je proširiti multipleksor na II ulazu ALU da bi se na njegov dodati ulaz doveo sadržaj 16-bitnog immediate=IR [15–0] polja instrukcije produžen, **kopiranjem nule (upotrebom Zero extend jedinice)**, do 32-bitne dužine, kao što je prikazano na slici 26. Shodno tome, prošireni multipleksor treba da ima 3 selekciona ulaza, na koje se dovodi 3-bitni *ALUSelB* kontrolni signal, te je sa ovom činjenicom (da je *ALUSelB* kontrolni signal multipleksora na II ulazu ALU – 3-bitni signal) potrebno uskladiti postavljanje ovog kontrolnog signala u prva 2–zajednička–koraka/taktna intervala. Primijetimo, uz to, da prva 4 ulaza ovog multipleksora ostaju nepromijenjena u odnosu na sliku 11, samo što se, u izmijenjenoj arhitekturi, adresiraju/selektiraju 3-bitnim kontrolnim signalom *ALUSelB*.

Shodno činjenicama detaljno razmotrenim u primjedbi iz sekcije 5.7.1, akcija (17) zahtijeva izvršavanje u 2 koraka/taktna intervala (III i IV). U III koraku/taktnom intervalu, izračunava se logička AND operacija, zahtijevana akcijom (17), dok se u IV koraku/taktnom intervalu, izračunati rezultat upisuje u odredišni registar Reg(IR [20–16]).

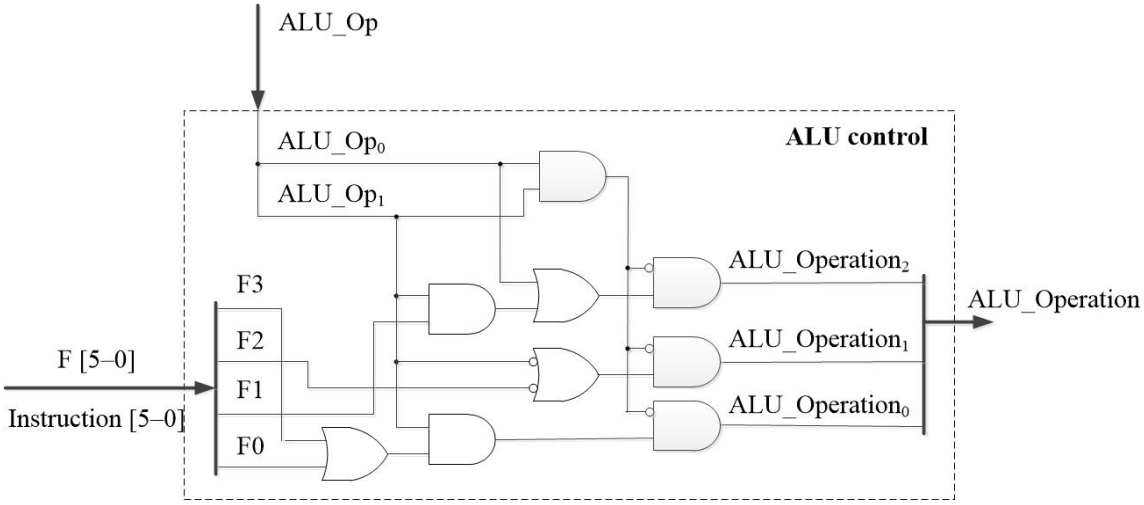
1. U cilju izračunavanja operacije zahtijevane akcijom (17), na prvi ulaz ALU potrebno je dovesti sadržaj registra koji je označen rs=IR [25–21] poljem instrukcije (*ALUSelA*=1), na drugi ulaz ALU – 16-bitno immediate=IR [15–0] polje instrukcije produženo, **kopiranjem nule**, do 32-bitne dužine (*ALUSelB*=100), te omogućiti ALU da izvrši operaciju AND.

   Primijetimo da je ALU dizajnirana za izvršavanje operacije AND (dakle, nije je potrebno redizajnirati), ali da je, u aktuelnoj implementaciji ALU control jedinice iz sekcije 5.4.1, postavljanje ove oparacije kontrolisano kombinacijom bitova ALUOp=10, odnosno funct=IR [5–0] poljem instrukcije (pogledaj "pojašnjenje" iz sekcije 5.7.2), kojim ne raspolažu immediate instrukcije (time ni *andi* instrukcija). Drugim riječima, za izvršavanje akcije (17), potrebno je obezbijediti izvršavanje logičke AND operacija koje neće biti kontrolisano funct=IR [5–0] poljem. U tom cilju, upotrebljava se kombinacija bitova ALUOp=11, kao što je prikazano u tabeli 16, koja nije upotrijebljena prilikom dizajniranja ALU control jedinice u sekciji 5.4.1. (Naravno, ova kombinacija ALUOp bitova može biti upotrijebljena isključivo pod pretpostavkom da nije ranije upotrijebljena – prilikom implementacije instrukcija *slti* i *ori*. Mogućnost redizajniranja postojeće arhitekture u cilju istovremene implementacije instrukcija *slti*, *ori* i *andi* razmatrana je u napomeni 2 iz sekcije 5.7.3).

   U skladu sa upotrebom kombinacije ALUOp=11 u svrhu kontrole operacije zahtijevane *andi* instrukcijom, ALU control jedinica iz sekcije 5.4.1 mora biti redizajnirana tako da obezbijedi generisanje $\text{ALU\_Operation}_{(2,1,0)}=(0,0,0)$ za ALUOp=11. U tom cilju, primijetimo da funkcionalna tabela 16 ALU control jedinice u odnosu na tabele 7 i 8 iz sekcije 5.4.1 sadrži samo jedan dodatni red više (za kombinaciju bitova ALUOp=11 – red odvojen debljom isprekidanom linijom), tako da je:

$$\text{ALU\_Operation}_{(2,1,0)} = \text{ALU\_Operation}^{\text{staro}}_{(2,1,0)} \wedge \overline{\text{ALUOp}_1 \wedge \text{ALUOp}_0}$$

   gdje kontrolni signali $\text{ALU\_Operation}^{\text{staro}}_{(2,1,0)}$ odgovaraju dizajnu ALU control jedinice sa slike 7 i već su izvedeni u sekciji 5.4.1 i dati izrazima (1)–(3). ALU control jedinica, redizajnirana u skladu sa prethodno izvedenim izrazima, prikazana je na slici 30.



Slika 30. ALU control jedinica, redizajnirana u cilju generisanja ALU_Operation signala potrebnih za kontrolu izvršavanja operacija zahtijevanih instrukcijama iz tabele 7, ali i dodatom *andi* instrukcijom. Elementi, dodati u cilju redizajniranja (u odnosu na dizajn sa slike 7), istaknuti su sijenčenjem.

Tabela 16. Funkcionalna tabela ALU control jedinice sa uključenom operacijom koju zahtijeva izvršavanje instrukcije *andi*.

Instr.	ALUOp		Funct (Instruction [5–0])						ALU_Operation		
	1	0	F5	F4	F3	F2	F1	F0	2	1	0
lw, *sw*	0	0	×	×	×	×	×	×	0	1	0
beq	0	1	×	×	×	×	×	×	1	1	0
add	1	0	×	×	0	0	0	0	0	1	0
sub	1	0	×	×	0	0	1	0	1	1	0
and	1	0	×	×	0	1	0	0	0	0	0
or	1	0	×	×	0	1	0	1	0	0	1
slt	1	0	×	×	1	0	1	0	1	1	1
andi	1	1	×	×	×	×	×	×	0	0	0

Sumarno, postavljanjem kontrolnih signala *ALUSelA*=1, *ALUSelB*=100, ALUOp=11 kreira se novouvedeno (u odnosu na dijagram toka sa slike 18) stanje kojim se obezbjedjuje izvršavanje III koraka/taktnog intervala tokom implementacije *andi* instrukcije (stanje 11 na slici 31).

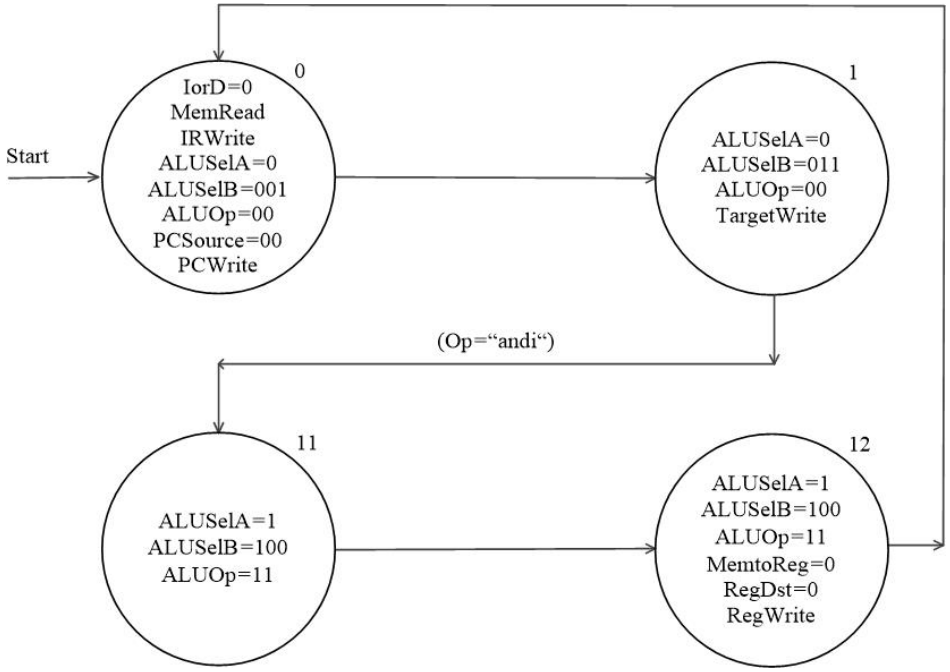
2. U cilju upisa izračunatog rezultata u odredišni registar (registar označen rt=IR [20–16] poljem instrukcije) potrebno je obezbijediti sljedeće:

   2.1. Nepromjenljivost rezultata (iz prethodnog koraka/taktnog intervala) na izlazu ALU, zadržavanjem istih kontrolnih signala koji se odnose na funkcionisanje ALU, a postavljeni su u prethodnom koraku, odnosno *ALUSelA*=1, *ALUSelB*=100, ALUOp=11 (za razloge pogledaj napomenu 2 u sekciji 5.6.5 i paragraf koji se nalazi nakon napomene 2 u poglavlju 5.5.2).

   2.2. Dovesti rezultat sa izlaza ALU na Write data ulaz Registers jedinice (*MemtoReg*=0), poljem rt=IR [20–16] instrukcije označiti/adresirati registar u koji je potrebno upisati dovedeni rezultat (*RegDst*=0), te, na koncu, dozvoliti upis dovedenog rezultata u označeni registar (*RegWrite*).

   Postavljeni kontrolni signali formiraju novouvudeno (u odnosu na dijagram toka sa slike 18) stanje 12, slika 31, kojim se kompletira izvršavanje instrukcije *andi*.

Primijetimo, na koncu, da su za implementaciju instrukcije *andi* neophodne izmjene/redizajniranje postojeće arhitekture sa slike 11 i to



Slika 31. Mašina sa konačnim brojem stanja za implementaciju instrukcije *andi*. Primijetimo da je u stanjima 0 i 1 uskladjeno postavljanje ALUSelB kontrolnog signala sa 3-bitnom strukturom ovog signala koja je posljedice redizajniranja arhitekure prikazane na slici 30.

- proširivanje multipleksora koji se nalazi na drugom ulazu ALU (slika 26),
- uključivanje u dizajn Zero extend jedinice (slika 26) i
- izmjene u dijelu kontrole funkcionisanja ALU (ALU control jedinice – slika 30 i tabela 16),

kao i uvodjenje novih stanja u funkcionisanje mašine sa konačnim brojem stanja kojom se implementira logika glavne kontrolne jedinice multitaktne implementacije.

VJEŽBE

RIJEŠENI ISPITNI ZADACI

18. Modifikovati datapath i controlpath da bi se mogla implementirati instrukcija *jal*. Prikazati modifikaciju dijagrama stanja za implementiranje ove instrukcije.

Rješenje:

Prilikom realizacije instrukcija koje nijesu realizovane postojećom multitaktnom arhitekturom, prvi korak je da prepoznamo tip instrukcije koju je potrebno realizovati, i da nađemo njoj najsličniju instrukciju, a čija realizacija postoji.

Zadata *jal* instrukcija pripada *j* tipu instrukcija. U izučavanoj arhitekturi postoji realizacija *j* instrukcije, odnosno osnovne instrukcije skoka. Posmatrajmo dijagram stanja. *j* instrukcija se izvršava u 3 takta (stanja 0, 1 i 9).

Uočimo da su stanja 0 i 1 zajednička za sve relizovane instrukcije. Stoga, i kada budemo relizovali nove instrukcije, ova stanja ćemo zadržavati, tj. **u njima neće biti promjena!** Takodje, prilikom realizacije novih instrukcija, **ne smije** se izvršiti promjena u datapath/controlpath/dijagramu stanja koja uzrokuje da neka od postojećih, već realizovanih funkcija ne radi ispravno!

Po čemu se razlikuju instrukcije *j* i *jal*?

I jedan i druga instrukcija vrše skok na određenu labelu (adresu). Međutim, osim skoka instrukcija *jal* vrši i „povezivanje" (*jal – jump and link*), odnosno pamti adresu sljedeće instrukcije (one koja se vrši neposredno nakon nje) i smješta tu adresu u $31. Podsjećanja radi, upotrebom *jal* instrukcije se vrši skok na proceduru, a činjenica da se pamti adresa sljedeće instrukcije nam omogućava povratak nazad na tačnu lokaciju nakon izvršenja pozvane procedure.

Šta ovo znači u smislu realizacije *jal* instrukcije? Osim skoka koji vrši već realizovana *j* instrukcija, potrebno je obezbjediti smještanje sljedeće adrese u $31.

Ko čuva informaciju o sljedećoj adresi? To je PC (Program Counter) registar. Dakle, sadržaj ovog registra moramo proslijediti u $31. Da bismo mogli (bilo koji) podatak smjestiti u $31, moramo omogućiti upis $31.

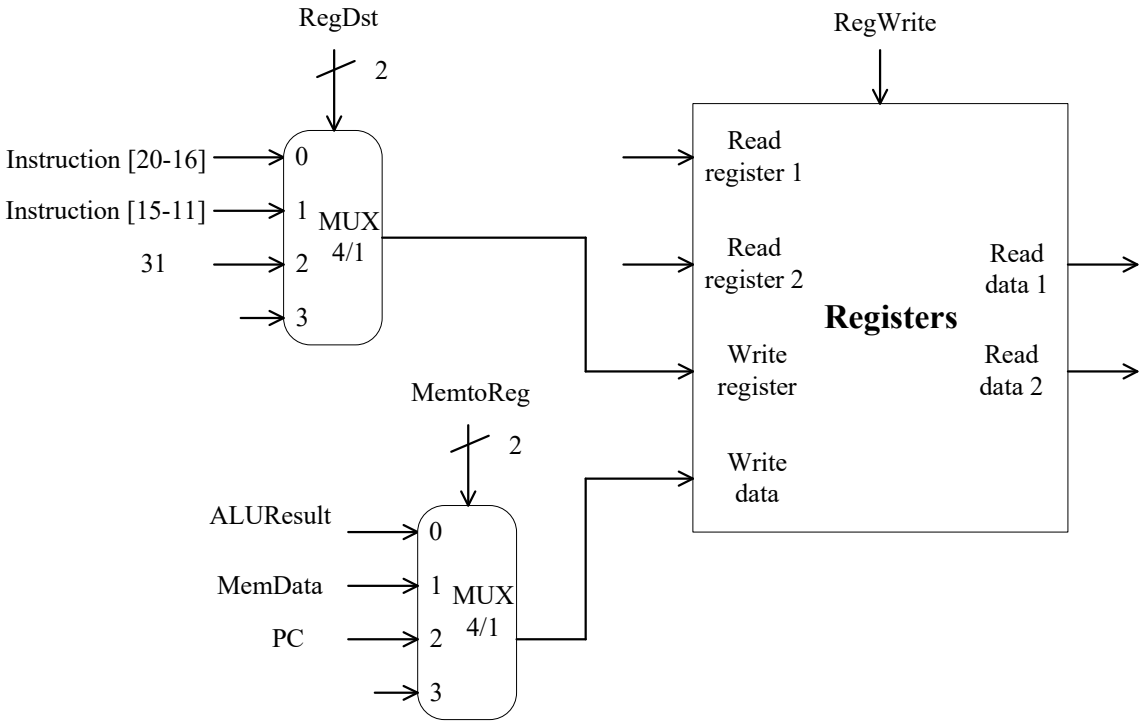
Obratite pažnju na Registers jedinicu u izučavanoj višetaktnoj realizaciji. Posljednja dva ulaza u Registers jedinicu su *Write register* i *Write data*. Na ulaz *Write register* se dovodi registar u koji se želi izvršti upis, a na ulaz *Write data* se dovodi podatak čiji upis se vrši. Dakle, u našem slučaju na ulaz *Write register* je potrebno dovesti $31, a na ulaz *Write data* je potrebno dovesti PC. Na koji način stižu podaci na ova dva ulaza Registers jedinice? Uočavamo da stižu iz izlaza dva multipeksora. Oba multipleksora su MUX 2/1 (podsjetite se principa rada MUXeva) i oba ova MUXa su iskorišćena u punom kapacitetu. To znači da na njihovim ulazima već postoje određeni podaci. Mogli bismo neke od tih podataka zamijeniti željenim podacima ($31 i PC), ali bismo u tom slučaju onemogućili rad jedne ili više standarnih instrukcija, što **nije dozvoljeno**. Preostaje nam da proširimo uočene MUXeve, tako da osim postojećih podataka mogu primiti i podatke koji su nama od interesa. Postojeći MUXevi imaju po dva ulaza, a da bi primili potrebne podatke neophodna su im po tri ulaza. To znači da ćemo umjesto MUXeva 2/1 koristiti MUXeve 4/1. Uočite da novi MUXevi 4/1 neće biti iskorišćeni u punom kapacitetu (od 4 ulaza nama će biti potrebna samo 3, i to 2 za postojeće podatke i 1 za novi podatak). Osim toga, MUX 4/1 ima 2 kontrolna/selekciona signala (za razliku od MUX 2/1 koji ima jedan kontrolni/selekcioni signal). Dakle, signali koji kontrolišu posmatrana dva MUX 2/1 (signali *RegDst* i *MemtoReg*) će umjesto dosadašnjih jednobitnih vrijednosti uzimati dvobitne vrijednosti.

U nastavku je data grafički objašnjena izmjena u postojećoj arhitekturi.

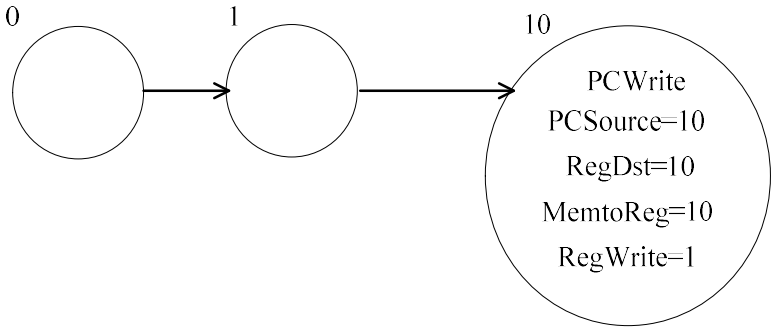
Ostatak arhitetkure nije mijenjan, te stoga nije ni predstavljen na slici.

Šta se dešava sa dijagramom stanja?

Kao što je već rečeno, stanja 0 i 1 su ista za sve instrukcije, i u njima nema promjena. Treći takt će sadržati sve ono što sadrži treći takt instrukcije *j* (stanje 9), jer nam se na taj način omogućava funkcija skoka. Uz to, sadržaće kontrolne signale koje smo koristili da bismo obezbijedili „povezivanje". To su signali *RegDst* i *MemtoReg*, koji će uzimati vrijednosti 10. Zašto ove vrijednosti? Zato što je neophodno propustiti ulaze 2 (binarno 10) sa posmatranih MUXeva na odgovarajuće izlaze. Postoji još jedan signal koji moramo setovati (postaviti na 1). To je signal *RegWrite*, koji omogućava upotrebu Registers jedinice. Kako registers jedinica nije bila potrebna u realizaciji *j* instrukcije, to se ovaj signal nije upotrebljavao. Međutim, prilikom realizacije *jal* instrukcije intenzivno koristimo Registers jedinicu, te nam je neophodno setovati *RegWrite*, i na taj način dozvoliti upisivanje u istu.



Dijagram stanja *jal* instrukcije će stoga izgledati ovako:



Obratite pažnju na numeraciju stanja. Posljednje stanje smo označili kao stanje 10, zato što je u pitanju novo stanje, i svojstveno samo intrukciji *jal*. Pogrešno bi bilo označiti ga kao 9, jer bi na taj način oštetili dijagram stanja instrukcije *j*. Kako nema izmjena u stanju 0 i 1, za njih koristimo iste oznake (i nije potrebno prepisivati njihove sadržaje).

Napomena: Ako je u dijagramu stanja neki signal naveden, a ne piše koja mu je vrijednost, podrazumijeva se da mu je vrijednost 1.

19. Modifikovati datapath i controlpath da bi se mogla implementirati instrukcija *nor*. Šta se dešava sa dijagramom stanja za implementiranje ove instrukcije?

Rješenje:

nor je instrukcija R tipa, pa ćemo se prilikom njene relizacije osloniti na instrukcije R tipa. Sve instrukcije R tipa se realizuju upotrebom ALU. Potrebno je znati strukturu jednobitne ALU. Jednobitna ALU realizuje 4 instrukcije (*and, or, add* i *slt*), te je MUX 4/1 koji ona upotrebljava iskorišćen u maksimalnom kapacitetu. Pošto tražena instrukcija *nor* nije realizovana, a potrebna nam je ALU za njenu realizaciju, proširićemo MUX 4/1, tako da na ulaz možemo dovesti izlaz iz osnovnog NILI (nor) kola. Kada proširujemo MUX 4/1 da bi dobili potrebni peti ulaz, jasno je da ćemo morati upotrijebiti MUX 8/1 (prvi sljedeći po veličini). To znači da ćemo imati tri neiskorišćena ulaza (što nam nije bitno), ali i da ćemo morati dodati još jedan selekcioni/kontrolni signal (MUX 4/1 je kontrolisan sa 2, a MUX 8/1 sa 3 selekciona/kontrolna signala). Na koji način će biti definisan dodatni kontrolni signal? Da bismo to riješili, pogledajmo najprije tabelu signala koji figurišu u ALU (ova tabela je sastavni dio dostupne literature):

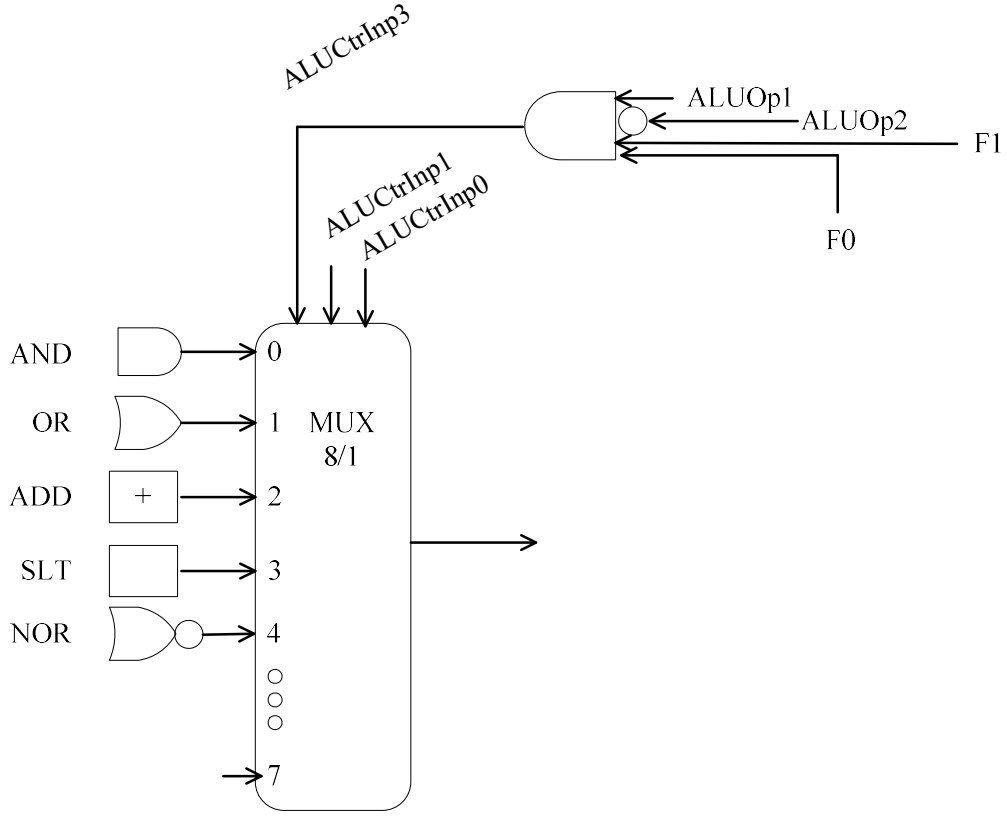
	ALUop		Function						ALU Control Inputs		
	ALUop1	ALUop2	F5	F4	F3	F2	F1	F0	2	1	0
lw, sw	0	0	x	x	x	x	x	x	0	1	0
beq	0	1	x	x	x	x	x	x	1	1	0
add	1	0	1	0	0	0	0	0	0	1	0
sub	1	0	1	0	0	0	1	0	1	1	0
and	1	0	1	0	0	1	0	0	0	0	0
or	1	0	1	0	0	1	0	1	0	0	1
slt	1	0	1	0	1	0	1	0	1	1	1

Pojašnjenje tabele:

Sa krajnje lijeve strane su navedene instrukcije koje koriste ALU. Uočićete to nijesu samo intrukcije R tipa. Posmatrajmo njihova *ALUop* polja: za sve instrukcije R tipa *ALUop* polja imaju vrijednost 10. To znači da će i naša instrukcija *nor* imati vrijednost 10 na poziciji ovih polja. *Function* polja su polja po kojem se sve instrukcije R tipa međusobno razlikuju, i postoje samo kod ovih instrukcija (primjetićete u tabeli da instrukcije *lw, sw, beq* nemaju vrijednosti na ovim poljima, te je stoga upisano x). Koje vrijednosti se upisuju na ovim pozicijama? Pogledajte kompletan spisak instrukcija. Uočićete npr. da *add* instrukcija ima funkcijsko polje 0x20. Oznaka 0x znači da je podatak nakon nje napisan u heksadecimalnom brojnom sistemu. Kada ovu vrijednost pretvorimo u binarni brojni sistem dobijamo: $24_{(16)}=10000_{(2)}$. Ovo su upravo biti iz gornje tabele. Instrukcija *and* ima funkcijsko polje 0x24, odnosno binarno 100100. Slično, naša instrukcija *nor* ima funkcijsko polje 0x27, odnosno njeni funkcijski biti su 100111. Preostale bite iz tabele, *ALU Control Inputs* bite tumačimo na sljedeći način: bit sa oznakom 2 označava da li u se u navedenoj operaciji dešava oduzimanje (1 ako se dešava, 0 u suprotnom); biti sa oznakama 1 i 0 su biti koji predstavljaju selekcione/kontrolne signale MUXa u posmatranoj ALU. Dakle, prikom proširenja MUXa sa MUX 4/1 na MUX 8/1, biće nam potreban još jedan ovakav bit (bit sa oznakom 3), kako bismo selekcione/kontrolne priključke setovali na 100, i omogući da ono što je na ulazu 4 (izlaz is nor kola) proslijedimo na izlaz MUXa 8/1, odnosno kompletne ALU.

	ALUop		Function						ALU Control Inputs			
	ALUop1	ALUop2	F5	F4	F3	F2	F1	F0	3	2	1	0
nor	1	0	1	0	0	1	1	1	1	0	0	0

Zašto smo bit sa oznakom 2 stavili na 0? Zato što u operaciji nor nema oduzimanja. Dakle, novododati bit 3 će uzeti vrijednost 0 za sve ostale instrukcije iz tabele, osim za našu instrukciju *nor* kada će imati vrijednost 1. Kako ćemo postići vrijednost 1 za novododato polje sa oznakom 3?

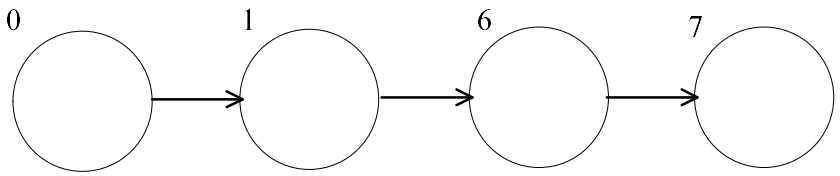


$$ALUCtrlInp3 = ALUop1 * \overline{ALUop2} * F5 * \overline{F4} * \overline{F3} * F2 * F1 * F0$$

Ili jednostavnije – možemo uočiti iz gornje tabele da ni jedna druga instrukcija (osim naše) nema na poljima F1 i F0 istovremeno 1. To znači da će ta kombinacija biti dovoljna da razlikuje novouvedeni signal od ostalih, odnosno:

$$ALUCtrlInp3 = ALUop1 * \overline{ALUop2} * F1 * F0$$

Dakle, proširenjem ALU i podešavanjem odgovarajućih signala smo postigli realizaciju instrukcije *nor*. Pri tome smo dodali samo jedan signal, *ALUCtrlInp3* koji nije sastavni dio dijagrama stanja. To znači da nam je dijagram stanja ostao isti kao za sve instrukcije R tipa, odnosno:



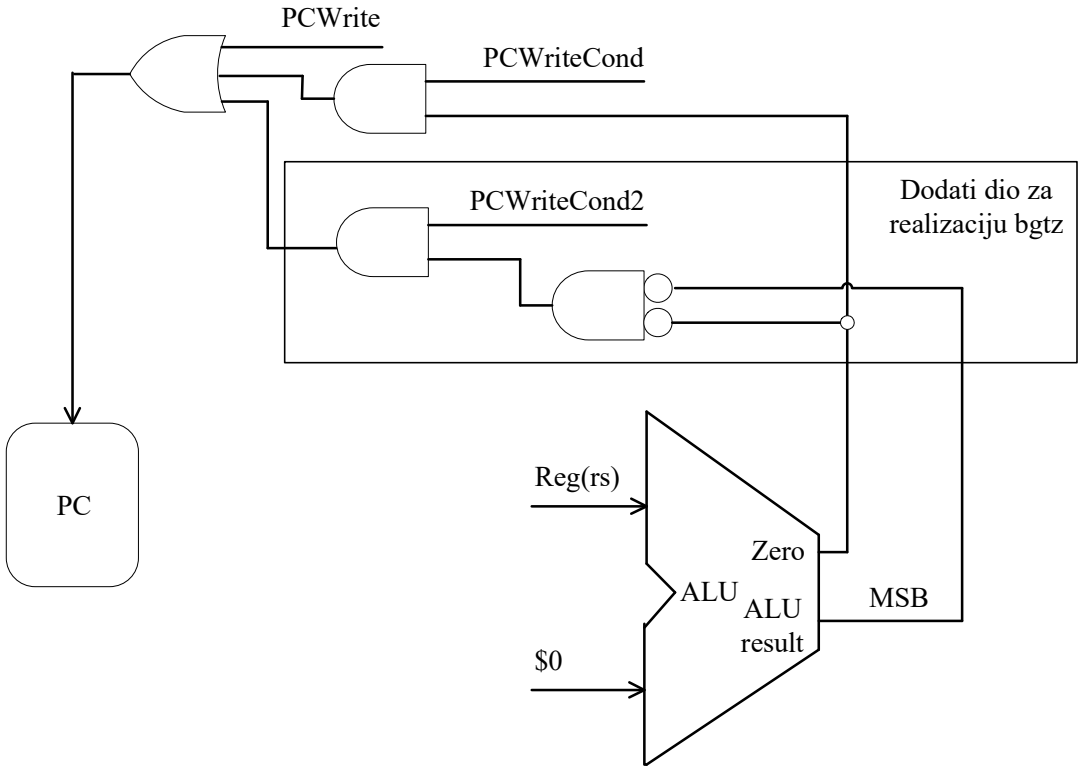
20. Modifikovati datapath i controlpath da bi se mogla implementirati instrukcija *bgtz*. Prikazati modifikaciju dijagrama stanja za implementiranje ove instrukcije.

Rješenje:

bgtz je branch instrukcija. Kod branch instrukcija se poređenje dvije vrijednosti obavlja tako što se one oduzmu (upotrebom ALU), a potom se provjeri rezultat. ALU ima dva izlaza (pogledajte šemu izučavane multitaktne realizacije). To su *ALU result* i *Zero*. *ALU Result* (očekivano) prikazuje rezultat operacije koju je obavljala ALU. Zero izlaz uzima jednu od moguće dvije vrijednosti: 1, ukoliko je rezultat oduzimanja jednak 0, i 0 ako rezultat oduzimanja nije jednak nuli. Kod instrukcija *beq* i *bne* za provjeru da li će doći do skoka na naznačenu labelu ili ne dovoljan nam je izlaz *Zero*. Međutim, za instrukciju *bgtz* (Be Greater Then Zero) to će nam biti potreban, ali ne i dovoljan podatak. Dakle, *Zero* signal će nam reći da li su dvije vrijednosti koje se porede jednake ili različite, ali ne i da li je prva vrijednost veća od nule. Za to ćemo morati iskoristiti *ALU result*. Međutim, nije nam potrebna kompletna vrijednost *ALU resulta*, već će nam biti dovoljan samo njen prvi bit (*MSB*). Zašto? Ukoliko je rezultat oduzimanja pozitivan, zasigurno će prva vrijednost biti veća, a ako to nije slučaj (odnosno ako je rezultat oduzimanja negativan), druga vrijednost će biti veća. Da sumiramo:

- Ako je *Zero*=0 → vrijednosti koje se porede su različite
- Ako je *MSB*=0 → prva vrijednost je veća od druge vrijednosti

Dakle, potrebni uslovi da bi prva vrijednost bila veća od druge vrijednosti (nule u našem slučaju) je da *Zero*=0 i da *MSB*=0. Ove uslove na neki način treba realizovati i unijeti u arhitekturu.



Imajući u vidu da je jedina realizovana branch naredba u izučavanoj arhitekturi beq oslonićemo se na nju, ali pri tome voditi računa da je izmjenama koje su nam potrebne za realizaciju *bgtz*, ne oštetimo/onemogućimo njen rad.

Na slici je prikazana promjena arhitekture (uokvireni dio se odnosni na našu instrukciju *bgtz*). Zašto smo uveli novi signal *PCWriteCond2*? (ime je potpuno proizvoljno) Signal *PCWriteCond* koristi

instrukcija *beq*, te je setovan samo kada se izvršava *beq*. Stoga, nijesmo mogli uzeti postojeći signal a da ne oštetimo realizovanu istrukciju. Novouvedeni signal *PCWriteCond2* za instrukciju *bgtz* ima istu ulogu kao *PCWriteCond* za instrukciju *beq*, a to je da je setovan kada je ispunjen ispitivani uslov.

Izmjene u dijagramu stanja će biti minimalne: umjesto *PCWriteCond* ćemo upisati naš novi signal, koji nam omogućava obavljanje nove istrukcije.