

- Umjesto objašnjavanja gradiva na "klasičan način", u ovom predmetu ćemo pokušati kroz zamišljeni projektni zadatak postupno "vesti" osnovne pojmove iz arhitekture procesora, asemblerorskog programiranja i povezivanja procesora s ulazno-izlaznim jedinicama

Arhitektura procesora FRISC

Ovaj dokument namijenjen je isključivo za osobnu upotrebu studentima Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

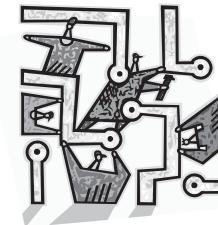
Svaka druga upotreba ili umnožavanje bilo kojeg dijela ovog dokumenta nije dozvoljena bez pismene dozvole autora.

© Mario Kovač, Danko Basch



© Kovač, Basch, FER, Zagreb

1



2

Projekt

Primarni cilj projekta:

- Računalni sustav za upravljanje radom automatskog klima uređaja u automobilu



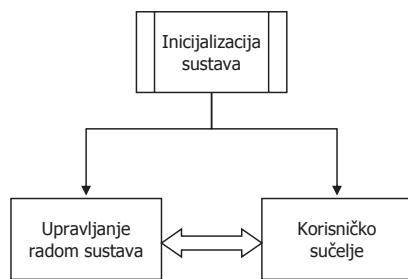
Dodatni zahtjev

- Sustav mora dozvoljavati moguća kasnija proširenja i za neke druge aplikacije (jedan sustav za više primjena-efikasnost)

© Kovač, Basch, FER, Zagreb

3

Pregled programskog dijela sustava

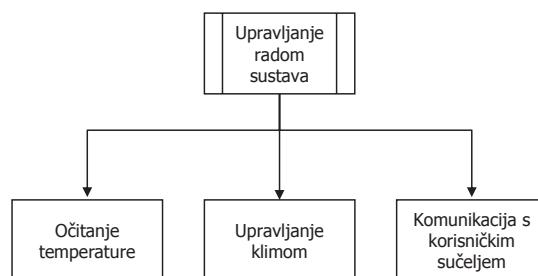


© Kovač, Basch, FER, Zagreb

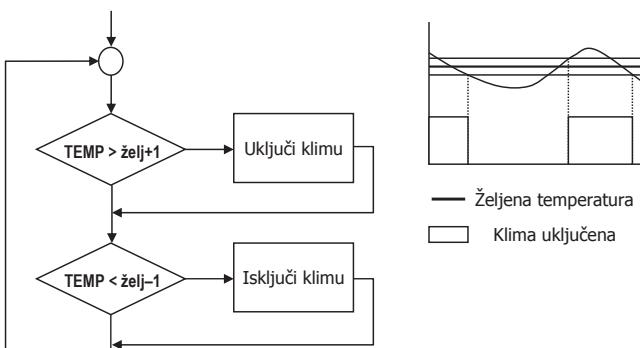
© Kovač, Basch, FER, Zagreb

4

Upravljanje radom sustava



Upravljanje klimom

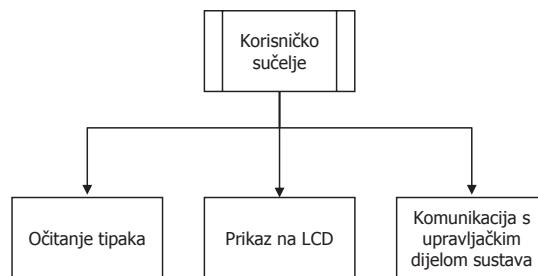


© Kovač, Basch, FER, Zagreb

© Kovač, Basch, FER, Zagreb

5

Korisničko sučelje



7

© Kovač, Basch, FER, Zagreb

8

Prvi koraci u projektu

- Na temelju zahtjeva potrebno je donijeti niz odluka:
 - Kakvo računalo odabrati ?
 - Kakav procesor odabrati ?
 - Kako povezati procesor sa sustavom kojim upravljamo ?
 - Kako napraviti program koji će upravljati sustavom ?
 - itd.
- Da bi donijeli ove odluke moramo nešto znati o vrstama računala, procesora, povezivanju sa sustavom, programiranju itd.

© Kovač, Basch, FER, Zagreb

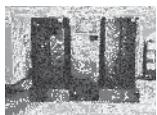
9

Vrste računala: Poslužiteljska računala



- Poslužiteljska računala (engl. serveri):

- Računala prvenstveno namijenjena izvođenju zahtjevnih programa od strane brojnih korisnika koji najčešće koriste dostupne resurse ovakvih računala preko mreže.
- Ova računala mogu biti po snazi u klasi boljih stolnih računala (najslabije inačice servera) pa sve do superračunala koja nude maksimalne performanse uz, naravno, veoma visoku cijenu.



© Kovač, Basch, FER, Zagreb

11

Vrste računala: Prisutnost na tržištu



- U zadnjem kvartalu 2010. godine proizvedeno je više smartphone-a nego osobnih računala
- Microsoft je objavio da će Windowsi 8, koji trebaju izaći (uskoro?), i aplikacije na tom OS-u podržavati platforme temeljene na procesoru ARM
- Android ima šanse postati "novi" MS Windows
- GPU (Graphics processing unit) za obradu općih podataka
- Plaća inženjera u porastu nakon nekoliko godina pada (USA)...

Izvori: Financial Times (23.1.2011, 11.2.2012), EET (24.1.2012.)

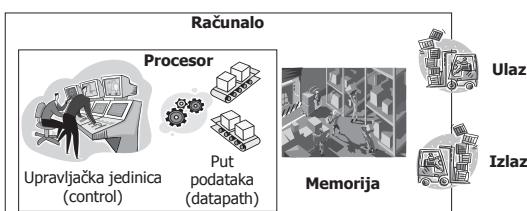
© Kovač, Basch, FER, Zagreb

13

Osnovni sastavni dijelovi svakog računala



- Svako računalo ima 5 osnovnih sastavnih dijelova: >>>
 - **Uzorak**
 - **Izlaz**
 - **Memorija**
 - **Put podataka** (datopath)
 - **Upravljačka jedinica** (control)



© Kovač, Basch, FER, Zagreb

Vrste računala: Stolno računalo



- Na temelju danih zahtjeva očito je da klima uređaj treba biti upravljan računalom
- Kakvo računalo odabrati?
- Stolna ili osobna računala:
 - Računala namijenjena su osobnoj uporabi (u ovu grupu spadaju računala poput osobnih računala (PC), radnih stanica, prijenosnih računala i sl.)



© Kovač, Basch, FER, Zagreb

10

Vrste računala: Ugradbena računala



- Ugradbena računala:

- Najbrojnija klasa računala koju karakterizira da se nalazi u nekom sustavu kojim se upravlja i obično ima unaprijed definiranu funkciju
- Nalaze se u velikoj većini proizvoda iz našeg svakodnevnog života
- Najčešće zahtjevaju detaljno projektiranje tako da se postignu potrebne performanse uz najnižu moguću cijenu ili potrošnju



© Kovač, Basch, FER, Zagreb

12

Odluka: Vrsta računala

- Izbor vrste računala je očit:
 - za naš projekt odabiremo **UGRADBENO RAČUNALO**
- Sljedeći zadatak:
 - Projektirati ugradbeni računalni sustav
- Kako bi mogli pristupiti projektiranju našeg računalnog sustava moramo znati koji su osnovni dijelovi svakog računala

© Kovač, Basch, FER, Zagreb

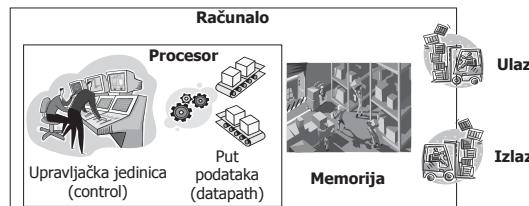
13

Osnovni dijelovi računala



- Najvažniji dio računala je procesor

- Procesor je aktivni dio računala koji upravlja radom računala i obavlja različite operacije nad podatcima u sustavu
- Procesor se sastoji od puta podataka (datapath) i upravljačke jedinice
- Procesor se naziva i CPU (od engl. Central Processing Unit)



© Kovač, Basch, FER, Zagreb

15

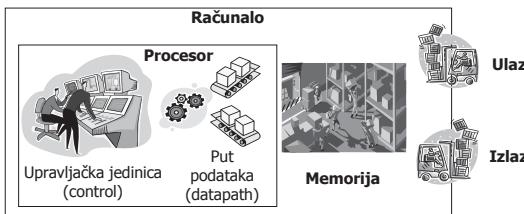
© Kovač, Basch, FER, Zagreb

16



- Put podataka

- Dio procesora koji:
 - obavlja određene operacije (npr. aritmetičke) nad podatcima
 - privremeno pamti podatke i međurezultate
 - prenosi podatke između dijelova za pamćenje i obradu



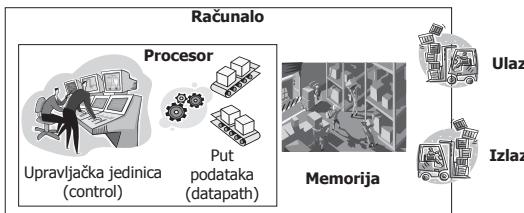
© Kovač, Basch, FER, Zagreb

17



- Memorija

- Dio računalnog sustava u kojem se spremaju programi koji se izvode na procesoru te podatci potrebeni za izvođenje tih programa



© Kovač, Basch, FER, Zagreb

19



- Osnovna funkcija procesora je izvođenje programa**

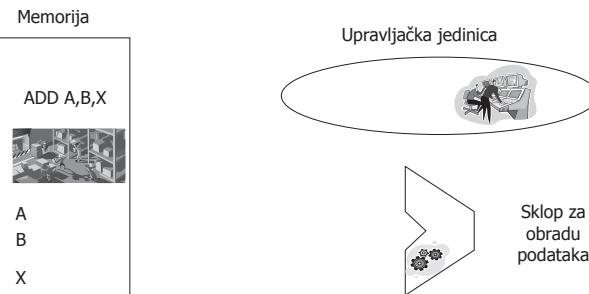
- Program se nalazi u memoriji i sastoji se od niza naredaba
 - svaka vrsta naredbi ima posebni oblik koji je razlikuje od drugih vrsta
- Procesor mora dohvaćati naredbe iz memorije, a dohvaćenu naredbu mora prepoznati da bi je mogao izvesti
- Dakle, rad procesora se odvija u tri osnovna koraka koji se stalno ponavljaju: >>>
 - dohvat naredbe (fetch)
 - dekodiranje ili prepoznavanje naredbe (decode)
 - izvođenje naredbe (execute)

© Kovač, Basch, FER, Zagreb

21



Naredba ADD A,B,X nalazi se u memoriji

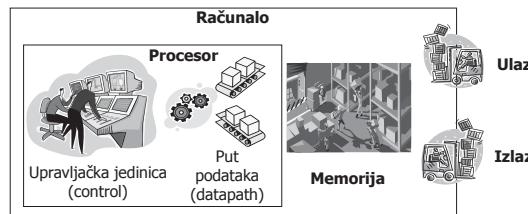


© Kovač, Basch, FER, Zagreb



- Upravljačka jedinica

- Dio procesora koji upravlja radom puta podataka, memorije i ulazno/izlaznih sklopova na temelju naredaba programa koji se izvodi



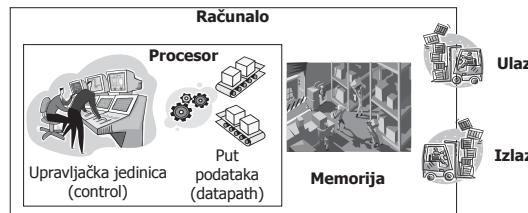
© Kovač, Basch, FER, Zagreb

18



- Ulaz i Izlaz

- Dijelovi računalnog sustava namijenjeni povezivanju s vanjskim svijetom
- Ulaz:** dio namijenjen primanju podataka iz vanjskog svijeta u računalo
- Izlaz:** dio namijenjen slanju podataka iz računala prema vanjskom svijetu



© Kovač, Basch, FER, Zagreb

20



- Dohvat** naredbe uvijek se sastoje od operacije čitanja iz memorije

- može biti više operacija čitanja, ako se naredba sastoji od više memorijskih riječi

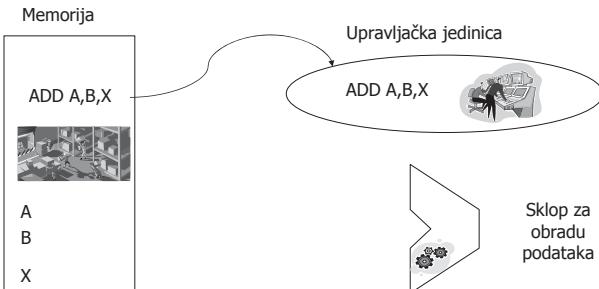
- Dekodiranje** naredbe odvija se interno unutar procesora

- Izvođenje** se odvija na različite načine, ovisno o vrsti naredbe
 - može se odvijati interno, ako procesor u sebi ima sve podatke za izvođenje
 - može uključivati operacije čitanja iz memorije, ako se podatci za izvođenje naredbe nalaze u memoriji
 - može uključivati pisanja u memoriju, ako rezultate izvođenja naredbe treba spremiti u memoriju

© Kovač, Basch, FER, Zagreb

22

Upravljačka jedinica (UJ) čita naredbu iz memorije. Naredba opisuje što treba napraviti: treba obaviti operaciju zbrajanja dva operanda A i B iz memorije i spremiti rezultat X natrag u memoriju



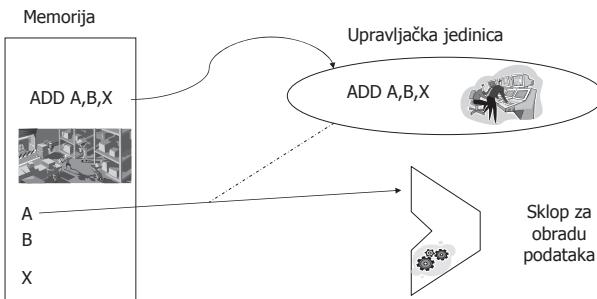
24

23

© Kovač, Basch, FER, Zagreb

Osnovni način rada procesora (npr izračunati $x=a+b$)

Upravljačka jedinica čita operand A iz memorije i dovodi ga do sklopa za obradu podataka

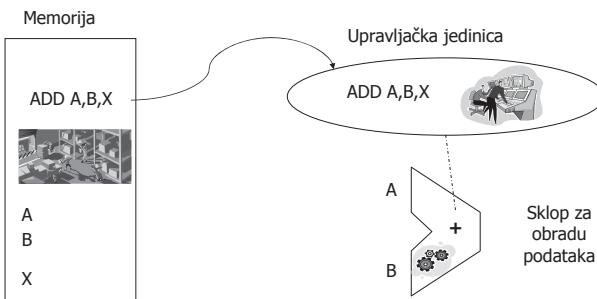


© Kovač, Basch, FER, Zagreb

25

Osnovni način rada procesora (npr izračunati $x=a+b$)

Upravljačka jedinica aktivira sklop za zbrajanje



© Kovač, Basch, FER, Zagreb

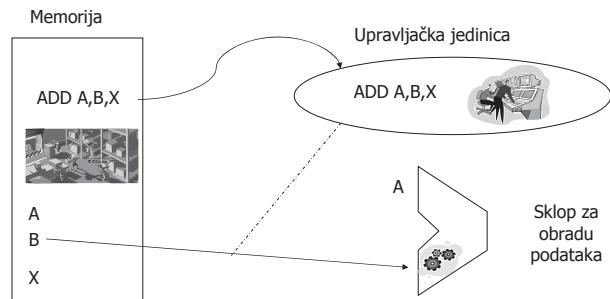
27

Razine apstrakcije

- Koliko detaljno trebamo razmatrati naše računalo? Do koje dubine ići?
- Pri razmatranju kompleksnih sustava kao što je procesor vrlo često se koriste različite razine apstrakcije
- Razine apstrakcije omogućuju pojedinim sudionicima u procesu projektiranja ili korištenja da se mogu koncentrirati na njihov dio zadatka bez potrebe da brinu o detaljima koji im nisu potrebni

Osnovni način rada procesora (npr izračunati $x=a+b$)

Upravljačka jedinica čita operand B iz memorije i dovodi ga do sklopa za obradu podataka

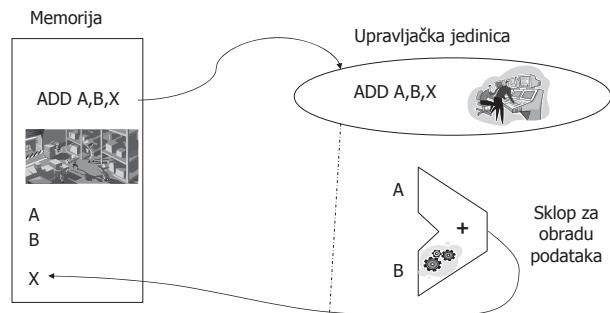


© Kovač, Basch, FER, Zagreb

26

Osnovni način rada procesora (npr izračunati $x=a+b$)

Upravljačka jedinica sprema rezultat zbrajanja natrag u memoriju



© Kovač, Basch, FER, Zagreb

28

Razine apstrakcije

Primjeri nekih razina apstrakcije kod procesora su:

- Sistemska razina
 - Visoka razina apstrakcije koju koriste programeri aplikacija koji ne moraju znati mnogo o načinu kako procesor izvodi program već se koncentriraju na funkcionalnost algoritma i cijelokupne programske podrške.
- Arhitektura skupa naredaba (Instruction set architecture level)
 - Najčešće spominjana razina apstrakcije između razine programa i sklopoljja.
 - Uključuje sve podatke o procesoru (registri, pristup memoriji, naredbe, pristup vanjskim sklopoljima i ostalo) koji su potrebni da bi se napisali programi u strojnom jeziku koji će se ispravno izvoditi.
 - Sa strane sklopoljja ova razina opisuje funkcionalnost koju sklopolje treba omogućiti.

© Kovač, Basch, FER, Zagreb

29

30

Razine apstrakcije

- Mikroarhitektura
 - Detaljan sklopoljski opis arhitekture procesora. Opisuje načine povezivanja pojedinih dijelova procesora te signale potrebne za njihovo upravljanje.
- Razina logičkih vrata (gate level)
 - Potpun sklopoljski opis procesora koji, između ostalog, često uključuje i podatke o svim vremenskim kašnjenjima signala unutar sklopa.
- Razina rasporeda (layout level)
 - Fizički opis svih sklopova procesora koji koristi definiranu tehnologiju izvedbe. Prikazuje sve detalje potrebne za preslikavanje ove razine u fizičko sklopolje u postupku proizvodnje.

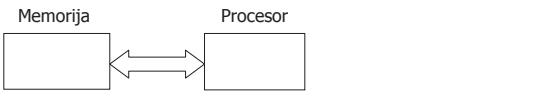
© Kovač, Basch, FER, Zagreb

Razine apstrakcije

- Razine apstrakcije koje ćemo koristiti na ovom predmetu su:
 - Arhitektura skupa naredaba (Instruction set architecture - ISA)
 - Mikroarhitektura

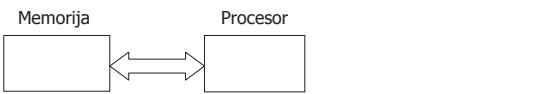
- Izvođenje zadatka na procesoru zahtijeva čitanje naredaba koje je zadao programer te čitanje i pisanje podataka koji se obrađuju (kao što smo vidjeli u jednostavnom primjeru ranije)
- Naredbe i podaci nalaze se u **memoriji** 
- Da bi se obavila jednostavna operacija zbrajanja, kao u našem prethodnom primjeru, procesor mora više puta pristupiti memoriji:
 - dohvat naredbe ADD iz memorije
 - dohvat prvog operanda A
 - dohvat drugog operanda B
 - spremanje rezultata X

Von Neumannova arhitektura



- Značajka von Neumannove arhitekture je u tome da su **program i podaci smješteni u jedinstvenu memoriju** koja ima samo jednu vezu prema procesoru 
- Jednostavna (i jeftina) arhitektura
- Nedostatak: Procesor ne može dohvatiti naredbu i podatke u istom trenutku => "**Von Neumannovo usko grlo**"
- Usko grlo predstavlja značajno ograničenje za brzinu obrade podataka kod računala koja jednostavnim operacijama obrađuju puno podataka, a trebaju biti efikasna

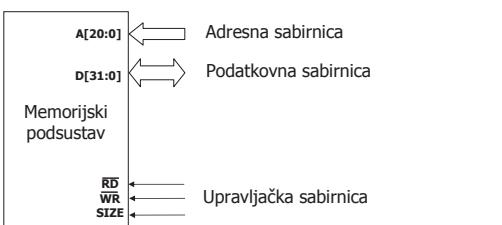
Odluka: Arhitektura mem. pristupa



- S obzirom da želimo projektirati jednostavan sustav i da nam performanse sustava nisu u popisu primarnih zahtjeva, odlučujemo se za jednostavniju i jeftiniju arhitekturu memorijskog pristupa:

Von Neumannova arhitektura

Memorijski podsustav (vrste sabirnica)



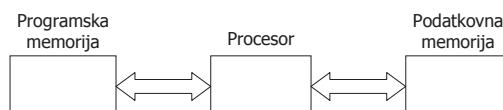
- Adresna sabirnica – određuje mem. lokaciju kojoj se pristupa 
- Podatkovna sabirnica – služi za prijenos podataka između procesora i memorije
- Upravljačka sabirnica – upravlja prijenosom podataka

- Pri definiranju našeg procesora stoe nam na raspolaganju dvije arhitekture s obzirom na način dovođenja naredaba i podataka iz memorije u procesor:

• von Neumannova arhitektura

• Harvardska arhitektura 

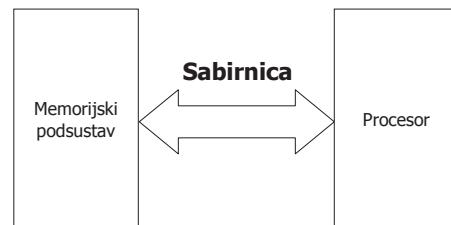
Harvardska arhitektura



- Jednostavno rješenje von Neumannovog uskog grla je **razdvajanje memorije za pohranu programa od memorije za podatke** 
- Procesor ima neovisne veze prema tim memorijama
- Ova arhitektura dozvoljava istovremeno dohvaćanje naredbe i jednog operanda čime se efikasnost znatno poboljšava*
- Skuplja od von Neumannove arhitekture te se koristi samo kad je potrebno povećanje performansi

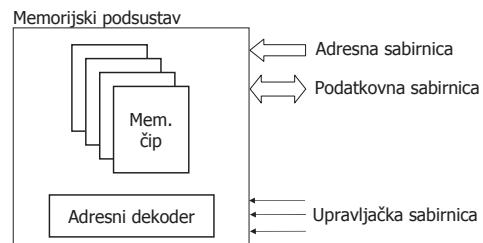
* DSP procesori (procesori za obradu signala) ponekad trebaju i više od jednog operanda pa postoje i modifikacije ove arhitekture, ali to nadilazi opseg ovih predavanja

Memorijski podsustav (osnovno)



- Memorijski podsustav načelno se spaja na procesor pomoću **sabirnicama**
- Sabirnice su spojni putovi (vodovi) koji povezuju dijelove računala

Memorijski podsustav (načelna građa)



- Adresni dekoder
 - Služi za odabir pojedinog memorijskog čipa prilikom prijenosa traženog podatka
 - Odabir se radi na temelju adresnih i upravljačkih signalova
 - Izveden kao jednostavni kombinacijski sklop

Memorijski podsustav (memorijska riječ)

- Organizacija memorijske riječi razlikuje se od procesora do procesora; jedna adresa može odgovarati memorijskoj lokaciji širine 8, 16, 32 ili više bita.

0	1	2	3
4	5	6	7
8	9	10	11
...

Najčešće su memorijske lokacije široke 8 bita (na slici)

- Kod takve organizacije mogu se čitati/pisati i podaci veće širine (16 ili 32 bita) što procesor određuje posebnim upravljačkim signalima

- Bitno je uočiti da širina podatkovne sabirnice nije nužno ista kao i širina osnovne memorijske lokacije



Mala digresija: Stog

- Struktura podataka koja radi po načelu LIFO (engl. last in first out): zadnji spremjeni (stavljeni) podatak je prvi koji se čita (uzima)
- Oprez: stog na razini arhitekture računala različit je od stoga na razini višeg programskog jezika (povezana lista i alokacija memorije) !!!**
- Stog (engl. stack) je niz memorijskih lokacija (ili kakvih drugih registara) u koje se podatci stavljuju i uzimaju po načelu LIFO
- Ova struktura se može slikovito zamisliti kao niz tanjura složenih jedan na drugi:

STAVI



UZMI

Stog

- Podatci se stavljuju na stog i uzimaju sa stoga pomoću posebnih naredaba:
 - PUSH (stavi podatak na vrh stoga) i
 - POP (uzmi podatak s vrha stoga)
- Ove naredbe koriste pokazivač stoga (SP) da bi znale odakle uzimaju (čitaju) ili gdje stavljuju (pišu) podatak
- Kod rada sa stogom uvijek moramo paziti da:
 - ne pokušamo staviti više podataka nego što ima mesta na stogu (da se stog ne "prepunii")
 - ne pokušamo uzimati podatke s praznog stoga (jer bi čitali podatke iz dijela memorije koja nije dio stoga)
 - ukratko: koliko se stavi na stog, toliko se treba uzeti

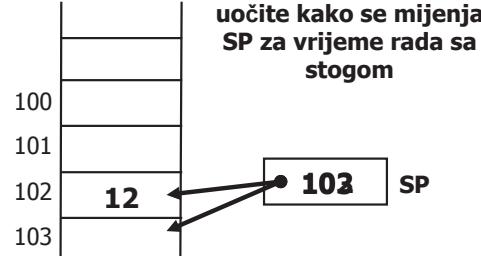


Stog

- Kako izgleda stog u memoriji računala?
- Određene memorijske lokacije koriste se kao područje za stog
- Registrar SP se obično nalazi u procesoru



STAVI



Povezivanje memorije i procesora

- Naš primjer povezivanja MEM-CPU:
 - Von Neumannova arhitektura: jedna (zajednička) memorija za naredbe i podatke
 - Veza prema CPU preko tri sabirnice: adresna, podatkovna i upravljačka
 - Adresna sabirница: služi za izbor memorijske lokacije
 - Podatkovna sabirница: prijenos podataka prema/iz memorije
 - Upravljačka sabirница:
 - određivanje smjera toka podataka, tj. operacije čitanja ili pisanja - RD i WR (read, write)
 - Određivanje širine podatka – SIZE
- Koristeći gore navedeno možemo reći da smo definirali najjednostavniji sustav za pristup memoriji

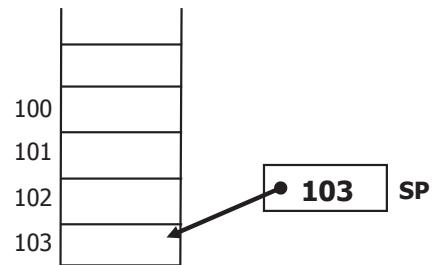
Stog

- S tanjurima je lako: uvijek znamo gdje stavljamo tanjur ili od kuda uzimamo tanjur
- Kako znati u koju memorijsku lokaciju treba stavljati ili iz koje uzimati podatak?
- Svaka lokacija se može adresirati, a adresa (polozaj) vrha stoga (ToS=Top of Stack) pamti se u pokazivaču stoga (SP=Stack Pointer)
- SP nije ništa drugo, nego obični register u kojem se pamti adresa vrha stoga

Stog

- Kako izgleda stog u memoriji računala?
- Određene memorijske lokacije koriste se kao područje za stog
- Registrar SP se obično nalazi u procesoru

>>>

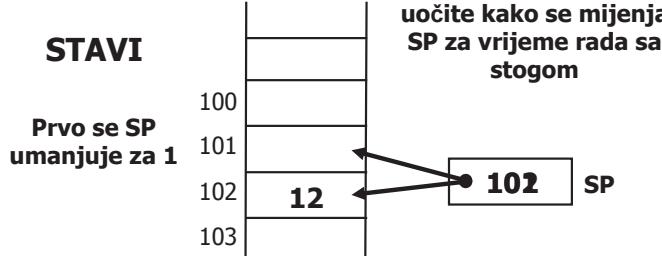


Stog

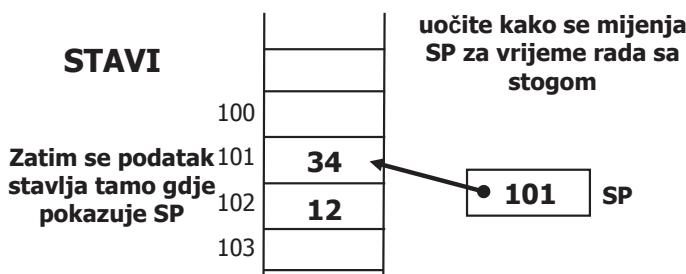
- Kako izgleda stog u memoriji računala?
- Određene memorijske lokacije koriste se kao područje za stog
- Registrar SP se obično nalazi u procesoru



STAVI



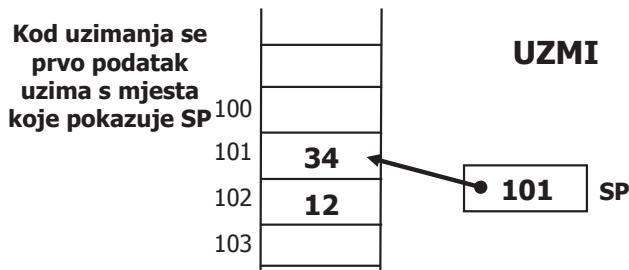
- Kako izgleda stog u memoriji računala?
- Određene memorijske lokacije koriste se kao područje za stog
- Registr SP se obično nalazi u procesoru



© Kovač, Basch, FER, Zagreb

49

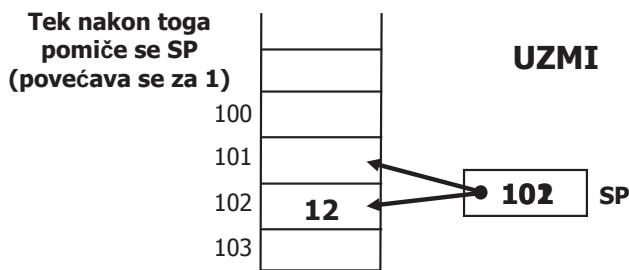
- Kako izgleda stog u memoriji računala?
- Određene memorijske lokacije koriste se kao područje za stog
- Registr SP se obično nalazi u procesoru



© Kovač, Basch, FER, Zagreb

51

- Kako izgleda stog u memoriji računala?
- Određene memorijske lokacije koriste se kao područje za stog
- Registr SP se obično nalazi u procesoru



© Kovač, Basch, FER, Zagreb

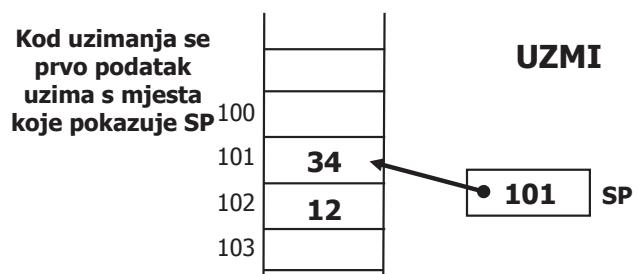
53

- Upravo smo vidjeli da se pokazivač stoga SP mijenja prilikom operacija sa stogom:
 - kod stavljanja podatka na stog → SP se smanjuje
 - kod uzimanja podatka sa stoga → SP se povećava *
- Vidjeli smo i redoslijed koraka prilikom operacija sa stogom:
 - stavljanje: umanji SP, zatim stavi podatak
 - uzimanje: uzmi podatak, zatim uvećaj SP **

* moguć je i obrnut smjer "rasta/pada" stoga, ali se on u praksi rijede koristi

** moguća je i drugačija realizacija stoga - više o tome kasnije

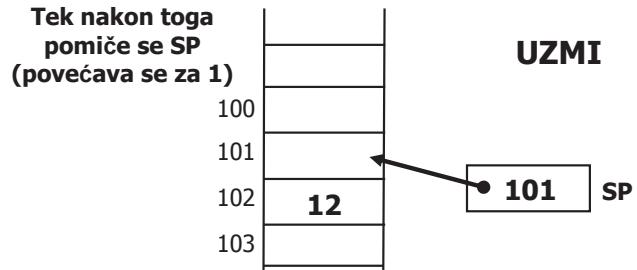
- Kako izgleda stog u memoriji računala?
- Određene memorijske lokacije koriste se kao područje za stog
- Registr SP se obično nalazi u procesoru



© Kovač, Basch, FER, Zagreb

50

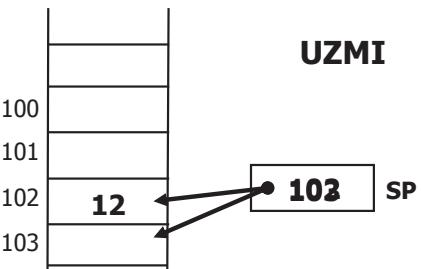
- Kako izgleda stog u memoriji računala?
- Određene memorijske lokacije koriste se kao područje za stog
- Registr SP se obično nalazi u procesoru



© Kovač, Basch, FER, Zagreb

52

- Kako izgleda stog u memoriji računala?
- Određene memorijske lokacije koriste se kao područje za stog
- Registr SP se obično nalazi u procesoru



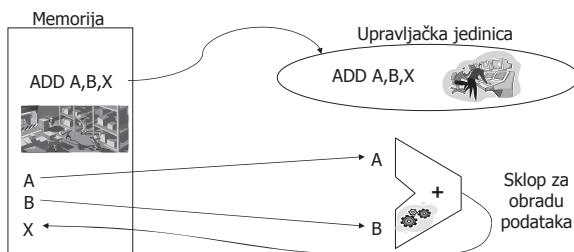
© Kovač, Basch, FER, Zagreb

54

- Kod stavljanja na stog, treba u naredbi PUSH reći što se stavlja na stog (npr. neki broj ili sadržaj nekog registra)
- Kod uzimanja sa stoga, treba u naredbi POP reći gdje se upisuje pročitani podatak (npr. u neki registar)
- Važno je još napomenuti:
 - Pri uzimanju, tj. čitanju podatka, taj se podatak ne "uzima" doslovno sa stoga, tj. nakon čitanja se podatak ne briše sa stoga.
 - U stvarnosti, podatak koji je "uzet" ostaje zapisan na stogu, a samo je registr SP promijenio svoju vrijednost tako da pokazuje na prethodni podatak!!!

Problem dovođenja podataka u procesor

- Već prije smo vidjeli kako bi se načelno izvodila naredba za zbrajanje: $X = A + B$

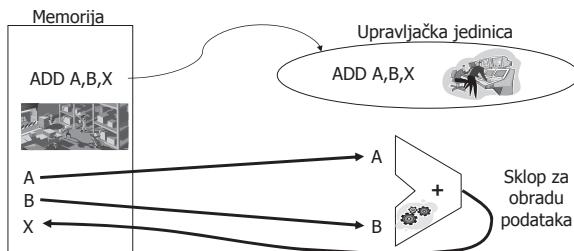


© Kovač, Basch, FER, Zagreb

57

Problem dovođenja podataka u procesor

- Također smo naučili da su procesor i memorija povezani sabirnicom i da ne postoje zasebni spojni putovi za sva čitanja i pisanja (kao što to izgleda na ovoj slici)



© Kovač, Basch, FER, Zagreb

59

Arhitekture s obzirom na smještaj operanada

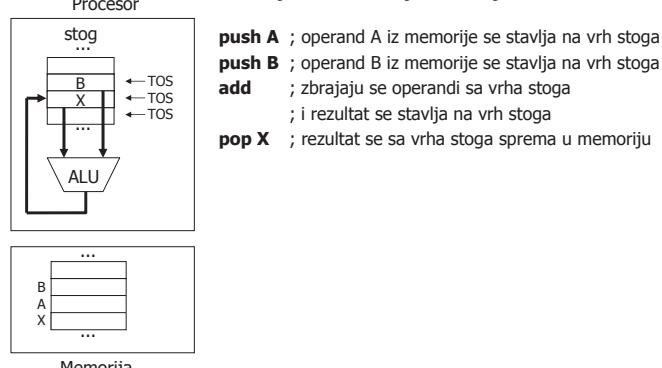
- Obzirom na smještaj operanada, arhitekture se obično dijele na: >>>
 - Stogovne arhitekture
 - Akumulatorske arhitekture
 - Arhitekture registar-memorija
 - Arhitekture registar-registar (tzv. load-store)

© Kovač, Basch, FER, Zagreb

61

Stogovna arhitektura

- Primjer: računanje funkcije $x=a+b$

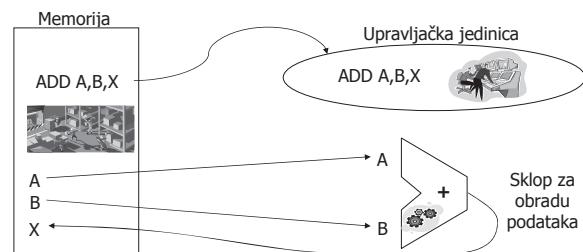


© Kovač, Basch, FER, Zagreb

Problem dovođenja podataka u procesor

- Za fazu izvođenja je bilo potrebno:

- čitanje operanada iz memorije
- spremanje rezultata u memoriju



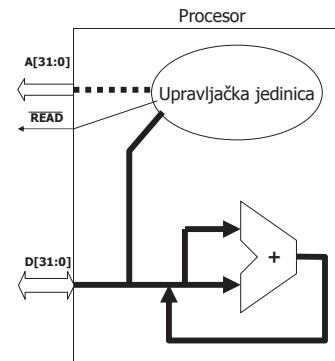
© Kovač, Basch, FER, Zagreb

58

Problem dovođenja podataka u procesor

Problem:

- Da bi obavili zbrajanje, moramo iz memorije preko sabirnice podataka do zbrajala dovesti dva podatka (operanda)
- To očito nije moguće napraviti u istom trenutku preko jedne sabirnice
- Postoji nekoliko arhitektura koje nude rješenje ovog problema....

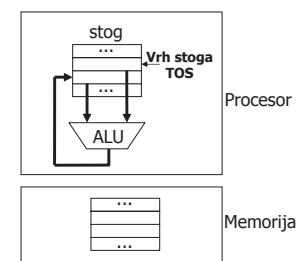


© Kovač, Basch, FER, Zagreb

60

Stogovna arhitektura

- Upotrebljava se stog za spremanje podataka koji se obrađuju (da bi se riješio ranije spomenuti problem dohvata podataka iz memorije)
- Ovaj stog **nalazi se u procesoru**, kao i pokazivač stoga!!!
- **Operandi su implicitno na vrhu stoga, gdje se smješta i rezultat**
- U naredbama za obradu podataka ne zadaje se eksplicitno gdje se nalaze operandi niti gdje se spremi rezultat
- Naredba za obradu podataka pristupa samo internom stogu



© Kovač, Basch, FER, Zagreb

62

Stogovna arhitektura

- Stogovna arhitektura koristila se kod prvih procesora i danas se u svom osnovnom obliku više ne koristi
- Dobre strane stogovne arhitekture:
 - Naredbe su jednostavnije i bez puno opcija što ih čini brzima za izvođenje
 - Izvedba upravljačke jedinice je jednostavna
 - Prevoditelji su jednostavniji
- Loše strane stogovne arhitekture:
 - Međurezultati se teško koriste
 - Prevodenje nije efikasno (jednostavna naredba višeg jezika se prevodi u dugačak niz naredaba strojnog jezika)
 - Veliki broj pristupa memoriji !!!

© Kovač, Basch, FER, Zagreb

63

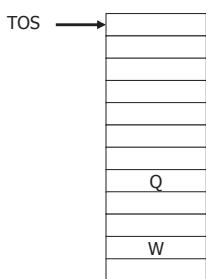
© Kovač, Basch, FER, Zagreb

64



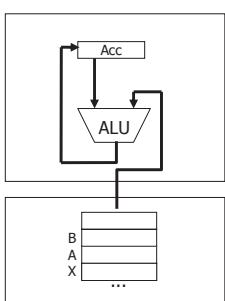
- Primjer nedostatka ove arhitekture

- Pretpostavimo da trebamo zbrojiti međurezultate Q i W sa stoga.....
- Za tako jednostavnu funkciju treba dosta operacija s memorijom



Akumulatorska arhitektura

Primjer: računanje izraza $x=a+b$

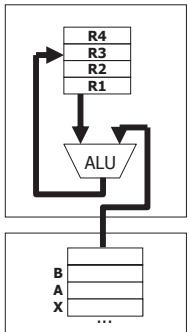


load A ; operand A se iz memorije
; stavlja u Acc

add B ; zbraja se Acc sa operandom
; B iz memorije i rezultat
; se stavlja u Acc

store X ; rezultat se iz Acc
; sprema u memoriju

Arhitektura registar-memorija



Jedan operand se nalazi u skupu registara opće namjene (registri koji se slobodno koriste i nemaju posebnu funkciju)

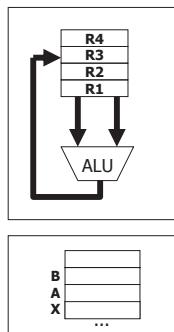
- Kao i kod drugih arhitektura, uvjek može postojati jedan ili više registara specifične namjene, ali se oni ne ubrajaju u općenamjenske registre

Drugi operand se čita iz memorije

Rezultat se sprema u neki od registara opće namjene

- Naredba za obradu podataka pristupa memoriji

Arhitektura registar-registar (load-store)



Oba operanda su u registrima opće namjene

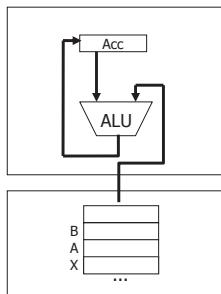
Rezultat se sprema u neki od registara opće namjene

- Naredba za obradu podataka pristupa isključivo općim registrima

- Podatci se mogu čitati iz memorije ili pisati u nju isključivo pomoću naredba LOAD i STORE



Jedan operand je uvijek u posebnom registru koji se naziva Akumulator (Acc)



Ako postoji, drugi operand se čita iz memorije

Rezultat se sprema u Acc

Naredba za obradu podataka pristupa memoriji

Akumulatorska arhitektura

Vrlo česta arhitektura prvih procesora, a danas se još može naći kod nekih jednostavnih mikrokontrolera

Dobre strane ove arhitekture:

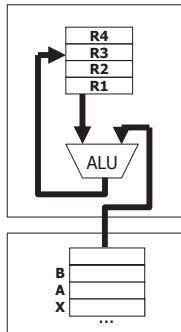
- Jednostavnija za izvedbu od stogovne (Acc umjesto stoga)
- Naredbe su jednostavne i bez puno opcija
- Jedan operand je u memoriji (ne mora ga se dohvaćati dodatnom naredbom)
- Prevoditelji su jednostavniji

Loše strane ove arhitekture:

- Međurezultati (osim zadnjeg) se ne mogu koristiti već sve mora biti pohranjeno u memoriju
- Jako velik broj pristupa memoriji !!!
- Prevođenje nije efikasno

Arhitektura registar-memorija

Primjer: računanje izraza $x=a+b$

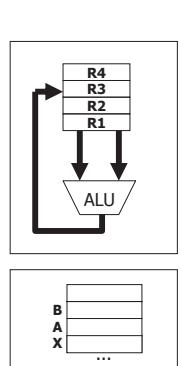


load R1,A ; operand A se iz memorije
; stavlja u R1

add R3,R1,B ; zbraja se R1 sa operandom B iz
; memorije i rezultat se stavlja u R3

store R3,X ; rezultat se iz R3 sprema u
; memoriju

Arhitektura registar-registar (load-store)



Primjer: računanje izraza $x=a+b$

load R1,A ; operand A se iz memorije
; stavlja u R1

load R2,B ; operand B se iz memorije
; stavlja u R2

add R3,R1,R2 ; zbraja se R1 sa R2
; i rezultat se stavlja u R3

store R3,X ; rezultat se iz R3 sprema u
; memoriju



- Na temelju prethodnih objašnjenja vidljive su neke prednosti i nedostaci pojedinih arhitektura:
- Stogovna i akumulatorska arhitektura bile su često korištene u prvim procesorima dok se danas gotovo ne koriste
- Neke ideje revitalizacije stogovne arhitekture postoje kod procesora koji izvode Java bytecode
 - npr. SUN picoJavaII: kombinacija dobrih osobina stogovne arhitekture (cirkularni stog) i registarskih arhitektura (operacije s podatcima koji su bilo gdje na stogu)
- Osim iznimaka, većina današnjih procesora ima registarsku arhitekturu (reg-mem ili reg-reg) među kojima su više zastupljene reg-reg (load-store) arhitekture

© Kovač, Basch, FER, Zagreb

73

Primjeri navedenih arhitektura



- Stogovna:
 - WISC CPU/16, MISC M17, picoJava
- Akumulatorska
 - EDSAC
- Registarsko-memorijska
 - Motorola 68000 i Intel 80x86 (imaju karakteristike reg-mem i reg-reg)
- Registarsko-registerska (load-store)
 - ARM, MIPS, SPARC

© Kovač, Basch, FER, Zagreb

75

Arhitekture s obzirom na skup naredaba

- S obzirom na skup naredaba procesora razvijene su dvije arhitekture:
 - CISC (Complex Instruction Set Computer)
 - RISC (Reduced Instruction Set Computer) >>>
- U današnje vrijeme komercijalni procesori nemaju više čistu arhitekturu CISC ili RISC:
 - obično u određenom procesoru prevladava jedna arhitektura, ali...
 - često se uključuju pojedina svojstva druge arhitekture
- Pogledajmo karakteristike CISC i RISC arhitektura...

© Kovač, Basch, FER, Zagreb

77

CISC



- Karakteristike CISC procesora
 - Velik broj naredaba i njihovih inačica
 - Velik broj načina adresiranja (više o tome kasnije)
 - Većinom su se naredbe unutar procesora izvodile korištenjem načela mikroprograma, tj. kompleksne naredbe izvodile su se u nizu ciklusa tijekom kojih je procesor izvodio niz jednostavnijih operacija (više o tome ćemo govoriti kasnije)
 - Registri imaju posebne namjene (brojači za petlje, za adresiranje, za podatke itd.)
 - Problem kompleksnih naredaba rješava se "unutar procesora" (možemo reći "sklopovski")
 - Skupo projektiranje i visoka cijena
- Primjeri CISC procesora: Intel 80x86, Motorola 68000

© Kovač, Basch, FER, Zagreb

- Karakteristike registrarskih arhitektura su:

- Brži pristup operandima
- Variable se mogu čuvati u registrima opće namjene (što je više registara to je manje potrebno komunicirati s memorijom)
- Prevoditelji efikasnije prevode programe korištenjem registara
- Naredbe veće i sporije

Reg-mem arhitekture

- Jednostavan pristup podatcima u memoriji
- Dodatan pristup memoriji pri izvođenju naredaba, vrijeme izvođenja varira

Load-store arhitekture

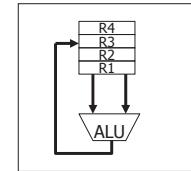
- Brzo izvođenje, jednostavan format naredaba, jednostavno generiranje koda, jednostavnost protične strukture, uniformno vrijeme izvođenja
- Veći broj naredaba u programu zbog zasebnih učitavanja podataka iz mem.

© Kovač, Basch, FER, Zagreb

74

Odluka: Arhitektura smještaja operanada

- Od nabrojenih arhitektura trenutno je najefikasnija i u svijetu dominantna tzv. LOAD-STORE arhitektura (u području ugradbenih računala)
- Dodatna pogodnost je da se ta arhitektura može i najjednostavnije projektirati te sklopovski izvesti
- Iz tog razloga za naš procesor izabiremo LOAD-STORE arhitekturu



© Kovač, Basch, FER, Zagreb

76

CISC

- U samom početku procesori su bili vrlo jednostavni, ali su tehnološki vrlo brzo napredovali...
- Uskoro je glavni trend u oblikovanju arhitekture procesora bilo uvođenje procesorskih naredaba bliskih naredbama viših programskih jezika, npr.:
 - učitaj operande, zbroji ih i spremi rezultat
 - umanji registar za 1 i skoči na početak petlje ako je registar veći od nule
 - pomoži matrice u memoriji
 - pronađi određeni podatak u bloku memorije
- Prednosti takvih procesorskih naredaba bile su:
 - Jednostavnije prevodenje programa iz viših programskih jezika
 - Ušteda memorije zbog manjeg broja naredaba (važno u to doba!!!)
 - Ubrzanje rada zbog manjeg broja dohvata naredaba iz memorije
- Procesori s takvim skupom naredaba nazivaju se CISC procesori

© Kovač, Basch, FER, Zagreb

78

RISC

- 70ih i početkom 80ih dominaciju na tržištu imali su 8-bitni CISC procesori
- Međutim, kompleksne naredbe zahtjevaju kompleksnu logiku za dekodiranje (sporo dekodiranje i izvođenje) i izuzetno skup i dugotrajan postupak projektiranja takvih procesora
- Početkom 80-tih, u okviru tri gotovo usporedna istraživačka projekta (IBM 801, Berkeley RISC i Stanford MIPS) razvijena je potpuno nova arhitektura procesora zasnovana na jednostavnim instrukcijama koje se mogu izvoditi velikom brzinom
- Procesori s takvim skupom naredaba nazivaju se RISC (Reduced Instruction Set Computer)
- RISC procesor razvijen na sveučilištu Berkeley imao je izuzetne performanse u usporedbi s komercijalnim CISC-procesorima uz znatno jednostavniju i jeftiniju sklopovsku izvedbu.

79

© Kovač, Basch, FER, Zagreb

80

- Primjeri jednostavnih "RISC-naredaba"
 - učitaj operand iz memorije u registar
 - zbroji dva podatka iz registra
 - spremi sadržaj registra u memoriju
 - umanji sadržaj registra za 1
 - skoči na neku naredbu (npr.) početak petlje ako je zastavica ZERO=1
- Umjesto jedne kompleksne "CISC-naredbe" može se napisati niz jednostavnijih "RISC-naredaba"
 - jednostavnije naredbe se puno brže izvode pa je rezultat ubrzanje izvođenja programa bez obzira na veći broj naredaba

Izbor RISC-CISC

- Zbog očitih prednosti na cjelokupnom tržištu ugradbenih uređaja koji u sebi sadrže procesor, RISC procesori danas dominiraju
- Mi ćemo zbog toga, a i zbog jednostavnosti arhitekture koju ćemo opisivati u okviru ovih predavanja, također odabrat da naš procesor bude tipa RISC
- **Prema ovoj vrsti arhitekture dat ćemo i ime našem procesoru. Procesor ćemo zvati FRISC što je kratica od FER RISC**



Odluka: Broj registara

- S obzirom da smo izabrali load-store arhitekturu moramo definirati koliko registara želimo u našem procesoru
- S obzirom da će se izbor pojedinog registra obavljati postavljanjem određenih bitova unutar naredbe onda je efikasno da broj registara bude potencija broja 2
- Većina današnjih procesora ima 8 ili 16 registara opće namjene



Odluka: Širina registara (širina sab. pod)

- Širinu registara (u bitovima) određuje statistika najčešće korištenih podataka u programima koje želimo izvoditi na procesoru. Na primjer:
 - Byte 8b
 - Short 16b
 - Int 32b
 - Long 64b
 - Float 32b
 - Double 64b
- Daleko najčešće korišteni tipovi podataka u općim primjenama su 32-bitni int i float
- Na temelju toga možemo izabrati **32b kao optimalnu širinu registara**
- S obzirom da smo za registre opće namjene izabrali širinu od 32b, tada smo implicitno definirali i širinu interne sabirnice podataka kao i širinu ulaza i izlaza aritmetičko-logičke jedinice



- Karakteristike RISC procesora
 - Relativno malen skup jednostavnih naredaba, manji broj inačica svake naredbe
 - Mali broj načina adresiranja
 - Pojedina naredba brzo se izvodi
 - Velik broj ravnopravnih registara opće namjene unutar procesora
 - Problem kompleksnih naredaba rješava se izvan procesora (možemo reći "programski")
 - Korištenje protočne strukture za ubrzanje rada (više o tome kasnije)
 - Relativno jeftino projektiranje i niska cijena
- Primjeri RISC procesora: MIPS, ARM, SPARC

Dosadašnje odluke

- Koristiti ćemo ugradbeno računalo
- Razmatrati ćemo problematiku na razini arhitekture skupa naredaba i mikroarhitekture
- Pristup memoriji će biti zasnovan na von Neumannovoj arhitekturi
- Arhitektura skupa naredaba bit će load-store arhitektura
- RISC procesor



Odluka: Broj registara

- Radi jednostavnosti arhitekture i što jednostavnijeg oblika naredbe izabiremo da naš procesor ima 8 registara opće namjene
- Nazovimo registre: R0, R1, R2, R3, R4, R5, R6 i R7



Odluka: Širina sabirnica

- S obzirom da smo izabrali 32b širinu registara opće namjene, tada je normalno da je i širina **sabirnice podataka 32b ***
- Širinu memoriske riječi odabiremo da bude jedan bajt (8b), ali zbog veće efikasnosti ćemo moći pročitati odjednom 4 bajta (32b) što nam dozvoljava širina podatkovne sabirnice

* Postoje procesori koji imaju različitu širinu unutarnjih i vanjskih sabirnica, no u to nećemo ulaziti u okvir ovog predmeta (razlog može biti npr. želja za što manjim brojem priključaka-pinova na procesoru).

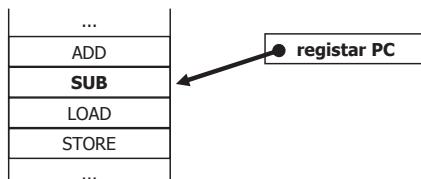


- Širina adresne sabirnice određena je maksimalnim željenim prostorom memorije
- Iako će naši programi biti vrlo kratki, radi jednostavnosti i uniformnosti arhitekture definirati ćemo da je i **adresna sabirnica 32b**
- Svaka adresa odgovara jednoj memorijskoj riječi, tj. jednom bajtu**

Programsko brojilo



- To znači da procesor u svakom trenutku mora "znati" adresu naredbe koju treba dohvatiti i izvesti
 - Kako procesor to "zna"?
 - Jednostavno: procesor ima jedan registar koji služi samo toj svrsi: PC (program counter) ili programsko brojilo (iako se tu zapravo ništa ne prebraja)



Programsko brojilo



- Naš registar PC bit će širok 32 bita, jer smo za adresnu sabirnicu odabrali istu širinu, a logično je da se registrom PC može adresirati cijeli memorijski prostor
- Registar PC se automatski uvećava (nakon dohvata svake naredbe) tako da pokazuje na sljedeću naredbu u memoriji
 - za tu svrhu, PC ima posebne sklopove za uvećavanje

Odabir skupa naredaba

Odabir skupa naredaba



- Jedna od najvažnijih odluka u projektiranju procesora je odabir skupa naredaba (instruction set)
- Postoji više vrsta naredaba, ovisno o procesoru, na primjer:
 - Aritmetičko-logičke naredbe obavljaju AL operacije
 - Registerske naredbe premještaju podatke između registara
 - Memorijske naredbe čitaju i spremaju podatke u/iz memorije
 - Naredbe za premještanje podataka mogu premještati podatak između registara, memorijskih lokacija i/ili puniti brojeve u registre i memorijske lokacije
 - Upravljačke naredbe omogućuju programske skokove

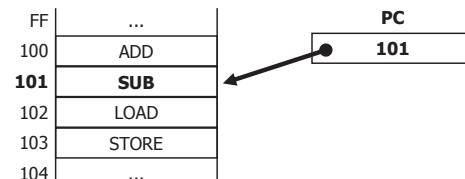
>>>

Programsko brojilo



VAŽNO:

- Programsko brojilo je registar u kojem procesor pamti adresu sljedeće naredbe koju treba izvesti**



Za sada zanemarimo širinu naredaba i točne adrese

Odluke: Rekapitulacija građe



- Osam 32-bitnih registara R0, R1, R2, R3, R4, R5, R6, R7
- 32-bitno programsko brojilo - registar PC
- Širine internih sabirnica i ALU su 32 bita
- Širina podatkovne sabirnice je 32 bita
- Širina adresne sabirnice je 32 bita
- Širina memorijске lokacije je 8 bita, ali se može odjednom čitati/pisati 32-bitni podatak

Odabir skupa naredaba



- Jedna od najvažnijih odluka u projektiranju procesora je odabir skupa naredaba (instruction set)
- Postoji više vrsta naredaba, ovisno o procesoru, na primjer:
 - Aritmetičko-logičke naredbe obavljaju AL operacije
 - Registerske naredbe premještaju podatke između registara
 - Memorijske naredbe čitaju i spremaju podatke u/iz memorije
 - Naredbe za premještanje podataka mogu premještati podatak između registara, memorijskih lokacija i/ili puniti brojeve u registre i memorijske lokacije
 - Upravljačke naredbe omogućuju programske skokove



<<< (nastavak)

- Ulazno-izlazne naredbe služe za rad s ulazno-izlaznim jedinicama
- Naredbe za rad s bitovima omogućuju ispitivanje i mijenjanje pojedinih bitova u podatku
- Naredbe za rad s blokovima podataka omogućuju pretraživanje, čitanje i pisanje većeg broja podataka (tj. bloka) koji se nalazi u memoriji
- Specijalne naredbe s posebnom namjenom (npr. odabir načina prekidnog rada, dozvoljavanje ili zabranjivanje prekida, programsko izazivanje iznimaka, naredbe za atomarno ispitivanje i postavljanje memorijskih lokacija, rad s koprocesorima itd.)
- itd. ...

Uglavnom: postoji puno vrsta naredaba

Aritmetičko-logičke naredbe

- Naredbe koje postoje u svim procesorima su aritmetičko-logičke naredbe pa će ih i naš procesor imati
- U prethodnim razmatranjima vidjeli smo da ALU može izvoditi zbrajanje. To naravno nije dovoljno za izvođenje bilo kakvog ozbiljnijeg programa pa moramo vidjeti koje bi nam još operacije trebale
- Tipične operacije koje su često potrebne su:
 - zbrajanje - ADD
 - oduzimanje - SUB
 - logički I na bitovima - AND
 - logički ILI na bitovima - OR
 - logički ESKLUZIVNI ILI na bitovima - XOR

Aritmetičko-logičke naredbe - primjer

Primjer korištenja AL naredaba:

S podatcima iz registara izračunati sljedeće: $R0 := (R1+R2) - (R3+R4)$

Rješenje:

```
ADD R1, R2, R5 ; prvi dio izraza
ADD R3, R4, R6 ; drugi dio izraza
SUB R5, R6, R0 ; razliku spremi u R0
```

Rješenje bez promjene registara R5 i R6:

```
ADD R1, R2, R0 ; R1+R2
SUB R0, R3, R0 ; R1+R2-R3
SUB R0, R4, R0 ; R1+R2-R3-R4
```

Memorijske naredbe

- AL-naredbe mogu raditi samo s podatcima u registrima
 - Što ako bi u prethodnom primjeru bilo zadano da se podatci nalaze u memoriji? Kako ih dohvatiti? Što ako bi rezultat trebalo spremiti u memoriju?
- S obzirom da je broj registara ograničen, u praksi se (skoro) svi podaci čuvaju u memoriji. Zato trebamo naredbe pomoću kojih bi mogli pročitati podatak iz memorije ili upisati podatak u memoriju
- To će obavljati memorijske naredbe, a kako smo izabrali load-store arhitekturu, onda su to jedine naredbe koje omogućuju razmjenu podataka između memorije i registara



Odabir skupa naredaba

Aritmetičko-logičke naredbe

Aritmetičko-logičke naredbe

- Za svaku od ovih operacija imat ćemo jednu naredbu, a za svaku naredbu moramo zadati operande
 - Budući da smo odabrali arhitekturu load-store (tj. registarsko-registarsku arhitekturu), svi operandi i rezultat nalazit će se u općim registrima
- Opći oblik naredaba izgledat će ovako:
naredba operand_1, operand_2, operand_3
- "izvedi operaciju između prva dva *operandi* i stavi rezultat u treći *operand*"
- Sada možemo napisati prvi program u kojem izračunavamo aritmetički izraz...

Odabir skupa naredaba

Memorijske naredbe

Memorijske naredbe

- Potrebne su nam dvije glavne operacije:
 - LOAD - čitanje 32-bitnog podatka iz memorije u registar
 - STORE - pisanje 32-bitnog podatka iz registra u memoriju
- Definirajmo operande za svaku naredbu:
LOAD register, (adresa)
STORE register, (adresa)
- Naredba LOAD "puni registar", tj. čita sadržaj četiriju memorijskih lokacija počevši od zadane *adrese* i stavlja ih u *register*
- Naredba STORE "sprema registar", tj. čita sadržaj zadanog *registra* i upisuje ga u četiri memorijске lokacije počevši od zadane *adrese*



Memorijske naredbe

- Adresu zadajemo kao običan broj (pozitivni cijeli broj), što znači da moramo poznavati položaj memorijske lokacije kojoj želimo pristupati
- Iako naredbe LOAD i STORE pristupaju četirima memorijskim lokacijama (jednobajtnim) odjednom, nećemo to posebno naglašavati
 - Podrazumijeva se da se zadana adresa odnosi na prvu lokaciju od potrebne četiri

© Kovač, Basch, FER, Zagreb

105

Memorijske naredbe - primjeri

Primjer: izračunavanje izraza s čuvanjem stanja registra

S podatcima iz registara izračunati: $R0 := (R1+R2) \text{ or } (R3-R4)$. Osim registra R0, ne smije se promjeniti sadržaj ostalih registara.

Rješenje:

```
STORE R5, (1000) ;;; pohrani R5  
  
ADD R1, R2, R5 ; prvi dio izraza (mijenja R5)  
SUB R3, R4, R0 ; drugi dio izraza  
  
OR R5, R0, R0 ; OR spremi u R0  
  
LOAD R5, (1000) ;;; obnovi R5
```

© Kovač, Basch, FER, Zagreb

107

Memorijske naredbe - primjeri

- U prethodnom primjeru vidjeli smo kako se određeni podatak može upisati u memoriju:

```
BR_55 DW 55  
BR_4ABC DW 4ABC
```

- Kasnije ćemo detaljnije objasniti način pisanja i značenje ovih redaka, a sada možemo dati intuitivno objašnjenje:
 - BR_55 i BR_4ABC su labele ili nazivi memorijskih lokacija, koje možemo pisati umjesto stvarnih adresa tih memorijskih lokacija
 - Pomoću DW (*define word*) definiramo da se u memoriju upisuje određeni broj, u ovom slučaju to su brojevi 55 i 4ABC
 - Ove dvije memorijske lokacije možemo neformalno promatrati kao dvije globalne varijable s imenima BR_55 i BR_4ABC kojima smo dodijelili početne vrijednosti 55 i 4ABC

© Kovač, Basch, FER, Zagreb

109

Odabir skupa naredaba

Upravljačke naredbe

Memorijske naredbe - primjeri

Primjer: izračunavanje izraza čiji operandi su u memoriji

Zbrojiti podatke iz memorijskih lokacija s adresama 1000 i 1004, a rezultat staviti na adresu 1008.

Rješenje:

```
LOAD R0, (1000) ; dohvati 1. broj iz memorije  
LOAD R1, (1004) ; dohvati 2. broj iz memorije  
  
ADD R0, R1, R2 ; zbroji R0+R1 i spremi u R2  
  
STORE R2, (1008) ; spremi rezultat u memoriju
```

© Kovač, Basch, FER, Zagreb

106

Memorijske naredbe - primjeri

Primjer: izračunavanje izraza s konstantnim operandima

Izračunati sljedeće: $R0 := (R1+55) - (R2 \text{ xor } 4ABC)$.

Rješenje:

```
LOAD R0, (BR_55) ; učitaj broj 55  
ADD R1, R0, R0 ; prvi dio izraza  
  
LOAD R3, (BR_4ABC) ; učitaj broj 4ABC  
XOR R2, R3, R3 ; drugi dio izraza  
SUB R0, R3, R0 ; razliku spremi u R0  
  
; Brojevi 55 i 4ABC moraju biti spremljeni  
; u memoriji na adresama BR_55 i BR_4ABC:  
BR_55 DW 55  
BR_4ABC DW 4ABC (DW >>>)
```

© Kovač, Basch, FER, Zagreb

108

Memorijske naredbe - važna napomena

- U primjerima smo čitali ili pisali podatak s određene memorijske lokacije (npr. s adrese 1000)
 - Na toj lokaciji se mora nalaziti potrebeni podatak u slučaju čitanja
 - Na toj lokaciji se mora nalaziti "slobodno" mjesto u slučaju pisanja
 - Na toj lokaciji ne smije se nalaziti neka druga naredba programa

© Kovač, Basch, FER, Zagreb

110

Upravljačke naredbe



- Za sada imamo:

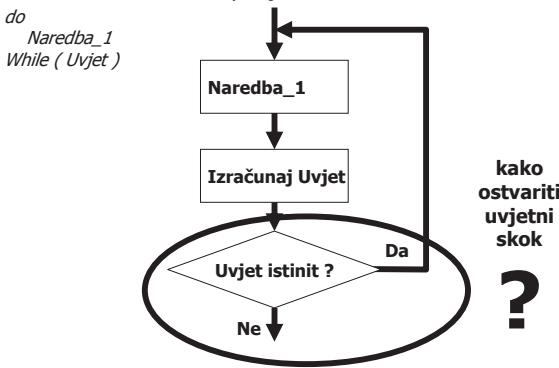
- Aritmetičko-logičke naredbe (ADD, SUB, AND, OR, XOR)
- Memorijske naredbe (LOAD, STORE)

- Što **možemo** napraviti s ovim naredbama?

- Ne puno, ali možemo ostvariti jednostavna izračunavanja pri čemu su podaci i rezultati u memoriji ili u registrima. Na primjer:
 - Izračunavanje aritmetičkih izraza: $a+3-4+b+(c-12)+d$
 - Izračunavanje operacija s bitovima:
(a OR 00001111) XOR (b AND 00111100)
 - Izračunavanje logičkih izraza: a AND b XOR true
 - Sve kombinacije gore navedenih izraza gdje se naredbe izvode **slijedno** jedna iza druge



- **Ne možemo** ostvariti petlju do-while:



© Kovač, Basch, FER, Zagreb

113



- Problem je što se AL-naredbe i memoriske naredbe izvode **isključivo slijedno**, tj. jedna iza druge - onim redoslijedom kojim su napisane
- **Zaključak:** nedostaje nam mogućnost mijenjanja redoslijeda normalnog slijednog izvođenja, tj. treba nam **naredba skoka**
- Naredbe skokova svrstavaju se u upravljačke (kontrolne) naredbe jer one upravljaju tijekom izvođenja programa.

© Kovač, Basch, FER, Zagreb

115



- Nazovimo naredbu **bezuvjetnog skoka** JP (od JUMP)
- Definirajmo način pisanja i operand naredbe JP:

JP adresa

- Naredba JP bezuvjetno skače na naredbu sa zadanom *adresom*
- *Adresa* je zadana običnim brojem kao kod memoriskih naredaba (vidjet ćemo kasnije da se adresa također odnosi na četiri memoriske lokacije, što nećemo posebno naglašavati)
- Uočite da je moguć skok unaprijed ili unazad
- Kao i kod memoriskih naredaba, za *adresu* je umjesto broja moguće pisati labelu

© Kovač, Basch, FER, Zagreb

117



- Promotrimo li malo bolje, vidimo da je naredba bezuvjetnog skoka samo **specijalni slučaj** uvjetnog skoka pri čemu je uvjet uvijek istinit
- Zato će postojati **samo jedna naredba za skok JP** koju pišemo na dva načina:

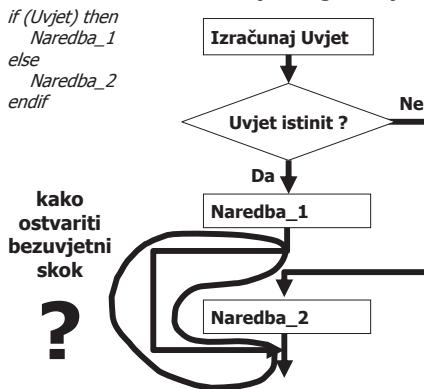

```
JP 100 // bezuvjetno skoči na naredbu na adresi 100
JP_uvjet 100 // skoči na 100 ako je uvjet istinit
```
- Kako se piše **uvjet**? >>>

© Kovač, Basch, FER, Zagreb

119



- **Ne možemo** ostvariti uvjetno grananje:



© Kovač, Basch, FER, Zagreb

114



- Prema prethodnim dijagramima toka, sigurno će nam trebati dvije vrste naredbe skoka i to:
 - Naredba bezuvjetnog skoka (promjena redoslijeda izvođenja)
 - Naredba uvjetnog skoka (grananje na jednu od dvije naredbe u ovisnosti o uvjetu)
- >>>
- Za obje naredbe, moramo imati operand kojim zadajemo **odredište skoka**, tj. adresu naredbe na koju želimo skočiti

© Kovač, Basch, FER, Zagreb

116



- Za **uvjetni skok** naredba se izvodi na dva moguća načina
 - Ako je uvjet ispunjen => skok se ostvaruje
 - Ako uvjet nije ispunjen => skok se ne ostvaruje, tj. izvodi se sljedeća naredba (ona koja je "ispod" naredbe skoka)
- Za naredbu uvjetnog skoka upotrijebimo isti naziv JP, ali ćemo mu dometnuti sufiks kojim ćemo označiti uvjet. Definirajmo način pisanja i operand naredbe JP:

JP_uvjet adresa

 - Naredba *JP_uvjet* skače na naredbu sa zadanom *adresom* samo ako je *uvjet* istinit, a inače nastavlja s izvođenjem sljedeće naredbe

© Kovač, Basch, FER, Zagreb

118



- Na primjer, uvjeti mogu biti sljedeći:
 - Jesu li dvije vrijednosti jednake?
 - Je li prva vrijednost veća od druge?
 - Je li prva vrijednost manja ili jednaka od druge?
 - itd. (općenito se uspoređuju dvije numeričke ili logičke vrijednosti)
- U uvodnom poglavlju već smo vidjeli kako se mogu usporediti dvije vrijednosti:
 1. prvo se izvede AL-operacija (najčešće je to oduzimanje)
 2. nakon toga se ispituju zastavice (Podsjetnik: zastavice su bistabili koji se postavljaju na temelju ALU-operacije)

© Kovač, Basch, FER, Zagreb

120

© Kovač, Basch, FER, Zagreb

Upravljačke naredbe - primjeri

Primjer uvjetnog grananja - naredba if:

Zbrojiti registre R0 i R1 i rezultat staviti u R2. Ako nema prijenosa, ne treba napraviti ništa, a ako ima treba pobrisati R2.

Rješenje:



© Kovač, Basch, FER, Zagreb

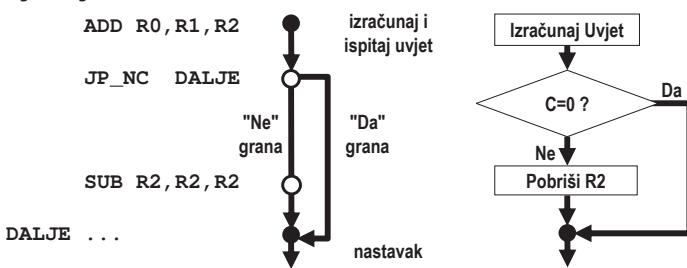
129

Upravljačke naredbe - primjeri

Primjer uvjetnog grananja - naredba if:

Zbrojiti registre R0 i R1 i rezultat staviti u R2. Ako nema prijenosa, ne treba napraviti ništa, a ako ima treba pobrisati R2.

Rješenje:



© Kovač, Basch, FER, Zagreb

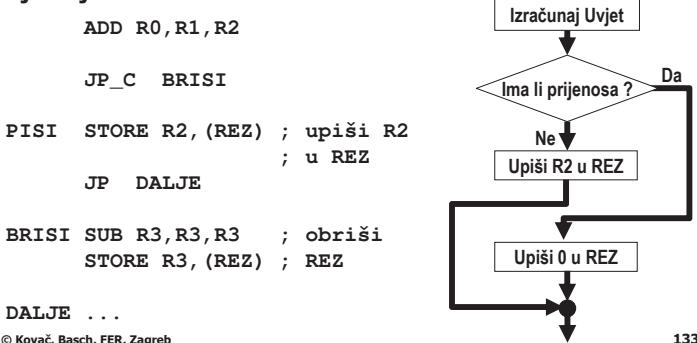
131

Upravljačke naredbe - primjeri

Primjer uvjetnog i bezuvjetnog grananja - naredba if-else:

Zbrojiti registre R0 i R1 i rezultat staviti u R2. Ako dođe do prijenosa treba obrisati memoriju REZ, a inače u nju treba upisati R2.

Rješenje:



© Kovač, Basch, FER, Zagreb

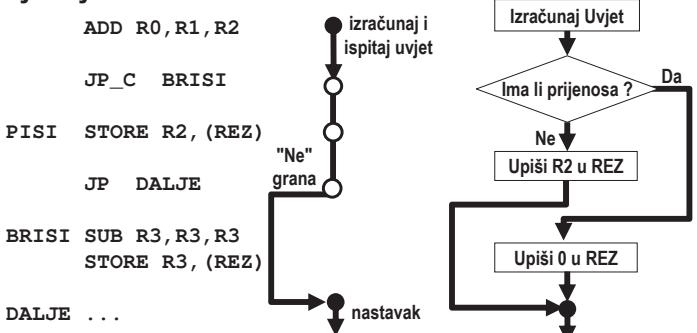
133

Upravljačke naredbe - primjeri

Primjer uvjetnog i bezuvjetnog grananja - naredba if-else:

Zbrojiti registre R0 i R1 i rezultat staviti u R2. Ako dođe do prijenosa treba na memoriju REZ upisati 0, a inače treba upisati R2.

Rješenje:



© Kovač, Basch, FER, Zagreb

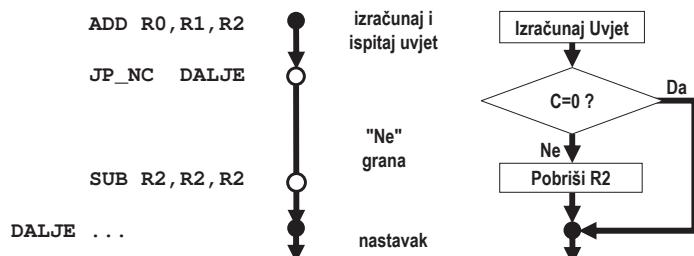
135

Upravljačke naredbe - primjeri

Primjer uvjetnog grananja - naredba if:

Zbrojiti registre R0 i R1 i rezultat staviti u R2. Ako nema prijenosa, ne treba napraviti ništa, a ako ima treba pobrisati R2.

Rješenje:



© Kovač, Basch, FER, Zagreb

130

Upravljačke naredbe - primjeri

ADD R0, R1, R2

JP_NC DALJE

SUB R2, R2, R2

DALJE ...

- Svaki uvjet može se napisati i na "obrnut" način. U praksi uvjet "okrećemo" tako da da dobijemo što kraći i razumljiviji program

• Prethodni program napisan s "obrnutim" uvjetom izgledao bi ovako:

```

    ADD R0, R1, R2
    JP_C BRISI
    JP DALJE
    BRISI SUB R2, R2, R2
    DALJE ...
  
```

© Kovač, Basch, FER, Zagreb

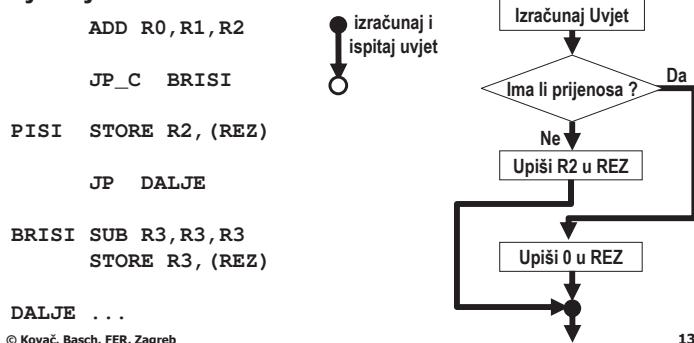
132

Upravljačke naredbe - primjeri

Primjer uvjetnog i bezuvjetnog grananja - naredba if-else:

Zbrojiti registre R0 i R1 i rezultat staviti u R2. Ako dođe do prijenosa treba na memoriju REZ upisati 0, a inače treba upisati R2.

Rješenje:



© Kovač, Basch, FER, Zagreb

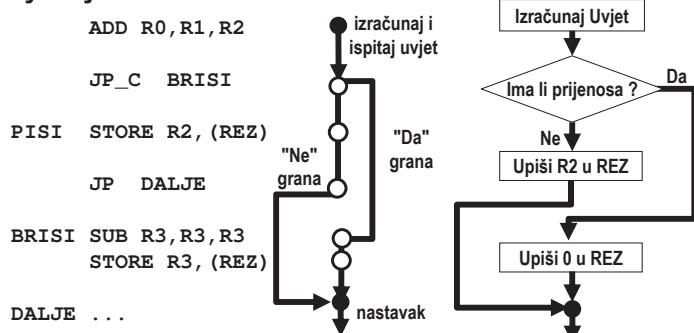
134

Upravljačke naredbe - primjeri

Primjer uvjetnog i bezuvjetnog grananja - naredba if-else:

Zbrojiti registre R0 i R1 i rezultat staviti u R2. Ako dođe do prijenosa treba na memoriju REZ upisati 0, a inače treba upisati R2.

Rješenje:



© Kovač, Basch, FER, Zagreb

136

Upravljačke naredbe - primjeri

Primjer petlje u postupku množenja:

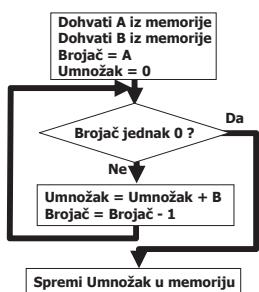
Treba pomnožiti dva NBC broja (označimo to kao A*B) koji su smješteni u memoriji na adresama 100 i 200. Rezultat množenja treba spremiti u memoriju na adresu 300.

Rješenje:

Program ćemo temeljiti na dijagramu toka. U programu ćemo vidjeti većinu onoga što smo do sada naučili:

- AL-naredbe,
- memoriske naredbe,
- rad s konstantama,
- naredbu uvjetnog i bezuvjetnog skoka.

Dodatno ćemo vidjeti i petlju s brojačem.



© Kovač, Basch, FER, Zagreb

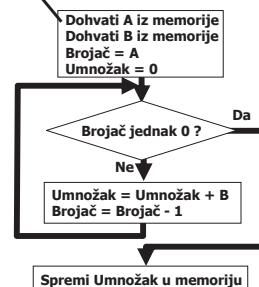
137

© Kovač, Basch, FER, Zagreb

138

LOAD R0, (100) ; R0 ≡ A

LOAD R0, (100) ; R0 ≡ A
LOAD R1, (200) ; R1 ≡ B



© Kovač, Basch, FER, Zagreb

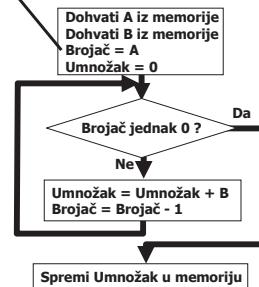
139

© Kovač, Basch, FER, Zagreb

140

LOAD R0, (100) ; R0 ≡ A
LOAD R1, (200) ; R1 ≡ B
OR R0, R0, R2 ; R2 ≡ Brojač := A

LOAD R0, (100) ; R0 ≡ A
LOAD R1, (200) ; R1 ≡ B
OR R0, R0, R2 ; R2 ≡ Brojač := A
LOAD R3, (NULA) ; R3 ≡ Umnožak := 0



© Kovač, Basch, FER, Zagreb

141

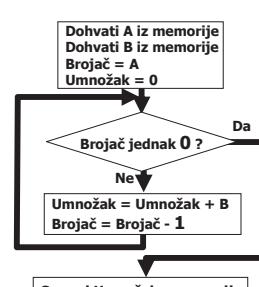
© Kovač, Basch, FER, Zagreb

142

LOAD R0, (100) ; R0 ≡ A
LOAD R1, (200) ; R1 ≡ B
OR R0, R0, R2 ; R2 ≡ Brojač := A
LOAD R3, (NULA) ; R3 ≡ Umnožak := 0
LOAD R4, (NULA) ; R4 ≡ 0
LOAD R5, (JEDAN) ; R5 ≡ 1

LOAD R0, (100) ; R0 ≡ A
LOAD R1, (200) ; R1 ≡ B
OR R0, R0, R2 ; R2 ≡ Brojač := A
LOAD R3, (NULA) ; R3 ≡ Umnožak := 0
LOAD R4, (NULA) ; R4 ≡ 0
LOAD R5, (JEDAN) ; R5 ≡ 1

Konstante 0 i 1 za kasnije korištenje u programu

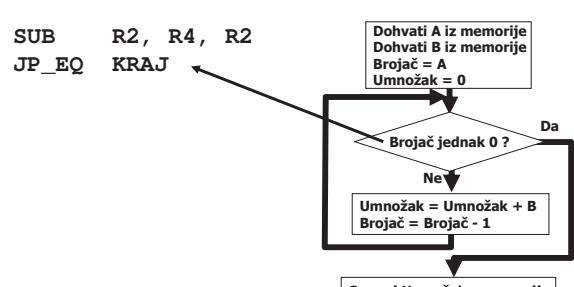


© Kovač, Basch, FER, Zagreb

143

© Kovač, Basch, FER, Zagreb

144



```

LOAD R0, (100) ; R0 ≡ A
LOAD R1, (200) ; R1 ≡ B
OR R0, R0, R2 ; R2 ≡ Brojač := A
LOAD R3, (NULLA) ; R3 ≡ Umnožak := 0
LOAD R4, (NULLA) ; R4 ≡ 0
LOAD R5, (JEDAN) ; R5 ≡ 1

```

```

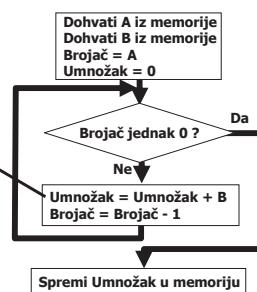
SUB R2, R4, R2
JP_EQ KRAJ

```

```

ADD R3, R1, R3

```



© Kovač, Basch, FER, Zagreb

145

```

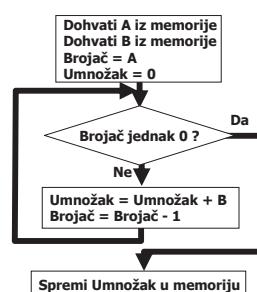
LOAD R0, (100) ; R0 ≡ A
LOAD R1, (200) ; R1 ≡ B
OR R0, R0, R2 ; R2 ≡ Brojač := A
LOAD R3, (NULLA) ; R3 ≡ Umnožak := 0
LOAD R4, (NULLA) ; R4 ≡ 0
LOAD R5, (JEDAN) ; R5 ≡ 1

```

```

PETLJA SUB R2, R4, R2
JP_EQ KRAJ
ADD R3, R1, R3
SUB R2, R5, R2
JP PETLJA

```



© Kovač, Basch, FER, Zagreb

147

```

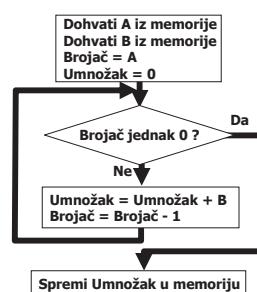
LOAD R0, (100) ; R0 ≡ A
LOAD R1, (200) ; R1 ≡ B
OR R0, R0, R2 ; R2 ≡ Brojač := A
LOAD R3, (NULLA) ; R3 ≡ Umnožak := 0
LOAD R4, (NULLA) ; R4 ≡ 0
LOAD R5, (JEDAN) ; R5 ≡ 1

```

```

PETLJA SUB R2, R4, R2
JP_EQ KRAJ
ADD R3, R1, R3
SUB R2, R5, R2
JP PETLJA

```



KRAJ STORE R3, (300)
...
NULA DW 0
JEDAN DW 1

© Kovač, Basch, FER, Zagreb

149

```

LOAD R0, (100) ; R0 ≡ A
LOAD R1, (200) ; R1 ≡ B
OR R0, R0, R2 ; R2 ≡ Brojač := A
LOAD R3, (NULLA) ; R3 ≡ Umnožak := 0
LOAD R4, (NULLA) ; R4 ≡ 0
LOAD R5, (JEDAN) ; R5 ≡ 1

```

```

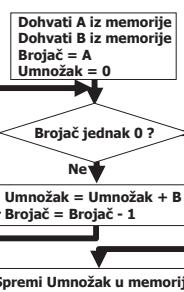
SUB R2, R4, R2
JP_EQ KRAJ

```

```

ADD R3, R1, R3
SUB R2, R5, R2

```



© Kovač, Basch, FER, Zagreb

146

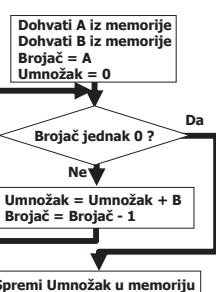
```

LOAD R0, (100) ; R0 ≡ A
LOAD R1, (200) ; R1 ≡ B
OR R0, R0, R2 ; R2 ≡ Brojač := A
LOAD R3, (NULLA) ; R3 ≡ Umnožak := 0
LOAD R4, (NULLA) ; R4 ≡ 0
LOAD R5, (JEDAN) ; R5 ≡ 1

```

```

PETLJA SUB R2, R4, R2
JP_EQ KRAJ
ADD R3, R1, R3
SUB R2, R5, R2
JP PETLJA
KRAJ STORE R3, (300)
...
```



© Kovač, Basch, FER, Zagreb

148

```

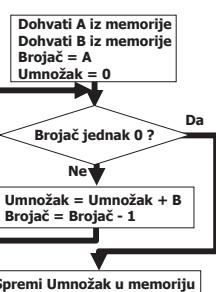
LOAD R0, (100) ; R0 ≡ A
LOAD R1, (200) ; R1 ≡ B
OR R0, R0, R2 ; R2 ≡ Brojač := A
LOAD R3, (400) ; R3 ≡ Umnožak := 0
LOAD R4, (400) ; R4 ≡ 0
LOAD R5, (410) ; R5 ≡ 1

```

```

PETLJA SUB R2, R4, R2
JP_EQ KRAJ
ADD R3, R1, R3
SUB R2, R5, R2
JP PETLJA

```



KRAJ STORE R3, (300)
...
400 DW 0
410 DW 1

© Kovač, Basch, FER, Zagreb

150

Strojni kôd naredaba

- Za sada smo definirali 8 naredaba i opisali što te naredbe rade
- Ponovimo:
 - procesor svaku naredbu mora dohvatiti, dekodirati i izvesti
 - dekodiranje je, zapravo, raspoznavanje naredbe kako bi se znalo što točno treba izvesti
 - svaka naredba je zapisana u memoriji (kao niz ništica i jedinica) i ima svoj posebni oblik po kojem procesor prepozna tu naredbu
- Dakle, svaka naredba mora imati **jedinstveni** zapis, koji će je **razlikovati** od svih drugih naredoba



Strojni kôd naredaba

Općenito

Strojni kôd naredaba

- Pomoću tog zapisa procesor **razlikuje** npr. naredbu zbrajanja od naredbe oduzimanja i od naredbe skoka ...
- Također, na isti način procesor **razlikuje** da li npr. naredba oduzimanja želi raditi nešto sa registrom R1 ili registrom R2 ili registrom R3 ili ...
- Zato se za procesor mora točno propisati (definirati) kako se svaka njegova pojedina naredba zapisuje u memoriji, tj. definira se **strojni kôd naredaba**
- Strojni kôd naredbe je niz bitova koji jednoznačno odgovara toj naredbi - svaka naredba ima svoj jedinstveni strojni kôd

Strojni kôd naredaba

- Strojni kôdovi naredaba definiraju se nizovima bitova
 - Strojni kôdovi obično su podijeljeni u više polja različitih širina u bitovima (polja ne moraju biti smještena u kontinuiranim bitovima, npr. vidi polje adrese na donjem dijelu slike). Na primjer:
- | | | | |
|-------------------------|-----------------------|-----------------------|-----------------------|
| polje operacijskog kôda | polje prvog operanda | polje drugog operanda | polje trećeg operanda |
| polje operacijskog kôda | polje adrese (1. dio) | polje operanda | polje adrese (2. dio) |

Strojni kôd naredaba

- Polja operanada** jednoznačno definiraju s kojim operandima naredba obavlja svoju zadaću
- Na primjer, za naredbu ADD:
 - Koji registar će biti prvi operand zbrajanja
 - Koji registar će biti drugi operand zbrajanja
 - U koji registar se spremi rezultat
- Registri se kodiraju tako da indeks registra bude zapisan binarnim brojem u polju operanda. Na primjer, za procesor s četiri registra R0, R1, R2 i R3, polje operanda bi imalo dva bita:
 - 00 u polju operanda označava registar R0
 - 01 u polju operanda označava registar R1
 - 10 u polju operanda označava registar R2
 - 11 u polju operanda označava registar R3

Strojni kôd naredaba

- Postoje i druge vrste polja, ovisno o naredbenom skupu pojedinog procesora
- Na primjer, u naredbi može biti **polje podatka** u kojem je zapisan podatak (npr. broj) s kojim naredba obavlja određenu zadaću (npr. spremi ga u registar ili ga pribraja registru)
- Na primjer, u naredbi može biti **polje načina adresiranja** koje definira da li naredba ADD treba zbrojiti dva registra ili treba zbrojiti podatak i registar

>>>

Strojni kôd naredaba

- Treba strogo razlikovati:
 - naredbu zapisanu tekstom** kao npr. "ADD R1,R2,R3" što je samo način kako programer piše naredbe prilikom programiranja
 - strojni kôd naredbe**, što je prikaz naredbe u obliku niza ništica i jedinica u memoriji računala
- Treba također biti svjestan uske veze ovog tekstovnog zapisa i strojnog kôda, jer su oni u odnosu jedan-na-jedan:
 - iz tekstovnog zapisa naredbe točno se zna njezin strojni kôd
 - iz strojnog kôda naredbe točno se zna njezin tekstovni zapis

Strojni kôd naredaba

- Polje operacijskog kôda** jednoznačno definira o kojoj se naredbi radi (ili o kojoj skupini srodnih naredba se radi):
 - Radi li se o naredbi ADD
 - Radi li se o naredbi SUB
 - Radi li se o naredbi LOAD
 - itd.
- Svaka naredba definirana je jedinstvenim brojem, na primjer:
 - Operacijski kôd naredbe ADD je 00000_2
 - Operacijski kôd naredbe SUB je 00001_2
 - Operacijski kôd naredbe LOAD je 10010_2
 - itd.

Strojni kôd naredaba

- Polje adrese** jednoznačno definira memorisku adresu potrebnu naredbi za obavljanje njene zadaće
- Na primjer, za naredbu LOAD:
 - S koje memoriske adrese naredba čita podatak
- Na primjer, za naredbu STORE:
 - Na koju memorisku adresu naredba spremi podatak
- Na primjer, za naredbu JP:
 - Na koju memorisku adresu treba skočiti (tj. na kojoj adresi se nalazi sljedeća naredba)

Strojni kôd naredaba

- Na primjer, u naredbi može biti **polje uvjeta** u kojem je definiran uvjet na temelju kojeg naredba obavlja određenu zadaću
 - To će nam trebati za naredbu JP koja, ovisno o uvjetu, izvodi skok ili ga ne izvodi
- U naredbi može postojati i bilo kakvo drugo polje koje pobliže definira način izvođenja naredbe, ili služi da je razlikuje od drugih naredaba, ili služi za razlikovanje načina adresiranja itd.



- Koliko bita mogu biti široke naredbe?
- Moguća su dva rješenja: >>>
 - definirati da procesor ima sve naredbe jednake širini procesorske riječi
 - definirati da procesor ima naredbe različitih širina

Strojni kôd naredaba – širina naredaba



- CISC procesori često imaju naredbe bez fiksne širine, nego su široke jednu ili dvije ili više riječi – ovisno o naredbi:

operacijski kôd {ADD}	1. operand {R0}	2. operand {R2}	3. operand {R3}	ADD R0, R2, R3
operacijski kôd {naredba skoka}	polje uvjeta {uvjet}	---	proširenje op. kôda {JP}	JPuvjet 200
		adresa {200}		

operacijski kôd {ALU naredba}	1. operand {R0}	polje načina adresiranja {apsolutno i reg.ind. s pom.}		ADD R0, (200), (R2+6)
		adresa {200}		
proširenje op. kôda {ADD}	3. operand {R2}	adresni pomak {6}		

Strojni kôd naredaba – širina naredaba



- Dvije i više riječi su potrebne kada se unutar strojnog kôda naredbe trebaju nalaziti podatci široki kao i procesorska riječ:
 - memorijска adresa
 - podatak s kojim naredba radi
 - adresni pomak
- Prednost:
 - Nema ograničenja na broj naredaba i njihovu složenost
- Nedostatci:
 - Za dohvata naredbe potrebno je više čitanja iz memorije
 - Komplikirana arhitektura i dekodiranje (nakon dohvata prve riječi strojnog kôda raspoznaće se koliko još riječi treba dohvatiti)

Strojni kôd naredaba procesora FRISC

Strojni kôd naredaba - FRISC



- Mi projektiramo ugradbeni procesor koji:
 - treba biti što jednostavniji
 - treba imati mali broj naredaba
 - ima mali broj registara
 - treba imati mali broj načina adresiranja
- Zato, odabiremo da širina strojnog kôda bude procesorska riječ, odnosno 32 bita za sve naredbe
- Također ćemo se truditi da polja budu raspoređena što pravilnije kako bi se pojednostavnilo i ubrzalo dekodiranje naredaba

Strojni kôd naredaba - FRISC



- Strojni kôd definiramo na sljedeći način:

operacijski kôd	ovisno o pojedinoj naredbi (tj. o ovisno o op. kôdu)
-----------------	---

- Za operacijski kôd moramo uzeti dovoljno bitova da možemo razlikovati sve potrebne naredbe (ili grupe naredaba)
- Odabiremo 5 bitova (najviši bitovi, tj. bitovi od 27 do 31) što nam daje ukupno $2^5 = 32$ kombinacija čime možemo izravno razlikovati 32 naredbe
(za sada imamo samo 8 naredaba, ali ćemo ih kasnije proširivati)

Strojni kôd - AL naredbe

- AL-naredbe imaju tri operanda, a svi su registri. Svaki operand kodiramo sa po tri bita (jer ima 8 registara, $8 = 2^3$)
- Nakon operacijskog kôda, kodiramo redom:
 - jedan bit (26) je neiskorišten
 - tri bita (23 do 25) za odredište (dest)
 - tri bita (20 do 22) za prvi operand (src1)
 - tri bita (17 do 19) za drugi operand (src2)
 - preostalih 17 bitova (0 do 16) se ne koriste

operacijski kôd	---	dest	src1	src2	---
31-27	26	25-23	22-20	19-17	16-0



Strojni kôd - memorijске naredbe

- Memorijske naredbe imaju dva operanda:
 - prvi operand je registar čije značenje ovisi o naredbi:
 - za LOAD znači odredišni registar u koji se puni vrijednost (dest)
 - za STORE znači izvorišni registar iz kojeg se čita vrijednost (src)
 - drugi operand je adresa memorijске lokacije s koje se čita/piše podatak
- Nakon operacijskog kôda, kodiramo redom:
 - jedan bit (26) je neiskorišten
 - tri bita (23 do 25) za prvi operand, tj. za registar (dest/src)
 - tri bita (20 do 22) se ne koriste
 - preostalih 20 bitova (0 do 19) sadrže drugi operand, tj. adresu

operacijski kôd	---	dest/src	---	20-bitna adresa
31-27	26	25-23	22-20	19-0



Strojni kôd - upravljačke naredbe

- Upravljačke naredbe imaju uvjet i jedan operand:
 - uvjet se piše kao dio naredbe (sufiks)
 - operand je adresa skoka (20 bita)
- Nakon operacijskog kôda, kodiramo redom:
 - jedan bit (26) je neiskorišten
 - četiri bita (22 do 25) za uvjet (cond)
 - dva bita (20 do 21) se ne koriste
 - preostalih 20 bitova (0 do 19) sadrže operand, tj. adresu skoka

operacijski kôd	---	cond	---	20-bitna adresa skoka
31-27	26	25-22	21-20	19-0



Strojni kôd - upravljačke naredbe

- Pogledajmo još je li za kodiranje uvjeta dovoljno 4 bita kojima se može kodirati 16 različitih uvjeta?
- Iz popisa uvjeta imamo sljedeće različite uvjete:
 - 8 uvjeta za izravno ispitivanje zastavica
 - 8 uvjeta za usporedbu brojeva
 - 2 uvjeta za ispitivanje jednakosti brojeva: EQ, NE
 - 2 uvjeta za ispitivanje predznaka: M i P
 - 1 uvjet koji je uvijek istinit (za bezuvjetni JP)
- Ukupno imamo 21 različitih uvjeta što je više od 16 mogućih pa na prvi pogled izgleda da imamo premalo bitova u polju

>>>

Strojni kôd - AL naredbe

- Iz ovako definiranog strojnog kôda vidimo da imamo neiskorišteno ukupno 18 bitova
- Njih ćemo kasnije upotrijebiti za daljnja proširenja i poboljšanja AL-naredaba

operacijski kôd	---	dest	src1	src2	---
31-27	26	25-23	22-20	19-17	16-0



Strojni kôd - memorijске naredbe

- Vidimo da je adresa ograničena na samo 20 bitova, iako smo definirali da će adresna sabirica i adresa imati 32 bita
- Na ovom primjeru vidimo kako odluka da svaka naredba ima samo 32 bita ograničava podatke i adrese koje moramo kodirati u naredbi
- Posljedice ovog ograničenja objasnit ćemo kasnije

operacijski kôd	---	dest/src	---	20-bitna adresa
31-27	26	25-23	22-20	19-0



Strojni kôd - upravljačke naredbe

- Usporedimo polje adrese s istim poljem u memorijskim naredbama:
 - i ovdje adresa ima samo 20 bitova od potrebnih 32
 - razlika u značenju adrese je da ovdje adresa predstavlja odredište skoka, a ne položaj podatka za čitanje/pisanje
 - budući da obje naredbe imaju jednako polje za adresu, strojni kôd je pravilniji pa će i dekodiranje i izvođenje naredaba biti jednostavnije i brže (barem tako očekujemo)

operacijski kôd	---	cond	---	20-bitna adresa skoka
31-27	26	25-22	21-20	19-0



Strojni kôd - upravljačke naredbe

- <<<
- Ali, ovdje se radi samo o različitim načinima **pisanja** uvjeta
- Bitno je koliko postoji različitih načina **ispitivanja zastavica**, tj. koliko postoji različitih načina izvođenja naredbe, jer se samo to mora razlikovati u strojnom kôdu
- Iz tablica uvjeta vidimo da dio uvjeta ispituje zastavice na jednak način i da postoji samo 15 različitih ispitivanja zastavica:
 - Z i NZ ispituju zastavice kao i uvjeti EQ i NE
 - N i NN ispituju zastavice kao i uvjeti M i P
 - C i NC ispituju zastavice kao i uvjeti ULT i UGE
- Dakle: 4 bita su dovoljna





Osnovna inačica procesora

Pregled arhitekture - Rekapitulacija

© Kovač, Basch, FER, Zagreb

177

OSNOVNA INAČICA PROCESORA



- Osnovne aritmetičko logičke naredbe našeg procesora:

ADD src1, src2, dest	AND src1, src2, dest
SUB src1, src2, dest	OR src1, src2, dest
	XOR src1, src2, dest

- src1, src2, dest* su parametri kojima izabiremo koji registar opće namjene će biti prvi operand (*src1*), koji će biti drugi (*src2*) i u koji će se spremi rezultat (*dest*)

- Primjeri korištenja:

ADD R1, R2, R3 ; R1+R2->R3
XOR R3, R7, R3 ; R3 xor R7->R3

operacijski kód	-	dest	src1	src2	--
31-27	26	25-23	22-20	19-17	16-0

© Kovač, Basch, FER, Zagreb

178

OSNOVNA INAČICA PROCESORA



- Osnovne upravljačke naredbe (zapravo samo jedna ☺)

JP adr
JP_uvjet adr

- adr* je parametar kojim definiramo adresu naredbe na koju treba skočiti ako je uvjet istinit
- uvjet* je sufiks koji definira način ispitivanja zastavica

- Primjeri korištenja:

JP 100 ; bezuvjetno skoči na naredbu na adresi 100
JP_NZ 100 ; ako zastavica Z nije postavljena, onda skoči na adresu 100

operacijski kód	-	cond	--	20-bitna adresa skoka
31-27	26	25-22	21-20	19-0

© Kovač, Basch, FER, Zagreb

179

Poboljšana inačica procesora

Detaljniji pogled na arhitekturu skupa naredaba i mikroarhitekturu

© Kovač, Basch, FER, Zagreb

181

© Kovač, Basch, FER, Zagreb

180

Poboljšana inačica procesora



- Za detaljnije definiranje naredaba moramo vidjeti:

- Potrebne/praktične promjene i proširenja u skupu naredaba
- Adresiranja
- Strojne kódove konačnog skupa naredaba
- Način izvođenja naredaba i protočnu strukturu

- Za detaljnije definiranje mikroarhitekture moramo vidjeti:

- Dijelove procesora i način njihovog spajanja
- Put podataka i upravljačku jedinicu
- Prikљučke procesora
- Spajanje procesora s memorijom i vanjskim jedinicama
- Način komunikacije procesora s memorijom i vanjskim jedinicama

182

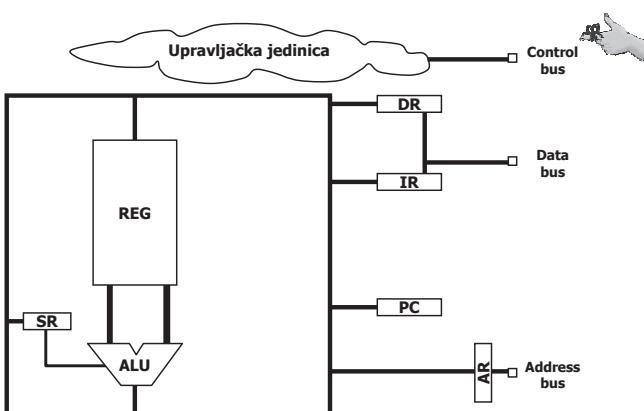
© Kovač, Basch, FER, Zagreb

184

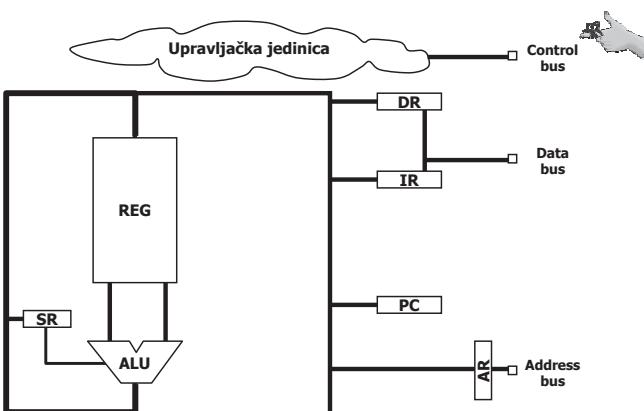


- Pogledajmo kako načelno izgleda mikroarhitektura našeg procesora:
 - koje dijelove sadrži
 - kako su ti dijelovi povezani
- Slika će za početak biti samo shematska i pojednostavljena, a kasnije ćemo je prikazivati sve detaljnije

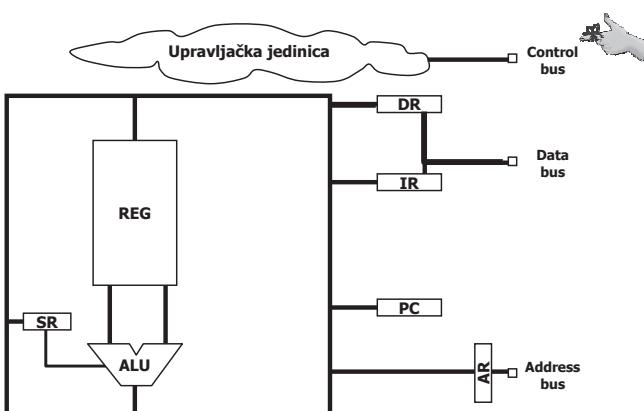
>>>



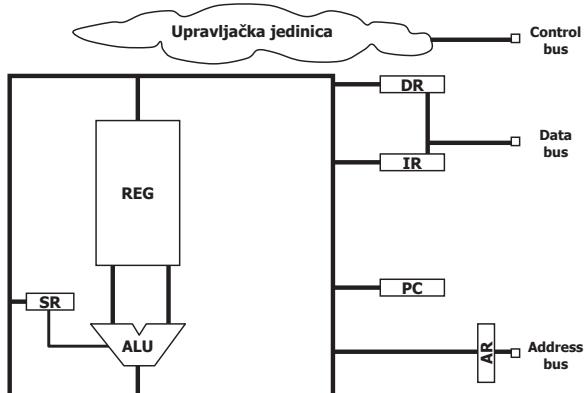
Iz registara opće namjene postoje dva spojna puta do ALU za slanje operanada u AL-naredbama



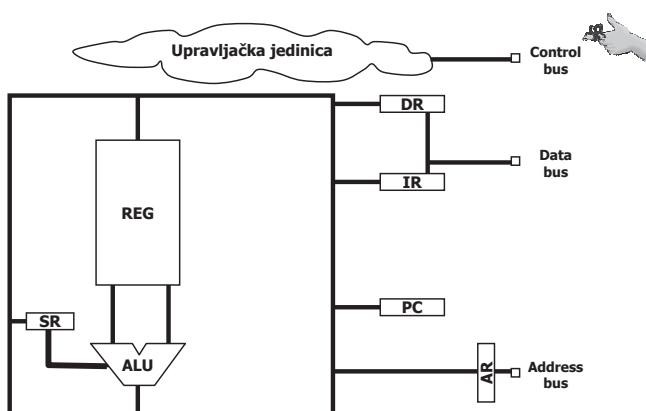
Postoji spojni put od izlaza ALU do skupa općih registara, kako bi u AL-naredbama rezultat mogao biti zapisan u jedan od općih registara



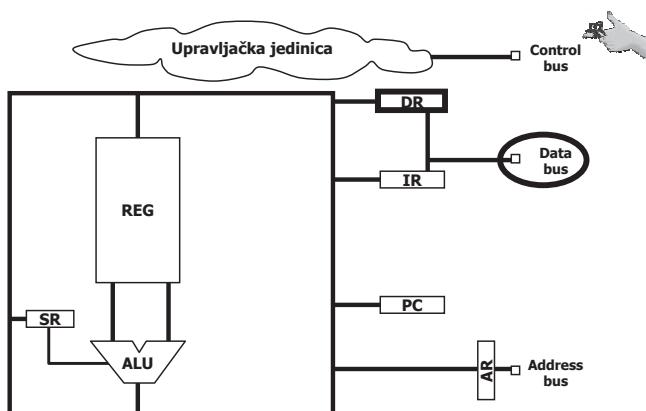
Podatci koji se čitaju i pišu preko sabirnice podataka uvijek prolaze kroz registar DR (uz jednu iznimku)



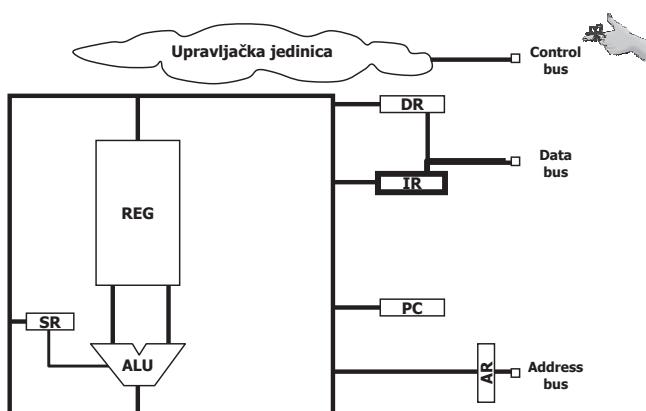
Na slici vidimo skup registara opće namjene (REG) i ostale registre procesora kao i način na koji su načelno spojeni >>>



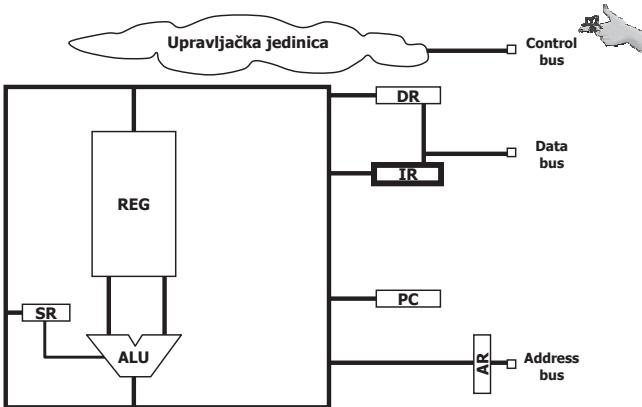
Iz ALU postoji veza do registra stanja SR, kako bi AL-operacija mogla utjecati na stanje zastavica



Registrar podataka DR (Data Register) služi kao međuspremnik između unutrašnjosti procesora i sabirnice podataka, koja je spojena na procesor preko podatkovnih priključaka



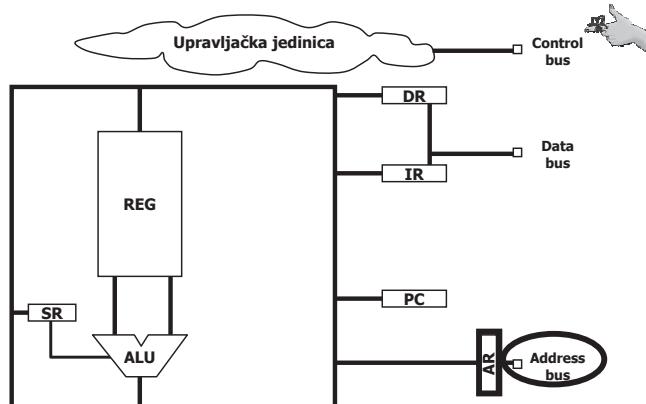
Ta iznimka je dohvatanje naredbe iz memorije. Tada se ona ne sprema u DR, nego izravno u naredbeni register IR (Instruction Register)



Za vrijeme dekodiranja naredbe njezin strojni kôd je spremljen u registru IR i čita ga upravljačka jedinica (veza od IR do upravljačke jedinice nije prikazana na slici).

© Kovač, Basch, FER, Zagreb

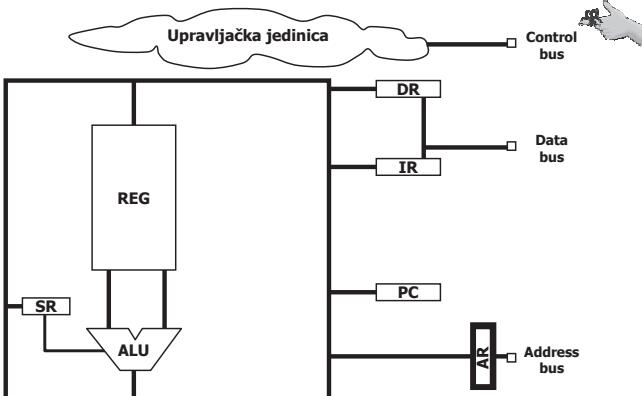
193



Adresni registar AR (Address Register) služi kao međuspremnik između unutrašnjosti procesora i adresne sabirnice, koja je spojena na procesor preko adresnih priključaka.

© Kovač, Basch, FER, Zagreb

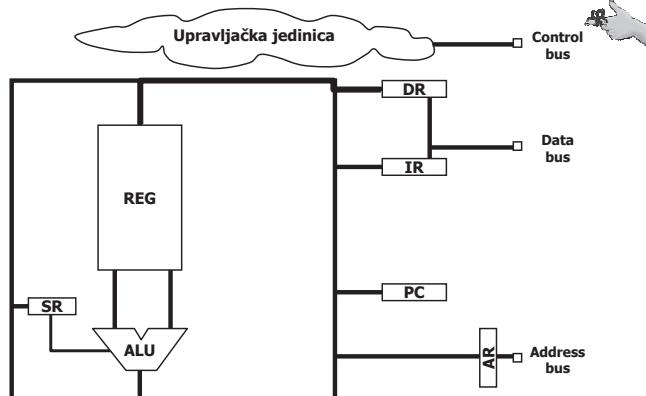
194



Registrar AR uvijek se koristi kad procesor pristupa memoriji. U AR se spremava adresa memorijске lokacije koju procesor želi čitati ili pisati.

© Kovač, Basch, FER, Zagreb

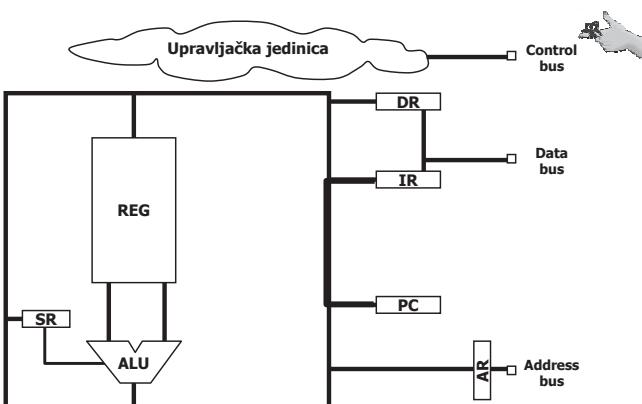
195



Postoji spojni put između registra DR i općih registara jer tim putom prolaze podatci prilikom izvođenja naredaba LOAD i STORE

© Kovač, Basch, FER, Zagreb

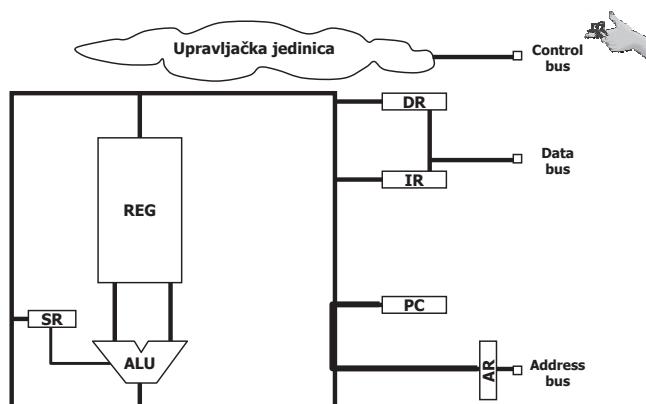
196



Registrar IR u sebi čuva strojni kôd u kojem može biti adresa za naredbu skoka JP. Ta adresa šalje se u PC.

© Kovač, Basch, FER, Zagreb

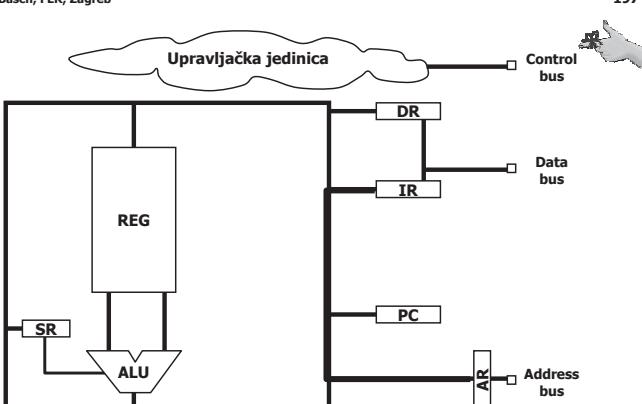
197



Postoji spojni put između registra PC i AR jer se tim putom šalje adresa sljedeće naredbe koju želimo dohvatiti iz memorije

© Kovač, Basch, FER, Zagreb

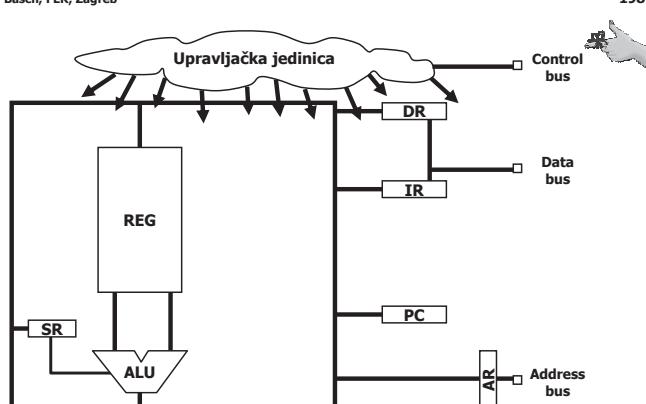
198



Registrar IR u sebi čuva strojni kôd u kojem može biti adresa za naredbe LOAD/STORE. Ta adresa šalje se u adresni registar AR.

© Kovač, Basch, FER, Zagreb

199



Veze od upravljačke jedinice do ostalih dijelova procesora nisu prikazane jer ih ima previše - svaki dio upravljan je s jednim ili više signalima koji dolaze iz upravljačke jedinice

© Kovač, Basch, FER, Zagreb

200



- Za sada će nam ovo objašnjenje mikroarhitekture biti dovoljno
- Novi važni dijelovi procesora koje smo upravo uveli su:
 - **DR** - podatkovni registar koji služi kao međusklop između unutrašnjosti procesora i podatkovne sabirnice
 - **AR** - adresni registar koji služi kao međusklop između unutrašnjosti procesora i adresne sabirnice
 - **IR** - naredbeni registar koji čuva strojni kôd naredbe za vrijeme njenog dekodiranja
- Ovi registri su interni, u smislu da programer nema izravnog pristupa do njih

Proširenja skupa naredaba



- Do sada uvedene naredbe omogućuju izvođenje većine zadaća koje nam mogu zatrebatи
- Ipak, da bi programiranje bilo lakše, uvest ćemo još nekoliko naredaba i načina adresiranja
 - način adresiranja = način na koji se pristupa podatu
- Pri tome moramo voditi računa o strojnim kôdovima, jer oni također ograničavaju: broj naredaba, broj načina adresiranja, vrste operanada, širine podataka, širine adresa itd.
- Pogledajmo neke praktične zadaće koje bi mogli lakše riješiti s drugačijim ili novim naredbama ...

Proširenja AL-naredaba

- Pretpostavimo da treba registar R0 uvećati za 20. Sada je moguće ovakvo rješenje:


```
LOAD R1, (100)
ADD R0, R1, R0
...
100 DW 20 ; na adresi 100 nalazi se broj 20
```
- Loše strane ovog rješenja:
 - Potrebne su dvije naredbe (brzina i zauzeće memorije)
 - Potrebna je dodatna memorijska lokacija (s brojem 20)
 - Treba koristiti dodatni registar (npr. R1). Moguće je da on nije slobadan, pa će ga trebati spremiti i kasnije obnoviti (za što trebaju još dvije dodatne naredbe i još jedna dodatna memorijska lokacija)

Proširenja AL-naredaba



- Proširimo naredbu ADD tako da kao drugi operand može imati:
 - registar (kao i do sada)
 - broj (novo)
- Moramo vidjeti ima li mesta za ovakvo proširenje u strojnem kôdu i kako će on sada izgledati
- Do sada smo imali ovakav strojni kôd:
 

operacijski kôd	---	dest	src1	src2	---
31-27	26	25-23	22-20	19-17	16-0

Detaljnija arhitektura i proširenja naredbenog skupa

Proširenja aritmetičko-logičkih naredaba

Proširenja AL-naredaba

- Bolje rješenje bi moglo izgledati ovako:


```
ADD R0, 20, R0
```
- Dobre strane ovog rješenja:
 - Potrebna je samo jedna naredba
 - Ne trebaju dodatne memorijske lokacije
 - Ne treba koristiti dodatne registre
- Ali, naredba ADD se komplificira:
 - Operandi više nisu samo registri, nego mogu biti i brojevi
- Jedno od pravila pri oblikovanju procesora:
 - Ubrzati rad onoga što se često koristi

Proširenja AL-naredaba

- Moramo imati različit strojni kôd za različite vrste operanada:
 - Neiskorišteni bit 26 (adr) će označavati o kojem se obliku naredbe radi (to je dodatno 1-bitno polje koje određuje **način adresiranja**):
 - bit 26 u 0 (ništici) označava da su oba operanda registri
 - bit 26 u 1 (jedinici) označava da je drugi operand broj

operacijski kôd	adr	operandi - ovisno o polju adr
31-27	26	25-0

Proširenja AL-naredaba

- Kad su oba operanda registri, strojni kôd ostaje kao prije s razlikom da je bit adr (26) u ništici:

operacijski kôd	0	dest	src1	src2	---
31-27	26	25-23	22-20	19-17	16-0

- Kad je drugi operand broj, bit adr (26) je u jedinici, a zatim redom kodiramo
 - tri bita (23 do 25) za prvi odredište (dest)
 - tri bita (20 do 22) za prvi operand (src1)
 - preostalih 20 bitova (0 do 19) sadrže drugi operand, tj. broj

operacijski kôd	1	dest	src1	20-bitni broj
31-27	26	25-23	22-20	19-0

© Kovač, Basch, FER, Zagreb

209

Proširenja AL-naredaba - primjer

- Pokažimo na primjeru što smo dobili uvođenjem broja kao drugog operanda. Rješenje ovog primjera smo već vidjeli ranije.

Primjer izraza s brojevima:

Izračunati sljedeće: $R0 := (R1+55)-(R2 \text{ xor } 4ABC)$. Za razliku od starog rješenja, koristiti novouvedeno zadavanje podatka u AL-naredbama.

Rješenje:

```
ADD  R1, 55, R0 ; prvi dio izraza
XOR  R2, 4ABC, R3 ; drugi dio izraza
SUB   R0, R3, R0 ; razliku spremi u R0
```

>>>

© Kovač, Basch, FER, Zagreb

211

Proširenja AL-naredaba - proširenje broja

- Aritmetičko-logička jedinica je 32-bitna pa, prema tome, očekuje 32-bitne operande
- U okviru strojnog kôda za kodiranje broja imamo na raspolaganju samo 20 bitova
- Što s gornjih 12 bitova koji nedostaju? Ove bitove treba nekako definirati.
- Procesor će 20-bitni broj (iz strojnog kôda) prije slanja u ALU **predznačno proširiti** na 32 bita



© Kovač, Basch, FER, Zagreb

213

Proširenja AL-naredaba - pomaci i rotacije



- U asemblerском programiranju često treba obaviti različite vrste pomaka i rotacija podataka:
 - logički pomak uljevo i udesno
 - aritmetički pomak udesno
 - rotacija uljevo i udesno
 - rotacija uljevo i udesno kroz zastavicu
- Mogu se dozvoliti pomaci i rotacije samo za jedan bit ili za željeni broj bitova
- Teorijski bi mogli odabrati samo dvije operacije pomaka/rotiranja, a ostale ostvariti programski.
 - To bi bilo u skladu s idejom "jednostavnog procesora"

Proširenja AL-naredaba



- Teorijski bi mogli imati i oblike:
 - ADD 20, R1, R2 - prvi operand je broj, a drugi je registar
 - ADD 20, 30, R4 - oba operanda su brojevi
- Međutim, to ima nedostataka:
 - Osim bita 26, trebao bi dodatni bit za odabir vrste prvog operanda
 - Ne dobiva se na fleksibilnosti naredaba
 - Oblik s dva podatka je besmislen jer troši vrijeme na izračunavanje konstante vrijednosti, a u strojnog kôdu i nema mesta za dva broja
- Zato, ostajemo na samo dva predložena oblika, ali će oni **vrijediti za sve aritmetičko-logičke naredbe** (radi pravilnosti i jednostavnosti arhitekture)

© Kovač, Basch, FER, Zagreb

210

Proširenja AL-naredaba - primjer



Staro rješenje vs Novo rješenje:

LOAD R0, (BR_55)	ADD R1, 55, R0
ADD R1, R0, R0	XOR R2, 4ABC, R3
LOAD R3, (BR_4ABC)	SUB R0, R3, R0
XOR R2, R3, R3	; Podatci u memoriji
SUB R0, R3, R0	BR_55 DW 55
; Podatci u memoriji	BR_4ABC DW 4ABC

ADD R1, 55, R0
XOR R2, 4ABC, R3
SUB R0, R3, R0

Komentar:

Prednosti u odnosu na prethodni program su očite:

- kraći program (3 naredbe prema 5)
- nema dodatnih podataka u memoriji (2 podatka)
- brže izvođenje (3 ciklusa prema 7 - o ciklusima kasnije)

© Kovač, Basch, FER, Zagreb

212

Proširenja AL-naredaba - proširenje broja



- Podsjetnik:

- Proširenje ništicama čuva iznos NBC brojeva**
(ali ne čuva iznos 2^k brojeva)
- Predznačno proširenje čuva iznos 2^k brojeva**
(ali ne čuva iznos NBC brojeva)

- U praksi je u aritmetičkim operacijama puno potrebnije imati i pozitivne i negativne brojeve, nego puni opseg NBC-a.
 - Zato naš procesor koristiti predznačno proširenje

© Kovač, Basch, FER, Zagreb

214

Proširenja AL-naredaba - pomaci i rotacije



- Međutim, da bi pojednostavnili programiranje i ubrzali izvođenje, odabrat ćemo nešto veći broj naredaba za operacije:
 - logičkog pomaka uljevo i udesno
 - aritmetičkog pomaka udesno
 - rotacije uljevo i udesno
- Također ćemo definirati da se izlazni bit sprema u zastavicu C kako bi ga mogli jednostavno ispitati nakon naredbe



- Definiramo pisanje i operande naredaba pomaka i rotacije:

SHL src1, src2, dest	logički pomak u lijevo (SHift Left)
SHR src1, src2, dest	logički pomak u desno (SHift Right)
ASHR src1, src2, dest	aritmetički pomak u desno (Arithmetic SHR)
ROTL src1, src2, dest	rotacija u lijevo (ROTate Left)
ROTR src1, src2, dest	rotacija u desno (ROTate Right)

- Podatak koji se pomiče/rotira uzima se iz prvog operanda (src1)
- Broj pomaka/rotacija zadan je drugim operandom (src2)
- Rezultat pomaka/rotacije stavlja se u treći operand (dest)

"pomakni podatak iz src1 za src2 bitova i spremi rezultat u dest"

Proširenja AL-nar. za višestruku preciznost



- Dvije AL-naredbe koje služe za rad s podatcima u višestrukoj preciznosti su naredbe zbrajanja i oduzimanja s prijenosom:
 - ADC (add with carry)
 - SBC (subtract with carry)
 - Kasnije ćemo vidjeti kako se ove naredbe koriste i kako točno rade
 - Ove naredbe imaju jednake operande kao i obično zbrajanje i oduzimanje. Pišu se ovako:
- ```
ADC src1, src2, dest ; src1+src2+prijenos->dest
SBC src1, src2, dest ; src1-src2-posudba->dest
```
- Sklopovska izvedba ovih naredaba je vrlo jednostavna (kasnije ćemo vidjeti kakva) i ne traži dodatno specijalno sklopovlje

## Rekapitulacija: proširenja AL-naredaba



- Ovime smo kompletirali skup aritmetičko-logičkih naredaba, kojih sada ima 13:
  - ADD, SUB, ADC, SBC
  - CMP
  - AND, OR, XOR
  - SHL, SHR, ASHR, ROTL, ROTR
- Osim toga, kao drugi operand sada osim registra smijemo pisati i 20-bitni broj

## Proširenja AL-naredaba - primjer

### Primjer:

U registru R0 nalaze se dvije NBC poluriće. Treba usporediti te poluriće. Ako je viša poluriječ veća od niže, onda treba skočiti na adresu 100, a ako nije, onda treba skočiti na adresu 200.

|                |                |    |
|----------------|----------------|----|
| viša poluriječ | niža poluriječ | R0 |
|----------------|----------------|----|

Postupak: da bismo mogli obaviti usporedbu, moramo poluriće imati u zasebnim registrima, npr. R0 i R1.

|   |                |    |
|---|----------------|----|
| 0 | niža poluriječ | R0 |
| 0 | viša poluriječ | R1 |

>>>



Broj pomaka/rotacija najčešće je:

- poznat unaprijed, tj. zadan je brojem (čest slučaj)
- koji puta se mora izračunati (rezultat izračunavanja je npr. u registru)
- Zato smo odabrali da se broj pomaka/rotacija zadaje drugim operandom, jer u AL-naredbama on jedini može biti i registar i podatak
- Efektivno, samo najnižih 5 bitova drugog operanda utječu na broj pomaka/rotacija koji će se izvesti, jer je smisleno napraviti pomak/rotaciju za 1 do 31 bit (dozvoljeno je zadati i 0)

## Proširenja AL-naredaba - naredba usporedbe



- Zadnja AL naredba koju ćemo uvesti, a koja postoji u većini procesora je naredba za usporedbu dvaju brojeva CMP (compare)
- Ova naredba slična je naredbi za oduzimanje SUB, s razlikom da se rezultat oduzimanja zanemaruje i ne upisuje u jedan od općih registara (CMP nema treći operand)
- Ovo je ujedno i prednost naredbe CMP, jer su u većini slučajeva registri zauzeti s različitim podatcima i međurezultatima
- Naredbu CMP zapravo pozivamo zato da postavi zastavice koje ćemo nakon toga ispitati naredbom uvjetnog skoka
- Naredba se piše ovako (drugi operand je registar ili broj):

CMP src1, src2 ; src1-src2

## Proširenja AL-naredaba - zastavice



- Utjecaj aritmetičko-logičkih naredaba na zastavice:
- ADD i ADC: C=prijenos, V=preljev, Z=niština, N=predznak
- SUB, SBC i CMP: C=posudba, V=preljev, Z=niština, N=predznak
- AND, OR i XOR: C=0, V=0, Z=niština, N=predznak
- SHL/R, ASHR, ROTL/R: C=izlazni bit, V=0, Z=niština, N=predznak
  - Napomena: u naredbama pomaka/rotacije, u zastavicu C upisuje se izlazni bit od zadnjeg koraka pomaka/rotacije.

## Proširenja AL-naredaba - primjer

### Rješenje:

```
ROTR R0, 10, R1 ; viša poluriječ u niži dio R1
AND R0, 0FFFF, R0 ; više bitove R0 stavljamo u 0
AND R1, 0FFFF, R1 ; više bitove R1 stavljamo u 0

CMP R1, R0 ; usporedba

JP_UGT 100 ; viša p.r. > niža p.r. ⇒ "goto 100"
JP 200 ; viša p.r. <= niža p.r. ⇒ "goto 200"
```



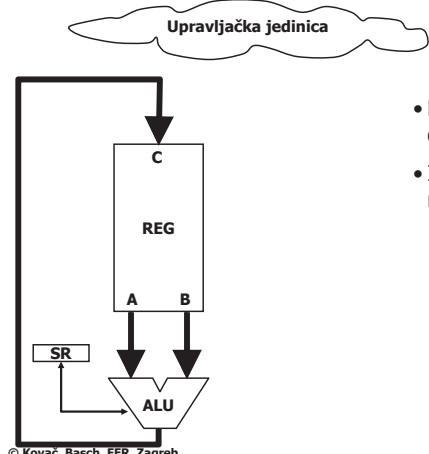
- Na kraju prethodnog poglavlja prikazali smo shematski mikroarhitekturu FRISC-a, bez ulaženja u detalje
- U ovom poglavlju ćemo postupno uvesti detaljniji prikaz arhitekture koji se u literaturi obično naziva **put podataka** (engl. datapath)

>>>

### Osnovna građa procesora

- Put podataka koji ćemo mi prikazati neće biti potpun jer bi to daleko prelazilo opseg gradiva ovog predmeta
- Ipak, on će biti dovoljno detaljan da će se iz njega vidjeti sva najvažnija načela mikroarhitekture procesora
- Za početak, prikažimo dio puta podataka zaduženog za izvođenje AL-naredaba ...

### Proširenja AL-naredaba - put podataka



### Proširenja AL-naredaba - put podataka

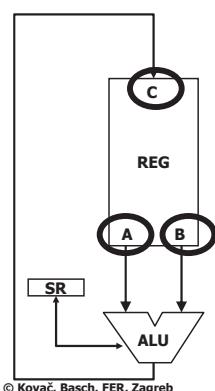


- Registarski izlazi A i B dovode se na ulaze ALU
- Izlaz iz ALU dovodi se na registarski ulaz C

### Osnovna građa procesora

- Put podataka pokazuje tijek podataka sa stajališta redoslijeda izvođenja
- Stvarni spojni putovi u procesoru mogu biti implementirani onako kako je prikazano putom podataka, ali i na drugačiji način, na primjer internim sabirnicama

### Proširenja AL-naredaba - put podataka



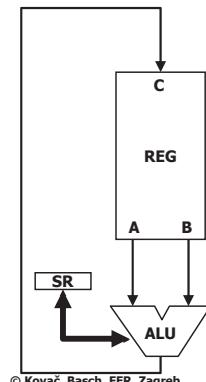
Prikažimo detaljnije put podataka potreban za izvođenje AL-naredaba (za početak bez mogućnosti korištenja broja kao drugog operanda)

- Skup registara ima jedan ulaz (C) i dva izlaza (A i B)
- Moguće je istovremeno pročitati sadržaje dvaju registara i staviti ih na izlaze A i B



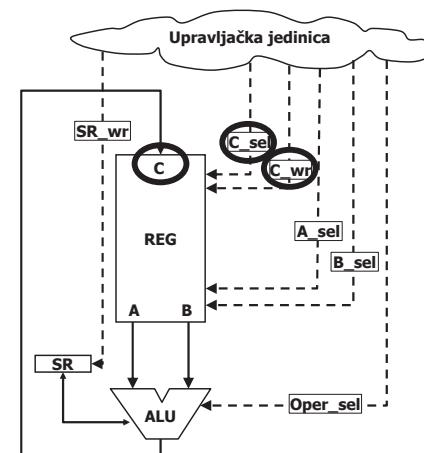
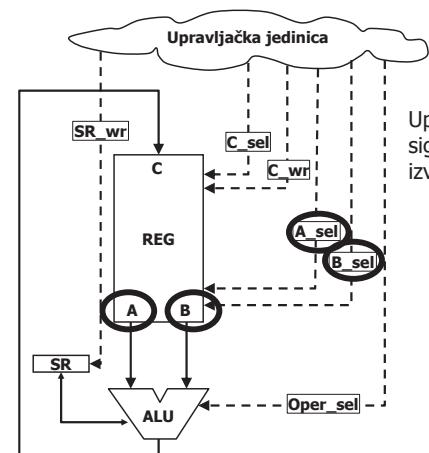
>>>

### Proširenja AL-naredaba - put podataka

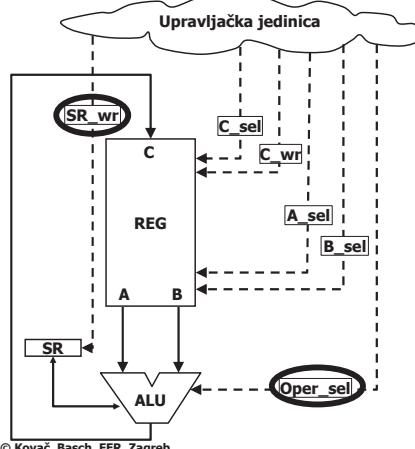


- Kao što smo već vidjeli na pojednostavljenoj shemi:
  - ALU utječe na stanje SR
  - SR utječe na izvođenje AL-operacije (za naredbe ADC i SBC)

### Proširenja AL-naredaba - put podataka

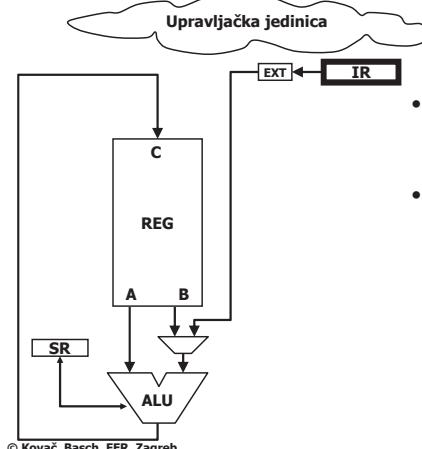


- C\_sel odabire register koji će se napuniti vrijednošću prisutnom na ulazu C
- C\_wr zadaje trenutak upisa u register odabran sa C\_sel



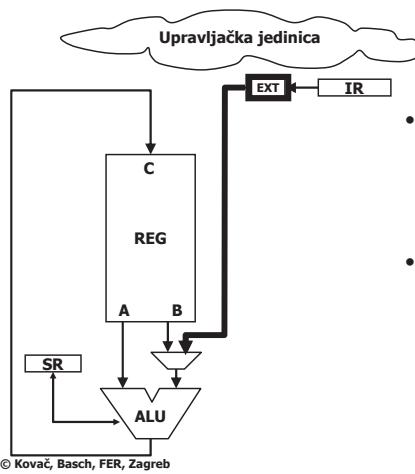
© Kovač, Basch, FER, Zagreb

233



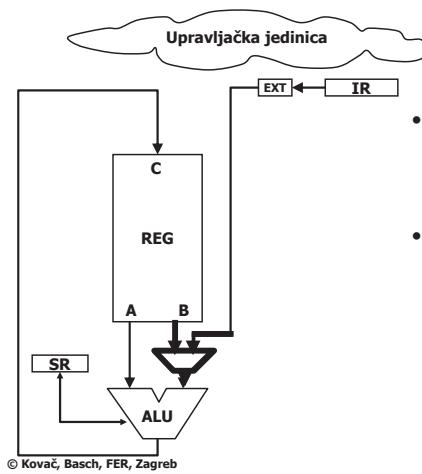
© Kovač, Basch, FER, Zagreb

234



© Kovač, Basch, FER, Zagreb

235



© Kovač, Basch, FER, Zagreb

236

## Uvođenje registrarskih naredaba

© Kovač, Basch, FER, Zagreb

237

## Uvođenje registrarskih naredaba



- Bolje rješenje bi bilo postojanje naredbe kojom možemo napuniti konstantu u registar. Ova naredba mogla bi izgledati ovako:

MOVE 32, R1

- Naredba bi značila sljedeće: upiši broj 32 u registar R1 (tj. upiši prvi operand u drugi operand)

© Kovač, Basch, FER, Zagreb

238

## Uvođenje registrarskih naredaba

- Koji puta (ali ne previše često) treba vrijednost iz jednog registra upisati u drugi registar
  - Sada to možemo rješiti pomoću naredbe LOAD:
 

```
LOAD R1, (BROJ)
BROJ DW 32
```
- Loše strane ovog rješenja su:
  - Potrebna je memorijska naredba (vidjet ćemo kasnije da se memorijske naredbe izvode sporije od npr. AL-naredaba)
  - Potrebna je dodatna memorijska lokacija (s brojem)
- Koji puta (ali ne previše često) treba vrijednost iz jednog registra upisati u drugi registar
  - Sada to možemo rješiti pomoću AL-naredaba, npr. naredbom OR, AND, ADD, SUB ili ROTL/ROTR. Pokažimo kako bi upisali vrijednost iz R1 u R2:
 

```
OR R1, R1, R2
AND R1, R1, R2
ADD R1, 0, R2
SUB R1, 0, R2
ROTL R1, 0, R2
```
- Nije estetski, ali uglavnom funkcioniра ☺ ...



- Međutim, AL-naredbe mijenjaju zastavice u registru SR. To koji puta može biti nepoželjno.
- Rješenje bi moglo biti da prvo spremimo SR, napunimo željeni registar vrijednošću drugog registra te na kraju obnovimo SR
  - ovo rješenje je "pomalo komplikirano"
  - a osim toga, ne znamo kako spremiti/obnoviti registar SR ???
- Ponovno bi rješenje mogla biti (nepostojeca) naredba MOVE:
 

```
MOVE R1, R2
```
- Naredba bi opet značila sljedeće: upiši prvi operand u drugi operand (tj. upiši podatak iz R1 u registar R2)



- Zbog svega navedenog, uvodimo novu naredbu MOVE
- Ona po svojim značajkama ne pripada ni jednoj postojećoj skupini naredaba (AL, memoriske, upravljačke)
  - svrstat ćemo je u zasebnu skupinu registarskih naredaba (jer ona prvenstveno radi s registrima)
- Definiramo pisanje i operative naredbe MOVE:

```
MOVE src, dest
```

- src - izvor podatka; može biti: SR, R0-R7 ili 20-bitni podatak koji se predznačno proširuje na 32 bita
- dest - odredište podatka; može biti: SR ili R0-R7

### Primjer učitavanja brojeva u registre:

U registar R0 treba upisati broj 12345, u registar R1 broj FFFF1234, u R2 broj -100A, u R3 broj 12345678, a u registar R4 broj 9ABCDEF0.

- Prvi način je izravno korištenje naredbe MOVE.
- Ovo je pogodno za brojeve malog apsolutnog iznosa.
- "Mali brojevi" su oni koji "stanu" u strojni kod.

```
MOVE 12345, R0 ; 12345 stane u 17+1 bit
MOVE 0FFFF1234, R1 ; isto kao -EDCC, a
 ; EDCC stane u 16+1 bit
MOVE -100A, R2 ; 100A stane u 13+1 bit
```

>>>



- 20-bitni broj iz strojnog kôda se prilikom izvođenja predznačno proširuje na 32 bita
  - Dakle, dobiveni 32-bitni broj će sigurno u svih gornjih 13 bitova imati ili sve ništice ili sve jedinice (ovisno o najvišem bitu 20-bitnog broja)
- Prilikom pisanja programa mogu se pisati pozitivni i negativni brojevi: 123, -2, FFFF5678 itd.
  - Asemblerski prevoditelj svaki ovaj broj prvo pretvori u 32-bitni zapis:
    - 32-bitni NBC ako je broj pozitivan
    - 32-bitni 2'k zapis ako je broj negativan
  - Ako su u dobivenom 32-bitnom zapisu gornjih 13 bitova isti, onda je sve u redu i u strojni kôd se upisuje najnižih 20 bitova 32-bitnog zapisu
  - Ako gornjih 13 bitova nisu isti, onda je to pogrešno napisana naredba (poruka: wrong number)



- Na prethodnom slajdu je spomenuto da ne znamo kako spremiti i obnoviti vrijednost registra SR
- Ovo je vrlo **velik nedostatak**, jer je to ključno učiniti pri obradi prekida i u nekim drugim situacijama
- Ponovno bi rješenje mogla biti naredba MOVE:
 

```
MOVE SR, R2
MOVE R3, SR
```
- Obje naredbe bi opet upisivale prvi operand u drugi operand:
  - Prva naredba bi vrijednost iz SR upisala u R2
  - Druga naredba bi vrijednost iz R3 upisala u SR



- Registar SR je 8-bitni (to još ne znamo, ali tako će ispasti ☺) pa moramo definirati kako MOVE barata s podatcima različitih širina (Ri označuje opći registar R0-R7):
  - **MOVE podatak, SR** → nakon predznačnog proširenja podatka na 32 bita, odbacuje se gornjih 24 bita i u SR se puni samo najnižih 8 bita podatka
  - **MOVE Ri, SR** → u SR se puni najnižih 8-bitu iz registra Ri
  - **MOVE SR, Ri** → SR se puni u najniže bitove od Ri, a viši bitovi se pune ništicama\*

\* Iz SR-a se u registar Ri zapravo kopira 8+3 bita, što će biti objašnjeno kasnije kod prekidnog prijenosa

<<<

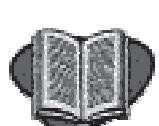
; Drugi način je pogodan za "velike brojeve" koji ne "stanu" u strojni kod. Broj se "puni" u registar u više koraka: Na primjer, kombinacijom MOVE i ADD

```
MOVE 1234, R3 ; 12345678 stane u 29+1 bit
ROTL R3, %D 16, R3
ADD R3, 5678, R3
```

; Treći način za "velike brojeve" znamo od prije: upotreba naredbe LOAD i dodatne memoriske lokacije

```
LOAD R4, (BR_9ABCDEF0)
```

```
BR_9ABCDEF0 DW 9ABCDEF0
```



- Za domaću zadaču proučite dokument "Proširivanje 20 na 32 bita" koji se nalazi u repozitoriju materijala na fer-webu ([www.fer.hr/predmet/arh1/repositorij](http://www.fer.hr/predmet/arh1/repositorij)) morate biti logirani na fer-web)
- U dokumentu "Proširivanje 20 na 32 bita" je prethodni slajd detaljnije objašnjen



- Strojni kôd registrarskih naredaba izgleda ovako:

|                 |    |       |       |       |      |
|-----------------|----|-------|-------|-------|------|
| operacijski kôd | 0  | dest  | 000   | src   | ---  |
| 31-27           | 26 | 25-23 | 22-20 | 19-17 | 16-0 |

|                 |    |       |       |               |
|-----------------|----|-------|-------|---------------|
| operacijski kôd | 1  | dest  | 000   | 20-bitni broj |
| 31-27           | 26 | 25-23 | 22-20 | 19-0          |

>>>

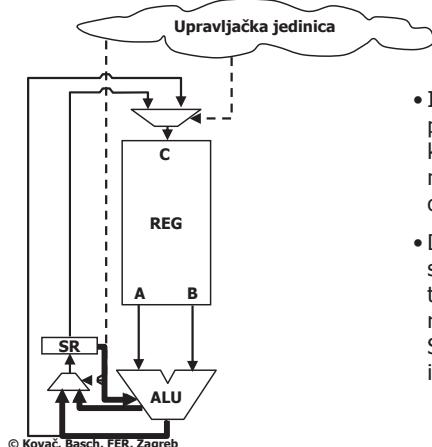


<<<

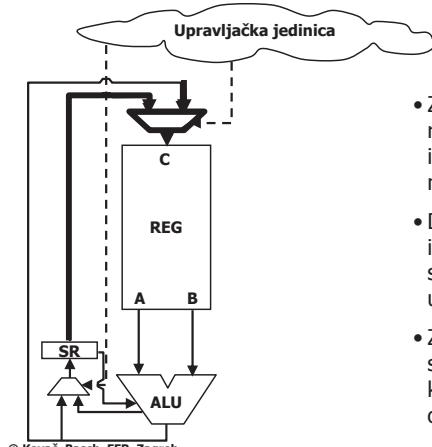
- Kada je izvor registar SR:
  - u bitovima 20 do 22 je podatak 010 u kojem bit 21 označuje da se ne koriste bitovi 0 do 19, niti bit 26
  - dest označuje odredišni registar

|                 |    |       |       |      |
|-----------------|----|-------|-------|------|
| operacijski kôd | -  | dest  | 010   | ---  |
| 31-27           | 26 | 25-23 | 22-20 | 19-0 |

- Dakle, ovisno o bitovima 20 do 22 moguće su sljedeće operacije:
  - za 000, izvodi se  $R_i/\text{podatak} \rightarrow R_i$
  - za 001, izvodi se  $R_i/\text{podatak} \rightarrow SR$
  - za 010, izvodi se  $SR \rightarrow R_i$



- Između ALU i SR od prije postoji dvosmjerna veza kojom AL-operacija utječe na stanje zastavica i obratno
- Dodatno, izlaz iz ALU spojen je na ulaz u SR, tako da se stanje općeg registra može proslijediti u SR (kroz ALU koji pritom ne izvodi nikakvu operaciju)



- Za upis SR-a u opći registar trebaju dodatni izlazi iz SR koji se dovode na registrski ulaz C
- Dovođenje na ulaz C nije izravno, jer tu već postoji spojni put od ALU prema ulazu C
- Zato se izlazi iz SR i ALU spajaju na mješavnik kojem upravljačka jedinica određuje odabir ulaza



<<<

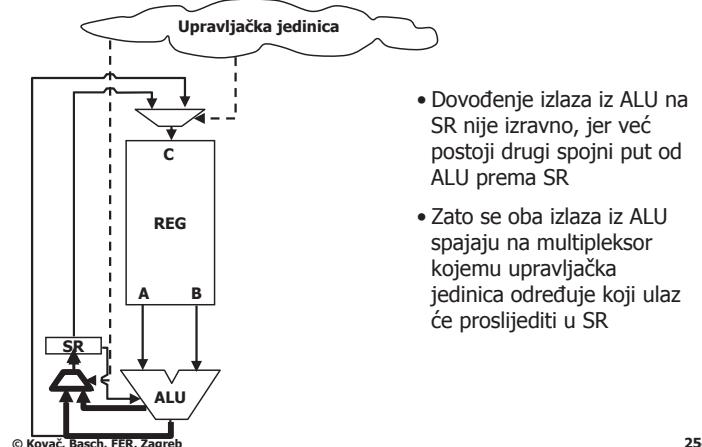
- Kada je odredište registar SR:
  - u bitovima od 20 do 22 je podatak 001 u kojem bit 20 označuje da se polje dest ne koristi
  - src označuje izvorišni registar
  - bit 26 određuje vrstu izvorišta (registar ili podatak)

|                 |    |       |       |       |      |
|-----------------|----|-------|-------|-------|------|
| operacijski kôd | 0  | ---   | 001   | src   | ---  |
| 31-27           | 26 | 25-23 | 22-20 | 19-17 | 16-0 |

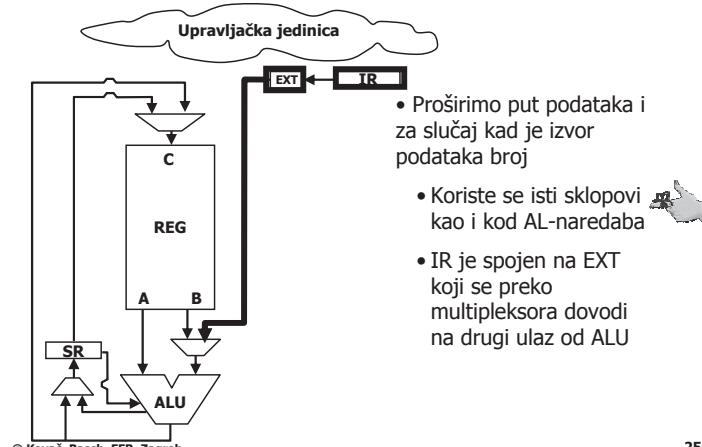
|                 |    |       |       |               |
|-----------------|----|-------|-------|---------------|
| operacijski kôd | 1  | ---   | 001   | 20-bitni broj |
| 31-27           | 26 | 25-23 | 22-20 | 19-0          |

>>>

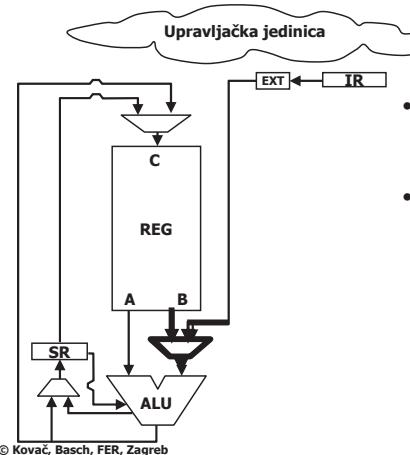
- Naredba MOVE bit će jedina u skupini registrarskih naredaba
- Da bi se ona mogla izvoditi, morat ćemo dopuniti put podataka s dodatnim spojnim putovima:
  - iz registra SR u opće registre
  - iz općeg registra u SR
  - iz IR u opći registar i SR
- Kako bi izbjegli stvaranje prevelikog broja specijaliziranih putova, iskoristit ćemo postojeće putove koji prolaze kroz ALU
  - za vrijeme izvođenja naredbe MOVE, ALU neće izvoditi operaciju nego će samo propuštati drugi operand na svoj izlaz



- Dovođenje izlaza iz ALU na SR nije izravno, jer već postoji drugi spojni put od ALU prema SR
- Zato se oba izlaza iz ALU spajaju na mješavnik kojemu upravljačka jedinica određuje koji ulaz će proslijediti u SR



- Proširimo put podataka i za slučaj kad je izvor podataka broj
- Koriste se isti skloovi kao i kod AL-naredaba
- IR je spojen na EXT koji se preko mješavnika dovodi na drugi ulaz od ALU



© Kovač, Basch, FER, Zagreb

257

- Ovisno o izvoru podataka bira se izlaz iz EXT ili izlaz B iz skupa općih registara
- ALU ne obavlja operaciju nego samo proslijeduje svoj drugi operand na izlaz koji se dalje vodi u SR ili opći registar

## Proširenje memorijskih naredaba

258

### Proširenje memorijskih naredaba



- Podsjetimo se da su memorijске naredbe LOAD i STORE imale zadavanje adrese **brojem**
- U strojnog kôdu je za adresu bilo na raspolaganju samo 20 bitova
- Budući da smo za adresnu sabirnicu odabrali širinu od 32 bita, **moramo** točno definirati vrijednost gornjih 12 bitova

© Kovač, Basch, FER, Zagreb

259

### Proširenje memorijskih naredaba



- Zbog pravilnosti arhitekture definirat ćemo da se i u naredbama LOAD i STORE **konačna 32-bitna adresa dobiva predznačnim proširivanjem 20-bitne adrese iz strojnog kôda**
  - Nedostatak: adrese je prirodnije promatrati kao NBC-brojeve jer su adrese uvijek pozitivne (bilo bi prirodnije proširenje ništicama)
  - Prednost: pravilnost arhitekture i naredaba

© Kovač, Basch, FER, Zagreb

260

### Proširenje memorijskih naredaba



- U strojnem kôdu ostalih naredaba postoji pravilnost:
  - AL-naredbe mogu imati u nižih 20 bitova zadan 20-bitni broj koji će biti drugi operand. Ovaj broj se predznačno proširuje prije dovođenja u ALU
  - Registarske naredbe mogu imati u nižih 20 bitova zadan broj koji se predznačno proširuje i stavlja u odredišni registar

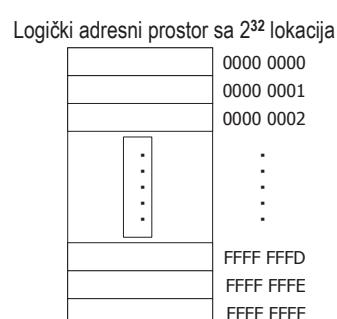
© Kovač, Basch, FER, Zagreb

260

### Proširenje memorijskih naredaba - adrese



- Kako predznačno proširenje adrese utječe na adresni prostor?**



**• Logički adresni prostor** je raspoloživi opseg adresa koje procesor može generirati na svojim adresnim priključcima



**• Stvarna fizička veličina memorije i adresnog prostora** u računalu manja je ili jednaka logičkom prostoru



© Kovač, Basch, FER, Zagreb

261

### Proširenje memorijskih naredaba - adrese



- Adrese koje se mogu dobiti predznačnim proširenjem 20-bitnog broja su:**

Logički adresni prostor sa  $2^{32}$  lokacija

|           |                                                               |
|-----------|---------------------------------------------------------------|
| 0000 0000 | od 0000 0000 do 0007 FFFF<br>(za "pozitivne" 20-bitne adrese) |
| 0000 0001 |                                                               |
| 0000 0002 |                                                               |
| ...       |                                                               |
| FFFF FFFD |                                                               |
| FFFF FFFE |                                                               |
| FFFF FFFF |                                                               |

262

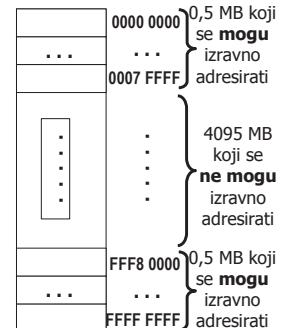
© Kovač, Basch, FER, Zagreb

262

### Proširenje memorijskih naredaba - adrese



- Adresama 0000 0000 do 0007 FFFF adresiramo najnižih  $2^{19}$  lokacija memorije (najnižih pola MB)
- Adresama FFF8 0000 do FFFF FFFF adresiramo najviših  $2^{19}$  lokacija memorije (najviših pola MB),
  - što ukupno daje  $2^{20}$  lokacija (tj. jedan MB)
- Na ovaj način **ne možemo** adresirati svih  $2^{32}$  lokacija (tj. 4 GB ili 4096 MB)



© Kovač, Basch, FER, Zagreb

263

© Kovač, Basch, FER, Zagreb

264

## Proširenje memorijskih naredaba - adrese

- Adresni prostor od 1 megabajta je dovoljan za potrebe ugradbenog računala, ali ne i za opću upotrebu procesora
- Trebamo zadavanje bilo koje 32-bitne adrese
- Moguća rješenja
  - Proširiti strojni kôd (nekih) naredaba tako da zauzimaju dvije riječi i u drugu riječ staviti 32-bitnu adresu
  - Upotrijebiti jedan od općih registara za adresiranje
- Budući da smo odlučili da sve naredbe budu jednake širine, odabiremo drugo rješenje

© Kovač, Basch, FER, Zagreb

265

## Proširenje memorijskih nar. - strojni kôd



- Pogledajmo kako ukloputi ovu promjenu u trenutni strojni kôd:

| operacijski kôd | --- | dest/src | ---   | 20-bitna adresa |
|-----------------|-----|----------|-------|-----------------|
| 31-27           | 26  | 25-23    | 22-20 | 19-0            |

- Morat ćemo imati različit strojni kôd za ova dva slučaja
  - Bit 26 (adr) će označavati o kojem se obliku naredbe radi (to je dodatno 1-bitno polje koje određuje način adresiranja):
    - ništica (0) označava da je adresa zadana 20-bitnim brojem
    - jedinica (1) označava da je adresa zadana registrom

| operacijski kôd | adr | dest/src | ovisno o bitu adr |  |
|-----------------|-----|----------|-------------------|--|
| 31-27           | 26  | 25-23    | 22-0              |  |

© Kovač, Basch, FER, Zagreb

267

## Proširenje memorijskih naredaba - primjer



### Primjer rada s adresama većim od 20 bita:

Zamijeniti vrijednosti memorijskih lokacija s adresa 200 i 300000.

#### Rješenje:

```
LOAD R2, (200) ; Dohvati prvi podatak.
LOAD R0, (A_300000) ; Stavi broj 300000 u R0 i s
LOAD R3, (R0) ; njim adresiraj drugi podatak.

STORE R3, (200)
STORE R2, (R0)

; U memoriji na adresi A_300000 mora biti upisan
; broj 300000 koji ćemo koristiti kao adresu:
A_300000 DW 300000 ; Služi kao adresa za drugi podatak
...
200 DW 1234 ; Prvi podatak.
...
300000 DW 2468 ; Drugi podatak.
```

© Kovač, Basch, FER, Zagreb

269

## Proširenje memorijskih naredaba - odmak



- Konačno, drugi oblik memorijskih naredaba izgleda ovako:

```
LOAD R1, (R4+20)
STORE R2, (R5-10)
STORE R3, (R6) // odmak=0, ne mora se pisati
```

- Adresa se tijekom izvođenja formira na sljedeći način:
  - 1) 20-bitni odmak se predznačno proširi do 32 bita,
  - 2) Vrijednost adresnog registra zbroji se s 32-bitnim odmakom,
  - 3) Ovaj zbroj je konačna 32-bitna adresa za memorisku naredbu.

© Kovač, Basch, FER, Zagreb

271

## Proširenje memorijskih naredaba

- Sada ćemo moći naredbe LOAD i STORE pisati na primjer ovako:

```
LOAD R0, (R5)
STORE R1, (R4)
STORE R2, (R3)
```



a prije smo mogli samo ovako:

```
LOAD R0, (100)
STORE R1, (20000)
STORE R2, (LABELA)
```

© Kovač, Basch, FER, Zagreb

266

## Proširenje memorijskih nar. - strojni kôd



- U prvom obliku kodiramo 20-bitnu adresu koja se predznačno proširuje čime daje konačnu adresu

| operacijski kôd | 0  | dest/src | ---   | 20-bitna adresa |
|-----------------|----|----------|-------|-----------------|
| 31-27           | 26 | 25-23    | 22-20 | 19-0            |

- U drugom obliku kodiramo registar (adrreg) u kojem će biti adresa

| operacijski kôd | 1  | dest/src | adrreg | ---  |
|-----------------|----|----------|--------|------|
| 31-27           | 26 | 25-23    | 22-20  | 19-0 |

© Kovač, Basch, FER, Zagreb

268

## Proširenje memorijskih nar. - strojni kôd

- U drugom obliku nam ostaje neiskorištenih 20 najnižih bitova:

| operacijski kôd | 1  | dest/src | adrreg | ---  |
|-----------------|----|----------|--------|------|
| 31-27           | 26 | 25-23    | 22-20  | 19-0 |

- Naredbu možemo učiniti još fleksibilnijom i praktičnijom za upotrebu ako uvedemo 20-bitni odmak (engl. offset)

- Dobivamo konačni drugi oblik memorijskih naredaba u kojem kodiramo adresni registar (adrreg), ali i dodatni 20-bitni odmak:

| operacijski kôd | 1  | dest/src | adrreg | 20-bitni odmak |
|-----------------|----|----------|--------|----------------|
| 31-27           | 26 | 25-23    | 22-20  | 19-0           |

© Kovač, Basch, FER, Zagreb

270

## Proširenje memorijskih naredaba - odmak



- Konačni oblik naredaba LOAD i STORE jeste:

```
LOAD dest, (adr)
LOAD dest, (adrreg+odmak)
STORE src, (adr)
STORE src, (adrreg+odmak)
```

- src i dest - izvor odnosno odredište podatka; može biti: R0-R7
- adr - adresa zadana 20-bitnim brojem
- adrreg - opći registar koji zadaje adresu, tj. adresni registar; može biti R0-R7;
- odmak - adresni odmak od adrese zadane sa adrreg; zadan je 20 bitnim brojem (adrreg i odmak razdvojeni su znakom + ili -)

© Kovač, Basch, FER, Zagreb

272

© Kovač, Basch, FER, Zagreb

272

## Proširenje memorijskih nar. - širina podatka

- Naredbe LOAD i STORE rade s 32-bitnim podatcima koji su u memoriji zapisani u četiri uzastopne memorijске lokacije
- Da bi mogli pristupati pojedinim memorijskim lokacijama, tj. bajtovima, **dodat ćemo još i naredbe LOADB i STOREB** (load byte i store byte) koje imaju jednake operative kao i LOAD i STORE
- Naredba LOADB čita jednobajtni podatak iz jedne memorijске lokacije i stavlja ga u najniži bajt registra. Gornja tri bajta registra se pune ništicama
- Naredba STOREB upisuje najniži bajt iz registra u jednu memorijsku lokaciju



## Proširenje memorijskih naredaba - strojni kôd

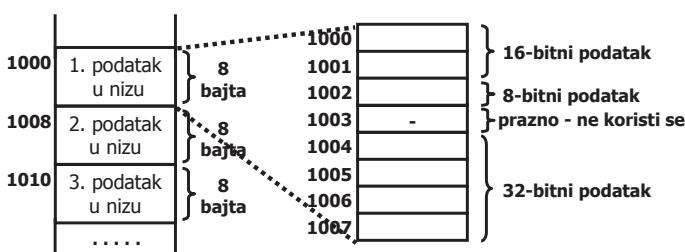
- Dodatne naredbe LOADB, STOREB, LOADH i STOREH imaju strojni kôd građen posve jednakom kao i LOAD i STORE
- Sve ove naredbe međusobno se razlikuju samo po najviših 5 bitova strojnog kôda, tj. po operacijskom kôdu
- Naredbe se pišu jednakom kao LOAD i STORE, osim imena same naredbe



## Proširenje memorijskih naredaba - primjeri

### Primjer adresiranja memorije adresnim registrom i odmakom:

U memoriji se od adrese 1000 nalazi niz 8-bajtnih složenih podataka kao na slici (npr. C-ova struktura sa short-om, char-om i int-om):

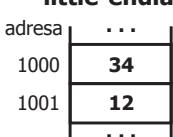


Treba kopirati 16 i 32-bitni dio iz prvog podatka u nizu u treći podatak u nizu, a 8-bitni dio treba kopirati iz trećeg podatka u prvi

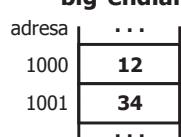
## Proširenje memorijskih naredaba - endianness

- Moramo još definirati raspored bajtova od 16 i 32-bitnih podataka u memoriji. Ovaj redoslijed se na engleskom naziva *endianness*.
- Ovisno o redoslijedu zapisivanja bajtova u memoriji imamo:
  - little-endian (npr. Intel) - niži bajt na nižu adresu
  - big-endian (npr. Motorola) - niži bajt na višu adresu
- Primjer zapisa 16-bitnog broja 1234<sub>16</sub> u 8-bitnoj memoriji:

### little-endian:



### big-endian:



## Proširenje memorijskih nar. - širina podatka

- Zbog pravilnosti, bilo bi dobro imati i memorijске naredbe koje rade s 16-bitnim podatcima, tj. s poluriječima (halfword)
- Zato **dodajemo naredbe LOADH i STOREH** (load halfword i store halfword) koje imaju jednake operative kao i LOAD/STORE te LOADB/STOREB
- Naredba LOADH čita dvobajtni podatak iz dvije uzastopne memorijске lokacije i stavlja ga u najniža dva bajta registra. Gornja dva bajta registra se pune ništicama
- Naredba STOREH upisuje dva najniža bajta iz registra u dvije uzastopne memorijске lokacije



## Proširenje memorijskih nar. - širina podatka

- U pristupanju bajtovima, riječima i poluriječima obično postoje ograničenja pa ih i mi uvodimo:
  - Riječi imaju adrese djeljive s 4
  - Poluriječi imaju adrese djeljive s 2
  - Bajtovi nemaju ograničenja na adresu
- U nekim procesorima će u slučaju zadavanja pogrešne adrese doći do tzv. iznimke. Budući da želimo imati jednostavan procesor, mi ćemo izbjegći ovakvu mogućnost na sljedeći način:
  - FRISC automatski stavlja ništice u dva najniža bita adrese prilikom izvođenja naredbe LOAD/STORE
  - FRISC automatski stavlja ništicu u najniži bit adrese prilikom izvođenja naredbe LOADH/STOREH
  - FRISC ne mijenja adresu prilikom izvođenja naredbe LOADB/STOREB



## Proširenje memorijskih naredaba - primjeri

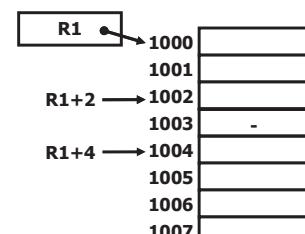
### ; registri za adresiranje:

```
MOVE 1000, R1 ; adresa prvog podatka
MOVE 1010, R3 ; adresa trećeg podatka
```

```
; kopiraj poluriječ iz 1. u 3. strukturu
LOADH R5, (R1+0) ; ili kraće pisanje (R1)
STOREH R5, (R3+0) ; ili kraće pisanje (R3)
```

```
; kopiraj bajt
; iz 3. u 1. strukturu
LOADB R5, (R3+2)
STOREB R5, (R1+2)
```

```
; kopiraj riječ
; iz 1. u 3. strukturu
LOAD R5, (R1+4)
STORE R5, (R3+4)
```



## Proširenje memorijskih naredaba - endianness

- S obzirom da su oba redoslijeda zapisa podjednako dobra, **odabiremo little-endian** kao redoslijed koji koristi naš procesor



Pojedini procesori mogu raditi s oba načina zapisa podataka pa se nazivaju bi-endian (PowerPC, ARM), a postoje i kombinirani redoslijedi (middle-endian), ali to ovdje nećemo razmatrati

## Proširenje memorijskih naredaba - primjeri

### Primjer rada s bajtovima u memoriji:

U memoriji se na adresama 1000 i 1001 nalaze 2 bajta 16-bitnog podatka zapisanih u big-endian redoslijedu. Podatak zapisati na lokacije 2000 i 2001, ali u little-endian redoslijedu.

**LOADB R0, (1000)**

**LOADB R1, (1001)**

**STOREB R0, (2001)**

**STOREB R1, (2000)**

U memoriji se lokacije 1000,1001,2000,2001 koriste ovako:

1000 11 ; viši bajt 16-bitnog broja 1122

1001 22 ; niži bajt 16-bitnog broja 1122

...

2000 0 ; mjesto za niži bajt, tj. za 22

2001 0 ; mjesto za viši bajt, tj. za 11

© Kovač, Basch, FER, Zagreb

281

## Proširenje memorijskih naredaba - stog



- Podsjetnik: stog se nalazi u memoriji, a u svakom trenutku moramo znati adresu vrha stoga
  - Trebamo pokazivač stoga SP u kojem će se pamtitи adresa vrha stoga
- Mogli bi uvesti posebni 32-bitni registar za tu namjenu
  - Tada bi morali imati i posebne naredbe koje bi mogle upisivati vrijednost u SP i čitati vrijednost registra SP
  - Time bi donekle zakomplicirali arhitekturu i proširili skup naredaba
- Zato ćemo "žrtvovati" jedan od općih registara i dodijeliti mu posebnu ulogu pokazivača stoga - to će biti R7:
 
  - R7 će se moći nazivati alternativnim imenom SP
  - Inače se R7 može ravnopravno koristiti u ostalim naredbama kao i preostali registri R0 do R6

© Kovač, Basch, FER, Zagreb

283

## Proširenje memorijskih naredaba - stog



- Budući da su oba pristupa jednako dobra, odabiremo da SP pokazuje na zadnji podatak na stogu
- Definiramo pisanje i operande naredaba PUSH i POP:
 
  - src - opći registar iz kojeg se uzima podatak koji se stavlja na stog
  - dest - opći registar u kojeg se stavlja podatak sa stoga

**PUSH src:**

**POP dest:**



$R7-4 \rightarrow R7$

$(R7) \rightarrow dest$

$src \rightarrow (R7)$

$R7+4 \rightarrow R7$

© Kovač, Basch, FER, Zagreb

285

## Proširenje memorijskih nar. - strojni kôd



- Strojni kôd će biti jednostavan:
  - operacijski kôd zadaje radi li se o naredbi PUSH ili POP
  - src/dest kodira registar opće namjene koji je izvor/odredište podatka



© Kovač, Basch, FER, Zagreb

287

## Proširenje memorijskih naredaba - stog



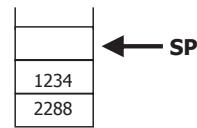
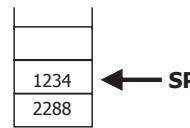
- Većina procesora koristi stog za spremanje podataka i povratnih adresa iz potprograma i prekidnih potprograma
- Već smo vidjeli što je stog i dvije osnovne operacije za stavljanje i uzimanje podataka sa stoga - PUSH i POP
- Da bi omogućili rad sa stogom, **uvest ćemo naredbe PUSH i POP** koje će:
  - stavljati na stog podatak iz jednog od općih registara (PUSH)
  - uzimati podatak sa stoga i stavljati ga u jedan od općih registara (POP)
- Budući da se stog nalazi u memoriji, PUSH i POP ćemo svrstati u skupinu memorijskih naredaba

© Kovač, Basch, FER, Zagreb

282

## Proširenje memorijskih naredaba - stog

- Već smo vidjeli redoslijed koraka u naredbama PUSH i POP kada **SP pokazuje na zadnji podatak na stogu** (lijeva slika):
  - PUSH: umanji SP, zatim stavi podatak
  - POP: uzmi podatak, zatim uvećaj SP
- Alternativni redoslijed koraka vrijeti kada **SP pokazuje na prvo slobodno mjesto na stogu** (desna slika):
  - PUSH: stavi podatak, zatim umanji SP
  - POP: uvećaj SP, zatim uzmi podatak



© Kovač, Basch, FER, Zagreb

284

## Proširenje memorijskih naredaba - stog



- Uočite da se R7 smanjuje i povećava za četiri, jer se na stog **uvijek stavlja i uzima 32-bitni podatak**
- Slično naredbama LOAD i STORE, da bi se izbjegla greška u naredbama PUSH i POP, automatski se stavljuju ništice na najniža dva bita adrese

Uočite da je moguće na stog staviti/uzeti i registar R7, ali to nije uobičajeno (niti previše korisno)

© Kovač, Basch, FER, Zagreb

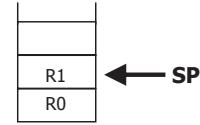
286

## Proširenje memorijskih naredaba - primjer

### Primjer rada sa stogom:

Zamjenjiviti sadržaje registara R0 i R1. Prepostavite da je SP već inicializiran i da pokazuje na područje u memoriji određeno za stog.

**PUSH R0**  
**PUSH R1**  
**POP R0**  
**POP R1**



Stog se može koristiti i za privremeno pohranjivanje stanja registra, kad nam za neko računanje treba dodatni registar kojemu ne smijemo promjeniti vrijednost. Alternativno rješenje smo već vidjeli - registar se može pohraniti na memorisku lokaciju koju prethodno rezerviramo u tu svrhu.

© Kovač, Basch, FER, Zagreb

288

## Proširenje memorijskih nar. - put podataka

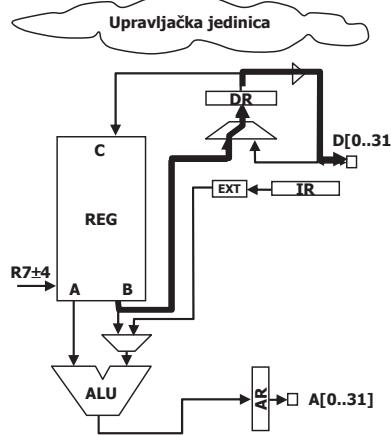
- Da bi se memorijске naredbe mogle izvoditi, morat će u putu podataka imati sljedeće spojne puteve:
  - iz općeg registra u DR - za naredbe STORE
  - iz DR u opći registar - za naredbe LOAD
  - iz ALU u AR - za adresiranje s brojem, ali i za adresiranje s registrom i odmakom

- Dodatno, za naredbe PUSH i POP moramo imati:
  - u skupu općih registara moramo dodati sklopove za povećavanje i smanjivanje registra R7 za 4

© Kovač, Basch, FER, Zagreb

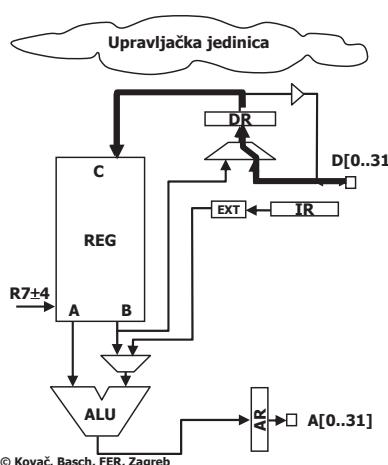
289

## Proširenje memorijskih nar. - put podataka



- Za STORE treba put od općih registara (izlaz B) do registra DR
- naravno, i put od DR do podatkovnih priključaka

## Proširenje memorijskih nar. - put podataka

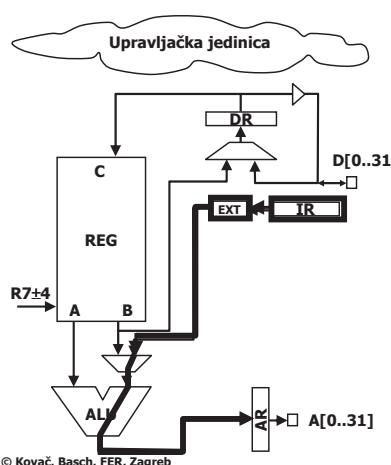


© Kovač, Basch, FER, Zagreb

291

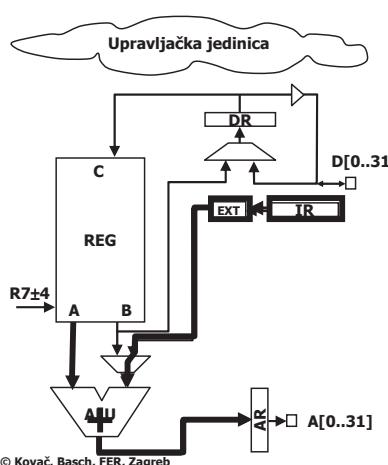
- Za LOAD treba put od DR do općih registara (ulaz C)
- naravno, i put od podatkovnih priključaka do DR

## Proširenje memorijskih nar. - put podataka



- Za obično adresiranje brojem u LOAD i STORE treba put iz sklopa za predznačeno proširivanje (EXT) do registra AR

## Proširenje memorijskih nar. - put podataka

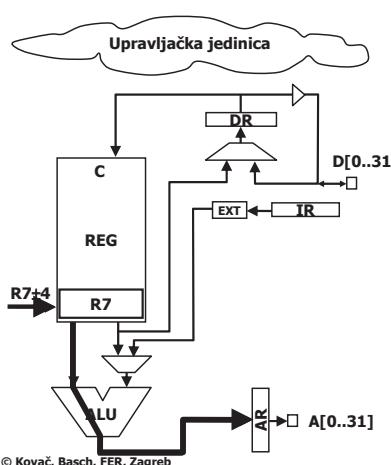


© Kovač, Basch, FER, Zagreb

293

- Za adresiranje pomoću adresnog registra i odmaka u naredbama LOAD i STORE, treba izračunati adresu i poslati je u AR:
  - Prvi operand (adresni registar) dovodi se na ulaz od ALU iz općih registara (izlaz A)
  - Drugi operand (odmak) dovodi se na ALU iz sklopa za predznačeno proširivanje (EXT)
  - ALU zbraja dijelove adrese koja se šalje u AR

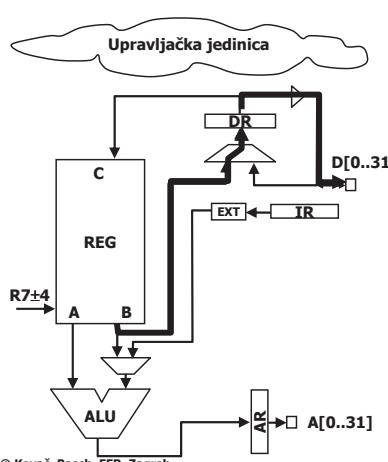
## Proširenje memorijskih nar. - put podataka



- U naredbama PUSH i POP za adresiranje se koristi R7

- R7 se povećava ili smanjuje za 4 na temelju upravljačkog signala R7±4
- R7 se dovodi u AR kroz ALU koja tada samo propušta svoj prvi operand na izlaz i ne izvodi nikavu operaciju

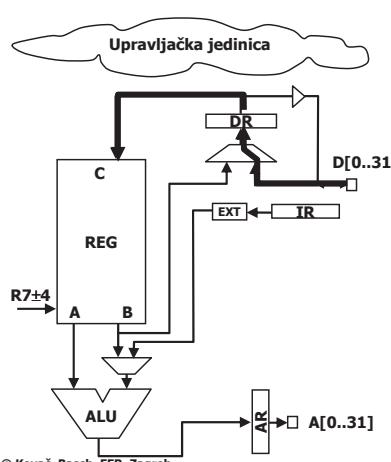
## Proširenje memorijskih nar. - put podataka



© Kovač, Basch, FER, Zagreb

295

- U naredbi PUSH se opći registar koji treba staviti na stog stavlja na izlaz B odakle se vodi na DR (kroz mimošteksor) i dalje na stog (tj. u memoriju)
  - Taj put koristi i STORE



- U naredbi POP se podatak sa stoga (tj. iz memorije) prvo dohvata u DR, a zatim se dovodi u opće registre preko ulaza C
  - Taj put koristi i LOAD

296



### Proširenje upravljačkih naredaba

© Kovač, Basch, FER, Zagreb

297

### Proširenje upravljačkih naredaba



- Da bi se moglo skočiti na bilo koju memorisku adresu, upotrijebit ćemo istu ideju kao kod naredba LOAD/STORE:
    - Jedan registar upotrijebit ćemo kao adresni registar koji će svojim 32-bitnim sadržajem zadati adresu skoka na bilo kojoj memoriskoj lokaciji u prostoru od 4 gigabajta
  - Trenutačno je strojni kôd građen ovako:
- |                 |     |       |       |                       |
|-----------------|-----|-------|-------|-----------------------|
| operacijski kôd | --- | cond  | ---   | 20-bitna adresa skoka |
| 31-27           | 26  | 25-22 | 21-20 | 19-0                  |
- Ponovno ćemo upotrijebiti bit 26 za zadavanje načina adresiranja ...

© Kovač, Basch, FER, Zagreb

298

### Proširenje upravljačkih nar. - strojni kôd



- U prvom obliku kodiramo: uvjet (cond) i 20-bitnu adresu skoka koja će predznačeno proširena dati konačnu 32-bitnu adresu odredišta skoka

| operacijski kôd | 1  | cond  | ---   | 20-bitna adresa |
|-----------------|----|-------|-------|-----------------|
| 31-27           | 26 | 25-22 | 21-20 | 19-0            |

- U drugom obliku kodiramo redom: uvjet (cond) i registar (adrreg) u kojem će biti adresa skoka
  - uočiti da ovdje nema odmaka kao u LOAD/STORE (nije potreban u naredbama skoka, a nema ni mesta u strojnem kôdu)

| operacijski kôd | 0  | cond  | ---   | adrreg | ---  |
|-----------------|----|-------|-------|--------|------|
| 31-27           | 26 | 25-22 | 21-20 | 19-17  | 16-0 |

© Kovač, Basch, FER, Zagreb

299

### Proširenje upravljačkih naredaba



- Nakon ovog proširenja možemo skočiti na bilo koju 32-bitnu adresu u memoriji

- Naredba JP piše se ovako:

```
JP adresa
JP (adrreg)
JP_uvjet adresa
JP_uvjet (adrreg)
```

- adresa je 20-bitni broj koji predznačeno proširen daje 32-bitnu adresu skoka
- adrreg je jedan od općih registara R0 do R7 čiji 32-bitni sadržaj zadaje 32-bitnu adresu skoka

© Kovač, Basch, FER, Zagreb

301

### Proširenje upravljačkih nar. - relativni skok

- Vidimo da način skakanja nije najpraktičniji, ali funkcioniра:
  - možemo s bilo kojeg mesta u memoriji skočiti na bilo koje drugo mjesto
- Nepraktično je:
  - za skok trebaju dvije naredbe i jedan slobodni registar
  - za skok treba dodatna memoriska lokacija u kojoj je adresa skoka (praktično je da je ova lokacija bude negdje unutar memorije koja se može adresirati s 20 bitova, jer inače i nju moramo dohvaćati indirektno preko nekog drugog registra)
- Pogledajmo kada je ovakav način skakanja naročito nepraktičan i koje je moguće rješenje...

© Kovač, Basch, FER, Zagreb

300

### Proširenje upravljačkih naredaba - primjeri

#### Primjer skoka na 32-bitnu adresu:

Treba skočiti na adresu 200000.

```
LOAD R0, (A_200000) ;;; Priprema adrese u R0
JP (R0) ;;; Skok "na R0"
...
;; Na adresi A_200000 nalazi se adresa skoka 200000
A_200000 DW 200000
...
;; Na adresi 200000 nalazi se
;; dio programa na koji skačemo...
200000 ADD R2, R3, R4 ; neka naredba
...
```

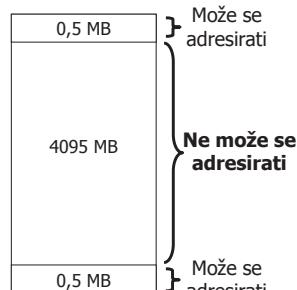
© Kovač, Basch, FER, Zagreb

302

### Proširenje upravljačkih nar. - relativni skok



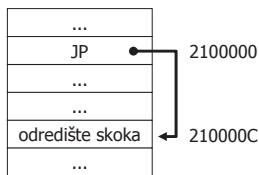
- Podsjetimo se koje adrese se mogu izravno adresirati sa 20 bitova koji se predznačeno proširuju:



- Četvrtina promila memorije se može izravno adresirati
  - $2^*0.5 \text{ MB} = 1 \text{ MB}$  je dovoljno za vrlo jednostavne aplikacije
  - Za "normalne" aplikacije treba više memorije
- Sa bilo kojeg mesta u memoriji (i u "sredini") može se skočiti na početak ili kraj memorije

## Proširenje upravljačkih nar. - relativni skok

- Iako korištenjem registra možemo skočiti bilo kuda, postoje slučajevi kad je to naročito nepraktično:

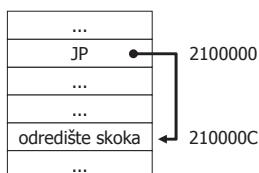


- Pretpostavimo da na procesor spojimo npr. 256MB memorije
- Pretpostavimo da na adresi 2100000 (u "sredini memorije") imamo neki dio programa u kojem se nalazi npr. petlja sa dvadesetak naredaba ili npr. naredba skoka kojom želimo preskočiti 2 naredbe (kao na slici)
- Ovakvi **kratki skokovi** su najčešći u programima

© Kovač, Basch, FER, Zagreb

305

## Proširenje upravljačkih nar. - relativni skok

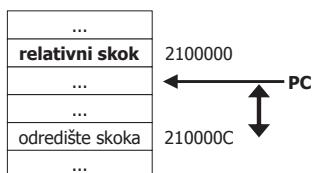


- Dodata nepraktičnost: ne možemo koristiti labelu za skok (labele povećavaju čitljivost)
- Također treba znati adresu na koju želimo skočiti (Znamo li je? Teorijski znamo, ali u praksi NE - zato što obično ne vodimo računa na kojim adresama se nalaze naredbe)
- Postoji li bolje rješenje?**

© Kovač, Basch, FER, Zagreb

307

## Proširenje upravljačkih nar. - relativni skok



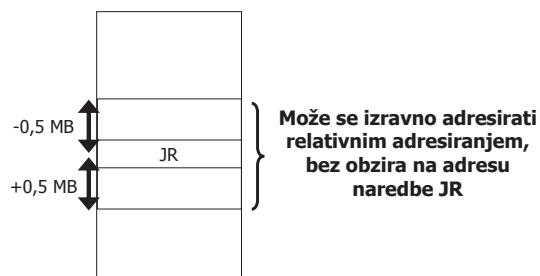
- U naredbi relativnog skoka treba zadati samo **udaljenost** odredišta skoka
- S obzirom da su skokovi većinom kratki, onda nam je npr. 20-bitna udaljenost i više nego dovoljna

© Kovač, Basch, FER, Zagreb

309

## Proširenje upravljačkih nar. - relativni skok

- Ovako definirana naredba daje mogućnost adresiranja 1 MB memorijskog prostora **u okolini trenutačne naredbe JR**:
- Moguć je skok unaprijed za otprilike 0,5 MB i unatrag za također 0,5 MB:



© Kovač, Basch, FER, Zagreb

311

## Proširenje upravljačkih nar. - relativni skok

- Budući da ne možemo zadati adresu sa 20-bitnim brojem, morali bi koristiti registar i dodatnu memorijsku lokaciju
- Budući da je program prepun ovakvih kratkih skokova, ovo bi bilo vrlo nepraktično i neefikasno
- ....

© Kovač, Basch, FER, Zagreb

306

## Proširenje upravljačkih nar. - relativni skok

- Boje rješenje je relativni skok**
- Relativni skok je takav skok kod kojeg znamo početni položaj naredbe skoka i "udaljenost" odredišta skoka
- Početni položaj zapisan je u registru PC (koji pokazuje jednu naredbu dalje od naredbe JP)

© Kovač, Basch, FER, Zagreb

308

## Proširenje upravljačkih nar. - relativni skok

- Uvodimo naredbu JR** (jump relative) kojoj se odredište skoka ne zadaje apsolutnom adresom, nego odmakom od same naredbe JR
- Odmak je 20-bitni i predznačeno se proširuje čime se dobivaju skokovi "unaprijed" i "unatrag"
- Programer ne mora izračunavati ovaj odmak (udaljenost, engl. offset), nego se za to brine asemblerski prevoditelj, a programer piše adresu skoka (obično kao labelu)
- Naredba može koristiti uvjet i piše se ovako:

JR adresa  
JR\_uvjet adresa

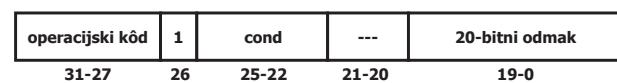


© Kovač, Basch, FER, Zagreb

310

## Proširenje upravljačkih nar. - strojni kôd

- Strojni kôd jednako je građen kao i za obični JP koji koristi 20-bitnu adresu. Razlika je što se umjesto 20-bitne adrese koristi 20-bitni odmak.
- Naredbe JP i JR se međusobno razlikuju po drugaćijem operacijskom kôdu



© Kovač, Basch, FER, Zagreb

© Kovač, Basch, FER, Zagreb

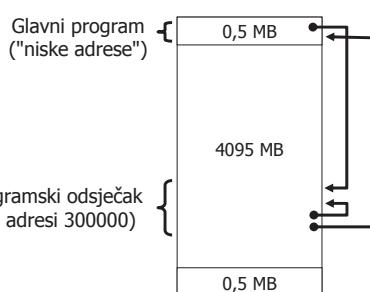
312

## Proširenje upravljačkih naredaba - primjer

### Primjer dugačkih i relativnih skokova:

Glavni program je smješten na "niskim adresama", a programski odsječak na "visokoj adresi" 300000.

Iz glavnog programa treba **skočiti na odsječak na adresu 300000**. Odsječak mora zbrojiti pet 32-bitnih podataka (**petlja**) počevši od adrese 1000 i staviti rezultat u R0. Nakon toga, odsječak **skače na labelu NASTAVI** koja se nalazi u glavnom programu.



© Kovač, Basch, FER, Zagreb

313

; Glavni program na nižim adresama memorije:

GLAVNI LOAD R1, (A\_ODSJECAK)  
JP (R1)

...

NASTAVI ... ; neke naredbe

A\_ODSJECAK DW 300000 ; adresa za skok na odsječak  
1000 DW 12F, 5B54C367, 23A9, 87DDB000, 33578

; Na adresi 300000 nalaze se naredbe odsječka:

300000 MOVE 5, R5 ; brojač za petlju  
MOVE 0, R0 ; suma  
MOVE 1000, R1 ; adresa podataka >>>

PETLJA LOAD R2, (R1)  
ADD R0, R2, R0  
ADD R1, 4, R1  
SUB R5, 1, R5  
JR\_NZ PETLJA

praktično je koristiti relativno adresiranje i naredbu JR, jer je skok kratak, a odredište skoka ima adresu širu od 20 bita

JP NASTAVI

© Kovač, Basch, FER, Zagreb

315

## Proširenje upravljačkih nar. - potprogrami

- S običnim skokovima i ostalim naredbama možemo isprogramirati svaki algoritam
- Međutim, da bi mogli programirati, u praksi nam treba mogućnost korištenja potprograma (bolja struktura i modularnost programa)
- Za potprograme nam trebaju posebne naredbe skoka:
  - naredba za poziv potprograma CALL**
  - naredba za povratak iz potprograma RET**
- Zbog pravilnosti i ove naredbe izvode se uvjetno



© Kovač, Basch, FER, Zagreb

317

## Proširenje upravljačkih nar. - potprogrami

- Naredba CALL **mora omogućiti povratak** na naredbu koja se nalazi neposredno iza nje, tj. mora spremiti povratnu adresu
- Kako naredba CALL "zna" adresu sljedeće naredbe?
  - Jednostavno: adresa se nalazi u registru PC
  - Podsjetnik: u PC-u se nalazi adresa sljedeće naredbe koju treba izvesti, a to je upravo naredba koja se nalazi neposredno iza naredbe CALL
- Nakon spremanja povratne adrese, CALL može skočiti u potprogram tako da u PC stavi adresu potprograma (adresa je zapisana unutar strojnog kôda ili unutar jednog od općih registara)

>>>

; Glavni program na nižim adresama memorije:

GLAVNI LOAD R1, (A\_ODSJECAK)

JP (R1) ←  
...

apsolutno adresiranje je nemoguće jer je adresa 300000 šira od 20 bitova, a relativno je nemoguće jer je skok predug

NASTAVI ... ; neke naredbe

A\_ODSJECAK DW 300000 ; adresa za skok na odsječak  
1000 DW 12F, 5B54C367, 23A9, 87DDB000, 33578

; Na adresi 300000 nalaze se naredbe odsječka:

300000 MOVE 5, R5 ; brojač za petlju  
MOVE 0, R0 ; suma  
MOVE 1000, R1 ; adresa podataka >>>

PETLJA LOAD R2, (R1)  
ADD R0, R2, R0  
ADD R1, 4, R1  
SUB R5, 1, R5  
JR\_NZ PETLJA

JP NASTAVI

© Kovač, Basch, FER, Zagreb

314

; Glavni program na nižim adresama memorije:

GLAVNI LOAD R1, (A\_ODSJECAK)  
JP (R1)

...

NASTAVI ... ; neke naredbe

A\_ODSJECAK DW 300000 ; adresa za skok na odsječak  
1000 DW 12F, 5B54C367, 23A9, 87DDB000, 33578

; Na adresi 300000 nalaze se naredbe odsječka:

300000 MOVE 5, R5 ; brojač za petlju  
MOVE 0, R0 ; suma  
MOVE 1000, R1 ; adresa podataka

PETLJA LOAD R2, (R1)  
ADD R0, R2, R0  
ADD R1, 4, R1  
SUB R5, 1, R5  
JR\_NZ PETLJA

JP NASTAVI

JR se ne može koristiti zbog prevelike udaljenosti odredišta skoka, a apsolutna adresa se može koristiti jer je odredišna adresa malena

© Kovač, Basch, FER, Zagreb

316

## Proširenje upravljačkih nar. - potprogrami

- Naredba CALL** se uvijek piše s adresom potprograma koji pozivamo (kao što se i naredba skoka JP piše s adresom na koju treba skočiti). Ova adresa se u praksi obično piše kao labela, koja se koristi kao ime potprograma. Definiramo pisanje naredbe CALL:

CALL adr  
CALL (adrreg)  
CALL\_uvjet adr  
CALL\_uvjet (adrreg)

- adr - 20-bitna adresa skoka koja se predznačno proširuje
- adrreg - opći register u kojem je adresa skoka

- Koriste se isti načini zadavanja adrese skoka kao u naredbi JP

© Kovač, Basch, FER, Zagreb

318

## Proširenje upravljačkih nar. - potprogrami

<<<

- Gdje se spremi povratna adresa? Na stog\***

- Dakle, naredba "CALL ADRESA" radi sljedeće:

- spremi PC na stog
- skoči na adresu potprograma (tj. na ADRESA)

- Ovako to radi "interno":

- SP → SP → PC → (SP) } PC → stog
- PC → (SP) } → ADRESA → PC

\* To nije jedina mogućnost spremanja povratne adrese, ali je najčešće korištena



- Strojni kôd naredbe CALL jednako je građen kao i strojni kôd naredbe JP (razlikuju se po operacijskom kôdu):

| operacijski kôd | 1  | cond  | ---   | 20-bitna adresa |      |
|-----------------|----|-------|-------|-----------------|------|
| 31-27           | 26 | 25-22 | 21-20 | 19-0            |      |
| operacijski kôd | 0  | cond  | ---   | adrreg          | ---  |
| 31-27           | 26 | 25-22 | 21-20 | 19-17           | 16-0 |

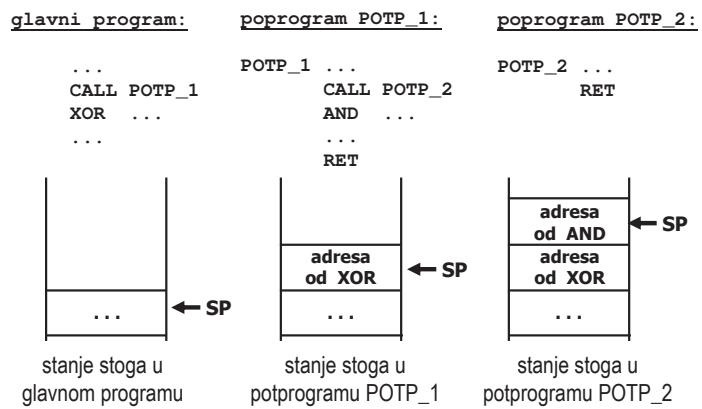


<<<

- Dakle, naredba "RET" radi sljedeće:
  - uzme povratnu adresu sa stoga i skoči na nju
- Ovako to radi "interno":
  - $(SP) \rightarrow PC$
  - $SP + 4 \rightarrow SP$



- Izgled stoga kod ugniježđenih poziva:



- Strojni kôd naredaba RET/RETI/RETN je jednostavan jer nema adresu:

| operacijski kôd | --- | cond  | ---  | i/n | int |
|-----------------|-----|-------|------|-----|-----|
| 31-27           | 26  | 25-22 | 21-2 | 1   | 0   |

- bitovi int i i/n određuju radi li se o naredbi RET, RETI ili RETN:
  - ako je int=0, onda se radi o naredbi RET
  - ako je int=1, onda bit i/n znači:
    - ako je i/n=0, onda se radi o naredbi RETI
    - ako je i/n=1, onda se radi o naredbi RETN

- Od ostalih upravljačkih naredba, ove naredbe razlikuju se po operacijskom kôdu



- **Naredba za povratak iz potprograma RET** piše se bez operanada (to je jedina naredba skoka u kojoj se ne zadaje adresa odredišta skoka)
- Naredbi RET adresa skoka nije potrebna, jer ona podrazumijeva da se adresa skoka nalazi na vrhu stoga
- Naravno, pretpostavka je da je negdje ranije izведен CALL koji je povratnu adresu stavio na stog, jer naredba RET ne može "znati" što je zaista na stogu
- Definiramo pisanje naredbe RET:

```
RET
RET_uvjet
```

- Zašto se povratna adresa spremi na stog?
- Zato jer je stog promjenjive veličine, a to omogućava jednostavno grijanje potprograma do potrebne dubine, kao i rekursivno pozivanje potprograma
- Spremanje na fiksnu memorijsku lokaciju (npr. u varijablu) ne bi bilo praktično, jer bi već drugi ugniježđeni poziv potprograma uništio prvu spremljenu povratnu adresu
- Kod spremanja na stog, novi podatak uvijek se spremi na novo mjesto, ovisno o trenutačnoj popunjenoosti stoga

- Definirajmo još dvije varijante naredbe RET
- One će nam trebati kod prekidnog U-I prijenosa, pa ih nećemo sada objašnjavati
- Definirat ćemo samo način njihovog pisanja:

```
RETI
RETI_uvjet

RETN
RETN_uvjet
```

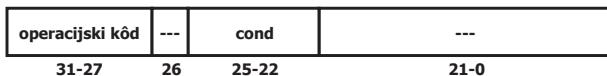
- Na kraju, dodajmo još jednu naredbu koja postoji u mnogim procesorima
- To je naredba za **zaustavljanje rada procesora** i naziva se **HALT**
  - Budući da ova naredba prekida normalni slijed izvođenja, svrstat ćemo je u upravljačke (iako nije naredba skoka)
  - Zbog pravilnosti će se i naredba HALT izvoditi uvjetno
- Naredba HALT piše se ovako:

```
HALT_uvjet
HALT
```



## Proširenje upravljačkih nar. - strojni kôd

- Strojni kôd naredbe HALT izgleda ovako:

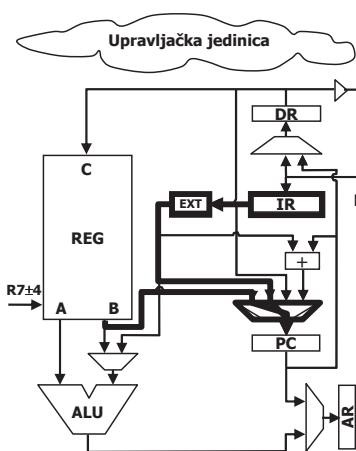


- Od ostalih upravljačkih naredba, HALT se razlikuje po operacijskom kôdu

© Kovač, Basch, FER, Zagreb

329

## Proširenje upravljačkih nar. - put podataka

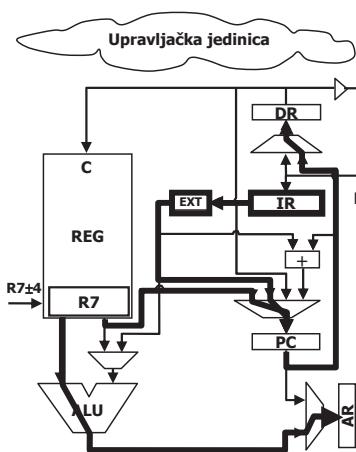


- Za naredbu JP s običnim adresiranjem uzima se izlaz iz sklopa za predznačeno proširenje (EXT) i stavlja se u PC
- Za naredbu JP, kod koje je adresa zadana registrom, uzima se izlaz B iz skupa općih registara i vodi se u PC
- Oba prethodna spojna puta ne dovode se u PC izravno, nego kroz multipleksor

© Kovač, Basch, FER, Zagreb

331

## Proširenje upravljačkih nar. - put podataka

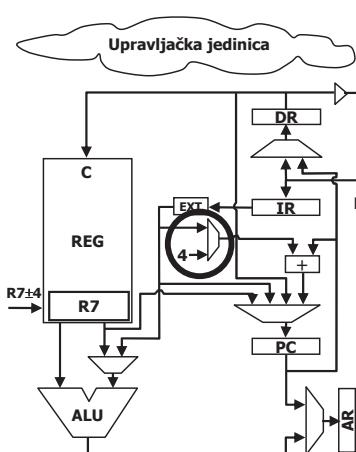


- Za naredbu CALL povratnu adresu iz PC-a treba spremiti na stog pa zato postoji spojni put od PC-a do registra DR
- R7 se dovodi kroz ALU do registra AR (isto kao u naredbi PUSH)
- Adresa skoka se u PC upisuje kao i u naredbi JP:
  - na izlazu od EXT nalazi se adresa koju treba staviti u PC ili se izlaz B iz općih registara dovodi u PC

© Kovač, Basch, FER, Zagreb

333

## Proširenje upravljačkih nar. - put podataka



- Nevezano za upravljačke naredbe, ali vezano uz PC:
  - PC se nakon dohvata strojnog kôda treba povećati za 4 (tako da pokaze na sljedeću naredbu)
  - na ovoj slici je dodan multiplekser koji to omogućuje (odabirom konstante 4 koja se dovodi u PC-jevo zbrajalo)

© Kovač, Basch, FER, Zagreb

335

## Proširenje upravljačkih nar. - put podataka

- Za izvođenje upravljačkih naredaba, put podataka je komplikiraniji nego kod ostalih naredaba

- Sve naredbe skoka izvode se tako da se željena adresa skoka treba staviti u registar PC

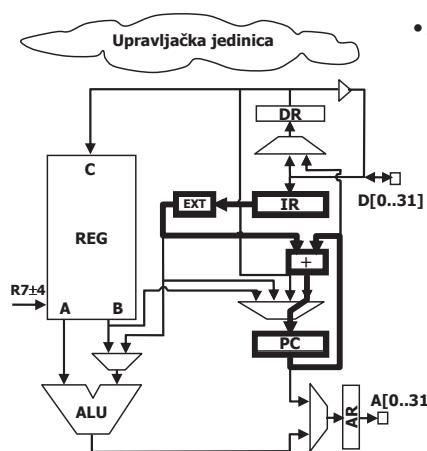
- Pri normalnom slijednom izvođenju, PC se nakon dohvata svake naredbe povećava za 4 (o tome više nešto kasnije)

- Dodatnu komplikaciju kod upravljačkih naredaba unose naredbe CALL i RET koje prebacuju povratnu adresu iz PC u memoriju (na stog) i obrnuto

© Kovač, Basch, FER, Zagreb

330

## Proširenje upravljačkih nar. - put podataka

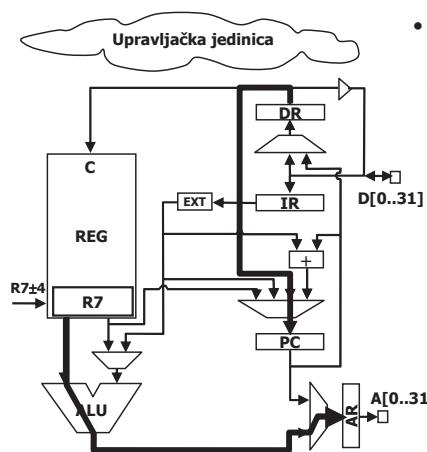


- Za naredbu JR uzima se izlaz iz sklopa za predznačeno proširenje (EXT)
  - na izlazu od EXT nalazi se odmak koji treba zbrojiti s prethodnim stanjem PC-a
  - zato se ovaj izlaz dovodi na posebno zbrajalo u koje ulazi i prethodno stanje PC-a
  - izlaz zbrajala vodi se kroz multipleksor u PC

© Kovač, Basch, FER, Zagreb

332

## Proširenje upravljačkih nar. - put podataka



- Za naredbu RET, u PC treba upisati povratnu adresu sa stoga:
  - iz registra DR vodi spojni put do multipleksora i dalje u PC
  - R7 se vodi kroz ALU do registra AR (isto kao u naredbi POP i ostalim naredbama koje rade sa stogom)

© Kovač, Basch, FER, Zagreb

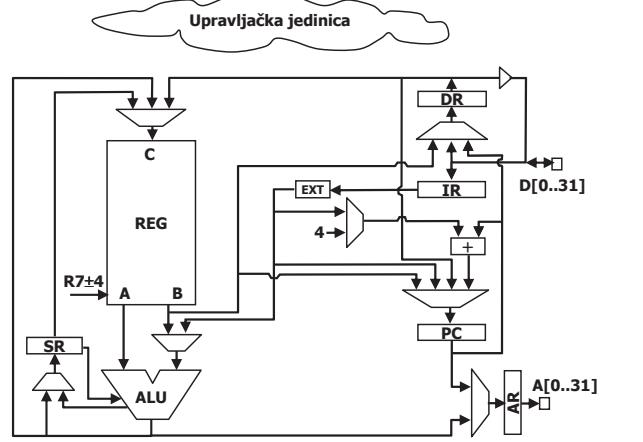
334

## Put podataka procesora FRISC

- Ovime smo kompletirali naredbeni skup procesora FRISC
- Za svaku vrstu naredaba vidjeli smo dijelove puta podataka potrebne za izvođenje tih naredaba
- Na kraju, kao rekapitulacija, sljedeći slajdovi prikazuju cjelokupni put podataka u kojem su objedinjene sve prethodne djelomične slike puta podataka..., a nakon toga je dat popis svih naredaba FRISC-a

© Kovač, Basch, FER, Zagreb

336



© Kovač, Basch, FER, Zagreb

337

## Popis naredaba procesora FRISC

<<<

**Memorijske naredbe:** LOAD/LOADB/LOADH i STORE/STOREB/STOREH

**Sintaksa:** LOAD dest, (adresa) // ili LOADB ili LOADH  
STORE src, (adresa) // ili STOREB ili STOREH  
src, dest - jedan od općih registara R0...R7  
adresa - 20-bitna adresa koju se predznačno proširuje pri izvođenju  
- Ri+odmak, gdje je Ri jedan od općih registara, a odmak je 20-bitni broj koji se predznačno proširuje pri izvođenju

**Primjeri:** LOAD R0, (100) STORE R2, (R5+12)  
LOADB R2,(R5+0) STOREH R1,(1200)

>>>

© Kovač, Basch, FER, Zagreb

339

## Popis naredaba procesora FRISC

<<<

**Upravljačke naredbe:** JR

**Sintaksa:** JR\_uvjet adresa  
uvjet - jedan od sufiksa: C, NC, V, NV, Z, NZ, N, NN,  
ULE, UGT,ULT, UGE, SLE, SGT, SLT, SGE,  
EQ, NE, M, P, (sufiks se može ispuštiti)  
adresa - adresa na koju se skače, pri prevođenju se pretvara u  
20-bitni odmak koji se tijekom izvođenja predznačno proširuje  
i dodaje registru PC

**Primjeri:** JR 1000 JR\_NATRAG  
JR\_UGE 100 JR\_NZ LOOP

© Kovač, Basch, FER, Zagreb

341

## Načini adresiranja



- Nakon definiranja strojnih kôdova i proširivanja imamo sljedeće naredbe:

**Aritmetičko-logičke naredbe:** ADD, SUB, ADC, SBC, CMP, AND, OR, XOR, SHL, SHR, ASHR, ROTL, ROTR

**Sintaksa:** ALU\_NAREDBA src1, src2, dest

src1, src2 i dest - jedan od općih registara R0...R7

src2 - dodatno može biti 20-bitni broj koji se predznačno proširuje pri izvođenju

**Primjeri:** ADD R0, R1, R2  
AND R1, 00FF, R2  
SUB R3, 10, R3  
XOR R2, R3, R4

>>>

© Kovač, Basch, FER, Zagreb

338

## Popis naredaba procesora FRISC

<<<

**Upravljačke naredbe:** JP, CALL

**Sintaksa:** JP\_uvjet adresa  
CALL\_uvjet adresa  
uvjet - jedan od sufiksa: C, NC, V, NV, Z, NZ, N, NN,  
ULE, UGT,ULT, UGE, SLE, SGT, SLT, SGE,  
EQ, NE, M, P, (sufiks se može ispuštiti)  
adresa - 20-bitna adresa koja se predznačno proširuje pri izvođenju  
- (R), gdje je Ri jedan od općih registara

**Primjeri:** JP 1000 JP (R2)  
JP\_UGE 100 CALL\_NZ 300  
CALL 1000 CALL (R4) >>>

340

## Popis naredaba procesora FRISC

<<<

**Upravljačke naredbe:** RET, RETI, RETN, HALT

**Sintaksa:** RET\_uvjet  
RETI\_uvjet  
RETN\_uvjet  
HALT\_uvjet  
uvjet - jedan od sufiksa: C, NC, V, NV, Z, NZ, N, NN,  
ULE, UGT,ULT, UGE, SLE, SGT, SLT, SGE,  
EQ, NE, M, P, (sufiks se može ispuštiti)

**Primjeri:** RET RETI  
RETN RETI\_NZ  
RETN\_N RETI\_UGE  
HALT HALT\_C

342

## Načini adresiranja

- Do sada smo već spominjali načine adresiranja i vidjeli ih u pojedinim naredbama. Pogledajmo sada i definirajmo načine adresiranja procesora FRISC.
- Pod načinima adresiranja (addressing modes) u širem smislu misli se na načine zadavanja operanada, rezultata ili odredišta skoka u naredbama
- U užem smislu, adresiranje se odnosi samo na načine adresiranja podataka ili odredišta skoka u memoriji, ali ovdje ćemo koristiti taj pojam u širem smislu
- Ovdje se radi o **procesorskim načinima adresiranja**

- Treba razlikovati procesorska adresiranja od asemblerских adresiranja
- Procesorska adresiranja:
  - odnose se na različite načine na koje procesor adresira podatke prilikom izvođenja naredaba
  - međusobno se raspoznavaju po strojnom kôdu naredbe
- Asemblerска adresiranja (detaljniji opis kasnije):
  - različiti načini pisanja adresa koje dozvoljava asemblerski prevoditelj
  - svako od asemblerских adresiranja će asemblerski prevoditelj prevesti u jedno od procesorskih adresiranja
  - različita asemblerска adresiranja daju isti strojni kôd



### Neposredno adresiranje (immediate addressing)

- Podatak se zadaje neposredno u naredbi kao broj. Nakon prevođenja, taj podatak je zapisan u strojnom kôdu.
- Podatak predstavlja broj s kojim se izvodi operacija, a ne adresu u memoriji.
- Primjeri:
 

```
ADD R0, %D 12, R2
MOVE 200, R3
ROTL R2, ZA_TRI, R2
```
- Napomene:
  - u jednoj naredbi svaki operand može imati svoje adresiranje
  - umjesto broja može se pisati i labela (ZA\_TRI)



### Relativno adresiranje (relative addressing)

- Koristi se samo u naredbi relativnog skoka JR
- Najčešće se piše kao labela (iako može i kao adresa zadana brojem) i po tome sliči asolutnom adresiranju
- Adresa/labela napisana u programu predstavlja adresu odredišta skoka, a asemblerski prevodilac izračunava **odmak** od same naredbe JR do zadane adrese
- **Odmak** se zapisuje kao običan broj u strojnom kôdu, a bitna razlika je da taj broj nije adresa (kao u asolutnom adresiranju) nego odmak do odredišne adrese. Ovaj se odmak tijekom izvođenja pribraja registru PC
- Primjeri:
 

```
JR 1200
JR PETLJA
```



### Registarsko indirektno adresiranje s odmakom (register indirect addressing with displacement; base plus offset addressing)

- Slično registarskom indirektnom adresiranju, ali se koristi u memorijskim naredbama
- Razlika je da se uz registar zadaje i broj koji predstavlja odmak (može biti pozitivan ili negativan).
- Adresa podatka u memoriji dobiva se zbrajanjem registra i odmaka
- Primjeri:
 

```
LOAD R5, (R0+4)
STORE R5, (R6-12)
STORE R5, (R6)
```



- Iako je naš procesor RISC arhitekture, ipak ima relativno velik broj adresiranja. To su:

### Registarsko adresiranje (register addressing)

- Podatak ili rezultat nalazi se u jednom od registara procesora
- Ovakvo adresiranje imaju praktično svi procesori
- Primjeri (registarsko adresiranje je podcrtnato):
 

```
ADD R0, 12, R2
MOVE SR, R3
LOAD R6, (1000)
PUSH R3
```



### Apsolutno adresiranje (absolute addressing)

- Kao kod neposrednog adresiranja, zadaje se broj u naredbi
- Bitna razlika je da taj broj predstavlja adresu podatka u memoriji, ili adresu odredišta skoka
- Primjeri:
 

```
LOAD R0, (1200)
JP 2000
CALL POTPROG
```
- Napomene:
  - u memorijskom naredbama adresa se piše u zagradama, a u upravljačkim naredbama adresa se piše bez zagrade
  - umjesto broja može se pisati i labela (POTPROG)



### Registarsko indirektno adresiranje (register indirect addressing)

- Zadaje se registrom zapisanim u zagradama. Koristi se u upravljačkim naredbama
- U registru se nalazi adresa odredišta skoka. Zgrade naglašavaju da se ne radi o običnom registarskom adresiranju.
- Primjeri:
 

```
JP_CC (R0)
CALL (R6)
```



### Registarsko indirektno adresiranje s odmakom (nastavak)

- Napomene:
  - U zadnjem primjeru "STORE R5, (R6)", ne radi se o registarskom indirektnom adresiranju iako nema odmaka.
  - Memoriske naredbe LOAD/STORE uvijek koriste registarsko indirektno adresiranje s odmakom.
  - Odmak je u ovoj naredbi jednak ništici pa ga se mora pisati, ali u strojnom kôdu ništica je zapisana kao odmak. Pravi zapis naredbe bi bio "STORE R5, (R6+0)".
  - Odmak se često naziva i pomakom, ali tako se naziva i operacija pomicanja bitova (shift) pa ćemo radite koristiti naziv odmak koji je bliži engleskom izvorniku *offset* ili *displacement*



## • Implicitno adresiranje (implied addressing)

- Specifično je po tome što se u naredbi **ne piše** položaj podatka ili odredišta skoka i sl.
- Umjesto eksplisitnog pisanja, iz same naredbe se zna gdje se nalazi podatak ili odredište skoka i sl.
- U FRISC-u implicitno adresiranje koriste naredbe PUSH i POP u kojima se podrazumijeva da se adresa podatka nalazi u registru SP (tj. da se podatak nalazi na stogu)
- Primjeri:

PUSH R5  
POP R6

## Asembleri - Uvod



- Asembleri prevoditelji, ili kraće asembleri, su programi koji prevode programe pisane u mnemoničkoj datoteci u strojni kôd određenog procesora (postupak prevođenja često nazivamo asembliranje)
- Mnemonički jezik je jezik niske razine i prilagođen je pojedinom procesoru
- Svaki strojni kôd ima odgovarajući mnemonik s kojim je u odnosu "jedan na jedan"
- Asembleri su prevoditelji, ali bitno jednostavniji od prevoditelja za više programske jezike. Asembleri se koriste internu prilikom prevođenja više programskih jezika, što je za korisnika obično nevidljivo

## Asembleri - Uvod

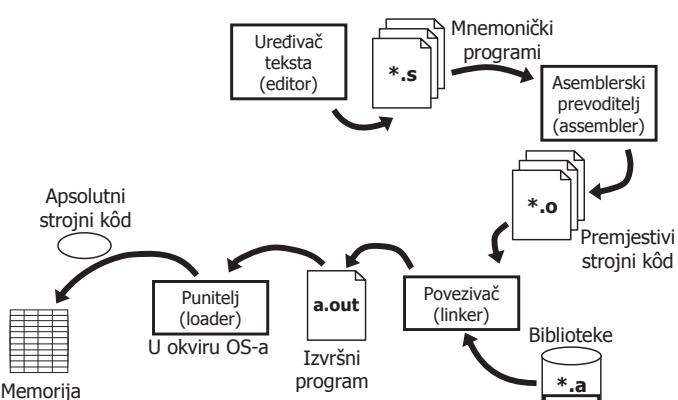


- Možemo spomenuti da podjela posla s prethodne slike može biti i drugačija:
  - Npr. povezivanje sa statičkim bibliotekama može obavljati povezivač, a povezivanje sa dinamičkim bibliotekama može obavljati punitelj
  - Npr. punjenje i povezivanje su zadaće koje može obavljati jedan program.
  - Npr. mnemonički program se obično stvara samo kao privremena datoteka koja se odmah dalje prevodi asembleriskim prevoditeljem, a ne kao datoteka koja će ostati zapisana na disku
  - Npr. izvršni program može biti u absolutnom ili premjestivom obliku

## Asembleri - Uvod



- Tipičan tijek pri prevođenju mnemoničkih programa:



## Arhitektura računala 1

### Asembleri prevoditelji (asembleri)

prof.dr.sc. Mario Kovač, prof.dr.sc. Danko Basch

Ovaj dokument namijenjen je isključivo za osobnu upotrebu studentima Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

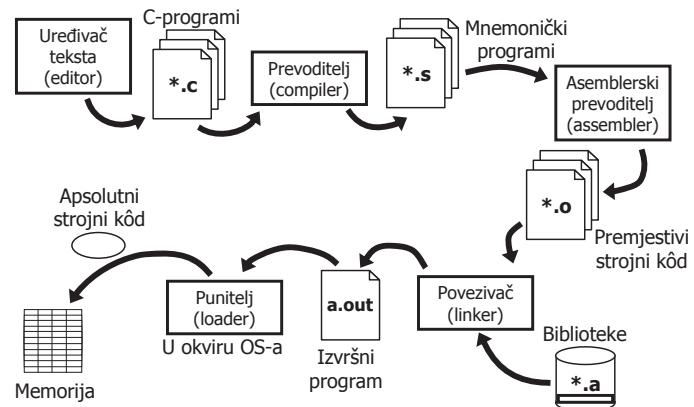
Skvaka druga upotreba ili umnožavanje bilo kojeg dijela ovog dokumenta nije dozvoljena bez pismene dozvole autora.

© Mario Kovač, Danko Basch



## Asembleri - Uvod

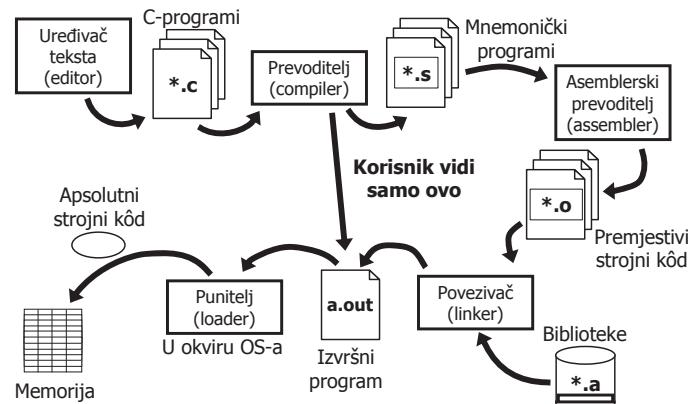
- Tipičan tijek pri prevođenju viših programskih jezika (UNIX):



## Asembleri - Uvod



- Za korisnika je većina ovog nevidljiva:



## Asembleri - Uvod



- Razdvajanje mnemoničkog programa u više datoteka omogućuje njegovu modularizaciju u cilju lakšeg programiranja i održavanja
- Prevođenjem ovih datoteka dobivaju se zasebne datoteke s premjestivim (relokabilnim) strojnim kôdom
  - Prednost premjestivog strojnog kôda je da može biti napunjeno u memoriju od bilo koje memorije adrese
  - Tako je moguće kombinirati veći broj premjestivih datoteka u konačni program, bez da jedna drugoj ograničavaju smještaj u memoriji
- Ako postoji više datoteka koje moraju zajedno sačinjavati program, one se moraju međusobno povezati, kako bi na primjer:
  - naredbe napisane u jednoj datoteci mogu pristupati podatcima definiranim u drugoj datoteci
  - naredbe napisane u jednoj datoteci mogu pozvati potprogram iz druge datoteke

## Asemblери - Uvod

- Nakon povezivanja može se dobiti program koji je još uvek premjestiv ili je u absolutnom obliku
  - Program u absolutnom obliku ima određene sve adrese podataka, potprograma itd. i spreman je za izravno punjenje u memoriju računala i izvođenje
  - Programu koji je u premjestivom obliku mora se prilikom punjenja odrediti početna adresa i na temelju toga preračunati sve adrese koje se u programu koriste.
- Nakon punjenja u memoriju računala, program se pokreće
  - Punjenje se tipično odvija pod upravljanjem operacijskog sustava (OS-a)
  - Korisnik zadaje punjenje pomoću ljske (npr. tcsh/bash na UNIX-u) ili grafičkog sučelja (Microsoft Windows, X Window System na UNIX-u)
  - Punjenje se ne zadaje izravno, nego se podrazumijeva kad pokrenemo neki program

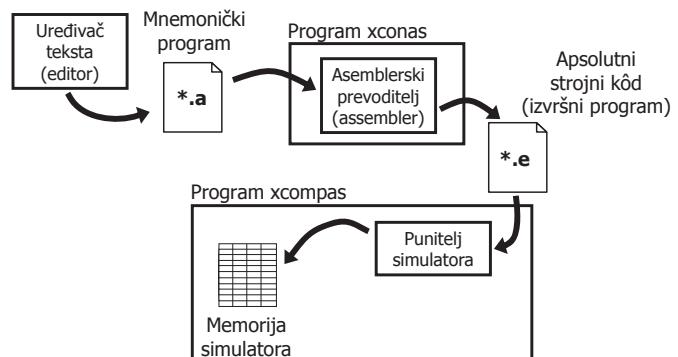
© Kovač, Basch, FER, Zagreb

8



## Asemblери - Uvod

- Prevođenje mnemoničkih programa u ATLAS-u:



© Kovač, Basch, FER, Zagreb

9

## Asemblери - Uvod

- U ATLAS-u se može koristiti samo jedna datoteka s mnemoničkim programom
- Ona se odmah prevodi u izvršnu datoteku u absolutnom obliku, tj. sadrži strojni kôd koji ima zadanu adresu punjenja u memoriju
- ATLAS je simulator računala na niskoj razini i u njemu ne postoji operacijski sustav - njegove najosnovnije uloge preuzima korisničko sučelje simulatora u kojem se programi mogu puniti i izvoditi

© Kovač, Basch, FER, Zagreb

10



## Asemblери - Mnemonički jezik



- Svaki proizvođač procesora propisuje simbolička imena (mnemonike) za naredbe svog procesora
- Asemblerski prevoditelji obično dodaju svoja pravila pisanja, ograničenja ili dopunske mogućnosti
- Ipak, postoje neka opća pravila pisanja mnemoničkih programa
- Ovdje ćemo koristiti pravila za program CONAS (CONfigurable ASsembler), koji je asemblerski prevoditelj programskega sustava ATLAS

© Kovač, Basch, FER, Zagreb

12

11

## Asemblери - Pravila pisanja



POLJE\_LABELE POLJE\_NAREDBE POLJE\_KOMENTARA

- Polja imaju sljedeća značenja i pravila pisanja:
- Polje labele:
  - Obavezno počinje od prvog stupca datoteke, ali se smije ispuštiti
  - Labela je simboličko ime za adresu
  - Labela se sastoji od niza slova i znamenaka te znaka podvlake, a prvi znak mora biti slovo
  - Duljina labele nije ograničena, ali se razlikuje samo prvih deset znakova

© Kovač, Basch, FER, Zagreb

14

© Kovač, Basch, FER, Zagreb

## Asemblери - Mnemonički jezik



- Od glavnog interesa će nam biti programiranje procesora u mnemoničkom ili asemblerskom jeziku\* (assembly language)
- Mnemonički jezik ovisi o procesoru za kojega je namijenjen, za razliku od viših programskih jezika koji ne ovise o računalu i/ili operacijskom sustavu na kojem će se izvoditi
- Datoteke u mnemoničkom jeziku su obične tekstovne datoteke pisane prema pravilima pojedinog procesora i asemblerskog prevoditelja\*

\* Napomena: i asemblerski jezik i asemblerski prevoditelj često se nazivaju skraćeno *asembler*

© Kovač, Basch, FER, Zagreb

## Asemblери - Pravila pisanja

- Mnemoničke datoteke nemaju slobodan format pisanja kao viši programski jezici, nego su **retkovno orijentirane**:
  - Naredba se **ne može** protezati kroz više redaka
  - U jednom retku može biti **najviše jedna** naredba
  - Smije se pisati prazan redak (zbog bolje čitljivosti)
- Svaki redak sastoji se od sljedećih polja:

POLJE\_LABELE POLJE\_NAREDBE POLJE\_KOMENTARA

Za razliku od C-a:

```
a = 2+i; a-= func(3*j); a += b[2][4];
- func(3*j)
+ b[2][4];
```

© Kovač, Basch, FER, Zagreb

13

## Asemblери - Pravila pisanja



POLJE\_LABELE POLJE\_NAREDBE POLJE\_KOMENTARA

- Polja imaju sljedeća značenja i pravila pisanja:
- Polje naredbe:
  - Polje naredbe ispred sebe obavezno mora imati prazninu (znak razmaka ili tabulatora), bez obzira stoji li ispred labela ili ne
  - Polje naredbe se smije ispuštiti (tada naravno nije potrebno stavljati praznine)
  - Naredba se piše prema pravilima definiranim za pojedini procesor
  - U polju naredbe umjesto naredbe smije stajati i pseudonaredba (bit će objašnjene kasnije)

15

## Asemblieri - Pravila pisanja

POLJE\_LABELE POLJE\_NAREDBE POLJE\_KOMENTARA

- Polja imaju sljedeća značenja i pravila pisanja:
- Polje komentara:
  - Polje komentara počinje znakom komentara i proteže se do kraja tekućeg retka
  - Znak komentara ovisi o procesoru
    - za FRISC to je znak točka-zarez ;
  - Polje komentara se također može ispuštiti
  - Polje komentara se zanemaruje prilikom prevođenja

© Kovač, Basch, FER, Zagreb

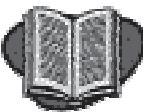
16

## Asemblieri - Vrste asemblera

- Asembleri se mogu podijeliti po broju prolaza na:
  - jednoproplazne ili apsolutne asemblere
  - dvoprolazne ili simboličke asemblere
  - troprolazne asemblere } tzv. makroasemblieri
  - četveroprolazne asemblere }
- Ovisno o broju prolaza, asembleri imaju različite mogućnosti - što više prolaza, to više mogućnosti



Objašnjenje načina prevođenja proučite za domaću zadaću (pogledajte datoteku nazvanu "03 Način asembliranja DZ")



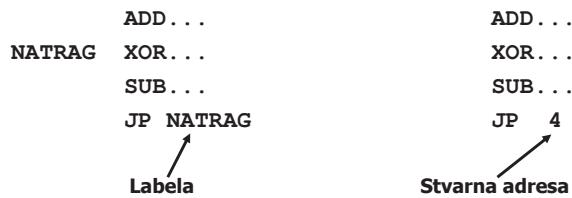
© Kovač, Basch, FER, Zagreb

18

## Asemblieri - Labele



- U asembljeru se **labele** također koriste kao **odredište skoka**
  - Za razliku od viših programskih jezika, u asembljeru je korištenje labela i naredbe skoka (npr. JUMP) jedini način za upravljanje tokom programa
  - Kao odredište skoka može se pomoći brojem zadati i stvarna adresa skoka, ali tada moramo **točno znati na koju adresu želimo skočiti**, tj. moramo tu adresu "ručno izračunati".



© Kovač, Basch, FER, Zagreb

20

## Asemblieri - Labele



- Labele su simbolički nazivi za adrese, a glavne prednosti su:
  - jednostavnije i brže programiranje
  - bolja čitljivost i lakše održavanje programa
  - izračunavanje adresa obavlja asemblerski prevoditelj što ujedno smanjuje mogućnost pogreške
- Asembler "izračunava" stvarne vrijednosti labela (tj. adrese) točno onako kako ih i mi "ručno izračunavamo"
- Korištenje labela naziva se **simboličko adresiranje**, a korištenje stvarnih adresa zadanih brojem naziva se **apsolutno adresiranje**
- **POZOR:** ovo su **asemblierska adresiranja** i ne treba ih miješati s **procesorskim adresiranjima**, naročito ne s istoimenim apsolutnim procesorskim adresiranjem



© Kovač, Basch, FER, Zagreb

22

## Asemblieri - Pravila pisanja

POLJE\_LABELE POLJE\_NAREDBE POLJE\_KOMENTARA

- Primjeri:

```

PETLJA ADD R0, R1, R2 ;naredba ADD
 SUB R3, R2, R3
PODATCI `ORG 200 ;pseudonaredba `ORG
LABELA_3 ; labela smije stajati bez naredbe
; komentar smije početi od prvog stupca
; ovaj bi red bio prazan da nema komentar :)

```

© Kovač, Basch, FER, Zagreb

17

## Asemblieri - Labele



- U C-u se labele koriste jedino kao odredište skoka u naredbi *goto\**

```

for(i=0; i<10; ++i) {
 for(j=0; j<10; ++j) {
 if(A[i][j] == Broj)
 goto pronasao;
 }
}
printf("Broj ne postoji u polju A[][]\n");
...
pronasao: printf("Broj je u A[%d][%]\n", i, j);
...

```

\* Naredba *goto* se inače ne prepriča za upravljanje tokom programa. Bolje je koristiti uvjetne naredbe *if-else* i petlje *while*, *do* i *for*. Ipak je *goto* koristan kod izlaska iz višestrukih petlji.

© Kovač, Basch, FER, Zagreb

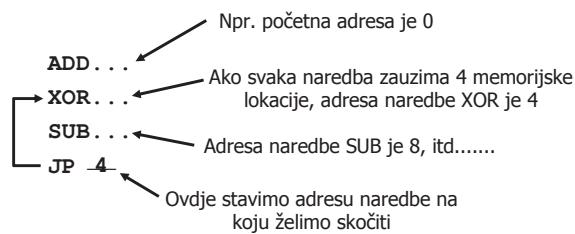
19

## Asemblieri - Labele



- Kako ručno izračunati adresu skoka?

- Prevođenje svake naredbe (i pseudonaredbe) daje strojne kôdove poznate veličine. Ako znamo početnu adresu programa, možemo na temelju toga redom "izračunati" adresu svakog strojnog kôda.
- Ako koristimo labele, onda ovo "izračunavanje" za nas obavlja asemblerski prevoditelj tijekom prevođenja



© Kovač, Basch, FER, Zagreb

21

## Asemblieri - Labele



- **Labele** se koriste i za **adresiranje podatka** koje čitamo i pišemo tijekom rada programa
  - Podatcima se pristupa npr. naredbama LOAD, STORE i sl.
  - Ovi podaci odgovaraju varijablama u višim programskim jezicima
  - Varijable u C-u imaju imena koja sintaksno nisu labele, ali imaju jednaku ulogu:
    - C-prevoditelj za svaku varijablu rezervira prostor u memoriji i dodjeljuje joj labelu (tj. ime varijable)
    - Prevođenjem C-a dobiva se mnemonički program koji za pristup varijablama koristi naredbe tipa LOAD i STORE te dodjeljenu labelu dotične varijable
- **Dakle, glavna namjena labela je da služe kao**
  - **adrese skoka**
  - **adrese podataka**



© Kovač, Basch, FER, Zagreb

23

© Kovač, Basch, FER, Zagreb



### • Napomene:

- Svaka labela se može slobodno **koristiti** u više različitih naredaba
- Međutim, navođenje labele na početku retka mnemoničke datoteke predstavlja mjesto njene **definicije**:
  - Zato se svaka labela smije navesti samo jednom na početku retka
  - U suprotnom bi pokušali redefinirati vrijednost labele što bi bila greška, jer labela ne bi imala jednoznačnu vrijednost
- Svaka labela **koja se koristi** u naredbama **mora biti definirana**, jer inače dolazi do greške u asembliranju (u drugom prolazu javlja se greška o nepoznatoj labeli)

## Asemblери - Pseudonaredbe xconas-a

- Komercijalni asemblери namijenjeni su jednom procesoru i imaju **točno pozname i propisane naredbe i pseudonaredbe** koje se međusobno razlikuju
- ATLAS nije namijenjen samo jednom procesoru pa u njemu **nije poznato kako sve mogu izgledati naredbe procesora**
  - Sve pseudonaredbe u ATLAS-u su unaprijed propisane
  - Da bi se moglo jednoznačno razlikovati naredbe od pseudonaredaba, sve pseudonaredbe počinju znakom jednostrukog obrnutog apostrofa: `



OPREZ: Pri objašnjavanju pseudonaredaba (na sljedećim slajdovima) pretpostavljamo da se svaka naredba prevodi u 32-bitni strojni kod, a da su memorije riječi široke 8 bita (kao kod FRISC-a)

## xconas - Pseudonaredba `ORG

- Moguće je u datoteci navesti više pseudonaredba `ORG čije adrese:
  - moraju biti u rastućem redoslijedu
  - ne smiju biti manje od adrese zadnjeg prevedenog strojnog kôda
- Primjer:

| program: | memorija: |
|----------|-----------|
| adresa   | sadržaj   |
| 0:       | ADD       |
| 4:       | STORE     |
| 8:       | 0         |
| C:       | 0         |
| 10:      | 0         |
| 14:      | LOAD      |
| 18:      | HALT      |

zapravo ADD zauzima adrese 0-3, STORE 4-7, itd.

"preskočeno" do adrese 14

brojevi su heksadekadski

## VAŽNO: Poravnanje naredaba i `ORG

- Čak ako sa `ORG zadamo adresu koja nije djeljiva s 4, prevoditelj će poravnati naredbe:

| program: | memorija: |
|----------|-----------|
| adresa   | sadržaj   |
| 0:       | ADD       |
| 4:       | STORE     |
| 8:       | ...       |
| 25:      | 0         |
| 26:      | 0         |
| 27:      | 0         |
| 28-2B:   | LOAD      |
| 2C-2F:   | HALT      |

asembleri prevoditelj automatski "poravnava" naredbe na adresu djeljivu s 4



- Podsjetnik: pseudonaredbe se mogu pisati u polju naredbe (mogu imati labelu, a neke moraju imati labelu)
- Bitne razlike između naredaba i pseudonaredaba:
  - **Naredbe:**
    - ovise o procesoru
    - imaju svoj strojni kôd
    - izvodi ih procesor tijekom izvođenja programa
  - **Pseudonaredbe:**
    - nemaju veze s procesorom
    - to su naredbe za asemblerski prevoditelj: one upravljaju njegovim radom govoreći mu pobliže kako treba obavljati prevodenje
    - "izvodi" ih asemblerski prevoditelj tijekom prevodenja

## xconas - Pseudonaredba `ORG



- Pseudonaredba `ORG (origin) zadaje asemblерu adresu punjenja strojnog kôda i piše se ovako:

|                  |
|------------------|
| `ORG      adresa |
|------------------|

- Adresa mora biti zadana brojem, a ne labelom
- Strojni kôdovi dobiveni prevodenjem sljedećih redaka datoteke smjestiti će se u memoriji od zadane adrese\*
- ATLAS-ov asemblér daje absolutni strojni kôd pa se mora znati početna adresa punjenja programa:
  - zadaje se pomoću `ORG u prvom retku datoteke
  - ako se `ORG ispusti, onda se prepostavlja početna adresa 0

\* zapravo, od adrese djeljive sa 4 (detaljnije objašnjenje slijedi)

## VAŽNO: Poravnanje naredaba za FRISC

- **Poravnanje naredaba za FRISC**

- Podsjetnik:
  - Memorije lokacije široke su jedan bajt (tj. najmanja količina memorije koja se može adresirati je jedan bajt)
  - Podatkovna sabirnica je širine 32 bita, što znači da se može odjednom pročitati sadržaj 4 memorije lokacije
- Naredbe su široke 32 bita pa su u memoriji uvijek **spremljene na adresama djeljivima s 4**
  - Kažemo da su naredbe poravnate na adresu djeljivu s 4 (memory aligned).



## xconas- Pseudonaredba `ORG



- Primjer:

| program: | memorija: |
|----------|-----------|
| adresa   | sadržaj   |
| 20:      | ADD       |
| 24:      | STORE     |
| 28:      | LOAD      |

greška: 0 je manje od adrese prethodnog `ORG-a

(iako bi na adresama 0 i 4 bilo mesta za strojni kod naredaba XOR i HALT)



- Primjer: program:

```
`ORG 10
ADD ...
JP ...

`ORG 1B
`DS 3
BROJACI `DW 9A, 9B, 9C
...
```

**uočite da pseudonaredba 'DS ne poravnava podatke na adresu djeljivu s 4 !!!**

## memorija:

| adresa | sadržaj |
|--------|---------|
| 10-13: | ADD     |
| 14-17: | JP      |
| 18:    | 0       |
| 19:    | 0       |
| 1A:    | 0       |
| 1B:    | 0       |
| 1C:    | 0       |
| 1D:    | 0       |
| 1E:    | 9A      |
| 1F:    | 9B      |
| 20:    | 9C      |
|        | ...     |



- Primjer:

## program:

```
...
ADD ...
STORE ...

`END
LOAD ...
35fgxm+*=*)+9
```

## memorija:

| sadržaj |
|---------|
| ...     |
| ADD     |
| STORE   |

} zamemareno:

- LOAD se ne prevodi
- pogrešno napisan redak se također zanemaruje



- Područje u kojem vrijedi baza zadana sa 'BASE su retci mnemoničke datoteke ispod dotičnog 'BASE pa sve do kraja datoteke ili do sljedeće pseudonaredbe 'BASE
- Ako ne koristimo 'BASE, onda je podrazumijevana baza ovisna o procesoru (za FRISC je to heksadekadska baza). Učinak je kao da na početku datoteke stoji 'BASE sa podrazumijevanom bazom.
- Baza zadana sa 'BASE odnosi se na sve brojeve koji se pišu u naredbama i pseudonaredbama



- Brojevi se pišu u podrazumijevanoj bazi\*:

- heksadekadska za FRISC

- To se odnosi na sve brojeve koji se pišu u pseudonaredbama i naredbama bez obzira predstavljaju li adresu, podatak, broj podataka ili bilo što drugo
- naravno, izuzetak su brojevi koji ne stoje samostalno, npr. u imenima registara: R0, R1, R2 i sl.

\* Osim ako se koristi pseudonaredba 'BASE, objašnjena na prethodnim info-slajdovima (mi je nećemo rabiti na predavanjima)

- Pseudonaredba 'END zadaje asembleru da treba prekinuti s prevodenjem na tom mjestu u mnemoničkoj datoteci:

```
`END
```

- Sve što se nalazi iza pseudonaredbe 'END zanemaruje se
- Ako se 'END ispusti, prevodenje se, naravno, završava na zadnjem retku mnemoničke datoteke
- Retci napisani iza 'END ne moraju biti napisani prema pravilima za mnemoničku datoteku, jer se ionako zanemaruju i ne prevode se



- Pseudonaredba 'BASE služi za definiranje baze brojevnog sustava koju želimo koristiti:

| LABELA | 'BASE | slово |
|--------|-------|-------|
|--------|-------|-------|

- Labela se može napisati, ali nema značenja, a slovo se obavezno piše i može biti jedno od sljedećih (sa značenjem):
  - B (binarna baza)
  - O (oktalna baza)
  - D (dekadika baza)
  - H (heksadekadska baza)



- Primjer:

|                                                                                          |                                                                                             |
|------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|
| `ORG 0<br>ADD R1, 12FF, R2<br>JP 30A                                                     | } Retci u kojima se koristi baza 16 (podrazumijevana): baza se odnosi na tamnosmeđe brojeve |
| `BASE D<br>BROJACI `DW 15, 16, 17<br>ADD R2, 18, R3<br>LOAD R3, (12)<br>LOAD R3, (R0-92) | } Retci u kojima se koristi baza 10: baza se odnosi na zelene brojeve                       |
| `BASE B<br>`ORG 1000000<br>XOR R1, 1001010, R1                                           | } Retci u kojima se koristi baza 2: baza se odnosi na plave brojeve                         |



- Bez obzira na zadalu bazu, svaki se broj pojedinačno može napisati u željenoj bazi ako se napiše u formatu:

% slovo broj

- Slovo** može biti: B (binarna baza), O (oktalna baza), D (Dekadska baza) ili H (heksadekadska baza)
- Broj** je niz znamenaka u bazi zadanoj sa **slovo**, a može započeti i znakom predznaka + ili -

- Primjeri:

|         |          |
|---------|----------|
| %B 010  | %B -1011 |
| %H 2A7  | %H +1FF  |
| %D 95   | %D -29   |
| %O 7611 | %O -37   |



- Budući da na svakom mjestu broja može stajati i labela, važno pravilo koje omogućuje njihovo razlikovanje je:



### Brojevi započinju znamenkom, a labele započinju slovom

- Ovo pravilo ne predstavlja problem, osim kod heksadekadskih brojeva koji mogu imati slova pa tako mogu i započeti slovom, ali onda će predstavljati labelu
  - Ovaj problem se jednostavno rješava tako da se heksadekadski broj započne sa znamenkom 0 ili predznakom ili sa oznakom baze %H
  - Na primjer:
    - AB12 je labela
    - 0AB12 je broj
    - +AB12 je broj
    - %H AB12 je broj

### Dodatne pomoćne pseudonaredbe



- Za definiranje 16-bitnih i 32-bitnih podataka moramo se koristiti **pomoćnim pseudonaredbama** sličnim pseudonaredbi `DW. Osim njih, zbog pravilnosti, postoji i pomoćna pseudonaredba za definiranje 8-bitnih podataka. Pomoćne pseudonaredbe su:
  - DW - za definiranje 32-bitnih podataka (define word)
  - DH - za definiranje 16-bitnih podataka (define halfword)
  - DB - za definiranje 8-bitnih podataka (define bajt)

#### Pomoćne pseudonaredbe uvijek počinju puniti podatak od adrese djeljive s 4

- U programima ćemo uvijek koristiti (i do sada smo koristili) pomoćne pseudonaredbe DW, DH i DB (prava pseudonaredba `DW koristila bi se samo iznimno, npr. kad bi željeli staviti bajt na neparnu adresu)

### Dodatne pomoćne pseudonaredbe



#### Primjer poravnjanja podataka

memorija:

program:

```
`ORG 1
DW 99887766
DH 5544
`DW 0F
DW 12345678
...
```

| adresa | sadržaj |
|--------|---------|
| 0:     | 0       |
| 1:     | 0       |
| 2:     | 0       |
| 3:     | 0       |
| 4:     | 66      |
| 5:     | 77      |
| 6:     | 88      |
| 7:     | 99      |
| 8:     | 44      |
| 9:     | 55      |
| A:     | 0F      |
| B:     | 0       |
| C:     | 78      |
| D:     | 56      |
| E:     | 34      |
| F:     | 12      |
| ...    | ...     |

`DW ne poravnava podatke

DB, DH i DW poravnaju podatke na adresu djeljivu s 4



### Programiranje u asembleru za FRISC

- Sljedeće poglavlje pokazuje primjere programiranja u asembleru
- Prvo su objašnjeni primjeri osnovnih i čestih zadaća koje u asembleru treba napraviti "ručno" jer za to ne postoje gotove naredbe (rad s bitovima, usporedbi brojeva itd.)
- Primjeri su rađeni tako da budu što razumljiviji i "školski" su napisani, a nije težnja da se zadatak riješi što efikasnije, kraće, brže itd.

### Dodatne pomoćne pseudonaredbe

### Dodatne pomoćne pseudonaredbe

#### Primjer poravnjanja podataka

memorija:

program:

```
`ORG 1
`DW 99, 88, 77, 66
...
`ORG 9
DB 55, 44, 34
`ORG 23
DH 12F0
```

| adresa | sadržaj |
|--------|---------|
| 0:     | 0       |
| 1:     | 99      |
| 2:     | 88      |
| 3:     | 77      |
| 4:     | 66      |
| ...    | ...     |
| 9:     | 0       |
| A:     | 0       |
| B:     | 0       |
| C:     | 55      |
| D:     | 44      |
| E:     | 33      |
| ...    | ...     |
| 23:    | 0       |
| 24:    | 0       |
| 25:    | FO      |
| ...    | ...     |

`DW ne poravnava podatke

DB, DH i DW poravnaju podatke na adresu djeljivu s 4

little endian

### Dodatne pomoćne pseudonaredbe

- Dodata pogodnost pomoćne pseudonaredbe DW je da se kao podatak može zadati labela (to za `DW nije moguće):

program:

```
`ORG 0
SEDEM EQU 7
ADRESA_BLOKA DW BLOK
VARIJABLA DW SEDAM
...
`ORG 10203000
DW 99887766
BLOK DH 555, 44, 123, ...
```

memorija:

| adresa | sadržaj |
|--------|---------|
| 0:     | 04      |
| 1:     | 30      |
| 2:     | 20      |
| 3:     | 10      |
| 4:     | 07      |
| 5:     | 0       |
| 6:     | 0       |
| 7:     | 0       |
| ...    | ...     |

|              |            |
|--------------|------------|
| labela       | vrijednost |
| SEDEM        | 7          |
| ADRESA_BLOKA | 0          |
| VARIJABLA    | 4          |
| BLOK         | 10203004   |

little endian

### FRISC - programiranje u asembleru

### Uspoređivanje brojeva

## Uspoređivanje brojeva

- FRISC ima sve potrebne uvjete u upravljačkim naredbama za jednostavno uspoređivanje brojeva (u formatima NBC i 2'k), tako da ne treba "ručno" ispitivati pojedine zastavice
- Usporedbe se (najčešće) obavljaju na sljedeći način:
  - Prvo se dva broja oduzmu naredbom CMP (ili SUB ako je potrebna i njihova razlika)
  - Naredbom uvjetnog skoka usporede se brojevi: ako je uvjet istinit, onda se skok izvodi, a inače se nastavlja s izvođenjem sljedeće naredbe
  - U naredbi skoka, uvjet se interpretira kao da je operator usporedbe stavljen "između" operanada koji su oduzimani



© Kovač, Basch, FER, Zagreb

56

<<<

- Često je, radi dobivanja jednostavnijeg i efikasnijeg programa, prikladnije postaviti "obrnut" uvjet.
- Na primjer: ako je  $R5 \geq R2$ , onda treba uvećati R7 za 7, a inače ne treba napraviti ništa. Izravnim pisanjem uvjeta dobivamo:

```
CMP R5, R2
JR_UGE UVECAJ_R7
NISTA JR DALJE
UVECAJ_R7 ADD R7, 7, R7
DALJE ...
```

- S obrnutim uvjetom program je nešto razumljiviji, kraći i brži:

```
CMP R5, R2
JR_ULT DALJE
UVECAJ_R7 ADD R7, 7, R7
DALJE ...
```

© Kovač, Basch, FER, Zagreb

58

## Uspoređivanje brojeva

### Primjer:

Usporediti dva 2'k-broja spremljena u registrima R0 i R1. Manji od njih treba staviti u R2. Drugim riječima treba napraviti:

$$R2 = \min(R0, R1)$$

### Rješenje:

```
CMP R0, R1
JR_SLT R0_MANJI
R0_VECI_JEDNAK MOVE R1, R2 ; R1 je manji
 JR KRAJ
R0_MANJI MOVE R0, R2 ; R0 je manji
KRAJ HALT
```

© Kovač, Basch, FER, Zagreb

60

## FRISC - programiranje u asembleru

## Rad s bitovima

## Uspoređivanje brojeva

- Na primjer, želimo li usporediti je li broj u R5 veći ili jednak od broja u R2 (uz pretpostavku da su to NBC-brojevi):
  - Želimo postaviti uvjet  $R5 \geq R2$  pa upotrijebimo sufiks UGE:
    - U: unsigned - jer uspoređujemo NBC-brojeve
    - G: greater - jer ispitujemo je li R5 veći od R2
    - E: equal - jer ispitujemo je li R5 jednak R2
- U naredbi CMP pišemo brojeve u istom redoslijedu kao u uvjetu  $R5 \geq R2$ :

```
CMP R5, R2
JR_UGE UVJET_ISTINIT
```

>>>

© Kovač, Basch, FER, Zagreb

57

- Često je potrebno samo ispitati je li broj pozitivan ili negativan:
- Moguće je usprediti broj sa ništicom ("školski pristup" ☺):

```
CMP R5, 0
JR_SGE POZIT
NEG ...
POZIT ...
```

- Umjesto oduzimanja može se upotrijebiti logička naredba AND ili OR, a također se može upotrijebiti razumljiviji uvjet:

```
AND R5, R5, R5 ; ne mijenja R5
JR_P POZIT ; _P znači pozitivan
NEG ...
POZIT ...
```

- Ako je broj u R5 dobiven prethodnom ALU-naredbom, onda su zastavice već postavljene pa nije potrebno dodatno izvoditi AND, OR ili CMP.

© Kovač, Basch, FER, Zagreb

59

## Uspoređivanje brojeva

### Primjer:

Ispitati 2'k-broj u registru R6. Ako je negativan, u R0 treba staviti broj -1. Ako je jednak ništici, u R0 treba upisati 0. Ako je pozitivan, treba u R0 upisati 1. Drugim riječima treba napraviti:

$$R0 = \text{signum}(R6)$$

### Rješenje:

```
OR R6, R6, R0 ; broj ispitaj i stavi u R0
JR_Z KRAJ ; ako je 0, u R0 je rezultat
JR_N NEG ; ispitaj predznak
POZ MOVE 1, R0
 JR KRAJ
NEG MOVE -1, R0
KRAJ HALT
```

© Kovač, Basch, FER, Zagreb

61

## Rad s bitovima

- Čest zadatak u asemblerском programiranju
- Potrebno je mijenjati ili ispitivati bitove u registru ili memorijskoj lokaciji
  - Rad s bitovima u registru jednostavno se ostvaruje kombinacijama aritmetičko-logičkih naredaba
  - Rad s bitovima u memorijskoj lokaciji nije moguć pa se zato podatak prvo prebaci u jedan od registara, radi se s bitovima te se podatak vrati u memorijsku lokaciju
- Osnovne operacije s bitovima:
  - postavljanje (set)
  - brisanje (reset)
  - komplementiranje (complement)
  - ispitivanje (test)









- Alternativno se umjesto brisanja bitova u registru i podatku mogu isti bitovi postavljati, ali se tada umjesto OR koristi operacija AND

Podatak:

\_\_\_\_\_ yyyy

Maskirani podatak:

1111 yyyy

Registrar nakon AND:

Poravnati podatak: 11 yyyy 11 → AND → xx yyyy xx

Registrar:

xx \_\_\_\_\_ xx

Maskirani register:

xx 1111 xx

- Također je moguće prvo poravnati podatak pa tek onda u njemu brisati (ili postavljati) bitove koji se ne upisuju u odredišni register

- Ako se rotacija obavlja za više bitova, onda u C odlazi samo izlazni bit od "zadnjeg koraka rotacije"



- Zato treba rotirati register jedan po jedan bit (u petlji) i ispitivati zastavicu C\*



- Prebrajanje bitova koristi se, npr. kod određivanja pariteta podatka

\*Drugi način je ispitivanje zastavice N, jer se u njoj nalazi najviši bit registra nakon rotacije (nije toliko uobičajeno rješenje)

### Primjer:

**Prebrojiti** koliko ništica ima u bitovima 3 do 8 registra R0. Broj ništica treba spremiti u memorijsku lokaciju NISTICE.

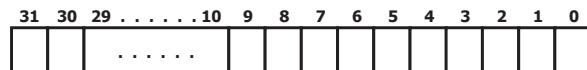
```

MOVE 0, R1 ; R1 = brojač ništica
MOVE 6, R2 ; R2 = brojač za petlju
ROTR R0, 3, R0 ; "izbaci" bitove 0 do 2

LOOP ROTR R0, 1, R0
 JR_C JEDAN
NULA ADD R1, 1, R1
JEDAN SUB R2, 1, R2
 JR_NZ LOOP

 STORE R1, (NISTICE)
 HALT

```



### Višestruka preciznost



- Dijelovi računala (memorijske lokacije, registri, ALU, sabirnice) su ograničeni na određen broj bita
- Npr. FRISC ima 32-bitnu arhitekturu (tj. riječ mu ima 32 bita) pa može normalno raditi s podatcima te širine
- Ako treba raditi s brojevima većeg opsega ili kakvim drugim podatcima širima nego što stanu u riječ procesora ili memorijsku lokaciju, onda koristimo **višestruku preciznost**
- Ovisno koliko procesorskih riječi se koristi za zapis podatka, govorimo o dvostrukoj, trostrukoj, itd. preciznosti



## Višestruka preciznost



- Načelno** se operacije u višestrukoj preciznosti uvijek obavljaju jednakom, bez obzira koristimo li dvostruku, trostruku ili neku veću preciznost
  - Zato ćemo pokazati kako se koristi dvostruka preciznost
- Kod zapisivanja podataka u višestrukoj preciznosti u memoriji, treba podatak zapisivati u više uzastopnih lokacija
  - slično zapisivanju 32-bitnih riječi u bajtnoj memoriji, treba voditi računa o rasporedu zapisivanja pojedinih dijelova podatka unutar memorijskih lokacija
  - Budući da FRISC koristi little-endian za zapis 32-bitnih riječi unutar memorije, onda možemo koristiti isti redoslijed i kod višestruke preciznosti (iako to nije nužno)

### Višestruka preciznost - Pohrana podataka



- Kod zapisivanja podatka u dvostrukoj preciznosti u registrima, također se moraju koristiti dva registra\*
- Kod označavanja podataka obično se koriste sufiksi L i H koji označavaju:
  - niži dio podatka (L - low)
  - viši dio podatka (H - high)
- Npr. podatak A u dvostrukoj preciznosti označava se (po dijelovima) oznakama AL i AH

\* Budući da registri nemaju adresu, onda nema smisla govoriti o zapisima little-endian ili big-endian

## Višestruka preciznost - Zbrajanje

- Kod izvođenja operacija u višestrukoj preciznosti moramo programski oponašati izvođenje operacije onako kako bi se ona izvodila sklopovski
- Zato je potrebno poznavati:

- kako je građen podatak (koji kod je upotrebljen, značenja pojedinih bitova, težine bitova, predznak itd.)
- kako se obavljaju operacije na dotičnom podatku (kako je sklopovski izvedena operacija, redoslijed operacija na bitovima, utjecaj operacije na zastavice, utjecaj zastavica na operaciju itd.)

© Kovač, Basch, FER, Zagreb

96

## Višestruka preciznost - Zbrajanje

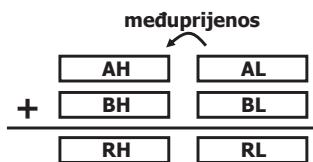


- Isti redoslijed moramo ostvariti programski:
  - prvo zbrajamo niže riječi podataka, a kasnije više riječi
  - ali, moramo **uračunati prijenose** koji idu od nižih ka višim riječima

R = A + B realizira se u dva koraka:

$$1) RL = AL + BL$$

$$2) RH = AH + BH + \text{međuprijenos}$$



© Kovač, Basch, FER, Zagreb

98

## Višestruka preciznost - Zbrajanje

### Primjer:

Zbrojiti NBC ili 2'k brojeve u dvostrukoj preciznosti. Prvi operand smješten je na memorijskim lokacijama AL (niži dio) i AH (viši dio), a drugi na lokacijama BL i BH. Rezultat se spremi na RL i RH.

```
; ZBROJI NIŽE DIJELOVE
LOAD R0, (AL)
LOAD R1, (BL)
ADD R0, R1, R2
STORE R2, (RL)

; ZBROJI VIŠE DIJELOVE
LOAD R0, (AH)
LOAD R1, (BH)
ADC R0, R1, R2
STORE R2, (RH)

HALT
```

© Kovač, Basch, FER, Zagreb

```
; OPERANDI
AL DW 0A3541E21
AH DW 942F075F

BL DW 936104A7
BH DW 017F3784

; MJESTO ZA REZULTAT
RL DW 0
RH DW 0
```

100

## Višestruka preciznost - Oduzimanje



- Kako uračunati međuposudbu? Pomoću **naredbe SBC**.
  - Podsjetnik: naredba SBC, osim umanjitelja, oduzima i vrijednost posudbe iz prethodne operacije oduzimanja
- Budući da je posudba od oduzimanja spremljena u zastavici C, onda naredba SBC zapravo radi ovako:

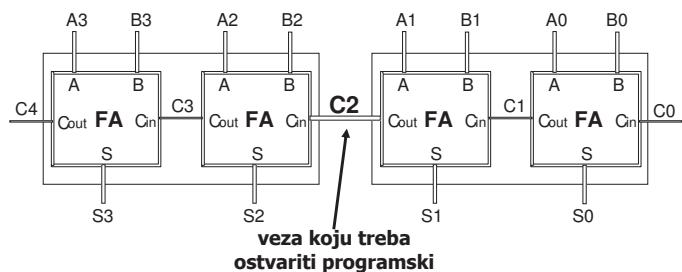
$$SBC X,Y,R \equiv X-Y-\text{posudba} \rightarrow R \equiv X-Y-C \rightarrow R$$

>>>

## Višestruka preciznost - Zbrajanje



- Npr. spoj dva 2-bitna zbrajala daje 4-bitno zbrajalo:



Sklopovsko zbrajanje: zbrajaju se pojedini bitovi, a prijenos propagira od nižih ka višim bitovima

© Kovač, Basch, FER, Zagreb

97

## Višestruka preciznost - Zbrajanje



- Kako uračunati međuprijenos? Pomoću **naredbe ADC**.

- Podsjetnik: naredba ADC, osim dva pribrojnika, pribraja i vrijednost prijenosa iz prethodne operacije zbrajanja

- Budući da je prijenos od zbrajanja spremljen u zastavici C, onda naredba ADC zapravo radi ovako:

$$ADC X,Y,R \equiv X+Y+\text{prijenos} \rightarrow R \equiv X+Y+C \rightarrow R$$

- Sklopovski se naredba ADC izvodi tako da se na ulaz C0, na najnižem potpunom zbrajalu, dovede stanje iz zastavice C (podsjetnik: kod običnog zbrajanja dovodi se 0)

© Kovač, Basch, FER, Zagreb

99

## Višestruka preciznost - Oduzimanje



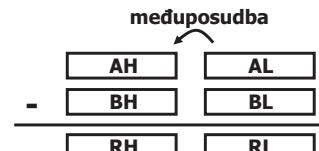
- Oduzimanje se obavlja slično zbrajanju

- U programski ostvarenom oduzimanju u dvostrukoj preciznosti:
  - prvo oduzimamo niže riječi podataka, a kasnije više riječi
  - ali, moramo **uračunati posudbe** koji idu od nižih ka višim riječima

R = A - B realizira se u dva koraka:

$$1) RL = AL - BL$$

$$2) RH = AH - BH - \text{međuposudba}$$



© Kovač, Basch, FER, Zagreb

101

## Višestruka preciznost - Oduzimanje



- Sklopovski se naredba SBC izvodi tako da se na ulaz C0, na najnižem potpunom zbrajalu, dovede komplement zastavice C, tj. not(C) (podsjetnik: kod običnog oduzimanja dovodi se 1)
- Zašto se naredba izvodi baš ovako? Zato jer se oduzimanje izvodi preko operacija dvojnog komplementa, tj. operacija X-Y se izvodi kao X+not(Y)+1
- Ako želimo napraviti X-Y-C, onda se to sklopovski treba izvesti kao: X+not(Y)+1-C. Budući da se na ulaze zbrajala dovedi X i not(Y), ostaje nam samo ulaz C0 da na njega dovedemo 1-C
- Vrijednost 1-C upravo je jednaka not(C)



## Primjer:

Oduzeti NBC ili 2'k brojeve u dvostrukoj preciznosti. Prvi operand smješten je na memorijskim lokacijama AL (niži dio) i AH (viši dio), a drugi na lokacijama BL i BH. Rezultat se spremi na RL i RH.

```
; ODUZMI NIŽE DIJELOVE
LOAD R0, (AL)
LOAD R1, (BL)
SUB R0, R1, R2
STORE R2, (RL)

; ODUZMI VIŠE DIJELOVE
LOAD R0, (AH)
LOAD R1, (BH)
SBC R0, R1, R2
STORE R2, (RH)

HALT
```

© Kovač, Basch, FER, Zagreb

104



- Sklopovski ostvareni pomaci i rotacije prenose bitove između pojedinih riječi pa to također treba napraviti i u programu
- Redoslijed operacija na pojedinim riječima podataka nije bitan, ali ovisno o operaciji može biti praktičniji jedan ili drugi redoslijed. Npr. pomak u lijevo za 1 bit:



- Prvo pomaknemo uljevo RL. Izlazni bit je u zastavici C. Pomaknemo RH uljevo i upišemo C u najniži bit od RH.
- Prvo pomaknemo uljevo RH. Zatim pomaknemo uljevo RL. Izlazni bit je u zastavici C. Upišemo C u najniži bit od RH.
- Oba redoslijeda izgledaju podjednako komplikirano...

© Kovač, Basch, FER, Zagreb

105

## Primjer:

**Rotirati u desno** za 1 mjesto podatak u dvostrukoj preciznosti zapisan u registrima R0 (niža riječ) i R1 (viša riječ).

Početno stanje prije rotacije i način rotacije:



Željeno stanje nakon rotacije u dvostrukoj preciznosti:



Nakon neovisnih rotacija izvedenih na R0 i R1:



**Vidimo da su bitovi V i N na ispravnom mjestu, ali bitovi X i Y moraju zamjeniti mjesta**

© Kovač, Basch, FER, Zagreb

>>>

106

Kako najjednostavnije zamjeniti bitove X i Y? Oni mogu imati sljedeće vrijednosti:

| X | Y | operacija          |
|---|---|--------------------|
| 0 | 0 | -                  |
| 0 | 1 | treba ih zamjeniti |
| 1 | 0 | treba ih zamjeniti |
| 1 | 1 | -                  |

zamjenu bitova koji su različiti jednostavno ostvarimo tako da ih komplementiramo

- Komplementiranje znamo napraviti od prije: XOR s maskom

- Ako je  $X \neq Y$ , onda trebamo komplementirati bitove i maska ima ništice na svim bitovima osim na poziciji bita X i Y (najniži bit)
- Ako je  $X=Y$ , onda ne trebamo komplementirati bitove i maska treba cijela biti ispunjena ništicama

- Takvu masku dobivamo ako napravimo XOR između R0 i R1 i zatim obrišemo sve bitove osim bita na poziciji X i Y (najniži bit)

© Kovač, Basch, FER, Zagreb

110



- Logičke operacije AND, OR, XOR, NOT rade neovisno na pojedinim bitovima podataka
- Zato nije bitan redoslijed obavljanja operacija na pojedinim riječima podatka - jedino treba obaviti operacije na svim riječima
- Također, ne postoji nikakvi podatci, kao međuprijenosni ili međuposudbe, koje bi trebalo prenosi između viših i nižih riječi podataka

© Kovač, Basch, FER, Zagreb

105



- Prvo pomaknemo uljevo RL. Izlazni bit je u zastavici C. Pomaknemo RH uljevo i upišemo C u najniži bit od RH:

```
SHL RL, 1, RL
JRC JEDAN
NULA SHL RH, 1, RH
JR DALJE
JEDAN SHL RH, 1, RH
OR RH, 1, RH
```

- Prvo pomaknemo uljevo RH. Zatim pomaknemo uljevo RL. Izlazni bit je u zastavici C. Upišemo C u najniži bit od RH:

```
SHL RH, 1, RH
SHL RL, 1, RL
ADC RH, 0, RH
```

Ipak je ovo

jednostavnija varijanta

© Kovač, Basch, FER, Zagreb

107

## Rješenje:

- varijanta: napraviti dvije obične rotacije na R0 i R1, a zatim im zamjeniti vrijednosti najviših bitova X i Y.

- varijanta: prvo zamjeniti najviše bitove X i Y, pa tek onda napraviti obje rotacije. Ovo će biti programski lakše.

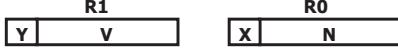
Početno stanje prije rotacije i način rotacije:



Nakon zamjene bitova X i Y:



Nakon neovisnih rotacija izvedenih na R0 i R1:



;

stvaranje maske u R3 (za zamjenu X i Y)

;

ako su bitovi X i Y jednaki => maska=0

;

ako su bitovi X i Y različiti => maska=1

```
XOR R0, R1, R3 ; Usporedi najviše bitove
AND R3, 1, R3 ; u R0 i R1, tj. bitove X i Y.
```

;

zamjena najnižih bitova R0 i R1 (tj. X i Y)

```
XOR R3, R0, R0
XOR R3, R1, R1
```

;

Nezavisna rotacija R0 i R1

```
ROTR R0, 1, R0
ROTR R1, 1, R1
```

HALT

© Kovač, Basch, FER, Zagreb

111

## Višestruka preciznost - Proširivanje podataka

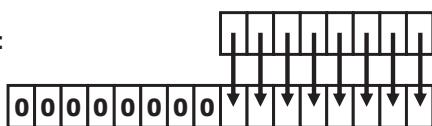
- Ponekad je prvi operand u jednostrukoj, a drugi operand u višestrukoj preciznosti
- Ponekad imamo podatke u jednostrukoj preciznosti, a nakon operacije trebamo dobiti rezultat u višestrukoj preciznosti (npr. sumiranje veće količine podataka)
- U takvim slučajevima se javlja potreba za **proširivanjem podataka** s jednostrukih na višu preciznost



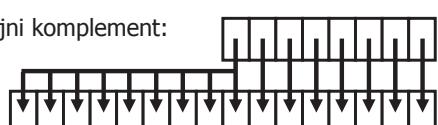
## Višestruka preciznost - Proširivanje podataka

- Sumirajmo vrste proširivanja za nekoliko tipova podataka (proširivanje sa 8 na 16 bita):

- NBC:



- Dvojni komplement:

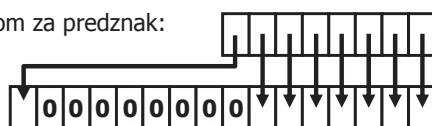


>>>

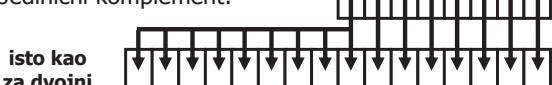
## Višestruka preciznost - Proširivanje podataka

<<<

- Prikaz s bitom za predznak:



- Jedinični komplement:



## Višestruka preciznost - Proširivanje podataka

- Ponekad podatak iz višestruke preciznosti moramo pretvoriti u podatak u jednostrukoj preciznosti (ili općenito u podatak u nižoj preciznosti)



- Moramo biti svjesni da to općenito nije moguće napraviti bez gubitka informacija

- Izuzetak je slučaj kad je broj u višestrukoj dovoljno malen da se može zapisati u nižoj preciznosti, što se može provjeriti:



- Npr. za NBC: svi viši bitovi koji se odbacuju moraju biti jednaki ništicama
- Npr. za 2'k: svi viši bitovi koji se odbacuju moraju biti ili jednaki ništicama ili jednaki jedinicama i dodatno moraju biti jednaki najvišem bitu u broju u nižoj preciznosti

<<<

```

PETLJA LOAD R0, (R1) ; dohvati podatak
 ADD R0, R3, R3 ; zbroji niži dio
 ADC R4, 0, R4 ; zbroji viši dio
 ADD R1, 4, R1
 SUB R2, 1, R2
 JR_NZ PETLJA Nema eksplicitnog proširivanja
 podatka, nego se u naredbi ADC
 zbraja s ništicom (tj. s višom
 riječ proširenog podatka)
 HALT
 ; blok sa 500 podataka
 `ORG 1000
 DW 1234, 563, 2A3523, ...

```



## Višestruka preciznost - Proširivanje podataka

### Primjer:

Zbrojiti 500 32-bitnih NBC-brojeva koji se nalaze u memoriji počevši od adrese 1000<sub>16</sub>. Rezultat u dvostrukoj preciznosti treba zapisati u registre R3 (niži dio) i R4 (viši dio).

### Rješenje:

```

MOVE 1000, R1 ; adresa podataka
MOVE %D 500, R2 ; brojač za petlju

MOVE 0, R3 ; niži dio rezultata
MOVE 0, R4 ; viši dio rezultata

```

>>>

## Višestruka preciznost - Proširivanje podataka

### Primjer:

Zbrojiti 500 32-bitnih 2'k-brojeva koji se nalaze u memoriji počevši od adrese 1000<sub>16</sub>. Rezultat u dvostrukoj preciznosti treba zapisati u registre R3 (niži dio) i R4 (viši dio).

### Rješenje:

```

MOVE 1000, R1 ; adresa podataka
MOVE %D 500, R2 ; brojač za petlju

MOVE 0, R3 ; niži dio rezultata
MOVE 0, R4 ; viši dio rezultata

```

>>>

&lt;&lt;&lt;

```

PETLJA LOAD R0, (R1) ; dohvati podatak
 OR R0, R0, R0 ; postavi zastavice
 JR_N NEG ; ispitaj predznak

POZ ADD R0, R3, R3 ; zbroji niži dio
 ADC R4, 0, R4 ; zbroji viši dio
 JR DALJE

NEG ADD R0, R3, R3
 ADC R4, -1, R4

DALJE ADD R1, 4, R1
 SUB R2, 1, R2
 JR_N PETLJA
 HALT
... ; podatci: kao u prošlom primjeru

```

© Kovač, Basch, FER, Zagreb

120

razlika u odnosu na  
prethodni primjer

## Potprogrami

- Namjena potprograma u asemblerском programiranju ista je kao i u višim programskim jezicima.

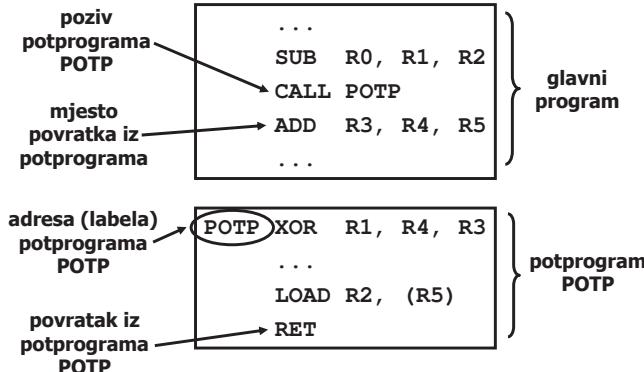
- Čemu služi potprogram znate već od prije (iz PIP-a):
  - bolja modularnost i organizacija programa
  - lakše programiranje
  - povećana čitljivost
  - lakše održavanje programa
  - manje zauzeće memorije

© Kovač, Basch, FER, Zagreb

122

## Potprogrami

- Kako izgleda glavni program koji poziva neki potprogram POTP:



© Kovač, Basch, FER, Zagreb

124

## Potprogrami - CALL i RET

### Gdje se u memoriji nalazi stog ???

- Položaj stoga u memoriji definiramo programski na početku glavnog programa gdje **MORAMO inicijalizirati pokazivač stoga** (SP)
- Ako to zaboravimo učiniti, doći će do greške u programu (u ATLAS-u će vjerojatno javiti grešku "čitanje sa sabirnice u stanju High Z")

© Kovač, Basch, FER, Zagreb

126

## Potprogrami

© Kovač, Basch, FER, Zagreb

121

## Potprogrami

- Kao i u višim programskim jezicima, moramo znati kako se koriste potprogrami, ali u asemblerском programiranju moramo dodatno znati i neke detalje "niže razine"
  - Trebamo znati:
    - Kako se potprogram poziva i kako se vraćamo iz potprograma
    - Gdje se spremi povratna adresa
    - Kako se prenose parametri i vraćaju rezultati iz potprograma
    - Gdje se čuvaju lokalne variable potprograma
    - Kako se ostvaruje da stanja registara iz glavnog programa ne budu promijenjena za vrijeme potprograma
  - U višim programskim jezicima, o ovim stvarima uglavnom ne moramo voditi računa jer se o njima brine prevoditelj (koji stvara asemblersku verziju našeg programa)

© Kovač, Basch, FER, Zagreb

123

## Potprogrami - CALL i RET

### Podsjetnik: kako rade naredbe CALL i RET?

- Naredba: **CALL adresa\_potprograma**
  - sprema povratnu adresu iz PC-a na stog
  - to je adresa naredbe IZA naredbe CALL, jer za vrijeme izvođenja naredbe CALL, PC već pokazuje na sljedeću naredbu, tj. sadrži povratnu adresu
  - skače na zadanu adresu **adresa\_potprograma**
- Naredba: **RET**
  - uzima podatak (povratnu adresu) sa stoga
  - skače na tu povratnu adresu

© Kovač, Basch, FER, Zagreb

125

## Potprogrami - stog

&lt;&lt;&lt;

0000 0000

Tipična organizacija memorije:

- Na najnižim adresama se nalazi program i variable/konstante.
- Iza toga se nalazi područje gomile (heap). Gomila dinamički raste prema najvišim adresama. (*nema veze sa strukturom podataka koja se također naziva heap - iz "Algoritama i struktura podataka"*).
- Na najvišim adresama se nalazi stog koji raste prema nižim adresama.

|                                                                                       |
|---------------------------------------------------------------------------------------|
| <b>Program i variabile/konst.<br/>(fiksna veličina)</b>                               |
| <b>Gomila (heap):<br/>malloc i free</b>                                               |
|  |
| <b>Stog (stack):<br/>push i pop</b>                                                   |
|  |

&gt;&gt;&gt;

© Kovač, Basch, FER, Zagreb

127



<<<

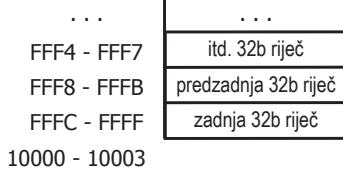
- Mi ćemo pretpostaviti da je na FRISC spojena memorija od 64 KB.**

Najviša adresa (8-bitne memorijске lokacije) u memoriji od 64 KB je FFFF

- Na stog se uвijek stavljuju 32-bitni podaci pa im adrese moraju biti djeljive sa 4

• FFFC je adresa 32-bitnog podatka koji je "na zadnjoj 32-bitnoj lokaciji", tj. "na najvišim lokacijama memorije"

• Budući da SP pokazuje na zadnji podatak na stogu (podatak koji na početku još ne postoji), onda SP inicijaliziramo na jednu 32-bitnu riječ dalje: 4+FFFC = 10000



© Kovač, Basch, FER, Zagreb

128



- U višim programskim jezicima parametri i povratna vrijednost imaju tipove koje provjerava prevoditelj

- U asembleru nema nikakvih provjera ni tipova:

- Dužnost je programera da se brine kakve podatke šalje u potprogram i kakve rezultate prima iz potprograma
- Velika fleksibilnost u programiranju
- Velika mogućnost greške u programu

- Terminologija:**

- parametar (formalni parametar) - "varijabla" u potprogramu preko koje potprogram prima vrijednosti od pozivatelja
- argument (stvarni parametar) - stvarna vrijednost koju pozivatelj šalje potprogramu preko njegovih parametara

© Kovač, Basch, FER, Zagreb

130



© Kovač, Basch, FER, Zagreb

131

- Za svaki potprogram zasebno, propiše se preko kojih registara prima podatke i preko kojih vraća rezultate
- Pozivatelj potprograma (glavni program ili bilo koji drugi potprogram) prije poziva mora staviti argumente u zadane registre
- Potprogram polazi od pretpostavke da su u zadanim registrima argumenti i koristi ih pri izračunavanju rezultata
- Potprogram mora staviti rezultat u zadane registre
- Pozivatelj, nakon povratka iz potrograma, pretpostavlja da se u zadanim registrima nalaze rezultati i koristi ih

© Kovač, Basch, FER, Zagreb

132

© Kovač, Basch, FER, Zagreb

133

### Primjer:

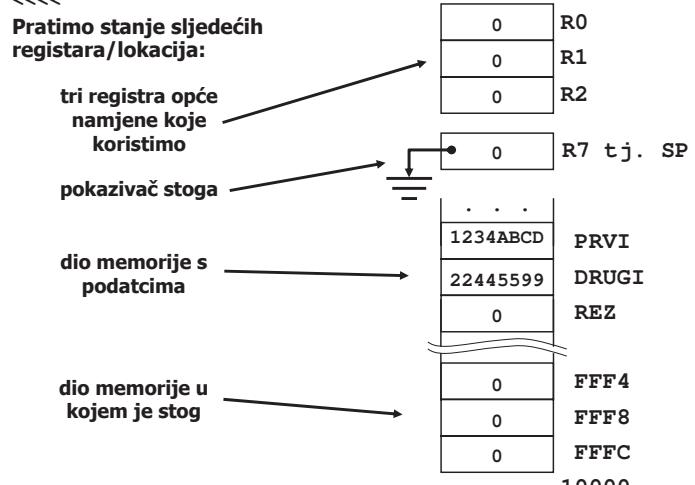
Napisati potprogram koji izračunava logičku operaciju NILI između registra R0 i R1 i vraća rezultat registrom R2. Glavni program iz memorije učitava dva podatka za koje računa NILI (pomoću potrograma) i rezultat sprema natrag u memoriju.

### Rješenje:

(na sljedećem slajdu)

>>>

© Kovač, Basch, FER, Zagreb



© Kovač, Basch, FER, Zagreb

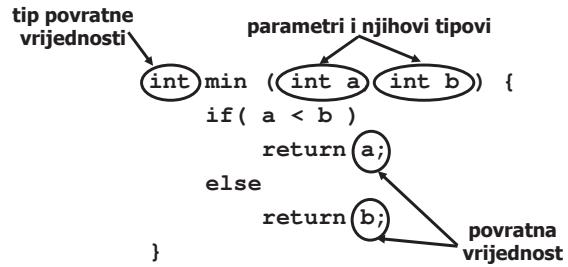
134

© Kovač, Basch, FER, Zagreb

135



- Kao i u višim programskim jezicima, potprogrami obavljaju izračunavanja u ovisnosti o parametrima te vraćaju pozivatelju izračunati rezultat ili povratnu vrijednost:



© Kovač, Basch, FER, Zagreb

129



<<< Izvođenje programa:

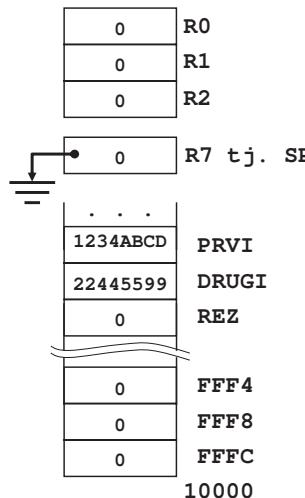
```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
LOAD R1, (DRUGI)
CALL NILI
STORE R2, (REZ)

HALT

NILI OR R0, R1, R2
XOR R2, -1, R2
RET

PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0
```

© Kovač, Basch, FER, Zagreb



136

<<< Izvođenje programa:

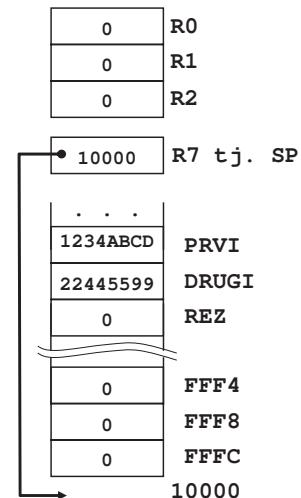
```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
LOAD R1, (DRUGI)
CALL NILI
STORE R2, (REZ)

HALT

NILI OR R0, R1, R2
XOR R2, -1, R2
RET

PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0
```

© Kovač, Basch, FER, Zagreb



10000

137

<<< Izvođenje programa:

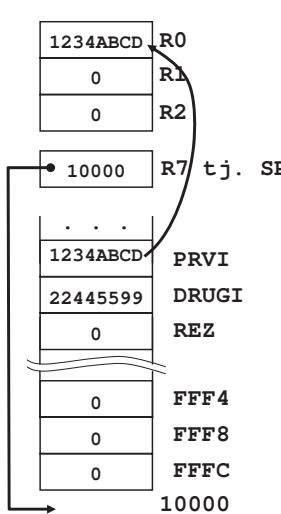
```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
LOAD R1, (DRUGI)
CALL NILI
STORE R2, (REZ)

HALT

NILI OR R0, R1, R2
XOR R2, -1, R2
RET

PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0
```

© Kovač, Basch, FER, Zagreb



138

<<< Izvođenje programa:

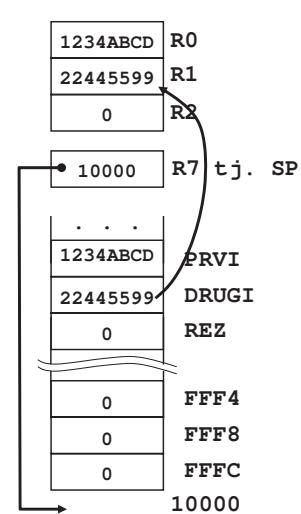
```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
LOAD R1, (DRUGI)
CALL NILI
STORE R2, (REZ)

HALT

NILI OR R0, R1, R2
XOR R2, -1, R2
RET

PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0
```

© Kovač, Basch, FER, Zagreb



10000

139

<<< Izvođenje programa:

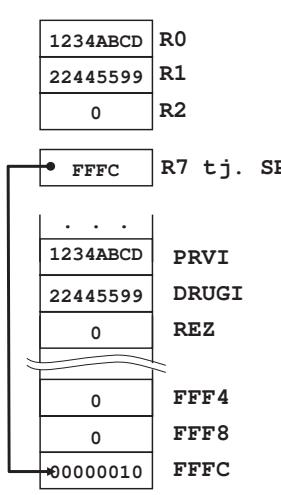
```
GLAVNI MOVE 10000, SP 0
LOAD R0, (PRVI) 4
LOAD R1, (DRUGI) 8
CALL NILI 10 ← C
STORE R2, (REZ) 10

HALT 14

NILI OR R0, R1, R2
XOR R2, -1, R2
RET

PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0
```

© Kovač, Basch, FER, Zagreb



140

00000010 je adresa naredbe STORE

<<< Izvođenje programa:

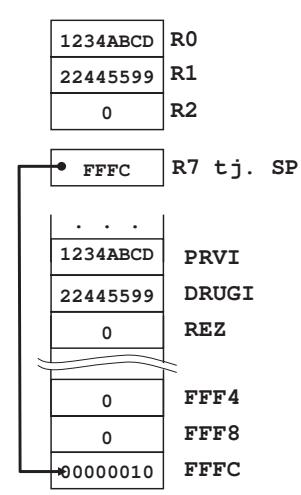
```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
LOAD R1, (DRUGI)
CALL NILI
STORE R2, (REZ)

HALT

NILI OR R0, R1, R2
XOR R2, -1, R2
RET

PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0
```

© Kovač, Basch, FER, Zagreb



10000

141

<<< Izvođenje programa:

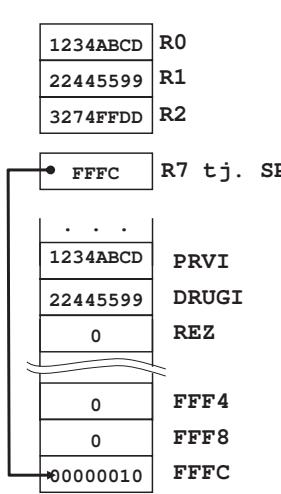
```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
LOAD R1, (DRUGI)
CALL NILI
STORE R2, (REZ)

HALT

NILI OR R0, R1, R2
XOR R2, -1, R2
RET

PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0
```

© Kovač, Basch, FER, Zagreb



142

<<< Izvođenje programa:

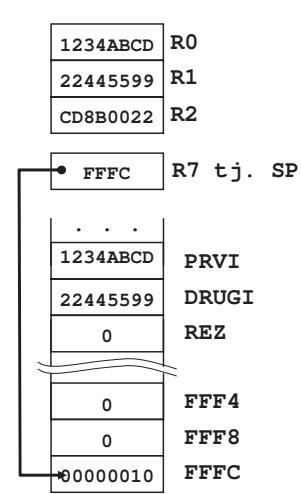
```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
LOAD R1, (DRUGI)
CALL NILI
STORE R2, (REZ)

HALT

NILI OR R0, R1, R2
XOR R2, -1, R2
RET

PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0
```

© Kovač, Basch, FER, Zagreb



10000

143

#### <<< Izvođenje programa:

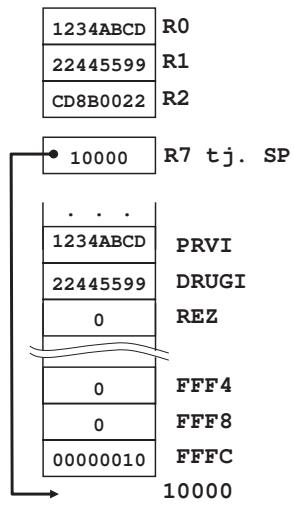
```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
LOAD R1, (DRUGI)
CALL NILI
STORE R2, (REZ)

HALT

NILI OR R0, R1, R2
XOR R2, -1, R2
RET

PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0
```

© Kovač, Basch, FER, Zagreb



144

#### <<< Izvođenje programa:

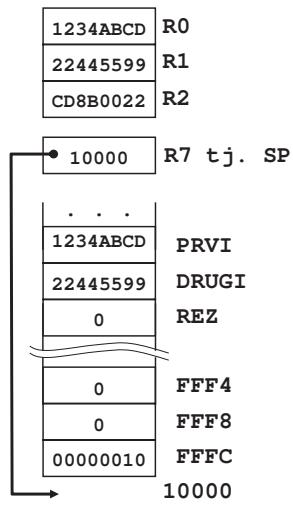
```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
LOAD R1, (DRUGI)
CALL NILI
STORE R2, (REZ)

HALT

NILI OR R0, R1, R2
XOR R2, -1, R2
RET

PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0
```

© Kovač, Basch, FER, Zagreb



145

#### <<< Izvođenje programa:

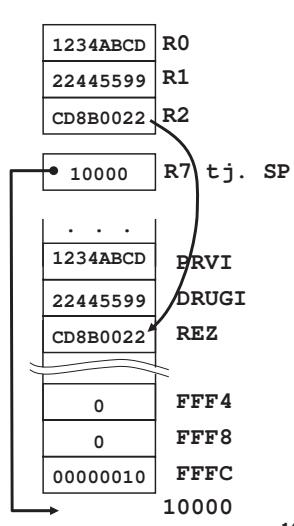
```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
LOAD R1, (DRUGI)
CALL NILI
STORE R2, (REZ) ←

HALT

NILI OR R0, R1, R2
XOR R2, -1, R2
RET

PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0
```

© Kovač, Basch, FER, Zagreb



146

#### <<< Izvođenje programa:

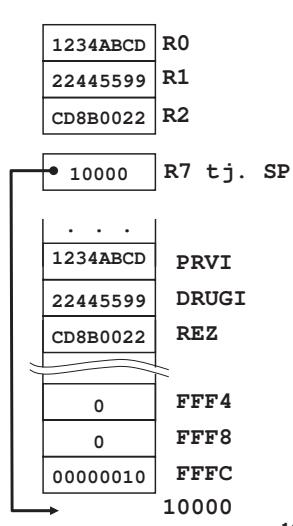
```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
LOAD R1, (DRUGI)
CALL NILI
STORE R2, (REZ)

HALT

NILI OR R0, R1, R2
XOR R2, -1, R2
RET

PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0
```

© Kovač, Basch, FER, Zagreb



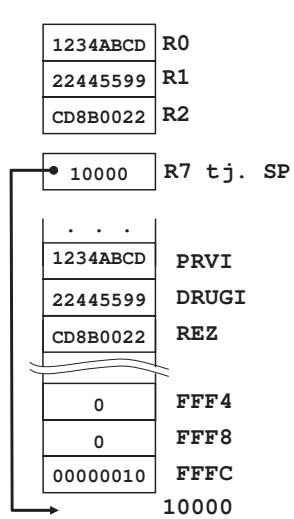
147

#### Komentar:

Treba uočiti da ovaj potprogram mijenja samo sadržaj registra R2 (ne računajući R7). Međutim, R2 je register preko kojeg se vraća vrijednost, tj. glavni program očekuje da će taj register biti promijenjen nakon povratka iz potprograma. Zato glavni program prije poziva potprograma ne smije imati u R2 spremljjen nikakav podatak koji bi mu još mogao zatrebatи.

Registre R0 i R1 promijenio je glavni program, jer je preko njih slao parametre. Zato je za prepostaviti da glavni program nije imao nikakve korisne podatke u R0 i R1 prije upisa parametara u njih.

© Kovač, Basch, FER, Zagreb



148

## Potprogrami - Prijenos registrima



### • Prijenos registrima

- Prednosti:
  - Brz prijenos (rad s registrima je najbrži)
  - Jednostavan prijenos (sa stajališta programiranja)
- Nedostatci:
  - Može se prenijeti ograničen broj argumenata (najviše 7)
  - Registri obično čuvaju međurezultate i nisu slobodni za korištenje kao parametri
  - Nije moguće rekurzivno pozivanje potprograma

### Prijenos fiksnim lokacijama

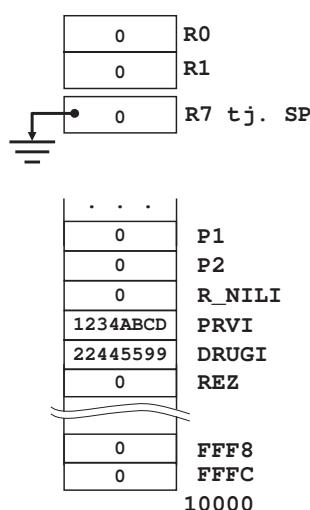


- Pod fiksnim lokacijama misli se na memoriske lokacije čija adresa je unaprijed zadana i ne mijenja se tijekom programa
- Za svaki potprogram zasebno, propiše se preko kojih fiksnih memoriskih lokacija prima podatke i preko kojih vraća rezultate
- Pozivatelj prije poziva mora staviti argumente u zadane lokacije
- Potprogram polazi od prepostavke da su u zadanim lokacijama argumenti i koristi ih pri izračunavanju rezultata
- Potprogram mora staviti rezultat u zadane lokacije
- Pozivatelj, nakon povratka iz potrograma, prepostavlja da se u zadanim lokacijama nalaze rezultati i koristi ih

© Kovač, Basch, FER, Zagreb

#### <<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
STORE R0, (P1)
LOAD R0, (DRUGI)
STORE R0, (P2)
CALL NILI
LOAD R0, (R_NILI)
STORE R0, (REZ)
HALT
```

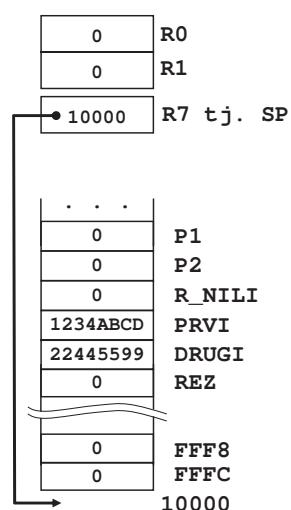


152

© Kovač, Basch, FER, Zagreb

#### <<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP ←
LOAD R0, (PRVI)
STORE R0, (P1)
LOAD R0, (DRUGI)
STORE R0, (P2)
CALL NILI
LOAD R0, (R_NILI)
STORE R0, (REZ)
HALT
```

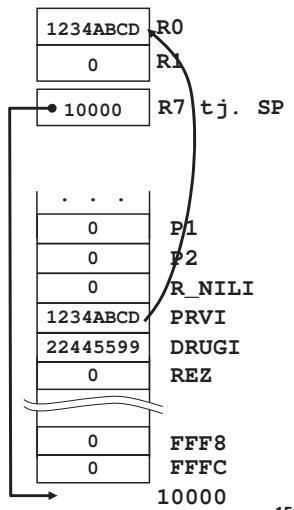


153

© Kovač, Basch, FER, Zagreb

#### <<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI) ←
STORE R0, (P1)
LOAD R0, (DRUGI)
STORE R0, (P2)
CALL NILI
LOAD R0, (R_NILI)
STORE R0, (REZ)
HALT
```

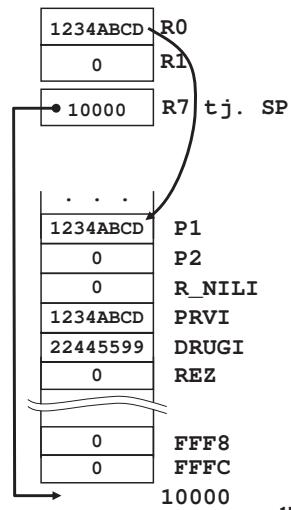


154

© Kovač, Basch, FER, Zagreb

#### <<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI) ←
LOAD R0, (DRUGI)
STORE R0, (P2)
CALL NILI
LOAD R0, (R_NILI)
STORE R0, (REZ)
HALT
```

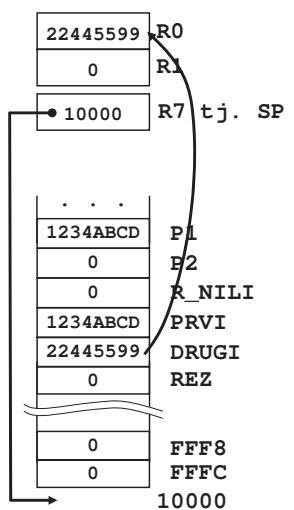


155

© Kovač, Basch, FER, Zagreb

#### <<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
STORE R0, (P1)
LOAD R0, (DRUGI) ←
STORE R0, (P2)
CALL NILI
LOAD R0, (R_NILI)
STORE R0, (REZ)
HALT
```

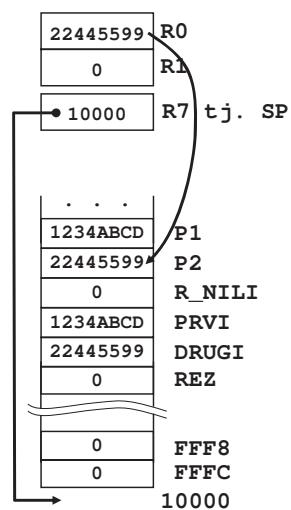


156

© Kovač, Basch, FER, Zagreb

#### <<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
STORE R0, (P1)
LOAD R0, (DRUGI) ←
STORE R0, (P2)
CALL NILI
LOAD R0, (R_NILI)
STORE R0, (REZ)
HALT
```



157

© Kovač, Basch, FER, Zagreb

© Kovač, Basch, FER, Zagreb

#### Primjer:

Napisati potprogram koji izračunava logičku operaciju NILI. Parametri se nalaze na memoriskim lokacijama P1 i P2, a rezultat se vraća lokacijom R\_NILI. Glavni program iz memorije učitava dva podatka za koje računa NILI (pomoću potprograma) i rezultat spremi natrag u memoriju.

#### Rješenje:

na sljedećem slajdu

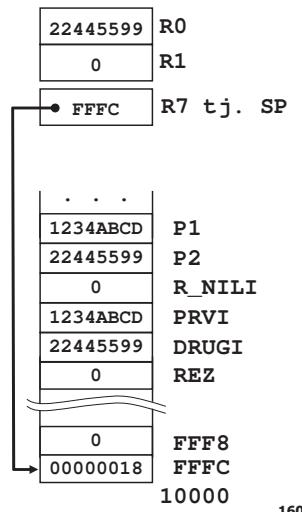
>>>

158

159

<<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
STORE R0, (P1)
LOAD R0, (DRUGI)
STORE R0, (P2)
CALL NILI
LOAD R0, (R_NILI)
STORE R0, (REZ)
HALT
```

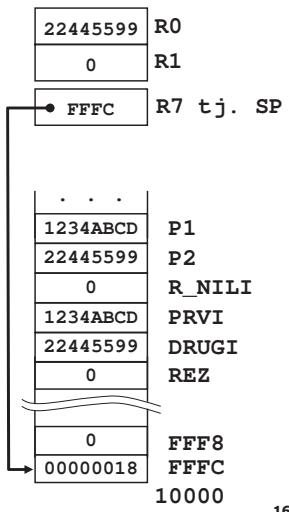


© Kovač, Basch, FER, Zagreb

<<< Izvođenje programa:

```
NILI LOAD R0, (P1)
LOAD R1, (P2)
OR R0, R1, R0
XOR R0, -1, R0
STORE R0, (R_NILI)
RET
```

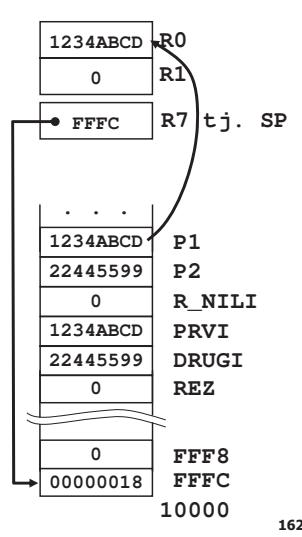
© Kovač, Basch, FER, Zagreb



161

<<< Izvođenje programa:

```
NILI LOAD R0, (P1)
LOAD R1, (P2)
OR R0, R1, R0
XOR R0, -1, R0
STORE R0, (R_NILI)
RET
```

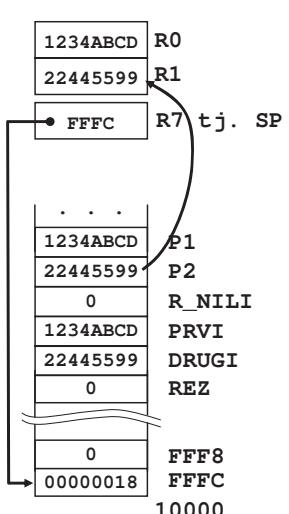


© Kovač, Basch, FER, Zagreb

<<< Izvođenje programa:

```
NILI LOAD R0, (P1)
LOAD R1, (P2)
OR R0, R1, R0
XOR R0, -1, R0
STORE R0, (R_NILI)
RET
```

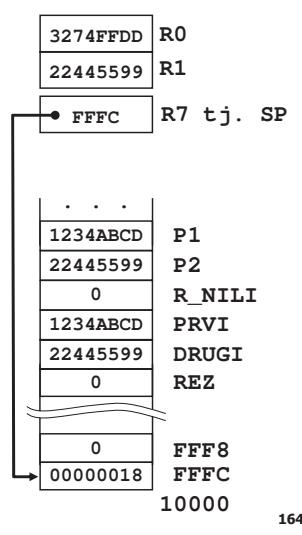
© Kovač, Basch, FER, Zagreb



163

<<< Izvođenje programa:

```
NILI LOAD R0, (P1)
LOAD R1, (P2)
OR R0, R1, R0
XOR R0, -1, R0
STORE R0, (R_NILI)
RET
```

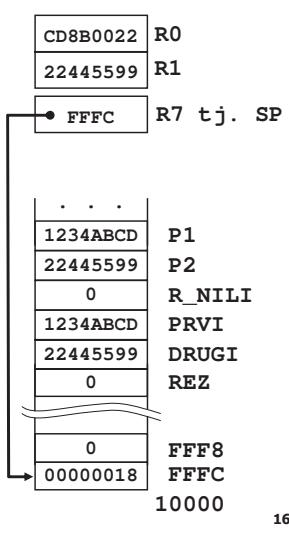


© Kovač, Basch, FER, Zagreb

<<< Izvođenje programa:

```
NILI LOAD R0, (P1)
LOAD R1, (P2)
OR R0, R1, R0
XOR R0, -1, R0
STORE R0, (R_NILI)
RET
```

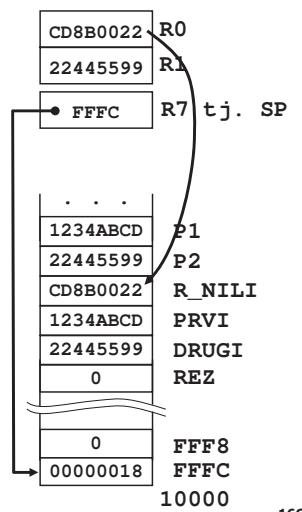
© Kovač, Basch, FER, Zagreb



165

<<< Izvođenje programa:

```
NILI LOAD R0, (P1)
LOAD R1, (P2)
OR R0, R1, R0
XOR R0, -1, R0
STORE R0, (R_NILI)
RET
```

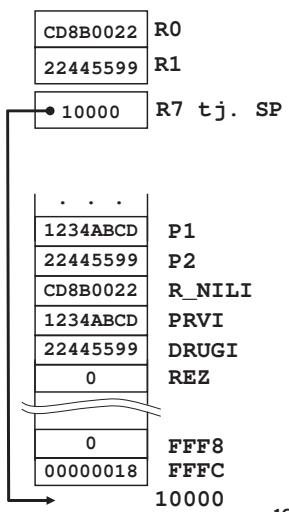


© Kovač, Basch, FER, Zagreb

<<< Izvođenje programa:

```
NILI LOAD R0, (P1)
LOAD R1, (P2)
OR R0, R1, R0
XOR R0, -1, R0
STORE R0, (R_NILI)
RET
```

© Kovač, Basch, FER, Zagreb



167

<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

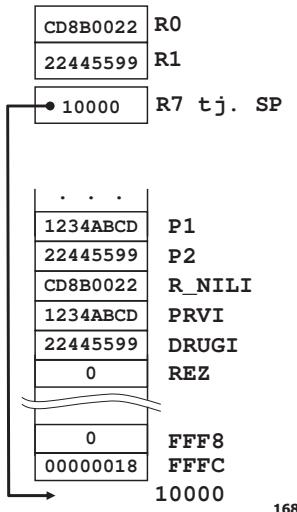
LOAD R0, (PRVI)  
STORE R0, (P1)

LOAD R0, (DRUGI)  
STORE R0, (P2)

CALL NILI

LOAD R0, (R\_NILI)  
STORE R0, (REZ)

HALT



© Kovač, Basch, FER, Zagreb

<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

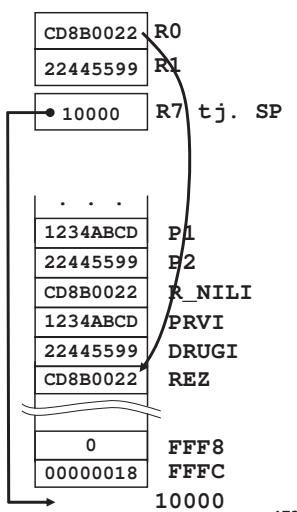
LOAD R0, (PRVI)  
STORE R0, (P1)

LOAD R0, (DRUGI)  
STORE R0, (P2)

CALL NILI

LOAD R0, (R\_NILI)  
STORE R0, (REZ)

HALT



© Kovač, Basch, FER, Zagreb

<<< (kompletan listing s komentarima)

```
; glavni program
GLAVNI MOVE 10000, SP ;važno: inicijaliziraj SP !!!
; stavi vrijednost u prvi parametar
LOAD R0, (PRVI)
STORE R0, (P1)
; stavi vrijednost u drugi parametar
LOAD R0, (DRUGI)
STORE R0, (P2)

CALL NILI ;poziv potprograma

; uzmi rezultat i spremi ga
LOAD R0, (R_NILI)
STORE R0, (REZ) ;spremanje rezultata

HALT
```

>>>

© Kovač, Basch, FER, Zagreb



172

Komentar:

**Prethodni potprogram ima značajan nedostatak, a to je promjena vrijednosti u registrima R0 i R1.** Ukoliko je glavni program imao u njima podatke koji će mu još trebati, oni će nakon poziva potprograma biti izgubljeni i glavni program neće raditi ispravno.

Jedna (loša) mogućnost je da se za svaki potprogram zna koje registre mijenja. Tada se pri izradi programa mora voditi računa da na mjestima pozivanja potprograma u tim registrima ne budu nikakvi korisni podatci.

**Bolje rješenje je da potprogram "sačuva" sadržaje registara koje će mijenjati.**

<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

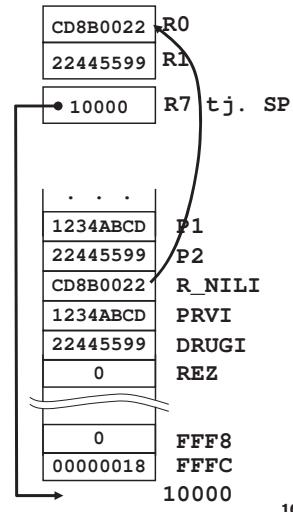
LOAD R0, (PRVI)  
STORE R0, (P1)

LOAD R0, (DRUGI)  
STORE R0, (P2)

CALL NILI

LOAD R0, (R\_NILI) ←  
STORE R0, (REZ)

HALT



© Kovač, Basch, FER, Zagreb

<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

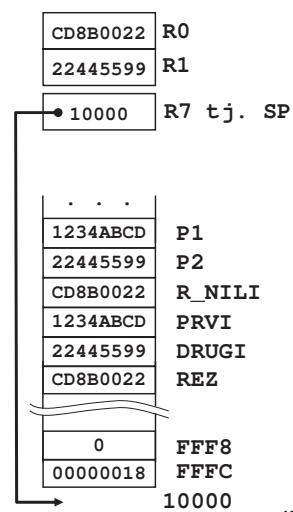
LOAD R0, (PRVI)  
STORE R0, (P1)

LOAD R0, (DRUGI)  
STORE R0, (P2)

CALL NILI

LOAD R0, (R\_NILI)  
STORE R0, (REZ)

HALT



© Kovač, Basch, FER, Zagreb

<<<

```
; potprogram NILI
NILI LOAD R0, (P1) ; dohvati prvi parametra
 ; dohvati drugog parametra
 ; Izračunavanje
 ; rezultata.

STORE R0, (R_NILI) ; upis rezultata u memoriju
RET

; fiksne lokacije za parametre
; i povratnu vrijednost
P1 DW 0
P2 DW 0
R_NILI DW 0

; podatci i mjesto za rezultat
PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0
```

>>>

© Kovač, Basch, FER, Zagreb



173

Ovo je bio naš potprogram koji mijenja registre (R0 i R1):

NILIT

```
LOAD R0, (P1) ; dohvati prvi parametra
LOAD R1, (P2) ; dohvati drugog parametra
OR R0, R1, R0 ; Izračunavanje
XOR R0, -1, R0 ; rezultata.

STORE R0, (R_NILI) ; upis rezultata u memoriju
```

RET

Ovo je potprogram koji sprema registre koje mijenja

Registri se spremaju na fiksne memorijske lokacije

```
NILI STORE R0, (R0_SPREM) ; spremi R0 i R1 na
 STORE R1, (R1_SPREM) ; fiksne lokacije

 LOAD R0, (P1) ; dohvati prvi parametar
 LOAD R1, (P2) ; dohvati drugi parametar

 OR R0, R1, R0 ; Izračunavanje rezultata.
 XOR R0, -1, R0 ; rezultata.

 STORE R0, (R_NILI) ; upis rezultata u memoriju
 LOAD R0, (R0_SPREM) ; obnovi vrijednosti
 LOAD R1, (R1_SPREM) ; registara R0 i R1
 RET

R0_SPREM DW 0 ; Dodatne lokacije za
R1_SPREM DW 0 ; spremanje registara
```

Ovakvo spremanje onemogućuje rekurzivne pozive i zahtijeva definiranje posebnih memorijskih lokacija - nije idealno

© Kovač, Basch, FER, Zagreb

176



Bolje rješenje je spremanje na stog:

```
NILI PUSH R0 ; spremi R0 i R1
 PUSH R1 ; na stog

 LOAD R0, (P1) ; dohvati prvi parametar
 LOAD R1, (P2) ; dohvati drugi parametar

 OR R0, R1, R0 ; Izračunavanje rezultata.
 XOR R0, -1, R0 ; rezultata.

 STORE R0, (R_NILI) ; upis rezultata u memoriju
 POP R1 ; obnovi vrijednosti registara
 POP R0 ; R0 i R1 (OPREZ: REDOSLIJED!!!)
 RET
```

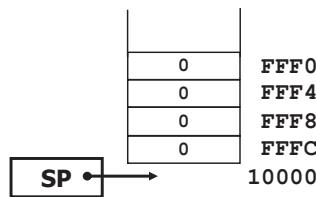
Ovakvo spremanje je bolje, omogućuje rekurzivne pozive i uobičajeno se koristi

© Kovač, Basch, FER, Zagreb

177

### Stog tijekom izvođenja:

```
GLAVNI ...
 CALL NILI
 LOAD R0, (R_NILI)
 ...
```



176

```
NILI PUSH R0
 PUSH R1

 ... ; naredbe
 ... ; potprograma

 POP R1
 POP R0
 RET
```

### Stog tijekom izvođenja:

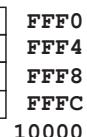
```
GLAVNI ...
 CALL NILI
 LOAD R0, (R_NILI)
```



```
NILI PUSH R0
 PUSH R1

 ... ; naredbe
 ... ; potprograma

 POP R1
 POP R0
 RET
```



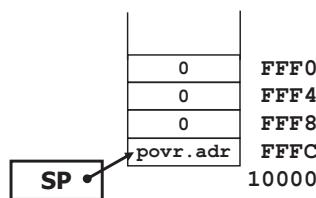
177

© Kovač, Basch, FER, Zagreb

178

### Stog tijekom izvođenja:

```
GLAVNI ...
 CALL NILI
 LOAD R0, (R_NILI)
 ...
```

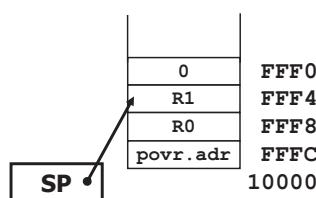


© Kovač, Basch, FER, Zagreb

178

### Stog tijekom izvođenja:

```
GLAVNI ...
 CALL NILI
 LOAD R0, (R_NILI)
 ...
```



© Kovač, Basch, FER, Zagreb

179

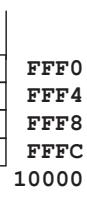
### Stog tijekom izvođenja:

```
GLAVNI ...
 CALL NILI
 LOAD R0, (R_NILI)
 ...
```



```
NILI PUSH R0
 PUSH R1

 ... ; naredbe
 ... ; potprograma
```



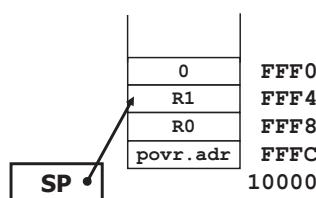
180

© Kovač, Basch, FER, Zagreb

180

### Stog tijekom izvođenja:

```
GLAVNI ...
 CALL NILI
 LOAD R0, (R_NILI)
 ...
```



© Kovač, Basch, FER, Zagreb

182

### Stog tijekom izvođenja:

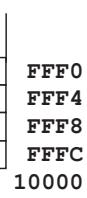
```
GLAVNI ...
 CALL NILI
 LOAD R0, (R_NILI)
 ...
```

```
NILI PUSH R0
 PUSH R1

 ... ; naredbe
 ... ; potprograma
```



```
POP R1
 POP R0
 RET
```



183

© Kovač, Basch, FER, Zagreb

Registri R0 i R1 se mijenjaju.  
Stanje stoga se ne mijenja  
(smije se mijenjati, ako nakon naredaba potprograma stanje bude jednako kao prije njih)

## Stog tijekom izvođenja:

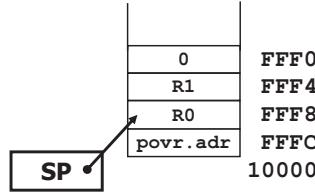
```

GLAVNI ...
 CALL NILI
 LOAD R0, (R_NILI)
 ...

NILI PUSH R0
 PUSH R1
 ...
 ; naredbe
 ; potprograma
 POP R1 ←
 POP R0
 RET

```

Registar R1 se obnavlja.



© Kovač, Basch, FER, Zagreb

184

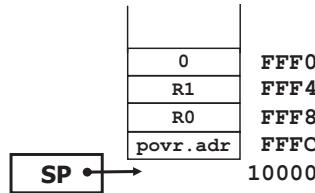
## Stog tijekom izvođenja:

```

GLAVNI ...
 CALL NILI
 LOAD R0, (R_NILI)
 ...

NILI PUSH R0
 PUSH R1
 ...
 ; naredbe
 ; potprograma
 POP R1
 POP R0
 RET ←

```



© Kovač, Basch, FER, Zagreb

186

## Potprogrami - Prijenos fiksnim lokacijama

© Kovač, Basch, FER, Zagreb

187

### • Prijenos fiksnim lokacijama

- Prednosti:
  - Nema ograničenja na broj parametara (osim veličine memorije)
  - Registri su slobodni za druge namjene
- Nedostatci:
  - Sporiji rad s memorijskim lokacijama, nego s registrima (upis i čitanje)
  - Nešto duži i komplikiraniji program, nego kad se koriste registri
  - Nije moguće rekurzivno pozivanje potprograma

© Kovač, Basch, FER, Zagreb

188

## Potprogrami - Prijenos stogom

© Kovač, Basch, FER, Zagreb

189

- Pozivatelj prije poziva mora staviti argumente na stog
  - Oprez: naredba **CALL** stavlja povratnu adresu na stog
- Potprogram polazi od prepostavke da su na stogu ispod povratne adrese argumenti i koristi ih pri izračunavanju rezultata (**kako?**)
- Potprogram mora staviti rezultat na stog (**kako?**)
  - Oprez: naredba **RET** uzima s vrha stoga povratnu adresu
- Pozivatelj, nakon povratka iz potrograma, prepostavlja da se na vrhu stoga nalaze rezultati koje uzima sa stoga i koristi ih

## Stog tijekom izvođenja:

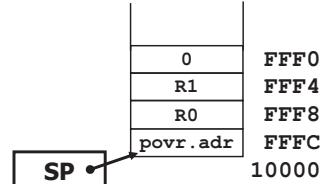
```

GLAVNI ...
 CALL NILI
 LOAD R0, (R_NILI)
 ...

NILI PUSH R0
 PUSH R1
 ...
 ; naredbe
 ; potprograma
 POP R1 ←
 POP R0
 RET

```

Registar R0 se obnavlja.



© Kovač, Basch, FER, Zagreb

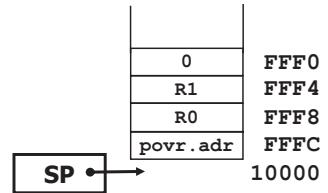
## Stog tijekom izvođenja:

```

GLAVNI ...
 CALL NILI
 LOAD R0, (R_NILI)
 ...

NILI PUSH R0
 PUSH R1
 ...
 ; naredbe
 ; potprograma
 POP R1
 POP R0 ←
 RET ←

```



© Kovač, Basch, FER, Zagreb

187

## Prijenos stogom

© Kovač, Basch, FER, Zagreb

189

## Potprogrami - Prijenos stogom

- Problem je povratna adresa koja se automatski stavlja i uzima na stog naredbama CALL i RET.
- Ova povratna adresa:
  - smeta dohvatu argumenata sa stoga
  - smeta stavljanju rezultata na stog
- Ispravno baratanje sa stogom moguće je na više načina, ali treba poznavati kako radi stog i naredbe CALL i RET

## Potprogrami - Prijenos stogom

### Primjer:

Napisati potprogram koji izračunava logičku operaciju NILI. Parametri i rezultat se prenose stogom. Glavni program iz memorije učitava dva podatka za koje računa NILI (pomoću potprograma) i rezultat spremi natrag u memoriju.

### Rješenje:

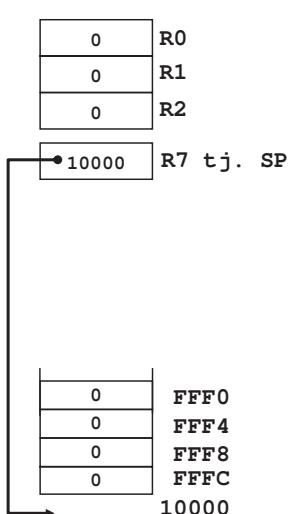
- U ovom rješenju potprogram mijenja registre R0, R1 i R2, ali **ne čuva njihove vrijednosti (LOŠE!!!)**
- Kasnije ćemo pokazati bolje rješenje (ovo je samo primjer kako se može zaobići povratna adresa pri slanju parametara i povratu rezultata)

© Kovač, Basch, FER, Zagreb

### <<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP ←
LOAD R0, (PRVI)
PUSH R0
LOAD R0, (DRUGI)
PUSH R0
CALL NILI
POP R0
STORE R0, (REZ)
HALT
```

192

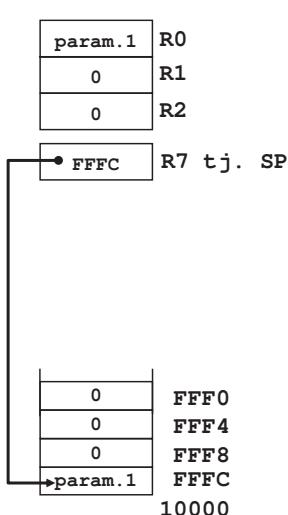


© Kovač, Basch, FER, Zagreb

### <<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
PUSH R0 ←
LOAD R0, (DRUGI)
PUSH R0
CALL NILI
POP R0
STORE R0, (REZ)
HALT
```

194

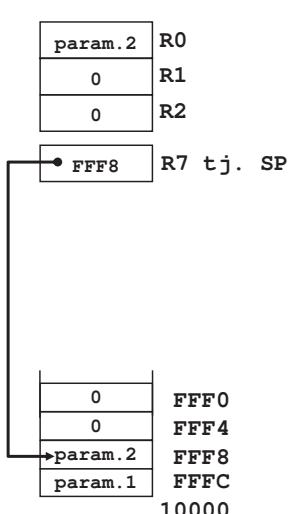


© Kovač, Basch, FER, Zagreb

### <<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
PUSH R0
LOAD R0, (DRUGI)
PUSH R0 ←
CALL NILI
POP R0
STORE R0, (REZ)
HALT
```

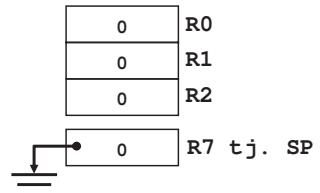
196



© Kovač, Basch, FER, Zagreb

### <<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
PUSH R0
LOAD R0, (DRUGI)
PUSH R0
CALL NILI
POP R0
STORE R0, (REZ)
HALT
```

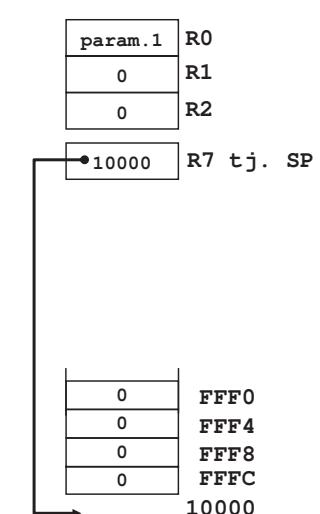


© Kovač, Basch, FER, Zagreb

### <<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP ←
LOAD R0, (PRVI)
PUSH R0
LOAD R0, (DRUGI)
PUSH R0
CALL NILI
POP R0
STORE R0, (REZ)
HALT
```

193

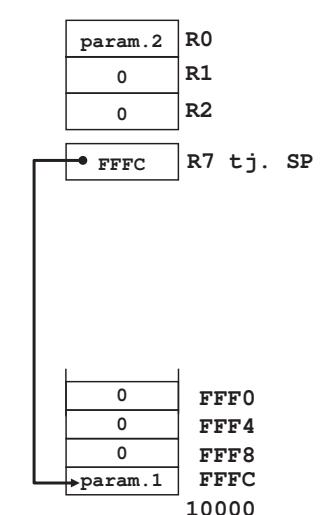


© Kovač, Basch, FER, Zagreb

### <<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
PUSH R0
LOAD R0, (DRUGI)
PUSH R0
CALL NILI
POP R0
STORE R0, (REZ)
HALT
```

195

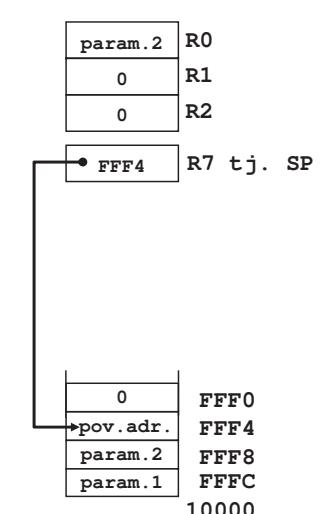


© Kovač, Basch, FER, Zagreb

### <<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
PUSH R0
LOAD R0, (DRUGI)
PUSH R0
CALL NILI ←
POP R0
STORE R0, (REZ)
HALT
```

197



© Kovač, Basch, FER, Zagreb

### <<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
PUSH R0
LOAD R0, (DRUGI)
PUSH R0
CALL NILI ←
POP R0
STORE R0, (REZ)
HALT
```

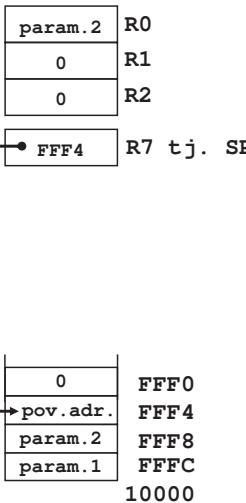
199

<<< Izvođenje programa:

```
NILI POP R2
 POP R0
 POP R1

 OR R0, R1, R0
 XOR R0, -1, R0

 PUSH R0
 PUSH R2
 RET
```



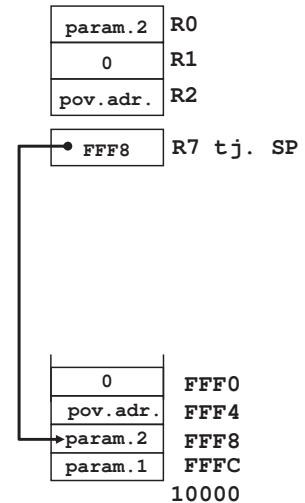
© Kovač, Basch, FER, Zagreb

<<< Izvođenje programa:

```
NILI POP R2
 POP R0
 POP R1

 OR R0, R1, R0
 XOR R0, -1, R0

 PUSH R0
 PUSH R2
 RET
```



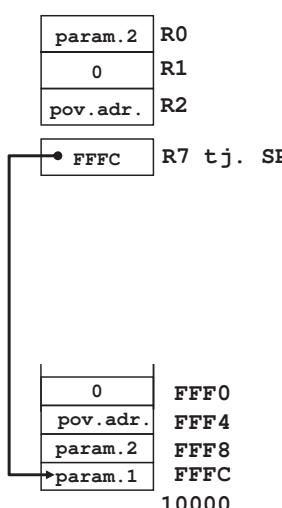
201

<<< Izvođenje programa:

```
NILI POP R2
 POP R0
 POP R1

 OR R0, R1, R0
 XOR R0, -1, R0

 PUSH R0
 PUSH R2
 RET
```



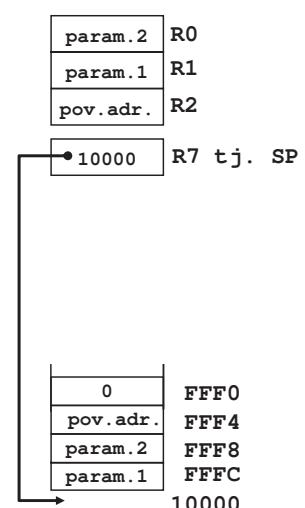
© Kovač, Basch, FER, Zagreb

<<< Izvođenje programa:

```
NILI POP R2
 POP R0
 POP R1

 OR R0, R1, R0
 XOR R0, -1, R0

 PUSH R0
 PUSH R2
 RET
```



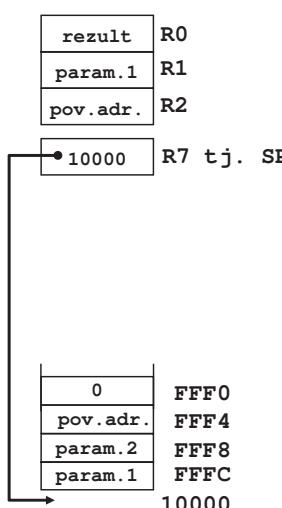
203

<<< Izvođenje programa:

```
NILI POP R2
 POP R0
 POP R1

 OR R0, R1, R0
 XOR R0, -1, R0

 PUSH R0
 PUSH R2
 RET
```



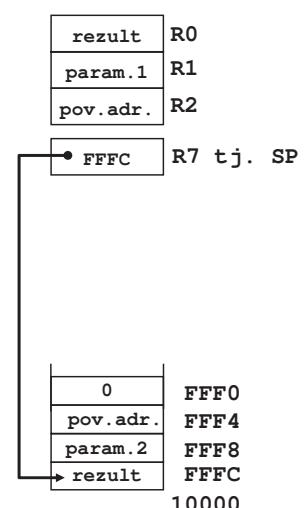
© Kovač, Basch, FER, Zagreb

<<< Izvođenje programa:

```
NILI POP R2
 POP R0
 POP R1

 OR R0, R1, R0
 XOR R0, -1, R0

 PUSH R0
 PUSH R2
 RET
```



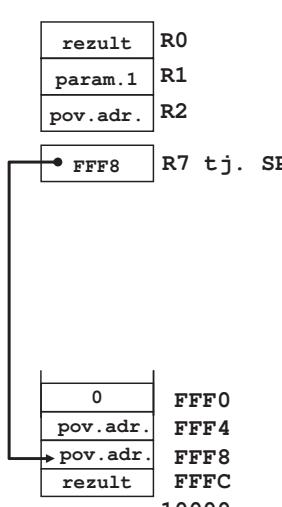
205

<<< Izvođenje programa:

```
NILI POP R2
 POP R0
 POP R1

 OR R0, R1, R0
 XOR R0, -1, R0

 PUSH R0
 PUSH R2
 RET
```



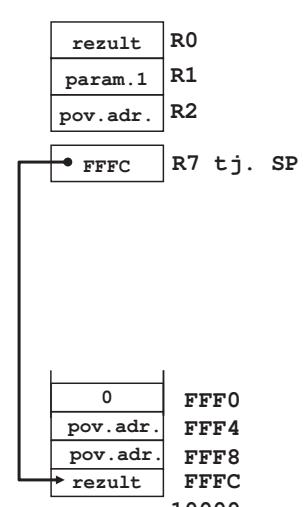
© Kovač, Basch, FER, Zagreb

<<< Izvođenje programa:

```
NILI POP R2
 POP R0
 POP R1

 OR R0, R1, R0
 XOR R0, -1, R0

 PUSH R0
 PUSH R2
 RET
```



207

#### <<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
PUSH R0
LOAD R0, (DRUGI)
PUSH R0
CALL NILI
POP R0
STORE R0, (REZ)
HALT
```

© Kovač, Basch, FER, Zagreb

#### <<< Izvođenje programa:

- dohvaćeni rezultat se koristi
- stog je u početnom stanju

```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
PUSH R0
LOAD R0, (DRUGI)
PUSH R0
CALL NILI
POP R0
STORE R0, (REZ)
```

© Kovač, Basch, FER, Zagreb

<<<

```
; potprogram NILI
NILI POP R2 ; uzmi povratnu adresu sa stoga
POP R0 ; dohvati prvi parametar
POP R1 ; dohvati drugi parametar

OR R0, R1, R0 ; Izračunavanje
XOR R0, -1, R0 ; rezultata.

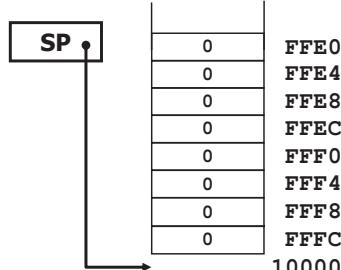
PUSH R0 ; stavi rezultat na stog
PUSH R2 ; vrati povratnu adresu na stog
RET
```

```
; podatci i mjesto za rezultat
PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0
```

© Kovač, Basch, FER, Zagreb

### Potprogrami - Prijenos stogom

Promatramo stanje na stogu za potprogram koji bi spremao registre na stog.



212

#### <<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
PUSH R0
LOAD R0, (DRUGI)
PUSH R0
CALL NILI
POP R0
STORE R0, (REZ)
HALT
```

© Kovač, Basch, FER, Zagreb

#### <<< (kompletan listing s komentarima)

```
; glavni program
GLAVNI MOVE 10000, SP ;važno: inicijaliziraj SP !!!
; stavi vrijednost prvog parametra na stog
LOAD R0, (PRVI)
PUSH R0
; stavi vrijednost drugog parametra na stog
LOAD R0, (DRUGI)
PUSH R0
CALL NILI ;poziv potprograma
; uzmi rezultat sa stoga i spremi ga
POP R0
STORE R0, (REZ) ;spremanje rezultata
HALT
```

>>>

© Kovač, Basch, FER, Zagreb

211

### Potprogrami - Prijenos stogom

- Pokušajmo riješiti prethodni zadatak tako da se čuvaju stanja registara koje potprogram mijenja
- Već smo vidjeli da je najzgodnije **registre spremiti na stog**

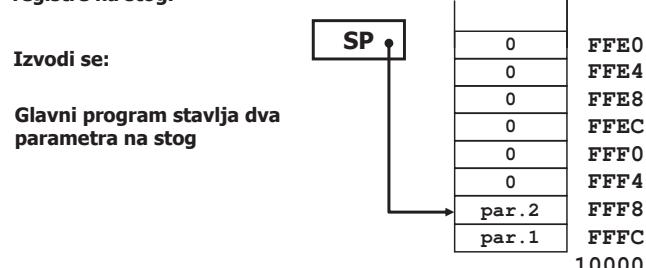
>>>

213

© Kovač, Basch, FER, Zagreb

### Potprogrami - Prijenos stogom

Promatramo stanje na stogu za potprogram koji bi spremao registre na stog.

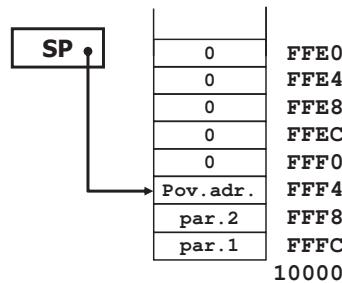


## Potprogrami - Prijenos stogom

Promatramo stanje na stogu za potprogram koji bi spremao registre na stog.

Izvodi se:

Glavni program poziva potprogram

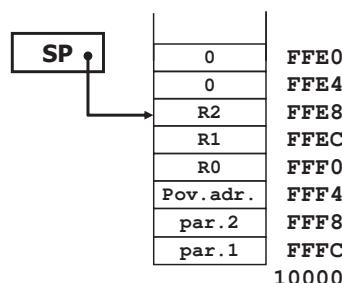


## Potprogrami - Prijenos stogom

Promatramo stanje na stogu za potprogram koji bi spremao registre na stog.



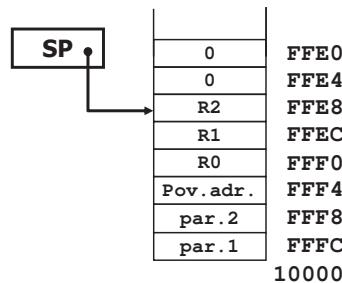
Kako dohvati parametre?



## Potprogrami - Prijenos stogom

Promatramo stanje na stogu za potprogram koji bi spremao registre na stog.

Jedna mogućnost (LOŠA!!!) je da presložimo podatke na stogu korištenjem fiksnih memorijskih lokacija (npr. tako da parametri dodu na vrh stoga)

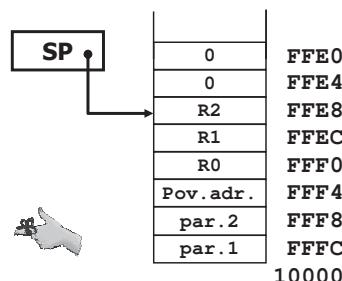


## Potprogrami - Prijenos stogom

Promatramo stanje na stogu za potprogram koji bi spremao registre na stog.

Rješenje je jednostavno:

Registarsko indirektno adresiranje s odmakom, pri čemu kao adresni register koristimo SP

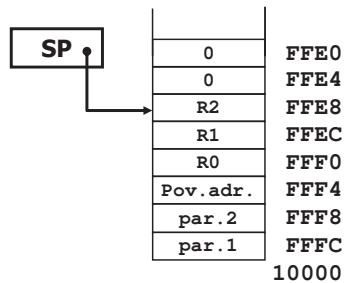


## Potprogrami - Prijenos stogom

Promatramo stanje na stogu za potprogram koji bi spremao registre na stog.

Izvodi se:

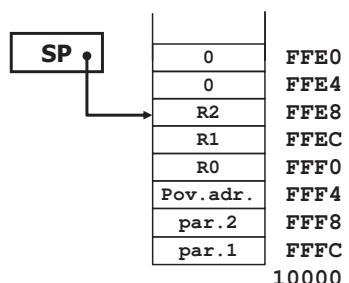
Potprogram spremi registre (koje će mijenjati) na stog  
(npr. tri registra R0, R1, R2)



## Potprogrami - Prijenos stogom

Promatramo stanje na stogu za potprogram koji bi spremao registre na stog.

Da smo prije spremanja registara dohvatali parametre, onda bi uništili registre prije nego ih spremimo:

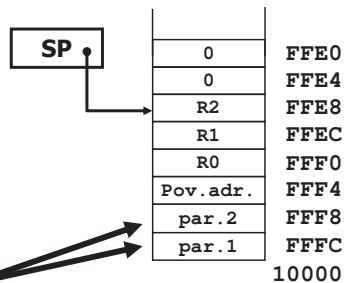


## Potprogrami - Prijenos stogom

Promatramo stanje na stogu za potprogram koji bi spremao registre na stog.

Rješenje bi bilo kad ne bi morali pristupati podacima na stogu samo pomoću PUSH i POP.

Kako možemo pristupiti podacima "ispod" vrha stoga, bez pomicanja vrha stoga?

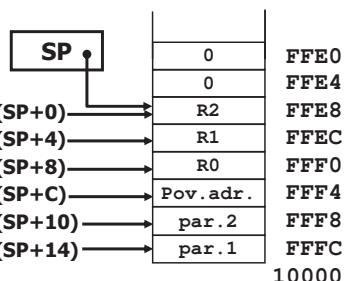


## Potprogrami - Prijenos stogom



Promatramo stanje na stogu za potprogram koji bi spremao registre na stog.

Adrese se zadaju ovako



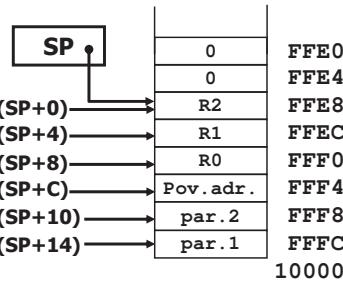
Naredbama LOAD i STORE možemo pristupati podacima "unutar" stoga.



Promatramo stanje na stogu za potprogram koji bi spremao registre na stog.

- Dok pišemo potprogram, točno znamo koliko on parametara ima i koliko registara mora spremiti na stog.

- Zato lako možemo izračunati odmake pojedinih parametara u odnosu na SP (vrh stoga).



## Potprogrami - Prijenos stogom

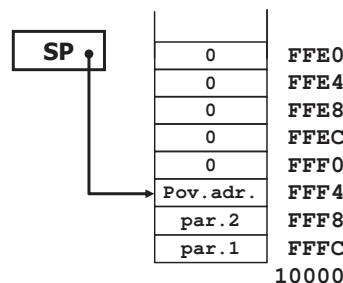
Promatramo stanje na stogu za potprogram koji bi spremao registre na stog.

Izvodi se:

Obnavljaju se registri R0, R1 i R2.

Povratak se lako ostavaruje sa RET

Ali, obnavljanjem R0 gubi se rezultat !!!



## Potprogrami - Prijenos stogom

### Primjer:

Napisati potprogram koji izračunava logičku operaciju NILI. Parametri se prenose stogom, a rezultat se vraća registrom R0. Glavni program iz memorije učitava dva podatka za koje računa NILI (pomoću potprograma) i rezultat spremi natrag u memoriju.

Potprogram mora čuvati vrijednosti svih registara koje mijenja.

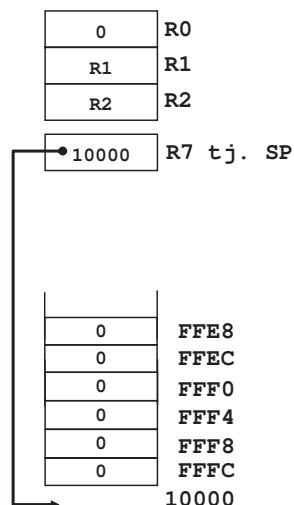
### Rješenje:

- U ovom rješenju potprogram mijenja registre R1 i R2 i spremi ih na stog (R0 se također mijenja, ali preko njega se ionako vraća povratna vrijednost)
- Parametrima se pristupa registarskim indirektnim adresiranjem s odmakom (korištenjem SP-a kao adresnog registra)

>>>

### <<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP ←
LOAD R0, (PRVI)
PUSH R0
LOAD R0, (DRUGI)
PUSH R0
CALL NILI
STORE R0, (REZ)
ADD SP, 8, SP
HALT
```

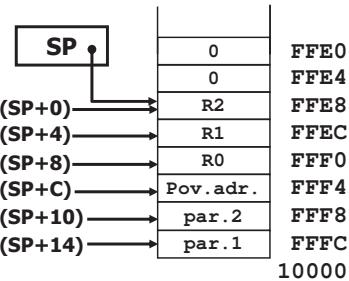


Promatramo stanje na stogu za potprogram koji bi spremao registre na stog.

Zadnji problem je povratna vrijednost.

Neka je upravo završeno izračunavanje rezultata potprograma i neka je rezultat npr. u registru R0.

Želimo se vratiti iz potprograma:



## Potprogrami - Prijenos stogom

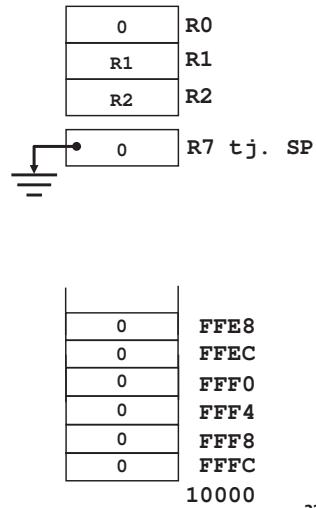


### Rješenje koje se koristi u praksi:

- Za povratak rezultata iz potprograma koristi se prijenos registrom, a ne stogom.
- Ovo rješenje je efikasno i jednostavno. Ograničenje je da se može vratiti samo jedan rezultat, ali je u praksi to obično dovoljno.
- Obično se odabire jedan registar koji svi potprogrami koriste za povratak vrijednosti, a na mjestu pozivanja se zna da u tom registru ne smije biti koristan podatak.

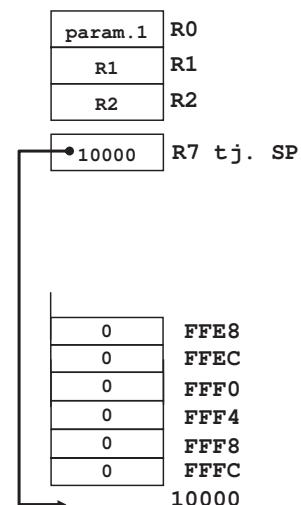
### <<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
PUSH R0
LOAD R0, (DRUGI)
PUSH R0
CALL NILI
STORE R0, (REZ)
ADD SP, 8, SP
HALT
```



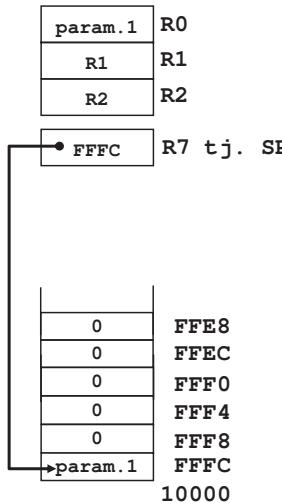
### <<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI) ←
PUSH R0
LOAD R0, (DRUGI)
PUSH R0
CALL NILI
STORE R0, (REZ)
ADD SP, 8, SP
HALT
```



<<< Izvođenje programa:

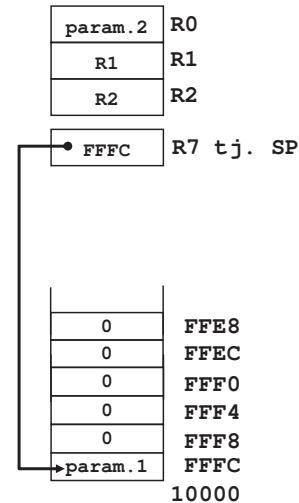
```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
PUSH R0
LOAD R0, (DRUGI)
PUSH R0
CALL NILI
STORE R0, (REZ)
ADD SP, 8, SP
HALT
```



© Kovač, Basch, FER, Zagreb

<<< Izvođenje programa:

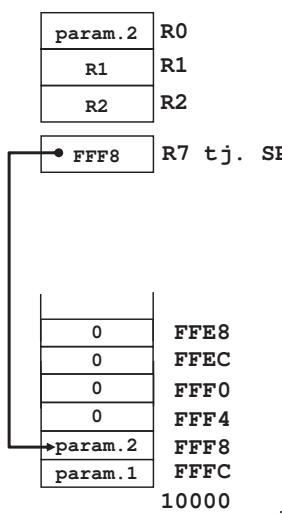
```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
PUSH R0
LOAD R0, (DRUGI)
PUSH R0
CALL NILI
STORE R0, (REZ)
ADD SP, 8, SP
HALT
```



© Kovač, Basch, FER, Zagreb 233

<<< Izvođenje programa:

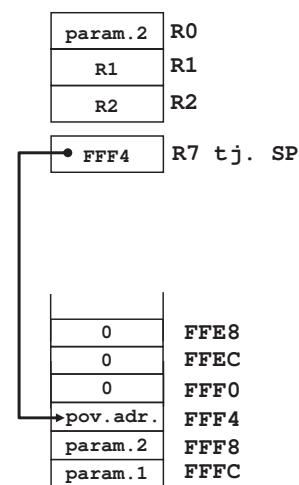
```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
PUSH R0
LOAD R0, (DRUGI)
PUSH R0
CALL NILI
STORE R0, (REZ)
ADD SP, 8, SP
HALT
```



© Kovač, Basch, FER, Zagreb

<<< Izvođenje programa:

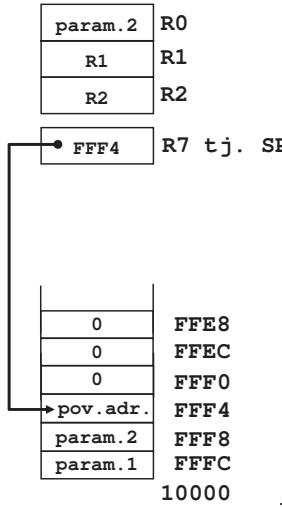
```
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI)
PUSH R0
LOAD R0, (DRUGI)
PUSH R0
CALL NILI
STORE R0, (REZ)
ADD SP, 8, SP
HALT
```



© Kovač, Basch, FER, Zagreb 235

<<< Izvođenje programa:

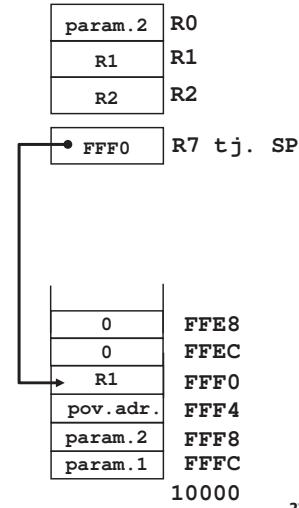
```
NILI PUSH R1
PUSH R2
LOAD R1, (SP+0C)
LOAD R2, (SP+10)
OR R1, R2, R0
XOR R0, -1, R0
POP R2
POP R1
RET
```



© Kovač, Basch, FER, Zagreb

<<< Izvođenje programa:

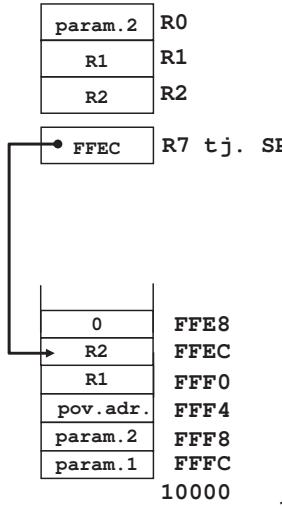
```
NILI PUSH R1
PUSH R2
LOAD R1, (SP+0C)
LOAD R2, (SP+10)
OR R1, R2, R0
XOR R0, -1, R0
POP R2
POP R1
RET
```



© Kovač, Basch, FER, Zagreb 237

<<< Izvođenje programa:

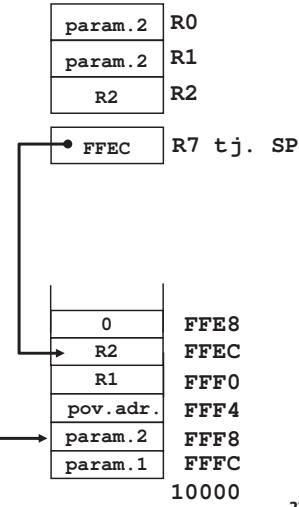
```
NILI PUSH R1
PUSH R2
LOAD R1, (SP+0C)
LOAD R2, (SP+10)
OR R1, R2, R0
XOR R0, -1, R0
POP R2
POP R1
RET
```



© Kovač, Basch, FER, Zagreb

<<< Izvođenje programa:

```
NILI PUSH R1
PUSH R2
LOAD R1, (SP+0C)
LOAD R2, (SP+10)
OR R1, R2, R0
XOR R0, -1, R0
POP R2
POP R1
RET
```



© Kovač, Basch, FER, Zagreb 239

<<< Izvođenje programa:

```

NILI PUSH R1
 PUSH R2

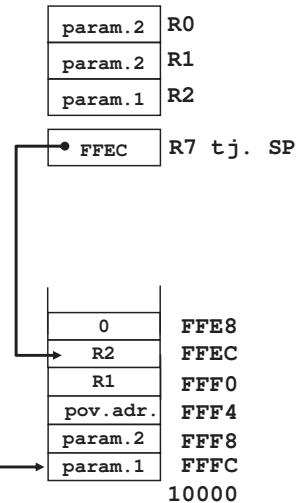
 LOAD R1, (SP+0C)
 LOAD R2, (SP+10) ←

 OR R1, R2, R0
 XOR R0, -1, R0

 POP R2
 POP R1

 RET

```



© Kovač, Basch, FER, Zagreb

<<< Izvođenje programa:

```

NILI PUSH R1
 PUSH R2

 LOAD R1, (SP+0C)
 LOAD R2, (SP+10) ←

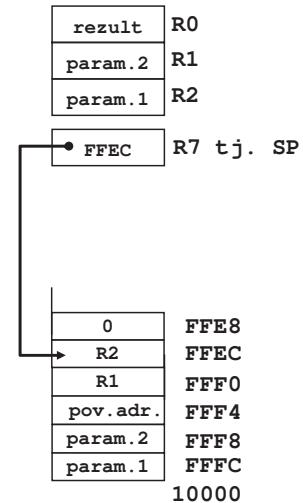
 OR R1, R2, R0
 XOR R0, -1, R0

 POP R2
 POP R1

 RET

```

© Kovač, Basch, FER, Zagreb



241

<<< Izvođenje programa:

```

NILI PUSH R1
 PUSH R2

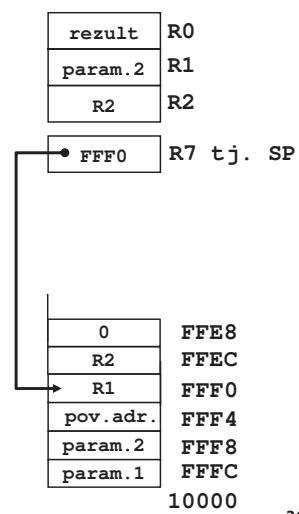
 LOAD R1, (SP+0C)
 LOAD R2, (SP+10)

 OR R1, R2, R0
 XOR R0, -1, R0

 POP R2
 POP R1

 RET

```



© Kovač, Basch, FER, Zagreb

<<< Izvođenje programa:

```

NILI PUSH R1
 PUSH R2

 LOAD R1, (SP+0C)
 LOAD R2, (SP+10)

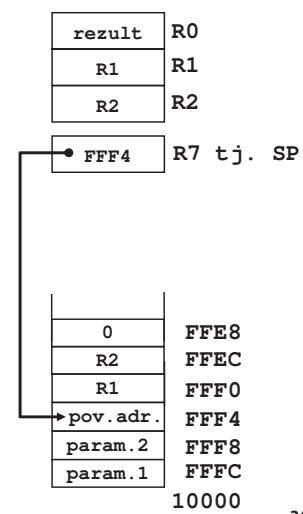
 OR R1, R2, R0
 XOR R0, -1, R0

 POP R2
 POP R1

 RET

```

© Kovač, Basch, FER, Zagreb



243

<<< Izvođenje programa:

```

NILI PUSH R1
 PUSH R2

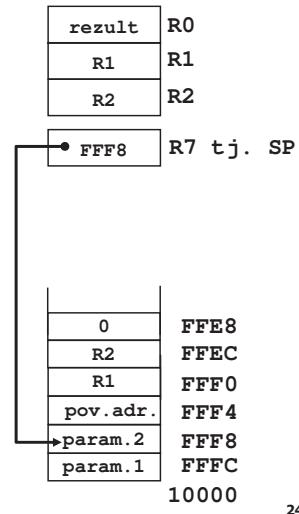
 LOAD R1, (SP+0C)
 LOAD R2, (SP+10)

 OR R1, R2, R0
 XOR R0, -1, R0

 POP R2
 POP R1

 RET

```



© Kovač, Basch, FER, Zagreb

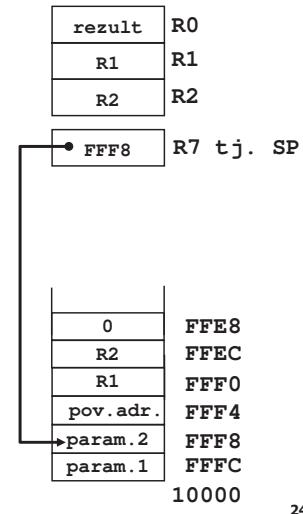
<<< Izvođenje programa:

```

GLAVNI MOVE 10000, SP
 LOAD R0, (PRVI)
 PUSH R0
 LOAD R0, (DRUGI)
 PUSH R0
 CALL NILI
 STORE R0, (REZ)
 ADD SP, 8, SP ←
 HALT

```

© Kovač, Basch, FER, Zagreb



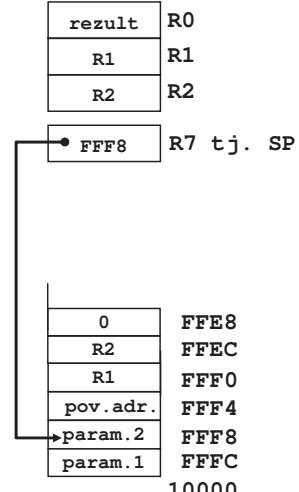
245

<<< Izvođenje programa:

```

GLAVNI MOVE 10000, SP
 LOAD R0, (PRVI)
 PUSH R0
 LOAD R0, (DRUGI)
 PUSH R0
 CALL NILI
 STORE R0, (REZ)
 ADD SP, 8, SP ←
 HALT

```



© Kovač, Basch, FER, Zagreb

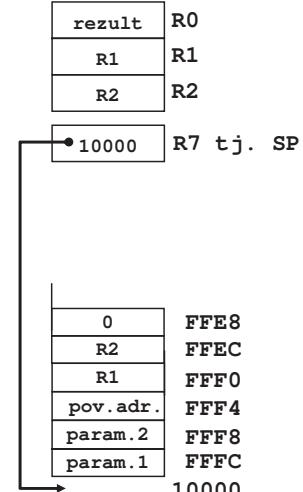
<<< Izvođenje programa:

```

GLAVNI MOVE 10000, SP
 LOAD R0, (PRVI)
 PUSH R0
 LOAD R0, (DRUGI)
 PUSH R0
 CALL NILI
 STORE R0, (REZ)
 ADD SP, 8, SP ←
 HALT

```

© Kovač, Basch, FER, Zagreb



247

glavni program uklanja parametre sa stoga

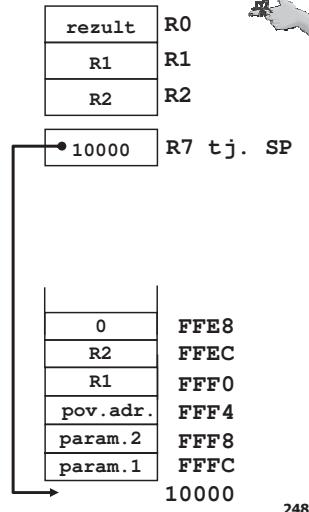
## Komentari:

- Za razliku od prethodnih primjera, gdje je potprogram pomoću naredaba POP "potrošio" svoje parametre, ovdje parametre uklanja pozivatelj.
- Ovo je "čišće" rješenje, jer je logičnije da je parametre sa stoga dužan ukloniti onaj tko ih je i stavio na stog.
- Umjesto niza naredaba POP, parametri se uklanjuju naredbom:

```
ADD SP, 8, SP
```

što je ne samo brže, nego dodatno čuva vrijednosti svih registara.

© Kovač, Basch, FER, Zagreb



<<< (kompletan listing s komentarima)

```
; glavni program
GLAVNI MOVE 10000, SP ; važno: inicijaliziraj SP !!!
LOAD R0, (PRVI) ; stavi vrijednost
PUSH R0 ; prvog parametra na stog
LOAD R0, (DRUGI) ; stavi vrijednost
PUSH R0 ; drugog parametra na stog
CALL NILI ; poziv potprograma
STORE R0, (REZ) ; spremi rezultat iz R0
ADD SP, 8, SP ; ukloni parametre sa stoga
HALT

; podatci i mjesto za rezultat
PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0
```

>>>

© Kovač, Basch, FER, Zagreb

249

<<<

```
; potprogram NILI
NILI PUSH R1 ; Spremanje
PUSH R2 ; registara.

LOAD R1, (SP+0C) ; Čitanje parametara
LOAD R2, (SP+10) ; u registre R1 i R2

OR R1, R2, R0 ; Izračunavanje
XOR R0, -1, R0 ; rezultata.

POP R2 ; Obnovi
POP R1 ; registre.

RET
```

>>>

© Kovač, Basch, FER, Zagreb

250

- Do sada nismo vidjeli kako u asembleru ostvariti **lokalne varijable**
- Lokalne varijable imaju sljedeća svojstva:
  - za svaki poziv potprograma postoje vlastite lokalne varijable
  - vidljive su samo unutar potprograma u kojem su definirane
  - stvaraju se prilikom poziva potprograma
  - nestaju prilikom povratka iz potprograma
- Vidimo da su lokalne varijable po svemu slične parametrima potprograma. Jedina je razlika u početnoj vrijednosti koja se za parametar definira od strane pozivatelja potprograma
- Dakle, prirodno je rješenje da se i lokalne varijable čuvaju na stogu, ili točnije u okviru stoga

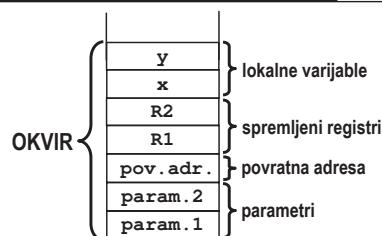
>>>

© Kovač, Basch, FER, Zagreb

252

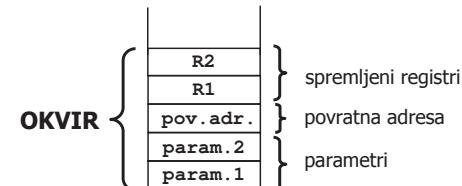
## Rekapitulacija okvira stoga

- Okvir stoga sadrži:
  - parametre
  - povratnu adresu
  - spremljene registre
  - lokalne varijable
- Način rada s okvirom:
  - Glavni program:
    - stavlja parametre
    - stavlja povratnu adresu (CALL)
    - uklanja parametre
  - Potprogram:
    - spremi i obnavlja registre
    - stvara i uklanja lokalne varijable
    - uklanja povratnu adresu (RET)



## Komentari:

- Način rada s parametrima i način vraćanja rezultata pokazan u ovom primjeru vrlo je sličan stvarnim potprogramima dobivenim prevodenjem viših programskih jezika u asembler.
- Razlog za ovaku organizaciju je njena praktičnost i efikasnost te općenitost koja omogućuje korištenje rekurzivnih i nerekurzivnih potprograma uz čuvanje stanja svih registara.
- Podatci na stogu su uniformno organizirani za svaki potprogram i takav **niz podataka za jedan potprogram** naziva se **okvir stoga** ili kraće okvir (frame, stack frame, activation record). Svaki potprogram ima svoj okvir.

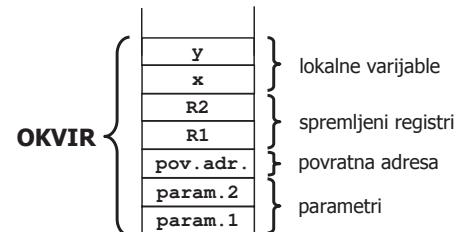


© Kovač, Basch, FER, Zagreb

251

<<<

- Potpunija verzija stogovnog okvira uključuje i lokalne varijable
- Lokalne varijable se stavljaju na stog nakon ulaska u potprogram, a mogu se staviti "ispod" ili "iznad" spremlijenih registara. Ovdje je odabранo da se stave "iznad" njih:



## Važna napomena:

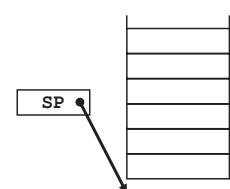
Prevoditelj u stvarnosti obično pokušava staviti lokalne varijable u registre, a tek kad se oni napune stavlja lokalne varijable na stog

© Kovač, Basch, FER, Zagreb

253

## Rekapitulacija rada s okvirom stoga:

Pozivatelj: \_\_\_\_\_ Potprogram: \_\_\_\_\_ Okvir: \_\_\_\_\_

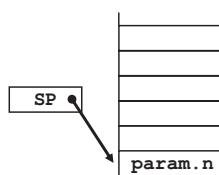


### Rekapitulacija rada s okvirom stoga:



Pozivatelj: Potprogram: Okvir:

Stavi parametre na stog



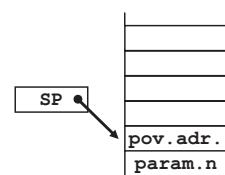
### Rekapitulacija rada s okvirom stoga:



Pozivatelj: Potprogram: Okvir:

Stavi parametre na stog

Pozovi potprogram



© Kovač, Basch, FER, Zagreb

256

### Rekapitulacija rada s okvirom stoga:

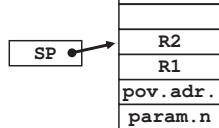


Pozivatelj: Potprogram: Okvir:

Stavi parametre na stog

Pozovi potprogram

Spremi registre



© Kovač, Basch, FER, Zagreb

257

### Rekapitulacija rada s okvirom stoga:



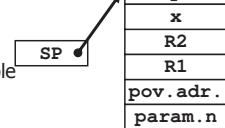
Pozivatelj: Potprogram: Okvir:

Stavi parametre na stog

Pozovi potprogram

Spremi registre

Stvori lokalne varijable



© Kovač, Basch, FER, Zagreb

258

### Rekapitulacija rada s okvirom stoga:



Pozivatelj: Potprogram: Okvir:

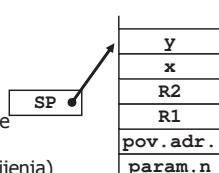
Stavi parametre na stog

Pozovi potprogram

Spremi registre

Stvori lokalne varijable

Izvođenje (okvir se ne mijenja)  
(ali se mogu mijenjati podatci u okviru,  
npr. parametri i lokalne varijable)



© Kovač, Basch, FER, Zagreb

259

### Rekapitulacija rada s okvirom stoga:



Pozivatelj: Potprogram: Okvir:

Stavi parametre na stog

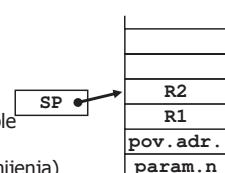
Pozovi potprogram

Spremi registre

Stvori lokalne varijable

Izvođenje (okvir se ne mijenja)

Povratak: ukloni lokalne



© Kovač, Basch, FER, Zagreb

260

### Rekapitulacija rada s okvirom stoga:



Pozivatelj: Potprogram: Okvir:

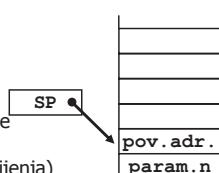
Stavi parametre na stog

Pozovi potprogram

Spremi registre

Stvori lokalne varijable

Izvođenje (okvir se ne mijenja)



© Kovač, Basch, FER, Zagreb

262

© Kovač, Basch, FER, Zagreb

261

### Rekapitulacija rada s okvirom stoga:



Pozivatelj: Potprogram: Okvir:

Stavi parametre na stog

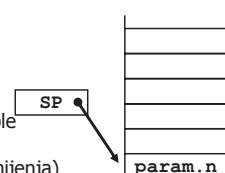
Pozovi potprogram (CALL)

Spremi registre

Stvori lokalne varijable

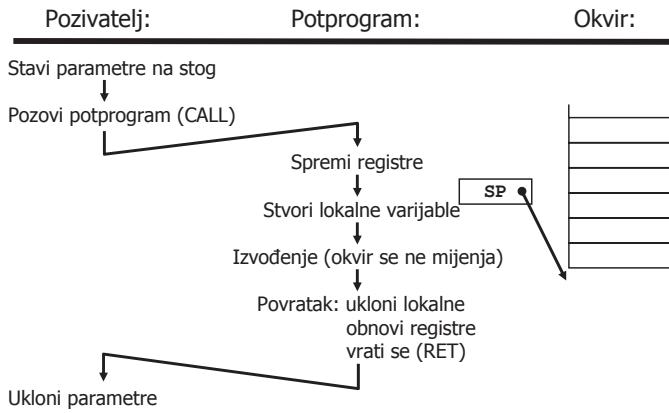
Izvođenje (okvir se ne mijenja)

Povratak: ukloni lokalne  
obnovi registre  
vrati se (RET)



263

© Kovač, Basch, FER, Zagreb



## Potprogrami - Prijenos stogom



- Rekursivni potprogrami su oni koji mogu pozvati sami sebe
  - (izravno, ili neizravno - preko drugih potprograma)
- Budući da imamo višestruki poziv istog potprograma, to je isto kao da je rekursivni potprogram istovremeno "više puta aktiviran"
  - Zato se parametri ne mogu prenosi registrima i fiksnim lokacijama (npr. prvi poziv bi napunio vrijednosti u parametre, a već bi drugi poziv prepisao preko njih svoje vrijednosti, čime bi prve vrijednosti za prvi poziv bile izgubljene)
- Zato rekursivni potprogrami koriste prijenos stogom i okvir stoga
- Uočite da povratna vrijednost nije problem i za njeno vraćanje se koristi prijenos registrom

```

<<<
; potprogram fakt (n)
FAKT PUSH R1 } Spremi registre
 PUSH R2
 LOAD R0, (SP+0C)
 CMP R0, 1
 JR_EQ VRATISE
 SUB R0, 1, R0
 PUSH R0
 CALL FAKT
 ADD SP, 4, SP
 LOAD R1, (SP+0C)
 MOVE 0, R2
 ADD R0, R2, R2
 SUB R1, 1, R1
 JR_NZ MNOZI
 MOVE R2, R0
VRATISE POP R2
 POP R1
 RET
>>>

```

## Potprogrami - Rekurzija



### Primjer:

Treba napisati rekursivni potprogram za računanje niza Fibonaccijevih brojeva:

$$\begin{aligned}
 Fib(1) &= 1 \\
 Fib(2) &= 1 \\
 Fib(n) &= Fib(n-1) + Fib(n-2) \quad \text{za } n=3,4, \dots
 \end{aligned}$$

Parametar se prenosi stogom, a rezultat se vraća registrom R0. Glavni program treba izračunati  $Fib(8)$ .

### Rješenje:

- Radi lakšeg objašnjenja, pokažimo prvo kako bi rješenje moglo izgledati u programskom jeziku C

&gt;&gt;&gt;

## Rekursivni potprogrami

## Potprogrami - Rekurzija

### Primjer:

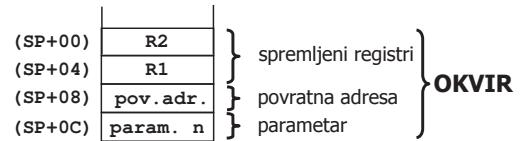
Treba napisati rekursivni potprogram za računanje faktorijela:

$$\begin{aligned}
 fakt(1) &= 1 \\
 fakt(n) &= fakt(n-1)*n \quad \text{za } n=2, 3, 4, \dots
 \end{aligned}$$

Parametar se prenosi stogom, a rezultat se vraća registrom R0.

### Rješenje:

Pokažimo prvo kako će izgledati okvir stoga:



&lt;&lt;&lt;

Pokažimo još samo kako bi mogao izgledati glavni program koji poziva fakt(3) i sprema rezultat na memoriju REZULT.

```

GLAVNI MOVE 10000, SP
 MOVE 3, R0
 PUSH R0
 CALL FAKT
 STORE R0, (REZULT)
 ADD SP, 4, SP
 HALT

```

```
REZULT DW 0
```

&lt;&lt;&lt;

```

int fib (int n) {
 int x, y; // lokalne varijable za medurezultate
 if (n == 1 || n == 2) // Fib(1) i Fib(2) = 1
 return (1);
 x = fib (n-1);
 y = fib (n-2); } // Fib(n) = Fib(n-1) + Fib(n-2)
 return (x+y);
}

main () {
 int rez;
 rez = fib (8);
}

```

&gt;&gt;&gt;

&lt;&lt;&lt;



Mogući prijevod C-programa u asembler:



Rješenje u C-u se moglo napisati i nešto kraće:

```
int fib (int n) {
 if (n == 1 || n == 2)
 return (1);
 return (fib (n-1) + fib (n-2))
}
```

- Ovdje nema lokalnih varijabli x i y. Međutim, treba voditi računa da rezultat od fib(n-1) mora biti negdje pohranjen dok se izračunava fib(n-2).

- Zato će C-prevodilac stvoriti asemblerски program koji je sličniji verziji s prethodnog slajda, iako je to za programera u C-u nevidljivo.

&gt;&gt;&gt;

&gt;&gt;&gt;

© Kovač, Basch, FER, Zagreb

272

&lt;&lt;&lt; ; potprogram FIB



```
FIB PUSH R1 ; Spremanje
 PUSH R2 ; registara na stog

 ; Stvaranje lokalnih varijabli x i y
 SUB SP, 8, SP

 ; if(n==1 || n==2) return (1);

 MOVE 1, R0 ; Pripremi rez. za slučaj povratka

 LOAD R1, (SP+14) ; dohvati parametar n
 CMP R1, 1 ; je li n==1 ?
 JR_EQ VRATI_SE ; Da: idi na dio za povratak

 CMP R1, 2 ; Ne: ispitaj je li n==2 ?
 JR_EQ VRATI_SE ; Da: idi na dio za povratak
```

&gt;&gt;&gt;

&gt;&gt;&gt;

© Kovač, Basch, FER, Zagreb

273

&lt;&lt;&lt; ; return ( x + y )



```
LOAD R1, (SP+4) ; dohvati x
LOAD R2, (SP+0) ; dohvati y
ADD R1, R2, R0 ; izračunaj povratnu vrijednost

VRATI_SE ADD SP, 8, SP ; ukloni x i y sa stoga

POP R2 ; Obnovi sadržaje
POP R1 ; spremlijenih registara.

RET ; povratak iz FIB
```

© Kovač, Basch, FER, Zagreb

273

&lt;&lt;&lt; ; n nije ni 1 ni 2 - nastavak izvodenja



```
; x = fib(n-1)

LOAD R1, (SP+14) ; dohvati parametar n
SUB R1, 1, R1 ; Oduzmi n-1 i stavi na stog
PUSH R1 ; za prvi rekursivni poziv.

CALL FIB ; pozovi fib(n-1)
ADD SP, 4, SP ; ukloni n-1 sa stoga
STORE R0, (SP+4) ; spremi rezultat u x

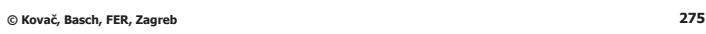
; y = fib(n-2)

LOAD R1, (SP+14) ; dohvati parametar n
SUB R1, 2, R1 ; Oduzmi n-2 i stavi na stog
PUSH R1 ; za drugi rekursivni poziv.

CALL FIB ; pozovi fib(n-2)
ADD SP, 4, SP ; ukloni n-2 sa stoga
STORE R0, (SP+0) ; spremi rezultat u y
```

© Kovač, Basch, FER, Zagreb

275



- Pokažimo ponašanje okvira na stogu ako je u glavnom programu pozvano Fib(4).
- Nećemo pratiti pojedine podatke u okvirima na stogu već promatramo pojedine **okvire kao cjeline**.



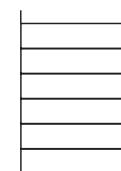
okviri na stogu

© Kovač, Basch, FER, Zagreb

276

main

izvodi se main



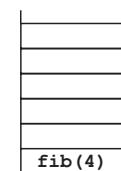
okviri na stogu

© Kovač, Basch, FER, Zagreb

277

main
main → fib(4)

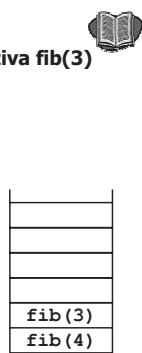
main poziva fib(4)



okviri na stogu

main  
main → fib(4)  
main      fib(4) → fib(3)

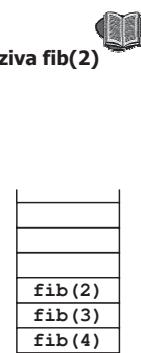
### **fib(4) poziva fib(3)**



okviri na stogu

main  
main → fib(4)  
main      fib(4) → fib(3)  
main      fib(3) → fib(2)  
main      fib(2)

### **fib(3) poziva fib(2)**

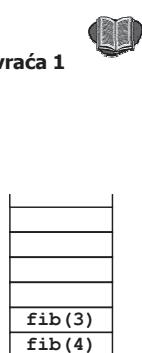


okviri na stogu

© Kovač, Basch, FER, Zagreb

main  
main → fib(4)  
main      fib(4) → fib(3)  
main      fib(4)      fib(3) → fib(2)  
main      fib(4)      fib(2) ← 1 fib(2)=1  
main      fib(4)

### **fib(2) vraća 1**



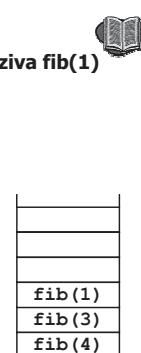
okviri na stogu

280

© Kovač, Basch, FER, Zagreb

main  
main → fib(4)  
main      fib(4) → fib(3)  
main      fib(4)      fib(3) → fib(2)  
main      fib(4)      fib(2) ← 1 fib(2)=1  
main      fib(4)      fib(3) → fib(1)

### **fib(3) poziva fib(1)**



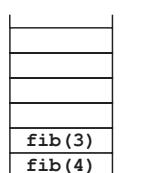
okviri na stogu

281

© Kovač, Basch, FER, Zagreb

main  
main → fib(4)  
main      fib(4) → fib(3)  
main      fib(4)      fib(3) → fib(2)  
main      fib(4)      fib(2) ← 1 fib(2)=1  
main      fib(4)      fib(3) → fib(1)  
main      fib(4)      fib(3) ← 1 fib(1)=1

### **fib(1) vraća 1**



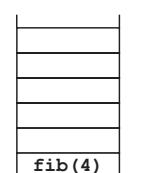
okviri na stogu

282

© Kovač, Basch, FER, Zagreb

main  
main → fib(4)  
main      fib(4) → fib(3)  
main      fib(4)      fib(3) → fib(2)  
main      fib(4)      fib(2) ← 1 fib(2)=1  
main      fib(4)      fib(3) → fib(1)  
main      fib(4)      fib(3) ← 1 fib(1)=1  
main      fib(4) ← 2 fib(3)=1+1

### **fib(3) vraća fib(2)+fib(1)**



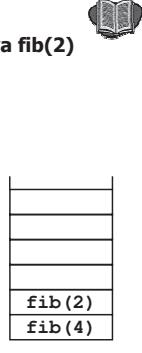
okviri na stogu

283

© Kovač, Basch, FER, Zagreb

main  
main → fib(4)  
main      fib(4) → fib(3)  
main      fib(4)      fib(3) → fib(2)  
main      fib(4)      fib(2) ← 1 fib(2)=1  
main      fib(4)      fib(3) → fib(1)  
main      fib(4)      fib(3) ← 1 fib(1)=1  
main      fib(4) ← 2 fib(3)=1+1  
main      fib(4) → fib(2)

### **fib(4) poziva fib(2)**



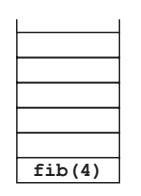
okviri na stogu

284

© Kovač, Basch, FER, Zagreb

main  
main → fib(4)  
main      fib(4) → fib(3)  
main      fib(4)      fib(3) → fib(2)  
main      fib(4)      fib(2) ← 1 fib(2)=1  
main      fib(4)      fib(3) → fib(1)  
main      fib(4)      fib(3) ← 1 fib(1)=1  
main      fib(4) ← 2 fib(3)=1+1  
main      fib(4) → fib(2)  
main      fib(4) ← 1 fib(2)=1

### **fib(2) vraća 1**



okviri na stogu

285

© Kovač, Basch, FER, Zagreb

286

© Kovač, Basch, FER, Zagreb

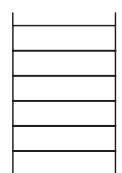
287

```

main
main → fib(4)
main fib(4) → fib(3)
main fib(4) fib(3) → fib(2)
main fib(4) fib(3) ← fib(2)=1
main fib(4) fib(3) → fib(1)
main fib(4) fib(3) ← fib(1)=1
main fib(4) ← fib(3)=1+1
main fib(4) → fib(2)
main fib(4) ← fib(2)=1
main ← 3 fib(4)=2+1
main

```

**fib(4) vraća fib(3)+fib(2)**



okviri na stogu

```

main
main → fib(4)
main fib(4) → fib(3)
main fib(4) fib(3) → fib(2)
main fib(4) fib(3) ← fib(2)=1
main fib(4) fib(3) → fib(1)
main fib(4) fib(3) ← fib(1)=1
main fib(4) ← fib(3)=1+1
main fib(4) → fib(2)
main fib(4) ← fib(2)=1
main ← 3 fib(4)=2+1
main

```

**nastavlja se izvoditi main**

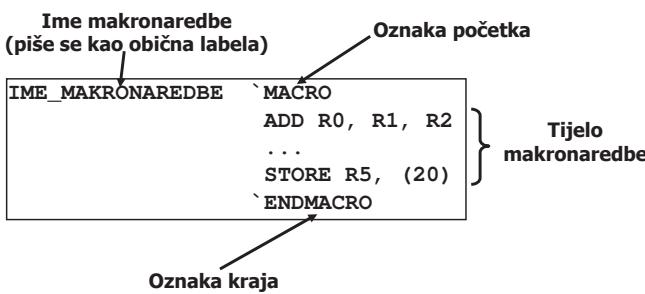


okviri na stogu

## Makronaredbe

### Makronaredbe

- Programer definira makronaredbu, slično kao što definira i potprogram. Definicija makronaredbe općenito izgleda ovako:



### Makronaredbe - Primjer

#### Primjer:

Napisati makronaredbu koja izračunava logičku operaciju NILI između sadržaja registara R0 i R1 i vraća rezultat u registru R2. Napisati i glavni program koji će pozvati makronaredbu za dva podatka iz memorije i rezultat također upisati u memoriju.

#### Rješenje:

; DEFINICIJA MAKRONAREDBE

```

NILI `MACRO
 OR R0, R1, R2
 XOR R2, -1, R2
 ENDMACRO

```

>>>

### Makronaredbe - Način prevođenja

- Asembleri koji podržavaju makronaredbe moraju biti troprolazni ili četveroprolazni i nazivaju se makroasemblerima
- U troprolaznom asembleru se svaka definicija makronaredbe mora nalaziti **ISPRED** njenog pozivanja
- U četveroprolaznom asembleru se definicija makronaredbe smije nalaziti **IZA** njenog pozivanja
- Objasnimo kako rade ove dvije vrste asemblera . . .

**Napomena:** Poziv se ostvaruje navođenjem imena makronaredbe. Poziv **nije naredba procesora**, već je sličniji pseudonaredbi asemblerorskog prevoditelja.

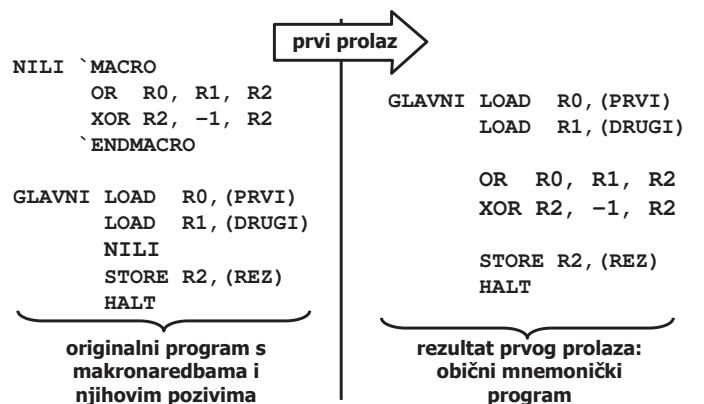
Prvi prolaz:

- Asembler čita redak po redak datoteke:
  - Ako nađe na definiciju makronaredbe, onda u tablici makronaredbi zapamti njeno ime i tijelo (slično kao što dvoprolazni asembler nailaskom na labelu u tablici labela pamti njeno ime i vrijednost)
  - Ako nađe na poziv makronaredbe, zamjenjuje ga njenim tijelom koje je zapamćeno u tablici (ovo je uvijek moguće napraviti zbog obaveznog redoslijeda u kojem definicija uvijek prethodi pozivu makronaredbe)
  - Prevoditelj ne mijenja ostale retke mnemoničkog programa
- Na kraju prvog prolaska, mnemonička datoteka više nema ni definicija makronaredaba ni poziva makronaredaba, već sadrži samo obične naredbe procesora i pseudonaredbe

>>>

## Makronaredbe - Troprolazni asembler

<<< Kako izgleda prevodenje prethodnog primjera?



## Makronaredbe - Četveroprolazni asembler

<<<

Drugi prolaz:

- Drugi prolaz zamjenjuje sve pozive makronaredba njihovim tijelima što daje obični mnemonički program

Treći i četvrti prolaz:

- Treći i četvrti prolaz u potpunosti odgovaraju prvom i drugom prolazu običnog dvoprolaznog asemblera (odnosno odgovaraju radu drugog i trećeg prolaza troprolaznog asemblera)

>>>

## Makronaredbe - Parametri

Primjer:

Napišite makronaredbu koja će izračunati logičku operaciju NILI. Uzlazni podaci i rezultat mogu biti smješteni na bilo kojim memorijskim lokacijama koje se zadaju kao parametri makronaredbe.

Napišite glavni program koji će izračunati NILI između podataka s lokacija PRVI i DRUGI, a rezultat će spremiti na REZULT.

Rješenje:

na sljedećem slajdu

>>>

<<<

Drugi i treći prolaz:

- Budući da je rezultat prvog prolaska obični mnemonički program, drugi i treći prolaz u potpunosti odgovaraju radu običnog dvoprolaznog asemblera:

- Drugi prolaz: ekvivalentan prvom prolasku dvoprolaznog asemblera
- Treći prolaz: ekvivalentan drugom prolasku dvoprolaznog asemblera

>>>

## Makronaredbe - Četveroprolazni asembler

Četveroprolazni asembler dozvoljava da poziv makronaredbe prethodi njenoj definiciji. Zato je potreban dodatni prolaz (kao što je u simboličkom dvoprolaznom asembleru potreban dodatni prolaz zbog labela koje se koriste prije nego što su definirane)

Prvi prolaz:

- Asembler čita redak po redak datoteke:
  - Ako nađe na definiciju makronaredbe, onda u tablici makronaredbi zapamti njeno ime i tijelo (slično kao što dvoprolazni asembler nailaskom na labelu u tablici labela pamti njeno ime i vrijednost)
  - Prevoditelj ne mijenja ostale retke mnemoničkog programa
- Na kraju prvog prolaska, mnemonička datoteka više nema ni definicija makronaredaba, ali sadrži pozive makronaredaba.

>>>

## Makronaredbe - Parametri

- Jedna od mogućnosti koju makroasembleri obično nude je i korištenje **parametara makronaredaba**
- Parametri se obično navode kao lista imena iza 'MACRO'
- Unutar tijela se parametri koriste na bilo kojim mjestima u naredbama
- Kod poziva makronaredbe programer zadaje argumente, tj. navodi npr. registre, adrese i sl. Ovi argumenti zamjenjuju parametre prilikom proširivanja makronaredbe.
- Ovisno o mjestima korištenja parametara, programeru je pri pozivanju ograničena njihova upotreba

```

; DEFINICIJA MAKRONAREDBE
NILI `MACRO P1, P2, REZ
 LOAD R0, (P1)
 LOAD R1, (P2)
 OR R0, R1, R2
 XOR R2, -1, R2
 STORE R2, (REZ)
`ENDMACRO

; GLAVNI PROGRAM
NILI PRVI, DRUGI, REZULT ;;; POZIV
HALT

; PODATCI I MJESTO ZA REZULTAT
PRVI DW 81282C34
DRUGI DW 29A82855
REZULT DW 0

```

>>>

Nakon prvog prolaza prevođenja, glavni program izgleda ovako:

```

LOAD R0, (PRVI)
LOAD R1, (DRUGI)
OR R0, R1, R2
XOR R2, -1, R2
STORE R2, (REZULT)

NILI PRVI, DRUGI, REZULT
HALT

PRVI DW 81282C34
DRUGI DW 29A82855
REZULT DW 0

```

**Napomena:** Mjesto upotrebe parametara u tijelu makronaredbe ograničava argumente koje možemo slati prilikom poziva. U ovom primjeru kao argumente možemo navoditi samo adrese zadane absolutnim ili simboličkim adresiranjem.

Dodatao se smije (iako to nije bila namjera pri pisanju makronaredbe) kao argument poslati i registar opće namjene, jer se u naredbama STORE i LOAD u zagradama smije koristiti i indirektno registarsko adresiranje s odmakom (odmak će u ovom slučaju biti 0)

```

NILI `MACRO P1, P2, REZ
LOAD R0, (P1)
LOAD R1, (P2)
OR R0, R1, R2
XOR R2, -1, R2
STORE R2, (REZ)
`ENDMACRO

```

Budući da se ovoj makronaredbi smiju poslati registri kao argumenti, što bi se dogodilo da se pozove ovako:

```
NILI PRVI, DRUGI, R0
```

Zašto makronaredba ne bi radila ispravno?

```

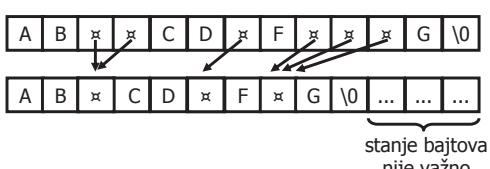
NILI `MACRO P1, P2, REZ
LOAD R0, (P1)
LOAD R1, (P2)
OR R0, R1, R2
XOR R2, -1, R2
STORE R2, (REZ)
`ENDMACRO

```

## Ostali primjeri

### Primjer:

Napisati potprogram SAZMI kojemu se preko R1 predaje adresa znakovnog niza u memoriji. Znakovi su zapisani ASCII kodom u bajtovima. Potprogram treba u znakovnom nizu sva uzastopna pojavljivanja razmaka sažeti u jednostrukе razmake. Niz je zaključen ASCII-znakom NUL.



- "Slova" (žuti) se kopiraju
- Razmaci označeni sa \* (zeleni) se sažimaju

- Makronaredbe općenito troše više memorije jer njihovo tijelo u memoriji postoji u više primjeraka (onoliko koliko ima poziva). Potprogrami se u memoriji nalaze samo u jednom primjerku.
- Potprogrami su sporiji jer se troši vrijeme na njihovo pozivanje i povratak, a također dio vremena troši i prijenos parametara i povratne vrijednosti. Makronaredbe se ne pozivaju nego se jednostavno izvode na mjestu na kojem je to potrebno.
- Što odabrat? Ovisno o tome što je kritično u pojedinom dijelu programa - brzina ili zauzeće memorije.

**Napomena:** Uočite da ovako definirana makronaredba mijenja sadržaje registara R0, R1 i R2 što nije poželjno (kao što nije bilo ni kod potprograma).

Registri se slično potprogramima mogu spremati na stog (ali i na fiksne lokacije jer se makronaredbe ne mogu pozivati rekursivno\*)

```

NILI `MACRO P1, P2, REZ
LOAD R0, (P1) ← mijenja R0
LOAD R1, (P2) ← mijenja R1
OR R0, R1, R2 ← mijenja R2
XOR R2, -1, R2
STORE R2, (REZ)
`ENDMACRO

```

\* Neki makroasembleri dozvoljavaju rekursivno i uvjetno pozivanje makronaredaba, ali to prelazi opseg gradiva na ovom predmetu

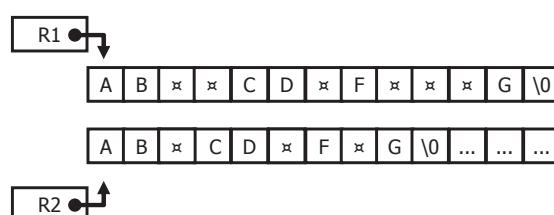
## FRISC - programiranje u asembleru

## Ostali primjeri

### Rješenje:

Upotrijebiti dva pokazivača:

- jedan čita znakove izvornog niza (R1)
- drugi pokazuje mjesto na koji se kopira znak izvornog niza (R2)





<<< ; TRENUTAČNI ZNAK JE RAZMAK

```
; JE LI PRETHODNI BIO RAZMAK ILI NIJE
RAZMAK CMP R0, RAZ
JR_EQ RAZMAK_PA_RAZMAK
```

```
SLOVO_PA_RAZMAK
MOVE RAZ, R0 ; stanje = razmak
STOREB R3, (R2) ; Kopiraj znak
ADD R1, 1, R1 ; Pomakni R1 i R2
ADD R2, 1, R2
JR PETLJA
```

```
RAZMAK_PA_RAZMAK
ADD R1, 1, R1 ; Pomakni R1
JR PETLJA
```

>>>

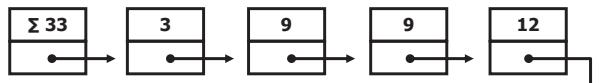
© Kovač, Basch, FER, Zagreb

320

## Ostali primjeri

### Primjer:

Jednostruko povezana lista ima čvorove koji u memoriji zauzimaju dvije 32-bitne riječi: prva riječ sadrži NBC-broj (koji predstavlja vrijednost čvora), a druga riječ je pokazivač na sljedeći čvor liste (tj. sadrži adresu sljedećeg čvora). Čvorovi su sortirani prema svojoj vrijednosti.



Prvi čvor liste ima specijalno značenje i u njemu se pamti zbroj svih vrijednosti preostalih čvorova liste. Prvi čvor je uvijek prisutan, bez obzira postoje li ostali čvorovi. Prepostavka je da u zbroju nikada neće doći do prekoračenja opsega.

Zadnji čvor u listi prepoznaje se po NULL-pokazivaču (tj. lokacija s pokazivačem sljedećeg čvora sadrži ništicu).

>>>

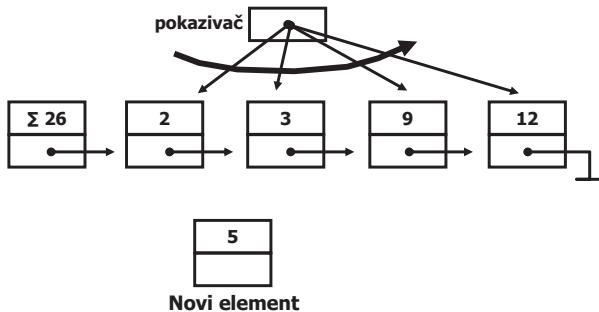
322

© Kovač, Basch, FER, Zagreb

## Ostali primjeri

<<< Idejno rješenje:

- Lista se pretražuje dok se ne nađe mjesto za ubacivanje.
- Jednim pokazivačem krećemo se po čvorovima koje ispitujemo.



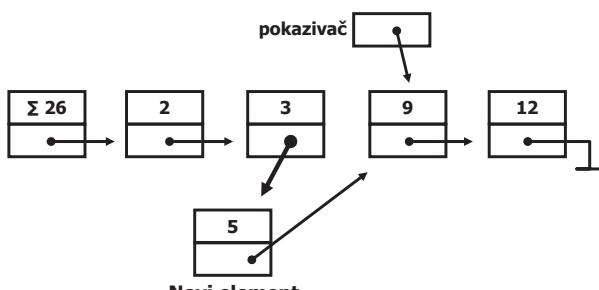
© Kovač, Basch, FER, Zagreb

324

## Ostali primjeri

<<< Idejno rješenje:

- Budući da se čvor mora ubaciti **ISPRED** čvora na kojeg pokazuje pokazivač, onda ne možemo dohvatiti prethodni čvor u kojem treba promjeniti pokazivač na sljedeći (plava strelica)



© Kovač, Basch, FER, Zagreb

<<<

```
KRAJ STOREB R3, (R2) ; Kopiraj NUL-znak
POP R3 ; Obnovi registre
POP R2
POP R1
POP R0
RET
```

; definiranje "konstanti"

```
SLO `EQU 0
RAZ `EQU 1
```

© Kovač, Basch, FER, Zagreb

321

## Ostali primjeri

<<<

Treba napisati potprogram UBACI koji ubacuje novi čvor u postojeću sortiranu listu tako da ona ostane sortirana.

Parametri potprograma su adresa prvog elementa liste i adresa novog čvora. Parametri se šalju preko stoga.

Povratna vrijednost je novi zbroj iz prvoga čvora liste. Vrijednost se vraća pomoću R0.

### Rješenje:

Potprogram će čuvati vrijednosti registara, a parametre će sa stoga uklanjati glavni program.

>>>

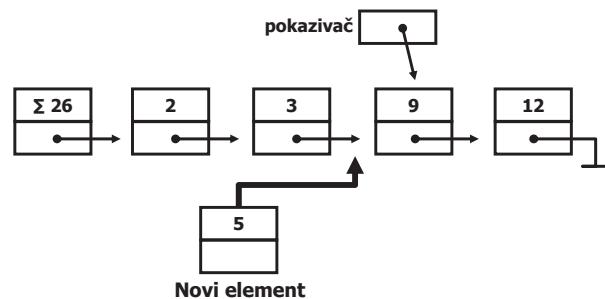
© Kovač, Basch, FER, Zagreb

323

## Ostali primjeri

<<< Idejno rješenje:

- Mjesto za ubacivanje je **ISPRED** prvog čvora koji ima veću ili jednaku vrijednost novom čvoru.



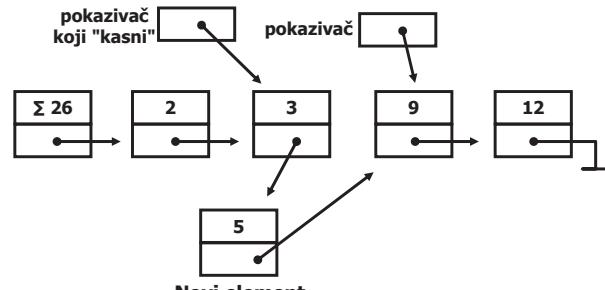
© Kovač, Basch, FER, Zagreb

325

## Ostali primjeri

<<< Idejno rješenje:

- Zato trebamo još jedan pokazivač koji će uvijek "kasniti" za jedan čvor u odnosu na pokazivač ispitivanog čvora, tj. pokazivač će pokazivati jedan čvor ispred onog čvora kojeg ispitujemo



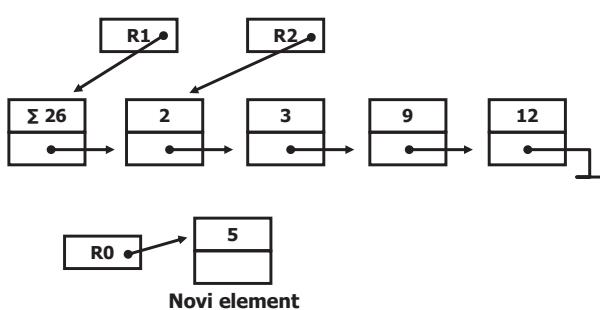
© Kovač, Basch, FER, Zagreb

327

## Ostali primjeri

### Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)



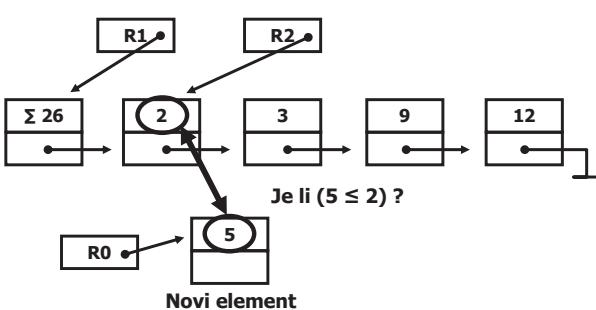
© Kovač, Basch, FER, Zagreb

Pokazuje čvor čiju vrijednost uspoređujemo s novim čvorom (jedan čvor dalje od R1)

## Ostali primjeri

### Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)

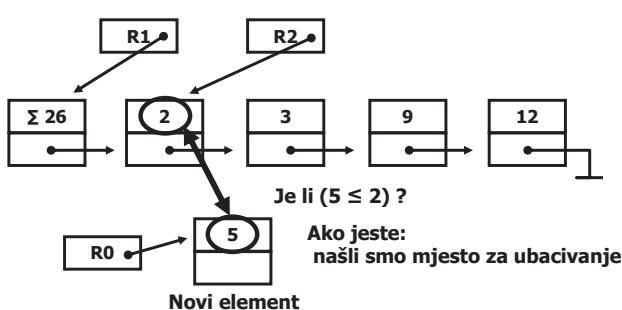


© Kovač, Basch, FER, Zagreb

## Ostali primjeri

### Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)



© Kovač, Basch, FER, Zagreb

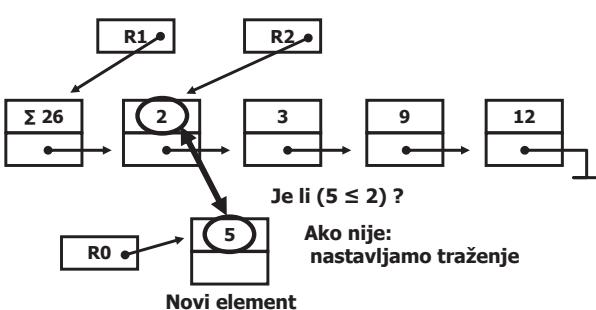
328

Pokazuje čvor čiju vrijednost uspoređujemo s novim čvorom (jedan čvor dalje od R1)

## Ostali primjeri

### Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)

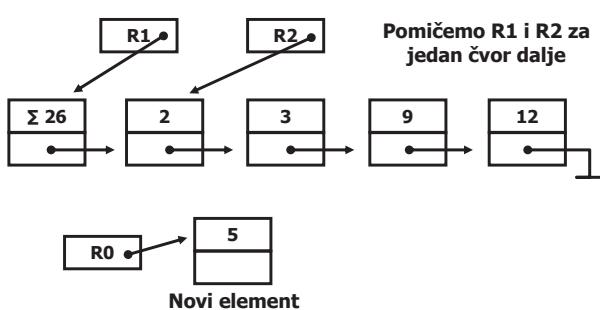


© Kovač, Basch, FER, Zagreb

## Ostali primjeri

### Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)



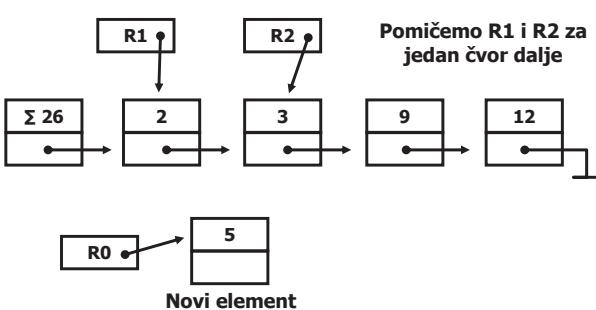
© Kovač, Basch, FER, Zagreb

330

Pokazuje čvor čiju vrijednost uspoređujemo s novim čvorom (jedan čvor dalje od R1)

## Ostali primjeri

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)

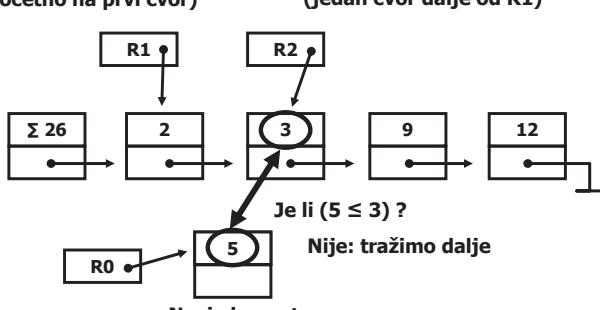


© Kovač, Basch, FER, Zagreb

## Ostali primjeri

### Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)



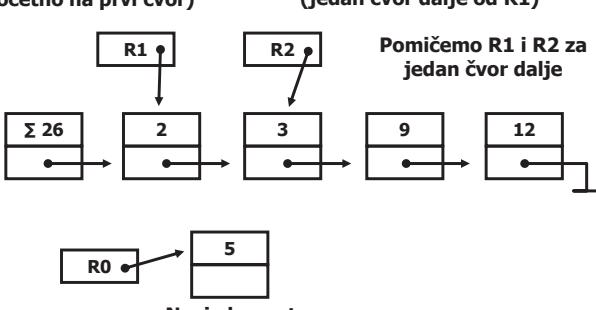
© Kovač, Basch, FER, Zagreb

332

Pokazuje čvor čiju vrijednost uspoređujemo s novim čvorom (jedan čvor dalje od R1)

## Ostali primjeri

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)



© Kovač, Basch, FER, Zagreb

334

© Kovač, Basch, FER, Zagreb

329

331

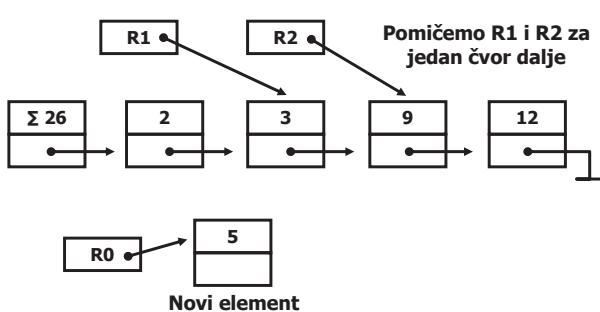
333

335

## Ostali primjeri

### Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)



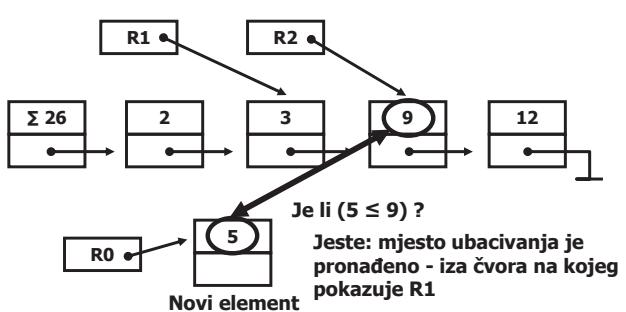
© Kovač, Basch, FER, Zagreb

Pokazuje čvor čiju vrijednost uspoređujemo s novim čvorom (jedan čvor dalje od R1)

## Ostali primjeri

### Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)



© Kovač, Basch, FER, Zagreb

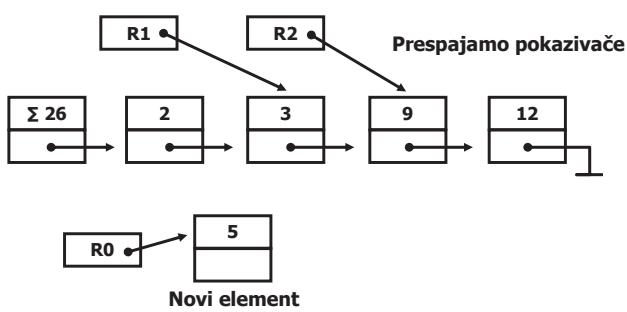
337

## Ostali primjeri

### Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)

Pokazuje čvor čiju vrijednost uspoređujemo s novim čvorom (jedan čvor dalje od R1)



© Kovač, Basch, FER, Zagreb

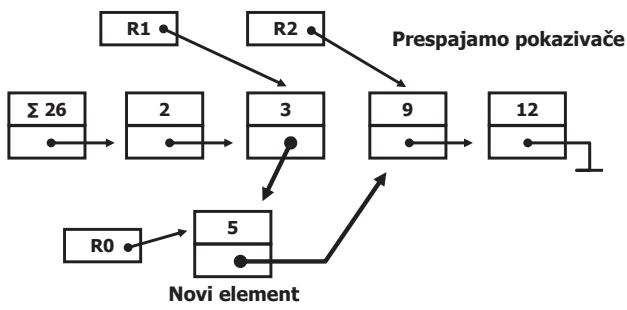
338

## Ostali primjeri

### Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)

Pokazuje čvor čiju vrijednost uspoređujemo s novim čvorom (jedan čvor dalje od R1)



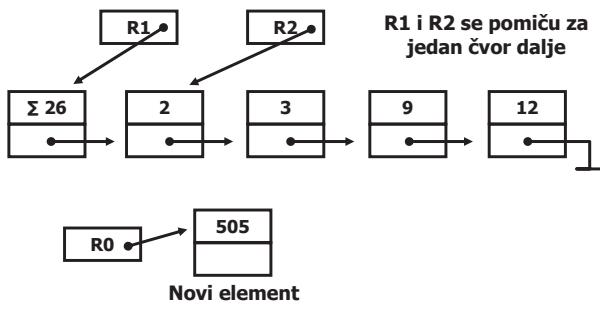
© Kovač, Basch, FER, Zagreb

340

## Ostali primjeri

### Način rada potprograma:

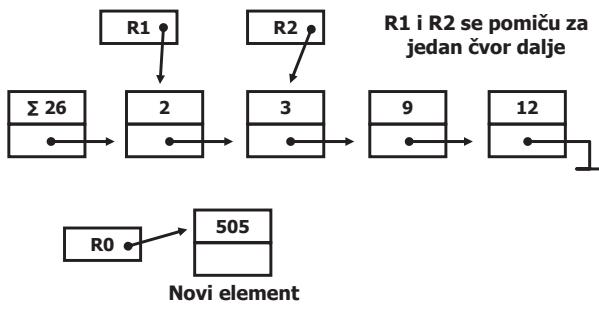
Pogledajmo još slučaj kad su u listi svi elementi manji od novog člana (ili je lista prazna)



© Kovač, Basch, FER, Zagreb

342

Pogledajmo još slučaj kad su u listi svi elementi manji od novog člana (ili je lista prazna)



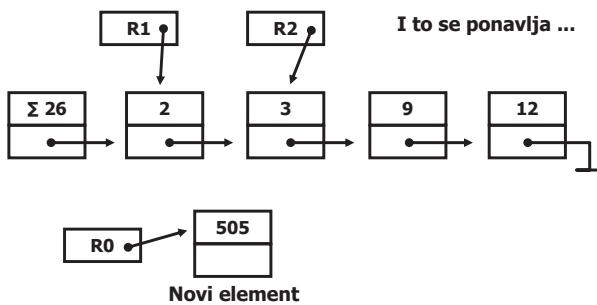
© Kovač, Basch, FER, Zagreb

343

## Ostali primjeri

### Način rada potprograma:

Pogledajmo još slučaj kad su u listi svi elementi manji od novog člana (ili je lista prazna)



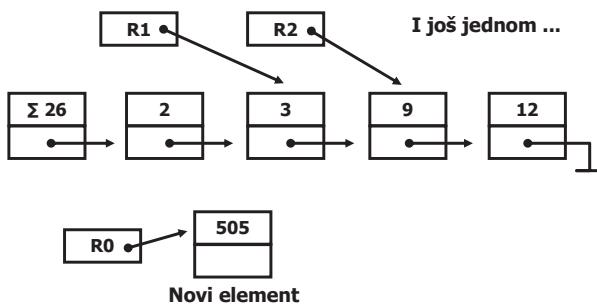
© Kovač, Basch, FER, Zagreb

344

## Ostali primjeri

### Način rada potprograma:

Pogledajmo još slučaj kad su u listi svi elementi manji od novog člana (ili je lista prazna)



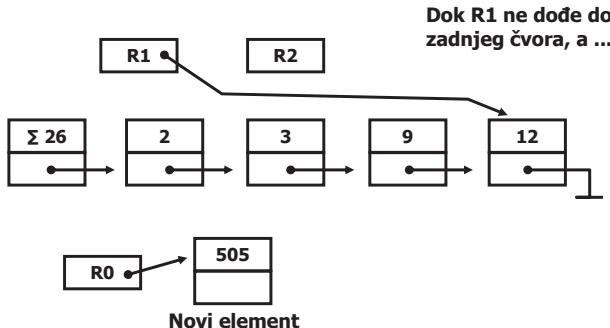
© Kovač, Basch, FER, Zagreb

346

## Ostali primjeri

### Način rada potprograma:

Pogledajmo još slučaj kad su u listi svi elementi manji od novog člana (ili je lista prazna)



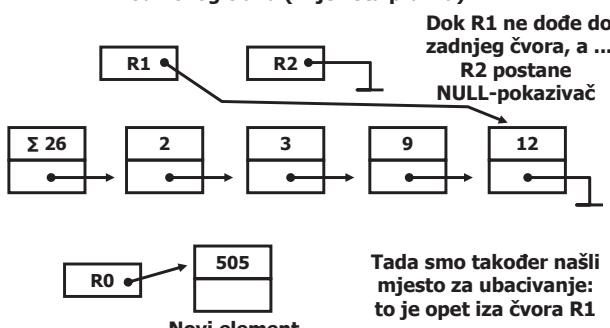
© Kovač, Basch, FER, Zagreb

348

## Ostali primjeri

### Način rada potprograma:

Pogledajmo još slučaj kad su u listi svi elementi manji od novog člana (ili je lista prazna)



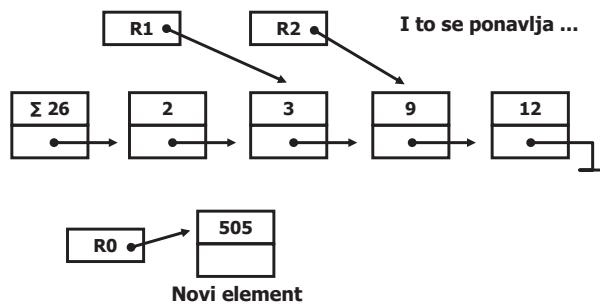
© Kovač, Basch, FER, Zagreb

350

## Ostali primjeri

### Način rada potprograma:

Pogledajmo još slučaj kad su u listi svi elementi manji od novog člana (ili je lista prazna)



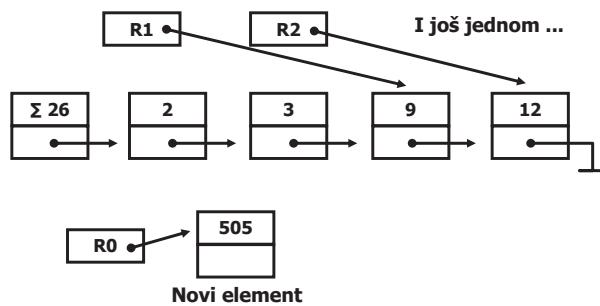
© Kovač, Basch, FER, Zagreb

345

## Ostali primjeri

### Način rada potprograma:

Pogledajmo još slučaj kad su u listi svi elementi manji od novog člana (ili je lista prazna)



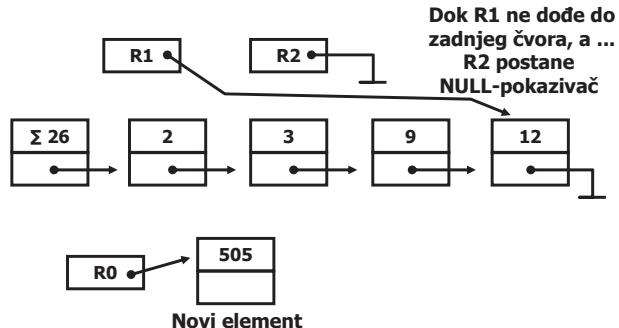
© Kovač, Basch, FER, Zagreb

347

## Ostali primjeri

### Način rada potprograma:

Pogledajmo još slučaj kad su u listi svi elementi manji od novog člana (ili je lista prazna)

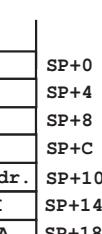


© Kovač, Basch, FER, Zagreb

349

## Ostali primjeri

UBACI ;;;; Potprogram UBACI  
; Parametri na stogu:  
; 1. parametar:  
; adresa prvog čvora (glava)  
; 2. parametar:  
; adresa novog čvora  
okvir  
PUSH R1 ; Spremi sve  
PUSH R2 ; registre koje  
PUSH R5 ; potprogram  
PUSH R6 ; mijenja (osim R0).  
  
; dohvati vrijednosti novog čvora za pretraživanje  
LOAD R0, (SP+14) ; adresa novog čvora u R0  
LOAD R6, (R0) ; vrijednost novog čvora u R6  
  
; priprema pokazivača R1 i R2 za pretraživanje  
LOAD R1, (SP+18) ; adresa prvog čvora u R1  
LOAD R2, (R1+4) ; adresa drugog čvora u R2



>>>

351

© Kovač, Basch, FER, Zagreb

© Kovač, Basch, FER, Zagreb

&lt;&lt;&lt;

;; PETLJA ZA TRAŽENJE MJESTA ZA UBACIVANJE

```

TRAZI CMP R2, 0 ; Ako je kraj liste:
JR_Z NASAO ; => onda ubacujemo na kraj

; dohvati u R5 vrijednost trenutačnog čvora i
; usporedi je s vrijednošću R6 novog čvora
LOAD R5, (R2)
CMP R6, R5

JR_ULE NASAO ; ako je novi ≤ trenutačni =>
 ; pronadeno je mjesto za ubacivanje

; inače treba nastaviti s petljom za traženje

MOVE R2, R1 ; pomakni se na sljedeći čvor
LOAD R2, (R1+4)

JR TRAZI ; nastavi s traženjem

```

© Kovač, Basch, FER, Zagreb

&gt;&gt;&gt;

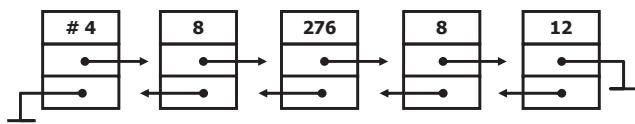
352

## Ostali primjeri



### Primjer:

U memoriji se nalazi dvostruko povezana nesortirana lista. Svaki čvor zauzima tri memorijske lokacije: na prvoj je NBC broj, na drugoj je pokazivač na sljedeći čvor liste, a na trećoj je pokazivač na prethodni čvor liste. Prvi čvor liste ima specijalno značenje i uvijek je prisutan u listi. U njemu se pamti ukupan broj preostalih čvorova u listi.



&gt;&gt;&gt;

© Kovač, Basch, FER, Zagreb

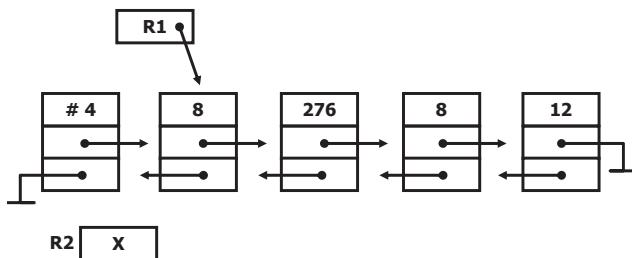
354

## Ostali primjeri



&lt;&lt;&lt; Način rada potprograma:

Pokazivač na čvor koji se uspoređuje s X (početno se pokazuje na drugi čvor)



Broj X bit će stalno spremljen u R2

© Kovač, Basch, FER, Zagreb

356

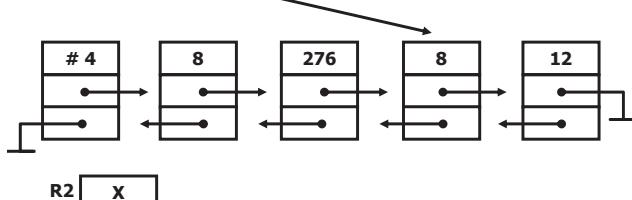
## Ostali primjeri



&lt;&lt;&lt; Način rada potprograma:

Pomičemo R1 za jedan čvor dalje ...

I to se ponavlja sve dok...



R2 X

© Kovač, Basch, FER, Zagreb

358

&lt;&lt;&lt;

;; DIO ZA UMETANJE NOVOG ČVORA U LISTU

```

NASAO STORE R0, (R1+4) ; stavi novi čvor iza prethodnog
STORE R2, (R0+4) ; stavi trenutačni čvor iza novog

; izračunavanje novog zbroja u prvom čvoru:

LOAD R1, (SP+18) ; dohvati adresu prvog
LOAD R0, (R1) ; dohvati dosadašnji zbroj

ADD R0, R6, R0 ; novi zbroj stavi u R0 (rezultat)
STORE R0, (R1) ; spremi ga u prvi čvor

POP R6 ; obnovi
POP R5 ; vrijednosti
POP R2 ; spremljenih
POP R1 ; registara

```

RET

© Kovač, Basch, FER, Zagreb

353

## Ostali primjeri



&lt;&lt;&lt;

Treba napisati potprogram IZBACI koji prima preko stoga dva parametra: pokazivač na prvi čvor liste i NBC broj (X). Potprogram traži prvo pojavljivanje čvora u kojemu je upisan broj X. Ako ga nađe, izbacuje taj čvor iz liste. Povratna vrijednost je adresa izbačenog čvora ili ništica ako čvor nije pronađen. Vrijednost se vraća pomoću R1. Potprogram ne smije mijenjati sadržaje registara u glavnom programu.

### Rješenje:

Parametre će sa stoga uklanjati glavni program.

&gt;&gt;&gt;

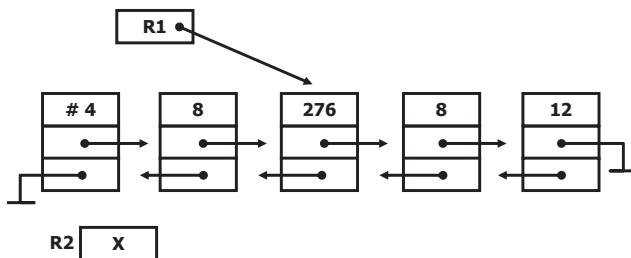
© Kovač, Basch, FER, Zagreb



## Ostali primjeri

&lt;&lt;&lt; Način rada potprograma:

Pomičemo R1 za jedan čvor dalje ...



R2 X

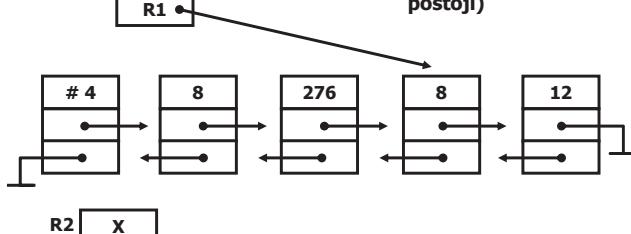
© Kovač, Basch, FER, Zagreb



## Ostali primjeri

&lt;&lt;&lt; Način rada potprograma:

Ne pronađemo čvor ili  
dođemo do kraja liste  
(tada traženi čvor ne  
postoji)



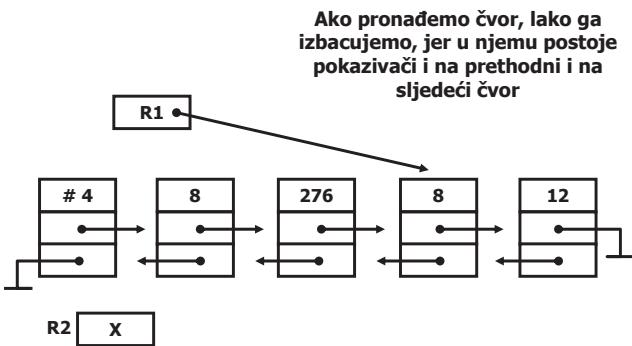
R2 X

© Kovač, Basch, FER, Zagreb

359



<<< Način rada potprograma:

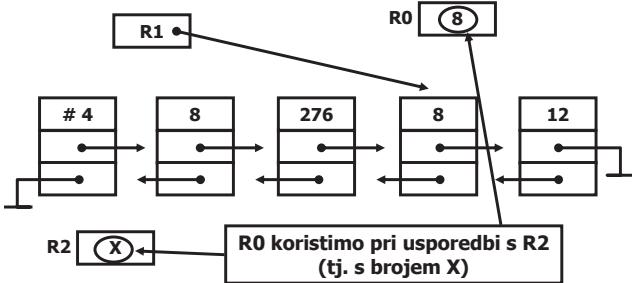


© Kovač, Basch, FER, Zagreb

360



<<< Način rada potprograma:



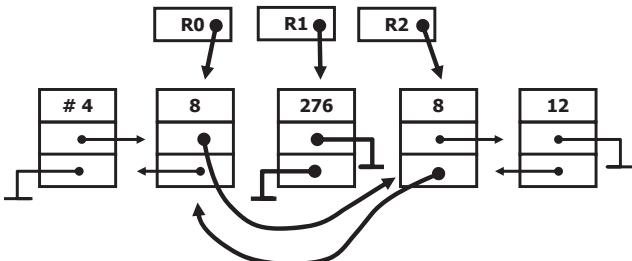
© Kovač, Basch, FER, Zagreb

362



<<< Način rada potprograma:

Pokazivači u čvorovima se prespajaju ovako:



© Kovač, Basch, FER, Zagreb

364

<<<  
NASA0 ;;; odspoji nadeni čvor iz liste

```

LOAD R0, (R1+8) ; stavi adresu prethodnog u R0
LOAD R2, (R1+4) ; stavi adresu sljedećeg u R2
STORE R2, (R0+4) ; stavi sljedeći iza prethodnog
OR R2, R2, R2 ; provjeri izbacuje li se zadnji
JR_Z DALJE ; čvor iz liste (tj. R2 je NULL)
STORE R0, (R2+8) ; stavi prethodni ispred sljedeć.
DALJE MOVE 0, R0 ; odspoji pokazivače
STORE R0, (R1+4) ; u čvoru kojeg
STORE R0, (R1+8) ; izbacuješ iz liste
;; smanji brojač čvorova u 1. čvoru liste
LOAD R0, (SP+10) ; dohvati adresu 1. čvora
LOAD R2, (R0+0) ; dohvati broj čvorova liste
SUB R2, 1, R2 ; smanji broj čvorova
STORE R2, (R0+0) ; upiši ga natrag u 1. čvor
IZLAZ POP R2 ; obnovi registre
 ; iz glavnog programa
POP R0
RET

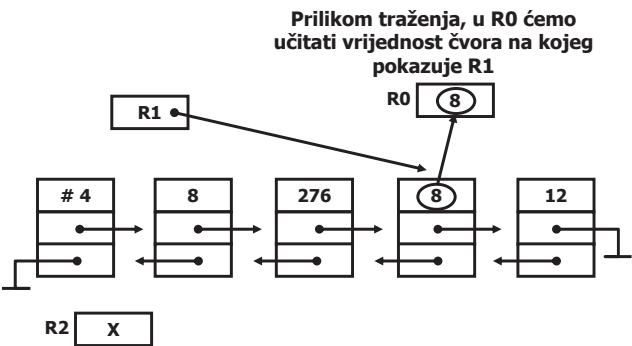
```

© Kovač, Basch, FER, Zagreb

366



<<< Način rada potprograma:



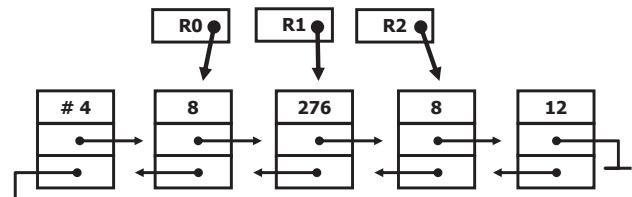
© Kovač, Basch, FER, Zagreb

361



<<< Način rada potprograma:

Kod izbacivanja čvora, registri se postavljaju ovako (na ovoj slici izbacuje se čvor 276):



© Kovač, Basch, FER, Zagreb

363

IZBACI ;;; Potprogram IZBACI

```

; Parametri na stogu:
; 1. parametar:
; adresa prvog čvora
; 2. parametar:
; broj X koji se traži
okvir { R2 SP+0
 R0 SP+4
 pov.adr. SP+8
 BROJ X SP+C
 GLAVA SP+10
}
PUSH R0 ; spremi registre
PUSH R2 ; iz glavnog programa
LOAD R2, (SP+0C) ; dohvati broj X
LOAD R1, (SP+10) ; dohvati adresu prvog čvora
LOAD R1, (R1+4) ; pripremi pokazivač za traženje
TRAZI CMP R1, 0 ; ako je kraj => čvor nije naden
 JR_Z IZLAZ
 LOAD R0, (R1+0) ; dohvati vrijednost čvora
 CMP R0, R2 ; usporedi je sa X
 JR_EQ NASAO ; ako su isti => čvor je naden
 LOAD R1, (R1+4) ; pomakni se na sljedeći čvor
 JR TRAZI ; nastavi s traženjem
 >>>

```

© Kovač, Basch, FER, Zagreb

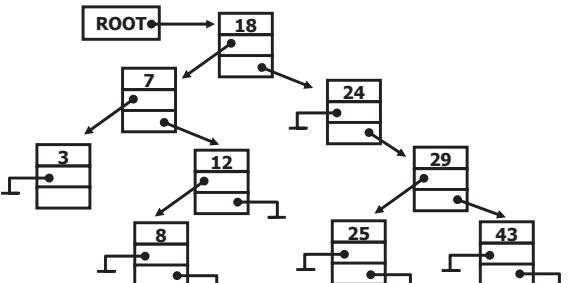
365



## Primjer:

U memoriji se nalazi sortirano binarno stablo. Svaki čvor zauzima tri memoriske lokacije: na prvoj je NBC broj, na drugoj je pokazivač na lijevo podstablo, a na trećoj je pokazivač na desno podstablo.

Poseban pokazivač (ROOT) u glavnom programu pokazuje na korijen stabla.



© Kovač, Basch, FER, Zagreb

367



<<<

Treba napisati potprogram PISI koji će ispisati sve brojeve iz čvorova stabla, ali u rastućem redoslijedu. Parametar potprograma PISI je adresa ishodišnog čvora, a prenosi se stogom. Povratna vrijednost potprograma je broj ispisanih čvorova, a vraća se preko R0.

Ispisivanje se obavlja tako da se pozove potprogram PRINT i kao parametar mu se pošalje broj koji se želi ispisati. Parametar za PRINT šalje se preko R0, a PRINT nema povratne vrijednosti. Pretpostavite da potprogram PRINT već postoji negdje u memoriji.

Treba napisati i glavni program, koji će ispisati stablo čija adresa je u pokazivaču ROOT, a broj ispisanih čvorova treba spremiti U R5.

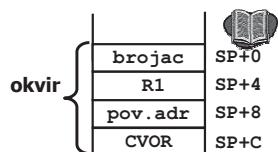
>>>

© Kovač, Basch, FER, Zagreb

```
GLAVNI LOAD R0, (ROOT)
PUSH R0
CALL PISI
ADD SP, 4, SP
MOVE R0, R5
HALT
```

ROOT DW ... ; adresa stabla

368



```
PISI ;;;; Potprogram PISI
; Parametar na stogu:
; 1. parametar:
; adresa čvora

PUSH R1 ; spremi registre

SUB SP, 4, SP ; lokalna varijabla brojac
LOAD R1, (SP+C) ; dohvati adresu čvora
MOVE 0, R0
CMP R1, 0 ; if(cvor == NULL)
JR_Z VAN ; idi na return
```

© Kovač, Basch, FER, Zagreb

>>>

370

© Kovač, Basch, FER, Zagreb

```
LOAD R0, (R1+4) ; cvor->ljevi
PUSH R0
CALL PISI ; PISI()
ADD SP, 4, SP
STORE R0, (SP+0) ; brojac =
; povratna vrijednost

LOAD R0, (R1) ; cvor->broj
CALL PRINT ; PRINT()

LOAD R0, (R1+8) ; cvor->desni
PUSH R0
CALL PISI ; PISI()
ADD SP, 4, SP

LOAD R1, (SP+0) ; stavi brojac u R1
ADD R1, R0, R0 ; brojac += povratna vrijednost
ADD R0, 1, R0 ; stavi (brojac+1) u R0 za return
; return
VAN ADD SP, 4, SP ; ukloni lokalnu var.
POP R1 ; obnovi registre
RET
```

© Kovač, Basch, FER, Zagreb

369



371

## Priklučci

- Svaki procesor, memorija ili bilo koja druga komponenta ostvarena kao čip ima određen broj priključaka ili izvoda (nazivaju se još i pinovi prema engleskom izvorniku)



DIP

Dual in line package



SMD

Surface-mount device



PGA

Pin grid array

- Pomoću priključaka se komponenta spaja na sabirnicu i putem sabirnica na ostale dijelove računalnog sustava

© Kovač, Basch, FER, Zagreb

2

## Priklučci - podjela po namjeni



### Priklučci - podjela po namjeni



1

### Priklučci - podjela po namjeni





- Priklučci koji imaju upravljačku namjenu (možemo promatrati kao da prenose logičko stanje true ili false):

• mogu biti aktivni u niskoj razini i tada se označavaju imenom s potezom. Na primjer IREQ može označavati zahtjev za prekid. Ako je priključak nisko, onda znači da postoji zahtjev za prekid, a u suprotnom ne postoji.

• mogu biti aktivni u visokoj razini i tada se nazivaju imenom bez dodatnih oznaka. Na primjer READ može označavati ciklus čitanja. Ako je priključak visoko, onda se trenutačno obavlja ciklus čitanja, a u suprotnom se ne obavlja ciklus čitanja

>>>

## Priklučci - "širina" priključaka



- Po širini priključci mogu biti:

- Jednostruki priključci su oni koji imaju svoju samostalnu namjenu. To su obično upravljački priključci
- Priklučci grupirani u skupinu po nekom zajedničkoj funkciji. Na primjer, mogu biti 32 adresna priključka, a pojedinačni se priključci označavaju nazivima ADR0, ADR1, ... ADR31

## Priklučci - smjer priključaka

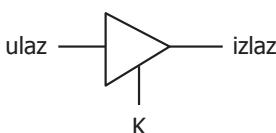


- Komponenta upravlja svojim izlaznim priključcima, a druge komponente (spojene preko sabirnice na njih) "osluškuju" njihovo stanje
- Za ulazne priključke komponenta samo "osluškuje" stanje na njima. Sabirnicama koje su povezane na ove priključke upravljaju druge komponente
- Dvosmjerni priključci spojeni su na sabirnicu kojom u različitim trenutcima upravljaju različite komponente. Pri tome **uvijek samo jedna komponenta upravlja sabirnicom u nekom trenutku**, a priključci svih ostalih komponenata su ili **ulazni** ili u **stanju visoke impedancije**

## Priklučci - smjer priključaka



- Ako je upravljački ulaz K neaktivan (0), onda je izlaz u stanju visoke impedancije (high Z)
- Ako je upravljački ulaz K aktivan (1), onda je stanje ulaza prenosi na izlaz



| K | ulaz | izlaz |
|---|------|-------|
| 0 | 0    | X Z   |
| 0 | 1    | X Z   |
| 1 | 0    | 0 → 0 |
| 1 | 1    | 1 → 1 |

- Efektivno, sklop sa tri stanja ponaša se kao ventil koji propušta ili ne propušta vodu



<<<

• mogu označavati jedno od dva moguća stanja i tada u imenu sadrže nazine oba stanja. Na primjer READ/WRITE može označavati da li se trenutačno izvodi ciklus čitanja ili pisanja. Ako je priključak nisko, onda se izvodi čitanje (READ ima potez). Ako je priključak visoko, onda se izvodi ciklus pisanja (WRITE nema potez)

- U pravilu, komponente koje osluškuju upravljačke priključke aktiviraju se na brid signala (tipično u trenutku kad signal prelazi iz neaktivnog u aktivno stanje)
- Kad se ne promatra brid, nego stanje signala, onda se stanje "očitava" u točno definiranim trenutcima

## Priklučci - smjer priključaka

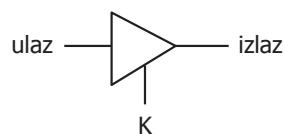


- Priklučci se mogu podijeliti po smjeru signala (podataka) koji njima putuje na:
  - ulazne
  - izlazne
  - dvosmjerne
- Smjer priključka uvijek se definira u odnosu na komponentu koju promatramo

## Priklučci - smjer priključaka



- Stanje visoke impedancije omogućuje jednostavno spajanje više komponenata na istu sabirnicu (**ponoviti iz "Digitalne"**)
- Sklopove s tri stanja simbolički prikazujemo ovako:



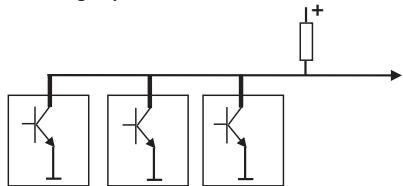
## Priklučci - smjer priključaka



- Sklop s tri stanja omogućuje spajanje više priključaka na istu sabirnicu
- Sabirnicom smije upravljati najviše jedan priključak i on određuje stanje sabirnice (0 ili 1)
- Svi ostali priključci moraju biti neaktivni i oni ne utječu na stanje na sabirnici



- Postoje i posebne vrste priključaka. To su tzv. *open collector* priključci (**ponoviti iz "Digitalne"**)
  - oni omogućuju da se **više izlaznih priključaka** spoji zajedno na jednu sabirnicu i da **svi zajedno njome upravljaju** u istom trenutku
  - stanje sabirnice određeno je logičkom funkcijom spojeni-I (wired-AND) između svih priključaka (naziva se još i wired-OR - za negativnu logiku)

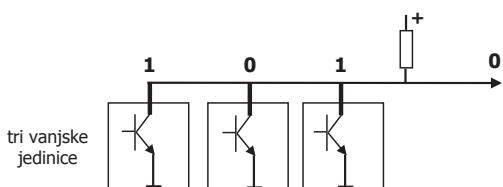


© Kovač, Basch, FER, Zagreb

13



- Ako se bilo koji prekidni priključak **aktivira** (aktivna razina je u niskom), onda se aktivira cijela sabirnica, tj. prelazi u nisku razinu



© Kovač, Basch, FER, Zagreb

15

## Priklučci procesora FRISC

© Kovač, Basch, FER, Zagreb

17

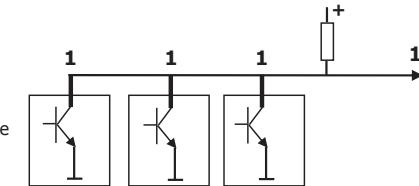


### FRISC ima 80 priključaka čije namjene su:

- Vcc i GND (voltage i ground) su priključci za napajanje
- CLOCK ili takt služi za dovođenje pravokutnog signala stabilne frekvencije koji daje takt rada procesoru. Izvor signala je obično kristalni oscilator, a signal se vodi i do drugih komponenata u računalnom sustavu pa se naziva i takt sustava



- Ako su svi prekidni priključki neaktivni (u visokoj su razini), onda je cijela sabirnica neaktivna (u visokoj je razini)

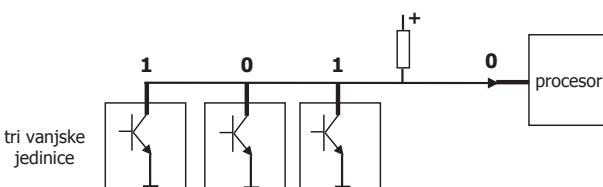


© Kovač, Basch, FER, Zagreb

14



- Open collector* priključci se koriste, npr. za spajanje prekidnih priključaka vanjskih jedinica na jednu sabirnicu
- Na ovaj način procesor koji osluškuje prekidnu liniju može detektirati da je netko postavio zahtjev za prekid

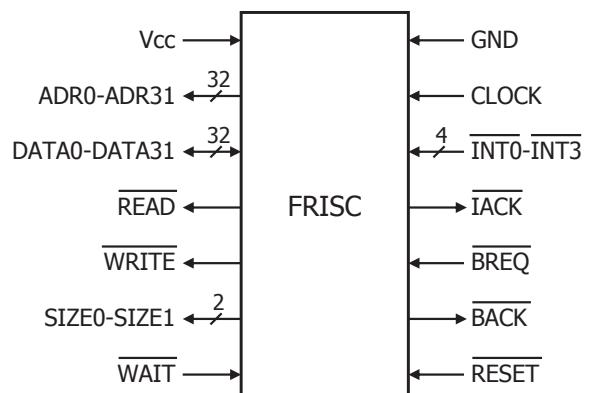


© Kovač, Basch, FER, Zagreb

16



- Priklučci našeg procesora prikazani su sljedećom slikom:



© Kovač, Basch, FER, Zagreb

>>>

18



- ADR (address) (izlazni) je 32-bitni skup adresnih priključaka. Pomoću njih FRISC adresira memoriju i vanjske jedinice
- DATA (dvosmjerni) je 32-bitni skup podatkovnih priključaka. Pomoću njih se podatak prenosi u procesor ili iz procesora - ovisno čita li se ili piše podatak iz memorije odnosno vanjske jedinice
- READ i WRITE (izlazni, aktivni nisko) su priključci kojima procesor označuje memoriji ili vanjskoj jedinici želi li obaviti operaciju čitanja ili pisanja:
  - operacija čitanja: READ aktivan, a WRITE neaktivan
  - operacija pisanja: tada je READ neaktivan, a WRITE aktivan
  - brid jednog ili drugog signala može upotrijebiti memorija ili vanjska jedinica da započne traženu operaciju



- SIZE (izlazni) je 2-bitni skup priključaka kojima procesor zadaje širinu podatka kojem želi pristupiti. Mogu se zadati tri veličine: 8, 16 i 32 bita:
  - SIZE=00 znači da nema pristupa memoriji
  - SIZE=01 znači da se pristupa bajtu (8 bita)
  - SIZE=10 znači da se pristupa poluriječi (16 bita)
  - SIZE=11 znači da se pristupa riječi (32 bita)
- Brid na SIZE0 ili SIZE1 može upotrijebiti memorija da započne traženu operaciju
- WAIT (ulazni, aktivan nisko) je priključak pomoću kojeg spore memorije ili UI jedinice mogu zatražiti od procesora da produlji normalni ciklus čitanja ili pisanja podatka



- INT i IACK ćemo detaljnije učiti kasnije

- INT (interrupt) (ulazni, aktivni nisko) su četiri priključka INT0 do INT3 preko kojih prekidne vanjske jedinice mogu procesoru postaviti zahtjev za prekid. Prekidni priključci vanjskih jedinica povezani su na spojeni-I sabirnicu.
- IACK (interrupt acknowledge) (izlazni, aktivan nisko) je priključak kojim procesor potvrđuje da je prihvatio zahtjev za prekid najviše razine.

## Sabirnice - uvod



- Prednosti sabirnice:
  - mala cijena (isti spojni put dijeli više uređaja)
  - prilagodljivost prilikom projektiranja i nadogradnje računala (jednostavno dodavanje uređaja)
  - standardiziranost
- Nedostatci sabirnice:
  - mala propusnost
  - ograničena duljina sabirnice
  - ograničen broj uređaja koji se mogu spojiti na jednu sabirnicu
  - problemi zbog uređaja različite brzine



- RESET (ulazni, aktivan nisko) je priključak pomoću kojeg se procesor može dovesti u početno stanje (tj. resetirati) koje je jednako stanju nakon priključenja na napajanje:
  - u R0 do R7, SR i PC je ništica
  - izlazni priključci su u neaktivnom stanju
  - dvosmjerni priključci su u visokoj impedanciji
  - nakon deaktiviranja priključka RESET, procesor započinje s dohvatom i izvođenjem naredaba (s adrese 0, jer je inicijalno u PC-u 0)

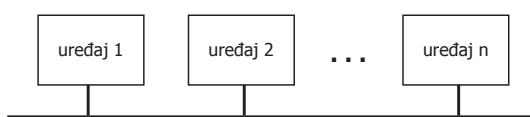


- BREQ i BACK ćemo detaljnije učiti kasnije

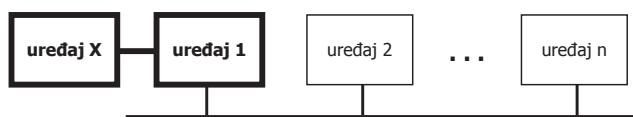
- BREQ (bus request) (ulazni, aktivan nisko) je priključak pomoću kojeg DMA-jedinica može zatražiti od procesora da joj prepusti upravljanje nad sabirnicom
- BACK (bus acknowledge) (izlazni, aktivan nisko) je priključak kojim procesor potvrđuje DMA-jedinici da je prihvatio njen zahtjev za upravljanje sabirnicom.



- Sabirnica (engl. bus) je spojni put koji povezuje više uređaja (tj. dijelova računalnog sustava), a sastoji se od skupa vodiča



- Alternativa sabirnici je **spajanje dva uređaja** (point-to-point) pomoću vlastitog spojnjog puta prilagođenog upravo tim uređajima
  - Prednost je veća brzina komunikacije
  - Nedostatak je veća cijena (više spojnih putova, više priključaka na čipu)
- Ovakvo spajanje koristi se u specijalnim slučajevima kad je potrebna visoka propusnost





- Sabirnice se mogu dijeliti i na:

- memorijsku sabirnicu
- ulazno-izlaznu (U/I) sabirnicu
- sabirnice specijalne namjene (npr. grafička)

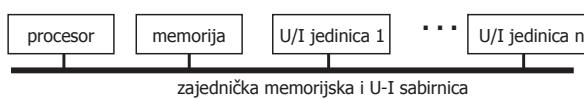
#### • Memorijska sabirnica:

- povezuje procesor i memoriju
- male duljine
- velike brzine rada
- prilagođena brzini memorije

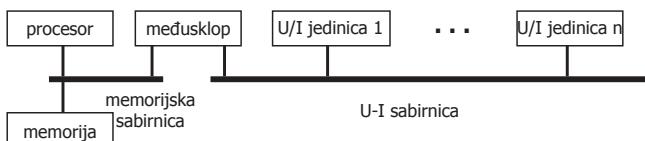
## Sabirnice - memorijske i UI - način spajanja



- Zajednička memorijska i U/I sabirnica (backplane bus)



- Spajanje memorijske i UI sabirnice pomoću posebnog međusklopa (tj. neizravno)



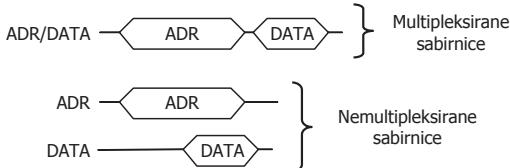
## Multipleksirane sabirnice



- Na primjer, adresna i podatkovna sabirnica mogu dijeliti iste spojne putove i to tako da:

- se u jednom trenutku prenosi adresa
- a u drugom trenutku se prenosi podatak

- Vremenski** ove funkcije moraju biti jasno **odijeljene** jer se ne mogu događati istodobno



## Multipleksirane sabirnice



- U prijašnje vrijeme (prije ~20-30 godina) je zbog tehnoloških razloga broj izvoda na čipu bio ograničen na svega nekoliko desetaka izvoda
- Da bi mogli imati sabirnice većih širina (npr. podatkovna i adresna), koristile su se tzv. **multipleksirane sabirnice**
- Sabirnice mogu biti "razdvojene" ili nemultiplexirane, u smislu da **postoje zasebni spojni putovi i priključci za svaku namjenu**
- Sabirnice mogu biti multipleksirane \*, što znači da **isti spojni putovi imaju više namjena, ali ne u isto vrijeme**

\* priključci na čipovima tada su također multipleksirani

## Multipleksirane sabirnice



- Prednosti multipleksiranih sabirnica su:
  - manji broj priključaka na čipu
  - manji broj spojnih vodova
  - jeftinija izvedba skloplavlja
- Nedostatci multipleksiranih sabirnica su:
  - komplikiranija izvedba prijenosa podataka
  - nemogućnost vremenskog preklapanja pojedinih prijenosa, tj. njihova slijednost

## Sabirnice - sabirnički protokoli

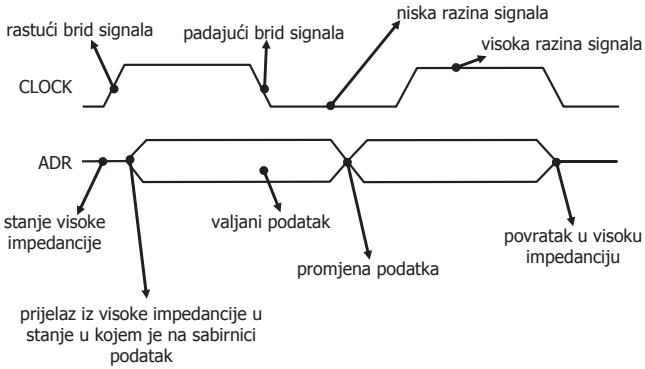
### Sabirnice - sabirnički protokoli



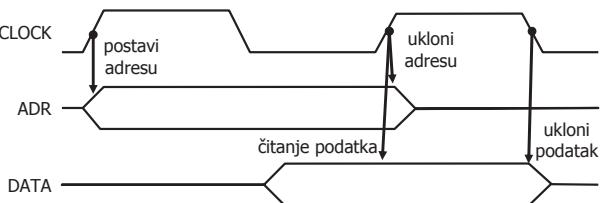
- Točan redoslijed svih koraka u komunikaciji naziva se **sabirnički protokol** (bus protocol)

- Sabirnička transakcija (bus transaction) je slijed koraka potrebnih da bi se na sabirnici izvela određena operacija, kao npr. operacija čitanja ili pisanja
- Sabirnička transakcija obično sadrži:
  - zahtjev (request)
  - odgovor (response)
- Unutar zahtjeva i/ili odgovora može se nalaziti podatak, adresa, naredba i sl.

- Tumačenje oznaka na vremenskom dijagramu:

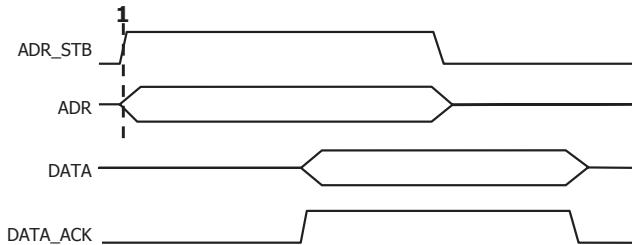


- Pojednostavljeni prikaz sinkrone komunikacije:



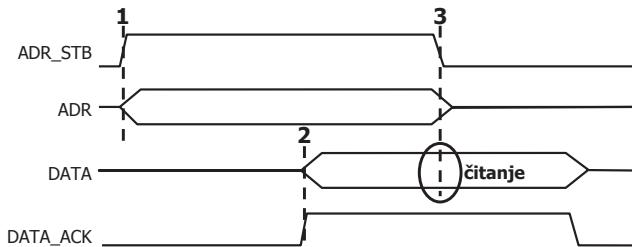
Sve operacije postavljanja adresa, čitanja podataka i uklanjanja adresa i podataka odvijaju se u vremenskim trenutcima **točno definiranim u odnosu na CLOCK**

- Pojednostavljeni prikaz asinkrone komunikacije (npr. čitanje):



- Procesor postavlja adresu na ADR i dojavljuje to memoriji aktiviranjem signala ADR\_STB (address strobe)

- Pojedostavljeni prikaz asinkrone komunikacije (npr. čitanje):



- Procesor je detektirao DATA\_ACK i zna da se na DATA nalazi podatak; procesor čita podatak sa DATA te uklanja adresu sa ADR i dojavljuje to memoriji deaktiviranjem ADR\_STB



- Sabirnice se prema načinu komunikacije dijele na\*:

- sinkrone
- asinkrone

### Sinkrone sabirnice:

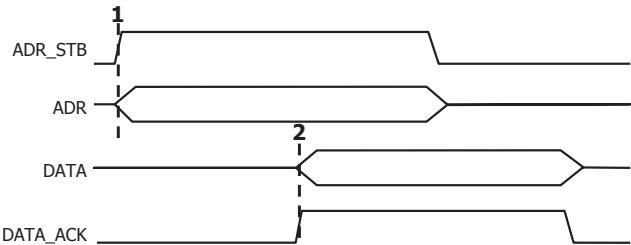
- sve operacije su sinkronizirane s taktom sustava (tj. clockom)**
- jednostavne su za implementaciju
- imaju veliku brzinu rada pa zato i malu duljinu
- bolje su prilagođene za slučaj kad svi uređaji imaju jednaku brzinu
- imaju mogućnost prilagodbe brzine rada, ali se komunikacija većinom odvija predviđenom brzinom
- češće se koriste za memoriske sabirnice

\* tako se dijele i sabirnički protokoli

### Asinkrone sabirnice:

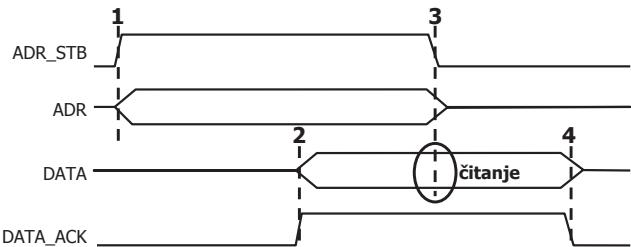
- ne koriste CLOCK za sinkronizaciju
- uređaji se sinkroniziraju tzv. rukovanjem** (engl. handshaking protocol)
  - rukovanje je postupak u kojem strane koje komuniciraju prelaze na sljedeći korak komunikacije tek kad obje strane potvrde da je prethodni korak dovršen
- složenije su za implementaciju
- imaju manju brzinu rada
- mogu imati veliku duljinu
- bolje su prilagođene za slučaj kad uređaji imaju različite brzine
- češće se koriste za U-I sabirnice

- Pojedostavljeni prikaz asinkrone komunikacije (npr. čitanje):



- Memorija je detektirala ADR\_STB i započela operaciju čitanja; nakon nekog vremena postavlja podatak na DATA i dojavljuje to procesoru aktiviranjem DATA\_ACK (acknowledge)

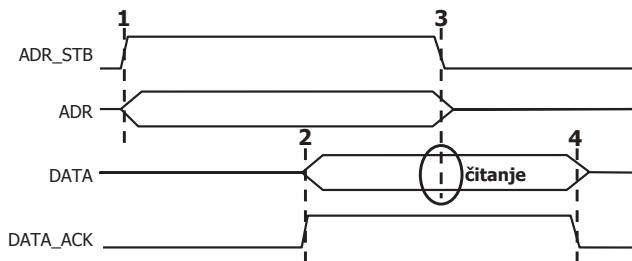
- Pojedostavljeni prikaz asinkrone komunikacije (npr. čitanje):



- Memorija je detektirala deaktiviranje ADR\_STB i zna da je procesor pročitao podatak; memorija uklanja podatak sa DATA i javlja to procesoru deaktiviranjem DATA\_ACK



- Pojednostavljeni prikaz asinkrone komunikacije (npr. čitanje):



Nakon koraka 4) procesor detektira deaktiviranje DATA\_ACK i može započeti novo čitanje ili pisanje podatka, tj. može ponovno započeti s korakom 1)



- Komentari:

- Neke memorijске sabirnice koriste asinkrone protokole
  - Međutim, to ne znači da te sabirnice nemaju CLOCK ili da ga uopće ne koriste
  - CLOCK se koristi u procesoru za njegov interni rad
  - Budući da interni rad procesora ovisi o njegovoj komunikaciji preko sabirnica, onda su i na asinkronoj sabirnici barem neki koraci (početni) sinkronizirani sa signalom CLOCK



- Karakteristike projektiranog sustava koje su važne za odabir sabirnice:
  - jednostavni procesor za ugradbeno računalo
  - želja je imati što jednostavniji i jeftiniji sustav
  - potreba za spajanjem U-I uređaja koji mogu imati različite brzine rada
  - želja za brzom komunikacijom s memorijom kako se ne bi ograničavale eventualne upotrebe procesora u drugim aplikacijama
- Iako su gornji zahtjevi djelomice međusobno suprotni, možemo napraviti kompromis...

>>>



<<<

- Ukupno gledano, za FRISC odabiremo:
  - neće se koristiti posebni spojni putovi između raznih dijelova sustava
  - zajednička memorijска и U-I sabirnica (tzv. backplane)
  - sinkrona sabirnica s mogućnošću prilagodbe brzine
    - brzina se prilagođava umetanjem tzv. ciklusa čekanja



<<<

- Ugradbeno računalo i jednostavni procesor
  - male dimenzije što znači da nije nužno imati asinkronu sabirnicu i jednostavnija implementacija sinkrone
- Jednostavnost i cijena sustava
  - pogodna je zajednička memorijска и U-I sabirnica bez potrebe za posebnim međusklopovima za povezivanje dviju sabirnica
- Različite brzine uređaja
  - moe se rješiti i sinkronom i asinkronom sabirnicom (nešto prirodne asinkronom)
- Brza komunikacijom s memorijom zbog šire primjene
  - bolji je odabir sinkrona sabirnica

>>>



- Definirajmo sabirničke protokole za FRISC

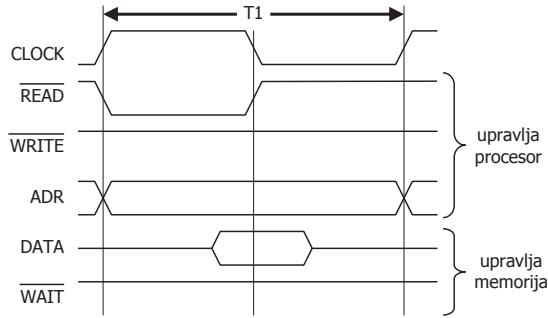
- Par napomena:

- Bit će jednaki za pristup memoriji ili U-I jedinicama (u objašnjenjima se spominje samo memorija, ali isto vrijedi i za U-I jedinice)
- Pokazat ćemo protokole za čitanje i pisanje podataka
- U slučaju normalne brzine, traju jedan takt CLOCK-a
- U slučaju sporih memorija ili U-I jedinica, umeću se dodatni ciklusi čekanja (svaki traje po jedan takt CLOCK-a)
- Promatramo samo one sabirničke vodove koji su relevantni za operacije čitanja i pisanja

## Protokol čitanja



- Sabirnički protokol čitanja bez stanja čekanja:



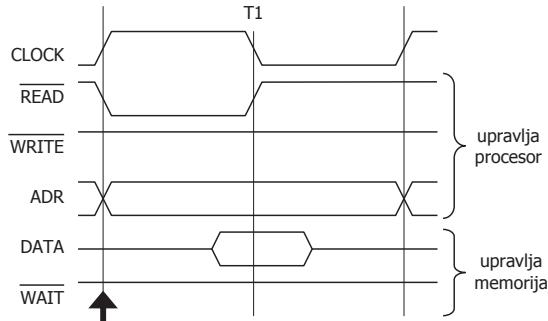
Sabirnička transakcija traje 1 takt signala CLOCK, označen sa T1

>>>

## Protokol čitanja



- Sabirnički protokol čitanja bez stanja čekanja:

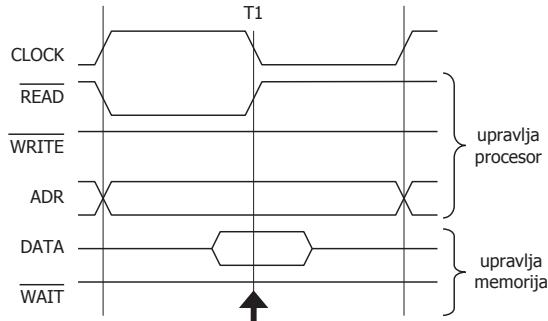


- Memorija na temelju brida na signalu READ zna da se na adresnoj sabirniči nalazi adresa; memorija započinje s dekodiranjem adrese i s dohvatom adresiranog podatka

## Protokol čitanja



- Sabirnički protokol čitanja bez stanja čekanja:

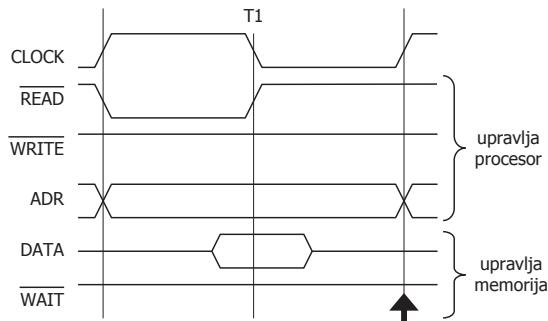


- Na padajući brid signala CLOCK, procesor ispituje WAIT koji je neaktivan pa procesor zna da je memorija stigla postaviti podatak; procesor preuzima podatak sa DATA i deaktivira READ

## Protokol čitanja



- Sabirnički protokol čitanja bez stanja čekanja:

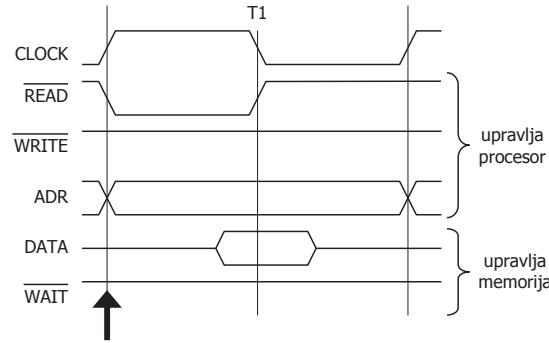


- Završetak trenutačne transakcije i početak nove

## Protokol čitanja



- Sabirnički protokol čitanja bez stanja čekanja:

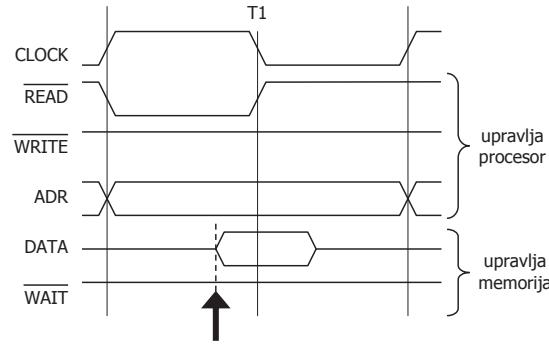


- Početak je na rastući brid CLOCK-a: procesor postavlja adresu na ADR i istovremeno aktivira READ čime naznačuje da želi čitati

## Protokol čitanja



- Sabirnički protokol čitanja bez stanja čekanja:

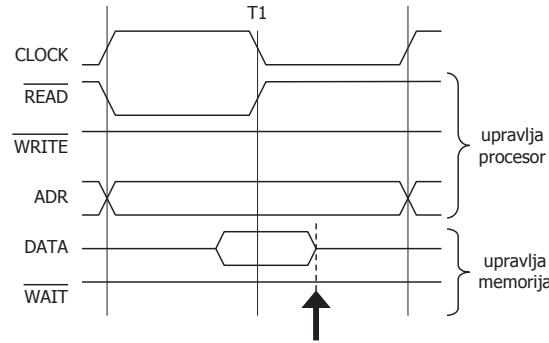


- Nakon vremena pristupa memorije, memorija postavlja traženi podatak na podatkovnu sabirnicu DATA; memorija je dužna ovo napraviti prije padajućeg brida signala CLOCK

## Protokol čitanja



- Sabirnički protokol čitanja bez stanja čekanja:

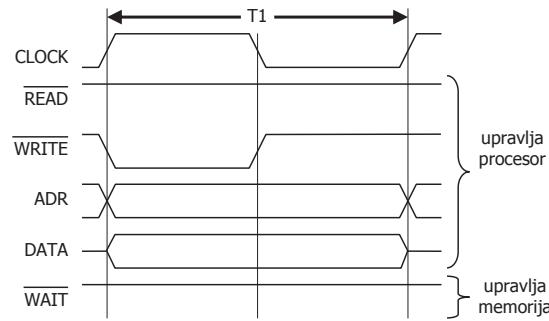


- Memorija prepoznaće da je READ deaktiviran i nakon nekog vremena osloboda podatkovnu sabirnicu postavljajući je u stanje visoke impedancije

## Protokol pisanja



- Sabirnički protokol pisanja bez stanja čekanja:

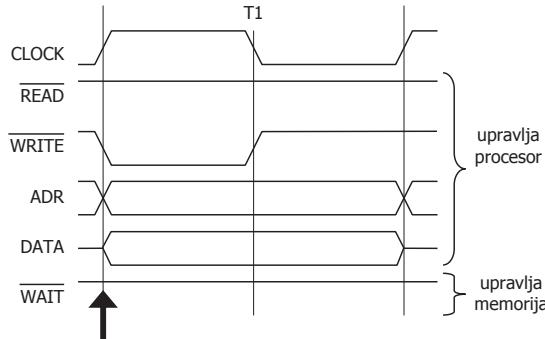


- Završetak trenutačne transakcije i početak nove

## Protokol pisanja



- Sabirnički protokol pisanja bez stanja čekanja:



- 1) Početak je na rastući brid CLOCK-a: procesor postavlja adresu na ADR, podatak na DATA i istovremeno aktivira WRITE čime naznačuje da želi pisati

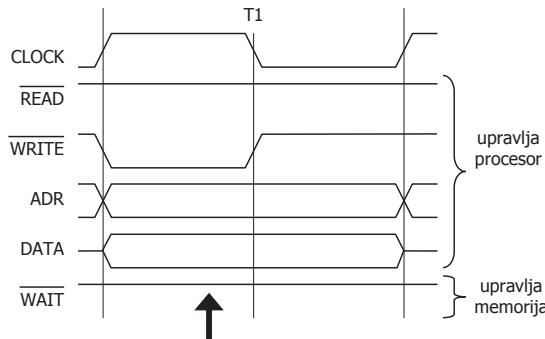
© Kovač, Basch, FER, Zagreb

61

## Protokol pisanja



- Sabirnički protokol pisanja bez stanja čekanja:



- 3) Nakon vremena pristupa memorije, memorija podatak s podatkovne sabirnice DATA pamti na zadanoj adresi; memorija je dužna ovo napraviti prije padajućeg brida signala CLOCK

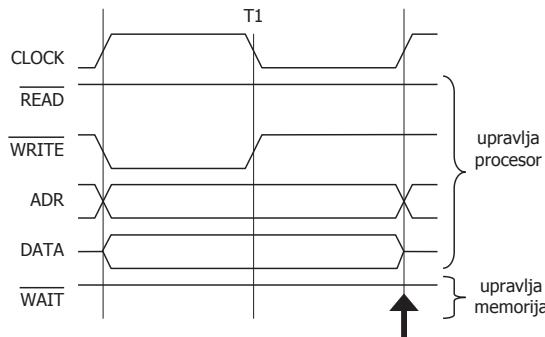
© Kovač, Basch, FER, Zagreb

63

## Protokol pisanja



- Sabirnički protokol pisanja bez stanja čekanja:



- 5) Procesor oslobađa podatkovnu sabirnicu DATA čime završava trenutačna transakcija i započinje nova

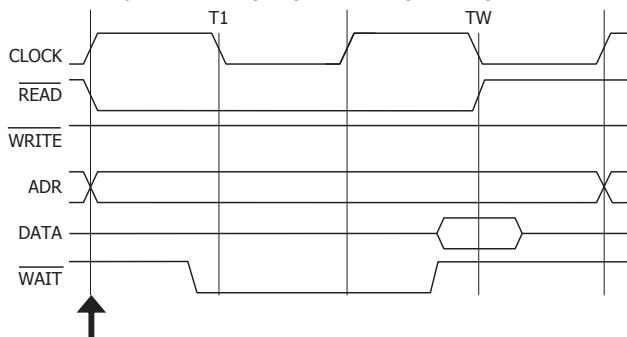
© Kovač, Basch, FER, Zagreb

65

## Čitanje sporih memorija



- Sabirnički protokol čitanja s jednim stanjem čekanja:



- 2) Memorija na temelju brida na signalu READ zna da se na adresnoj sabirnici nalazi adresa; memorija započinje s dekodiranjem adrese i s dohvatom adresiranog podatka

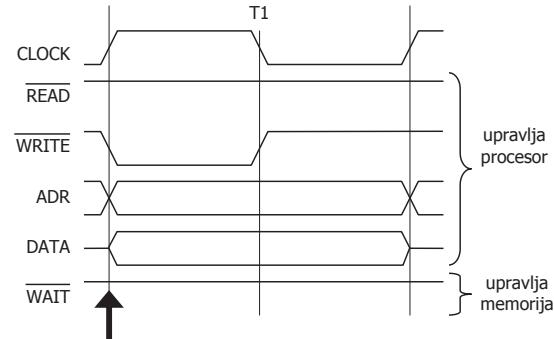
© Kovač, Basch, FER, Zagreb

67

## Protokol pisanja



- Sabirnički protokol pisanja bez stanja čekanja:



- 2) Memorija na temelju brida na signalu WRITE zna da se na adresnoj sabirnici nalazi adresa; memorija započinje s dekodiranjem adrese

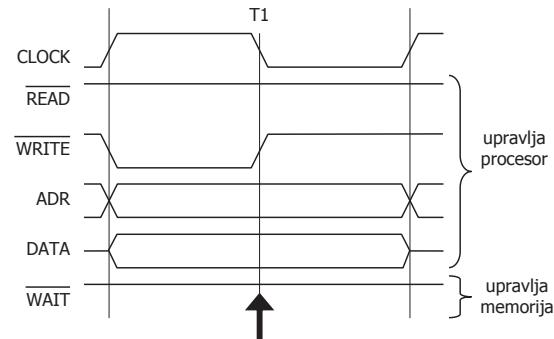
© Kovač, Basch, FER, Zagreb

62

## Protokol pisanja



- Sabirnički protokol pisanja bez stanja čekanja:



- 4) Na padajući brid signala CLOCK, procesor ispituje WAIT koji je neaktiviran pa procesor zna da je memorija stigla zapamtiti podatak; procesor deaktivira WRITE

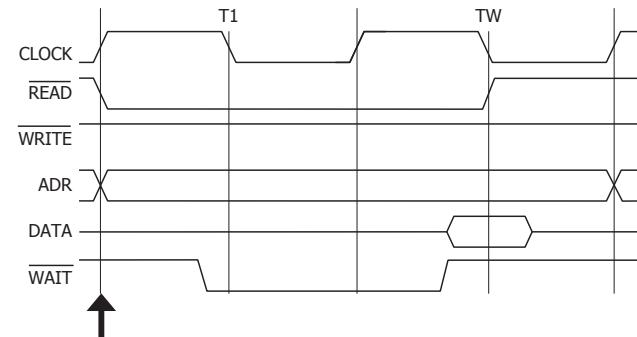
© Kovač, Basch, FER, Zagreb

64

## Čitanje sporih memorija



- Sabirnički protokol čitanja s jednim stanjem čekanja:



- 1) Početak je na rastući brid CLOCK-a: procesor postavlja adresu na ADR i istovremeno aktivira READ čime naznačuje da želi čitati

>>>

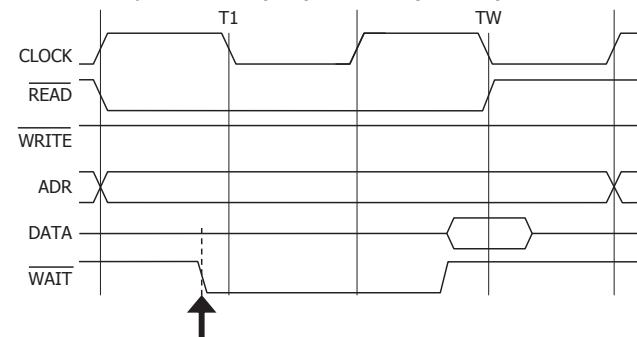
© Kovač, Basch, FER, Zagreb

66

## Čitanje sporih memorija



- Sabirnički protokol čitanja s jednim stanjem čekanja:



- 3) Budući da je memorija spora, mora prije padajućeg brida CLOCK-a aktivirati WAIT i zatražiti dodatni ciklus čekanja (TW)

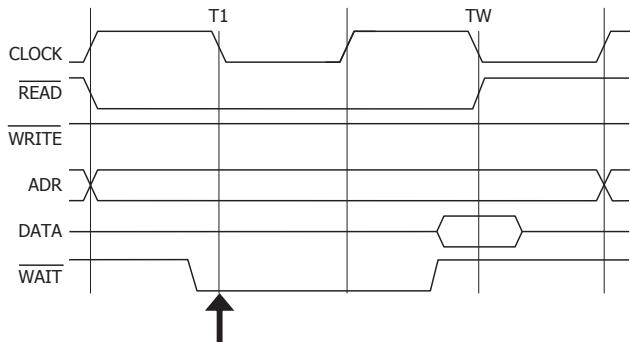
© Kovač, Basch, FER, Zagreb

68

## Čitanje sporih memorija



- Sabirnički protokol čitanja s jednim stanja čekanja:



4) Na padajući brid signala CLOCK, procesor ispituje WAIT koji je aktivan pa procesor zna da treba ubaciti ciklus čekanja; procesor ne preuzima podatak sa DATA i sve se odgađa jedan takt CLOCK-a

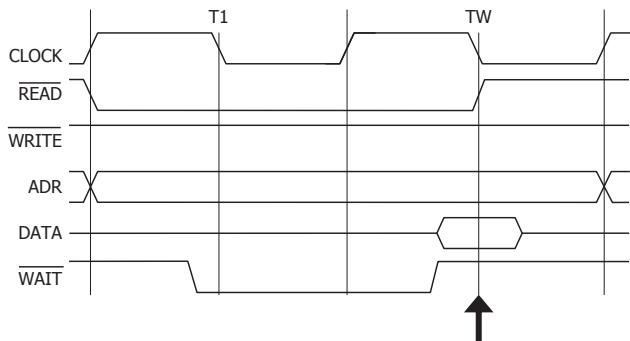
© Kovač, Basch, FER, Zagreb

69

## Čitanje sporih memorija



- Sabirnički protokol čitanja s jednim stanja čekanja:



6) Na padajući brid signala CLOCK u TW, procesor ponovno ispituje WAIT koji je sada neaktivan pa procesor preuzima podatak sa DATA i deaktivira READ

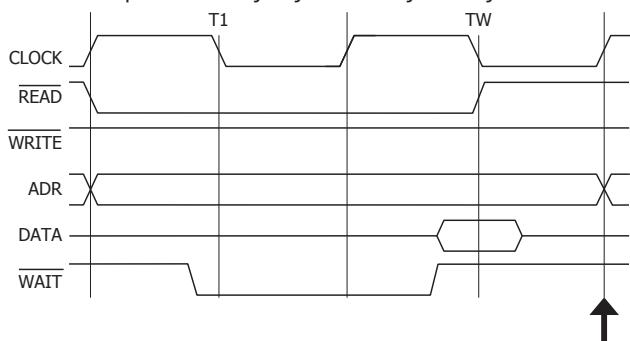
© Kovač, Basch, FER, Zagreb

71

## Čitanje sporih memorija



- Sabirnički protokol čitanja s jednim stanja čekanja:



8) Završetak trenutačne transakcije i početak nove

© Kovač, Basch, FER, Zagreb

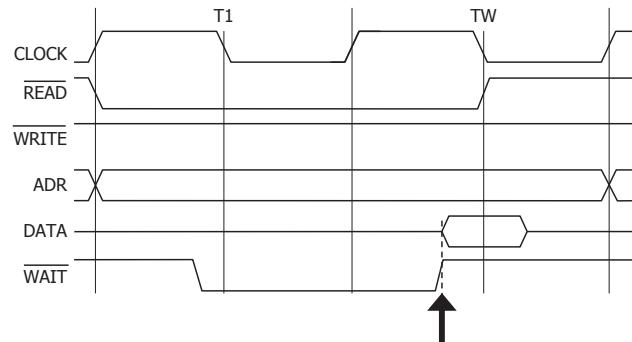
73

## Spajanje FRISC-a i memorije



## Čitanje sporih memorija

- Sabirnički protokol čitanja s jednim stanja čekanja:



5) Nakon vremena pristupa memorije, memorija postavlja traženi podatak na podatkovnu sabirnicu DATA i istovremeno deaktivira WAIT naznačujući da je obavila traženu operaciju

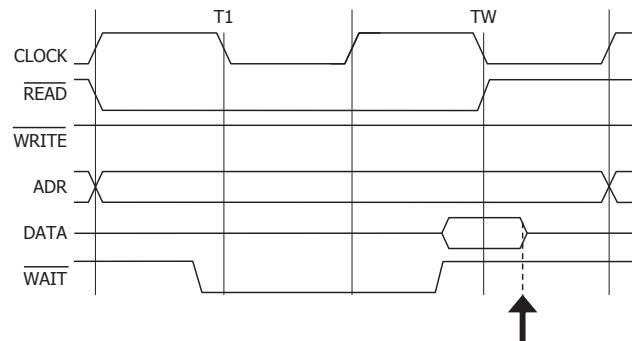
© Kovač, Basch, FER, Zagreb

70

## Čitanje sporih memorija



- Sabirnički protokol čitanja s jednim stanja čekanja:



7) Memorija prepoznaće da je READ deaktiviran i nakon nekog vremena oslobađa podatkovnu sabirnicu postavljajući je u stanje visoke impedancije

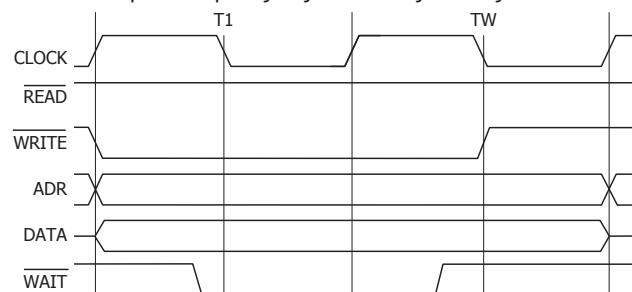
© Kovač, Basch, FER, Zagreb

72

## Pisanje u spore memorije



- Sabirnički protokol pisanja s jednim stanja čekanja:



Analogno pisanju bez čekanja što se tiče redoslijeda koraka;  
Analogno čitanju s čekanjem što se tiče dodavanja ciklusa čekanja

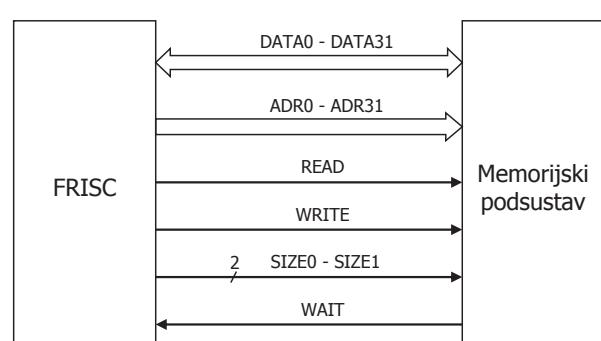
© Kovač, Basch, FER, Zagreb

74

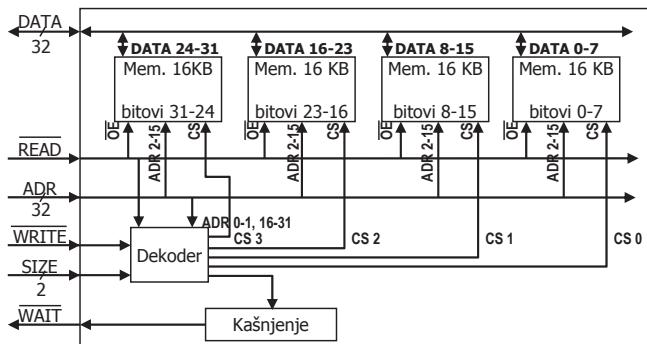
## Spajanje FRISC-a i memorije



- Načelna shema spajanja:



- Načelna shema memorijskog podsustava (primjer za memorijski prostor veličine 64 kB):



© Kovač, Basch, FER, Zagreb

77

## Memorijski podsustav

- Memorijski blok aktiviran pomoću CS:
  - postavlja podatak s tražene adrese na sabirnicu podataka onda kad mu je aktiviran ulaz OE (output enable) koji se aktivira kad se aktivira READ - čitanje
  - pamti podatak sa sabirnice podataka u lokaciji sa zadanom adresom onda kad mu je ulaz OE neaktiviran (kad je READ neaktiviran, tj. kad je WRITE aktiviran)
- Dekoder mora na temelju bitova ADR 16-31 adrese "raspozнати" da li je na sabirničkoj adresi iz ispravnog opsega - samo onda se aktiviraju izlazi iz dekodera
- Dekoder aktivira izlaze samo kad se radi o pristupu memoriji što se prepoznaje ili po aktiviranju signala READ ili po aktiviranju signala WRITE

© Kovač, Basch, FER, Zagreb

79

## Memorijski podsustav

| ADR 0-31 | ADR 2-15 | za SIZE=01 (BYTE): CS(3,2,1,0) | za SIZE=10 (HALFW.): CS(3,2,1,0) | za SIZE=11(WORD): CS(3,2,1,0) |
|----------|----------|--------------------------------|----------------------------------|-------------------------------|
| 0        | 0        | 0001                           | 0011                             | 1111                          |
| 1        | 0        | 0010                           | -                                | -                             |
| 2        | 0        | 0100                           | 1100                             | -                             |
| 3        | 0        | 1000                           | -                                | -                             |
| 4        | 4        | 0001                           | 0011                             | 1111                          |
| 5        | 4        | 0010                           | -                                | -                             |
| 6        | 4        | 0100                           | 1100                             | -                             |
| 7        | 4        | 1000                           | -                                | -                             |
| 8        | 8        | 0001                           | 0011                             | 1111                          |
| 9        | 8        | 0010                           | -                                | -                             |
| 10       | 8        | 0100                           | 1100                             | -                             |
| 11       | 8        | 1000                           | -                                | -                             |
| 12       | 12       | 0001                           | 0011                             | 1111                          |
| 13       | 12       | 0010                           | -                                | -                             |
| ...      | ...      | ...                            | ...                              | ...                           |

© Kovač, Basch, FER, Zagreb

81

## Izvođenje naredaba i protočna struktura

Ovaj dokument namijenjen je isključivo za osobnu upotrebu studentima Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

Svaka druga upotreba ili umnožavanje bilo kojeg dijela ovog dokumenta nije dozvoljena bez pismene dozvole autora.

© Mario Kovač, Danko Basch



© Kovač, Basch, FER, Zagreb

- Pretpostavka je da na raspolaganju imamo 4 memorija čipa sa širinom riječi od 1 bajta i kapacitetom 16KB (16\*1024 bajta) - čipovi imaju 14-bitnu adresu, a cijela memorija ima 64KB, tj. ima 16-bitnu adresu
- Zbog adresiranja memorije po bajtovima te zbog pristupanja pojedinim bajtovima, poluirječima i riječima, moraju postojati blokovi memorije za svaki bajt (1 čip je jedan blok, zeleni pravokutnici na slici)
- Podatkovni priključci svakog memorijskog bloka spojeni su na odgovarajući "bajt" sabirnice podataka DATA (24-31, 16-23, 8-15 i 0-7)
- Adresni priključci svakog memorijskog bloka spojeni su na bitove 2 do 15 adresne sabirnice ADR (to je točno 16 K adresa što je kapacitet svakog bloka). Bitovi 0 i 1 koriste se u odabiru pojedinog bloka (pomoću dekodera)
- Memorijski blok se aktivira onda kad mu je aktiviran kontrolni ulaz CS (chip select) (ulazima CS upravlja dekoder)

© Kovač, Basch, FER, Zagreb

78

## Memorijski podsustav

- Dekoder na temelju bitova SIZE određuje koji memorijski blokovi se trebaju aktivirati (vidi tablicu na sljedećem slajdu):
  - Npr., ako je SIZE = 01<sub>2</sub>, onda se čita bajt i aktivira se samo jedan memorijski blok, ovisno o bitovima ADR 0-1
  - Npr., ako je SIZE = 10<sub>2</sub>, onda se čita 16-bitna poluirječ i aktiviraju se dva memorijskih bloka, ovisno o bitu ADR 1
  - Npr., ako je SIZE = 11<sub>2</sub>, onda se čita 32-bitna riječ i aktiviraju se sva četiri memorijskih bloka

| Mem. 16KB<br>bitovi 31-24 | Mem. 16 KB<br>bitovi 23-16 | Mem. 16 KB<br>bitovi 8-15 | Mem. 16 KB<br>bitovi 0-7 |
|---------------------------|----------------------------|---------------------------|--------------------------|
| 3                         | 2                          | 1                         | 0                        |
| 7                         | 6                          | 5                         | 4                        |
| 11                        | 10                         | 9                         | 8                        |
| ...                       | ...                        | ...                       | ...                      |

adrese bajtova u blokovima

© Kovač, Basch, FER, Zagreb

80

## Memorijski podsustav

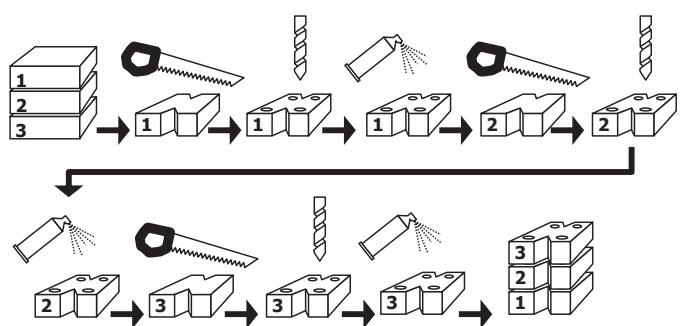
- Sklop za kašnjenje mora postojati ako je memorija sporija od brzine procesora FRISC. Kašnjenje se aktivira u trenutku kad dekoder raspozna da se pristupa ovoj memoriji.
- Sklop za kašnjenje mora se posebno projektirati i on mora aktivirati signal WAIT onoliko dugo koliko je memoriji potrebno da obavi zadatu operaciju čitanja ili pisanja.
- Sklop za kašnjenje može se napraviti, npr., korištenjem pomačnog registra povezanog na signal vremenskog vođenja radi sinkronizacije s procesorom.

© Kovač, Basch, FER, Zagreb

82

## Efikasno izvođenje poslova

- Prije diskusije o izvođenju naredaba na procesoru pogledajmo jedan neračunarski primjer: izrada proizvoda koje treba izrezati, izbušiti i lakirati.



Za izradu 3 proizvoda treba ukupno 9 koraka.

© Kovač, Basch, FER, Zagreb

2

## Efikasno izvođenje poslova



- Ovakav primitivni pristup izvođenju poslova još davnio je odbačen kao vrlo neefikasan.
- U 1901. Ransome Eli Olds uveo je u industriju prve verzije linije za proizvodnju.
- Henry Ford je 1913-14 uveo prve pokretnе trake u sklapanju automobila u tvornici Ford Highland Park, Michigan.
- Pojedini radnik radio je isključivo mali dio posla (npr. montirao volan i ništa osim toga) te je slijedeći radnik radio drugi mali dio posla i tako za cijelu proizvodnju.
- Ovakva organizacija proizvodnje uvela je revoluciju u proizvodnju automobila zbog svojih velikih ušteda:
  - vrijeme za sklapanje automobila (model T) smanjilo se s preko 12 sati na 93 minute

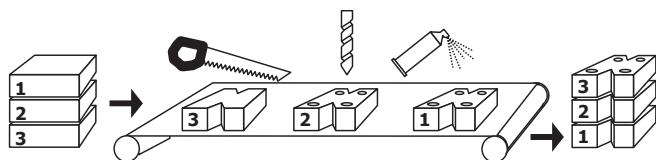
© Kovač, Basch, FER, Zagreb

3

## Efikasno izvođenje poslova



- Puno brže i efikasnije je upotrijebiti pokretnu traku i obavljati sva tri koraka istodobno (bitno je da su koraci međusobno NEOVISNI i da jedan drugom ne "smetaju"):



Za izradu 3 proizvoda ukupno treba 5 koraka.

Nakon što prvi proizvod bude napravljen u 3 koraka, svi sljedeći proizvodi izlaze u jednom koraku !!

Uvjet koji ne smijemo zaboraviti: za ovakvu proizvodnju treba nam više radne snage: 3 radnika !!! -> Cijena rada sustava je veća.

© Kovač, Basch, FER, Zagreb

4

## Izvođenje naredaba u procesoru



### • Podsjetnik - tipične faze u izvođenju naredbe\* su:

- dohvat naredbe iz memorije (fetch)
- dekodiranje naredbe (decode)
- izvođenje naredbe (execute)

\* Moguće su i drugačije podjele na faze (obično na veći broj faza: npr. faza izvođenja se može podjeliti na dohvati operanada, izvođenje ALU-operacije i spremanje rezultata itd.)

© Kovač, Basch, FER, Zagreb

5

## Izvođenje naredaba i protočna struktura



- Podjelimo li faze tako da su međusobno neovisne, možemo primjeniti načela pokretnih traka, pa se sve tri faze mogu izvoditi istodobno:



- Ovakva organizacija izvođenja se u literaturi naziva cjevodov (engl. pipeline), a mi ga nazivamo **protočna struktura**.
- Svaka **razina** protočne strukture (engl. pipeline stage) izvodi jednu fazu.
- Glavna namjena je **ubrzanje izvođenja**

© Kovač, Basch, FER, Zagreb

7

## Izvođenje naredaba i protočna struktura



- Podijelimo li naredbu na **N faza**, onda možemo izračunati ubrzanje:
  - u sljедnjem izvođenju:
    - za svaku naredbu treba N vremenskih perioda
    - za M naredaba sljedno izvođenje traje:  $M * N$
  - u protočnom izvođenju:
    - prva naredba traje N perioda, a svaka sljedeća traje 1 period
    - za M naredaba protočno izvođenje traje:  $N + (M-1)$
  - za veliki M vrijedi:  $(M*N) / (N+M-1) \approx (M*N) / (M) = N$
- Protočno izvođenje u N razina je u prosjeku N puta brže od izvođenja korak-po-korak** (uz pretpostavku linearne programa bez hazarda - više o tome kasnije)

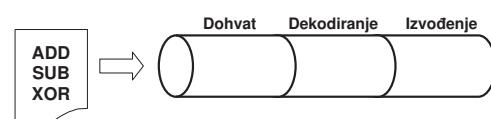


© Kovač, Basch, FER, Zagreb

6

## Izvođenje naredaba i protočna struktura

- Sada se prije spomenute tri naredbe ADD,SUB i XOR mogu puno efikasnije i brže izvesti:



|                                                                    |                |                  |               |
|--------------------------------------------------------------------|----------------|------------------|---------------|
| ADD                                                                | Dohvati<br>ADD | Dekodiraj<br>ADD | Izvedi<br>ADD |
|                                                                    | Dohvati<br>SUB | Dekodiraj<br>SUB | Izvedi<br>SUB |
|                                                                    | Dohvati<br>XOR | Dekodiraj<br>XOR | Izvedi<br>XOR |
| T1      T2      T3      T4      T5      T6      T7      T8      T9 |                |                  |               |
|                                                                    |                |                  |               |
|                                                                    |                |                  |               |

© Kovač, Basch, FER, Zagreb

8

## Izvođenje naredaba i protočna struktura



- Radi većeg ubrzanja, dobro bi bilo naredbu podijeliti na što više faza (tj. protočnu strukturu na što više razina)
- Brzina cijele protočne strukture ovisi o brzini najsporije razine, tj. najsportija razina predstavlja ograničenje (usko grlo)
- Zato se pokušavaju odabrati faze tako da svaka traje čim kraće, ali i tako da sve imaju podjednako trajanje
  - Primjer brzih operacija su interne operacije u procesoru (npr. rad s registrima, jednostavne AL operacije itd.)
  - Primjer vrlo sporih operacija su naredbe pisanja/čitanja iz memorije



- RISC arhitekture upravo koriste navedena načela da se izvodi puno brzih i jednostavnih naredaba čime se vrijeme izvođenja skraćuje
- Zato se protočna struktura i počela koristiti s pojavom procesora RISC arhitekture
- Danas se, zbog napretka tehnologije, protočnost koristi u većini procesora bez obzira jesu li RISC ili CISC
- U nastavku ćemo definirati protočnu strukturu procesora FRISC

## Protočna struktura FRISC-a

- Kako bi zadržali jednostavnost našeg procesora, za efikasnije izvođenje naredaba ćemo najjednostavniju moguću protočnu strukturu
- Odabiremo protočnu strukturu FRISC-a sa dvije razine, koje ćemo nazvati:**
  - razina za dohvata**
  - razina za izvođenje**



## Podjela naredaba po brzini izvođenja



- Trajanje operacija u svakoj razini je jedan period signala vremenskog vođenja CLOCK-a (uz pretpostavku da memorija nije spora)
- Naredbe procesora FRISC razlikuju se po načinu kako se izvode:
  - Faza dohvata svih naredaba traje jedan period**
  - Faza izvođenja također traje jedan period, ali kod nekih naredaba nije moguće preklapanje s fazom dohvata sljedeće naredbe**

## Podjela naredaba po brzini izvođenja



- Kod složenijih naredaba nije moguće preklapanje faze izvođenja s fazom dohvata sljedeće naredbe
  - Takve naredbe zovu se **dvociklusne** naredbe jer im efektivno vrijeme izvođenja u protočnoj strukturi iznosi dva perioda (ciklusa) CLOCK-a.
  - Ove naredbe ne koriste efikasno protočnu strukturu
- U dvociklusne naredbe spadaju:
  - Memorijske naredbe
  - Upravljačke naredbe

## Izvođenje naredaba i protočna struktura procesora FRISC

## Protočna struktura FRISC-a



- Podjela operacija između razina je ovakva:
  - Razina za dohvata**
    - Dohvat naredbe
    - Dekodiranje naredbe
    - Dohvat operanada
  - Razina za izvođenje**
    - Izvođenje AL operacije
    - Spremanje rezultata

## Podjela naredaba po brzini izvođenja



- Kod jednostavnih naredaba moguće je preklapanje faze izvođenja sa fazom dohvata sljedeće naredbe.
  - Ove naredbe zovu se **jednociklusne** jer im efektivno vrijeme izvođenja u protočnoj strukturi iznosi jedan period (ciklus) CLOCK-a.
  - Ove naredbe efikasno koriste protočnu strukturu.
- U jednociklusne naredbe spadaju:
  - Aritmetičko-logičke naredbe
  - Registerske naredbe

## Jednociklusne naredbe



- Izvođenje slijeda jednociklusnih naredaba **N1, ..., N4**, u razinama **Dohvat** i **Izvođenje** tijekom vremenskih trenutaka **Ti**, možemo simbolički prikazati ovako (sličan primjer izvođenja već je pokazan na slajdu 8):

|    |               |               |               |                            |
|----|---------------|---------------|---------------|----------------------------|
| N1 | Dohvati<br>N1 | Izvedi<br>N1  |               |                            |
| N2 |               | Dohvati<br>N2 | Izvedi<br>N2  |                            |
| N3 |               |               | Dohvati<br>N3 | Izvedi<br>N3               |
| N4 |               |               |               | Dohvati<br>N4 Izvedi<br>N4 |

T1      T2      T3      T4      T5



## Razina dohvata:

Rastući brid CLOCK-a:

$PC \rightarrow AR$ , A0-A31

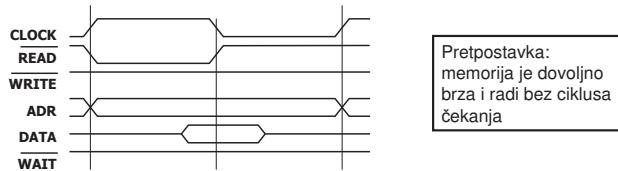
$0 \rightarrow READ$  // ovaj redak ćemo nadalje ispuštati

Padajući brid CLOCK-a:

$PC+4 \rightarrow PC$

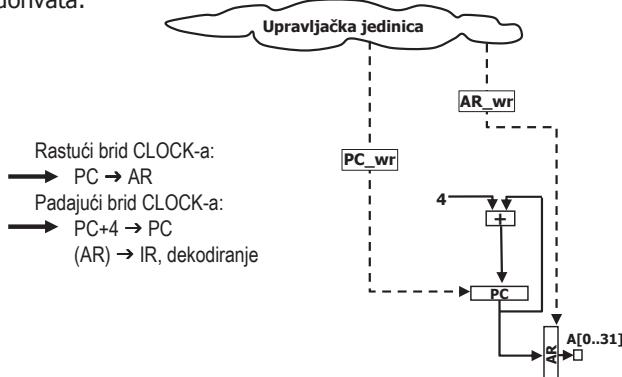
$(AR) \rightarrow IR$ , dekodiranje

$1 \rightarrow READ$  // ovaj redak ćemo nadalje ispuštati  
dohvata operanada i slanje u ALU (ako je potrebno)



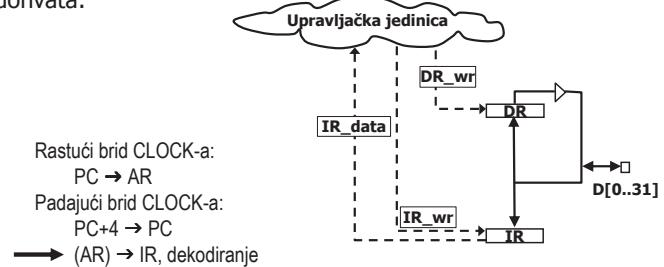
## Dohvat naredbe - arhitektura

- Do trenutka dekodiranja, **sve** naredbe imaju jednaku fazu dohvata:



## Dohvat naredbe - arhitektura

- Do trenutka dekodiranja, **sve** naredbe imaju jednaku fazu dohvata:



## Jednociklusne naredbe, primjer: ADD



### PRIMJER: izvođenje naredbe ADD R1, R2, R3

#### Razina dohvata:

Rastući brid CLOCK-a:

$PC \rightarrow AR$

Padajući brid CLOCK-a:

$PC+4 \rightarrow PC$

$(AR) \rightarrow IR$ , dekodiranje

$R1 \text{ i } R2 \rightarrow ALU$

#### Razina izvođenja:

Rastući brid CLOCK-a:

ALU: izvodi zbrajanje

Padajući brid CLOCK-a:

$ALU \rightarrow R3$

postavljanje zastavice u SR-u



## Jednociklusne naredbe, primjer: MOVE



### PRIMJER: izvođenje naredbe MOVE 100, R1

#### Razina dohvata:

Rastući brid CLOCK-a:

$PC \rightarrow AR$

Padajući brid CLOCK-a:

$PC+4 \rightarrow PC$

$(AR) \rightarrow IR$ , dekodiranje

ext 100 → ALU

#### Razina izvođenja:

Rastući brid CLOCK-a:

ALU: samo proslijedi drugi operand

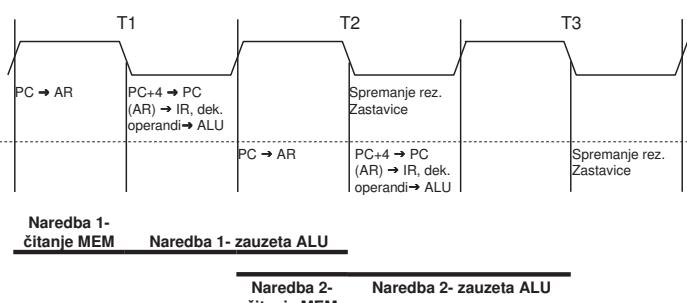
Padajući brid CLOCK-a:

$ALU \rightarrow R1$

## Jednociklusne naredbe

Iz primjera se može uočiti da se aritmetičke i registarske naredbe izvode na isti način što se tiče redoslijeda pojedinih operacija.

Pogledajmo sada kako se prilikom izvođenja jednociklusnih naredaba preklapaju faze dohvata i izvođenja



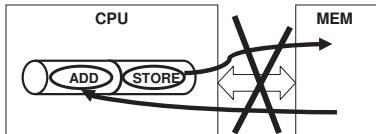
## Hazardi

- Do sada objašnjeni rad protočne strukture za jednociklusne naredbe bio je **idealan**, no u stvarnom radu postoje situacije koje uzrokuju da protočna struktura djelomično gubi svoju efikasnost
- U nekim situacijama protočna struktura ne može obraditi sljedeću naredbu odmah u sljedećem periodu
- Takve situacije nazivaju se **hazardi**
- Postoje tri osnovna tipa hazarda:
  - Strukturni**
  - Upravljački**
  - Podatkovni**

## Strukturalni hazard



- Strukturalni hazard je pojava kad procesor u određenom trenutku ne može izvesti sve faze onih naredaba koje se nalaze u protočnoj strukturi, jer struktura (tj. sklopovlje) procesora ne omogućuje istodobno izvođenje svih tih faza
- Jednostavan primjer strukturalnog hazarda je izvođenje naredbe STORE kod procesora s Von Neumannovom arhitekturom. Struktura memorijskog sučelja (Von Neumannova arhitektura) ne dozvoljava istovremeno dva pristupa memoriji:
  - jedan pristup treba za izvođenje naredbe STORE (spremanje podatka)
  - drugi pristup je dohvata strojnog kôda sljedeće naredbe (npr. ADD)



© Kovač, Basch, FER, Zagreb

27

## Naredbe FRISC-a i strukturalni hazard



- Zbog Von Neumannove arhitekture memorijskog sučelja, sve memorijске naredbe procesora FRISC (LOAD, STORE, PUSH, POP) uzrokovat će strukturalni hazard.
- Te naredbe imati će fazu izvođenja koja se ne može preklapati s fazom dohvata sljedeće naredbe te će njihovo izvođenje efektivno trajati dva perioda. Zato ćemo te naredbe svrstati u grupu **dvociklusnih** naredaba.
- U nastavku će biti opisan način izvođenja tih naredaba

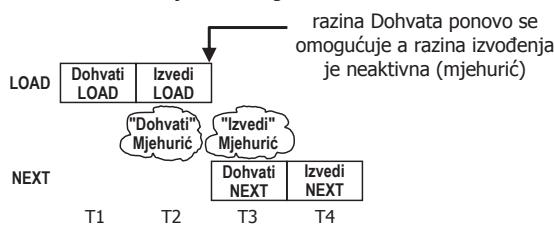
© Kovač, Basch, FER, Zagreb

29

## Izvođenje memorijskih naredaba



- Nakon izvođenja naredbe LOAD/STORE, spojni putovi prema memoriji su slobodni za dohvaćanje sljedeće naredbe
- Na kraju izvođenja se zato **omogućuje** rad razine za dohvat u sljedećem ciklusu
- Međutim, za vrijeme tog (odgođenog) dohvata bit će neaktivna razina za izvođenje (mjhurić prelazi iz razine dohvata u razinu izvođenja), jer nema dohvaćene naredbe koja bi se mogla izvoditi



© Kovač, Basch, FER, Zagreb

31

## Izvođenje memorijskih naredaba: STORE



**PRIMJER:** izvođenje naredbe `STORE R1, (R2+30)`

### Razina dohvata:

Rastući brid CLOCK-a:  
 $PC \rightarrow AR$   
Padajući brid CLOCK-a:  
 $PC+4 \rightarrow PC$   
 $(AR) \rightarrow IR$ , dekodiranje  
 $ext\ 30\ i\ R2 \rightarrow ALU$   
onemogući dohvat u sljedećem ciklusu

kao u prethodnim naredbama

### Razina izvođenja:

Rastući brid CLOCK-a:  
ALU: izvodi zbrajanje  
 $ALU \rightarrow DR$   
 $R1 \rightarrow DR$   
Padajući brid CLOCK-a:  
 $DR \rightarrow (AR)$   
omogući dohvat u sljedećem ciklusu

© Kovač, Basch, FER, Zagreb

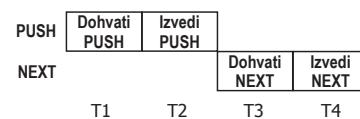
32

## Izvođenje memorijskih naredaba



### Memorijske naredbe PUSH i POP:

- Prilikom izvođenja naredaba PUSH i POP također se pristupa memoriji kao i kod naredba LOAD i STORE, pa se naredbe izvode na sličan način (na slici nisu prikazani mjhurići)
- Dodatno, naredba PUSH smanjuje register SP, a POP povećava SP
- Za povećanje odnosno smanjenje registra SP postoji poseban sklop u okviru registrarskog skupa.



33

© Kovač, Basch, FER, Zagreb

34



**PRIMJER:** izvođenje naredbe **PUSH R5**

#### Razina dohvata:

Rastući brid CLOCK-a:  
 $PC \rightarrow AR$   
 Padajući brid CLOCK-a:  
 $PC+4 \rightarrow PC$   
 $(AR) \rightarrow IR$ , dekodiranje  
 onemogući dohvat u sljedećem ciklusu  
 $R7-4 \rightarrow R7$

} kao u prethodnim  
naredbama

#### Razina izvođenja:

Rastući brid CLOCK-a:  
 $R7 \rightarrow AR$   
 $R5 \rightarrow DR$   
 Padajući brid CLOCK-a:  
 $DR \rightarrow (AR)$   
 omogući dohvat u sljedećem ciklusu

## Upravljački hazard



- Drugi od tri ranije spomenuta hazarda je **upravljački hazard**. Ovaj hazard dešava se kad naredba koja se nalazi u protočnoj strukturi i spremna je za izvođenje nije naredba koja se u stvari treba izvesti
- Ovaj hazard događa se kod izvođenja naredaba grananja kad je procesor već učitao sljedeću naredbu i pripremio se za njen izvođenje, ali zbog grananja program treba nastaviti s izvođenjem naredbe na nekoj drugoj adresi
- Zbog toga se ovaj hazard naziva još i hazardom grananja
- Naredbe koje uzrokuju ovaj hazard kod FRISC-a su upravljačke naredbe

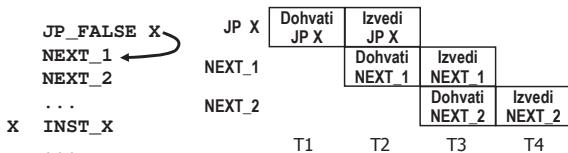
## Naredbe FRISC-a i upravljački hazard



#### • Upravljačke naredbe:

<<<

- **ako uvjet skoka nije istinit**, onda bi se moglo nastaviti s izvođenjem dohvaćene naredbe
- mnogi procesori rade upravo na ovaj način čime se ciklus gubi samo kad je to nužno (tj. kad se skok izvodi). Protočna razina mora imati mogućnost "poništavanja" dohvaćene naredbe.



## Izvođenje upravljačkih naredaba



#### • Specifičnosti pojedinih upravljačkih naredaba su:

- Adresa skoka zadana je brojem ili registrom: zbrajanje nije potrebno za izračun adrese skoka (za razliku od memorijskih naredaba s indirektnim registarskim adresiranjem s odmakom)
- Izuzetak je naredba JR u kojoj je zadana relativna adresa koja se pribraja registru PC, ali za to se ne koristi ALU, nego zasebni sklop za zbrajanje koji postoji uz registar PC
- Naredbe CALL i RET su specifične po tome što osim promjene tijeka izvođenja zahtjevaju dodatni pristup memoriji, tj. stoga.
  - Po tome su one slične naredbama PUSH i POP
- Pokažimo sada izvođenje ovih četiriju naredaba
  - zelenom bojom su označeni koraci koji ispituju uvjet i oni koji se izvode samo ako je uvjet istinit
  - to su ujedno koraci koji se međusobno razlikuju od naredbe do naredbe



**PRIMJER:** izvođenje naredbe **POP R5**

#### Razina dohvata:

Rastući brid CLOCK-a:  
 $PC \rightarrow AR$   
 Padajući brid CLOCK-a:  
 $PC+4 \rightarrow PC$   
 $(AR) \rightarrow IR$ , dekodiranje  
 ispitivanje istinitosti uvjeta  $Z=1$   
 ako  $Z=1$ : ext 1000  
 onemogući dohvat u sljedećem ciklusu

} kao u prethodnim  
naredbama

#### Razina izvođenja:

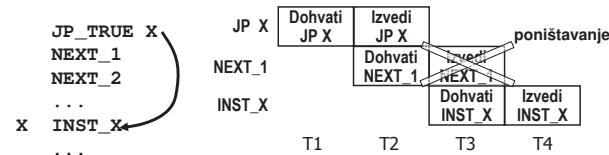
Rastući brid CLOCK-a:  
 $R7 \rightarrow AR$   
 $R7+4 \rightarrow R7$   
 Padajući brid CLOCK-a:  
 $(AR) \rightarrow DR$   
 $DR \rightarrow R5$   
 omogući dohvat u sljedećem ciklusu

## Naredbe FRISC-a i upravljački hazard



#### • Upravljačke naredbe:

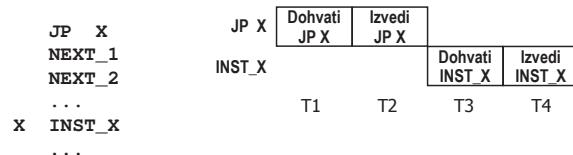
- Izvođenje upravljačkih naredaba može promijeniti normalni slijed izvođenja, ovisno o uvjetu, tj. ovisno o stanju zastavica
- Kada bi upravljačke naredbe bile jednociklusne, onda bi se tijekom njihovog izvođenja već dohvatila i dekodirala sljedeća naredba
  - međutim, **ako je uvjet skoka istinit**, onda bi trebalo "poništiti" dohvaćenu naredbu i započeti novi dohvat



## Naredbe FRISC-a i upravljački hazard



- Da bi pojednostavnili način izvođenja, u FRISC-u će naredbe skoka **uvijek biti** dvociklusne
- Nedostatak ovog rješenja je manja brzina izvođenja upravljačke naredbe u slučaju kad skok ne treba izvesti (tj. uvjet nije istinit)
- Kao i kod memorijskih naredaba, kad se dekodira upravljačka naredba, onemoguće je dohvat u sljedećem ciklusu, a nakon izvođenja upravljačke naredbe ponovno se omogućuje dohvat



## Izvođenje upravljačkih naredaba: JP



**PRIMJER:** izvođenje naredbe **JP\_EQ 1000**

#### Razina dohvata:

Rastući brid CLOCK-a:  
 $PC \rightarrow AR$   
 Padajući brid CLOCK-a:  
 $PC+4 \rightarrow PC$   
 $(AR) \rightarrow IR$ , dekodiranje  
 ispitivanje istinitosti uvjeta  $Z=1$   
 ako  $Z=1$ : ext 1000  
 onemogući dohvat u sljedećem ciklusu

} kao u prethodnim  
naredbama

#### Razina izvođenja:

Rastući brid CLOCK-a:  
 $R7 \rightarrow AR$   
 $R7+4 \rightarrow R7$   
 Padajući brid CLOCK-a:  
 $(AR) \rightarrow DR$   
 $DR \rightarrow R5$   
 omogući dohvat u sljedećem ciklusu

## Izvođenje upravljačkih naredaba: JR



**PRIMJER:** izvođenje naredbe `JR_NC 10`

### Razina dohvata:

Rastući brid CLOCK-a:  
 $PC \rightarrow AR$   
 Padajući brid CLOCK-a:  
 $PC+4 \rightarrow PC$   
 $(AR) \rightarrow IR$ , dekodiranje  
 ispitivanje istinitosti uvjeta:  $C=0$   
 ako  $C=0$ : ext 10  
 onemogući dohvati u sljedećem ciklusu

10 ovdje predstavlja relativni odmak  
 zapisan u strojnom kodu, a ne adresu  
 odredišta koja se inače piše u naredbi

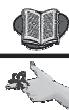
### Razina izvođenja:

Rastući brid CLOCK-a:  
 -  
 Padajući brid CLOCK-a:  
 ako  $C=0$ : ext 10 + PC  $\rightarrow$  PC  
 omogući dohvati u sljedećem ciklusu

© Kovač, Basch, FER, Zagreb

43

## Izvođenje upravljačkih naredaba: RET



**PRIMJER:** izvođenje naredbe `RET`

### Razina dohvata:

Rastući brid CLOCK-a:  
 $PC \rightarrow AR$   
 Padajući brid CLOCK-a:  
 $PC+4 \rightarrow PC$   
 $(AR) \rightarrow IR$ , dekodiranje  
 ispitivanje istinitosti uvjeta: True  
 onemogući dohvati u sljedećem ciklusu

### Razina izvođenja:

Rastući brid CLOCK-a:  
 ako True:  $R7 \rightarrow AR$   
 ako True:  $R7+4 \rightarrow R7$   
 Padajući brid CLOCK-a:  
 ako True:  $(AR) \rightarrow DR$   
 ako True:  $DR \rightarrow PC$   
 omogući dohvati u sljedećem ciklusu

© Kovač, Basch, FER, Zagreb

45

## Izvođenje niza naredaba - primjer



<<<

I bez crtanja stanja protočne strukture, trajanje se može jednostavno odrediti ako se zna koje naredbe su jednokiklanske, a koje dvokiklanske. Na primjer:

CMP ROTL JP MOVE POP SUB  
 1 1 2 1 2 1+1 → Σ=9

Dodajemo +1 jer iz zadnjeg ciklusa "nema ničega" pa računamo i ciklus izvođenja zadnje naredbe (tj. izlazak zadnje naredbe iz razine za dohvati ili praznjenje protočne strukture)\*

\* Alternativno gledanje je da se +1 dodaje na prvu naredbu, gdje dodatni ciklus predstavlja punjenje protočne strukture (tj. punjenje prve naredbe u razinu za dohvati)

© Kovač, Basch, FER, Zagreb

47

## Podatkovni hazard



- Podatkovni hazard** javlja se kod izvođenja naredaba u protočnoj strukturi kada se naredba ne može izvesti jer podaci potrebeni za njeno izvođenje još nisu spremni
- Kod FRISC-a nema pojave podatkovnog hazarda pa ćemo primjer ovog hazarda proučiti kod procesora ARM

© Kovač, Basch, FER, Zagreb

## Izvođenje upravljačkih naredaba: CALL



**PRIMJER:** izvođenje naredbe `CALL (R2)`

### Razina dohvata:

Rastući brid CLOCK-a:  
 $PC \rightarrow AR$   
 Padajući brid CLOCK-a:  
 $PC+4 \rightarrow PC$   
 $(AR) \rightarrow IR$ , dekodiranje  
 ispitivanje istinitosti uvjeta: True  
 onemogući dohvati u sljedećem ciklusu

kao u prethodnim naredbama

### Razina izvođenja:

Rastući brid CLOCK-a:  
 ako True:  $R7 \rightarrow AR$   
 ako True:  $PC \rightarrow DR$   
 Padajući brid CLOCK-a:  
 ako True:  $DR \rightarrow (AR)$   
 ako True:  $R2 \rightarrow PC$   
 omogući dohvati u sljedećem ciklusu

© Kovač, Basch, FER, Zagreb

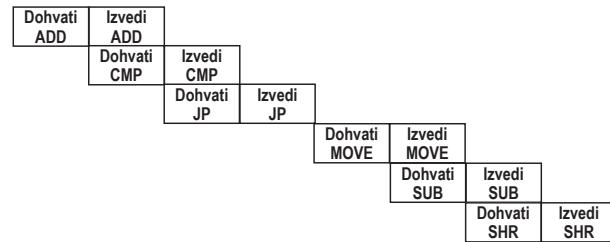
44

## Izvođenje niza naredaba - primjer



### Primjer

Odrediti trajanje izvođenja sljedećeg niza naredaba (zanemarujemo eventualno preklapanje za prvu i zadnju naredbu, jer nije definirano što se događa prije i poslije ovog programskega odsječka): ADD, CMP, JP, MOVE, SUB, SHR



© Kovač, Basch, FER, Zagreb

>>> 46

## Izvođenje niza naredaba - primjer



### Primjer

Odrediti trajanje izvođenja sljedećeg niza naredaba:

ADD SUB STORE MOVE SUB LOAD LOAD ADD HALT  
 1 1 2 1 1 2 2 1 2

Ukupno: 13 (na upravljačku naredbu HALT ne dodajemo +1 jer za nju smo već računali da traje 2 ciklusa)

### Primjer

Odrediti trajanje izvođenja sljedećeg niza naredaba:

LOAD SUB CALL MOVE SUB LOAD LOAD STORE  
 2 1 2 1 1 2 2 2

Ukupno: 13 (na memoriju naredbu STORE ne dodajemo +1 jer za nju smo već računali da traje 2 ciklusa)

© Kovač, Basch, FER, Zagreb

48

## Cjelovit rad procesora



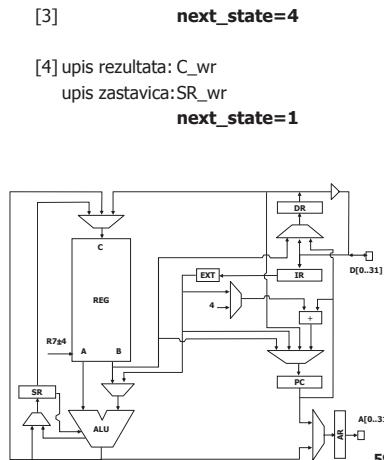
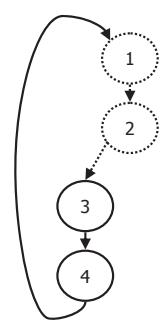
- U prethodnim predavanjima proučili smo arhitekturu puta podataka i detaljno proučili način izvođenja svih naredaba
- U nastavku ćemo ukratko objasniti na koji način funkcionira procesor kao cjelina, odnosno koja su načela upravljanja sa svim dijelovima puta podataka
- Prisjetimo se nakratko mikroarhitekturu puta podataka procesora FRISC

49

© Kovač, Basch, FER, Zagreb

50





© Kovač, Basch, FER, Zagreb

**Upravljačka jedinica - mikroprogramiranje**

- Mikroprogramiranje je metoda izvedbe kompleksnih upravljačkih jedinica. (npr. upravljačku jedinicu procesora Pentium bi bilo izuzetno teško izvesti pomoću FSM-a jer bi trebalo tisuće stanja i tisuće prelaza,... što bi bilo skupo i kompleksno )
- Kod takvih procesora upravljačka jedinica izvedena je kao jednostavno računalo unutar računala. Dohvat, dekodiranje, izvođenje naredaba te upravljanje radom procesora obavlja se izvođenjem internog mikroprograma.
- Da bi se naredbe za izvedbu upravljačke jedinice razlikovale od naredaba koje su dostupne programerima one se nazivaju mikronaredbama. One se izuzetno brzo izvode (RISC načelo) te omogućuju brz i efikasan rad procesora.

© Kovač, Basch, FER, Zagreb

© Kovač, Basch, FER, Zagreb

**6. Dio****Povezivanje računala s okolinom:  
ulazno-izlazni prijenos podataka**

Ovaj dokument namijenjen je isključivo za osobnu upotrebu studentima Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

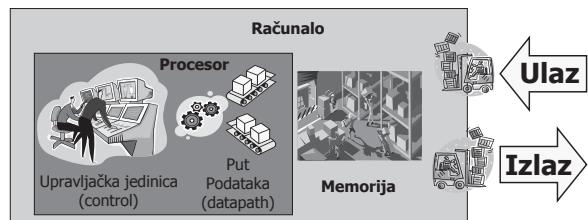
Svaka druga upotreba ili umnožavanje bilo kojeg dijela ovog dokumenta nije dozvoljena bez pismene dozvole autora.

© Mario Kovač, Danko Basch

© Kovač, Basch, FER, Zagreb

**Povezivanje računala s okolinom**

- Podsjetnik:
  - ulaz i izlaz služe za primanje i slanje podatka između računala i vanjskog svijeta
  - ulaz i izlaz se nalaze na "granicu" računala

**Uvod - Povezivanje računala s okolinom**

© Kovač, Basch, FER, Zagreb

**Povezivanje računala s okolinom**

- Bez ulaza i izlaza (input/output, IO) računalo bi bilo beskoristan i od svoje okoline izolirani uređaj:
  - ne bi mogli vidjeti rezultate obrade nikakvih podataka
  - ne bi čak mogli ni unijeti nikakve podatke koje bi željeli obraditi
  - ne bi bilo moguće zadati pokretanje programa za obradu podataka
  - također ne bi bilo moguće niti napuniti memoriju programom
  - s takvim računalom ne bi se moglo ništa raditi**

© Kovač, Basch, FER, Zagreb

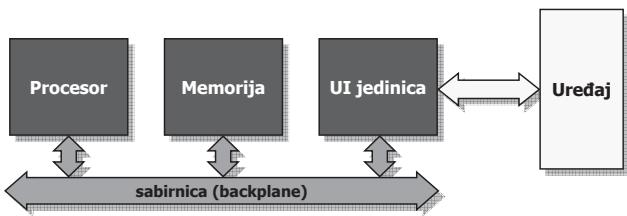
**Povezivanje računala s okolinom**

- Kako bi mogli povezati različite vanjske uređaje s računalom?
- Teoretski bi mogli uređaje spajati izravno na sabirnicu na koji su spojeni procesor i memorija, ali to je samo teorijski jer u praksi bi imali "nekoliko" problema:
  - mnogi uređaji daju ili očekuju analogne umjesto digitalnih signala
  - naponske razine na uređaju ne moraju odgovarati naponskim razinama na sabirnici
  - brzine uređaja su znatno različite od brzina na kojima rade procesor i memorija (obično su uređaji puno sporiji)
  - način komunikacije procesora i uređaja je različit
  - posluživanje određenih uređaja moglo bi zahtjevati veliki protok podataka što bi zauzimalo sabirnicu i zahtjevalo intenzivno posluživanje od strane procesora
  - itd...

## Povezivanje računala s okolinom



- Zato se uređaji nikada ne spajaju izravno na sabirnicu procesora već preko "ulaza i izlaza", tj. preko posebnih međusklopova namijenjenih upravo toj svrsi (oni služe kao posrednici)
- Ovakve međusklopove zovemo ulazno-izlaznim (UI) jedinicama (engl. IO unit) ili vanjskim jedinicama (skačeno VJ)



© Kovač, Basch, FER, Zagreb

6

## Povezivanje računala s okolinom



- Kakve sve mogu biti UI jedinice?

- Možemo ih podijeliti prema smjeru prijenosa:

- ulazne
- izlazne
- dvosmjerne

- Možemo ih podijeliti prema namjeni:

- za prijenos podataka
- za brojanje impulsa
- za mjerjenje vremena
- za generiranje impulsa
- za AD i DA pretvorbu
- specijalne namjene, itd.

© Kovač, Basch, FER, Zagreb

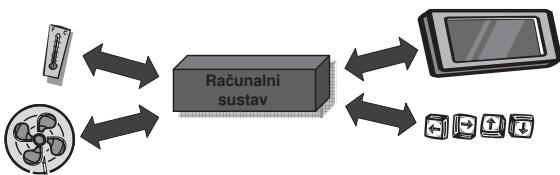
8

## Povezivanje računala s okolinom



- Podsjetnik:

- U našem projektu zamisili smo da je računalo povezano sa:
  - tipkama
  - zaslonom
  - mjeračem temperature
  - klima uređajem



© Kovač, Basch, FER, Zagreb

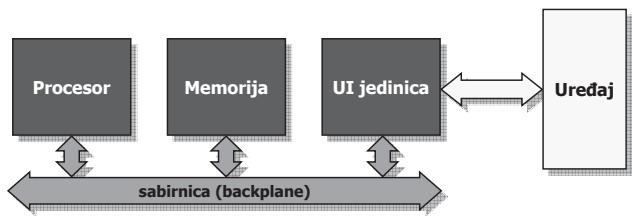
10

## Komunikacija procesora s UI jedinicama

## Povezivanje računala s okolinom



- Zadaća UI jedinice je oslobađanje procesora od učestale komunikacije s uređajem i prilagodba načina komunikacije uređaja komunikaciji putem sabirnice računala
- Na taj način procesor obavlja jednostavnu komunikaciju s UI jedinicom, a ona se dalje brine za komunikaciju s uređajem



© Kovač, Basch, FER, Zagreb

7

## Povezivanje računala s okolinom

- Što sve može biti uređaj?

- Senzor preko kojeg primamo podatke o vanjskom svijetu (npr. o tlaku, temperaturi, brzini vrtnje, itd.), npr. primamo podatke o procesu kojim upravljamo (u sebi sadrži AD pretvornik)
- Relej kojim uključujemo ili isključujemo neki uređaj kojim upravljamo (npr. motor, klima uređaj, alarm)
- Pisač na koji šaljemo tekst za ispis
- Zaslон i tipkovnica preko kojih komunicira korisnik
- itd.

© Kovač, Basch, FER, Zagreb

9

## Povezivanje računala s okolinom

- Na koji način se konkretno povezuju ovakvi ili bilo koji drugi uređaji s računalom?
- Kako je najpraktičnije komunicirati s ovakvim uređajima?
- Kako se komunicira s UI jedinicama i kako ih se adresira?
- itd...

- **Odgovore na ova i još neka pitanja dat će ovo poglavlje...**

© Kovač, Basch, FER, Zagreb

11

## Načini adresiranja UI jedinica

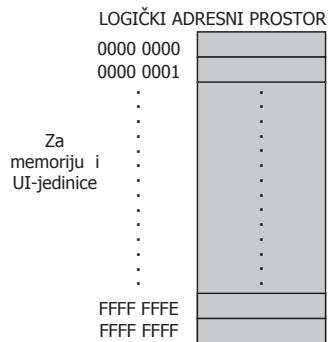


- Načelno, kod UI komunikacije postoje dvije operacije kao i kod pristupanja memoriji:
  - čitanje
  - pisanje
- Na sabirnicu je spojen veći broj UI jedinica i procesor mora točno odabrati onu jedinicu s kojom želi komunicirati - to se radi pomoću adresne sabirnice
  - analogna situacija je kad procesor pristupa jednoj od mnogobrojnih lokacija u memoriji - lokacija se odabire adresom
- Dva osnovna načina adresiranja UI jedinica su:
  - **memorijsko UI preslikavanje** (memory mapped IO)
  - **izdvojeno UI adresiranje** (isolated IO)



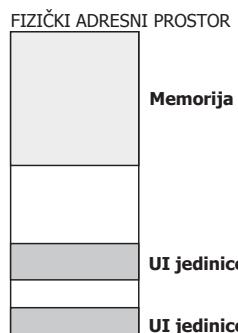
- Memorijsko UI preslikavanje ima sljedeće značajke:

- isti adresni prostor **dijele** memorija i UI jedinice



© Kovač, Basch, FER, Zagreb

14



© Kovač, Basch, FER, Zagreb

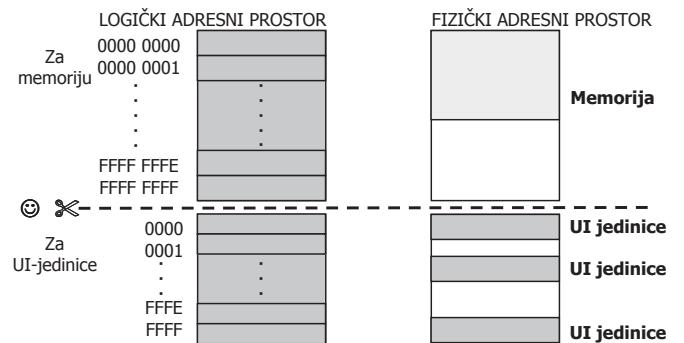
15

## Izdvojeno UI adresiranje



- Izdvojeno UI adresiranje ima sljedeće značajke

- adresni prostor za memoriju izdvojen je od adresnog prostora za UI jedinice (koji je obično manji od memorijskog prostora)



© Kovač, Basch, FER, Zagreb

16

## Vrste komunikacije procesora s UI jedinicama



- Usporedba memorijskog preslikavanja i izdvojenog adresiranja:
  - memorijsko preslikavanje je jednostavnije jer postoji samo jedan sabirnički protokol za pristup memoriji i UI jedinicama
  - kod memorijskog preslikavanja treba manje priključaka jer se ne koristi MEM/IO ili sličan priključak
  - kod memorijskog preslikavanja arhitektura procesora je jednostavnija jer ne postoje zasebne naredbe tipa IN i OUT
  - kod izdvojenog adresiranja UI adrese se "ne miješaju" s memorijskim adresama niti smanjuju memorijski adresni prostor
  - kod izdvojenog adresiranja procesor može na različite načine pristupati memoriji i UI jedinicama pa se komunikacija može bolje prilagoditi svojstvima memorije odnosno svojstvima UI jedinica čime komunikacija može biti efikasnija

© Kovač, Basch, FER, Zagreb

18

## Vrste UI prijenosa podataka



- UI jedinice rade svojom brzinom koja je **različita** (obično manja) od brzine procesora
- Prije prenošenja podataka, obično se procesor i UI jedinica trebaju **sinkronizirati**, tj. uskladiti svoj rad da bi se prijenos podataka uspješno izveo
- S obzirom na to, prijenos se može dijeliti na:
  - **programski prijenos**
  - **sklopovski prijenos**

© Kovač, Basch, FER, Zagreb

20



<<<

- Memorijsko UI preslikavanje ima sljedeće značajke:

- procesor na jednak način pristupa memorijskim lokacijama i UI jedinicama (ne razlikuje ta dva pristupa) - npr. naredbama LOAD i STORE
- sabirnički protokoli čitanja i pisanja jednaki su za pristup memoriji i UI jedinicama

© Kovač, Basch, FER, Zagreb

15

## Izdvojeno UI adresiranje



<<<

- Izdvojeno UI adresiranje ima sljedeće značajke

- procesor na različite načine pristupa memoriji i UI jedinicama. Na primjer:
  - memoriji pristupa naredbama LOAD i STORE
  - UI jedinicama pristupa specijalnim naredbama IN i OUT
- procesor ima posebne priključke (npr. MEM/IO ili MEMRQ i IORQ) pomoću kojih signalizira da li pristupa memoriji ili vanjskim jedinicama
  - po tome se razlikuje je li neka adresa (npr. 10), koja se nalazi na adresnoj sabirnici, namijenjena memoriji ili UI jedinici

© Kovač, Basch, FER, Zagreb

17

## Komunikacija FRISC-a s UI jedinicama



- Za FRISC ćemo odabrati memorijsko preslikavanje:

- bolje je prilagođeno RISC procesorima zbog veće jednostavnosti (samo jedan sabirnički protokol i nepotrebne dodatne naredbe)
- memorijski adresni prostor je više nego dovoljan za naše potrebe pa možemo jedan njegov dio odvojiti za UI jedinice
- komunikacija s UI jedinicama odvija se pomoću naredaba LOAD i STORE budući da su to jedine naredbe koje pristupaju memoriji
  - umjesto adresa memorijskih lokacija, u naredbama LOAD i STORE koristit će se adrese UI jedinica
- **u zadatcima i na labosima po dogovoru\*** uzimamo da je na FRISC spojena memorija na adresama 0 do FFFF (64 kB), a UI jedinice nalazit će se na adresama FFFF0000 do FFFFFFFF (sve možemo dohvatiti 20-bitnom adresom)

\* ako nije drugačije navedeno

© Kovač, Basch, FER, Zagreb

19

## Programski prijenos



- Glavne karakteristike **programskog prijenosa** su:

- Prijenos se obavlja **pod upravljanjem programa**, tj. prijenos obavlja procesor (upravljan programom)
- Svi podatci koji se prenose **"prolaze kroz procesor"**. Na primjer:
  - svaki podatak se prvo iz VJ učita u procesor, a zatim se spremi u memoriju ili koristi na neki drugi način (kod ulaznog smjera prijenosa)
  - svaki podatak će procesor prvo čitati iz memorije ili izračunavati ili dobavljati, a zatim će ga slati na VJ (kod izlaznog smjera prijenosa)
- Programski prijenos je **sporiji** od sklopovskog prijenosa
- Procesor dio vremena troši na prijenos podataka što **usporava izvođenje ostalih poslova**
- U nastavku ćemo vidjeti tri vrste programiranog prijenosa:
  - **bezuvjetni**
  - **uvjetni**
  - **prekidni**

© Kovač, Basch, FER, Zagreb

21



- Glavne karakteristike **sklopovskog prijenosa** su:

- prijenos se obavlja pod upravljanjem specijaliziranog sklopovlja, tj. **prijenos obavlja specijalna jedinica** (DMA kontroler), a procesor ne sudjeluje u prijenosu
- podaci koji se prenose prolaze kroz specijalnu jedinicu, a ovisno o njenoj gradi i organizaciji moguće je čak i da se prenose izravno između UI jedinice i memorije
- prijenos se općenito odvija **velikim brzinama**
- procesor ne troši vrijeme na prijenos podataka, ali **izvođenje programa je ipak usporeno** za vrijeme obavljanja prijenosa
- na kraju ovog poglavlja ćemo vidjeti jednu vrstu sklopovskog prijenosa:
  - **izravni pristup memoriji (DMA)**

## Što slijedi ?

- U ostatku poglavlja:

- detaljnije ćemo objasniti svaki od ovih načina prijenosa podataka
- pokazati ćemo nekoliko "standardnih" UI jedinica kakve postoje za većinu procesora

## Bezuvjetni prijenos



## Bezuvjetni prijenos

- **Bezuvjetni prijenos** je najjednostavnija vrsta prijenosa

- Glavna značajka je da se prije prijenosa **ne provjerava** je li UI jedinica spremna za prijenos podataka
  - nema sinkronizacije brzine rada između procesora i UI jedinice
- Dijagram toka je trivijalan i sastoji se samo od prijenosa:



- Prijenos je jedna naredba (eventualno više njih) za čitanje iz VJ ili za pisanje u VJ

## Bezuvjetni prijenos

- UI jedinica je sklopovski **najjednostavnija**
- **Najbrži** prijenos među programiranim prijenosima
- **Nedostatak je mogućnost da UI jedinica nije spremna za prijenos.** Može se dogoditi:
  - **Za ulaznu jedinicu:** procesor može pročitati podatak koji je prethodno već bio pročitao jer VJ nije "pripremila" novi podatak čime se višestruko "prima" isti podatak
  - **Za izlaznu jedinicu:** procesor može poslati novi podatak prije nego je VJ uspjela "obraditi" prethodni podatak. Ovisno o tome kako VJ radi, može doći do prepisivanja prethodnog podatka ili do zanemarenja novog podatka kojeg smo upravo poslali - u oba slučaja gube se podatci

## Osnovna građa bezuvjetne UI jedinice

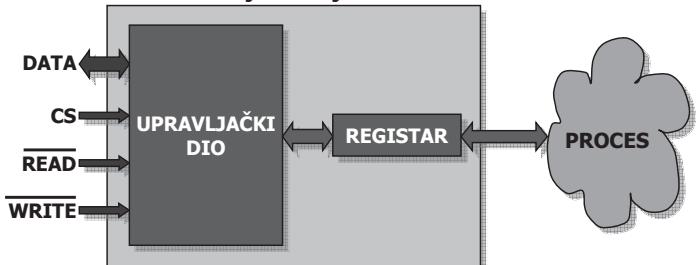


- Najjednostavnija građa opće bezuvjetne VJ može se prikazati sljedećom blok-shemom:

>>>

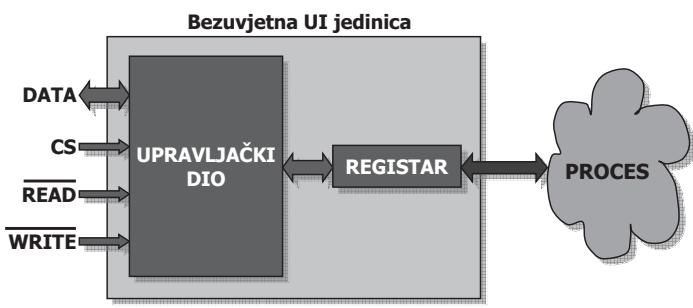


## Bezuvjetna UI jedinica





- Imma podatkovne priključke za povezivanje s procesom (vanjskim uređajem) preko kojih se prenose podaci:

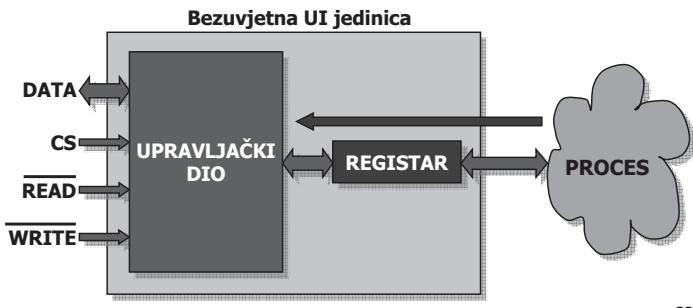


© Kovač, Basch, FER, Zagreb

30



- Registr služi za pamćenje podatka:
  - podatak je primljen iz vanjskog procesa, a pamti se do trenutka kad ga procesor pročita (ulazna VJ)
  - ako proces neprekidno drži podatak na podatkovnim priključcima prema VJ, onda registr nije nužan

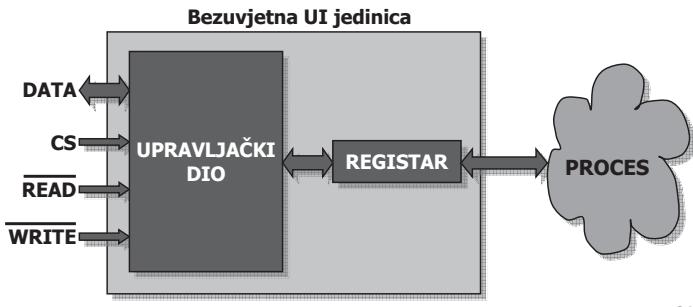


© Kovač, Basch, FER, Zagreb

32



- Ovakva VJ **zauzimala bi samo jednu lokaciju** preko koje bi se izvodilo čitanje ili pisanje
- VJ može biti namijenjena za prijenos podatka određene širine u bitovima (npr. 32-bitne riječi) ili može prenositi podatke različitih širina (tada mora imati i ulaze SIZE)



© Kovač, Basch, FER, Zagreb

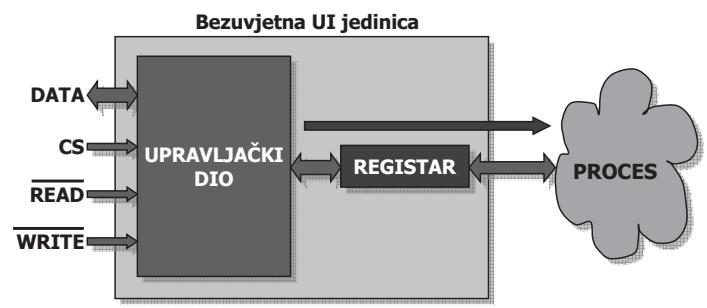
34



- Sve naše VJ zauzimat će određeni broj uzastopnih lokacija (jednu ili više njih)
- Zbog jednostavnosti, sve ove lokacije će biti 32-bitne pa ćemo s VJ komunicirati korištenjem naredaba LOAD i STORE
  - Dakle, svaka lokacija će zauzimati 4 adrese



- Registr služi za pamćenje podatka (tzv. registr podatka):
  - podatak se pamti do trenutka kad ga vanjski proces preuzeme (za izlaznu VJ)

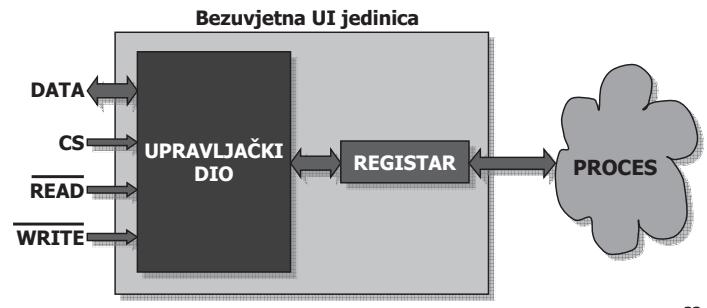


© Kovač, Basch, FER, Zagreb

31



- Upravljački dio na temelju stanja na priključcima za povezivanje s procesom upravlja radom VJ tako da ona izvede traženu operaciju (u ovom slučaju čitanje ili pisanje)

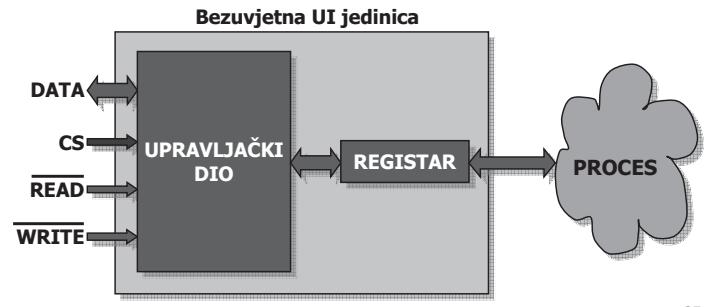


© Kovač, Basch, FER, Zagreb

33



- Pomoću ulaznog priključka CS (chip select) i vanjskog adresnog dekodera možemo smjestiti VJ bilo gdje u adresnom prostoru (fleksibilnost)

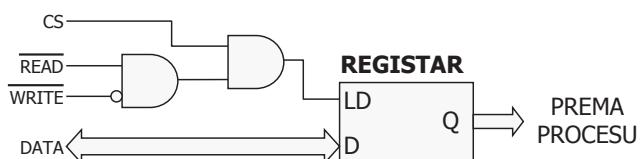


© Kovač, Basch, FER, Zagreb

35

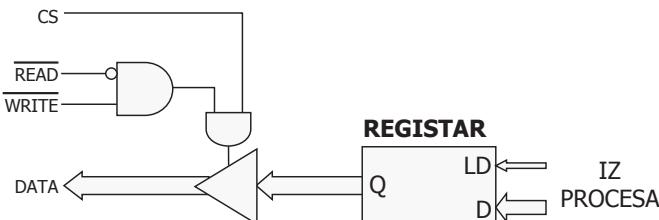


- Pojednostavljena građa upravljačkog dijela za **izlaznu** VJ može izgledati ovako:





- Pojednostavljena građa upravljačkog dijela za **ulaznu** VJ može izgledati ovako:



## Bezuvjetni prijenos - Primjeri

### Rješenje:

U programu su bezuvjetni prijenosi označeni drugom bojom.

```
RELEJ `EQU 0FFFF0000 ; Definiranje naziva
TIPKA `EQU 0FFFF0010 ; vanjskih jedinica

; Glavni program

PETLJA LOAD R0, (TIPKA) } učitaj stanje s tipke
 OR R0, R0, R0 } ispitaj je li tipka
 JR_Z ISKLJUCI } pritisnuta ili nije

UKLJUCI MOVE 1, R0 } ako je tipka pritisnuta,
 STORE R0, (RELEJ) } onda uključi relej
 JR PETLJA

ISKLJUCI MOVE 0, R0 } ako tipka nije pritisnuta,
 STORE R0, (RELEJ) } onda isključi relej
 JR PETLJA
```

## Bezuvjetni prijenos - Primjeri

### Rješenje:

```
; Definiranje adrese vanjske jedinice
TERMOMET `EQU 0FFFF0000

; Glavni program

MOVE 10000, R7
; Inicijalizacija varijabli
MOVE 0, R0
STORE R0, (SUMA)
STORE R0, (BROJ_OCITANJA)
STORE R0, (SR_TEMP)
MOVE %D 36, R1 ; brojač za 3 sata
; 36*5 min = 3 sata
 ;>>>
```

## Bezuvjetni prijenos - Primjeri

```
<<<
; Potprogram za izračun sredine:
RACUNAJ_SREDNU ; Parametar R0: nova temperatura
; Rezultat u R0
PUSH R1
LOAD R1, (SUMA)
ADD R0, R1, R1
STORE R1, (SUMA)
LOAD R1, (BROJ_OCITANJA)
ADD R1, 1, R1
STORE R1, (BROJ_OCITANJA)
CALL DIJELI
POP R1
RET
>>>
```

### Primjer:

Na bezuvjetnu izlaznu VJ0 na adresi FFFF0000 spojen je relej. Slanjem broja 0 relej se isključuje, a slanjem 1 se uključuje.

Na bezuvjetnu ulaznu VJ1 na adresi FFFF 0010 spojena je tipka. Kad je tipka pritisnuta, s VJ se očitava stanje 1, a kad tipka nije pritisnuta, očitava se stanje 0.

Program treba ispitivati je li tipka pritisnuta i samo tada držati relej uključen. Građa VJ0 i VJ1 je kao u prethodnom opisu.

## Bezuvjetni prijenos - Primjeri

### Primjer:

Na bezuvjetnu ulaznu vanjsku jedinicu na adresi FFFF0000 spojen je digitalni termometar s kojeg se može očitati trenutna vrijednost temperature. Vanjska jedinica građena je kao što je pokazano na prethodnim slikama.

Program mora svakih 5 minuta očitati temperaturu, izračunati srednju vrijednost svih dotadašnjih temperatura i spremiti je u lokaciju SR\_TEMP. Nakon tri sata treba zaustaviti procesor.

Prepostavka je da se vrijednost temperature vraća u nižih 8 bitova, a gornjih 24 bita su u ništicama.

## Bezuvjetni prijenos - Primjeri

&lt;&lt;&lt;

```
PETLJA LOAD R0, (TERMOMET) ; Učitaj temperaturu
 CALL RACUNAJ_SREDNU ; Izračunaj i spremi
 STORE R0, (SR_TEMP) ; srednju temperaturu.

 CALL KASNI_5_MINUTA

 SUB R1, 1, R1 ; Istečela 3 sata?
 JR_NZ PETLJA

 HALT
>>>
```

## Bezuvjetni prijenos - Primjeri

&lt;&lt;&lt;

```
; Potprogram za dijeljenje:
DIJELI ; Parametri su u fiksnim memorijskim
; lokacijama SUMA i BROJ_OCITANJA.
; Rezultat se vraća registrom R0.
```

...

RET

; Varijable

|               |    |   |
|---------------|----|---|
| SUMA          | DW | 0 |
| BROJ_OCITANJA | DW | 0 |
| SR_TEMP       | DW | 0 |

&gt;&gt;&gt;

```
<<< ; Potprogram za cca 5-minutno kašnjenje
; Nema parametara ni rezultata
KASNI_5_MINUTA
 PUSH R0
 LOAD R0, (KONST) ; 109 prolazaka
LOOP SUB R0, 1, R0 ; 1 takt
 JR_NZ LOOP ; 2 taka } 3 taka
 POP R0
 RET
KONST DW %D 1000000000
```

Ovaj potprogram napisan je za CLOCK frekvencije 10 MHz:

$$5 \text{ min} = 300 \text{ sek} = 300 * 10^6 \text{ taktova} = 3 * 10^9 \text{ taktova}$$



## Rješenje:

```
; Definiranje adresa vanjskih jedinica

LED `EQU OFFFF0000
TEMP1 `EQU OFFFF0100
TEMP2 `EQU OFFFF0200
TEMP3 `EQU OFFFF0300

;; Glavni program

MOVE 10000, R7

;; Na početku ugasi sve LED-ice
MOVE 0, R0
STORE R0, (LED)
```



```
<<< ; slično kao za prvu temperaturu

DRUGA CALL SEKUNDA ; čekaj sekundu

LOAD R0, (TEMP2) ; Učitaj i ispitaj
CMP R0, 30 ; 2. temperaturu
JR_ULE MANJA_2

VECA_2 LOAD R0, (LED) ; upali 2. LED-icu, a
OR R0, 2, R0 ; ostale ne diraj
STORE R0, (LED)
JR TRECA

MANJA_2 LOAD R0, (LED) ; ugasi 2. LED-icu, a
AND R0, -3, R0 ; ostale ne diraj
STORE R0, (LED)
>>>
```



## Primjer:

Na tri bezuvjetne ulazne vanjske jedinice na adresama FFFF0100, FFFF0200 i FFFF0300 spojeni su digitalni termometri s kojih se može očitati trenutačna vrijednost temperature (u opsegu od 5 do 100°C).

Na bezuvjetnu izlaznu vanjsku jedinicu na adresi FFFF0000 spojene su 3 LED diode (na tri najniža bita). LED diode pale se slanjem 1 na odgovarajući bit, a gase se slanjem 0. Svaka LED dioda služi kao indikacija da li je jedna od temperatura premašila 30°C.

Program mora redom očitavati temperature: svake sekunde očita se jedna temperatura. Ako je temperatura veća od 30, treba uključiti pripadnu LED-icu, a inače je treba isključiti.

Potprogram za kašnjenje od jedne sekunde napišite sami.



<<<

```
PRVA CALL SEKUNDA ; čekaj sekundu

LOAD R0, (TEMP1) ; Učitaj i ispitaj
CMP R0, 30 ; 1. temperaturu
JR_ULE MANJA_1

VECA_1 LOAD R0, (LED) ; upali 1. LED-icu, a
OR R0, 1, R0 ; ostale ne diraj
STORE R0, (LED)
JR DRUGA

MANJA_1 LOAD R0, (LED) ; ugasi 1. LED-icu, a
AND R0, -2, R0 ; ostale ne diraj
STORE R0, (LED)
```

>>>



<<< ; slično kao za prvu temperaturu

```
TRECA CALL SEKUNDA ; čekaj sekundu

LOAD R0, (TEMP3) ; Učitaj i ispitaj
CMP R0, 30 ; 3. temperaturu
JR_ULE MANJA_3

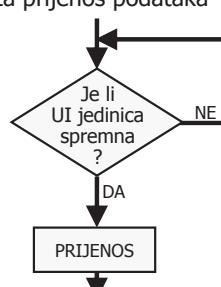
VECA_3 LOAD R0, (LED) ; upali 2. LED-icu, a
OR R0, 4, R0 ; ostale ne diraj
STORE R0, (LED)
JR PRVA ; povratak na početak

MANJA_3 LOAD R0, (LED) ; ugasi 2. LED-icu, a
AND R0, -5, R0 ; ostale ne diraj
STORE R0, (LED)
JR PRVA ; povratak na početak
```

## Uvjetni prijenos



- **Uvjetni prijenos** rješava probleme gubitka i uvišestručenja podataka kod bezuvjetnog prijenosa
- Glavna značajka je da se prije prijenosa **uvijek provjerava** je li VJ spremna za prijenos podataka



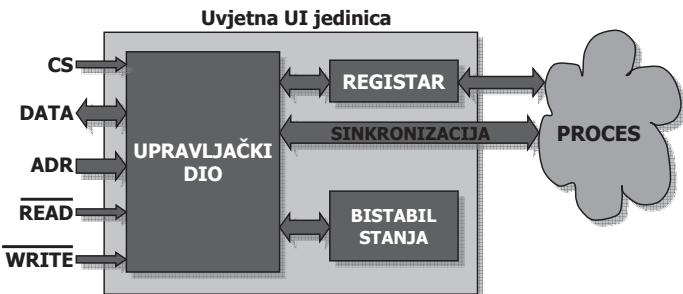


- Glavni **nedostatak** uvjetnog prijenosa:
  - Budući da je procesor tipično puno brži od vanjskih uređaja, može se dogoditi da procesor puno vremena gubi na čekanje da VJ postane spremna
- Uvjetna VJ je sklopovski složenija od bezuvjetne VJ:
  - VJ mora imati stanje spremnosti koje se pamti u **bistabilu stanju** (u tzv. status-bistabilu)
  - VJ mora imati dodatne **sinkronizacijske priključke** za povezivanje s vanjskim uređajem

## Osnovna građa uvjetne UI jedinice



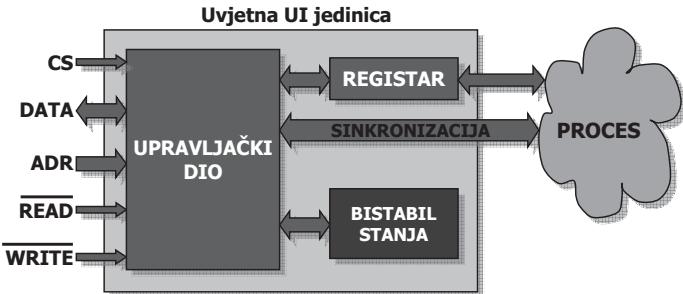
- Najjednostavnija građa opće uvjetne UI jedinice može se prikazati sljedećom blok shemom:



## Osnovna građa uvjetne UI jedinice



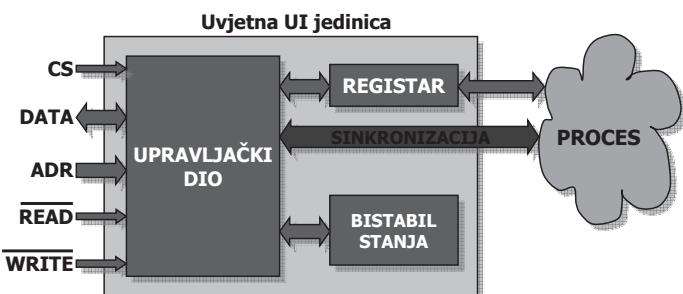
- Dodatno postoji **bistabil stanja** koji pamti spremnost VJ:
  - bistabil je u stanju 1 kad je VJ spremna
  - bistabil je u stanju 0 kad VJ nije spremna
  - "spremnost" znači spremnost VJ za komunikaciju s procesorom



## Osnovna građa uvjetne UI jedinice



- Sinkronizacijski priključci služe za sinkronizaciju rada VJ i vanjskog procesa (uređaja) i slični su rukovanju u asinkronim sabirničkim protokolima. Koriste se prilikom:
  - prenošenja podataka iz VJ u proces
  - prenošenja podataka iz procesa u VJ

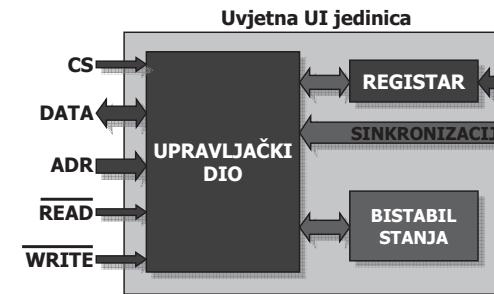


- Kad možemo koristiti uvjetni prijenos?
- Općenito:
  - kad nam je bitno da nema gubitaka/uvijestručenja podataka (tj. bitna nam je sinkronizacija)
  - kad ne znamo kojom brzinom ćemo pristupati VJ pa se može dogoditi da ona još nije spremna
- Na primjer: želimo slati znakove na pisač (naravno, pri tome je bitno da svi znakovi budu primljeni i ispisani)
- Na primjer: želimo čitati niz podataka koji od nekuda (npr. s mreže ili iz tipkovnice) pristižu na VJ i koje sve želimo bez gubitaka pohraniti u memoriju

## Osnovna građa uvjetne UI jedinice



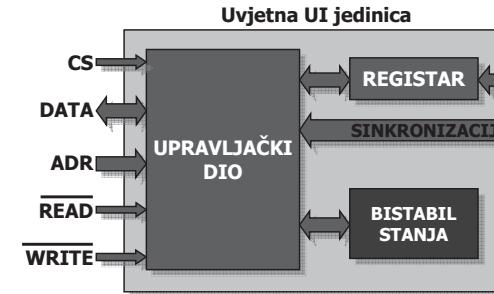
- Svi djelovi koji postoje i u bezuvjetnoj VJ imaju istu namjenu kao što je već opisano



## Osnovna građa uvjetne UI jedinice



- Dakle, bistabil stanja služi za sinkronizaciju rada VJ i procesora
- Bistabil stanja se postavlja u ovisnosti o brzini vanjskog uređaja/procesa koji je spojen na VJ



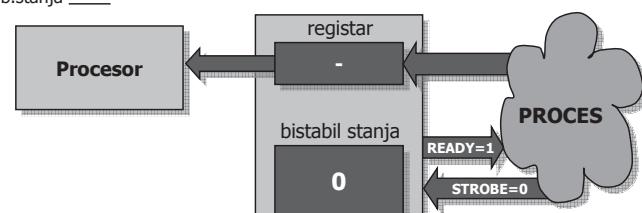
## Redoslijed uvjetne komunikacije - ULAZ



- 0) Na početku je register podatka "prazan/slobodan" pa VJ nije spremna za komunikaciju s procesorom (bist. stanja=0). S druge strane, VJ je spremna za komunikaciju s vanjskim procesom, tj. od vanjskog procesa može primiti podatak u register (READY=1).

DATA  
READY  
STROBE  
b.stanja

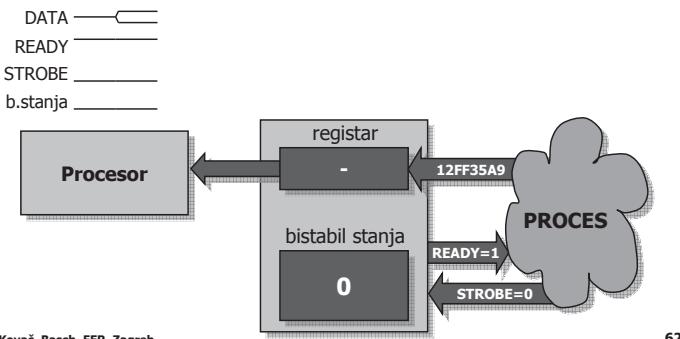
} Sinkronizacijski priključci (priključci za rukovanje)



## Redoslijed uvjetne komunikacije - ULAZ



- 1) Budući da je vanjski proces prethodno detektirao rastući brid na READY, zna da može slati novi podatak u VJ. Zato vanjski proces postavlja podatak na podatkovne veze prema VJ

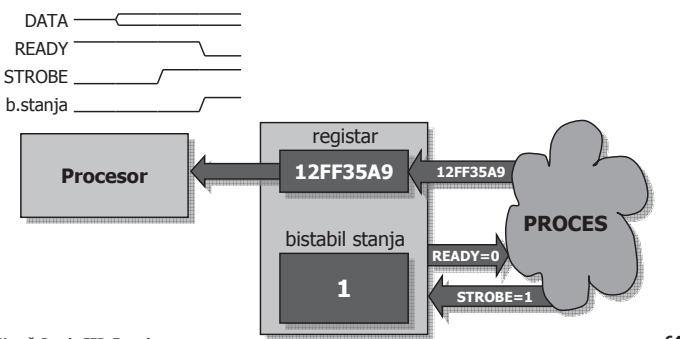


62

## Redoslijed uvjetne komunikacije - ULAZ



- 3) Nakon primanja podatka iz vanjskog procesa, VJ automatski postaje spremna za komunikaciju s procesorom (b.stanja=1), tj. za slanje podatka procesoru. Istodobno, VJ postaje nespremna za komunikaciju s vanjskim procesom pa deaktivira READY=0.

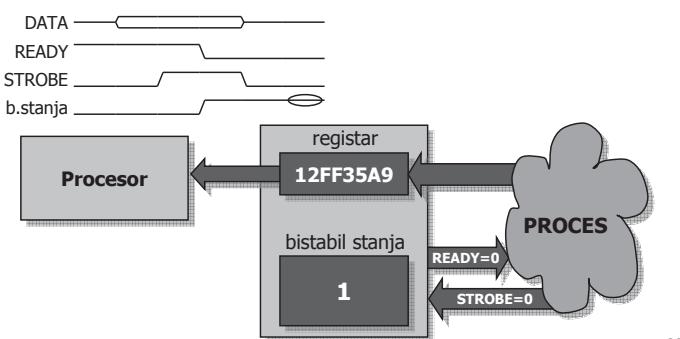


64

## Redoslijed uvjetne komunikacije - ULAZ



- 5) Za to vrijeme, nakon ispitivanja stanja spremnosti, procesor prepozna da je VJ spremna te nakon toga smije čitati podatak iz VJ

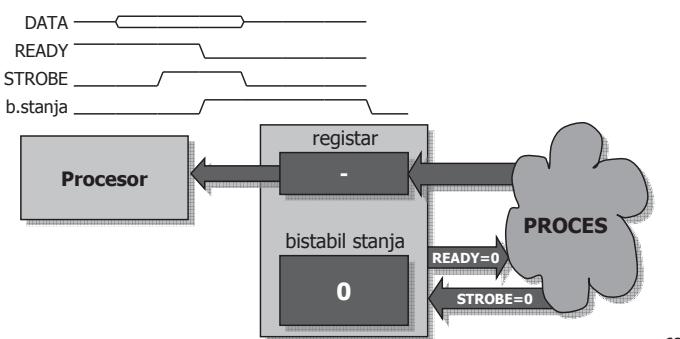


66

## Redoslijed uvjetne komunikacije - ULAZ



- 7a) Zbog "praznog" registra, VJ postaje nespremna za komunikaciju s procesorom, jer nema za njega novi podatak:
- B.stanja se automatski briše nakon čitanja podatka iz registra podatka
  - B.stanja treba brisati programski nakon čitanja podatka iz registra podatka

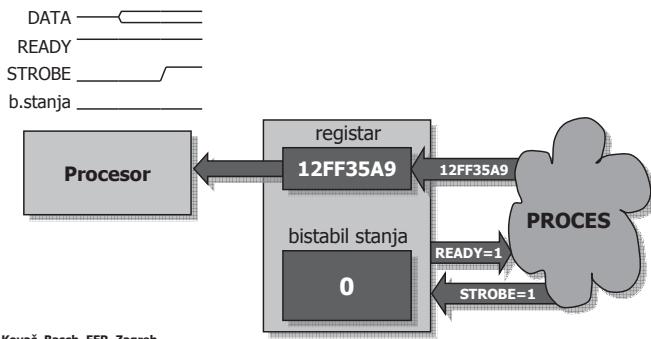


68

## Redoslijed uvjetne komunikacije - ULAZ



- 2) Zatim proces aktiviranjem STROBE=1 javlja VJ da je poslao podatak. VJ na rastući brid od STROBE zapamti podatak u registar (registar se time "zauzme/napuni")

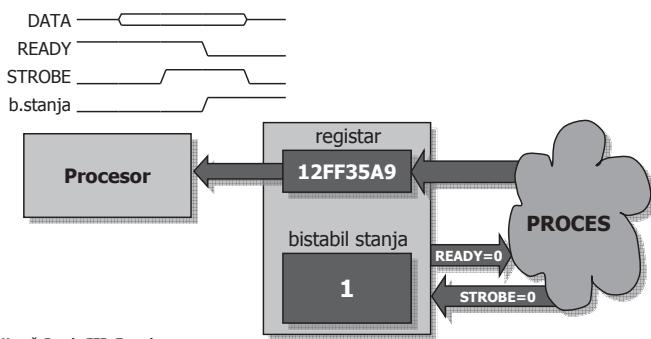


63

## Redoslijed uvjetne komunikacije - ULAZ



- 4) Vanjski proces vidi da je READY deaktiviran pa zna da je VJ preuzeala podatak. Zato vanjski proces uklanja podatak sa sabirnice prema VJ i istodobno deaktivira STROBE=0.

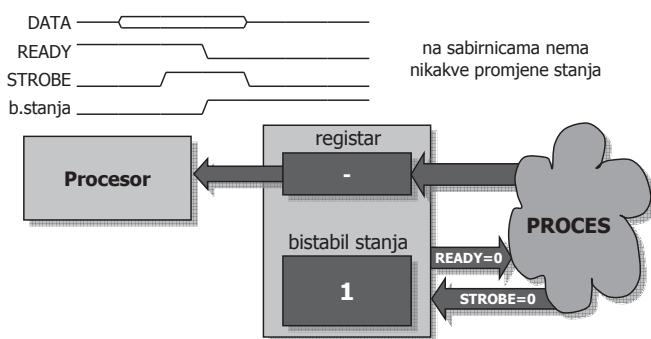


65

## Redoslijed uvjetne komunikacije - ULAZ



- 6) Procesor čita iz VJ podatak iz registra podatka (čime registar postaje spreman za prihvatanje novog podatka iz procesa)

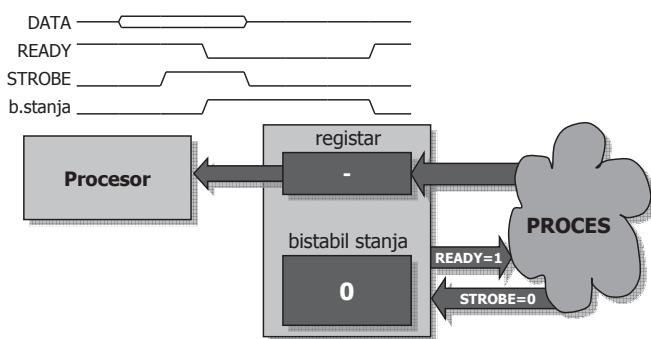


67

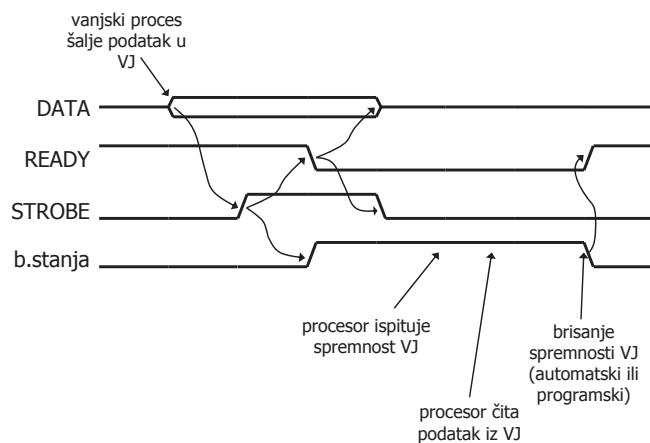
## Redoslijed uvjetne komunikacije - ULAZ



- 7b) Istodobno s brisanjem b.stanja, VJ postavlja READY=1 jer može od vanjskog procesa primiti novi podatak. Ovime dolazimo u početno stanje 0.



69

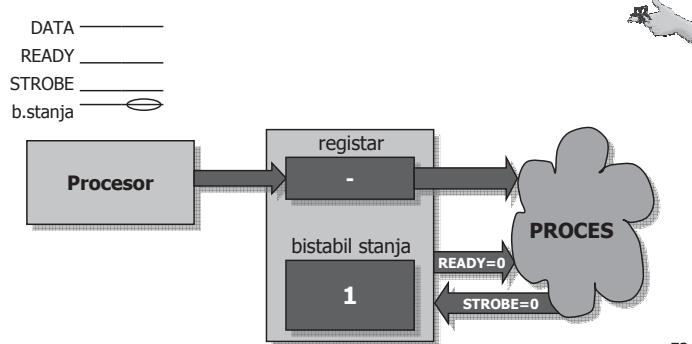


© Kovač, Basch, FER, Zagreb

70



- 1) Procesor ispituje stanje spremnosti i prepoznaže da je VJ spremna te nakon toga smije slati podatak u VJ

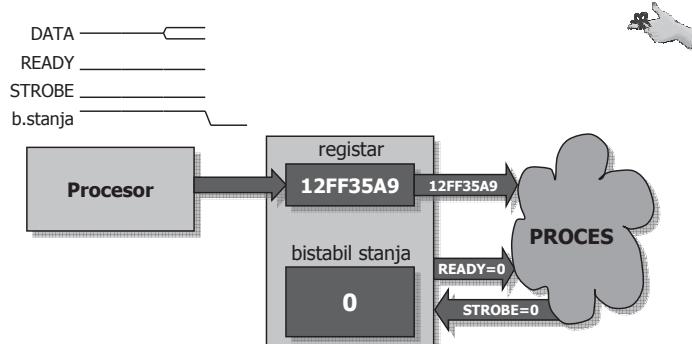


© Kovač, Basch, FER, Zagreb

72



- 3a) Zbog "punog" registra, VJ postaje nespremna za komunikaciju s procesorom, jer ne može od njega primiti novi podatak:
- B.stanja se automatski briše nakon pisanja podatka u register podatka
  - B.stanja treba brisati programski nakon pisanja podatka u register

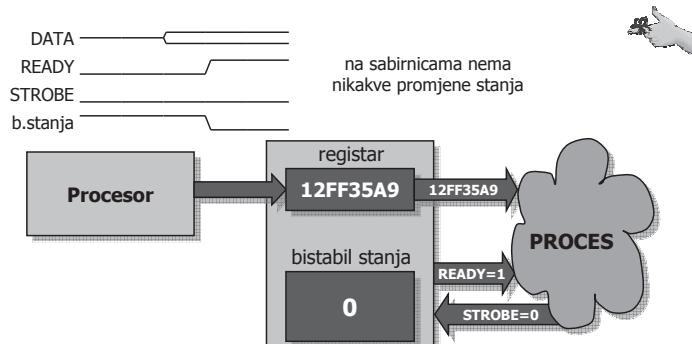


© Kovač, Basch, FER, Zagreb

74



- 4) Nakon što je detektirao rastući brid na READY, vanjski proces čita podatak

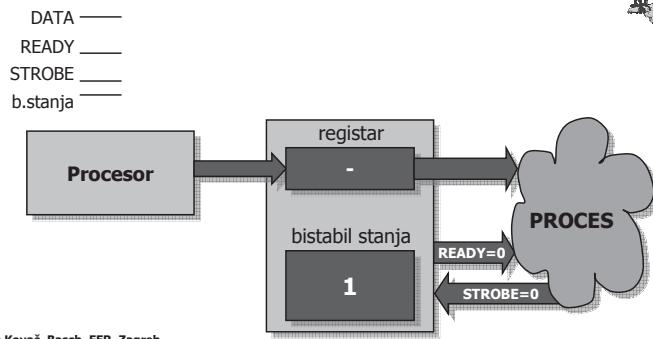


© Kovač, Basch, FER, Zagreb

76



- 0) Na početku je registar podatka "prazan/slobodan" pa je VJ spremna za primanje podatka od procesora (b.stanja=1). S druge strane, VJ nije spremna za komunikaciju s vanjskim procesom (READY=0).

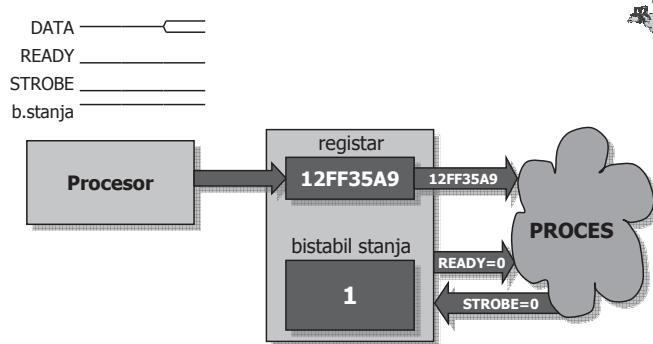


© Kovač, Basch, FER, Zagreb

71



- 2) Procesor šalje podatak u VJ čime se "zauzima/puni" register podatka

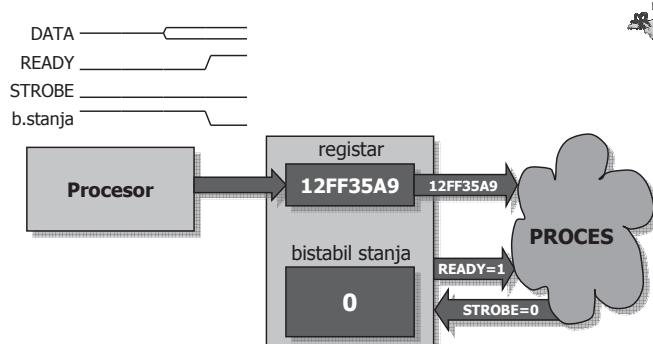


© Kovač, Basch, FER, Zagreb

73



- 3b) Istodobno s brisanjem b.stanja, VJ postavlja READY=1 čime javlja vanjskom procesu da za njega ima novi podatak.

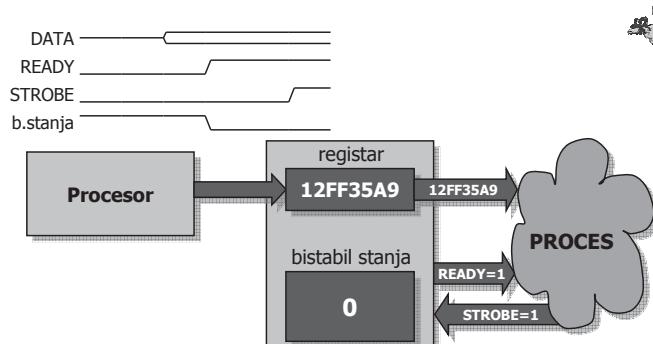


© Kovač, Basch, FER, Zagreb

75



- 5) Nakon što je pročitao podatak, vanjski proces dojavljuje to VJ aktiviranjem STROBE=1

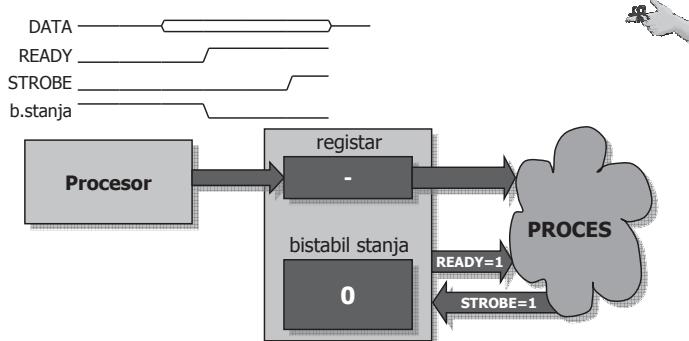


© Kovač, Basch, FER, Zagreb

77



- 6a) VJ nakon što je detektirala rastući brid na STROBE, zna da je vanjski proces preuzeo podatak i "ispraznio" registar, tj. registar opet postaje "prazan/slobodan". Zato VJ oslobađa DATA

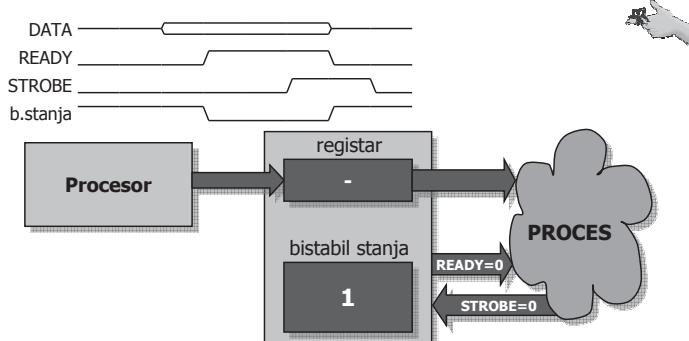


© Kovač, Basch, FER, Zagreb

78



- 7) Vanjski proces nakon detektiranja padajućeg brida na READY deaktivira STROBE=0. Ovime dolazimo u početno stanje.



© Kovač, Basch, FER, Zagreb

80

## Uvjetni prijenos - rekapitulacija



- Iz prethodnih opisa vidi se da je uvijek vrijedi:
  - Dok je VJ spremna za komunikaciju s procesorom, nije spremna za komunikaciju s vanjskim procesom (i obrnuto)
- Bistabil stanja služi za sinkronizaciju između VJ i procesora:
  - Procesor programski ispituje b.stanja da bi utvrdio spremnost VJ
  - VJ postavlja bistabil stanja kad postane spremna
  - Bistabil stanja briše se (programski ili automatski) nakon obavljenog prijenosa
- Brisanjem bistabila stanja, omogućuje se nastavak komunikacije s vanjskim procesom**
- Sinkronizacijski priključci služe za sinkronizaciju između VJ i vanjskog procesa:
  - Vanjski proces sklopovski ispituje sinkronizacijske priključke da bi utvrdio spremnost VJ

© Kovač, Basch, FER, Zagreb

82

## Osnovna građa uvjetne UI jedinice



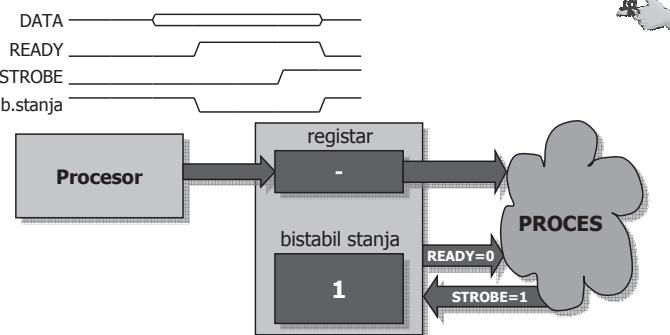
- Svaka lokacija zauzima 4 adrese, a cijela VJ zauzima ukupno 8 adresa).
- Početna adresa vanjske jedinice određena je vanjskim adresnim dekoderom čiji izlaz je spojen na CS (početna adresa mora biti djeljiva s 8)
- Prva i druga lokacija razlikuju se po adresnom bitu ADR2 pa njega dodatno dovodimo u vanjsku jedinicu ( $ADR0=ADR1=0$ )
- Na primjer, ako je početna adresa FFFF0000, onda vrijedi:
  - adresa prve lokacije je FFFF0000 (najnižih 8 bita adrese su 0000 0000<sub>(2)</sub>)
  - adresa druge lokacije je FFFF0004 (najnižih 8 bita adrese su 0000 0100<sub>(2)</sub>)

© Kovač, Basch, FER, Zagreb

84

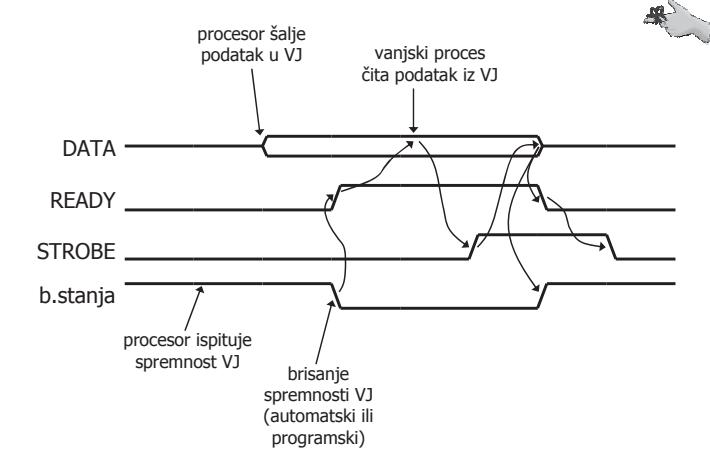


- 6b) Nakon "pražnjenja" registra, VJ ponovo postaje spremna za komunikaciju s procesorom, tj. za primitak novog podatka od njega pa postavlja b.stanja=1. Istodobno, VJ nema više podataka za vanjski proces pa mu to javlja deaktiviranjem READY=0



© Kovač, Basch, FER, Zagreb

79



© Kovač, Basch, FER, Zagreb

81

## Osnovna građa uvjetne UI jedinice



- Ovakva vanjska jedinica **zauzimat će dvije uzastopne 32-bitne lokacije**
  - Na prvoj lokaciji se čita ili piše podatak
  - Na drugoj lokaciji se pristupa bistabilu stanja
- Na lokaciji za bistabil stanja može se:
  - Procitati trenutačni sadržaj bistabila (ispitivanje stanja)
  - Obrisati bistabil (operacijom upisa bilo kojeg podatka - poslani podatak se zanemaruje) \*

\* Za naše uvjetne UI jedinice ćemo pretpostavljati da se bistabil stanja uvijek mora brisati programski (tj. pretpostavljamo da se to ne radi automatski)

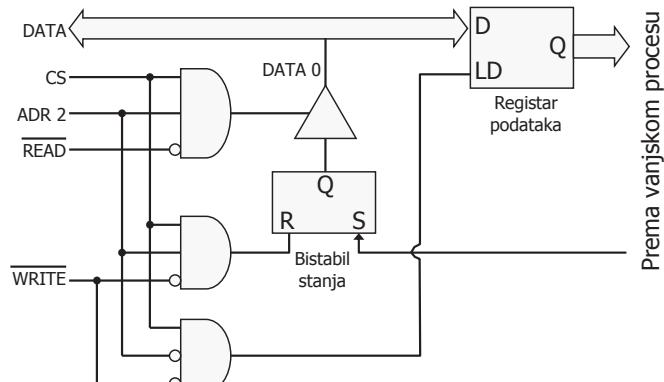
© Kovač, Basch, FER, Zagreb

83

## Osnovna građa uvjetne UI jedinice



- Upravljački dio (pojednostavljeni prikaz):



© Kovač, Basch, FER, Zagreb

85

## Uvjetni prijenos - Primjeri

### Primjer:

Napišite program koji će na uvjetnu vanjsku jedinicu (građenu kao što je upravo opisano) poslati 200 32-bitnih podataka koji se nalaze u memoriji od lokacije PODATCI. Nakon što su svi podatci poslani, treba zaustaviti procesor.

### Rješenje:

Budući da adresa vanjske jedinice nije zadana u zadatku, sami odabiremo proizvoljnu adresu 0FFFF0000.

&gt;&gt;&gt;

## Uvjetni prijenos - Primjeri

&lt;&lt;&lt;

```
CEKAJ LOAD R2, (ISPITAJ); čitaj stanje iz VJ
 OR R2, R2, R2 ; postavi zastavice
 JR_Z CEKAJ ; ispitaj stanje VJ

SPREMNA LOAD R2, (R0) ; čitaj podatak iz mem.
 STORE R2, (SALJI); šalji podatak u VJ
 STORE R2, (BRISI); briši stanje VJ

 ADD R0, 4, R0
 SUB R1, 1, R1
 JR_NZ CEKAJ

HALT
```

```
PODATCI DW 12, 5452, 331A3, ...
```

## Uvjetni prijenos - Primjeri

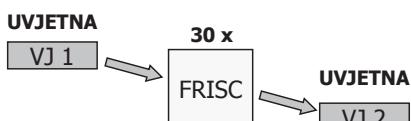
### Primjer:

Na FRISC su spojene dvije uvjetne vanjske jedinice: ulazna vj1 na adresi FFFF1000 te izlazna vj2 adresi FFFF2000.

FRISC treba prenijeti 30 podataka sa VJ1 na VJ2 nakon čega nastavlja s izvođenjem glavnog programa.

### Rješenje:

Nacrtajmo shemu slanja podataka kako bi lakše riješili zadatak:



## Uvjetni prijenos - Primjeri

&lt;&lt;&lt;

```
CEKAJ_1 LOAD R0, (ISPITAJ_1)
 OR R0, R0, R0 } Čekaj da VJ1
 JR_Z CEKAJ_1 } postane spremna
 LOAD R1, (PRIMI_1)
 STORE R0, (BRISI_1) } Posluži VJ1

CEKAJ_2 LOAD R0, (ISPITAJ_2)
 OR R0, R0, R0 } Čekaj da VJ2
 JR_Z CEKAJ_2 } postane spremna
 STORE R1, (SALJI_2)
 STORE R0, (BRISI_2) } Posluži VJ2

 SUB R3, 1, R3
 JR_NZ CEKAJ_1 } Ispitaj je li preneseno
 svih 30 podataka

NASTAVAK ... ; nastavak glavnog programa
```

## Uvjetni prijenos - Primjeri

; Definiranje adresa vanjske jedinice

```
SALJI `EQU 0FFFF0000
ISPITAJ `EQU 0FFFF0004
BRISI `EQU 0FFFF0004
```

; Glavni program

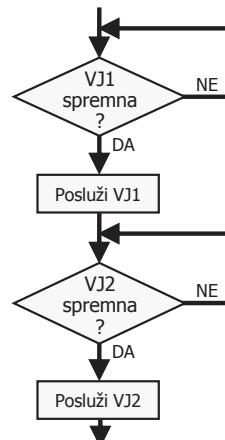
```
; Inicijalizacija varijabli
MOVE PODATCI, R0 ; adresa podataka
MOVE %D 200, R1 ; brojač petlje
```

&gt;&gt;&gt;

## Posluživanje više uvjetnih vanjskih jedinica



- Ako su **jedinice zavisne**, onda moramo za svaku **čekati** da postane spremna
- Zavisne jedinice znače da je redoslijed njihovog posluživanja bitan
- Na slici je primjer gdje se podatak primljen od VJ1 prenosi na VJ2



## Uvjetni prijenos - Primjeri

&lt;&lt;&lt;

; Definiranje adresa vanjskih jedinica

```
PRIMI_1 `EQU 0FFFF1000
ISPITAJ_1 `EQU 0FFFF1004
BRISI_1 `EQU 0FFFF1004

SALJI_2 `EQU 0FFFF2000
ISPITAJ_2 `EQU 0FFFF2004
BRISI_2 `EQU 0FFFF2004
```

; Glavni program

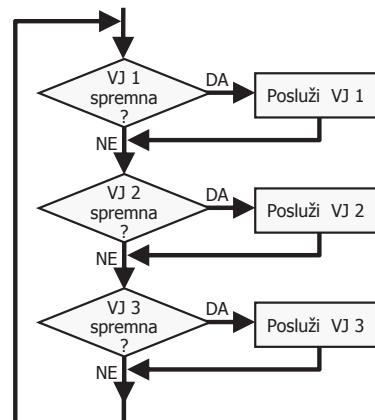
```
MOVE 30, R3 ; brojač podataka
```

&gt;&gt;&gt;

## Posluživanje više uvjetnih vanjskih jedinica



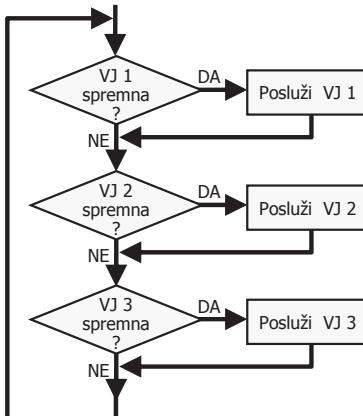
- Ako imamo **više nezavisnih uvjetnih vanjskih jedinica**, onda možemo primijeniti postupak **prozivanja** (eng. polling)
- Nezavisne jedinice znače da jedna ne ovisi o drugoj, tj. da redoslijed posluživanja nije bitan
- Na slici je primjer za tri nezavisne uvjetne vanjske jedinice





- Prozivanje umanjuje nedostatak uvjetnog prijenosa:** čekanje na spremnost pojedine vanjske jedinice
  - Veća je vjerojatnost da će jedna od nekoliko jedinica postati spremna

- Nakon posluživanja jedne jedinice prelazi se na sljedeću jedinicu
  - Ne bi bilo dobro vratiti se na ispitivanje prve zbog mogućeg "izgladnjivanja" zadnjih jedinica u lancu ispitivanja



## Uvjetni prijenos - Primjeri

<<<

```

;; Definiranje adresa vanjskih jedinica
SALJI_1 `EQU OFFFFF1000
ISPITAJ_1 `EQU OFFFFF1004
BRISI_1 `EQU OFFFFF1004
SALJI_2 `EQU OFFFFF2000
ISPITAJ_2 `EQU OFFFFF2004
BRISI_2 `EQU OFFFFF2004

;; Glavni program
MOVE 10000, R7
MOVE 2000, R1 ; brojač podataka
MOVE PODATCI, R2 ; adresa podataka

```

>>>

## Uvjetni prijenos - Primjeri

```

<<< ; potprogram za posluživanje vj1
SALJI_VJ1 LOAD R0, (R2) ; podatak iz memorije
 STORE R0, (SALJI_1) ; šalji podatak
 STORE R0, (BRISI_1) ; briši spremnost
 ADD R2, 4, R2 ; pomakni pokazivač
 SUB R1, 1, R1 ; smanji brojač
 RET

 ; potprogram za posluživanje vj2
SALJI_VJ2 LOAD R0, (R2)
 STORE R0, (SALJI_2)
 STORE R0, (BRISI_2)
 ADD R2, 4, R2
 SUB R1, 1, R1
 RET

```

} analogno gornjem potprogramu

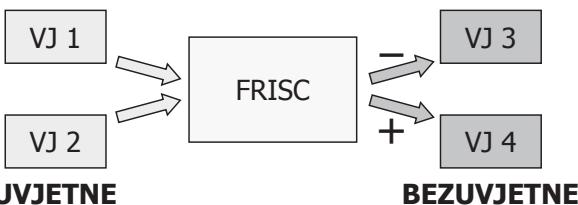
## Uvjetni prijenos - Primjeri

### Primjer:

Na FRISC su spojene: dvije ulazne uvjetne vanjske jedinice vj1 i vj2 na adresama FFFF1000 i FFFF2000 te dvije izlazne bezuvjetne vanjske jedinice vj3 i vj4 na adresama FFFF3000 i FFFF4000. FRISC s ulaznih jedinica čita podatke i šalje negativne na vj3, a pozitivne na vj4.

### Rješenje:

Prvo nacrtajmo shemu prijenosa podataka kako bi lakše riješili zadatka:



### Primjer:

Na FRISC su spojene dvije uvjetne vanjske jedinice: vj1 na adresi FFFF1000 i vj2 adresi FFFF2000.

Obje jedinice rade neovisno i FRISC im šalje 2000 podataka smještenih od adrese PODATCI. Kako koja jedinica postane spremna tako joj se pošalje podatak (to znači da brža jedinica prima više podataka).

Kada je završen prijenos svih 2000 podataka, treba zaustaviti procesor.

### Rješenje:

Nacrtajmo shemu slanja podataka kako bi lakše riješili zadatku:



## Uvjetni prijenos - Primjeri

<<<

### ;;;;;; POSTUPAK PROZIVANJA

|          |         |                 |                                              |
|----------|---------|-----------------|----------------------------------------------|
| PROZIVAJ | LOAD    | R0, (ISPITAJ_1) | } Ako je VJ1 spremna, onda je posluži        |
|          | OR      | R0, R0, R0      |                                              |
|          | CALL_NZ | SALJI_VJ1       |                                              |
|          | OR      | R1, R1, R1      | } Ispitaj je li preneseno svih 2000 podataka |
|          | JR_Z    | KRAJ            |                                              |
|          | LOAD    | R0, (ISPITAJ_2) | } Ako je VJ2 spremna, onda je posluži        |
|          | OR      | R0, R0, R0      |                                              |
|          | CALL_NZ | SALJI_VJ2       |                                              |
|          | OR      | R1, R1, R1      | } Ispitaj je li preneseno svih 2000 podataka |
|          | JR_NZ   | PROZIVAJ        |                                              |

KRAJ HALT

>>>

## Posluživanje više VJ - Komentar

• Načelno, kod posluživanja više vanjskih jedinica treba odrediti redoslijede čitanja/pisanja u ovisnosti o samim jedinicama:

- Uvjetnim VJ treba uvijek ispitati spremnost i poslužiti ih čim se ustanovi da su spremne
- Ako su uvjetne VJ nezavisne, onda ih se proziva
- Ako su uvjetne VJ zavisne, onda se mora čekati da postanu spremne
- Bezuvjetne VJ treba čitati/pisati tek kad zatreba.
  - Nema smisla npr. pročitati podatak s bezuvjetne, a onda čekati da uvjetna postane spremna kako bi joj poslati taj podatak.
  - Treba pokušati raditi s "najsvježijim" podatcima: npr. čekati da uvjetna postane spremna i tek onda pročitati podatak s bezuvjetne i odmah ga poslati uvjetnoj

## Uvjetni prijenos - Primjeri

### ;; Definiranje adresa vanjskih jedinica

```

CITAJ_1 `EQU OFFFF1000
ISPITAJ_1 `EQU OFFFF1004
BRISI_1 `EQU OFFFF1004
CITAJ_2 `EQU OFFFF2000
ISPITAJ_2 `EQU OFFFF2004
BRISI_2 `EQU OFFFF2004
SALJI_3 `EQU OFFFF3000
SALJI_4 `EQU OFFFF4000

```

>>>

## Uvjetni prijenos - Primjeri

<<<

```
;;;;;; Glavni program
GLAVNI MOVE 10000, R7
 ; Prozivanje uvjetnih vj1 i vj2
PROZIVAJ LOAD R0, (ISPITAJ_1) ; ispitaj vj1
 OR R0, R0, R0
 CALL_NZ POSLUZI_1

 LOAD R0, (ISPITAJ_2) ; ispitaj vj2
 OR R0, R0, R0
 CALL_NZ POSLUZI_2

 JR PROZIVAJ
 >>>
```

© Kovač, Basch, FER, Zagreb

102

## Uvjetni prijenos - Primjeri

<<<

```
; Potprogram za slanje na vj3 ili vj4 (ovisno
; o predznaku podatka).
; Podatak koji treba poslati je u R0.

SALJI OR R0, R0, R0 ; postavi predznak
 JR_P POZIT ; ispitaj predznak

NEGAT STORE R0, (SALJI_3) ; negativni na vj3
 RET

POZIT STORE R0, (SALJI_4) ; pozitivni na vj4
 RET
```

© Kovač, Basch, FER, Zagreb

104

## Uvjetni prijenos - Primjeri



### Rješenje:

Budući da je važan redoslijed posluživanja, postoji zavisnost između vj1 i vj2. **Zato ne možemo koristiti prozivanje.**

```
; Definiranje adresa vanjskih jedinica
PRIMI_1 `EQU 0FFFF1000
ISPITAJ_1 `EQU 0FFFF1004
BRISI_1 `EQU 0FFFF1004
SALJI_2 `EQU 0FFFF2000
ISPITAJ_2 `EQU 0FFFF2004
BRISI_2 `EQU 0FFFF2004

; Glavni program
MOVE 300, R3 ; brojač podataka
 >>>
```

© Kovač, Basch, FER, Zagreb

106

## Prekidni prijenos

## Uvjetni prijenos - Primjeri

<<<

```
POSLUZI_1 ; Potprogram za posluživanje vj1
 LOAD R0, (CITAJ_1) ; čitaj s uvjetne vj1
 STORE R0, (BRISI_1)

 CALL SALJI ; šalji podatak na vj3/vj4
 RET

POSLUZI_2 ; Potprogram za posluživanje vj2
 LOAD R0, (CITAJ_2) ; čitaj s uvjetne vj2
 STORE R0, (BRISI_2)

 CALL SALJI ; šalji podatak na vj3/vj4
 RET
```

>>>

© Kovač, Basch, FER, Zagreb

103

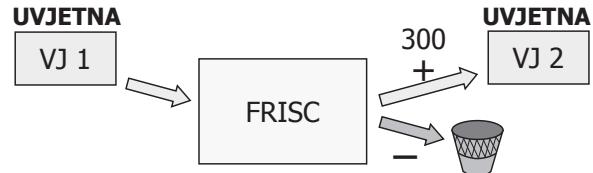
## Uvjetni prijenos - Primjeri



### Primjer:

Na FRISC su spojene dvije vanjske jedinice: ulazna vj1 na adresi FFFF1000 i izlazna vj2 na adresi FFFF2000. Jedinice su iste kao u prethodnim zadatcima, s razlikom da je vj1 ulazna.

FRISC treba prenijeti 300 podataka sa vj1 na vj2, ali samo ako su podatci pozitivni. Negativni podatci se zanemaruju i ne šalju se. Nakon slanja treba zaustaviti procesor.



© Kovač, Basch, FER, Zagreb

105

## Uvjetni prijenos - Primjeri



|          |                      |                              |
|----------|----------------------|------------------------------|
| PETLJA1  | LOAD R0, (ISPITAJ_1) | } čekaj spretnost od vj1     |
|          | OR R0, R0, R0        |                              |
|          | JR_Z PETLJA1         |                              |
| SPREMNA1 | LOAD R1, (PRIMI_1)   | } primi podatak od vj1       |
|          | STORE R0, (BRISI_1)  |                              |
|          | OR R1, R1, R1        |                              |
| PETLJA2  | JR_N PETLJA1         | } zanemari negativne podatke |
| SPREMNA2 | LOAD R0, (ISPITAJ_2) |                              |
|          | OR R0, R0, R0        |                              |
| KRAJ     | JR_Z PETLJA2         | } čekaj spretnost od vj2     |
| HALT     | STORE R1, (SALJI_2)  |                              |
|          | STORE R0, (BRISI_2)  |                              |
|          | SUB R3, 1, R3        | } šalji podatak u vj2        |
|          | JR_NZ PETLJA1        |                              |
|          | HALT                 |                              |

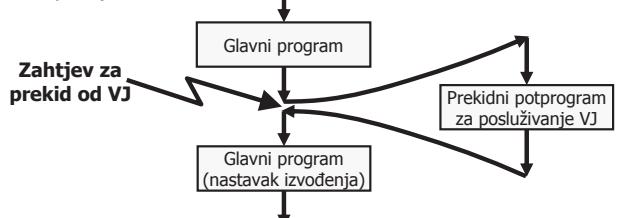
© Kovač, Basch, FER, Zagreb

107

## Prekidni prijenos



- Glavna značajka prekidnog prijenosa je da **UI jedinica samostalno dojavljuje** svoju spremnost procesoru koji za to vrijeme normalno izvodi neki program
  - Spremnost se dojavljuje zahtjevom za prekid (engl. interrupt request)



- Iz dijagrama toka vidi se da zahtjev za prekidom (ili kraće prekid) može doći u bilo kojem trenutku izvođenja glavnog programa





## • Ispitivanje prekida kod FRISC-a:

- Postojanje prekida ispituje se na padajući brid CLOCK-a
  - Naredba koja je u razini izvođenja se izvodi do kraja\*
- Ispitivanje i prihvatanje prekida ovisi o stanju zastavica i trenutačnim zahtjevima za prekid:
  - Ako je IIF=0, prekidi se ne prihvataju
  - U suprotnom, ako je INT3 prisutan, on se prihvata, a ako INT3 nije prisutan, onda se ispituju maskirajući prekidi
  - Ako je GIE=0, maskirajući prekidi se ne prihvataju
  - U suprotnom se maskirajući prekid prihvata uz uvjet da je barem jedan zahtjev INTi prisutan i da je odgovarajuća zastavica EINTi=1

\* Zbog protočne strukture situacija je komplikirana, ali to prelazi opseg gradiva ovog predmeta



## • Prihvatanje maskirajućeg prekida kod FRISC-a:

- Briše se GIE (zabranjivanje dalnjih maskirajućih prekida)
- Sprema se PC na stog
- Dohvat **adrese** prekidnog potprograma (tzv. prekidnog vektora) s memorijске lokacije na adresi 8 i skok u prekidni potprogram, tj. (8) → PC
- Komentari:
  - Nema automatske dojave o prihvatanju prekida - to će trebati napraviti programski u prekidnom potprogramu
  - Prekidni vektor omogućuje postavljanje prekidnog potprograma na bilo koju adresu u memoriji, ali zahtjeva jedan ciklus čitanja više za dohvat prekidnog vektora. Prekidni vektor mora biti zapisan na adresi 8



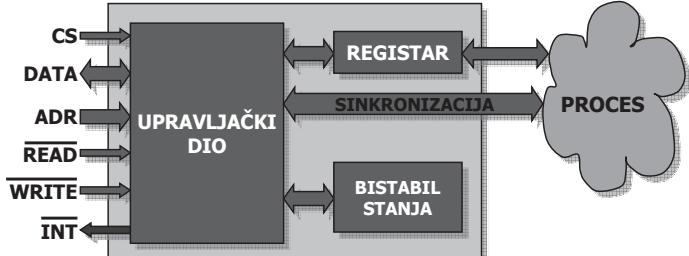
- Naredbe za povratak iz potprograma rade kao i običan RET, ali dodatno dozvoljavaju prekid koji je FRISC bio automatski zabranio kod prihvatanja prekida (drugim riječima, obnavljaju stanje prekidne zastavice GIE odnosno IIF):

- Za **maskirajući prekid** je prije prihvatanja prekida vrijedilo GIE=1, a u trenutku prihvatanja maskirajućeg prekida se GIE automatski briše
  - RET** (RETurn from maskable Interrupt) obnavlja stanje GIE=1
- Za **nemaskirajući prekid** je prije prihvatanja prekida vrijedilo IIF=1, a u trenutku prihvatanja nemaskirajućeg prekida se IIF automatski briše
  - RETN** (RETurn from Nonmaskable interrupt) obnavlja stanje IIF=1



- Glavna razlika u odnosu na uvjetnu jedinicu je postojanje izlaznog priključka INT za postavljanje zahtjeva za prekid

## Prekidna UI jedinica



## • Prihvatanje nemaskirajućeg prekida kod FRISC-a:

- Aktivira se priključak IACK (VJ treba deaktivirati INT3)
- Briše se IIF (zabranjivanje svih dalnjih prekida)
- Sprema se PC na stog
- Deaktivira se IACK
- Skok u prekidni potprogram na adresi  $12_{10}$ , tj.  $12_{10} \rightarrow \text{PC}$
- Komentari:
  - Dojava VJ da je prihvacen zahtjev za prekid obavlja se sklopovski i automatski (pomoći IACK)
  - Prekidni potprogram mora biti uvijek na memorijskoj adresi  $12_{10}$  (tj.  $0C_{16}$ )



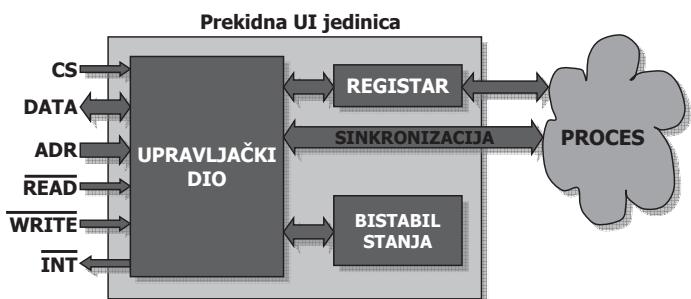
## • Prekidni potprogram kod FRISC-a:

- Sprema se kontekst (registre koje potprogram mijenja)
- Otkriva se uzročnik prekida (ako ih ima više), tj. otkriva se koja VJ je izazvala prekid
- Dojavljuje se VJ da je njen prekid prihvacen
  - programski za maskirajući
  - sklopovski za nemaskirajući - priključkom IACK
- Poslužuje se VJ
- Obnavljanje konteksta
- Dojava VJ da je njezin prekid obrađen
- Povratak iz potprograma s dozvoljavanjem prekida
  - naredbom RETI za maskirajući
  - naredbom RETN za nemaskirajući

\* Koraci 5. i 6. mogu se izvesti i u obratnom redoslijedu



- Najjednostavnija građa opće prekidne UI jedinice može se prikazati sljedećom blok shemom (sve je slično kao kod uvjetne UI jedinice)



- Prekidna VJ može biti spremna ili nespremna kao i uvjetna VJ:
  - Uvjetna VJ je "pasivna": procesor treba ispitivati spremnost što znači da je cijeli tijek prijenosa pod upravljanjem programa
  - Prekidna VJ je "aktivna": kad postane spremna, sama od procesora zahtjeva posluživanje, postavljajući zahtjev za prekid.
- Prekidnoj VJ se programski može zabraniti ili dozvoliti da postavlja prekid kad postane spremna (to je **različito** od dozvoljavanja i zabranjivanja prihvatanja prekida u procesoru)
  - Obično VJ kojoj se zabrani postavljanje prekida i dalje normalno radi te je se može poslužiti kao uvjetnu VJ
- U početnom stanju je dozvoljeno postavljanje prekida
- Zabranjivanje postavljanja prekida ima učinak zaustavljanja VJ u prekidnom načinu rada



- Nakon što bistabil stanja postane 1 (kad VJ postane spremna), automatski se postavlja zahtjev za prekid (uz pretpostavku da je dozvoljeno postavljanje prekida - u suprotnom bistabil stanja ne utječe na stanje prekidnog priključka)
- Brisanje bistabila stanja:
  - uklanja zahtjev za prekid** pa ima ulogu dojave o prihvaćanju zahtjeva za prekid (radi se na početku prekidnog potprograma)
  - ne omogućava nastavak komunikacije s vanjskim procesom** (za razliku od brisanja bistabila stanja kod uvjetne jedinice)
- Nastavak komunikacije s vanjskim procesom moguće je tek nakon što se VJ dojaviti da je njen prekid obrađen
  - to znači da tek nakon toga VJ može ponovno postati spremna i postaviti novi prekid (radi se na kraju prekidnog potprograma)



## • Za VJ koja bi se spajala na nemaskirajući prekid INT3 vrijedi:

- Ima dodatni ulaz IACK čije aktiviranje briše bistabil stanja
  - Ako ne bi imala IACK, dojava prihvaćanja prekida bi se morala obaviti programski.
  - Mi ćemo u zadatcima pretpostavljati da VJ spojene na INT3 imaju IACK (ako nije drugačije navedeno).

```
VJ_DATA `EQU 0FFFF3000
VJ_STAT `EQU 0FFF3004
VJ_IEND `EQU 0FFF3008
VJ_STOP `EQU 0FFF300C

`ORG 0
MOVE 10000, R7 ; početak izvodenja
JP GLAVNI ; preskoči p.p.

`ORG 0C ; adresa p.p. za INT3
PUSH R0 ; spremi kontekst
PUSH R1
MOVE SR, R0
PUSH R0
; IACK se radi automatski >>>
```

```
<<<
VAN STORE R0, (VJ_IEND) ; dojaviti kraj posluživanja
POP R0 ; obnova konteksta
MOVE R0, SR
POP R1
POP R0
RETN ; povratak i IIF=1

PROC_HALT DW 0 ; oznaka za glavni program
; 0 = nastavi rad
; 1 = zaustavi procesor

BROJAC DW 100 ; brojač prenesenih podataka
ADR_PODAT DW 1000 ; adresa za spremanje u blok
>>>
```



## • Prekidna VJ zauzimat će četiri uzastopne 32-bitne lokacije:

- Na **prvoj lokaciji** se čita ili piše podatak
- Na **drugoj lokaciji** se pristupa bistabili stanja:
  - Cita se trenutačni sadržaj bistabila (ispitivanje stanja)
  - Briše se bistabil operacijom upisa bilo kojeg podatka (poslani podatak se zanemaruje)
- Na **trećoj lokaciji** se upisom bilo kojeg podatka (poslani podatak se zanemaruje) dojavljuje da je prekid obrađen
- Pomoću **četvrte lokacije** upravlja se postavljanjem prekida:
  - Upis 0 zabranjuje, a upis 1 dozvoljava postavljanje zahtjeva za prekid
  - Citanje vraća trenutačnu o(ne)mogućnost postavljanja prekida
  - Inicijalno ćemo pretpostaviti da je dozvoljeno postavljanje zahtjeva za prekid

## Primjer:

FRISC treba primiti 100 podataka od prekidne VJ spojene na INT3. Adresa VJ je FFFF3000. Primljene podatke treba spremati u memorijski blok podataka na adresi 1000, samo ako su pozitivni.

Nakon primanja 100 podataka treba zaustaviti rad VJ i rad programa.

## Rješenje:

Pretpostavljamo da VJ ima IACK (jer nije drugačije zadano).

Prekidni potprogram mora početi na adresi 12<sub>10</sub> (0C<sub>16</sub>).

Obavezno inicijalizirati SP - obavezno jer se koriste prekidi

Na početku glavnog programa ne treba omogućavati prekide jer je INT3 inicijalno omogućen (a ne može ga se zabraniti).

```
<<<
LOAD R0, (ADR_PODAT) ; primi podatak i...
LOAD R1, (VJ_DATA) ; ...spremi ga u blok...
OR R1, R1, R1 ; ...ako je pozitivan
JR_M NEMOJ
SPREMI STORE R1, (R0)
ADD R0, 4, R0
STORE R0, (ADR_PODAT)

NEMOJ LOAD R0, (BROJAC) ; provjera brojača...
SUB R0, 1, R0 ; ... primljenih podataka
STORE R0, (BROJAC)
JR_NZ VAN ; ima još podataka ->VAN

STOP STORE R0, (VJ_STOP) ; zaustavi VJ i procesor
MOVE 1, R0
STORE R0, (PROC_HALT)
>>>
```

```
<<<
GLAVNI ; ne treba dozvoliti INT3 (niti se to može)
PETLJA LOAD R0, (PROC_HALT) ; "koristan posao"
OR R0, R0, R0
JR_Z PETLJA ; nastavi ako je 0
HALT
```

Napomena: u praksi se u glavnom programu umjesto prazne petlje izvodi neki koristan posao, koji je povremeno prekidan od prekidnih vanjskih jedinica

## Prekidni prijenos - Primjeri

### Primjer:

FRISC treba poslati 100 16-bitnih podataka iz bloka memorije na adresi 1000 na prekidnu VJ na adresi FFFF0000. VJ je spojena na INT0. Nakon prijenosa cijelog bloka treba zaustaviti rad prekidne VJ, a glavni program treba nastaviti s radom.

### Rješenje:

Budući da je VJ spojena na INT0, očito se radi o maskirajućem prekidu

Prekidni potprogram stavit ćemo na lokaciju 200 (adresu smo proizvoljno odabrali, jer nije zadana), a **prekidni vektor na adresi 8 moramo inicijalizirati na tu adresu** (200).

Na početku glavnog programa **moramo omogućiti prekide**.

&gt;&gt;&gt;

© Kovač, Basch, FER, Zagreb

134

## Prekidni prijenos - Primjeri

&lt;&lt;&lt;

```
; GLAVNI PROGRAM
GLAVNI MOVE 1000, R0 ; adresa podataka
STORE R0, (PODATAK)

MOVE 100, R0 ; brojač podataka
STORE R0, (BROJAC)

; DOZVOLI PREKID NA INT0
MOVE %B 10010000, SR

; "koristan posao"
PETLJA JP PETLJA
```

&gt;&gt;&gt;

© Kovač, Basch, FER, Zagreb

136

## Prekidni prijenos - Primjeri

```
<<< SUB R0, 1, R0 ; smanjenje brojača
STORE R0, (BROJAC)
JR_NZ IMA_JOS

ZADNJI MOVE 0, R0 ; ako je zadnji podatak
STORE R0, (STOP) ; zaustavi VJ

IMA_JOS POP R0 ; obnavljanje konteksta
MOVE R0, SR
POP R2
POP R1
POP R0

STORE R0, (IEND) ; kraj posluživanja
RETI
```

&gt;&gt;&gt;

© Kovač, Basch, FER, Zagreb

138

## Prekidni prijenos - Primjeri

Komentari:

- U kontekst prekidnog potprograma ulazi i SR\* (osim u rijetkim slučajevima kad se SR ne mijenja u prekidnom potprogramu)
- Za obične potprograme SR ne ulazi u kontekst, jer pozivatelj može pretpostaviti da će potprogram promijeniti SR i zato pozivatelj nikada nema u SR-u neko stanje koje će mu trebati nakon povratka iz potprograma (Ako pozivatelju običnog potprograma treba stanje iz SR-a, onda ga pozivatelj treba spremiti).

\* Neki procesori automatski spremaju statusni registar prilikom prihvatanja prekida

## Prekidni prijenos - Primjeri

```
SEND `EQU 0FFFF0000
IACK `EQU 0FFFF0004
IEND `EQU 0FFFF0008
STOP `EQU 0FFFF000C

`ORG 0
MOVE 10000, R7 ; početak izvođenja
JP GLAVNI ; preskakanje vektora

; PREKIDNI VEKTOR na adresi 8
`ORG 8
DW 200 ; adresa p.p.
```

&gt;&gt;&gt;

© Kovač, Basch, FER, Zagreb

135

## Prekidni prijenos - Primjeri

```
<<< ; PREKIDNI POTPROGRAM NA ADRESI 200
`ORG 200
PUSH R0 ; spremanje konteksta
PUSH R1
PUSH R2
MOVE SR, R0
PUSH R0

STORE R0, (IACK) ; prihvaćen prekid
LOAD R0, (BROJAC) ; dohvati varijabli
LOAD R1, (PODATAK)

LOADH R2, (R1) ; čitanje iz memorije
STORE R2, (SEND) ; i slanje na VJ
ADD R1, 2, R1 ; pomicanje pokazivača
STORE R1, (PODATAK)
```

© Kovač, Basch, FER, Zagreb

137

## Prekidni prijenos - Primjeri

```
<<<
BROJAC DW 0 ; variabile za p.p.
PODATAK DW 0

; Podatci iz memorije koji se šalju na VJ
`ORG 1000
DH 12, 4, 456A, 1, 0AB, 2, 885, ...
```

© Kovač, Basch, FER, Zagreb

139

## Posluživanje više prekidnih VJ



- Do sada smo vidjeli samo najjednostavniji slučaj kad je na procesor spojena jedna prekidna VJ
- Kad postoji više prekidnih VJ, one se mogu posluživati sa ili bez grijježenja (engl. nesting):
  - **bez grijježenja** prekidnih potprograma:
    - dok se poslužuje jedna VJ, drugi prekidi se ne prihvataju
    - jednostavniji slučaj
  - **sa grijježenjem** prekidnih potprograma:
    - dok se poslužuje jedna VJ, može se prihvati drugi prekid (većeg prioriteta)
    - komplikiraniji slučaj

## Posluživanje više prekidnih VJ

• Svim vanjskim jedinicama treba dodijeliti različite prioritete, što se može napraviti:

- programski (**FRISC za maskirajuće**)

• sklopovski

- sam procesor ima više prekidnih priključaka s različitim prioritetima (**FRISC: INT3 je prioritetniji od INT0-INT2**)
- prioritetni lanac vanjskih jedinica (daisy-chain)
- jedinica za kontrolu prioriteta (programmable interrupt controller ili priority interrupt controller)

### Prioriteti imaju dvojaku ulogu:

- kod istovremenih prekida određuje se kojoj VJ se prihvata prekid (služi i za grijanje prekida i kad nema grijanja)
- za vrijeme obrade jednog prekida određuje hoće li se prihvati novi prekid (služi samo za grijanje prekida)



## Posluživanje više prekidnih VJ

• Bez obzira kako se poslužuju, **uvijek treba odrediti uzročnike prekida** o čemu ovisi koju VJ ćemo poslužiti

• Ovisno o prekidnom sustavu procesora, moguća su različita rješenja:

- VJ sklopovski utječe na odabir adrese prekidnog potprograma čime se automatski određuje uzročnik prekida
- Adresa prekidnog potprograma bira se na temelju ulaznog prekidnog priključka čiji prekid je prihvaci (FRISC: INT3 ima različitu adresu p.p. od INT0-INT2)
- Programski se određuje koja jedinica je izazvala prekid (FRISC: za maskirajuće prekide)
  - ispitivanje ovisi o načinu spajanja VJ (svaka VJ na svom priključku INTi ili više VJ na istom priključku INTi)

## Posluživanje više prekidnih VJ



- Ispitivanje uzročnika prekida kod maskirajućeg prekida\*:

- **ispituje se spremnost VJ** (bistabil stanja)\*\*
- ovo **ne treba miješati s ispitivanjem uvjetnih VJ**, jer se ovdje samo jednom ispita spremnost, tj. nema čekanja da VJ postane spremna niti se obavlja proviziranje
- **redoslijed ispitivanja definira prioritete VJ:**
  - jedinice koje se prije ispituju imaju veći prioritet

\* Ako se na INT3 želi spojiti više VJ, onda one ne mogu koristiti IACK

\*\* Mogu se ispitivati i signali INT0, INT1, INT2 pomoću virtualnih bitova registra SR, ako se želi ispitati s kojeg prekidnog ulaza dolazi zahtjev za prekid

## Prekidni prijenos - Primjer

```

PRIMIO `EQU 0FFFF0000
SALJI1 `EQU 0FFF1000
ISPITAJ1 `EQU 0FFF1004
BRISI1 `EQU 0FFF1004
POSLUZEN1 `EQU 0FFF1008

SALJI2 `EQU 0FFF2000
ISPITAJ2 `EQU 0FFF2004
BRISI2 `EQU 0FFF2004
POSLUZEN2 `EQU 0FFF2008

SALJI3 `EQU 0FFF3000
POSLUZEN3 `EQU 0FFF3008

`ORG 0
MOVE 10000, SP
JP GLAVNI
 >>>

```

## Prekidni prijenos - Primjer

```

<<<
;;;;;;; Glavni program
GLAVNI ; dozvoli prekid na INT0
MOVE %B 10010000, SR

 ; "koristan posao"
PETLJA JR PETLJA

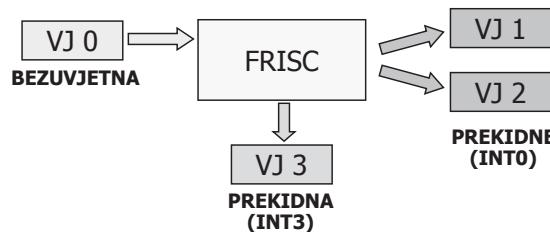
 ; brojač poslanih podataka
BROJAC DW 0
 >>>

```

## Prekidni prijenos - Primjer

### Primjer:

Na FRISC su spojene vj0, vj1, vj2 i vj3 na adresama FFFF0000, FFFF1000, FFFF2000 i FFFF3000. Vj0 je ulazna bezuvjetna, vj1 i vj2 izlazne prekidne jedinice spojene na INT0 (ne mogu se međusobno prekidati), a vj3 je izlazna prekidna jedinica spojena na INT3. Procesor šalje podatke s vj0 na vj1 i vj2 i prebraja koliko je podataka poslao. Kad vj3 zatraži prekid, treba joj poslati broj do tada prenesenih podataka.



## Prekidni prijenos - Primjer

```

<<<
; prekidni vektor za maskirajući prek.
`ORG 8
DW 100

; prekidni potprogram za
; nemaskirajući prekid na adresi 1210
`ORG 0C
PUSH R0
LOAD R0, (BROJAC) ; broj poslanih
STORE R0, (SALJI3) ; pošalji na vj3
POP R0
STORE R0, (POSLUZEN3) ; dojava kraja
RETN
 >>>

```

## Prekidni prijenos - Primjer

```

<<<
; prekidni potprogram
`ORG 100
PUSH R0 ; spremanje
MOVE SR, R0 ; konteksta
PUSH R0

ISPITAJ LOAD R0, (ISPITAJ1) ; otkrivanje
AND R0, 1, R0 ; uzročnika
JR_NZ P_VJ1 ; prekida
JR P_VJ2

VAN POP R0
MOVE R0, SR ; obnova
POP R0 ; konteksta
RETI
 >>>

```

## Prekidni prijenos - Primjer

&lt;&lt;&lt;

P\_VJ1 ; dio za posluživanje vj1

```

STORE R0, (BRISI1) ; prihvaćen prekid
LOAD R0, (PRIMI0) ; čitaj bezuvjetnu vj0
STORE R0, (SALJI1) ; šalji na vj1

LOAD R0, (BROJAC) ; povećaj
ADD R0, 1, R0 ; brojač poslanih
STORE R0, (BROJAC) ; podataka

STORE R0, (POSLUZEN1) ; kraj posluživanja
JR VAN ; povratak

```

P\_VJ2 ; dio za posluživanje vj2
; analogno kao i P\_VJ1

...

© Kovač, Basch, FER, Zagreb

150

## Prekidni prijenos - Primjer

### Primjer:

Na FRISC su spojene dvije prekidne VJ: vj1 i vj2 na adresama FFFF1000 i FFFF2000. Obje VJ su spojene na INTO.

Program treba primati podatke od vj1 i spremati ih redom prispjeća u blok memorije od adrese 1000. Kad se u blok spremi 100 podataka, sljedeći podatci opet se pune od adrese 1000. Za svaki spremljeni podatak treba pozvati potprogram OBRADI kojemu se preko R0 predaje adresa novog podatka. Pretpostavlja se da je trajanje izvođenja potprograma OBRADI značajno.

Izlazna vj2 svako toliko zahtjeva prekid. Tada treba pozvati potprogram HITNO koji vrši rezultat preko R0 i taj rezultat treba poslati na vj1.

Vj2 je prioritetsnija od vj1. Prekidi se **mogu gnijezditi**. Program se izvodi beskonačno. Potprograme OBRADI i HITNO ne treba pisati.

© Kovač, Basch, FER, Zagreb

152

## Prekidni prijenos - Primjer

```

PRIMI1 `EQU 0FFF1000
ISPITAJ1 `EQU 0FFF1004
BRISI1 `EQU 0FFF1004
POSLUZEN1 `EQU 0FFF1008

SALJI2 `EQU 0FFF2000
ISPITAJ2 `EQU 0FFF2004
BRISI2 `EQU 0FFF2004
POSLUZEN2 `EQU 0FFF2008

`ORG 0
MOVE 10000, SP
JP GLAVNI

`ORG 8
DW 200 ; prekidni vektor >>>

```

© Kovač, Basch, FER, Zagreb

154

## Prekidni prijenos - Primjer

&lt;&lt;&lt; ;;;;;;; Prekidni potprogram

```

`ORG 200
PUSH R0 ; spremanje konteksta
MOVE SR, R0
PUSH R0
PUSH R1

LOAD R0, (ISPITAJ2) ; ispitivanje uzročnika
AND R0, 1, R0 ; prekida: prvo "jača"
JR NZ PVJ2 ; vj2 pa tek onda vj1
JR PVJ1

VAN POP R1 ; mjesto izlaska iz
POP R0 ; prekidnog potprograma sa
MOVE R0, SR ; obnovom konteksta
POP R0
RETI >>>

```

© Kovač, Basch, FER, Zagreb

156

## Prekidni prijenos - Primjer

&lt;&lt;&lt;

### Komentar:

Vj1 i vj2 se ne mogu međusobno prekidati, ali u prekidnom potprogramu se prvo ispituje vj1 pa će ona biti prioritetsnija od vj2 u smislu da će kod istovremenog prekida prva biti poslužena vj1.

Kod istovremenog prekida dešava se sljedeće:

Prihvata se prekid čime se automatski zabrani prihvaćanje daljnjih prekida. U p.p.-u se ustanovi da je vj1 izazvala prekid te se obraduje njen prekid. Cijelo to vrijeme vj2 zahtjeva prekid, ali je prihvaćanje prekida u procesoru zabranjeno i prekid od vj2 se ne prihvata: kažemo da je prekid "na čekanju" (tzv. pending interrupt).

Nakon povratka iz prekidnog potprograma od vj1, doći će do omogućavanja prekida (naredba RETI). Tada će prekid od vj2 konačno biti prihvatan te će se skočiti u prekidni potprogram koji će sada poslužiti vj2.

© Kovač, Basch, FER, Zagreb

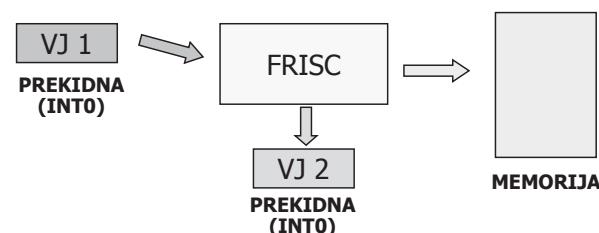
151

## Prekidni prijenos - Primjer

### Rješenje:

Prekidni potprogram prvo mora ustanoviti uzročnike prekida. Najprioritetniji među njima bit će poslužen.

Ukoliko je uzročnik prekida vj1, onda treba dozvoliti prekide.



© Kovač, Basch, FER, Zagreb

153

## Prekidni prijenos - Primjer

&lt;&lt;&lt;

;;;;;; Glavni program

```

GLAVNI MOVE 1000, R0 ; inicijalizacija
 STORE R0, (ABLOK) ; varijabli za prvo
 MOVE 100, R0 ; izvođenje prekida
 STORE R0, (BROJAC) ; nog potprograma
 ; dozvoli prekid na INTO
 MOVE %B 10010000, SR
 ; "beskonačan koristan posao"
PETLJA JR PETLJA

ABLOK DW 0 ; adresa za sljedeći podatak
BROJAC DW 0 ; brojač poslanih podataka

```

&gt;&gt;&gt;

© Kovač, Basch, FER, Zagreb

155

## Prekidni prijenos - Primjer

&lt;&lt;&lt; ;;;;;;; Posluživanje prioritetsnije vj2

```

PVJ2 STORE R0, (BRISI2) ; dojava prihvata
CALL HITNO
STORE R0, (SALJI2) ; slanje na vj2
STORE R0, (POSLUZEN2) ; dojava kraja
JR VAN

```

;;;;;; Posluživanje manje prioritetsne vj1

```

PVJ1 STORE R0, (BRISI1) ; dojava prihvata
 ;;;;;;; Dozvoli gniježdenje prekida
 MOVE SR, R0
 OR R0, %B 10000000, R0 ; "digni" GIE
 MOVE R0, SR

```

&gt;&gt;&gt;

© Kovač, Basch, FER, Zagreb

157

## Prekidni prijenos - Primjer

```
<<< LOAD R0, (ABLOK) ; učitaj adresu za podatak
LOAD R1, (PRIMI1) ; čitaj podatak sa vj1
STORE R1, (R0) ; spremi ga u blok
CALL OBRADI ; obradi podatak (R0 je OK)

ADD R0, 4, R0 ; povećaj adresu podatka
LOAD R1, (BROJAC) ; učitaj brojač
SUB R1, 1, R1 ; smanji ga i provjeri je
JR NZ DALJE ; li blok napunjeno

ZADNJI MOVE 1000, R0 ; ako je blok pun, onda
MOVE 100, R1 ; vrati početne vrijednosti

DALJE STORE R0, (ABLOK) ; spremi varijable
STORE R1, (BROJAC)

STORE R0, (POSLUZEN1) ; dojava kraja
JR VAN
```

© Kovač, Basch, FER, Zagreb

158

## Prekidni prijenos - Primjer

<<< Komentari:

Uočite da je na jedan prekidni priključak moguće spojiti najviše dvije VJ koje mogu prekidati jedna drugu. "Slabija" VJ omogući prekid i tada "jača" može izazvati prekid koji će se ugnijezditi. Za vrijeme slabije VJ ne može izazvati novi prekid jer njen prekidni potprogram nije dovršen (nije joj dojavljen kraj).

Prepostavimo da na INT0 spojimo vj1, vj2 i vj3, pri čemu je vj1 najslabija, a vj3 najjača. Za vrijeme obrade vj2 bi trebali dozvoliti prekidanje tako da se može prihvati prekid sa vj3. Međutim, tada bi bio prihvачen i prekid sa vj1 koja je slabija.

Ovaj problem se može rješiti sklopovski tako da postoje različiti prekidni priključci za svaku jedinicu, ili da se jedinici zabrani postavljanje prekida dok se obrađuje prekid jače jedinice (daisy chain), ili da jedinice postavljaju prekide posredno preko posebnog sklopa koji propušta samo prekide višeg prioriteta od jedinice koja se trenutačno obrađuje (priority interrupt controller). Moguće je i programsko rješenje (relativno složeno).

© Kovač, Basch, FER, Zagreb

160

## Prekidni prijenos - Primjeri



### Rješenje a (bez prekida u prekidu):

```
READ_0 `EQU 0FFF0000
IACK_0 `EQU 0FFF0004
READY_0 `EQU 0FFF0004
IEND_0 `EQU 0FFF0008
STOP_0 `EQU 0FFF000C

READ_1 `EQU 0FFF1000
IACK_1 `EQU 0FFF1004
READY_1 `EQU 0FFF1004
IEND_1 `EQU 0FFF1008
STOP_1 `EQU 0FFF100C

READ_2 `EQU 0FFF2000
IACK_2 `EQU 0FFF2004
READY_2 `EQU 0FFF2004
IEND_2 `EQU 0FFF2008
STOP_2 `EQU 0FFF200C
```

>>>

© Kovač, Basch, FER, Zagreb

162

## Prekidni prijenos - Primjeri



```
<<< ; PREKIDNI POTPROGRAM NA ADRESI 500
`ORG 500
PP PUSH R0 ; spremanje konteksta: R0 i SR
MOVE SR, R0
PUSH R0

; OTKRIVANJE UZROČNIKA PREKIDA
CHK_0 LOAD R0, (READY_0) ; je li VJ0 spremna?
AND R0, 1, R0
JR Z CHK_1 ; ako nije, provjeri VJ1

STORE R0, (IACK_0) ; potvrda prekida
LOAD R0, (READ_0) ; čitaj podatak iz VJ0
PUSH R0 ; pošalji ga preko stoga
CALL PVJ0 ; pozovi posluživanje VJ0
ADD SP, 4, SP ; ukloni parametar
STORE R0, (IEND_0) ; kraj posluživanja
JR NATRAG ; povratak

>>>
```

© Kovač, Basch, FER, Zagreb

164

## Prekidni prijenos - Primjer

<<< Komentari:

U posluživanju VJ1 (slajd 152) se prvo dojavljuje jedinici VJ1 da je prekid prihvачen (na što VJ1 uklanja svoj zahtjev za prekidom), a tek onda se ponovno omogući prekid.

Da je prvo omogućen prekid, onda bi VJ1 još uvijek držala svoj zahtjev za prekidom i odmah bi došlo do novog prekida što bi se ponavljalo, tj. došlo bi do beskonačnog grijezđenja prekida sa VJ1.

>>>

© Kovač, Basch, FER, Zagreb

159

## Prekidni prijenos - Primjeri



### Primjer:

Na FRISC su spojene tri prekidne jedinice vj0, vj1 i vj2 redom na prekidne priključke INT0, INT1 i INT2. Adrese su im redom FFFF0000, FFFF1000 i FFFF2000. Vj0 ima najveći, a vj2 najmanji prioritet.

S navedenih jedinica primaju se podatci koji se preko stoga šalju odgovarajućem potprogramu za obradu primljenog podatka. Prepostavka je da potprogrami već postoje i da se zovu PVJ0, PVJ1 i PVJ2 (Posluži VJ n). FRISC treba kontinuirano primati podatke i obradivati ih.

Zadatak treba rješiti:

- a) bez grijezđenja prekida
- b) s grijezđenjem prekida

>>>

© Kovač, Basch, FER, Zagreb

161

## Prekidni prijenos - Primjeri



<<<

```
`ORG 0
MOVE 10000, SP
JP GLAVNI ; preskoči prekidni vektor

`ORG 8 ; prekidni vektor na adresi 8
DW 500 ; adresa p.p. je 500

;;;;;; GLAVNI PROGRAM ;;;;;;
; dozvoli sve maskirajuće prekide
GLAVNI MOVE %B 11110000, SR

; "koristan posao"
PETLJA JP PETLJA
```

>>>

© Kovač, Basch, FER, Zagreb

163

## Prekidni prijenos - Primjeri



<<<

```
CHK_1 LOAD R0, (READY_1) ; je li VJ1 spremna?
AND R0, 1, R0
JR Z CHK_2 ; ako nije, provjeri VJ2

STORE R0, (IACK_1) ; sve isto
LOAD R0, (READ_1) ; kao za VJ0
PUSH R0
CALL PVJ1
ADD SP, 4, SP
STORE R0, (IEND_1)
JR NATRAG
```

>>>

© Kovač, Basch, FER, Zagreb

165



&lt;&lt;&lt;

```

CHK_2 ; VJ2 ne treba stvarno provjeravati, jer ako
; VJ0 i VJ1 nisu spremne onda je sigurno VJ2
; izazvala prekid

STORE R0, (IACK_2) ; sve isto kao za
LOAD R0, (READ_2) ; VJ0 i VJ1
PUSH R0
CALL PVJ2
ADD SP, 4, SP
STORE R0, (IEND_2)

NATRAG POP R0 ; obnova konteksta
MOVE R0, SR
POP R0
RETI

```

© Kovač, Basch, FER, Zagreb

166



```

<<< ; PREKIDNI POTPROGRAM NA ADRESI 500
`ORG 500
PP PUSH R0 ; spremanje konteksta: R0 i SR
MOVE SR, R0
PUSH R0

; OTKRIVANJE UZROČNIKA PREKIDA
CHK_0 LOAD R0, (READY_0) ; je li VJ0 spremna?
AND R0, 1, R0
JR_Z CHK_1 ; ako nije, provjeri VJ1
STORE R0, (IACK_0) ; potvrda prekida
LOAD R0, (READ_0) ; čitaj podatak iz VJ0
PUSH R0 ; pošalji ga preko stoga
CALL PVJ0 ; pozovi posluživanje VJ0
ADD SP, 4, SP ; ukloni parametar
STORE R0, (IEND_0) ; kraj posluživanja
JR NATRAG ; povratak
>>>

```

© Kovač, Basch, FER, Zagreb

168



```

<<<
CHK_2 ; VJ2 ne treba stvarno provjeravati, jer ako
; VJ0 i VJ1 nisu spremne onda je sigurno VJ2
; izazvala prekid

STORE R0, (IACK_2) ; sve isto kao za
; dozvoli prekid prioritetnijim VJ0 i VJ1
MOVE %B 10110000, SR

LOAD R0, (READ_2) ; VJ0 i VJ1
PUSH R0
CALL PVJ2
ADD SP, 4, SP
STORE R0, (IEND_2)

NATRAG POP R0 ; obnova konteksta
MOVE R0, SR
POP R0
RETI
>>>

```

© Kovač, Basch, FER, Zagreb

170



## Rješenje b (s prekidom u prekidu):

program je vrlo sličan, istaknute su samo razlike

```

READ_0 `EQU 0FFFFF0000
...
STOP_2 `EQU 0FFFFF200C

`ORG 0
MOVE 10000, SP
JP GLAVNI ; preskoči prekidni vektor
`ORG 8 ; prekidni vektor na adresi 8
DW 500 ; adresa p.p. je 500
;;;;;; GLAVNI PROGRAM ;;;;;;;
GLAVNI MOVE %B 11110000, SR ; dozvoli prekide
PETLJA JP PETLJA ; "koristan posao"

```

© Kovač, Basch, FER, Zagreb

&gt;&gt;&gt;

167



```

<<<
CHK_1 LOAD R0, (READY_1) ; je li VJ1 spremna?
AND R0, 1, R0
JR_Z CHK_2 ; ako nije, provjeri VJ2
STORE R0, (IACK_1)

; dozvoli prekid prioritetnijoj VJ0
MOVE %B 10010000, SR

LOAD R0, (READ_1)
PUSH R0
CALL PVJ1
ADD SP, 4, SP
STORE R0, (IEND_1)
JR NATRAG
>>>

```

© Kovač, Basch, FER, Zagreb

169



## Komentari:

- Programi a) i b) su vrlo slični, ali se u slučaju b) izvode na bitno drugačiji način:
  - za vrijeme posluživanja VJ2 prihvaćaju se prekidi od VJ0 i VJ1
  - za vrijeme posluživanja VJ1 prihvaća se prekid od VJ0
  - za vrijeme posluživanja VJ0 ne prihvaćaju se daljnji prekidi
- Spajanjem svake jedinice na vlastiti prekidni priključak omogućeno je da imamo tri razine prekida
- Uočite da se u prekidnom potprogramu mijenjaju bitovi EINTi u SR-u. Izlaskom iz prekidnog potprograma, obnavlja se stanje SR-a tako da se opet mogu prihvati upravo oni prekidi koji su bili omogućeni prije ulaska u prekidni potprogram.

© Kovač, Basch, FER, Zagreb

171

## Sklop FRISC-CT

- Sklop FRISC-CT, ili kraće CT (kratica od Counter Timer) služi za brojenje impulsa i mjerjenje vremena
  - Za razliku od dosadašnjih VJ, CT ne prenosi podatke
  - Slični sklopovi postoje i za komercijalne procesore
- CT osobađa procesor od nepotrebnih čekanja ili čestih posluživanja prekida:
  - Ako se čeka na istek vremena, onda procesor izvodi praznu petlju da bi ostvario zadano kašnjenje, pri čemu se ne obavlja nikakav koristan posao
  - Ako se prebrajavaju impulsi koji pristižu relativno velikom brzinom, onda bi procesor npr. mogao na svaki takav impuls dobiti zahtjev za prekid te u prekidnom potprogramu povećavati neki brojač u memoriji

## Sklop FRISC-CT

- Za razliku od do sada pokazanih vanjskih jedinica koje su bile općenite i imale su nepromjenjivu zadaću, CT je sklop koji se može "programirati" (točnije: konfigurirati)
- Na taj način se ponašanje i funkcija vanjske jedinice prilagođavaju konkretnoj zadaći koju treba napraviti
- Programabilnost je često svojstvo kod stvarnih vanjskih jedinica za komercijalne procesore
- Programiranje sklopa se ostvaruje slanjem posebnih upravljačkih riječi na određene adrese na kojima se sklop nalazi

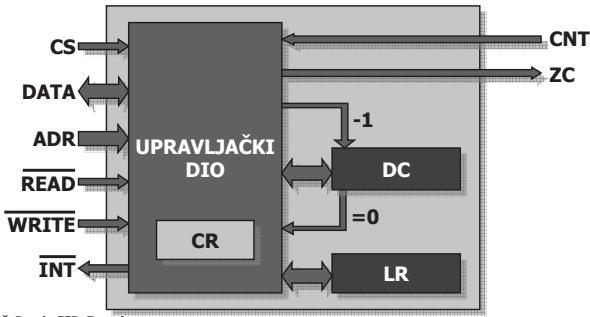
© Kovač, Basch, FER, Zagreb

174

## Blok-shema CT-a

- LR (Limit Register) je 16-bitni registar granice ili registar vremenske konstante (vrijednost mu se zadaje programski)
- DC (down counter) je 16-bitno brojilo na dolje (vrijednost upisana u LR automatski se upiše i u DC)

**FRISC-CT**



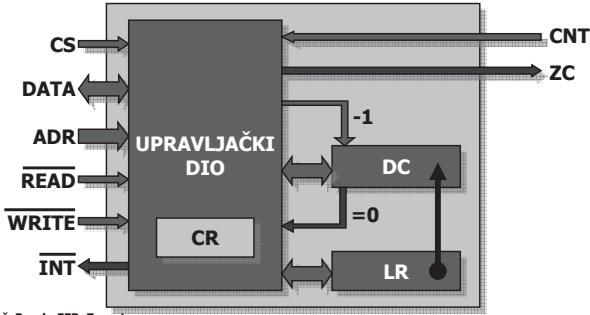
© Kovač, Basch, FER, Zagreb

176

## Blok-shema CT-a

- Kad DC smanjivanjem dođe do ništice, dešava se sljedeće:
  - Generira se impuls na izlazu ZC
  - Vrijednost iz LR puni se u DC i nastavlja se s brojenjem
  - Postavlja se stanje spremnosti, a prekid se generira ako je omogućen pomoću registra CR

**FRISC-CT**



© Kovač, Basch, FER, Zagreb

178

## Ponašanje CT-a

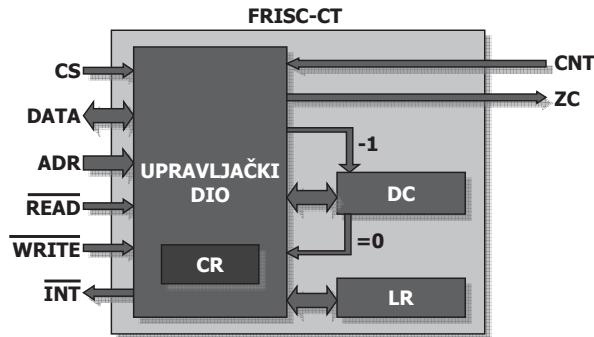
- Upravljačka riječ koja se upisuje u upravljački register CR:
- |                                                  |            |     |
|--------------------------------------------------|------------|-----|
| 31-2                                             | 1          | 0   |
| neiskorišteno                                    | STOP/START | INT |
| 0 - brojilo je zaustavljeno<br>1 - brojilo broji |            |     |
| 0 - ne postavlja prekid<br>1 - postavlja prekid  |            |     |
- Ako se bitom 0 u CR-u zada da se ne postavljaju prekidi, CT se može koristiti kao uvjetna vanjska jedinica
  - Tek nakon dojave kraja posluživanja, CT može ponovno postati spreman (naravno, kad brojilo dode do ništice):
    - Ako brojilo ponovo dode do ništice prije nego je dojavljen kraj posluživanja, ne postavlja se spremnost i taj ciklus brojenja je izgubljen

© Kovač, Basch, FER, Zagreb

180

## Blok-shema CT-a

- Sučelje za spajanje s FRISC-om, isto je kao za prekidne jedinice
- CR (Control Register) je upravljački registar kojim se može:
  - omogućiti ili onemogućiti postavljanje zahtjeva za prekid
  - omogućiti ili zaustaviti rad CT-a (brojenje)

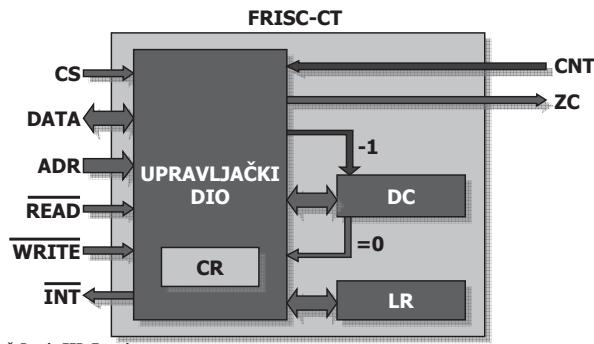


© Kovač, Basch, FER, Zagreb

175

## Blok-shema CT-a

- Preko ulaznog priključka CNT (count) dolaze impulsi koje CT prebraja
- Na svaki impuls smanjuje se brojilo DC za jedan (ako je omogućen rad CT-a pomoću upravljačkog registra CR)



© Kovač, Basch, FER, Zagreb

177

## Adrese CT-a

- CT zauzima četiri uzastopne 32-bitne lokacije, počevši od početne adrese (PA) djeljive sa 16:

| Adresa | Pisanje                                              | Čitanje                      |
|--------|------------------------------------------------------|------------------------------|
| PA     | Upis u LR (i DC)                                     | trenutačno stanje DC-a       |
| PA+4   | Upis u CR                                            | trenutačno stanje spremnosti |
| PA+8   | brisanje stanja spremnosti (dojava prihvata prekida) | -                            |
| PA+12  | dojava kraja posluživanja                            | -                            |

- Prilikom upisa u LR uzima se u obzir samo nižih 16 bita, a ostali se zanemaruju (upisani podatak odmah se proslijedi u i DC)
- Prilikom dojave prihvata prekida i kraja prekida, poslani podatak se zanemaruje

© Kovač, Basch, FER, Zagreb

179

## Ponašanje CT-a

- Iz opisa ponašanja CT-a u slučaju kad brojilo DC dođe do ništice vidi se sljedeće:
  - Brojilo DC se automatski puni prethodnom konstantom iz LR
  - Brojilo nastavlja svoju funkciju
- Ovo je naročito praktično ako treba ponavljati prethodni ciklus brojenja (što je čest slučaj). U tom slučaju ne treba ponovno programirati CT, nego se sve obavi automatski
- Ako treba zaustaviti brojenje ili promijeniti konstantu brojenja, onda treba obaviti preprogramiranje CT-a odmah po primitu prekida (ili čim se uvjetnim posluživanjem ustanovi spremnost CT-a)

© Kovač, Basch, FER, Zagreb

181

**Primjer (brojenje impulsa):**

Na CT-ov priključak CNT spojen je izlaz iz stroja koji za svaki proizvedeni vijak generira impuls. Računalo mora u lokaciji BR\_PAK prebranjati proizvedene pakete od po 200 vijaka. Treba riješiti zadatak CT-om (na adresi FFFF0000) tako:

- a) da CT radi u uvjetnom načinu,
- b) da CT radi u prekidnom načinu i spojen je na INT0.

&gt;&gt;&gt;

&lt;&lt;&lt;

```
PETLJA LOAD R0, (CTSTAT) } čekanje spremnosti
 AND R0, 1, R0 } tj. čekanje da se
 JR_Z PETLJA } proizvede paket

 STORE R0, (CTCLR) }-brisanje spremnosti
 LOAD R0, (BR_PAK) }
 ADD R0, 1, R0 } povećaj brojač
 STORE R0, (BR_PAK) }

 STORE R0, (CTEND) }-kraj posluživanja
 JR PETLJA

BR_PAK DW 0
```

&lt;&lt;&lt;

```
GLAVNI ; GLAVNI PROGRAM

; INICIJALIZACIJA CT-a
MOVE %D 200, R0 ; GRANICA BROJENJA
STORE R0, (CTLR)

; KONTR. RIJEČ
MOVE %B 11, R0 ; START/STOP = 1
STORE R0, (CTCR) ; INT = 1

MOVE %B 10010000, SR ; OMOGUĆI PREKID

; "KORISTAN POSAO"
PETLJA JR PETLJA
 >>>
```

**Primjer (mjerjenje vremena):**

FRISC treba svake sekunde izvesti potprogram POTP. Vremensko kašnjenje treba ostvariti pomoću sklopova CT, a ne programskom petljom za kašnjenje. Prepostavka je da program POTP već postoji i da njegovo izvođenje sigurno traje kraće od jedne sekunde. CLOCK ima frekvenciju 10 MHz.

**Rješenje:**

Ovaj primjer pokazuje upotrebu CT-a za ostvarivanje kašnjenja.

Na ulaz CNT treba dovesti signal poznate frekvencije (npr. CLOCK), a iznosom u registru LR određujemo trajanje između dva prekida.

Trebamo podijeliti frekvenciju 10MHz na 1Hz (1 sek) što je djeljenje od 10M, tj. u LR bi trebali upisati broj 10 000 000.

&gt;&gt;&gt;

**Rješenje a (uvjetni način):**

```
CTLR `EQU 0FFFF0000
CTCR `EQU 0FFFF0004
CTSTAT `EQU 0FFFF0004
CTCLR `EQU 0FFFF0008
CTEND `EQU 0FFFF000C

`ORG 0
; GLAVNI PROGRAM

; INICIJALIZACIJA CT-a
GLAVNI MOVE %D 200, R0
STORE R0, (CTLR)

MOVE %B 10, R0 ; START/STOP = 1, INT = 0
STORE R0, (CTCR)
 >>>
```

**Rješenje b (prekidni način):**

```
CTLR `EQU 0FFFF0000
CTCR `EQU 0FFFF0004
CTIACK `EQU 0FFFF0008
CTIEND `EQU 0FFFF000C

`ORG 0
MOVE 10000, R7
JP GLAVNI

; PREKIDNI VEKTOR
`ORG 8
DW 1000
 >>>
```

Napomena:

Iako je rješenje s prekidom, dulje, izvodi se puno efikasnije, jer uvjetno posluživanje troši gotovo svo vrijeme samo na ispitivanje spremnosti CT-a.

```
<<< `ORG 1000 ;;;;;
 PUSH R0
 MOVE SR, R0 } spremanje konteksta
 PUSH R0
 STORE R0, (CTIACK) }-obriši spremnost

 LOAD R0, (BR_PAK) } povećaj
 ADD R0, 1, R0 } brojač
 STORE R0, (BR_PAK) } paketa

 POP R0
 MOVE R0, SR } obnova konteksta
 POP R0
 STORE R0, (CTIEND) }-dojava kraja
 RETI

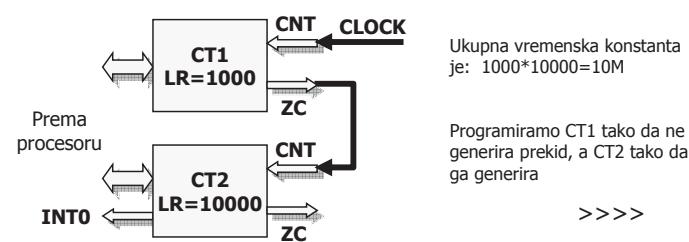
BR_PAK DW 0 ; BROJAČ PAKETA
```

&lt;&lt;&lt;

S obzirom da je LR 16-bitni, u njega se može upisati najveća konstanta 65536<sub>10</sub> (tj. 0), što nije dovoljno.

Jedan CT nije dovoljan pa koristimo dva CT-a. Na prvi CT dovodimo CLOCK, a izlaz prvog CT-a (tj. priključak ZC) spajamo na drugi CT.

To je glavna namjena priključka ZC: **ulančavanje više CT-ova**.



<<<

```

CTLR1 `EQU OFFFF1000
CTCR1 `EQU OFFFF1004

CTLR2 `EQU OFFFF2000
CTCR2 `EQU OFFFF2004
CTIACK2 `EQU OFFFF2008
CTIEND2 `EQU OFFFF200C

`ORG 0
MOVE 10000, R7
JP GLAVNI

`ORG 8
DW 1000 ; prekidni vektor >>>

```

© Kovač, Basch, FER, Zagreb

190

© Kovač, Basch, FER, Zagreb

>>>

<<<

```

; prekidni potprogram - izvodi se svake sek.
; kontekst se ne spremi jer se ne mijenjaju registri
; POTP mora spremati registre koje mijenja (i SR)

`ORG 1000
STORE R0, (CTIACK2) ; potvrda prekida
CALL POTP ;;; poziv zadano potprograma
STORE R0, (CTIEND2) ; potvrda kraja
RETI

```

© Kovač, Basch, FER, Zagreb

192

© Kovač, Basch, FER, Zagreb

193

## Sklop FRISC-PIO

- Sklop FRISC-PIO, ili kraće PIO (kratica od Parallel Input-Output) je sklop koji **služi za paralelni prijenos podataka** (8 bitnih)
  - Posrednik između procesora i procesa (djelomično sličan općim VJ kojima smo do sada prenosi podatke)
  - Slični skloovi postoje i za komercijalne procesore
- PIO je **programabilni sklop** koji može raditi u **4 načina rada**:
  - ulazni
  - ispitivanje bitova
  - izlazni
  - postavljanje bitova

© Kovač, Basch, FER, Zagreb

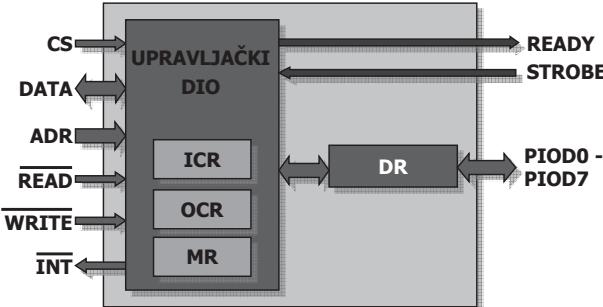
194

## Blok-shema PIO-a



- READY i STROBE su sinkronizacijski priključci za povezivanje s vanjskim procesom
- Način njihovog rada (rukovanje) jednak je kao što je pokazano za opću uvjetnu VJ

### FRISC-PIO



© Kovač, Basch, FER, Zagreb

196

<<<

```

GLAVNI ; inicijaliziraj CT1
MOVE %D 1000, R0 ; LR1=1000
STORE R0, (CTLR1)
MOVE %B 10, R0 ; CR1: bez INT
STORE R0, (CTCR1)
; inicijaliziraj CT2
MOVE %D 10000, R0 ; LR2=10000
STORE R0, (CTLR2)
MOVE %B 11, R0 ; CR2: postavlja INT
STORE R0, (CTCR2)
MOVE %B 10010000, SR ; omogući prekid INTO
; nastavak glavnog programa
...

```

>>>

© Kovač, Basch, FER, Zagreb

191

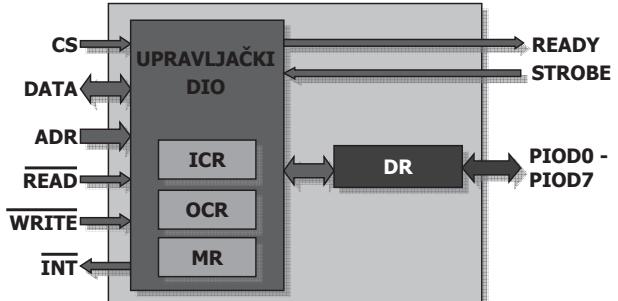
## Sklop FRISC-PIO

## Blok-shema PIO-a



- Sučelje za spajanje s FRISC-om, isto je kao za prekidne jedinice
- DR (Data Register) je 8-bitni podatkovni register iste namjene kao i kod općih VJ
- PIOD0-PIOD7 (PIO Data) su dvosmjerni priključci za prijenos podataka prema vanjskom procesu

### FRISC-PIO



© Kovač, Basch, FER, Zagreb

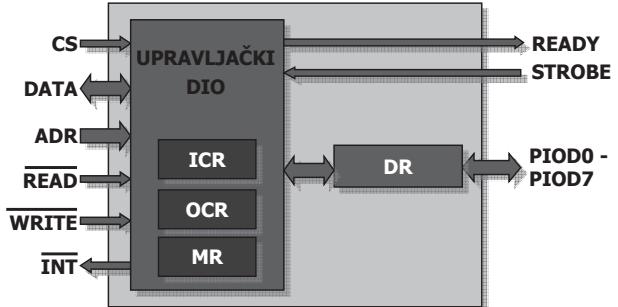
195

## Blok-shema PIO-a



- U upravljačkom dijelu su:
  - Ulazni upravljački registar ICR (Input Control Register)
  - Izlazni upravljački registar OCR (Output Control Register)
  - Registr maske MR (Mask Register)

### FRISC-PIO



© Kovač, Basch, FER, Zagreb

197

## Adrese sklopa PIO



- PIO zauzima četiri uzastopne 32-bitne lokacije, počevši od početne adrese (PA) djejive s 16:

| Adresa | Pisanje                                                 | Čitanje                      |
|--------|---------------------------------------------------------|------------------------------|
| PA     | - Upis u ICR<br>- Upis u OCR<br>- Upis u MR             | trenutačno stanje spremnosti |
| PA+4   | Upis u DR / PIOD                                        | čitanje DR / PIOD            |
| PA+8   | brisanje stanja spremnosti<br>(dojava prihvata prekida) | -                            |
| PA+12  | dojava kraja posluživanja                               | -                            |

- Prilikom upisa u DR, uzima se u obzir samo nižih 8 bita, a ostali se zanemaruju
- Prilikom dojave prihvata prekida i kraja prekida, poslani podatak se zanemaruje

© Kovač, Basch, FER, Zagreb

198

## Registar OCR



- Riječ koja se šalje u OCR građena je ovako:

- najniži bit mora biti 0**
- bit 1 kontrolira hoće li se postavljati prekid kad PIO postane spreman
- bit 2 kontrolira hoće li PIO raditi u izlaznom načinu ili u načinu postavljanja bitova
- ostali bitovi se ne koriste

| 31-3                                         | 2                                               | 1           | 0 |
|----------------------------------------------|-------------------------------------------------|-------------|---|
| -                                            | MODE                                            | INT         | 0 |
| 0 - izlazni način<br>1 - postavljanje bitova | 0 - ne postavlja prekid<br>1 - postavlja prekid | mora biti 0 |   |

© Kovač, Basch, FER, Zagreb

200

## Registar ICR



- bit 3 (mask follows):
  - ako je u stanju 1 onda će sljedeća riječ koja se pošalje na ovu adresu biti upisana u registar maske MR
  - ako je u stanju 0, onda će sljedeća riječ koja se pošalje na ovu adresu biti upisana u ICR ili OCR
- bit 4 definira **aktivnu** razinu bitova koji se ispituju
- bit 5 definira način ispitivanja bitova za postavljanje spremnosti:
  - ako je odabran OR, onda **barem jedan bit** mora biti **aktivan** da bi se postavila spremnost
  - ako je odabran AND, onda **svi ispitivani bitovi** moraju biti **aktivni** da bi se postavila spremnost

| 31-6              | 5                                                  | 4                                         | 3                                         | 2-0 |
|-------------------|----------------------------------------------------|-------------------------------------------|-------------------------------------------|-----|
| ...               | OR/AND                                             | ACTIVE                                    | MASK F.                                   | ... |
| 0 - OR<br>1 - AND | 0 - aktivna razina je 0<br>1 - aktivna razina je 1 | 0 - ne slijedi maska<br>1 - slijedi maska | 0 - ne slijedi maska<br>1 - slijedi maska |     |

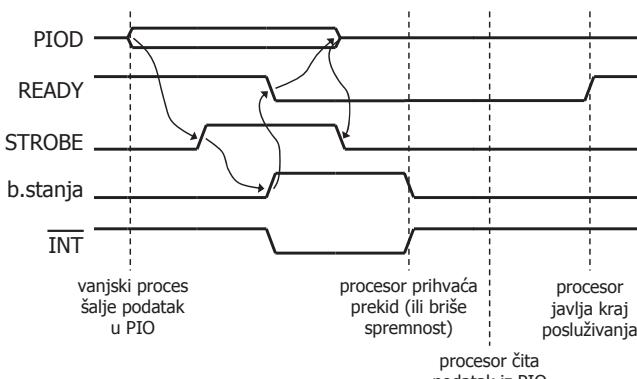
© Kovač, Basch, FER, Zagreb

202

## Uzlazni način rada



- Primaju se bajtovi podataka preko PIOD uz korištenje sinkronizacijskih priključaka (slično kao što je objašnjeno za uvjetne VJ)



© Kovač, Basch, FER, Zagreb

204

## Adrese sklopa PIO



- Kako se na jednu adresu mogu zapisivati tri različite stvari (ICR, OCR i MR)?
- Ako se na tu adresu pošalje podatak, onda najniži bit poslanog podatka određuje radi li se o upisu u ICR ili OCR:

|                                  |         |
|----------------------------------|---------|
| 31-1                             | 0       |
| značenje ovisi o bitu 0          | ICR/OCR |
| 0 - upis u OCR<br>1 - upis u ICR |         |

© Kovač, Basch, FER, Zagreb

199

## Registar ICR



- Riječ koja se šalje u ICR građena je ovako:

- najniži bit mora biti 1**
- bit 1 kontrolira hoće li se postavljati prekid kad PIO postane spreman
- bit 2 kontrolira hoće li PIO raditi u ulaznom načinu ili u načinu ispitivanja bitova
- bitovi 3, 4 i 5:
  - ne koriste se ako je bitom MODE odabran ulazni način
  - koriste se ako je bitom MODE odabran način ispitivanja bitova

| 31-6              | 5                                                  | 4                                         | 3                                          | 2                                               | 1           | 0 |
|-------------------|----------------------------------------------------|-------------------------------------------|--------------------------------------------|-------------------------------------------------|-------------|---|
| -                 | OR/AND                                             | ACTIVE                                    | MASK F.                                    | MODE                                            | INT         | 1 |
| 0 - OR<br>1 - AND | 0 - aktivna razina je 0<br>1 - aktivna razina je 1 | 0 - ne slijedi maska<br>1 - slijedi maska | 0 - ulazni način<br>1 - ispitivanje bitova | 0 - ne postavlja prekid<br>1 - postavlja prekid | mora biti 1 |   |

© Kovač, Basch, FER, Zagreb

201

## Registar MR



- Registar maske MR služi za odabir bitova koji će biti ispitivani** u načinu ispitivanja bitova

- Svaki bit u 8-bitnom registru maske odgovara jednom bitu na priključcima PIOD
- Ako je u MR upisano 0, onda se odgovarajući PIOD ne ispituje
- Ako je u MR upisano 1, onda se odgovarajući PIOD ispituje
- Ispitivanje nemaskiranih bitova provodi se na temelju bitova ACTIVE i AND/OR, kako je opisano prethodnim slajdom i na temelju toga se postavlja spremnost

| 31-6              | 5                                                  | 4                                         | 3                                         | 2-0 |
|-------------------|----------------------------------------------------|-------------------------------------------|-------------------------------------------|-----|
| ...               | OR/AND                                             | ACTIVE                                    | MASK F.                                   | ... |
| 0 - OR<br>1 - AND | 0 - aktivna razina je 0<br>1 - aktivna razina je 1 | 0 - ne slijedi maska<br>1 - slijedi maska | 0 - ne slijedi maska<br>1 - slijedi maska |     |

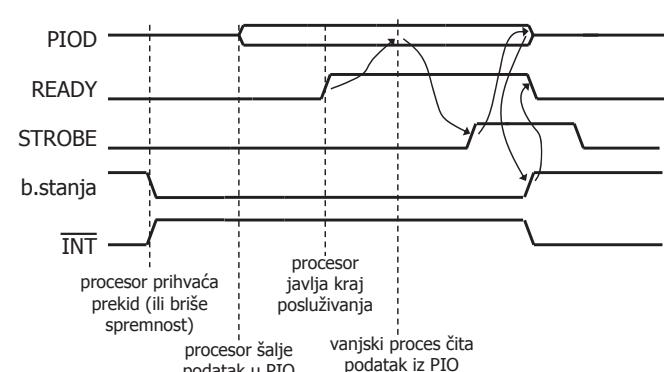
© Kovač, Basch, FER, Zagreb

203

## Izlazni način rada



- Šalju se bajtovi podataka preko PIOD uz korištenje sinkronizacijskih priključaka (slično kao što je objašnjeno za uvjetne VJ)



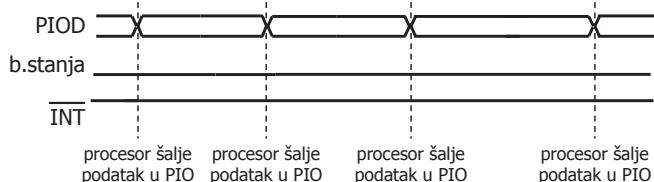
© Kovač, Basch, FER, Zagreb

205

## Način postavljanja bitova



- Šalju se podaci preko PIOD, ali ne koriste se sinkronizacijski priključci niti se postavlja spremnost (ni prekid)



- Podatci se jednostavno bezuvjetno šalju na PIO koji ih pamti u registru podataka DR te se odmah pojavljuju na priključcima PIOD

- Koristi se kad se želi izravno upravljati bitovima PIOD

## PIO - Primjeri

### Primjer:

Na pisač koji je spojen na PIO treba poslati  $80_{10}$  znakova smještenih u memoriji od lokacije ZNAKOVI (bajtni podaci). PIO radi u prekidnom načinu, a adresa mu je FFFF0000. Prepostavka je da nema drugih izvora prekida, a PIO je spojen na INT0.

### Rješenje:

PIO ćemo programirati da radi u izlaznom načinu, jer će s pisačem biti potrebna sinkronizacija (prepostavka je da pisač ima linije za rukovanje kompatibilne sa READY i STROBE).

>>>

## PIO - Primjeri

```
<<<
; glavni program

; inicializacija sklopa PIO
GLAVNI MOVE %B 010, R0 ; 0=mode OUT
 ; 1=INT
 ; 0=OCR
STORE R0, (PIOC) ; pošalji u OCR

MOVE %B 10010000, SR ; dozvoli INT0

PETLJA JR PETLJA ; "koristan posao"

BROJAC DW 0 ; brojač poslanih znakova
ZNAKOVI DB ... ; 80 znakova za slanje
>>>
```

## PIO - Primjeri

```
<<<
ADD R1, 1, R1 ; povećanje brojača
STORE R1, (BROJAC)

CMP R1, %D 80 ; je li poslan
JR_NE JOS ; zadnji znak ?

KRAJ ; zabrani PIO-u da dalje zahtijeva prekide
MOVE %B 000, R0 ; 0=izlaz, 0=no INT, 0=OCR
STORE R0, (PIOC)

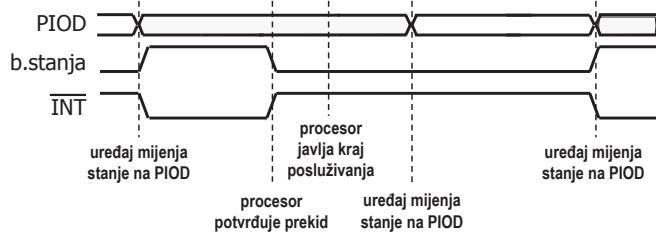
JOS ; ima još znakova za slanje
POP R0
MOVE R0, SR
POP R2
POP R1
POP R0
STORE R0, (PIOIEND) ; dojava kraja posluživ.

RETI
>>>
```

## Način ispitivanja bitova



- Vanjski proces upravlja sa PIOD i proizvoljno mijenja podatke bez korištenja sinkronizacijskih priključaka
- Spremnost se postavlja ovisno o uvjetu zadanom u registru ICR (na slici žuta boja označava stanje koje izaziva spremnost)



- Ako uređaj postavi novo stanje koje izaziva spremnost, a procesor još nije dojavio kraj posluživanja, neće se postaviti stanje spremnosti

## PIO - Primjeri

<<<

```
PIOC EQU 0FFFF0000
PIOD EQU 0FFFF0004
PIOIACK EQU 0FFFF0008
PIOIEND EQU 0FFFF000C

`ORG 0
MOVE 10000, R7
JP GLAVNI

`ORG 8 ; prekidni vektor
DW 500 >>>
```

## PIO - Primjeri

<<<

```
`ORG 500 ; prekidni potprogram
PUSH R0
PUSH R1
PUSH R2
MOVE SR, R0
PUSH R0

STORE R0, (PIOIACK) ; potvrda prekida
LOAD R1, (BROJAC) ; dohvati brojača
MOVE ZNAKOVI, R0 ; dohvati početne adrese
ADD R0, R1, R0 ; računanje adrese znaka
LOADB R2, (R0) ; dohvati znak iz mem.
STORE R2, (PIOD) ; slanje znaka na PIO
>>>
```

## PIO - Primjeri

### Primjer:

Na prvi PIO na PIOD3-PIOD7 spojeno je 5 senzora (aktivna razina im je niska). Svaki puta kad se svih 5 senzora aktivira, FRISC treba bezuvjetno poslati procesu podatak iz bloka memorije s početnom adresom BLOK. Proses je spojen na drugi PIO. Prvi PIO radi u prekidnom načinu.

Nakon slanja 10 podataka, treba zabraniti daljnje generiranje prekida od strane PIO1 i nastaviti izvođenje glavnog programa.

### Rješenje:

Odaberimo adrese za PIO1 i PIO2: FFFF1000 i FFFF2000. PIO1 ćemo spojiti na INT2 i programirati da radi u načinu ispitivanja bitova, a PIO2 u načinu postavljanja bitova.

>>>

## PIO - Primjeri

&lt;&lt;&lt;

PIO1 generirat će prekid kad se **svi** senzori aktiviraju, tj. odabrat ćemo AND u ulaznoj upravljačkoj riječi. Aktivnu razinu postavljamo na 0. Maskom biramo ispitivanje samo viših 5 bitova.

```

PIOC1 `EQU OFFFF1000
PIOD1 `EQU OFFFF1004
PIOIACK1 `EQU OFFFF1008
PIOIEND1 `EQU OFFFF100C

PIOC2 `EQU OFFFF2000
PIOD2 `EQU OFFFF2004

`ORG 0
MOVE 10000, R7
JP GLAVNI

`ORG 8 ; prekidni vektor
DW 500

>>>

```

© Kovač, Basch, FER, Zagreb

214

## PIO - Primjeri

```

<<< ; inicijalizacija sklopa PIO2
MOVE %B 100, R0 ; 1=mod BIT
 ; 0=no INT
 ; 0=OCR
STORE R0, (PIOC2) ; pošalji u OCR

MOVE %B 11000000, SR ; dozvoli INT2

PETLJA JR PETLJA ; "koristan posao"

PODAT DW 0
BROJAC DW 0

BLOK DW 3, 1, 5, 7, 3, 9, 2, 6, 5, 4
>>>

```

© Kovač, Basch, FER, Zagreb

216

## PIO - Primjeri

```

<<< LOAD R0, (BROJAC) ; dohvati
SUB R0, 1, R0 ; i smanji
STORE R0, (BROJAC) ; brojač

JR_NZ JOS ; ima li još podataka

KRAJ ; zabrani prekide na PIO1
MOVE %B 100101, R0 ; 1=AND, 0=active
 ; 0=no mask, 1=mod BIT
 ; 0=no INT, 1=ICR
STORE R0, (PIOC1) ; pošalji u ICR

JOS POP R0 ; obnovi kontekst
MOVE R0, SR
POP R1
POP R0
STORE R0, (PIOIEND1) ; dojava kraja posluž.
RETI

>>>

```

© Kovač, Basch, FER, Zagreb

218

## PIO - Primjeri



&lt;&lt;&lt;

```

PIOC `EQU OFFFF0000
PIOD `EQU OFFFF0004
PIOACK `EQU OFFFF0008
PIOEND `EQU OFFFF000C

`ORG 0
; glavni program: inicijalizacija PIO
MOVE %B 001, R0 ; 0=mode IN
 ; 0=NOT INT
 ; 1=ICR
STORE R0, (PIOC) ; pošalji u ICR

MOVE PODATCI, R1 ; adresa podataka
MOVE 40, R4 ; brojač
>>>

```

© Kovač, Basch, FER, Zagreb

220

## PIO - Primjeri

&lt;&lt;&lt; ; glavni program

```

GLAVNI MOVE BLOK, R0 ; adresu podataka
 STORE R0, (PODAT) ; stavi u PODAT
 MOVE %D 10, R0 ; broj podataka
 STORE R0, (BROJAC) ; stavi u BROJAC

; inicijalizacija sklopa PIO1
MOVE %B 101111, R0 ; 1=AND, 0=aktivna razina
 ; 1=Mask follows
 ; 1=mod BIT, 1=INT
 ; 1=ICR
STORE R0, (PIOC1) ; pošalji u ICR
MOVE %B 11111000, R0 ; pripremi masku i
STORE R0, (PIOC1) ; pošalji je u MR

```

&gt;&gt;&gt;

© Kovač, Basch, FER, Zagreb

215

## PIO - Primjeri

&lt;&lt;&lt; ; prekidni potprogram

```

`ORG 500

PUSH R0 ; spremi kontekst
PUSH R1
MOVE SR, R0
PUSH R0

STORE R0, (PIOIACK1); dojavi prihvat na PIO1
LOAD R0, (PODAT) ; dohvati adresu podatka
LOAD R1, (R0) ; dohvati podatak
STORE R1, (PIOD2) ; šalji podatak na PIO2
ADD R0, 4, R0 ; pomakni adresu na
STORE R0, (PODAT) ; sljedeći podatak

```

&gt;&gt;&gt;

© Kovač, Basch, FER, Zagreb

217

## PIO - Primjeri



### Primjer:

Zadatak je napisati program koji će **uvjetno** prenijeti 40<sub>16</sub> podataka s mjerne vanjske jedinice spojene preko sklopa PIO. Primljene podatke treba upisati u memoriju od lokacije PODATCI (upisuju se bajtovi).

### Rješenje:

PIO ćemo programirati da radi u ulaznom načinu, jer će s mjernom jedinicom biti potrebna sinkronizacija (zadano je da je prijenos uvjetni). Pretpostavka je da merna jedinica ima linije za rukovanje kompatibilne sa READY i STROBE.

&gt;&gt;&gt;

© Kovač, Basch, FER, Zagreb

219

## PIO - Primjeri



&lt;&lt;&lt;

```

CEKAJ LOAD R0, (PIOC) ; učitaj spremnost
 AND R0, 1, R0
 JR_Z CEKAJ ; čekaj spremnost PIO-a

 STORE R0, (PIOACK) ; brisanje stanja
 LOAD R0, (PIOD) ; prijenos podatka
 STOREB R0, (R1)
 STORE R0, (PIOEND) ; kraj posluživanja

 ADD R1, 1, R1 ; pomakni adresu podatka
 SUB R4, 1, R4 ; umanji brojač podataka
 JR_NZ CEKAJ
 HALT

```

PODATCI `DS 40 ; prostor za podatke (40 bajtova)

© Kovač, Basch, FER, Zagreb

221



## Primjer:

FRISC pomoću dva sklopa PIO upravlja radom stroja. PIO1 je na adresi FFFF1000, a PIO2 na FFFF2000.

Na PIO 1 spojeni su izlazi iz senzora za temperaturu (PIO00), pritisak (PIO1D) i ulaz sirovina (PIO2D). Ako bilo koja od tih vrijednosti prijeđe dopuštenu razinu, senzor šalje jedinicu na PIO, a u suprotnom se šalje ništice. PIO 1 spojen je na INT1.

Na PIO 2 je spojena samo jedna sklopka (PIO00) koja se uključuje jedinicom, a isključuje ništicom. Sklopka uključuje i isključuje stroj, a početno je isključena.

Program na početku uključuje stroj (pretpostavka je da su sve mjerene veličine ispravne). Nakon toga se prate senzori i mora se uključiti stroj ako bilo koja od mjerjenih veličina poprimi nedopuštenu razinu. Kada se sve vrijednosti vrati u normalnu razinu, treba ponovno uključiti stroj. Ovo se beskonačno ponavlja.

© Kovač, Basch, FER, Zagreb

222



&lt;&lt;&lt;

```

PIO1C `EQU OFFFFF1000
PIO1D `EQU OFFFFF1004
PIO1IACK `EQU OFFFFF1008
PIO1IEND `EQU OFFFFF100C

PIO2C `EQU OFFFFF2000
PIO2D `EQU OFFFFF2004

`ORG 0
MOVE 10000, SP
JP GLAVNI

; prekidni vektor
`ORG 8
DW 2000
 >>>

```

© Kovač, Basch, FER, Zagreb

224



```

<<<
MOVE %B 10100000, SR ; omogući INT1

MOVE ON, R0
STORE R0, (PIO2D) ; uključi stroj
STORE R0, (STANJE) ; STANJE=uključen

PETLJA ...
; "koristan posao"

; stanje stroja:
OFF `EQU 0 ; dvije "konstante" ON i OFF
ON `EQU 1
STANJE DW OFF ; "varijabla" za trenutačno
; stanje stroja
 >>>

```

© Kovač, Basch, FER, Zagreb

226



```

<<<
; stroj je uključen i neka od
; mjerjenih veličina je neispravna
UKLJUCEN_JE ; => treba isključiti stroj

MOVE OFF, R0
STORE R0, (PIO2D) ; uključi stroj
STORE R0, (STANJE) ; STANJE=uključen

; nova kontrolna riječ za PIO1:
; sada treba čekati uvjet za ponovno
; uključenje (maska se ne mijenja)
MOVE %B 100111, R0 ; 1=AND, 0=AKT.RAZINA
; 0=MASK, 1=mode BIT
; 1=INT, 1=ICR
STORE R0, (PIO1C)

JR VAN ; povratak iz p.p.
 >>>

```

© Kovač, Basch, FER, Zagreb

228



## Rješenje:

PIO2 ćemo programirati da radi u načinu postavljanja bitova i na početku ćemo pomoći njega uključiti stroj, a kasnije ga po potrebi isključivati i uključivati.

PIO1 ćemo programirati da radi u načinu ispitivanja bitova i da generira prekid, ali na dva načina - ovisno treba li čekati uvjet da se stroj isključi ili treba čekati uvjet da se stroj ponovno uključi.

Da bi se stroj uključio, barem jedan senzor mora dati neispravnu razinu mjerene vrijednosti. To znači da programiramo aktivnu razinu 1 i funkciju OR.

Da bi se stroj ponovno uključio, svi senzori moraju davati ispravnu razinu mjereni vrijednosti. To znači da programiramo aktivnu razinu 0 i funkciju AND.

&gt;&gt;&gt;

© Kovač, Basch, FER, Zagreb

223



&lt;&lt;&lt; ; glavni program

```

GLAVNI ; inicijalizacija sklopa PIO 1
MOVE %B 011111, R0 ; 0=OR, 1=AKT.RAZINA
; 1=MASK, 1=mode BIT
; 1=INT, 1=ICR
STORE R0, (PIO1C)

; maska - ispituju se 3 najniža bita
MOVE %B 00000111, R0
STORE R0, (PIO1C)

; inicijalizacija sklopa PIO 2
MOVE %B 100, R0 ; 1=mode BIT, 0=NOT INT
; 0=OCR
STORE R0, (PIO2C)
 >>>

```

© Kovač, Basch, FER, Zagreb

225



&lt;&lt;&lt; ; prekidni potprogram

```

`ORG 2000
PUSH R0 ; pohrana konteksta
MOVE SR, R0
PUSH R0
STORE R0, (PIO1IACK) ; dojavi prihvat prekida
LOAD R0, (STANJE) ; ispitaj je li stroj
CMP R0, ON ; isključen ili uključen
JR_EQ UKLJUCEN_JE
JR ISKLJUCEN_JE

VAN ; dio za povratak iz p.p.
POP R0
MOVE R0, SR ; obnova konteksta
POP R0
STORE R0, (PIO1IEND) ; dojava kraja posluž.
RETI
 >>>

```

© Kovač, Basch, FER, Zagreb

227



<<< ; stroj je isključen i sve mjerene
; veličine su opet ispravne
ISKLJUCEN\_JE ; => treba uključiti stroj

```

MOVE ON, R0
STORE R0, (PIO2D) ; uključi stroj
STORE R0, (STANJE) ; STANJE=uključen

; nova kontrolna riječ za PIO1:
; sada treba čekati uvjet za ponovno
; uključenje (maska se ne mijenja)
MOVE %B 010111, R0 ; 0=OR, 1=AKT.RAZINA
; 0=MASK, 1=mode BIT
; 1=INT, 1=ICR
STORE R0, (PIO1C)

JR VAN ; povratak iz p.p.
 >>>

```

© Kovač, Basch, FER, Zagreb

229



- Izravni pristup memoriji ili DMA (Direct Memory Access) je **sklopovski** ulazno-izlazni prijenos
- Prijenos ne obavlja procesor, nego posebna DMA-jedinica (ili DMA-sklop, engl. DMA controller)
- Velika brzina prijenosa (podaci ne prolaze kroz procesor)
- Moguć je prijenos između memorije i UI jedinice ili neka druga kombinacija izvora i odredišta podataka

## Sklopovski UI prijenos - DMA

### DMA prijenos



- Osnovna ideja:

- Procesor inicijalizira DMA-sklop, kako bi ovaj znao koliko podataka treba prenijeti, gdje su izvor i odredište podataka itd.
- DMA-sklop preuzme upravljanje nad sabirnicom i obavlja prijenos
- Procesor je za vrijeme prijenosa neaktivan
- Procesor i DMA-sklop moraju se međusobno sinkronizirati kako bi u svakom trenutku samo jedan od njih upravljao sabirnicom: za to se koriste linijama BREQ i BACK
- Pod upravljanjem sabirnicom misli se nainiciranje sabirničkih transakcija čitanja i pisanja (npr. upravljanje adresnom sabirnicom, sa READ i WRITE itd.)

### DMA - Brzina prijenosa



- Približna usporedba bezuvjetnog prijenosa i DMA za N podataka (npr. iz VJ u memoriju):

| naredba | trajanje             |
|---------|----------------------|
| PETLJA  | LOAD R0, (VJ_READ) 2 |
|         | STORE R0, (R2) 2     |
|         | ADD R2, 4, R2 1      |
|         | SUB R3, 1, R3 1      |
| JR_NZ   | PETLJA 2             |

Ukupno programske: N \* 8 ciklusa

Ukupno FRISC-DMA: N \* 2 ciklusa (DMA čita podatak preko sabirnice (prije ciklus) i onda ga zapisuje (drugi ciklus))

Kod mnogih stvarnih VJ čest je slučaj da je DMA dio VJ pa DMA ne treba komunicirati sa VJ preko procesorske sabirnice te je za jedan podatak potreban samo jedan ciklus !!!

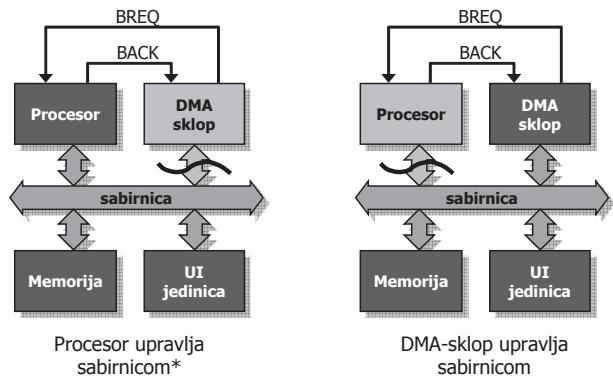
### DMA - Zaustavljanje procesora



- DMA zaustavljanjem procesora odvija se ovako:

- DMA-sklop preuzme upravljanje nad sabirnicom
- DMA-sklop prenese **sve podatke**
- Tek tada upravljanje nad sabirnicom se vraća procesoru koji je cijelo vrijeme bio zaustavljen
- Dok DMA-sklop upravlja sabirnicom, procesor ne može dohvaćati naredbe iz memorije i potpuno je neaktivan (jedino čeka dojavu od DMA da može nastaviti s radom)
- Nedostatak: procesor može dulje vrijeme biti neaktivan
- Prednost: najbrži prijenos

### DMA - Shema



\*Slika simbolično prikazuje stanje na sabirnici. DMA sklop ne upravlja sabirnicom, ali mu procesor može normalno pristupati kao i drugim UI jedinicama i memoriji

### DMA - Vrste prijenosa

- Vrste DMA prijenosa:

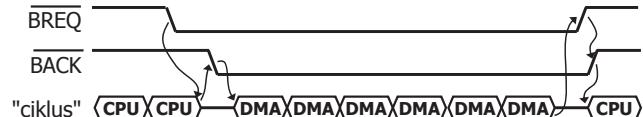
- Zaustavljanje procesora (engl. continuous, halting)
- Krađa ciklusa (engl. cycle stealing, word-at-a-time)
- Blokovski (engl. burst)
- Multiplexirani (engl. multiplexed)



### DMA - Zaustavljanje procesora



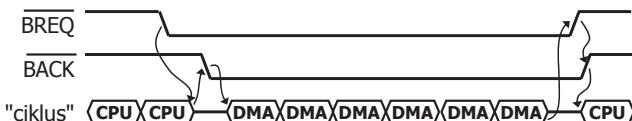
- Sinkronizacija za upravljanje sabirnicom (1):



- DMA-sklop aktiviranjem BREQ (Bus Request) zahtijeva sabirnicu od procesora
- Na kraju naredbe procesor prihvata zahtjev od DMA-sklopa:
  - postavlja priključke u visoku impedanciju
  - dojavljuje to DMA-sklopu aktiviranjem BACK (Bus Acknowledge)
  - BREQ je prioritetniji od svih zahtjeva za prekid



- Sinkronizacija za upravljanje sabirnicom (2):



- DMA-sklop na temelju padajućeg brida na BACK zna da je sabirnica slobodna i preuzima upravljanje nad njom
- DMA-sklop prenosi podatke
- Nakon prijenosa željenog broja podataka, DMA-sklop vraća upravljanje nad sabirnicom procesoru:
  - postavlja priključke u visoku impedanciju
  - deaktivira zahtijev za sabirnicom BREQ

## DMA - Krađa ciklusa



- DMA kradom ciklusa odvija se ovako:
  - DMA-sklop preuzeće upravljanje nad sabirnicom
  - DMA-sklop prenese **jedan podatak**
  - Upravljanje nad sabirnicom se vraća procesoru
  - Gornja tri koraka se ponavljaju dok se ne prenesu svi podatci
- Prednost: procesor se usporava, ali ipak izvodi glavni program
- Nedostatak: sporije od zaustavljanja procesora (dio vremena troši se na sinkronizaciju oko upravljanja sabirnicom)

## DMA - Blokovski prijenos



- DMA blokovskim prijenosom odvija se ovako:
  - DMA-sklop preuzeće upravljanje nad sabirnicom
  - DMA-sklop prenese **nekoliko podataka (tj. blok)**
  - Upravljanje nad sabirnicom se vraća procesoru
  - Gornja tri koraka se ponavljaju dok se ne prenesu svi podatci
- Kompromis između zaustavljanja procesora i krađe ciklusa
- Zaustavljanje procesora može se promatrati kao blokovski prijenos kod kojeg je veličina bloka jednaka svim podatcima
- Krađa ciklusa može se promatrati kao blokovski prijenos kod kojeg je u bloku samo jedan podatak

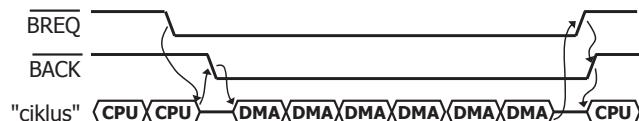
## DMA-sklopovi



- DMA-sklopovi mogu imati različite mogućnosti:
  - prijenos podataka između različitih izvora i odredišta
    - memorija → VJ
    - memorija → memorija (kopiranje bloka, inicijalizacija bloka)
    - VJ → memorija
    - VJ → VJ
  - traženje podataka u bloku ili kombinirani prijenos s traženjem
  - odabir načina prijenosa
  - dojava kraja prijenosa
    - postavljanje spremnosti
    - prekid
    - generiranje impulsa (mogu se prebrajati npr. CT-om)
  - itd.



- Sinkronizacija za upravljanje sabirnicom (3):

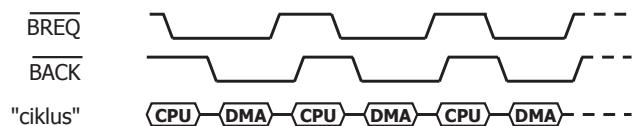


- Procesor je cijelo vrijeme bio neaktivan i osluškivao je priključak BREQ. Kad detektira na njemu rastući brid, zna da je sabirnica slobodna i preuzima upravljanje nad njom
  - Procesor deaktivira BACK jer je preuzeo upravljanje nad sabirnicom
  - Procesor nastavlja s izvođenjem naredaba

## DMA - Krađa ciklusa



- Sinkronizacija nad sabirnicom:



- Sinkronizacija se obavlja kao i za zaustavljanje procesora, ali se prenosi samo po jedan podatak
- Vrijeme ostavljeno procesoru prije ponovnog zahtijevanja sabirnice ovisi o konkretnom DMA-sklopu

## DMA - Multipleksirani prijenos



- Načelo multipleksiranog DMA prijenosa je da se za DMA-prijenos koriste trenutci kad procesor ne koristi sabirnicu
  - Korišteno na nekadašnjim procesorima kod kojih je nakon dohvata naredbe trebalo određeno vrijeme za izvođenje naredbe (za vrijeme izvođenja se nije komuniciralo s memorijom)
  - Kako su se brzine procesora povećavale u odnosu na brzinu memorije, tako je veza prema memoriji bila sve više zauzeta (Von Neumannovo usko grlo)
  - Kasnije, kad su se počele koristiti priručne memorije (cache), opet mogu postojati trenutci kad procesor ne komunicira s memorijom
  - DMA-sklop je najsloženiji jer mora raspoznavati trenutke neaktivnosti na sabirnicama, ali se prijenos obavlja bez usporenja rada procesora

## Sklop FRISC-DMA



## Sklop FRISC-DMA



- Sklop FRISC-DMA ima sljedeće mogućnosti:

- Prijenos 32-bitnih podataka
- Odabir vrste prijenosa:
  - zaustavljanje procesora
  - krađa ciklusa
- Zbog memorijskog preslikavanja, tj. jednakog pristupa memoriji i UI jedinicama, izvor i odredište mogu biti bilo koja kombinacija ovih komponenata
- Pristup UI jedinicama obavlja se bez provjere (bezuvjetno) što znači da UI jedinica mora biti sposobna primati/slati podatke brzinom DMA-prijenosu
- Dojava kraja prijenosa obavlja se postavljanjem spremnosti i prekidom

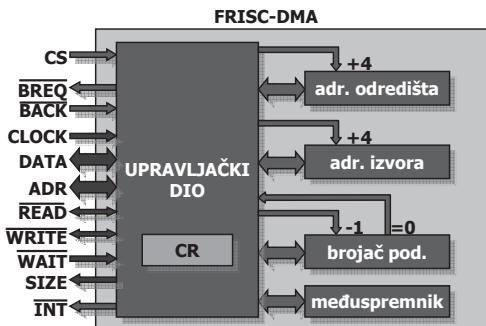
© Kovač, Basch, FER, Zagreb

246

## Blok-shema sklopa FRISC-DMA



- Priključci koji su inače ulazni za druge UI jedinice, ovdje su dvosmjerni (ADR, READ, WRITE) zato što
  - njima DMA-sklop upravlja kad obavlja prijenos
  - ulazni su kad FRISC upravlja sabirnicom (npr. inicijalizira DMA-sklop)

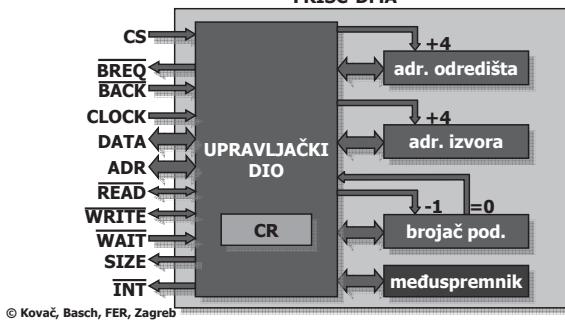


248

## Blok-shema sklopa FRISC-DMA



- Međuspremnik se koristi kad DMA-sklop obavlja prijenos
  - prvo se čita podatak iz izvora i sprem u međuspremnik
  - zatim se podatak iz međuspremnika sprem u odredište
  - (slična uloga kao registar DR u FRISC-u)

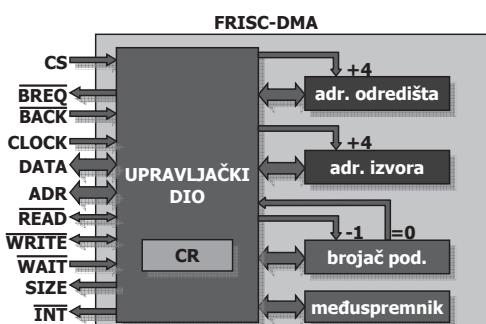


250

## Blok-shema sklopa FRISC-DMA



- Registri adrese služe za adresiranje izvora i odredišta podatka
  - Pone se tijekom inicijalizacije DMA-sklopa
  - Mogu se povećavati prilikom prijenosa svakog podatka (za 4)

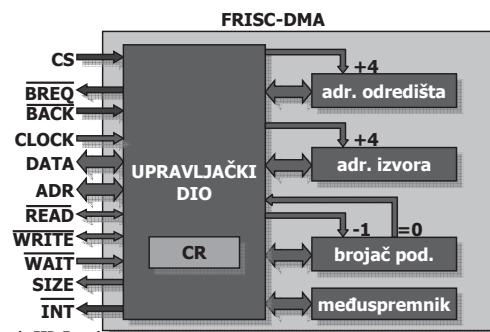


252

## Blok-shema sklopa FRISC-DMA



- Sučelje za spajanje s FRISC-om je specifično:
  - CS se koristi kad FRISC programira DMA-sklop
  - BREQ i BACK se koriste za sinkronizaciju sa FRISC-om
  - Pomoću INT se dojavljuje da su svi podaci preneseni



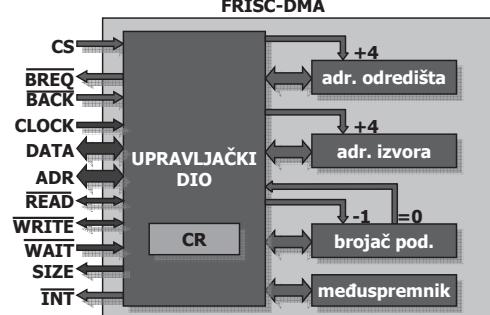
© Kovač, Basch, FER, Zagreb

247

## Blok-shema sklopa FRISC-DMA



- CLOCK se koristi zbog sinkronog sabirničkog protokola
- WAIT se koristi ako DMA-sklop upravlja sporom memorijom/VJ
- DATA se koristi kao dvosmjeren i kad FRISC upravlja s DMA-sklopom i kad DMA-sklop upravlja sabirnicom
- SIZE se koristi kad DMA-sklop upravlja sabirnicom



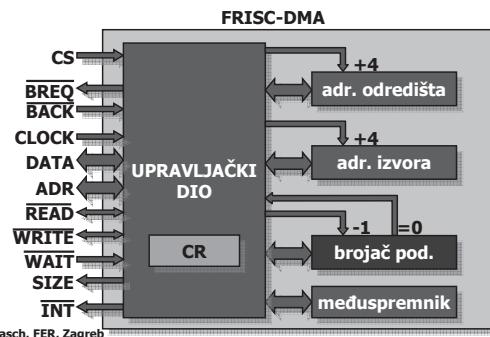
© Kovač, Basch, FER, Zagreb

249

## Blok-shema sklopa FRISC-DMA



- Brojač podataka kontrolira koliko je podataka preneseno
  - puni se tijekom inicijalizacije DMA-sklopa
  - smanjuje se nakon svakog prenesenog podatka
  - kad postane 0, onda je DMA-prijenos gotov



© Kovač, Basch, FER, Zagreb

251

## FRISC-DMA - adrese sklopa



- FRISC-DMA zauzima 6 uzastopnih 32-bitnih lokacija, počevši od početne adrese (PA) djeljive s 32:

| Adresa | Pisanje                                              | Čitanje                   |
|--------|------------------------------------------------------|---------------------------|
| PA     | upis adrese izvora                                   | čitanje adrese izvora     |
| PA+4   | upis adrese odredišta                                | čitanje adrese odredišta  |
| PA+8   | upis u brojač podataka                               | čitanje brojača podataka  |
| PA+12  | upis upravljačke riječi                              | čitanje bistabilna stanja |
| PA+16  | pokretanje DMA-prijenosu                             | -                         |
| PA+20  | brisanje stanja spremnosti (dojava prihvata prekida) | -                         |

- Prilikom pokretanja DMA-prijenosu i dojave prihvata prekida, poslani podatak se zanemaruje

© Kovač, Basch, FER, Zagreb

© Kovač, Basch, FER, Zagreb

253



- Upravljački registar ima sljedeće bitove:

- bit 0 regulira hoće li se ili neće postavljati prekid kad se postavi spremnost (tj. kad se prenesu svi podaci). Ima smisla samo za prijenos kradom ciklusa
- bitom 1 odabire se način prijenosa
- bitovima 2 i 3 odabire se je li izvor podataka (source) odnosno odredište podataka (destination) memorija ili VJ.

| 31-3                   | 3                      | 2                                                   | 1                                                     | 0   |
|------------------------|------------------------|-----------------------------------------------------|-------------------------------------------------------|-----|
| -                      | DESTINATION            | SOURCE                                              | MODE                                                  | INT |
| 0 - memorija<br>1 - VJ | 0 - memorija<br>1 - VJ | 0 - zaustavljanje<br>procesora<br>1 - krađa ciklusa | 0 - ne postavlja<br>prekid<br>1 - postavlja<br>prekid |     |
|                        |                        |                                                     |                                                       |     |

## Inicijalizacija sklopa FRISC-DMA



- Prilikom inicijalizacije treba zadati (u bilo kojem redoslijedu):
  - broj podataka
  - adresu izvora
  - adresu odredišta
  - upravljačku riječ
- Nakon toga treba pokrenuti DMA-prijenos upisom na adresu PA+16 (odmah nakon toga počinje DMA-prijenos)
  - ako je odabrano zaustavljanje procesora, dio programa iza naredbe za pokretanje DMA-prijenos neće se izvoditi sve dok se prijenos ne završi
  - ako je odabrana krađa ciklusa, dio programa iza naredbe za pokretanje se izvodi, ali usporeno

## DMA - Primjeri

```
<<<
DMA_SRC `EQU 0FFFF0000
DMA_DEST `EQU 0FFFF0004
DMA_SIZE `EQU 0FFFF0008
DMA_CTRL `EQU 0FFFF000C
DMA_START `EQU 0FFFF0010
DMA_ACK `EQU 0FFFF0014

INIT MOVE 10000, SP
;;;;; inicijalizacija DMA-sklopa
 MOVE 0FFFF3330, R0 ; upis adrese
 STORE R0, (DMA_SRC) ; izvora

 MOVE 4000, R0 ; upis adrese
 STORE R0, (DMA_DEST) ; odredišta

 MOVE %D 1000, R0 ; upis broja
 STORE R0, (DMA_SIZE) ; podataka >>>
```

## DMA - Primjeri

**Primjer:**

Treba riješiti prethodni zadatak, ali prijenos treba obaviti krađom ciklusa.

**Rješenje:**

Glavna razlika je u detekciji kraja DMA-prijenos, koja u prethodnom primjeru nije bila potrebna, jer se dio programa iza naredbe za pokretanje prijenosa izvodio tek nakon završetka prijenosa.

U ovom primjeru ćemo detekciju kraja napraviti pomoću ispitivanja stanja, a ne prekidom.

&gt;&gt;&gt;



- Ono što je zadano bitovima 2 i 3 ne mora odgovarati onom što je zaista spojeno na tim adresama
- Ovi bitovi određuju samo hoće li se nakon prijenosa svakog podatka uvećavati odgovarajući adresni registri ili neće
  - ako se u bit upiše 0 (memorija), onda se adresni registar povećava
  - ako se u bit upiše 1 (VJ), onda se adresni registar ne mijenja

| 31-3                   | 3                      | 2                                                   | 1                                                     | 0   |
|------------------------|------------------------|-----------------------------------------------------|-------------------------------------------------------|-----|
| -                      | DESTINATION            | SOURCE                                              | MODE                                                  | INT |
| 0 - memorija<br>1 - VJ | 0 - memorija<br>1 - VJ | 0 - zaustavljanje<br>procesora<br>1 - krađa ciklusa | 0 - ne postavlja<br>prekid<br>1 - postavlja<br>prekid |     |
|                        |                        |                                                     |                                                       |     |

## DMA - Primjeri

**Primjer:**

Treba prenijeti 1000<sub>10</sub> podataka iz VJ na adresi FFFF3330 u memoriju od adrese 4000. Adresa DMA-sklopa je FFFF0000. Prijie ili za vrijeme prijenosa može se izvoditi potprogram POTPR. Kad je prijenos gotov, treba pozvati potprogram GOTOVO te nastaviti s radom glavnog programa (prepostavka je da su ovi potprogrami već napisani i da postoje u memoriji).

Prijenos treba obaviti zaustavljanjem procesora.

**Rješenje:**

VJ zauzima jednu adresu, pa se adresni registar izvora ne treba povećavati tijekom prijenosa: zato u upravljačkoj riječi za SOURCE biramo vanjsku jedinicu

Za DESTINATION biramo memoriju, jer je odredište blok memorije pa će adresu trebati uvećavati tijekom prijenosa

&gt;&gt;&gt;

257

## DMA - Primjeri

```
<<< ; Upis upravljačke riječi:
; destin=mem, source=VJ, halt, no INT
MOVE %B 0100, R0
STORE R0, (DMA_CTRL)

; pokretanje DMA-prijenos
GLAVNI STORE R0, (DMA_START)

; dio programa koji će se izvesti tek nakon
; završenog DMA-prijenos
STORE R0, (DMA_ACK) ; brisanje statusa DMA
CALL POTPR
CALL GOTOVO
...
; nastavak glavnog programa
```

## DMA - Primjeri

```
<<<
DMA_SRC `EQU 0FFFF0000
DMA_DEST `EQU 0FFFF0004
DMA_SIZE `EQU 0FFFF0008
DMA_CTRL `EQU 0FFFF000C
DMA_START `EQU 0FFFF0010
DMA_ACK `EQU 0FFFF0014

INIT MOVE 10000, SP
;;;;; inicijalizacija DMA-sklopa
 MOVE 0FFFF3330, R0 ; upis adrese
 STORE R0, (DMA_SRC) ; izvora

 MOVE 4000, R0 ; upis adrese
 STORE R0, (DMA_DEST) ; odredišta

 MOVE %D 1000, R0 ; upis broja
 STORE R0, (DMA_SIZE) ; podataka
```

kao u  
prethodnom  
primjeru

>>>

```

<<< ; Upis upravljačke riječi:
 ; destin=mem, source=VJ, cycle st., no INT
MOVE %B 0110, R0
STORE R0, (DMA_CTRL)

 ; pokretanje DMA-prijenosu
GLAVNI STORE R0, (DMA_START)

; dio programa koji se izvodi usporeno i istodobno
; s DMA-prijenosom

 CALL POTPR

; dio programa koji se mora izvesti NAKON DMA-prijenosu
CEKAJ LOAD R0, (DMA_CTRL) ; učitaj spremnost DMA-sklopa
AND R0, 1, R0
JR_Z CEKAJ ; čekaj završetak prijenosa
STORE R0, (DMA_ACK) ; brisanje statusa DMA
CALL GOTODO
...
 ; nastavak glavnog programa

```

© Kovač, Basch, FER, Zagreb

262

## Izvedba sustava automobilske klime

Ovaj dokument namijenjen je isključivo za osobnu upotrebu studentima Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

Svaka druga upotreba ili umnožavanje bilo kojeg dijela ovog dokumenta nije dozvoljena bez pismene dozvole autora.

© Mario Kovač, Danko Basch



© Kovač, Basch, FER, Zagreb

## Auto-klima

- Upravljanje pomoću tri tipkala:

- uključi/isključi klima uređaj (on/off)
- smanji željenu temperaturu (-)
- povećaj željenu temperaturu (+)

- Prikaz pomoću LED-prikaznika:

- lijevo se prikazuje trenutačna (stvarna) temperatura vozila
- desno se prikazuje željena temperatura koju je korisnik namjestio (dok je klima isključena prikazuje se "--")



stvarna  
temperatura  
vozila      namještena  
(željena)  
temperatura

© Kovač, Basch, FER, Zagreb

## Mjerenje temperature

- Vanjska jedinica za mjerenje temperature

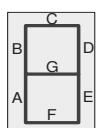
- Bazna adresa je 0FFFF0100
- Uvjetna vanjska jedinica
- Ima A/D konverter
- Rezolucija A/D konverzije je 1°C (unsigned int)
- Mjerenje u temperaturnoj jedinici pokreće se upisivanjem 1 na baznu adresu
- Rad temperaturne jedinice zastavlja se upisivanjem 0 na baznu adresu
- Čitanjem s bazne adrese dobiva se vrijednost zadnje obavljene A/D konverzije (tj. vrijednost zadnje izmjerene temperature)

© Kovač, Basch, FER, Zagreb

## 7-segmentni LED

- Za prikaz temperature koriste se četiri 7-segmentna LED prikaznika
- Dva se koriste za prikaz temperature u autu, dok se druga dva koriste za namještanje željene temperature
- Postoje dvije adresne linije kojima se vrši selektiranje jedne od četiri znamenke LED prikaznika
- Upravljanje 7-segmentnim prikaznikom
  - 7 LED dioda koje se neovisno mogu kontrolirati
  - Svaka led dioda ima svoj odvojeni signal za kontrolu
  - Postavljanjem određenog signala u visoko pali se LED dioda

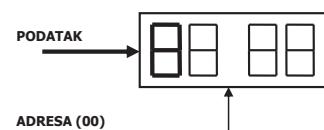
| ZNAK | BINARNO |   |   |   |   |   | HEX |    |
|------|---------|---|---|---|---|---|-----|----|
|      | G       | F | E | D | C | B | A   |    |
| 0    | 0       | 1 | 1 | 1 | 1 | 1 | 1   | 3F |
| 1    | 0       | 0 | 1 | 1 | 0 | 0 | 0   | 18 |
| 2    | 1       | 1 | 0 | 1 | 1 | 0 | 1   | 6D |
| 3    | 1       | 1 | 1 | 1 | 1 | 0 | 0   | 7C |
| 4    | 1       | 0 | 1 | 1 | 0 | 1 | 0   | 5A |
| 5    | 1       | 1 | 1 | 0 | 1 | 1 | 0   | 76 |
| 6    | 1       | 1 | 1 | 0 | 1 | 1 | 1   | 77 |
| 7    | 0       | 0 | 1 | 1 | 1 | 0 | 0   | 1C |
| 8    | 1       | 1 | 1 | 1 | 1 | 1 | 1   | 7F |
| 9    | 1       | 1 | 1 | 1 | 1 | 1 | 0   | 7E |
| -    | 1       | 0 | 0 | 0 | 0 | 0 | 0   | 40 |



4 © Kovač, Basch, FER, Zagreb

## 7-segmentni LED

- Isključi sve segmente od znamenke (stavi 0 na podatak)
- Postavi adresu znamenke
- Postavi podatak za znamenku



Zašto prvo isključujemo sve segmente (postavljamo nulu na podatak)?

Kad ne bi postavili prvo nulu na podatkovnu sabirnicu prilikom odabira nove znamenke na novoj znamenici bi se na trenutak prikazala vrijednost prijašnje znamenke što bi izazvalo treperenje segmenata.

© Kovač, Basch, FER, Zagreb

6 © Kovač, Basch, FER, Zagreb

3

5

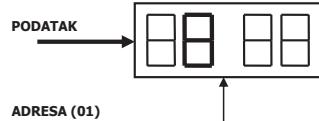
7

## 7-segmentni LED

- 1.a. Postavi podatak 0
- 1.b. Postavi adresu znamenke 0
- 1.c. Postavi podatak

10ms čekanja

- 2.a. Postavi podatak 0
- 2.b. Postavi adresu znamenke 1
- 2.c. Postavi podatak



## 7-segmentni LED

- 1.a. Postavi podatak 0
- 1.b. Postavi adresu znamenke 0
- 1.c. Postavi podatak

10ms čekanja

- 2.a. Postavi podatak 0
- 2.b. Postavi adresu znamenke 1
- 2.c. Postavi podatak

10ms čekanja

- 3.a. Postavi podatak 0
- 3.b. Postavi adresu znamenke 2
- 3.c. Postavi podatak



© Kovač, Basch, FER, Zagreb

8 © Kovač, Basch, FER, Zagreb

9

## 7-segmentni LED

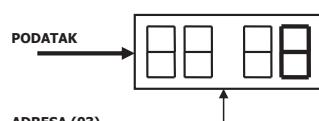
- 1.a. Postavi podatak 0
- 1.b. Postavi adresu znamenke 0
- 1.c. Postavi podatak

10ms čekanja

- 2.a. Postavi podatak 0
- 2.b. Postavi adresu znamenke 1
- 2.c. Postavi podatak

10ms čekanja

- 3.a. Postavi podatak 0
- 3.b. Postavi adresu znamenke 2
- 3.c. Postavi podatak



- 4.a. Postavi podatak 0
- 4.b. Postavi adresu znamenke 3
- 4.c. Postavi podatak

10ms čekanja

Idemo na početak (1.a)

10ms čekanja

© Kovač, Basch, FER, Zagreb

10 © Kovač, Basch, FER, Zagreb

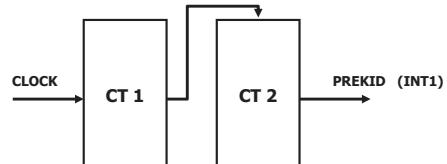
11

## • PIO-3 (0FFFF0400);

- Ispitivanje bitova, generira prekid (INT0), OR, aktivno visoko
- Na ulaze od PIOD0-PIOD2 spojena su tri tipkala
- Na ulaz PIOD3 spojen je senzor koji detektira radi li motor vozila ili ne.
  
- Tipkala prilikom aktiviranja daju na ulazima visoko stanje. Dok u neaktivnom stanju daju nisku razinu
  - PIOD0 – uvećava temperaturu (+)
  - PIOD1 – smanjuje temperaturu (-)
  - PIOD2 – pali/gasi klimu (ON/OFF)
  
- Senzor rada motora vozila postavlja ulaz PIOD3 u visoko ako motor vozila radi; ako se motor vozila ugasi, onda senzor postavlja nisku razinu.

## • CT1 (0FFFF1000) i CT2 (0FFFF1010)

- CT1 i CT2 spojeni su u lanac: na ulaz u CT1 je spojen signal vremenskog vođenja (10MHz), izlaz iz CT1 spojen na ulaz CT2, a CT2 generira prekid (INT1)
- Lanac CT-ova (tj. CT2) generira prekid svakih 10 ms
- Koristiti ćemo ga za ispis vrijednosti na 7-segmentne prikaznike
  - Svakih 10 ms ispisujemo podatak na jedan 7-segmentni prikaznik
  - Svake sekunde (dakle, svakih 100 prekida CT2-a) pročitamo temperaturu



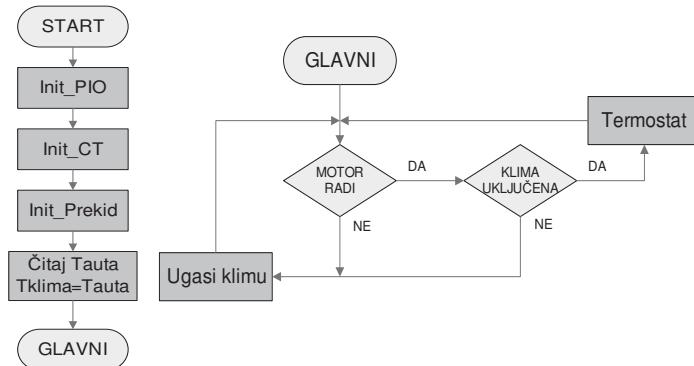
© Kovač, Basch, FER, Zagreb

12 © Kovač, Basch, FER, Zagreb

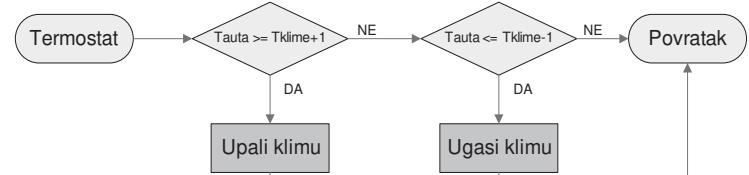
13

## • Glavni program

- Pretpostavlja se da računalo radi cijelo vrijeme (ima malu potrošnju i napajanje iz akumulatora)
- Ako se motor vozila isključi, onda treba automatski isključiti klimu da se ne potroši akumulator



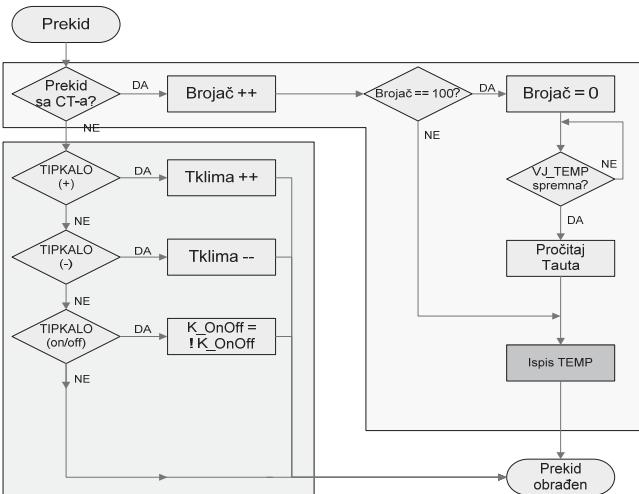
## • Potprogram Termostat



© Kovač, Basch, FER, Zagreb

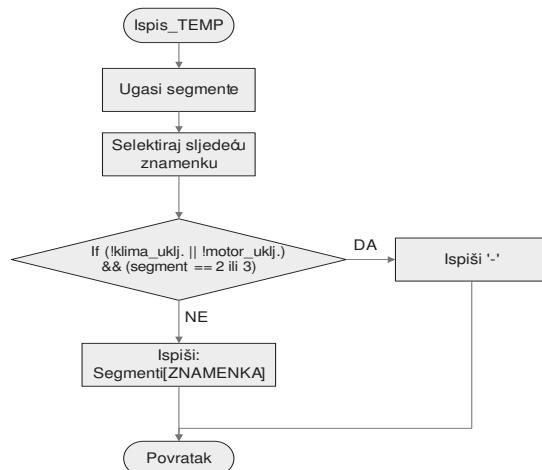
14 © Kovač, Basch, FER, Zagreb

15



© Kovač, Basch, FER, Zagreb

16 © Kovač, Basch, FER, Zagreb



17

## Dio ARM1: Arhitektura računala zasnovanog na procesoru ARM

### Procesor ARM: Uvod

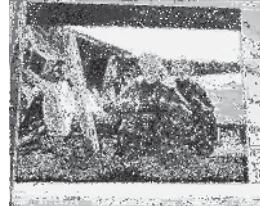
Ovaj dokument namijenjen je isključivo za osobnu upotrebu studentima Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

Svaka druga upotreba ili umnožavanje bilo kojeg dijela ovog dokumenta nije dozvoljena bez pismene dozvole autora.

© Mario Kovač, Danko Basch

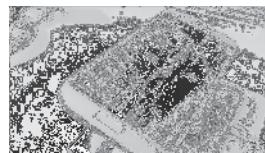
© Kovač, Basch, FER, Zagreb

1



### Primjer razvoja sustava

- Danas je to već moguće na većini pametnim telefonima
- Međutim brzina procesora nije jedini važan faktor u sustavu, već se mora gledati i propusna moć memorijskog sustava, tip aplikacije i slično...



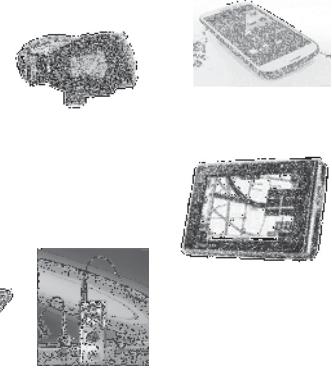
© Kovač, Basch, FER, Zagreb

3

© Kovač, Basch, FER, Zagreb

### Primjer razvoja sustava

- Danas već možemo gledati video na mnogim MALIM i SNAŽNIM uređajima i mobilnim telefonima.
- Arhitektura većine današnjih mobilnih telefona zasnovana je na procesoru ARM!!!
- Vidi npr:
  - <http://armdevices.net/>
  - <http://www.arm.com/markets/>



### ARM najvažnija tržišta

- Embedded
  - Automotive Infotainment
  - Embedded Computing
  - General-Purpose MCUs
  - Smart Cards
  - Smart Meter
- Mobile
  - Computing
  - Smartphones
  - Feature Phones
  - Connectivity and Modem
  - Mobile Payments
- Home
  - Blu-ray and DVD
  - Computing
  - Digital Set-top Box
  - Digital Still Cameras
  - Digital TV
  - Gaming
- Enterprise
  - HDD/SSD
  - Flash Cards and UFD
  - Home Networking

© Kovač, Basch, FER, Zagreb

5

© Kovač, Basch, FER, Zagreb

### Zahtjevi industrije i tržišta

- Procesorska snaga
- Cijena
- Niska potrošnja
- Sigurnost/Pouzdanost
- Prilagodljivost
- Modularnost
- Široka primijenjenost
- ...

Nažalost mnogi od ovih zahtjeva su dijametralno suprotni i nemoguće ih je sve zadovoljiti pa tražimo optimalno rješenje za pojedinu primjenu

4

© Kovač, Basch, FER, Zagreb

6

## Primjer...

- Životni ciklus nekih proizvoda u današnje vrijeme je izuzetno kratak... (posebice potrošačka elektronika)

- Novi proizvod se mora

- Projektirati
- Proizvesti prototip
- Ispitati
- Pripremiti proizvodnju
- Pripremiti tržište
- Proizvesti
- Distribuirati

- Ponekad za samo 3-6 mjeseci !!!

© Kovač, Basch, FER, Zagreb

7

## Stavljanje novog proizvoda na tržište

- Dokazano je da je jedan od najvažnijih faktora uspjeha što ranije izlazak na tržište sa kvalitetnim proizvodom
  - To se može postići isključivo minimizacijom potrebe za redizajniranjem (tj. pokušava se iskoristiti što više postojećeg)
  - Nove funkcionalnosti koje su potrebne možda se mogu KUPITI na tržištu (ako se procjeni da je to isplativije nego razvijati ih unutar kuće)
- Upravo na tom načelu se zasniva uspjeh procesora ARM na tržištu

© Kovač, Basch, FER, Zagreb

9

## Kratka povijest ARM-a

- Prva verzija procesora ARM razvijena je u *Acorn Computers Limited*, Cambridge, Engleska, između 1983. i 1985. godine.
- U to vrijeme dominaciju na tržištu imali su 8-bitni mikroprocesori sa relativno kompleksnim skupom instrukcija (CISC - Complex Instruction Set Computers).
- Samo nešto ranije, početkom 80-tih, u okviru tri gotovo usporedna istraživačka projekta (IBM 801, Berkeley RISC i Stanford MIPS) razvijena je potpuno nova arhitektura procesora koja se zasniva na jednostavnim instrukcijama koje se mogu izvoditi velikom brzinom (RISC -Reduced Instruction Set Computers).
- RISC procesor razvijen na sveučilištu Berkeley u to vrijeme pokazivao je izuzetne performanse u usporedbi sa komercijalnim procesorima uz znatno jednostavniju (i jeftiniju) sklopovsku izvedbu.

© Kovač, Basch, FER, Zagreb

11

## Advanced RISC Machines

- Kako je ova arhitektura postala izuzetno uspješna na tržištu (zbog relativno niske cijene i dobrih performansi) tako je i 1990. od dijela firme Acorn u kojoj je stvoren prvi procesor ARM formirana nova firma pod nazivom Advanced RISC Machines koja je preuzeila projekt širenja tržišta i daljnog razvoja ARM arhitekture.
- Od tada i skraćenica imena procesora (ARM) mijenja svoje značenje i postaje Advanced RISC Machine.

© Kovač, Basch, FER, Zagreb

## Stavljanje novog proizvoda na tržište

- U tih 3-6 mjeseci nemoguće je **uvijek iznova** projektirati uređaj
- Takav postupak bio bi
  - Skup
  - Spor
- U današnje vrijeme teži se sve većem "ponovnom" korištenju intelektualnog vlasništva (IP, Intellectual Property) što se obično naziva "reusability"
- Ideja je da dijelove sustava (sklopovske i programske) napravimo tako da su modularni kako bi ih mogli koristiti u što više proizvoda i u što više budućih inačica

© Kovač, Basch, FER, Zagreb

8

## Stvarni svijet - predavanja: Što ih povezuje

- 4/8/16/32/64-bitni CPU ?
  - Neki procesori su ZANIMLJIVI (npr. nova Intel Penryn mikroarhitektura)
  - Neki procesori su JEDNOSTAVNI (npr. Microchip PIC MCU)
  - Neki procesori se KORISTE
- Naš izbor je ARM (jer se KORISTI)
  - Ali "ARM" u stvari NIJE PROCESOR u onom obliku kako se obično zamišlja procesor već predstavlja ideju pristupa tržištu
  - Približno 70% tržišta 32-bitnih procesora !!!!!!
- Objasnimo što zapravo ARM radi na tržištu

© Kovač, Basch, FER, Zagreb

10

## Prvi ARM procesor

- U to vrijeme razviti procesor zahtjevalo je milionske investicije, stotine projektanata i dugo vrijeme razvoja što si je moglo priuštiti samo nekoliko najvećih firmi – a Acorn nije spadao u tu grupu
- Projektanti u Acornu vidjeli su u RISC tipu arhitekture mogućnost da stvore svoj procesor bez potrebe dugogodišnjeg razvoja te velikog broja projektanata i finansijskih sredstava
- Tako je nastao prvi procesor ARM, čije ime je bila skraćenica od Acorn RISC Machine.
- Prvi prototip procesora napravila je firma VLSI Technology Inc. i isporučila ga je Acornu u travnju 1985.

© Kovač, Basch, FER, Zagreb

12

## ARM danas

- Od tada do danas procesor ARM doživljava brojna poboljšanja i proširenja te su na tržištu prisutne razne generacije i varijante ARM arhitekture.
- Još jedna od specifičnosti i izuzetnih karakteristika firme ARM je u tome što je ona bila prva firma koja svoj cijeli poslovni model zasniva na licenciranju arhitekture svojih procesora bez da ima vlastite poluvodičke tvornice i bez da sama proizvodi procesore.
- Procesore zasnovane na ARM arhitekturi proizvode Intel, Motorola, ST, Philips, ... iako svi oni imaju i svoje vlastite procesore

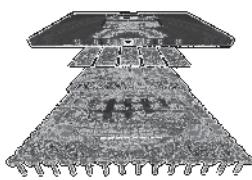
13

© Kovač, Basch, FER, Zagreb

14

## Što radi ARM

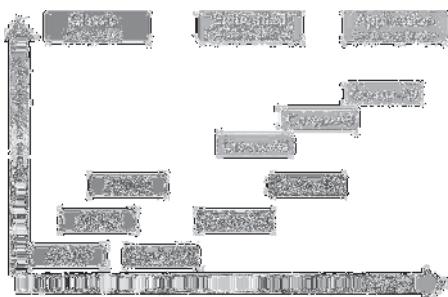
- Projektiranje ARHITEKTURA 32-bitnih procesora i sustava
- Licenciranje takvih sustava vodećim svjetskim elektroničkim firmama
- Razvoj i partnerstvo u razvoju ALATA i USLUGA namijenjenih razvoju ARM ARHITEKTURE
- Osnovna podloga su serije efikasnih 32-bitnih procesorskih jezgri koje rezultiraju u optimalnim procesorima u pogledu cijene, performansi i ostalih bitnih karakteristika



© Kovač, Basch, FER, Zagreb

15

## ARM porodice procesora



ARM7 :

- Na svijetu najčešća 32-bitovna arh. ugradbenih procesora
- više od 170 poluvodičkih tvrtki licenciralo tehnologiju
- preko 10 milijardi komada prodano od uvođenja u 1994.

© Kovač, Basch, FER, Zagreb

17

## Literatura i ostali izvori informacija o ARMu

- Knjiga
  - Osnove procesora ARM (drugo ispravljeno izdanje)
  - Oba izdanja su rasprodana ☺
- NOVO: Zbirka zadataka ARM (izašla JUČER!!)
- WEB
  - [www.arm.com](http://www.arm.com)
  - Sve o ARM procesorima ali na tisućama stranica ☺
  - Tehničku dokumentaciju možete naručiti na CD-u



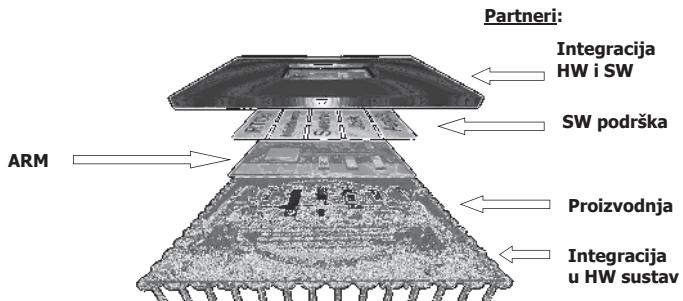
© Kovač, Basch, FER, Zagreb

19

## Programski alati (akademski)

- ARM simulator (u ATLAS-u)
  - Koristiti će se za laboratorijske vježbe
  - Simulacija rada procesora ARM i ostalih vanjskih sklopova
- ARM Development Suite (ADS) studentska verzija
  - ARM202u[1].zip
  - Nema vremenskog ograničenja
  - Nešto pojednostavljena, ali još uvjek izvrsna
  - Slobodna za korištenje uz poštivanje licence !!!
  - Dohvatljiva na WEB stranicama predmeta
- GNU Razvojni alati (ARM)

## ARM...

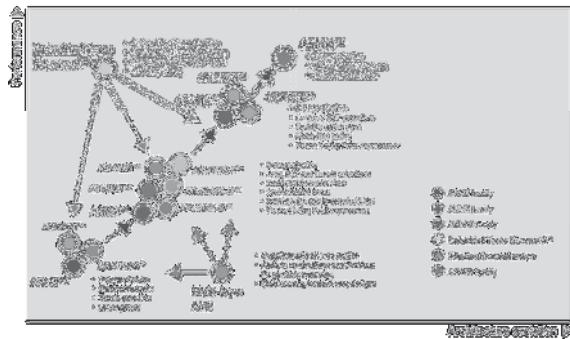


© Kovač, Basch, FER, Zagreb

16

## ARM procesorske jezgre

- Porodice i arhitekture
  - Nije baš jednostavno podudaranje
  - Na ovoj slici možete vidjeti razlike između pojedinih arhitektura

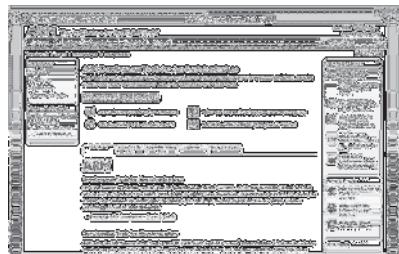


© Kovač, Basch, FER, Zagreb

18

## Programski alati (profesionalni)

- Na ARM web stranicama možete doći do informacija o mnogim profesionalnim razvojnim alatima
  - <http://www.arm.com/support/university/tools.php>



© Kovač, Basch, FER, Zagreb

20

## Dio ARM2: Programski model procesora ARM

Ovaj dokument namijenjen je isključivo za osobnu upotrebu studentima Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

Svaka druga upotreba ili umnožavanje bilo kojeg dijela ovog dokumenta nije dozvoljena bez pismene dozvole autora.

© Mario Kovač, Danko Basch

## Osnovne karakteristike arhitekture procesora ARM

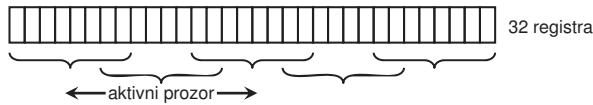
### • Vrsta arhitekture

- Procesor ARM po većini svojih karakteristika spada u grupu RISC procesora, ali ima i neke karakteristike CISC-a.

### • Skup registara

- relativno veliki, uniformni skup registara
- nema općenitih registarskih prozora\*

*i* \* primjer za 5 prozora sa po 8 registara i s preklapanjem od 2 registra:



© Kovač, Basch, FER, Zagreb

2

3

## Opće karakteristike

- Mogućnost proširenja skupa naredaba i registara korištenjem koprocesorskih naredaba
  - Neki kodovi nisu iskorišteni te ih možemo sami definirati
- Protočna struktura
  - Protočna struktura od tri (ARM7), pet (ARM9), šest (ARM10) ili osam (ARM11) razina
  - Visoka efikasnost
- Memorjska arhitektura
  - Von Neumannova ili Harvardska arhitektura (različito za pojedine jezgre)
- Modularna arhitektura
  - Namijenjena za SoC (System on Chip)
- Mala potrošnja

© Kovač, Basch, FER, Zagreb

4

5

## Programski model procesora ARM

- Programski model - skup značajki određene arhitekture koji je na raspolaganju programeru:

- tipovi podataka
- procesorski načini rada
- registri
  - registri opće namjene
  - registri programske stanje
- iznimke
- memorija

© Kovač, Basch, FER, Zagreb

6

7

## Procesorski načini rada

- ARM arhitektura podržava sedam procesorskih **načina rada** (engl. modes).
- Postoji jedan korisnički način rada (User) i šest privilegiranih načina rada
- U određenom procesorskom načinu omogućen je ili onemogućen pristup određenim resursima sustava
- Onemogućavanje pristupa nekim resursima u korisničkom načinu je od velike važnosti za izvedbu operacijskih sustava
- Procesorski način se definira postavljanjem najviših 5 bitova u posebnom registru CPSR što će kasnije biti objašnjeno

© Kovač, Basch, FER, Zagreb

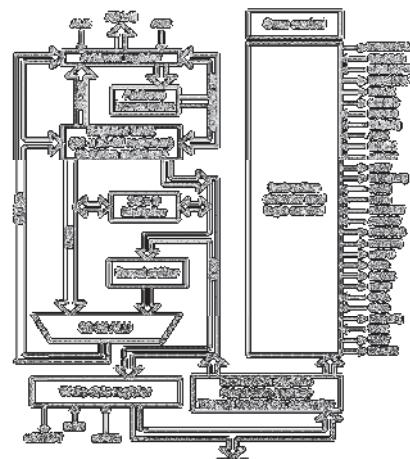
8

## Skup naredaba

- "Load-store" arhitektura
  - Prednosti/nedostaci kao što smo ranije objašnjavali
- Sve naredbe su iste duljine od 32 bita
  - Prednosti/nedostaci kao što smo ranije objašnjavali
- Naredbe za obradu podataka sa 3 adrese (3 operanda)
- Uvjetno izvođenje gotovo svake naredbe
  - Drugi procesori uglavnom nemaju ovu mogućnost
  - Služi za ubrzanje malih odsječaka programskog koda
- Izvođenje općenitog pomaka u aritmetičko-logičke naredbe u jednom vremenskom periodu
  - Barrel shifter na ulazu u ALU
- "Thumb" skup naredaba
  - Za sustave kojima je važna minimizacija memorije

© Kovač, Basch, FER, Zagreb

## ARM 7



© Kovač, Basch, FER, Zagreb

## Tipovi podataka

- Procesor ARM je 32-bitni procesor, ali može obrađivati i podatke manje širine. Programeru su na raspolaganju sljedeći tipovi podataka:
  - bajt (byte, 8 bitova), označava se slovom **B**
  - poluriječ (halfword, 16 bitova), označava se slovom **H**
  - riječ (word, 32 bita), označava se slovom **W**
- Iako ARM može koristiti kraće tipove podataka, sve osnovne operacije obavljaju se nad 32-bitnim riječima.

© Kovač, Basch, FER, Zagreb

## Procesorski načini rada

| Način          | Oznaka | Opis načina                                                       |
|----------------|--------|-------------------------------------------------------------------|
| User           | usr    | Normalno izvođenje programa                                       |
| System         | sys    | Podrška za izvođenje privilegiranih zadataka operacijskog sustava |
| Supervisor     | svc    | Zaštićen način za operacijski sustav                              |
| Abort          | abt    | Podrška za izvedbu virtualne memorije i zaštitu memorije          |
| Undefined      | und    | Podrška za programsku emulaciju koprocесora                       |
| Interrupt      | irq    | Podrška za obradu općenitog prekida                               |
| Fast Interrupt | fiq    | Podrška za brzi prekid                                            |

Privilegirani načini rada

© Kovač, Basch, FER, Zagreb

9

## "User" način

- Normalan način u kojem se izvode korisnički programi.
- U ovom načinu program ne može koristiti zaštićene resurse sustava te ne može promijeniti način rada.
- Pokušaj pisanja u CPSR[23:0] u User načinu rada procesor ignorira, tako da programi koji se izvode u User načinu ne mogu promijeniti način u neki od privilegiranih

## Ostali procesorski načini

- Ostali načini smatraju se privilegiranim i služe pri izvođenju specifičnih operacija. Programi koji se izvode u privilegiranim načinima imaju potpuni pristup svim resursima sustava.
- Načini fiq, irq, svc, abt i und aktiviraju se kad se u sustavu generira iznimka
- Način sys namijenjen je izvođenju privilegiranih sistemskih funkcija, ali bez potrebe korištenja alternativnih registara. Obrada nekih važnijih iznimaka bit će objašnjena kasnije.
- Programi pisani u ATLASU imat će pristup svim resursima sustava.

## Registri

- Procesor ARM ima ukupno 37 registara širine **32 bita**. Od ukupnog broja registara, postoje:
  - 31 registar opće namjene (uključujući i programsko brojilo).
    - Važno je uočiti da **u jednom trenutku možemo koristiti samo 16 registara** opće namjene (R0-R15) dok ostali nisu u to vrijeme dostupni !!!
  - 6 registara programskog stanja. Registri programskog stanja su također 32-bitni, no samo neki bitovi se koriste te ostali bitovi ne moraju nužno biti izvedeni u sklopoljvu.

## Registri

| User   | System  | Superv.  | Abort    | Undefined | Interrupt | Fast Int. |
|--------|---------|----------|----------|-----------|-----------|-----------|
| R0     |         |          |          |           |           |           |
| R1     |         |          |          |           |           |           |
| R2     |         |          |          |           |           |           |
| R3     |         |          |          |           |           |           |
| R4     |         |          |          |           |           |           |
| R5     |         |          |          |           |           |           |
| R6     |         |          |          |           |           |           |
| R7     |         |          |          |           |           |           |
| R8     |         |          |          |           | R8_fiq    |           |
| R9     |         |          |          |           | R9_fiq    |           |
| R10    |         |          |          |           | R10_fiq   |           |
| R11    |         |          |          |           | R11_fiq   |           |
| R12    |         |          |          |           | R12_fiq   |           |
| R13    | R13_svc | R13_abt  | R13_und  | R13_irq   | R13_fiq   |           |
| R14    | R14_svc | R14_abt  | R14_und  | R14_irq   | R14_fiq   |           |
| R15=PC |         |          |          |           |           |           |
| CPSR   |         | SPSR_svc | SPSR_abt | SPSR_und  | SPSR_irq  | SPSR_fiq  |

## Registri

- Registri opće namjene (R0-R15)
- Registri opće namjene, R0-R15 mogu se podijeliti u tri grupe:
  - Jednoznačno definirani registri R0-R7
  - Višezačno definirani registri R8-R14
  - Programsko brojilo R15
- Registri programskog stanja (CPSR, SPSR)
- NAPOMENA: u ATLAS-u su svi registri jednoznačni i procesor ima samo jedan način rada - System!!!**

vidi sliku na sljedećem slajdu >>>

## Jednoznačno definirani registri R0-R7

- Procesor ARM sadrži osam fizičkih registara nazvanih R0-R7.
- Registri R0-R7 su jednoznačno definirani, te će program pristupati istim fizičkim registrima bez obzira na to u kojem se načinu rada procesor nalazi.
- Ovi registri su, u punom značenju pojma, registri opće namjene jer za njih nije predviđeno nikakvo posebno značenje te se slobodno mogu koristiti u svim mogućim situacijama.
- Prema tome, za sve načine rada, registri R0- R7 su identični.

## Višezačno definirani registri R8-R14

- Za razliku od R0-R7, prilikom pristupa nekom od registara R8-R14 adresirat će se određeni fizički registar ovisno o tome u kojem načinu rada se procesor nalazi.
- Na primjer, ako se procesor nalazi u privilegiranom načinu Supervisor:
  - pri adresiranju registra R13 procesor će pristupiti posebnom fizičkom registru R13\_svc koji je dostupan samo u ovom procesorskom stanju. Isto vrijedi i za registar R14.
  - pri pristupanju registrima koji nisu višezačno definirani (R0-R12 i R15 za način Supervisor), uvjek se pristupa zajedničkim fizičkim registrima.

## Višeznačno definirani registri R8-R14

- Razlog za korištenje dodatnih fizičkih registara R13 i R14 je u tome što se registri R13 i R14 koriste za posebne namjene.
- Registr R13 se (po dogovoru) koristi kao pokazivač na vrh stoga (SP, Stack Pointer) te je ovime omogućeno da se u svakom načinu rada može definirati neovisan stog.
- Registr R14 se koristi kao registr za pohranjivanje povratne adrese za vraćanje iz potprograma ili iznimke.
- Oba ova registra se mogu koristiti i kao registri opće namjene ukoliko ih sustav ne koristi za ove posebne namjene.

© Kovač, Basch, FER, Zagreb

18

## Višeznačno definirani registri R8-R14

- U načinu brzog prekida (fiq) adresira se skup od 7 posebnih fizičkih registara (R8\_fiq-R14\_fiq).
- To omogućuje da se obrada brzog prekida izvede čim brže - bez potrebe za spremanjem konteksta.

© Kovač, Basch, FER, Zagreb

19

## Programsko brojilo R15

- Registr R15 je **programsko brojilo** te se u većini slučajeva korištenje ovog registra za opće namjene ne preporuča, a često i nije dozvoljeno.
- Čitanjem podatka zapisanog u R15 dobije se vrijednost adrese trenutne naredbe + 8 bajtova (zbog protočne strukture).
- Valja uočiti da su pri dohvatu naredbe najniža dva bita uvijek postavljena u logičku nulu zbog toga jer su naredbe ARM-a uvijek poravnate na širinu riječi.
- Upis vrijednosti u R15 izvodi neku vrstu forsiраног skoka, no ovaj način korištenja se ne preporučuje.

© Kovač, Basch, FER, Zagreb

20

## Registri programskog stanja (CPSR, SPSR)

- U registru trenutnog programskega stanja CPSR (Current Program Status Register) spremljeni su bitovi koji definiraju različita stanja procesora i programa.
- Registr CPSR je dostupan u svim procesorskim načinima.
- Registr pohranjenog programskega stanja SPSR (Saved Program Status Register) koristi se kad procesor obrađuje iznimke. U njega se pohranjuje vrijednost registra CPSR

© Kovač, Basch, FER, Zagreb

21

## Bitovi registara CPSR/SPSR

|    |    |    |    |    |      |   |   |   |   |   |   |   |   |
|----|----|----|----|----|------|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | .... | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| N  | Z  | C  | V  | Q  |      | I | F | T |   | M |   |   |   |

- Bitovi M4-M0: način rada procesora (npr. M[4:0]=10000 znači način "usr").
- Bit T označava da ARM izvodi Thumb naredbu.
- Postavljanje bita I ili F u 1 onemogućit će prekid (I) ili brzi prekid (F).
- Najviših pet bitova u CPSR predstavljaju zastavice:
  - N (Negative) - negativna vrijednost,
  - Z (Zero) - nula,
  - C (Carry) - prijenos (pažnja: kod oduzimanja C je PRIJENOS, a ne posudba),
  - V (Overflow) - preljev.
  - Q je zastavica koja označava da je došlo do preljeva i-ili zasićenja kod proširenih DSP naredaba (ARM procesori u E izvedbi).
- Ostali bitovi ne koriste se i sačuvani su za buduća proširenja.

© Kovač, Basch, FER, Zagreb

22

## Iznimke

- Tijekom rada procesor često treba obraditi neke događaje, poput obrade prekida ili obrade nepostojeće naredbe. U takvim situacijama sustav će generirati **iznimke**.
- Iznimke se općenito obrađuju ovako:
  - procesor pohranjuje povratnu adresu u registr R14
  - procesor pohranjuje trenutno programsko stanje (CPSR) u registr SPSR
  - procesor se prebacuje u potrebno privilegirano stanje
  - procesor započinje s izvođenjem potprograma za obradu iznimke
- Više o iznimkama u posebnom poglavljiju...

© Kovač, Basch, FER, Zagreb

23

## Zapis podataka u memoriji

- ARM arhitektura koristi jedinstveni memorijski adresni prostor od  $2^{32}$  8-bitnih podataka
- Podaci u memoriji organizirani su kao:
  - 8-bitni bajtovi (byte)
  - 16-bitne poluriječi (half-word)
  - 32-bitne riječi (word).

© Kovač, Basch, FER, Zagreb

24

## Zapis podataka u memoriji

- Pri normalnom pristupu, pretpostavlja se da su podaci poravnati na sljedeći način:
  - riječi (32 bita) su poravnate tako da počinju na adresi djeljivoj s 4 (npr. 0, 4, 8, ...)
  - poluriječi (16 bitova) su poravnate tako da počinju na parnoj adresi (npr. 0, 2, 4, 6, 8, ...)
  - bajtovi (8 bitova) su bilo gdje u memorijskom prostoru

© Kovač, Basch, FER, Zagreb

25

## Zapis podataka u memoriji

- Ako se podatak sastoji od nekoliko bajtova, tada je potrebno definirati kojim redoslijedom se ti bajtovi zapisuju u memoriju (endianness).
- U inicijalnom stanju, procesor ARM podržava NV (Niži pa Viši) zapis bajtova tako da se na nižu memoriju adresu zapisuje bajt manje važnosti (little-endian)\*

\* Kod procesora ARM moguće je odabrati i da zapisuje bajtove u redoslijedu VN (Viši pa Niži), tj. u redoslijedu big-endian

## Skup naredaba procesora ARM - ukratko

- Sve naredbe iz skupa naredaba procesora ARM mogu se podijeliti u šest osnovnih grupa:
  - Naredbe load i store
  - Naredbe za obradu podataka
  - Naredbe grananja
  - Naredbe za prijenos registara stanja
  - Koprocesorske naredbe
  - Naredbe za generiranje iznimke

## Naredbe load/store multiple, swap

- Postoje i naredbe za prijenos bloka podataka gdje se podaci spremaju ili uzimaju iz **više registara** (load/store multiple registers)
  - postoje razne modifikacije koje, na primjer, određuju kako su podaci smješteni u memoriji (povećanje/smanjenje pokazivača na podatak prije/poslije izvođenja naredbe).
  - efikasan način zapisivanja ili čitanja podataka sa stoga.
- U ovu grupu naredbi spadaju i naredbe za zamjenu sadržaja memorije i registara (swap) koje se najčešće koriste pri izvedbi semafora kod operacijskih sustava.

## Aritmetičko-logičke naredbe

- Imaju zajedničku karakteristiku da mogu obraditi do dva operanda i spremiti rezultat u zadani registar te, ako je zadano, osvježiti zastavice stanja.
- Od dva operanda koji se mogu koristiti u operaciji jedan uvijek mora biti registar, a drugi može biti ili neposredna vrijednost ili vrijednost registra nad kojom se još može obaviti određen pomak.
- U okvir aritmetičko-logičkih naredbi postoje i četiri naredbe za usporedbu koje su definirane vrlo slično aritmetičko-logičkim naredbama s razlikom da se rezultat nigdje ne spremi i da se uvijek osvježavaju zastavice stanja.

## Pregled skupa naredaba procesora ARM

### Naredbe load i store

- Programer ima na raspolaganju naredbe za prijenos podataka iz memorije u registar (load register, LDR) i iz registra u memoriju (store register, STR) u nekoliko različitih verzija.
- Najčešće korištene naredbe obavljaju prijenos samo jednog podatka koji može biti širok 8, 16 ili 32 bita
  - Može se odabrati predznačeno proširivanje bitova do pune 32-bitne riječi (pri prijenosu bajta ili poluriječi) ili proširivanje ništicama
- Postoji velik broj načina adresiranja

### Naredbe za obradu podataka

- Naredbe za obradu podataka obavljaju razne operacije nad podacima koji se nalaze u registrima.
- Naredbe za obradu podataka dijele se na:
  - aritmetičko-logičke naredbe
  - naredbe za množenje i
  - naredba za brojenje vodećih nula.

### Naredbe za množenje

- Množenje se uvijek obavlja nad dva 32-bitna podatka.
- Različite naredbe za množenje omogućuju da se spremaju samo 32 bita rezultata u izabrani registar ili da se spreme svih 64 bita rezultata u dva registra.
- U oba slučaja moguće je još omogućiti zbrajanje sa prethodnom vrijednosti rezultantnih registara, čime se ostvaruje tzv. operacija MAC (Multiply-And-Accumulate) koja se vrlo često koristi u algoritmima za obradu signala.

## Naredba clz

- Naredba CLZ (Count Leading Zeroes) služi za brojenje vodećih nula u podatku.
- Ovo je specifična naredba koja je veoma korisna kod izvedbe matematičkih algoritama (normiranje brojeva) i algoritama za kompresiju (npr. metode duljine niza).

© Kovač, Basch, FER, Zagreb

34

## Naredbe grananja

- Programsko brojilo kod procesora ARM je izvedeno kao običan registar te je grananja moguće izvesti bilo kojom naredbom load ili nekom aritmetičko/logičkom naredbom.
- Pored toga ARM definira i standardne naredbe za grananje poput:
  - Branch (B)
  - Branch and Link (BL).
- Posebnost procesora ARM je što neke izvedbe posjeduju takozvani 'Thumb' skup naredaba te postoje specijalne naredbe grananja BX kojima programer prebacuje procesor iz stanja dekodiranja standardnog skupa naredaba u 'Thumb' skup naredaba i obratno.

© Kovač, Basch, FER, Zagreb

35

## Naredbe za pristup registrima stanja

- Posebna grupa naredaba omogućuje prijenos podataka iz registara stanja (CPSR, SPSR) procesora ARM u neki registar opće namjene i obratno.
- Pisanjem u CPSR, na primjer, programer može postaviti stanja zastavica, stanja bitova za omogućavanje prekida kao i procesorska stanja.

© Kovač, Basch, FER, Zagreb

36

## Koprocesorske naredbe

- Služe za komunikaciju procesora ARM i koprocesora ako se oni nalaze u sustavu (prijenos podataka, operacije nad podacima, ...)

© Kovač, Basch, FER, Zagreb

37

## Naredbe za generiranje iznimke

- Omogućuju programeru i sistemu da se pokrene proces obrade iznimke
- Programski ekvivalent sklopovskim iznimkama koje se postavljaju na priključke procesora

© Kovač, Basch, FER, Zagreb

38

## Strojni kodovi ARM-a

- Procesor ARM ima strojne kodove fiksne širine 32 bita (osim za naredbe tzv. "Thumb" skupa)
- Ova fiksna duljina koda naredbe ima svoje prednosti i nedostatke.
  - Prednost je jednostavnost izvođenja i efikasnost protočne strukture za naredbe
  - Nedostatak je ograničenost broja bitova za opis pojedinih dijelova naredbe.
- Formati strojnih kodova dosta ovise o pojedinoj naredbi (vidi primjere na sljedećem slajdu)

© Kovač, Basch, FER, Zagreb

39

## Primjeri formata naredaba

| Naredba                               | 31    | 30 | 29 | 28 | 27     | 26 | 25 | 24 | 23 | 22           | 21    | 20              | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------------------------------------|-------|----|----|----|--------|----|----|----|----|--------------|-------|-----------------|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| Load/Store Multiple                   | Uvjet | 1  | 0  | 0  | P      | U  | S  | W  | L  | Rn           |       | Popis registara |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
| B, BL                                 | Uvjet | 1  | 0  | 1  | L      |    |    |    |    |              |       | 24-bitni odmak  |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
| Aritmetičko logička, neposredni pomak | Uvjet | 0  | 0  | 0  | Op kod | S  | Rn |    | Rd | Iznos pomaka | pomak | 0               | Rm |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |

↑  
Polje uvjeta

© Kovač, Basch, FER, Zagreb

40

## Polje uvjeta (condition field)

- Većina ARM-ovih naredaba **može se izvoditi uvjetno**. Te naredbe u strojnem kodu imaju **polje uvjeta** (condition field) u kojem se zadaje uvjet koji mora biti zadovoljen da bi se naredba izvela.
- Svi uvjeti temelje se na zastavicama stanja u CPSR.
- Ako uvjet nije zadovoljen, umjesto naredbe izvest će se NOP\*.
- Uvjetno izvođenje omogućuje programeru izvedbu programa bez korištenja naredbi grananja, čime se može ubrzati izvođenje nekog kratkog dijela programa

\* NOP je naredba koja ne izvodi ništa (kratica od No Operation)

© Kovač, Basch, FER, Zagreb

41

## Uvjeti

| Opcode [31:28] | Mnemonički naziv | Puni naziv (engleski)             | Stanje zastavica |
|----------------|------------------|-----------------------------------|------------------|
| 0000           | EQ               | Equal                             | Z                |
| 0001           | NE               | Not equal                         | !Z               |
| 0010           | CS/HS            | Carry set/unsigned higher or same | C                |
| 0011           | CC/LO            | Carry clear/unsigned lower        | !C               |
| 0100           | MI               | Minus/negative                    | N                |
| 0101           | PL               | Plus/positive or zero             | IN               |
| 0110           | VS               | Overflow                          | V                |
| 0111           | VC               | No overflow                       | IV               |
| 1000           | HI               | Unsigned higher                   | C and !Z         |
| 1001           | LS               | Unsigned lower or same            | !C or Z          |
| 1010           | GE               | Signed greater than or equal      | N == V           |
| 1011           | LT               | Signed less than                  | N != V           |
| 1100           | GT               | Signed greater than               | !Z and N == V    |
| 1101           | LE               | Signed less than or equal         | Z or N != V      |
| 1110           | AL               | Always (unconditional)            | -                |
| 1111           | (NV)             | See Condition code 0b1111         | -                |

© Kovač, Basch, FER, Zagreb

42

## Prvi program ☺ (verzija za ADS)

; Program za 32-bitno zbrajanje, verzija 1

AREA add\_32\_v1, CODE, READWRITE ; definicije asembleru

ENTRY ; oznaka početka programskega odsječka  
; (ulazna točka)

|                                                                       |
|-----------------------------------------------------------------------|
| LDR r0, A ; prvi operand se učitava u r0                              |
| LDR r1, B ; drugi operand se učitava u r1                             |
| ADDS r4, r0, r1 ; obavlja se zbrajanje i postavljaju se zastavice (S) |
| STR r4, REZ ; rezultat se spremi u memoriju                           |

kraj

|                                                 |
|-------------------------------------------------|
| MOV r0, #0x18 ; angel_SWIreason_ReportException |
| LDR r1, =0x20026 ; ADP_Stopped_ApplicationExit  |
| SWI 0x123456 ; ARM semihosting SWI              |

END  
© Kovač, Basch, FER, Zagreb

44

## Neke osnovne razlike ATLAS – ARM RVDS (1)

|                                                                  | ATLAS                                                                                                                               | ARM RVDS                                                    |
|------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------|
| Prepostavljena (default) baza za pisanje brojeva                 | Heksadekadika (osim za zadavanje iznosa pomaka/rotacija gdje se koristi <b>dekadika</b> baza, npr.: MOV R0,#1A<8, ADD R0,R1,LSL#17) | Dekadska                                                    |
| Pisanje heksadekadskog broja                                     | %H                                                                                                                                  | 0x ili &                                                    |
| Pisanje dekadskog broja                                          | %D                                                                                                                                  | Izravno se piše broj                                        |
| Pisanje binarnog broja                                           | %B                                                                                                                                  | 2_                                                          |
| Pisanje broja u bilo kojoj bazi <b>N</b>                         | Nije moguće                                                                                                                         | N_                                                          |
| Definicija labele podataka <b>prije</b> poziva pseudonaredbe ADR | Nije dozvoljeno                                                                                                                     | Dozvoljeno                                                  |
| Pseudonaredbe za definiranje sadržaja memorije                   | Bajt: `DW, DB<br>Poluriječ: DH<br>Riječ: DW<br>Prostor: `DS                                                                         | Bajt: DCB<br>Poluriječ: DCW<br>Riječ: DCD<br>Prostor: SPACE |

© Kovač, Basch, FER, Zagreb

46

## Način pisanja programa

- Na međuispitima možete koristiti način pisanja programa za ATLAS ili ADS - kako god želite, ali u istom zadatku ne smijete mijenjati stil
- Bez obzira na način pisanja sam program je uvijek isti
- U primjerima na predavanju i u knjizi većinom se koristi način pisanja za ADS

© Kovač, Basch, FER, Zagreb

48

## Prvi program ☺ (verzija za ATLAS)

Napisati program za zbrajanje dva 32-bitna broja koji su zapisani u memoriji odmah iza programa. Rezultat treba spremi u memoriju iza operanada.

;Program za 32-bitno zbrajanje, verzija 1

`ORG 0

|                                                                       |
|-----------------------------------------------------------------------|
| LDR r0, A ; prvi operand se učitava u r0                              |
| LDR r1, B ; drugi operand se učitava u r1                             |
| ADDS r4, r0, r1 ; obavlja se zbrajanje i postavljaju se zastavice (S) |
| STR r4, REZ ; rezultat se spremi u memoriju                           |

kraj SWI 123456 ; ARM semihosting SWI

|                           |
|---------------------------|
| A DW %D100 ; prvi operand |
| B DW 20 ; drugi operand   |
| REZ `DS 4 ; rezultat      |

`END

© Kovač, Basch, FER, Zagreb

43

## Postupak prevodenja za ADS



- Izvorni kod se nalazi u datoteci: add\_32\_v1.s
- Postupak prevodenja za ADS

armasm -g add\_32\_v1.s -o test.o

- opcijom -g uključujemo debug-tablicu tako da možemo u ADS-u pregledavati izvorni kod za vrijeme simulacije
- opcijom -o definiramo ime izlazne datoteke

armlink test.o -o test.axf

- test.axf sada možemo učitati u ADS i obaviti simulaciju

© Kovač, Basch, FER, Zagreb

45

## Neke osnovne razlike ATLAS – ARM RVDS ()

|                                                               | ATLAS                                                                                                  | ARM RVDS                                                                                               |
|---------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| Pisanje neposredne vrijednosti u naredbama za obradu podataka | baza<rotacija_ulijevo npr. 3A<8 je zapis heksadekadskog broja 3A00                                     | Piše se željeni broj, a asembler automatski provjerava je li u dozvoljenom opsegu, npr. piše se 0x3A00 |
| Definiranje niza registara u naredbama LDM i STM              | U popisu registri moraju biti razdvojeni zarezima i navedeni u rastućem redoslijedu, npr. {R2,R4,R15}. | Sve kombinacije su dozvoljene.                                                                         |
| Zaustavljanje procesora                                       | HALT ili                                                                                               | Dodatno, oblik {Rx-Ry} nije dozvoljen, izuzetak je navođenje {R0-R15}                                  |
|                                                               | SWI 123456                                                                                             | MOV r0, #0x18<br>LDR r1, =0x20026<br>SWI 0x123456                                                      |

© Kovač, Basch, FER, Zagreb

47

## Dio ARM3: Skup naredaba procesora ARM

Detaljan pregled svih naredbi

Ovaj dokument namijenjen je isključivo za osobnu upotrebu studentima Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

Svaka druga upotreba ili umnožavanje bilo kojeg dijela ovog dokumenta nije dozvoljena bez pisменe dozvole autora.

© Mario Kovač, Danko Basch



© Kovač, Basch, FER, Zagreb

1

## Naredbe load i store

- Među najvažnijim naredbama za svaki procesor, pa tako i ARM, su naredbe za prijenos podataka između procesora i memorije (naredbe load/store).
- Naredbe Load čitaju podatak iz memorije i spremaju ga u registar dok naredbe Store spremaju podatak iz registra u memoriju.
- Ovo su jedine naredbe ARM-a kojima se može pristupiti podatku iz memorije !!!

© Kovač, Basch, FER, Zagreb

2

## Naredbe load i store

- Procesor ARM posjeduje dva osnovna tipa naredaba load/store:
  - **Prvi tip** naredaba može učitati ili upisati 32-bitnu riječ ili bajt bez predznaka.
  - Takve naredbe imaju općeniti format: $LDR|STR\{<cond>\}\{B\} Rd, <addressing\_mode>$
- gdje pojedine oznake znače:
  - $\{<cond>\}$  polje uvjeta
  - $\{B\}$  adresiranje bajta
  - $Rd$  odredišni register
  - $<addressing\_mode>$  opis načina adresiranja

© Kovač, Basch, FER, Zagreb

3

## Load / store

- **Drući tip** može učitati ili upisati 16-bitnu poluriječ bez predznaka, a također se može učitati poluriječ ili bajt te ih predznačeno proširiti do širine riječi.
- Ovakve naredbe imaju općeniti format: $LDR|STR\{<cond>\}H|SH|SB Rd, <addressing\_mode>$
- gdje pojedine oznake znače:
  - $\{<cond>\}$  polje uvjeta
  - H|SH|SB adresiranje poluriječi (H), predznačene poluriječi (SH) ili predznačenog bajta (SB)
  - $Rd$  odredišni register
  - $<addressing\_mode>$  opis načina adresiranja

© Kovač, Basch, FER, Zagreb

4

## Load / store

| Ime naredbe | Engleski naziv                       |
|-------------|--------------------------------------|
| LDR         | Load Word                            |
| LDRB        | Load Unsigned Byte (zero extend)     |
| LDRH        | Load Unsigned Halfword (zero extend) |
| LDRSB       | Load Signed Byte (sign extend)       |
| LDRSH       | Load Signed Halfword (sign extend)   |
| STR         | Store Word                           |
| STRB        | Store Byte                           |
| STRH        | Store Halfword                       |

© Kovač, Basch, FER, Zagreb

5

## Load/store: Bazni register

- Za sve naredbe load/store se adresa memorije izračunava korištenjem dva dijela:
  - vrijednost u nekom baznom registru
  - odmak (offset) u odnosu na vrijednost u baznom registru
- Bazni register može biti bilo koji register opće namjene\*

\* Ako se kao bazni register izabere PC, tada se može postići relativno adresiranje u odnosu na trenutačnu poziciju u programu, te na taj način i izvedba programa koji su potpuno neovisni o položaju u memoriji. Kod direktnog kodiranja (bez pomoći asemblera i korištenja labela) programer mora paziti na vrijednost PC-a u trenutku izvođenja naredbe

© Kovač, Basch, FER, Zagreb

6

## Load/store: Odmak (offset)

- Za pojedinu naredbu programer može izabrati jedan od tri formata odmaka:
  - U najjednostavnijem formatu odmak se definira kao **broj, odnosno neposredna vrijednost** (immediate) koji se izravno upisuje u kôd naredbe. Neposredna vrijednost zapisana je jednim bitom predznaka i iznosom odmaka od 12 ili 8 bitova (ovisno o naredbi). 12-bitnim odmakom se može adresirati memoristička lokacija odmaknuta za +/- 4095 mjesto, a 8-bitnim odmaknuta za +/- 255 mesta u odnosu na vrijednost izabranog baznog registra
  - Odmak se može definirati i pomoću **vrijednosti iz registra opće namjene\***
  - Treća mogućnost je definiranje odmaka pomoću **vrijednosti registra opće namjene\* koja je još pomaknuta ulijevo ili udesno**

\* osim registra PC

© Kovač, Basch, FER, Zagreb

7

## Load/store...

- Osim prethodno opisana tri načina za definiciju odmaka, procesor ARM omogućuje još tri različite inačice adresiranja memorije.
- Ove inačice zadaju treba li i kako mijenjati bazni register tijekom izvođenja naredbe load/store. Ove tri inačice su:
  - **Osnovni odmak**
    - Adresa se izračuna zbrajanjem ili oduzimanjem odmaka od baznog registra, a zatim se pristupi memoriji. Vrijednost baznog registra ostaje nepromijenjena: $Reg \leftrightarrow mem[BazniReg + Odmak]$

© Kovač, Basch, FER, Zagreb

8

## Load /store

- **Predindeksiranje**
  - Odmak se zbroji ili oduzme od baznog registra, a izračunata vrijednost se spremi natrag u bazni register. Zatim se pristupi memoriji. $BazniReg = BazniReg + Odmak$  $Reg \leftrightarrow mem[BazniReg]$
- **Postindeksiranje**
  - Memoriji se pristupi samo na temelju vrijednosti baznog registra. Tek nakon obavljenog prijenosa, odmak se zbroji ili oduzme od baznog registra, a izračunata vrijednost se spremi natrag u bazni register. $Reg \leftrightarrow mem[BazniReg]$  $BazniReg = BazniReg + Odmak$

© Kovač, Basch, FER, Zagreb

9



- Osnovni odmak:  
Reg  $\leftrightarrow$  mem[BazniReg + Odmak]
- Predindeksiranje:  
BazniReg = BazniReg + Odmak  
Reg  $\leftrightarrow$  mem[BazniReg]
- Postindeksiranje:  
Reg  $\leftrightarrow$  mem[BazniReg]  
BazniReg = BazniReg + Odmak

|                       | Osnovni odmak    | Predindeksiranje | Postindeksiranje |
|-----------------------|------------------|------------------|------------------|
| Neposredni            | [R0, #4]         | [R0, #4]!        | [R0], #4         |
| Registarski           | [R7, -R3]        | [R7, R3]!        | [R7], R3         |
| Registarski s pomakom | [R3, R5, LSL #2] | [R3,R5,LSL #2]!  | [R3],R5,LSL #2   |

### Primjeri naredaba load/store

```
LDR R1, [R0] ; Učitaj riječ u R1 s adrese R0
LDR R8, [R3, #4] ; Učitaj riječ u R8 s adrese R3+4
LDR R12,[R13, #-4] ; Učitaj riječ u R12 s adrese R13-4
STR R2, [R1, #0x100] ; Spremi R2 na adresu R1+0x100

LDRB R3, [R8, #3] ; Učitaj u najniži bajt od R3
 ; bajt s adrese R8+3 (obriši gornja 3 bajta u R3)
LDR R11, [R1, R2] ; Učitaj riječ u R11 s adrese R1+R2
STRB R10, [R7, -R4] ; Spremi najniži bajt iz R10 na adresu R7-R4

LDR R11, [R3,R5,LSL #2] ; Učitaj riječ u R11 s adrese R3+(R5*4)

LDR R1, [R0, #4]! ; Učitaj riječ u R1 s adrese R0+4, zatim R0 = R0+4
STRB R7, [R6, R1]! ; Spremi najniži bajt iz R7 na adresu R6+R1,
 ; zatim R6 = R6+R1
```

### Primjeri naredaba load/store

```
LDR R3, [R9], #4 ; Učitaj riječ u R3 s adrese R9, zatim R9 = R9+4
STR R2, [R5], #8 ; Spremi R2 na adresu R5, zatim R5 = R5+8

LDR R0, [PC, #40] ; Učitaj riječ u R0 s adrese PC+40
 ; PC = adresa naredbe LDR + 8
LDR R0, [R1], R2 ; Učitaj riječ u R0 s adrese R1, zatim R1 = R1+R2

LDRH R1, [R0] ; Učitaj u nižu poluriječ od R1 poluriječ s adrese R0 (obriši
 ; gornju poluriječ od R1)
STRH R2, [R1, #0x80] ; Spremi nižu poluriječ iz R2 na adresu R1+0x80

LDRSH R5, [R9] ; Učitaj u nižu poluriječ od R5 poluriječ s adrese R9
 ; (predznačeno proširi nižu poluriječ u R5)
LDRSB R3, [R8, #3] ; Učitaj u niži bajt od R3 bajt s adrese R8+3
 ; (predznačeno proširi najniži bajt u R3)

LDR R0, LABEL ; Učitaj riječ u R0 s adrese zadane labelom LABEL
 ; (asemblerski prevoditelj automatski pretvara ovaj zapis
 ; u adresiranje s registrom PC kao baznim registrom)
```

### Primjer (add\_16s\_v1)

```
; ****
; Primjer 3.3. 16-bitno zbrajanje predznačenih brojeva REZ=A+B (add_16s_v1)
; ****

AREA add_16s_v1, CODE, READWRITE ; definicije asemblera
ENTRY ; oznaka početka programskog odsječka

LDRSH R0, A ; prvi operand se učitava u R0 i on je proširen s predznakom na 32 bita
 ; (da smo željeli unsigned, koristili bi LDRH)
LDRSH R1, B ; drugi operand se učitava u R1 i on je proširen s predznakom na 32 bita
 ; (da smo željeli unsigned, koristili bi LDRH)
ADDS R4, R0, R1 ; izvodi se zbrajanje i postavljaju se zastavice (S)
STR R4, REZ ; rezultat se spremu u memoriju
kraj ; priprema za izlaz iz programa
MOV R0, #0x18 ; angel_SWIReason_ReportException
LDR R1, =0x20026 ; ADP_Stopped_ApplicationExit
SWI 0x123456 ; izvođenje SWI i izlaz iz programa

A DCW -100 ; prvi operand (poluriječ)
B DCW 0x20 ; drugi operand (poluriječ)
REZ SPACE 4 ; rezultat (rijec)

END
```

### Strojni kod prethodnog programa

```
00008000 [0xe1df01f4] ldrsh r0,[pc,#0x14]
00008004 [0xe1df11f2] ldrsh r1,[pc,#0x12]
00008008 [0xe0904001] add r4,r0,r1
0000800c [0xe58f400c] str r4,[pc, #0x10]
kraj [0xe3a00018] mov r0,#0x18
00008014 [0xe59f1008] ldr r1,0x00008024 ; = #0x00020026
00008018 [0xef123456] swi 0x123456
A (0000801C) [0xf9fc] dcw 0xffff .. .
B (0000801E) [0x0020] dcw 0x0020 .. .
REZ (00008020) [0x00000000] dcd 0x00000000
```

Kako asembler pretvoriti adresu labele A u [pc,#0x14] (u naredbi ldrsh r0,A)?

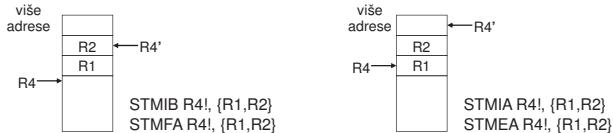
- Naredba je na adresi 0x8000
- Labela A je na adresi 0x801C
- Ako se uračuna PC=PC+8 kod izvođenja, tada je adresa (A) = PC + 0x14

### Naredbe Load Multiple i Store Multiple

- Pored osnovnih naredaba load i store, koje obavljaju prijenos podataka samo iz jednog registra, ARM ima dvije naredbe (Load Multiple i Store Multiple) koje programeru omogućuju da jednom naredbom obavi prijenos podataka između memorije i bilo kojeg podskupa registara ili čak svih registara.
- Osnovno ime naredbe je LDM i STM nakon čega se stavlja neki od nastavaka kojima se opisuje kako se spremi niz podataka (tj. niz registara) u memoriju

### LDM/STM: načini adresiranja

- Pri pisanju (čitanju) više podataka u memoriju mora se definirati gdje će biti zapisan prvi te svaki sljedeći podatak.
- Da bi definirali gdje će se zapisati prvi podatak, moramo izabrati neki registar opće namjene koji pokazuje na početak memorijskog područja u koje će se upisivati podaci.
- Nakon toga moramo izabrati jednu od četiri kombinacije načina adresiranja niza: IB, IA, DB ili DA. Ove kratice znače:
  - IB - Increment Before (uvećaj prije)
  - IA - Increment After (uvećaj poslije)
  - DB - Decrement Before (smanji prije)
  - DA - Decrement After (smanji poslije)



R4 = stanje prije naredbe      R4' = stanje poslije naredbe

© Kovač, Basch, FER, Zagreb

18

### *ldm/stm - adresiranje*

- Još jedna mogućnost koja se pruža programeru je automatsko pomicanje pokazivača, odnosno osvježavanje vrijednosti koja se nalazi u registru koji služi za adresiranje.

- Tako se na primjer naredbe

LDMIB R2, {R4,R8,R9}

LDMIB R2!, {R4,R8,R9} ; iza R2 piše se uskličnik !

razlikuju po tome što će nakon izvođenja prve naredbe vrijednost registra R2 ostati **nepromjenjena**, dok će nakon druge naredbe registro R2 biti **promijenjen** (uvećan za 12<sub>10</sub>, jer se R2 uvećava za 4 nakon čitanja podatka za svaki registar iz niza)

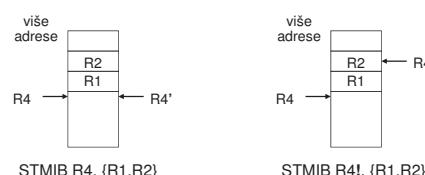
© Kovač, Basch, FER, Zagreb

20

© Kovač, Basch, FER, Zagreb

19

### *Ldm/stm korištenje*



R4 = stanje prije naredbe      R4' = stanje poslije naredbe

© Kovač, Basch, FER, Zagreb

21

### *Ldm/stm PRAVILA!!!*

- Bez obzira na izbor nekog od četiri adresiranja i bez obzira na redoslijed kojim su registri napisani u naredbi, **uvijek je na najnižoj adresi zapisan registar sa najnižim brojem**
- Ako se podaci žele zapisati u memoriju naredbom STM, a zatim procitati u istom redoslijedu naredbom LDM te ako se koriste nastavci za OBICNO ADRESIRANJE (ne ekvivalenti za stog!!!), tada način adresiranja u naredbi LDM mora biti **INVERZAN** načinu adresiranja u naredbi STM
- Primjer:

- Zapisivanje      STMIA ←  
• Čitanje      LMDDB ←
- U naredbama se  
koristi inverzno  
adresiranje  
IA ↔ DB

© Kovač, Basch, FER, Zagreb

22

© Kovač, Basch, FER, Zagreb

23

### *Primjeri korištenja LDM i STM ...*

U donjim primjerima pretpostavljene vrijednosti prije izvođenja naredaba LDM/STM su:  
r0=0    r1=1    r2=2    r3=3    r13=0x9000

a) STMIB r13!, {r0,r1,r2,r3} ; vrijednosti se spreme, r13 ostane nepromijenjen

Nakon izvođenja gornje naredbe, stanje u memoriji je (heksadekadski, little-endian):  
0x00009000: 10 00 FF E7 00 00 00 01 00 00 02 00 00  
0x00009010: 03 00 00 00 00 E8 00 E8 10 00 FF E7 00 E8 00 E8  
r13=9000

b) STMIB r13!, {r3,r1,r2,r2} ; redoslijed pisanja registara ne utječe na redoslijed spremanja

0x00009000: 10 00 FF E7 00 00 00 01 00 00 02 00 00  
0x00009010: 03 00 00 00 00 E8 00 E8 10 00 FF E7 00 E8 00 E8  
r13=9000

c) STMIB r13!, {r0,r3} ; osvježava se r13

0x00009000: 10 00 FF E7 00 00 00 03 00 00 00 E8 00 E8 00  
r13=9008

© Kovač, Basch, FER, Zagreb

24

© Kovač, Basch, FER, Zagreb

25

### *... Primjeri korištenja LDM i STM ...*

d) STMDB r13!, {r0,r1,r2,r3} ; primjer koristenja DB  
0x00008FF0: 00 00 00 01 00 00 00 02 00 00 00 03 00 00 00  
0x00009000: 10 00 FF E7 00 E8 00 E8 10 00 FF E7 00 E8 00 E8  
r13 = 8FF0

e) STMDA r13!, {r0,r1,r2,r3} ; primjer koristenja DA i kasnijeg obnavljanja registara  
;0x00008FF0 00 E8 00 E8 00 00 00 01 00 00 00 02 00 00  
;0x00009000 03 00 00 00 00 E8 00 E8 10 00 FF E7 00 E8 00 E8  
;r13 = 8FF0

Da bi u registre r0, r1, r2 i r3 učitali iste podatke koje smo iz njih spremili u memoriju pomoću naredbe STMDA, moramo upotrijebiti naredbu LDM s inverznim nastavkom IB:

LDMIB r13!, {r0,r1,r2,r3}

## Primjeri korištenja LDM i STM



f) STMFA r13!, {r0,r1,r2,r3} ; primjer spremanja registra pomoću STM i kasnijeg obnavljanja registra naredbom LDM uz korištenje ; nastavka za STOG

```
0x000009000: 00 E8 00 E8 00 00 00 00 01 00 00 00 02 E8 00 E8
0x000009010: 03 00 00 00 E8 00 E8 10 00 FF E7 00 E8 00 E8
r13 = 9010
```

Da bi u registre r0, r1, r2 i r3 učitali iste podatke koje smo iz njih spremili u memoriju pomoću naredbe STMFA, moramo upotrijebiti naredbu LDM s istim nastavkom FA:

LDMFA r13!, {r0,r1,r2,r3} ; za adresiranje stoga nastavci moraju biti isti

## Aritmetičko-logičke naredbe i naredbe za usporedbu

| Ime | Engleski naziv              | Opis naredbe                                   |
|-----|-----------------------------|------------------------------------------------|
| AND | Logical AND                 | Rd := Rn AND drugi_operand (logičko I)         |
| eor | Logical Exclusive OR        | Rd := Rn EOR drugi_operand (logičko ekskl.ILI) |
| ORr | Logical (inclusive) OR      | Rd := Rn OR drugi_operand                      |
| BIC | Bit Clear                   | Rd := Rn AND NOT(drugi_operand)                |
| ADD | Add                         | Rd := Rn + drugi_operand                       |
| ADC | Add with Carry              | Rd := Rn + drugi_operand + C zastavica         |
| SUB | Subtract                    | Rd := Rn - drugi_operand                       |
| SBC | Subtract with Carry         | Rd := Rn - drugi_operand - NOT(C zastavica)    |
| RSB | Reverse Subtract            | Rd := drugi_operand - Rn                       |
| RSC | Reverse Subtract with Carry | Rd := drugi_operand - Rn - NOT(C zastavica)    |
| CMP | Compare                     | Osvježi zastavice nakon Rn - drugi_operand     |
| CMN | Compare Negated             | Osvježi zastavice nakon Rn + drugi_operand     |
| TST | Test                        | Osvježi zastavice nakon Rn AND drugi_operand   |
| TEQ | Test Equivalence            | Osvježi zastavice nakon Rn EOR drugi_operand   |
| MOV | Move                        | Rd := drugi_operand (prvog operanda nema)      |
| MVN | Move Not                    | Rd := NOT drugi_operand (prvog operanda nema)  |

## Aritmetičko-logičke naredbe

- ARM u zastavici C **uvijek čuva prijenos\*** (i kod zbrajanja i kod oduzimanja) pa se za posudbu mora koristiti inverz od C
- Ako se aritmetičko-logičkoj naredbi doda nastavak 'S' (Save condition codes) tada će se nakon izvršene naredbe osvježiti zastavice stanja
- Bez nastavka 'S', naredba **ne mijenja** zastavice\*\*
- Naredbe za usporedbu uvijek osvježavaju zastavice stanja (i nigdje ne spremaju rezultat)

\* Različito od FRISC-a, koji u svojoj zastavici C čuva prijenos kod zbrajanja, ali kod oduzimanja čuva posudbu

\*\* Različito od FRISC-a, koji uvijek osvježava zastavice

## Primjer programa za aritm. naredbe (atoi\_v1)

Program za pretvorbu 4 dekadskih ASCII-znaka na adresi 0x8100 u cijeli broj. Rezultat treba spremiti iza ASCII-znakova.

```
ADDR EQU 0x8100 ; adresa na kojoj se nalaze podatci
MOV R4, #ADDR ; postavlja registar R4 na početak podataka
MOV R7, #4 ; brojač ASCII-znakova = 4
MOV R6, #0 ; resetiranje rezultata
PETLJA
 LDRB R0, [R4], #1 ; učitavanje ASCII-znaka
 SUB R0, R0, #0x30 ; pretvorba ASCII-znaka u binarni broj:
 ; prethodni rezultat se množi sa 10
 ADD R6, R6, R6, LSL #2 ; množenje sa 5
 ADD R6, R0, R6, LSL #1 ; množenje sa 2 i zbrajanje
 ; tako da je: R6 = (5*R6)*2 + R0
 SUBS R7, R7, #1 ; brojač prolaza petlje
 BHI PETLJA
KRAJ
 STR R6, [R4] ; spremanje rezultata
```

## Naredbe za obradu podataka

- ARM-ove naredbe za obradu podataka obrađuju podatke iz registara
- Postoje tri grupe naredaba za obradu podataka:
  - aritmetičko-logičke naredbe i naredbe za usporedbu
  - naredbe za množenje
  - naredba za brojenje vodećih nula

## Aritmetičko-logičke naredbe

- Aritmetičko-logičke naredbe imaju dva operanda (osim naredaba MOV i MVN koje imaju samo jedan operand) i spremaju rezultat u zadani registar\* (osim naredaba za usporedbu koje zanemaruju rezultat)
- Od dva operanda koji se mogu koristiti u operaciji jedan uvijek mora biti registar, a drugi može biti
  - neposredna vrijednost
  - registar
  - vrijednost registra nad kojom se obavlja određen pomak

\* Za razliku od FRISC-a kod kojeg se određeni registar piše na kraju - iza operanada, kod ARM-a se određeni registar zadaje na početku - prije operanada

## Primjeri korištenja naredaba za obradu podataka

|                                                   |         |
|---------------------------------------------------|---------|
| AND R1, R2, R0 ; R1 = R2 and R0                   |         |
| ADD R0, R1, R0, LSL #2 ; R0 = R1 + R0 * 4         |         |
| MOV R2, #0xFF ; R2 = 0xFF                         |         |
| EOR R1, R0, R0, ROR #16 ; R1 = A^C, B^D, C^A, D^B |         |
| BIC R1, R1, #0xFF0000 ; R1 = A^C, 0, C^A, D^B     |         |
| Operandi u naredbi EOR: R0                        | A B C D |
| R0 ROR #16                                        | C D A B |

## Primjer programa za aritm. naredbe (add\_64)

; Program za 64-bitno zbrajanje dvaju brojeva na adresi 0x8100  
; Rezultat se spremi iza operanada

```
ADDR EQU 0x8100 ; adresa na kojoj se nalaze podatci
MOV R4, #ADDR ; upisuje 8100(16) u registar R4
LDMIA R4!, {R5,R6,R7,R8} ; učitavanje oba operanda korištenjem
 ; naredbe Load Multiple
ADDS R5, R5, R7 ; zbrajanje niža 32 bita s postavljanjem zastavica
ADCS R6, R6, R8 ; zbrajanje viša 32 bita i prijenosa iz prethodne operacije
STMIA R4!, {R5,R6} ; spremanje rezultata iza operanada
```

## Naredbe za množenje

- Naredbe za množenje nisu smještene u grupu osnovnih aritmetičko-logičkih naredbi, već se definiraju zasebno
- Postoji šest naredbi za množenje: MUL, MLA, SMULL, UMULL, SMLAL i UMLAL
  - U ATLAS-u su implementirane samo naredbe MUL i MLA!!!**
- Obično množenje (MUL) se obavlja nad dva 32-bitna podatka iz registara opće namjene, a 32-bitni rezultat (nižih 32 bita umnoška) se ponovo spremi u registar
- Naredbe za dugo množenje (SMULL i UMULL) množe dva 32-bitna podatka iz registara i spremaju svih 64 bita rezultata u dva odabrana registra za rezultat. Dugo množenje može množiti brojeve s predznakom (SMULL) ili bez predznaka (UMULL)

© Kovač, Basch, FER, Zagreb

34

## Naredbe za množenje

| Ime   | Engleski naziv                    | Opis naredbe                                                                                                             |
|-------|-----------------------------------|--------------------------------------------------------------------------------------------------------------------------|
| MUL   | Multiply                          | $Rd = (Rm * Rs)[31:0]$                                                                                                   |
| MLA   | Multiply Accumulate               | $Rd = (Rm * Rs + Rn)[31:0]$                                                                                              |
| SMULL | Signed Multiply Long              | $RdHi = (Rm * Rs)[63:32] /*Sa predznakom*/$<br>$RdLo = (Rm * Rs)[31:0]$                                                  |
| UMULL | Unsigned Multiply Long            | $RdHi = (Rm * Rs)[63:32] /*Bez predznaka*/$<br>$RdLo = (Rm * Rs)[31:0]$                                                  |
| SMLAL | Signed Multiply Accumulate Long   | $RdLo=(Rm * Rs)[31:0]+RdLo /*Sa predznakom*/$<br>$RdHi = (Rm * Rs)[63:32] + RdHi + prijenos iz ((Rm * Rs)[31:0] + RdLo)$ |
| UMLAL | Unsigned Multiply Accumulate Long | $RdLo=(Rm * Rs)[31:0]+RdLo /*Bez predznaka*/$<br>$RdHi = (Rm * Rs)[63:32] + RdHi + prijenos iz ((Rm * Rs)[31:0] + RdLo)$ |

© Kovač, Basch, FER, Zagreb

36

## Pojašnjenje množenja (4/8 bita)

**NBC i 2'k:**

|                                                                                                       |                                                                                                                                                                                                                       |                                                                                                                                                                                      |
|-------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\begin{array}{r} 0010 * 0011 \\ \hline 0010 \\ 0010 \\ 0000 \\ + 0000 \\ \hline 0000110 \end{array}$ | <b>mali pozitivni brojevi</b><br><div style="border: 1px solid black; padding: 5px; display: inline-block;">           NBC opseg:<br/>           0 ... 15 (4-bitni)<br/>           0 ... 255 (8-bitni)         </div> | <b>2'k opseg:</b><br><div style="border: 1px solid black; padding: 5px; display: inline-block;">           -8 ... +7 (4-bitni)<br/>           -128 ... +127 (8-bitni)         </div> |
|-------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

→ 4-bitni rezultat: 0110  
→ 8-bitni rezultat: 0000 0110

za NBC: operandi su 2 i 3, a rezultat je:  
 6 (u 4 bita) - OK  
 6 (u 8 bita) - OK

za 2'k: operandi su 2 i 3, a rezultat je:  
 6 (u 4 bita) - OK  
 6 (u 8 bita) - OK

© Kovač, Basch, FER, Zagreb

38

## Pojašnjenje množenja (4/8 bita)

|                                                                                                       |                                                                                   |
|-------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| $\begin{array}{r} 1110 * 0001 \\ \hline 1110 \\ 0000 \\ 0000 \\ + 0000 \\ \hline 0001110 \end{array}$ | <b>NBC: mali pozitivni brojevi</b><br><b>2'k: mali pozitivan i negativan broj</b> |
|-------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|

→ 4-bitni rezultat: 1110  
→ 8-bitni rezultat: 0000 1110

za NBC: operandi su 14 i 1, a rezultat je:  
 14 (u 4 bita) - OK  
 14 (u 8 bita) - OK

za 2'k: operandi su -2 i 1, a rezultat je:  
 -2 (u 4 bita) - OK  
 14 (u 8 bita) - pogrešan rezultat ←

© Kovač, Basch, FER, Zagreb

40

## Naredbe za množenje

- Obično 32-bitno množenje koristimo kad znamo da su operandi "dovoljno mali" da rezultat stane u 32-bitu, a dugo 64-bitno množenje koristimo u općenitim slučajevima
- Kao dodatna opcija gore navedenim naredbama nudi se mogućnost da se umnožak pribraja sadržaje nekog registra (ili paru registara za dugo množenje). Takve naredbe općenito su poznate kao Multiply Accumulate, a kod procesora ARM nazivaju se: MLA, SMLAL i UMLAL
- Ako se naredbama za množenje doda nastavak S, tada će se nakon izvođenja osvježiti zastavice N i Z.

© Kovač, Basch, FER, Zagreb

35

## Pojašnjenje množenja (4/8 bita)

- Zašto za 64-bitno množenje postoje zasebne naredbe za signed i unsigned brojeve, a za 32-bitno množenje ne postoje?
- Zato jer 32-bitno množenje "funkcionira" za obje vrste brojeva, a 64-bitno množenje se razlikuje.
- Na sljedećim slajdovima je nekoliko primjera za 4/8-bitno množenje, koji ovo ilustriraju
  - Pokazana su množenja 4-bitnih brojeva s 4 i 8 bitnim rezultatima
  - Operandi su mali ili veliki brojevi (pod "mali" se misli da im je umnožak malen, analogno za "veliki") s različitim kombinacijama predznaka
  - Problem nije u prekoračenju opsega u 4-bitnom rezultatu, nego u **pogrešnom 8-bitnom rezultatu**

© Kovač, Basch, FER, Zagreb

37

## Pojašnjenje množenja (4/8 bita)

|                                                                                                        |                                                                            |
|--------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------|
| $\begin{array}{r} 1110 * 1101 \\ \hline 1110 \\ 0000 \\ 1110 \\ + 1110 \\ \hline 10110110 \end{array}$ | <b>NBC: veliki pozitivni brojevi</b><br><b>2'k: mali negativni brojevi</b> |
|--------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------|

→ 4-bitni rezultat: 0110  
→ 8-bitni rezultat: 1011 0110

za NBC: operandi su 14 i 13, a rezultat je:  
 6 (u 4 bita) - prekoračenje opsega  
 182 (u 8 bita) - OK

za 2'k: operandi su -2 i -3, a rezultat je:  
 6 (u 4 bita) - OK  
 -74 (u 8 bita) - pogrešan rezultat ←

© Kovač, Basch, FER, Zagreb

39

## Pojašnjenje množenja (4/8 bita)

|                                                                                                       |                   |                                 |
|-------------------------------------------------------------------------------------------------------|-------------------|---------------------------------|
| $\begin{array}{r} 0110 * 0011 \\ \hline 0110 \\ 0110 \\ 0000 \\ + 0000 \\ \hline 0010010 \end{array}$ | <b>NBC i 2'k:</b> | <b>veliki pozitivni brojevi</b> |
|-------------------------------------------------------------------------------------------------------|-------------------|---------------------------------|

→ 4-bitni rezultat: 0010  
→ 8-bitni rezultat: 0001 0010

za NBC: operandi su 6 i 3, a rezultat je:  
 2 (u 4 bita) - prekoračenje opsega  
 18 (u 8 bita) - OK

za 2'k: operandi su 6 i 3, a rezultat je:  
 2 (u 4 bita) - prekoračenje opsega  
 18 (u 8 bita) - OK

© Kovač, Basch, FER, Zagreb

41



$$\begin{array}{r}
 1110 * 0011 \\
 1110 \\
 1110 \\
 0000 \\
 + 0000 \\
 \hline
 0101010
 \end{array}$$

4-bitni rezultat: 1010  
8-bitni rezultat: 0010 1010

NBC: veliki pozitivni brojevi  
2'k: mali pozitivan i negativan broj

za NBC: operandi su 14 i 3, a rezultat je:  
10 (u 4 bita) - prekoračenje opsega  
42 (u 8 bita) - OK

za 2'k: operandi su -2 i 3, a rezultat je:  
-6 (u 4 bita) - OK  
42 (u 8 bita) - pogrešan rezultat ←

$$\begin{array}{r}
 1110 * 1001 \\
 1110 \\
 0000 \\
 0000 \\
 + 1110 \\
 \hline
 1111110
 \end{array}$$

4-bitni rezultat: 1110  
8-bitni rezultat: 0111 1110

za NBC: operandi su 14 i 9, a rezultat je:  
14 (u 4 bita) - prekoračenje opsega  
126 (u 8 bita) - OK

za 2'k: operandi su -2 i -7, a rezultat je:  
-2 (u 4 bita) - prekoračenje opsega  
126 (u 8 bita) - pogrešan rezultat ←

### Primjeri korištenja naredaba za množenje:

|                      |                                             |
|----------------------|---------------------------------------------|
| MUL R4, R2, R1       | ;R4 = R2 x R1                               |
| MULS R4, R2, R1      | ;R4 = R2 x R1, set N and Z flags            |
| MLA R7, R8, R9, R3   | ;R7 = R8 x R9 + R3                          |
| SMULL R4, R8, R2, R3 | ;R4 = (R2 x R3)[31:0] R8 = (R2 x R3)[63:32] |
| UMULL R6, R8, R0, R1 | ;R8, R6 = R0 x R1                           |
| UMLAL R5, R8, R0, R1 | ;R8, R5 = R0 x R1 + R8, R5                  |

### Primjer programa za naredbu MUL

|                    |                                     |
|--------------------|-------------------------------------|
| ADDR EQU 0x8100    | ; adresa na kojoj se nalaze podaci  |
| MOV R4, #ADDR      | ; postavlja u R4 adresu podataka    |
| LDR R5, [R4], #4   | ; multiplikator se učitava u R5     |
| LDR R6, [R4], #4   | ; multiplikand se učitava u R6      |
| SMULL R7,R8,R5,R6  | ; množenje - rezultati su u R7 i R8 |
| STMIA R4!, {R7,R8} | ; spremi rezultate iza operanada    |

### Naredba za brojenje vodećih nula

- Naredba za brojenje vodećih nula je specifična naredba koja je vrlo korisna kod izvedbe matematičkih algoritama (normiranje brojeva) i algoritama za kompresiju (npr. metode duljine niza).
- U ATLAS-u nije implementirana naredba CLZ!!!**
- Ova naredba uzima jedan registar kao operand i vraća rezultat u drugom registru.
- Rezultat predstavlja broj nula koje prethode najvišoj jedinici u ulaznom operandu (promatranom kao niz binarnih znamenaka).

### Naredbe grananja

- Naredba B (Branch): bezuvjetno ili uvjetno grananje na memorijsku adresu koja se nalazi 32 MB ispred ili iza trenutačne naredbe\* (relativan skok u odnosu na sadržaj PC)
- Drugi način grananja je da se u registar PC izravno stavi neka vrijednost, čime se skok ne ograničava na udaljenost od 32 MB, već se može skočiti na bilo koju adresu u adresnom području (tj. može se skočiti bilo gdje unutar 4 GB) \*\*

\* Slično FRISC-ovoj naredbi JR

\*\* Sličan učinak kod FRISC-a možemo dobiti naredbom JP (Rx)

### Naredbe grananja - uvjeti

| Mnemonički naziv | Puni naziv (engleski)             | Ispitivanje zastavica |
|------------------|-----------------------------------|-----------------------|
| EQ               | Equal                             | Z                     |
| NE               | Not equal                         | !Z                    |
| CS/HS            | Carry set/unsigned higher or same | C                     |
| CC/LO            | Carry clear/unsigned lower        | !C                    |
| MI               | Minus/negative                    | N                     |
| PL               | Plus/positive or zero             | !N                    |
| VS               | Overflow                          | V                     |
| VC               | No overflow                       | !V                    |
| HI               | Unsigned higher                   | C and !Z              |
| LS               | Unsigned lower or same            | !C or Z               |
| GE               | Signed greater than or equal      | N == V                |
| LT               | Signed less than                  | N != V                |
| GT               | Signed greater than               | !Z and N == V         |
| LE               | Signed less than or equal         | Z or N != V           |
| AL               | Always (unconditional)            | -                     |
| (NV)             | See Condition code 0b1111         | -                     |

## Grananje u potprogram

- Naredba BL (Branch and Link) poziva potprogram:
  - Sprema adresu sljedeće naredbe (povratnu adresu) **u registar R14** (koji se tada naziva Link Registar, LR)
  - nakon toga izvodi grananje na početak potprograma.
- Povratak iz potprograma izvodi se jednostavnim kopiranjem sadržaja registra LR u PC (npr. **naredbom MOV PC,LR**)
- **Ovakvim pozivom i povratkom iz potprograma nije moguće ugniježđeno pozivanje potprograma!**
- **Za ugniježđene pozive, na početku svakog potprograma mora se spremiti vrijednost iz R14 na stog, a prije povratka iz potprograma treba vratiti tu vrijednost sa stoga u R14.**

© Kovač, Basch, FER, Zagreb

50

## Primjer uvjetnog izvođenja niza naredaba

### 1. način - korištenjem naredbe uvjetnog grananja:

```
CMP R0, #0
BNE DALJE
MOV R1, #1 ; uvjetni dio koda
MOV R2, #2 ; uvjetni dio koda
MOV R3, #3 ; uvjetni dio koda
DALJE
...
```

Ovisno o protočnoj strukturi i ovisno koliko često je ispitivani uvjet istinit, može se odrediti koji način pisanja je efikasniji u pojedinoj situaciji.

### 2. način - korištenjem uvjetnog izvođenja naredaba:

```
CMP R0, #0
MOVEQ R1, #1 ; uvjetni dio koda
MOVEQ R2, #2 ; uvjetni dio koda
MOVEQ R3, #3 ; uvjetni dio koda
```

© Kovač, Basch, FER, Zagreb

52

## Naredbe za pristup registrima stanja

| Oznaka | Bitovi  | Naziv polja u registru       |
|--------|---------|------------------------------|
| f      | [31:24] | Polje zastavica (flags)      |
| s      | [23:16] | Polje stanja (status)        |
| e      | [15:8]  | Polje proširenja (extension) |
| c      | [7:0]   | Polje upravljanja (control)  |

- Svaki register stanja podijeljen je u 4 polja kojima se može neovisno pristupati. Prilikom upisa u register stanja naredbom MSR, programer mora definirati u koje polje želi upisati podatak
- Primjer naredba:
  - MRS R0, CPSR
  - MSR CPSR\_cef, R0
  - MSR CPSR\_f, R0
  - MSR CPSR\_fsc, R0

© Kovač, Basch, FER, Zagreb

54

## Dio ARM4: Adresiranje

Načini adresiranja kod procesora ARM

Ovaj dokument namijenjen je isključivo za osobnu upotrebu studentima Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

Svaka druga upotreba ili umnožavanje bilo kojeg dijela ovog dokumenta nije dozvoljena bez pismene dozvole autora.

© Mario Kovač, Danko Basch



© Kovač, Basch, FER, Zagreb

## Primjeri naredaba za grananje

```
B labela ; bezuvjetni skok
BCC labela ; uvjetni skok (carry clear)
BEQ labela ; uvjetni skok (equal)

MOV PC, #0 ; R15 = 0, tj. skoči na adresu 0
MOV PC, R3 ; R15 = R3, skok na bilo koju 32-bitnu adresu
 ; koja je zadana registrom R3

BL func ; poziv potprograma
MOV PC, LR ; R15=R14, tj. povratak iz potprograma
```

© Kovač, Basch, FER, Zagreb

51

## Naredbe za pristup registrima stanja

- Omogućuju prijenos podataka iz registara stanja (CPSR, SPSR) procesora ARM u neki register opće namjene i obratno.
- Pisanjem u CPSR, na primjer, programer može postaviti stanja zastavica, stanja bitova za omogućavanje prekida kao i procesorski način.
- Naredba **MRS** (Move to Register from Status register) kopira sadržaj registra stanja CPSR ili SPSR iz načina rada u kojem se procesor trenutno nalazi u jedan od registara opće namjene koji se može dalje ispitivati ili mijenjati.
- Naredbom **MSR** Move to Status register from Register može se upisati neposredna vrijednost ili sadržaj registra opće namjene u registre stanja CPSR ili SPSR iz načina rada u kojem se procesor trenutno nalazi.

© Kovač, Basch, FER, Zagreb

53

## Primjer korištenja

- primjer omogućavanja prekida (brisanje bita I u registru CPSR)

```
MRS R0, CPSR
BIC R0, R0, #0x80
MSR CPSR_C, R0
```

© Kovač, Basch, FER, Zagreb

55

## Načini adresiranja

- Procesor ARM omogućuje različita adresiranja. Oblici adresiranja vrstani su u pet načina (engl. Addressing Modes) i u tablicama u Prilogu su označeni brojevima 1 do 5.
- Načini adresiranja koriste se u pojedinim naredbama prema sljedećoj tablici:
  - načini adresiranja 1: naredbe za obradu podataka
  - načini adresiranja 2: naredbe Load i Store word ili unsigned byte
  - načini adresiranja 3: naredbe Load i Store halfword ili signed byte
  - načini adresiranja 4: naredbe Load i Store Multiple
  - načini adresiranja 5: naredbe Load i Store Coprocessor

© Kovač, Basch, FER, Zagreb

2



## Primjeri dozvoljenih i nedozvoljenih vrijednosti

- Dozvoljene vrijednosti
  - 0xFF, 0x104, 0xF00000F,...
- Nedozvoljene vrijednosti
  - 0x101, 0xFF1, 0xFF04, 0xFFFFFFFF,...
- Napomena:
  - mnoge vrijednosti mogu se dobiti upotrebom naredbe MVN (Move NOT) koja radi komplement operanda
  - Npr: običnom naredbom ne možemo koristiti neposrednu vrijednost 0xFFFFFFFF00. No, tu neposrednu vrijednost možemo učitati u registar koristeći naredbu MVN, SUB i slično:

NE!!! MOV R0, #0xFFFFFFFF  
Da!!! MVN R0, #0

© Kovač, Basch, FER, Zagreb

11

## Primjeri dozvoljenih i nedozvoljenih vrijednosti

- NE!!! MOV R4, #0xFFFFFFFF  
Da!!! MVN R4, #0xF00000F
- Neke aritmetičke operacije možemo zamijeniti KOMPLEMENTARNIM OPERACIJAMA
  - NE!!! ADDS R4, R1, #0xFFFFFFFF  
Da!!! SUBS R4, R1, #1
- Primjer dozvoljene neposredne vrijednosti kodiran u binarnom obliku:  
[0x e29142ff] adds r4,r1,#0xf00000f

| Naredba                                    | 31    | 30   | 29 | 28 | 27     | 26   | 25 | 24   | 23   | 22         | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |  |  |  |          |  |  |  |  |  |  |  |
|--------------------------------------------|-------|------|----|----|--------|------|----|------|------|------------|----|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|--|--|--|----------|--|--|--|--|--|--|--|
| Aritmetičko logička, neposredna vrijednost | Uvjet | 0    | 0  | 1  | Op kod | S    | Rn |      | Rd   | rotate_imm |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |  |  |  |          |  |  |  |  |  |  |  |
| ADDS R4,R1,#0xF00000F                      |       | 1110 | 0  | 0  | 1      | 0100 | 1  | 0001 | 0100 | 0010       |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |  |  |  | 11111111 |  |  |  |  |  |  |  |

© Kovač, Basch, FER, Zagreb

12

## ATLAS - NAPOMENA

- U primjerima koje radite za laboratorijske vježbe (**u ATLAS-u**) neposredne vrijednosti **morate kodirati sami** (asembler ne podržava automatsku pretvorbu)
- Način pisanja:  
baza<rotacija> (u ATLAS-u se koristi **LIJEVA** rotacija)
- Primjer:  
broj 100<sub>16</sub> piše se kao 1<8    1 0000 0000<sub>2</sub> = 1<sub>2</sub><8  
broj 180<sub>16</sub> piše se kao 18<4    11000 0000<sub>2</sub> = 11000<sub>2</sub><4  
ili kao 6<6    11000 0000<sub>2</sub> = 110<sub>2</sub><6

© Kovač, Basch, FER, Zagreb

13

## Načini adresiranja 1 u Prilogu

- Pogledajte tablicu za npr. naredbu MOV (strana 5 Priloga)
- MOV{cond}{S} Rd, <Oprnd2>
- Na strani 1 možete vidjeti da se opis <Oprnd2> nalazi u tablici Operand2 (strane 3 i 4)
- Iz tablice izaberete neki od načina adresiranja
- Napomena!!! **Ne postoji** naredba npr. LSL R0, #2 (dešava se na ispitima). Treba koristiti naredbu MOV (u ovom primjeru: MOV R0, R0, LSL #2).

© Kovač, Basch, FER, Zagreb

15

14

## Načini adresiranja 2

## Načini adresiranja 2

- Karakteristični su za naredbe load i store kojima se premješta riječ ili nepredznačeni bajt
- Postoje devet mogućih kombinacija adresiranja s obzirom na vrstu odmaka te vrstu indeksiranja
- Odmak se zadaje neposrednim podatkom, registrom ili pomaknutim registrom\*. Ako je izabran pomak registra, tada se vrsta pomaka (LSL, LSR, ASR, ROR, RRX) zadaje kao u načinu adresiranja 1, no uz ograničenje da se iznos pomaka zadaje samo neposredno

\* Odmak se ne mora napisati i tada se podrazumijeva da je 0 (zadan neposredno)

© Kovač, Basch, FER, Zagreb

16

## Načini adresiranja 2 u Prilogu

- Pogledajte tablicu za npr. naredbu LDR (strana 6 priloga)
- LDR{cond} Rd, <a\_mode2>
- Na strani 1 možete vidjeti da se opis <a\_mode2> nalazi u tablici Način adresiranja 2 (strana 2)
- Iz tablice izaberete neki od načina adresiranja i upotrijebite ga u naredbi. Npr.

LDRHI R0, [R3, -R5, ROR #22]!

| Adresiranje                                 | Sintaksa adresiranja                                                                                                                                                          |
|---------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Neposredni odmak                            | [<Rn>, #+/-<offset_12>]                                                                                                                                                       |
| Registarski odmak                           | [<Rn>, +/-<Rm>]                                                                                                                                                               |
| Registarski odmak s pomakom                 | [<Rn>, +/-<Rm>, LSL #<shift_imm>]<br>[<Rn>, +/-<Rm>, LSR #<shift_imm>]<br>[<Rn>, +/-<Rm>, ASR #<shift_imm>]<br>[<Rn>, +/-<Rm>, ROR #<shift_imm>]<br>[<Rn>, +/-<Rm>, RRX]      |
| Neposredni predindeksirani                  | [<Rn>, #+/-<offset_12>!]                                                                                                                                                      |
| Registarski predindeksirani                 | [<Rn>, +/-<Rm>!]                                                                                                                                                              |
| Registarski predindeksirani odmak s pomakom | [<Rn>, +/-<Rm>, LSL #<shift_imm>!]<br>[<Rn>, +/-<Rm>, LSR #<shift_imm>!]<br>[<Rn>, +/-<Rm>, ASR #<shift_imm>!]<br>[<Rn>, +/-<Rm>, ROR #<shift_imm>!]<br>[<Rn>, +/-<Rm>, RRX!] |
| Neposredni postindeksirani                  | [<Rn>, #+/-<offset_12>]                                                                                                                                                       |
| Registarski postindeksirani                 | [<Rn>, +/-<Rm>]                                                                                                                                                               |
| Registarski postindeksirani odmak s pomakom | [<Rn>, +/-<Rm>, LSL #<shift_imm>]<br>[<Rn>, +/-<Rm>, LSR #<shift_imm>]<br>[<Rn>, +/-<Rm>, ASR #<shift_imm>]<br>[<Rn>, +/-<Rm>, ROR #<shift_imm>]<br>[<Rn>, +/-<Rm>, RRX]      |

17

© Kovač, Basch, FER, Zagreb

18

## Načini adresiranja 3

- Šest formata adresa koje se koriste kod ostalih naredaba load i store (za poluriće i predznačeni bajt)
- Ovih šest formata adresa slični su formatima iz načina adresiranja 2. Razlike su:
  - odmak se može zadati neposrednom vrijednošću ili registrom (ne može se zadati registar s pomakom)
  - neposredna vrijednost odmaka se zapisuje samo sa osam bitova

## Načini adresiranja 3

| Adresiranje                       | Sintaksa adresiranja    |
|-----------------------------------|-------------------------|
| Neposredni odmak                  | [<Rn>, #+/-<offset_8>]  |
| Registarski odmak                 | [<Rn>, +/-<Rm>]         |
| Neposredni predindeksirani odmak  | [<Rn>, #+/-<offset_8>]! |
| Registarski predindeksirani odmak | [<Rn>, +/-<Rm>]!        |
| Neposredni postindeksirani odmak  | [<Rn>], #+/-<offset_8>  |
| Registarski postindeksirani odmak | [<Rn>], +/-<Rm>         |

## Načini adresiranja 3 u Prilogu

- Pogledajte tablicu za npr. naredbu LDR (str. 6 priloga)
- LDR{cond}SB Rd, <a\_mode3>
- Na strani 1 možete vidjeti da se opis <a\_mode3> nalazi u tablici Način adresiranja 3 (strana 2)
  - Iz tablice izaberete neki od načina adresiranja i upotrijebite ga u naredbi. Npr.

LDRGTSB R0, [R3], #200

## Načini adresiranja 4 u Prilogu

- Pogledajte tablicu za npr. naredbu LDM (strana 6 priloga)
- LDM{cond} <a\_mode4L> Rd{!}, <reglist-pc>
- Na strani 1 možete vidjeti da se opis <a\_mode4L> nalazi u tablici Način adresiranja 4 (strana 3)
  - Iz tablice izaberete neki od načina adresiranja i upotrijebite ga u naredbi. Npr.
- LDMGTIA R0!, {R4,R5,R6}
- Napomena: <reglist-pc> u opisu naredbe znači da se u popisu registara ne smije staviti PC

## Načini adresiranja 4

- Naredbama load i store multiple moguće je pročitati ili upisati sadržaj jednog, sadržaj više ili čak sadržaje svih registara u memoriju, odnosno u niz uzastopnih memorijskih lokacija.
- Registar s najmanjim brojem je uvijek zapisan na najmanju memorijsku adresu, a registar s najvećim brojem na najveću.
- Na isti način, kod čitanja: u registar s najmanjim brojem učitava se podatak s najmanje adrese, a u registar s najvećim brojem učitava se podatak s najveće adrese.
- IA, IB, DA, DB (FD, FA, ED, EA)

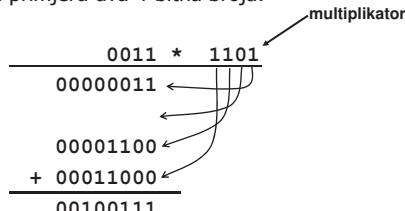
## Načini adresiranja 5

- Koriste se kod naredaba za prijenos podataka prema koprocesoru
- Nećemo ih koristiti

## Primjer: množenje dva 16-bitna broja

Treba pomnožiti dva 16-bitna broja metodom pomaka. Operandi su spremljeni od adrese 0x8100, a 32-bitni rezultat treba staviti iza operanada. Multiplikand može biti u zapisu NBC ili 2'k, a multiplikator može biti samo u zapisu NBC.

Načelo množenja na primjeru dva 4-bitna broja:



## Primjer programa za množenje

```
; Program za množenje 16-bitnih brojeva metodom pomaka, v.3 - adrese kao prije
ADDR EQU 0x8100 ;adresa na kojoj se nalaze podatci
MOV R4, #ADDR ;postavlja u registar R4 adresu podataka
LDRH R5, [R4], #2 ;multiplikator se učitava u R5
LDRSH R6, [R4], #2 ;multiplikand se učitava u R6 (predznak sačuvan)
MOV R7, #0 ;čisti se registar za spremanje rezultata (R7)

PETLJA
 MOVS R5, R5, LSR #1 ;pomak multiplikatora za jedan bit udesno,
 ;najniži bit otici će u C
 ADDCS R7, R7, R6 ;ako je zastavica C=1, R6 se dodaje privremenom rezultatu
 MOV R6, R6, LSL #1 ;pomak R6 za jedan bit ulijevo
 ;(priprema za mogući sljedeći korak)
 CMP R5, #0 ;provjerava je li multiplikator različit od nule
 BNE PETLJA ;ako jeste, onda se petlja ponavlja
 STR R7, [R4] ;spremanje rezultata
 . . .
```

## Primjer dijeljenja metodom pomaka



## Postupak...



- Dijeljenje A/B (dijelitelja B, djeljeniku A) odvija se u dva koraka:
- Prvi korak:** poravnanje brojeva – traženje najvećeg višekratnika djelitelja sadržanog u djeljeniku
  - Dijelitelj B pomicemo u lijevo sve dok vrijedi  $A >= 2B$  i MSB(B)=0 te pamtimo **broj\_pomaka**
  - Ponavlja se dok  $A >= 2B$  zato da pronađemo **najveći** višekratnik
  - Prije ispitivanja  $A >= 2B$  provjeravamo da li je MSB(B)=1
    - Ako MSB(B)=1 tada smo sigurni da je to najveći višekratnik jer bi množenjem s 2 dobili broj koji je veći od opsega brojeva koji se mogu prikazati u 32 bita što bi sigurno bilo veće od djeljenika
  - U slučaju MSB(B)=1 moramo preskočiti ispitivanje  $A >= 2B$  jer bi usporedba mogla dovesti do krivog rezultata (B pomaknuto u lijevo za jedan bit prelazi 32 bita registra)
- Drugi korak:** Nakon što od djeljenika oduzmemmo najveći višekratnik djelitelja, postupak se ponavlja **broj\_pomaka** puta (kao dijeljenje na papiru - vidi sljedeći slajd)

© Kovač, Basch, FER, Zagreb

27

## Primjer programa za dijeljenje



Program za 32-bitno dijeljenje metodom pomaka, verzija 2

```

ADDR EQU 0x8100 ; adresa na kojoj se nalaze podatci

MOV R4, #ADDR ; postavlja u register R4 adresu podataka
LDR R5, [R4], #4 ; djeljenik (A) se učitava u R5
LDR R6, [R4], #4 ; djelitelj (B) se učitava u R6
MOV R7, #0 ; čisti se register za spremanje rezultata (R7)
MOV R8, #0 ; brojač inicijalnih pomaka
CMP R6, R5 ; ako je B>A, nema potrebe za dijeljenjem
BHI KRAJ

PORAVNAJ ;inicijalni korak: pronalaženje najvećeg višekratnika djelitelja

ANDS R3, R6, #0x80000000 ; provjerava je li MSB djelitelja jednak 1
BNE DIV ; ako jeste, poravnanje je nepotrebno
CMP R5, R6, LSL #1 ; provjerava je li A >= 2*B
MOVHS R6, R6, LSL #1 ; ako jeste, pomicne B u lijevo
ADDHS R8, R8, #1 ; povećava brojač pomaka
BHI PORAVNAJ ; samo ako je A>2*B, poravnanje se nastavlja

```

© Kovač, Basch, FER, Zagreb

29

28

## Primjer programa za dijeljenje (2. dio)



DIV

```

CMP R5, R6 ; uspoređuje trenutni ostatak i B
SUBHS R5, R5, R6 ; ako je ostatak >= B, onda ga umanji za B
ADDHS R7, R7, #1 ; i poveća rezultat za 1
CMP R8, #0 ; ako je brojač pomaka > 0, nastavi, a inače KRAJ
MOVHI R6, R6, LSR #1 ; pomakne B udesno,
MOVHI R7, R7, LSL #1 ; a rezultat ulijevo
SUBHI R8, R8, #1 ; umanji brojač pomaka
BHI DIV ; i ponovi petlju

```

KRAJ

```

STR R5, [R4], #4 ; spremanje ostatka
STR R7, [R4] ; spremanje rezultata
*** *

```

© Kovač, Basch, FER, Zagreb

30

## Napomena za upotrebu zastavice C



Primjer jednog dijela programa:

- ```

CMP R7,R6
MOVHS R7,R6

• Uvjet HS je po definiciji istinit ako je C=1. Iz toga slijedi da će se naredba MOVHS izvršiti ako je C=1.
• Ako je R7>R6, kako je moguće da je tada C=1? Naime, mi smo za FRISC rekli da se kod oduzimanja u zastavicu C upisuje bit posudbe koji je komplement bita prijenosa.
• Kod procesora ARM za naredbe oduzimanja i usporedbe se zastavica C postavlja ovako:
  C Flag = NOT BorrowFrom(Rn - shifter_operand)

te se tako ona i postavlja i sve je OK. Naime, kod ARM-a je C zastavica uvijek prijenos, a ne posudba bez obzira na operaciju koja se obavlja. Kod FRISC-a je drugačije definirano pa je će prilikom oduzimanja u zastavici C biti posudba.
```

© Kovač, Basch, FER, Zagreb

31

32

... nastavak

```

ISPIS MOV R1, #0x8200 ; učitavanje početne adrese
POM CMP R0, #0
BEQ KRAJ
LDMFD SPI, {R2}
STR R2, [R1], #4 ; prema početku originalnog niza
SUB R0, R0, #1 ; smanjujemo brojač
B POM

KRAJ
MOV R0, #0x18 ;
LDR R1, =0x20026 ;
SWI 0x123456 ; Normalan završetak simulacije

END

```

Ispitivanje specijalnog broja...

```

; Ispitivanje je li broj specijalan
; specNum.s
; Napisati program koji ispituje je li troznamenasti dekadski broj 'xyz' zapisan
; kao niz ASCII znamenaka (tzv. string) "xyz\0" specijalan broj za kojeg vrijedi:
; xyz = xy * xy - z * z, gdje su x y i z znamenke stotice, desetice i jedinice.
; Primjer takvog broja su brojevi: 100 = 10*10 - 0*0 i 147 = 14*14 - 7*7
; Broj zapisan kao niz ASCII znamenaka sa završnim null-znakom nalazi se na
; adresi koja je označena labelom 'num'. Ukoliko broj zadovoljava gornji uvjet,
; tada je potrebno u register r1 staviti sve jedinice, a ako uvjet nije ispunjen,
; tada u r1 treba staviti sve nule.
;
```

© Kovač, Basch, FER, Zagreb

33

© Kovač, Basch, FER, Zagreb

34

Specbroj...

```

AREA primjer, CODE, READONLY
ENTRY

; ovaj odsječak vadi znamenke iz ASCII zapisa: u r1 će se nalaziti znamenka
; stotica, u r2 će se nalaziti znamenka desetica, a u r3 će biti znamenka jedinica
main    ADR r0, num
        LDRB r1, [r0], #1
        LDRB r2, [r0], #1
        LDRB r3, [r0]
        SUB r1, r1, #48      ; znamenka stotica (48 je ASCII znak od 0)
        SUB r2, r2, #48      ; desetice
        SUB r3, r3, #48      ; jedinice

; množenje s konstantom 100 ostvareno je kombinacijom naredaba ADD i MOV
; s pomakom, što je efikasnije od korištenja naredbe MUL (multiply)
        ADD r5, r1, r1, LSL #3   ; r5 = r1+8*r1 = 9*r1
        ADD r5, r5, r1, LSL #4   ; r5 = r5+16*r1 = 9*r1+16*r1 = 25*r1
        MOV r5, r5, LSL #2      ; konačno r5 = 4*r5 = 4*25*r1 = 100 * r1

```

© Kovač, Basch, FER, Zagreb

35

Specbroj...

```

; množenje s konstantom 10 pomoću kombinacije naredaba ADD i MOV s
; pomakom
        MOV r6, r2
        ADD r6, r6, r6, LSL #2
        MOV r6, r6, LSL #1      ; r6 = 10 * r2

        ADD r7, r5, r6
        ADD r7, r7, r3      ; konačan broj u binarnom zapisu, potreban za ispitivanje
                           ; uvjeta zadatka xyz = xy*xy - z*z

; generiranje broja xy a nakon toga i broja xy*xy te z*z
        ADD r1, r1, r1, LSL #2
        MOV r1, r1, LSL #1
        ADD r1, r1, r2      ; u r1 se nalazi broj xy
        MUL r4, r1, r1      ; u r4 se nalazi broj xy*xy
        MUL r5, r3, r3      ; u r5 se nalazi broj z*z

```

© Kovač, Basch, FER, Zagreb

36

Specbroj...

```

SUB r4, r4, r5      ; r1 = xy*xy - z*z
CMP r4, r7
MVNEQ r1, #0          ; sve jedinice u R1
MOVNE r1, #0          ; sve nule u R1

MOV r0, #0x18
LDR r1, =0x20026
SWI 0x123456          ; Normalan završetak simulacije

num   = "147", 0      ; Zapis stringa u ADS-u
END

```

© Kovač, Basch, FER, Zagreb

37

© Kovač, Basch, FER, Zagreb

38

Specbroj...

Komentari:

Na početku programa pokazano je korištenje pseudonaredbe "ADR reg, labela" koja u zadani registar stavlja adresu zadanu labelom

Množenje kombinacijom ALU-operacija može se ostvariti kad množimo s konstantom. Obično se može ostvariti na više načina. Na primjer, množenje sa 100:

100 = ((1+8)+16)*4	(iz primjera: 3 naredbe)
100 = 128-32+4	(3 naredbe)
100 = (1+32)*3+1	(3 naredbe)
100 = 64+32+4	(3 naredbe)
100 = (1+16)*3*2-2	(4 naredbe)

Dio ARM5: Korištenje potprograma

Ovaj dokument namijenjen je isključivo za osobnu upotrebu studentima Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

Svaka druga upotreba ili umnožavanje bilo kojeg dijela ovog dokumenta nije dozvoljena bez pismene dozvole autora.

© Mario Kovač, Danko Basch



© Kovač, Basch, FER, Zagreb

1

© Kovač, Basch, FER, Zagreb

2

Primjer atoh_v1.s

Zadatak:

Napisati program koji 32-bitni heksadekadski broj zapisan u ASCII kodu (niz od 8 ASCII znakova) pretvara u broj. Znakovи se nalaze u memoriji od adrese 8100₁₆, a rezultat treba spremiti u memoriju iza znakova.

Za pretvorbu ASCII heksadekadske znamenke u njenu brojevnu vrijednost treba koristiti potprogram. Potprogram prima ASCII znamenku preko R0 i vraća rezultat preko R0. Prepostavka je da je heksadekadska ASCII znamenka ispravna, tj. da može sadržati dekadske znamenke od "0" do "9", mala i velika slova od "a" do "f".

Podsjetnik za ASCII kodove:

'0'	48	'A'	65	'a'	97
'1'	49	'B'	66	'b'	98
...
'9'	57	'F'	70	'f'	102

© Kovač, Basch, FER, Zagreb

3

Primjer atoh_v1.s

```

ADDR EQU 0x8100      ; adresa na kojoj se nalaze podatci

GLAVNI MOV R4, #ADDR
                  MOV R7, #8
                  MOV R6, #0      ; postavlja R4 na početak podataka
                           ; brojač hex znamenki = 8
                           ; resetiranje rezultata

PETLJA
        LDRB R0, [R4], #1      ; učitavanje ASCII-znaka
        BL FUNC_ATOH          ; poziv potprograma za pretvorbu
        ADD R6, R0, R6, LSL #4  ; spremanje rezultata iz R0 u registar R6 uz
                           ; pomak trenutačnog rezultata za 4 bita

        SUBS R7, R7, #1          ; brojač prolaza petlje
        BHI PETLJA

        STR R6, [R4]            ; spremanje rezultata

KRAJ  MOV R0, #0x18
      LDR R1, =0x20026
      SWI 0x123456          ; angel_SWIreason_ReportException
                           ; ADP_Stopped_ApplicationExit
                           ; ARM semihosting SWI

```

© Kovač, Basch, FER, Zagreb

4

Primjer atoh_v1.s

```

FUNC_ATOH      ; potprogram za pretvorbu ASCII-znaka u HEX
                ; ulazni ASCII-znak je u R0, a rezultat je također u R0

SUB R0, R0, #48 ; ASCII:0 = DEC: 48
CMP R0, #10    ; ako je broj 10 ili veći, znači da se radi o slovu
MOVLO PC,LR    ; povratak iz potprograma ako je R0<10

SUB R0, R0, #7  ; ASCII:A = DEC: 65
                ; treba oduzeti 65-55=10
                ; ranije smo već oduzeli 48, pa trebamo još 7 (48+7=55)
CMP R0, #16    ; ako je broj 16 ili veći, znači da se radi o malom slovu
MOVLO PC,LR    ; povratak iz potprograma ako je R0<16

SUB R0, R0, #32 ; ASCII:a = DEC: 97
                ; treba oduzeti 97-87=10
                ; ranije smo već oduzeli 55, pa trebamo još 32 (55+32=87)
MOV PC, LR     ; povratak iz potprograma

```

Napomena: potprogram ne mijenja registre pa nema spremanja konteksta

Poziv potprograma s parametrima na stogu

Potprogram računa $f(X,A,B) = (X \ll A) + B$, parametri i rezultat se prenose stogom. Parametre uklanja potprogram. Potprogram mijenja registre (LOŠE RIJEŠENJE)!!!

```

GLAVNI MOV R13, #0x8100      ; definicija vrha stoga          (1)
        MOV R0, #1            ; parametar X=1
        MOV R1, #4            ; parametar A=4
        MOV R2, #3            ; parametar B=3
        STMF D R13!, {R0, R1, R2} ; parametri se spremaju na stog   (2)
        BL FUNC_PARAM         ; poziv potprograma
        LDMFD R13!, {R0}       ; rezultat se sa stoga učitava u R0 (5)
        DALJE ...             ; nastavak glavnog programa

        FUNC_PARAM           ; potprogram
        LDMFD R13!, {R4, R5, R6} ; učitavanje parametara X,A,B   (3)
        ADD R4, R6, R4, LSL R5 ; računanje funkcije
        STMF D R13!, {R4}       ; spremanje rezultata          (4)
        MOV PC, LR             ; povratak iz potprograma

```

Poziv potprograma s parametrima na stogu

Potprogram računa $f(X,A,B) = (X \ll A) + B$ kao i prije, ali uz čuvanje registara (DOBRO RIJEŠENJE). Parametri se prenose stogom i uklanjanjem ih pozivatelj. Rezultat se vraća u R0.

```

GLAVNI MOV R13, #0x8100      ; definicija vrha stoga          (1)
        MOV R0, #1            ; parametar X=1
        MOV R1, #4            ; parametar A=4
        MOV R2, #3            ; parametar B=3
        STMF D R13!, {R0, R1, R2} ; parametri se spremaju na stog   (2)
        BL FUNC_PARAM         ; poziv potprograma
        ADD R13, R13, #12      ; ukloni parametre sa stoga
        DALJE STR R0, REZ      ; spremi rezultat i nastavi
        ...

        FUNC_PARAM           ; potprogram
        STMF D R13!, {R4, R5, R6} ; spremi registre
        ADD R0, R13, #12      ; sa R0 "preskoči" spremljene registre   (3)
        LDMFD R0, {R4, R5, R6} ; učitavanje ulaznih parametara X,A,B   (4)
        ADD R0, R6, R4, LSL R5 ; računanje funkcije
        LDMFD R13!, {R4, R5, R6} ; obnovi registre
        MOV PC, LR             ; povratak iz potprograma

```

Poziv potprograma s parametrima iza poziva

Potprogram računa $f(X,A,B)=(X \ll A) + B$. Parametri A i B prenose se lokacijom iza naredbe, a parametar X je u registru R0. Rezultat se vraća preko R0. Potprogram mijenja registre (LOŠE RIJEŠENJE)!!!

```

GLAVNI MOV R0, #1            ; parametar X=1
        BL FUNC_PARAM        ; poziv potprograma
        DCD 4                ; parametar A=4
        DCD 3                ; parametar B=3
        DALJE ...             ; nastavak programa, rezultat je u R0

        FUNC_PARAM           ; potprogram, adresa prvog parametra je u R14
        LDR R1, [R14], #4      ; prvi parametar (A) se učitava u R1, a R14 se
                                ; poveća tako da pokazuje na drugi parametar
        LDR R2, [R14], #4      ; drugi parametar (B) se učitava u R2, a R14 se
                                ; poveća tako da je u njemu povratna adresa
        ADD R0, R2, R0, LSL R1 ; računanje funkcije
        MOV PC, LR             ; povratak iz potprograma

```

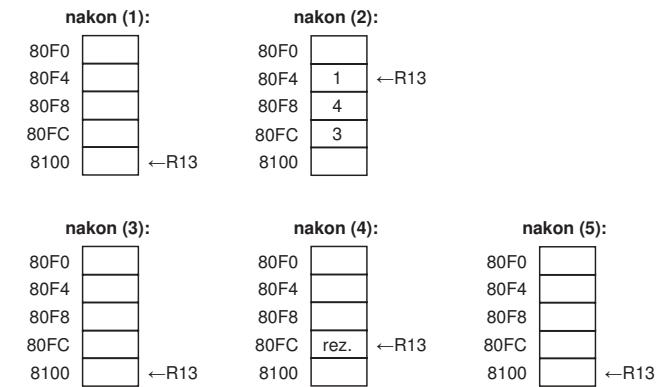
Prijenos parametara

- preko registara*
 - problem sa brojem registara koji su na raspolaganju
 - isto kao kod FRISC-a
- preko stoga
 - često najpraktičnije rješenje
- parametri iza naredbe poziva
- preko fiksnih memorijskih lokacija
 - isto kao kod FRISC-a

* pokazano u prethodnom primjeru

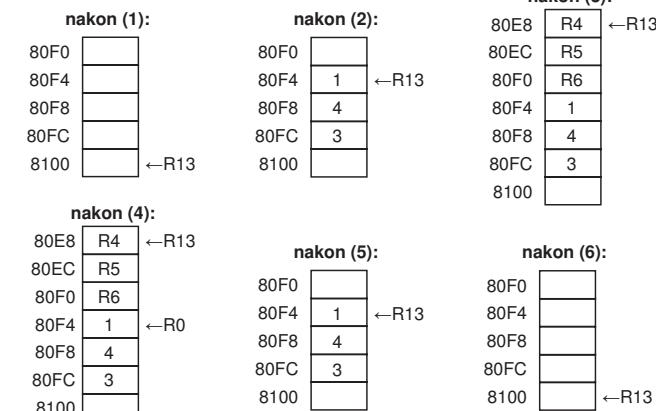
Poziv potprograma s parametrima na stogu

Izgled stoga:



Poziv potprograma s parametrima na stogu

Izgled stoga:



Poziv potprograma s parametrima iza poziva

Potprogram računa $f(X,A,B)=(X \ll A) + B$, parametri A i B su u lokacijama iza naredbe, a parametar X je u registru R0. Rezultat se vraća preko R0. Potprogram čuva registre (DOBRO RIJEŠENJE).

```

GLAVNI MOV R13, #0x8100      ; definicija vrha stoga
        MOV R0, #1            ; parametar X=1
        BL FUNC_PARAM        ; poziv potprograma
        DCD 4                ; parametar A=4
        DCD 3                ; parametar B=3
        DALJE ...             ; nastavak programa, rezultat je u R0

        FUNC_PARAM           ; potprogram, adresa prvog parametra je u R14
        STMF D R13!, {R1, R2}  ; spremi registre
        LDR R1, [R14], #4      ; prvi parametar (A) se učitava u R1, a R14 se
                                ; poveća tako da pokazuje na drugi parametar
        LDR R2, [R14], #4      ; drugi parametar (B) se učitava u R2, a R14 se
                                ; poveća tako da je u njemu povratna adresa
        ADD R0, R2, R0, LSL R1 ; računanje funkcije
        LDMFD R13!, {R1, R2}   ; obnovi registre
        MOV PC, LR             ; povratak iz potprograma

```

Ugniježđeni pozivi

- Problem: R14 služi za spremanje povratne adrese
 - već prvi ugniježđeni poziv uništava početni R14
- Jedino dobro rješenje je spremanje R14 na stog na početku potprograma te čitanje povratne adrese sa stoga prije povratka iz potprograma
- Primjeri:
 - Poziv potprograma iz potprograma
 - Rekursivna funkcija

© Kovač, Basch, FER, Zagreb

13

Primjer računanja rekursivne funkcije (faktorijela)

Napisati potprogram za računanje faktorijele korištenjem rekursivnog poziva. Parametar funkcije neka se prenosi preko registra R0, a rezultat funkcije neka se vraća u registru R1.

Glavni program treba izračunati faktorijela(6). Potprogram treba čuvati registre.

```
int faktorijela (int x) {
    if( x == 1)
        return (1);
    return ( faktorijela(x-1) * x );
}
```

© Kovač, Basch, FER, Zagreb

15

Stanje stoga tijekom rekursije u programu

f(6)	f(5)	f(4)	f(3)	f(2)	f(1)
8100					
80FC	800C	800C	800C	800C	800C
80F8	6	6	6	6	6
80F4		8038	8038	8038	8038
80F0	5	5	5	5	5
80EC		8038	8038	8038	8038
80E8		4	4	4	4
80E4			8038	8038	8038
80E0			3	3	3
80DC				8038	8038
80D8				2	2
80D4					8038
80D0					1

© Kovač, Basch, FER, Zagreb

17

...glavni program

Primjer je napisan korištenjem sintakse za ATLAS

```
'ORG 0
GLAVNI MOV R13, #1<16 ; inicijalizacija pokazivača stoga
MOV R0, #9 ; Prva točka je (9, 3)
MOV R1, #3
STMD R13!, {R0, R1} ; koordinate prve točke se stavljaju
; kao parametri na stog

MOV R0, #5 ; Druga točka je (5, 8)
MOV R1, #8
STMD R13!, {R0, R1} ; koordinate druge točke se stavljaju
; kao parametri na stog

BL MANH
ADD R13, R13, #%D16 ; uklanjuju se parametri sa stoga
; rezultat je u R7

SWI 123456 ; kraj (može se pisati i HALT)
```

© Kovač, Basch, FER, Zagreb

Primjer poziva potprograma iz potprograma

Treba napisati funkciju $c(x)=(x/2) + 3$, ali tako da se koriste pomoćne funkcije $a(x)=x/2$ i $b(x)=x+3$. Potprogrami trebaju čuvati registre.

```
glavni MOV R13, #0x8000 ; inicijalizacija SP
MOV R0, #7 ; parametar x=7
BL FUNC_C ; poziv potprograma
...
FUNC_A MOV R0, R0, ASR #1 ; R0= R0/2
MOV PC, LR ; povratak iz potprograma

FUNC_B ADD R0, R0, #3 ; R0 = R0+3
MOV PC, LR ; povratak iz potprograma

FUNC_C STMD r13!, {R0,R14} ; R1 = R0/2 + 3
BL FUNC_A ; x=x/2
BL FUNC_B ; x=x/2 +3
MOV R1, R0 ; rezultat stavi u R1
LDMFD r13!, {R0,R14}
MOV PC, LR ; povratak iz potprograma
```

© Kovač, Basch, FER, Zagreb

14

Primjer računanja rekursivne funkcije (faktorijela)

```
GLAVNI MOV R13, #0x8100 ; vrh stoga
MOV R0, #6 ; ulazni parametar=6
BL FUNC_FACT ; rezultat je u R1
KRAJ MOV R0, #0x18
LDR R1, =0x20026
SWI 0x123456 ; ARM semihosting SWI

FUNC_FACT STMD R13!, {R0, LR} ; spremanje registara na stog
CMP R0, #1
BNE X_NIJE_1 ; ako x != 1 onda rekurzija

X_JE_1 MOV R1, #1 ; inicijalna vrijednost faktorijele
LDMFD R13!, {R0, LR} ; obnovi registre sa stoga
MOV PC, LR ; povratak iz potprograma za x==1

X_NIJE_1 SUB R0, R0, #1 ; R0 = x-1
BL FUNC_FACT ; R1 = f(x-1)
ADD R0, R0, #1 ; R0 = x
MUL R1, R0, R1 ; izračunaj R1 = x*f(x-1)
LDMFD R13!, {R0, LR} ; obnovi registre sa stoga
MOV PC, LR ; povratak iz potprograma
```

© Kovač, Basch, FER, Zagreb

16

Primjer Manhattan

- Poziv potprograma iz potprograma
- Prijenos parametara preko stoga
- Čuvaju se registri

Zadatak:

Napisati potprogram MANH koji računa Manhattan-udaljenost dvije točke: (x_1, y_1) i (x_2, y_2) . Manhattan-udaljenost definirana je formulom:

$$|x_2 - x_1| + |y_2 - y_1|$$

Potprogram MANH preko stoga prima koordinate točaka x_1, y_1, x_2, y_2 kao parametre, a rezultat vraća u registru R7. Apsolutna vrijednost razlike dva broja računa se posebnim potprogramom ADIFF koji parametre prima preko stoga i rezultat vraća u registru R8.

© Kovač, Basch, FER, Zagreb

18

... potprogram MANH

```
MANH
STMD R13!, {R0,R1,R2,R3,R4,R8,R14} ; registri koji se koriste moraju se sačuvati
ADD R0, R13, #%D28 ; R0 preskače nove podatke na stogu
LDMFD R0, {R1, R2, R3, R4} ; učitavaju se pohranjeni parametri
; R1=x2,R2=y2,R3=x1,R4=y1

STMD R13!, {R1, R3} ; poziva se ADIFF(R1,R3)
BL ADIFF ; uklanjuju se parametri sa stoga
ADD R13, R13, #8 ; rezultat iz r8 sprema se u r1

STMD R13!, {R2, R4} ; poziva se ADIFF(R2,R4)
BL ADIFF ; uklanjuju se parametri sa stoga
ADD R13, R13, #8 ; rezultat iz r8 sprema se u r2

ADD R7, R1, R2 ; zbrajaju se dvije apsolutne razlike

LDMFD R13!, {R0,R1,R2,R3,R4,R8, R14} ; obnavljaju se registri
MOV PC, LR
```

© Kovač, Basch, FER, Zagreb

20

19

... potprogram ADIFF

ADIFF

```
STMF D R13!, {R0, R1, R2} ; registri koji se koriste moraju se sačuvati
ADD R0, R13, #%D12 ; R0 preskače nove podatke na stogu
LDMFD R0, {R1,R2} ; učitavaju se pohranjeni parametri
SUBS R8, R1, R2 ; izračunavanje razlike i postavljanje zastavica

RSBLT R8, R8, #0 ; ako je R8 negativan, onda R8 = 0-R8 = -R8

LDMFD R13!, {R0, R1, R2} ; obnavljaju se pohranjeni registri
MOV PC, LR
```

© Kovač, Basch, FER, Zagreb

21

Sažimanje

Za procesor ARM napisati potprogram SAZMI koji blok od 25 32-bitnih podataka sažima na sljedeći način: Prvi podatak iz izvornog bloka se prepisuje u blok sa sažetim podacima te služi kao bazni broj za sljedeća 24 podatka. Ostatak bloka sa sažetim podacima sadrži 8-bitne brojeve u formatu dvojnog komplementa koji predstavljaju razliku između izvornog podatka i baznog broja.

Pretpostavka je da će razlika uvijek biti u opsegu -128 do 127. Primjer za 4 podatka bio bi sljedeći: izvorni niz: 00000050, 00000052, 0000005A, 00000048; sažeti niz: 00000050, 02, 0A, -08.

Adresa bloka kojeg treba sažeti prenosi se preko R0, dok se adresa mesta od kojeg treba pohraniti rezultat prenosi preko R1. Registri R0 i R1 se mijenjaju tako da pokazuju na podatke koji još nisu obradeni, tako da se potprogram može pozivati više puta da bi se obradio veći blok podataka.

Napisati glavni program koji pomoću potprograma SAZMI sažima blok od 100 podataka koji počinje na adresi 2000 te sažeti niz pohranjuje od adrese 500. Niz se sažima pozivom potprograma SAZMI, za svakih 25 podataka pojedinačno (ukupno 40 poziva).

© Kovač, Basch, FER, Zagreb

2

Palindrom

Napisati program koji ispituje je li zadani string palindrom (jednako se čita i straga i sprjeda). Ukoliko string jeste palindrom, onda u registar r8 treba upisati sve jedinice, a ako string nije palindrom, u registar r8 treba upisati sve nule.

Neka string započinje na adresi koja je označena labelom 'table', a kraj stringa je označen sa 0 (tj. Sa NUL-znakom). Pri ispitivanju je li dani string palindrom, završni NUL-znak ne uzima se u obzir.

© Kovač, Basch, FER, Zagreb

4

Block count

Napisati potprogram koji broji koliko blokova binarnih jedinica ima u 32-bitnoj riječi prenesenoj preko registra R0. Rezultat vratiti u R0.

Na primjer, sljedeći broj:

0101 1111 0011 0011 1111 1111 1111 0000
1 2 3 4

ima 4 bloka binarnih jedinica.

U glavnom programu potrebno je u bloku podataka koji počinje od adrese 0x100 za svaki podatak odrediti broj blokova binarnih jedinica. Dobivene rezultate spremi od adrese 0x200. Blok podataka zaključen je s brojem 0.

Primjeri zadataka

Zadaci za vježbu

Ovaj dokument namijenjen je isključivo za osobnu upotrebu studentima Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

Svaka druga upotreba ili umnožavanje bilo kojeg dijela ovog dokumenta nije dozvoljena bez pismene dozvole autora.

© Mario Kovač, Danko Basch



© Kovač, Basch, FER, Zagreb

1

Sažimanje: rješenje

```
AREA sazmi_1, CODE, READONLY
ENTRY
    POD EQU 0x2000 ; izvorni blok
    SAZ EQU 0x500 ; sažeti blok

main
    MOV R0, #POD
    MOV R1, #SAZ
    MOV R2, #40 ; brojač
    MOV R13, #0x9000 ; stog

petija
    BL sazmi ; sažmi dio
    SUBS R2, R2, #1 ; ponavljaj
    BNE petija ; 40 puta

kraj
    MOV R0, #0x18
    LDR R1, =0X20026
    SWI 0x123456

    PET
        LDR R3, [R0], #4 ; čitaj podatak
        SUB R3, R3, R2 ; razlika od baze
        STRB R3, [R1], #1 ; spremi je

        SUBS R4, R4, #1
        BNE PET

        LDMFD R13!, {R2, R3, R4} ;obnavljanje
        MOV PC, LR

    END
```

© Kovač, Basch, FER, Zagreb

3

Palindrom: rješenje

```
main
    ADR r0, table ; adresa stringa u r0
    MOV r1, #0 ; brojač znakova

    ; pro računamo duljinu stringa
    ; zajedno s NUL-znakom
count
    LDRB r2, [r0], #1
    CMP r2, #0 ; postavlja Z
    ADD r1, r1, #1 ; ne mijenja Z
    BNE count

    ; vrati r0 na početak stringa
    SUB r0, r0, r1

    ; r1 je odmak od r0 do zadnjeg znaka
    ; (ne računajući NUL-znak)
    SUB r1, r1, #2

    ; r3 je odmak od r0 do prvog znaka
    MOV r3, #0

    test
        LDRB r4, [r0, r3] ; znak spreda
        LDRB r5, [r0, r1] ; znak straga
        CMP r4, r5 ; jesu li isti?
        BNE nisu_isti

        ADD r3, r3, #1 ; znak naprijed
        SUB r1, r1, #1 ; znak natrag
        CMP r3, r1 ; srednji znak?
        BLO test

    nisu_isti
        MVN r8, #0 ; string je palindrom
        B kraj

    kraj
        MOV R0, #0x18
        LDR R1, =0X20026
        SWI 0x123456

    table = "neven", 0 ; string
```

© Kovač, Basch, FER, Zagreb

5

Block count: rješenje (za ATLAS)

```
'ORG 0
GLAVNI
    MOV R13, #10<12 ; stog

    MOV R1, #1<8 ; adresa podataka koje treba obraditi
    MOV R2, #2<8 ; adresa rezultata

L1
    LDR R0, [R1], #4 ; učitaj podatak
    CMP R0, #0 ; provjeri je li to oznaka kraja
    BEQ KRAJ
    BL BLCNT ; obradi podatak potprogramom BLCNT
    STR R0, [R2], #4 ; spremi rezultat
    B L1

KRAJ HALT

    ; podaci koje treba obraditi (5 podataka + oznaka kraja)
    'ORG 100
    DW 0802, OFFFFFFFF, OFOFOFOF
    DW %B 101110111111111000000100011101,
```

BLCNT STMD R13!, {R1, R2, R3, R4}

```

MOV R4, #%D32 ; brojač za petlju
MOV R2, #0 ; pamti vrstu prethodnog bloka (početno blok 0)
MOV R3, #0 ; brojač blokova jedinica

LOOP
    ANDS R1, R0, #1 ; ispituj bit po bit u r0
    BEQ ZERO ; je li tekući bit 0 ili 1
    ONE
        CMP R2, #0 ; kakav je bio prethodni blok
        ADDEQ R3, R3, #1 ; bio je blok 0: povećaj brojač blokova jedinica
        MOVEQ R2, #1 ; bio je blok 0: zapamti vrstu bloka (blok 1)
        B DALJE
    ZERO
        MOV R2, #0 ; tekući bit je 0
        MOV R3, #0 ; zapamti vrstu bloka (blok 0)

DALJE
    MOV R0, R0, LSR #1 ; pomakni se na sljedeći bit
    SUBS R4, R4, #1 ; smanji brojač bitova (tj. brojač petlje)
    BNE LOOP ; stavi povratnu vrijednost u R0
    MOV R0, R3 ; stavi povratnu vrijednost u R0

LDMFD R13!, {R1, R2, R3, R4}
MOV PC,LR

```

© Kovač, Basch, FER, Zagreb

8

Iznimke

© Kovač, Basch, FER, Zagreb

2

Iznimke: adrese potprograma i prioriteti

Tip iznimke	ARM se prebacuje u način rada	Adresa potprograma*	Prioritet
Reset	Supervisor	0x00000000	1
Undefined instruction	Undefined	0x00000004	6
Software interrupt (SWI)	Supervisor	0x00000008	6
Prefetch Abort (instruction fetch memory abort)	Abort	0x0000000C	5
Data Abort (data access memory abort)	Abort	0x00000010	2
IRQ (interrupt)	IRQ	0x00000018	4
FIQ (fast interrupt)	FIQ	0x0000001C	3

* Uočite da je za svaki potprogram na raspolaganju samo 4 bajta, pa se tu u pravilu stavlja naredba skoka na odsječak za obradu iznimke (osim potprograma na adresi 0x1C)

© Kovač, Basch, FER, Zagreb

4

Reset

- Povratak iz iznimke Reset nije predviđen. U slučaju inicijalizacije procesora (npr. uključivanje napajanja) očekuje se da će vanjska logika aktivirati signal Reset kako bi procesor normalno započeo s radom.
- Prema tome, svaki program mora pretpostaviti obradu iznimke Reset i nakon toga prelazak na izvođenje izabrane aplikacije.

Dio ARM6: Računalni sustav s procesorom ARM

Ovaj dokument namijenjen je isključivo za osobnu upotrebu studentima Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

Svaka druga upotreba ili umnožavanje bilo kojeg dijela ovog dokumenta nije dozvoljena bez pismene dozvole autora.

© Mario Kovač, Danko Basch



© Kovač, Basch, FER, Zagreb

1

Obrada iznimaka

- U ARM-u se **iznimkama** nazivaju razne vrste događaja koje ne ulaze u normalno slijedno izvođenje naredaba. Na primjer: pojавa prekida, dohvata neispravnog strojnog koda, resetiranje procesora, itd.
- Procesor ARM načelno obrađuje iznimke na sljedeći način:
 - ARM pohranjuje R15 (programsко brojilo) u registar LR, a zatim pohranjuje CPSR (registar trenutnog programske stanja) u registar SPSR*
 - ARM prelazi u privilegirani način rada koji ovisi o vrsti iznimke
 - ARM skače u potprogram za obradu iznimke za dotični način rada
- Adrese potprograma za obradu iznimke su fiksno definirane za svaku pojedinu iznimku
- Ako se u isto vrijeme pojave dvije ili više iznimaka, ARM definira prioritete izvođenja

* osim za RESET

© Kovač, Basch, FER, Zagreb

3

Reset

- Slijed operacija koje se obave pri pojavi pojedine iznimke vrlo je sličan za sve iznimke
- Kada se na ulazu u procesor aktivira signal Reset, procesor odmah prekida izvođenje naredbe. Nakon što se Reset deaktivira, obavi se sljedeći niz operacija:

```

• R14_svc = UNPREDICTABLE value
• SPSR_svc = UNPREDICTABLE value
• CPSR[4:0] = 0b10011 /* Enter Supervisor mode */
• CPSR[5] = 0 /* Execute in ARM state */
• CPSR[6] = 1 /* Disable fast interrupts */
• CPSR[7] = 1 /* Disable normal interrupts */
• PC = 0x00000000

```

© Kovač, Basch, FER, Zagreb

5

IRQ (prekid)

- Ako je u programu omogućeno prihvatanje prekida (bit I u CPSR je obrisan), procesor će na kraju izvođenja svake naredbe provjeriti je li ulaz IRQ aktiviran. Ako je neki vanjski sklop aktivirao signal prekida (IRQ), procesor će nakon završetka trenutne naredbe obaviti sljedeći niz operacija:

```

• R14_irq = address of next instruction to be executed + 4
• SPSR_irq = CPSR
• CPSR[4:0] = 0b10010 /* Enter IRQ mode */
• CPSR[5] = 0 /* Execute in ARM state */
• /* CPSR[6] is unchanged */
• CPSR[7] = 1 /* Disable normal interrupts */
• PC = 0x00000018

```

IRQ

- Za povratak iz IRQ-potprograma treba izvesti naredbu:
SUBS PC, R14, #4
- Ova naredba* će obnoviti PC (iz R14_irq) i CPSR (iz SPSR_irq) te nastaviti izvođenje programa na mjestu gdje je prekinut

* SUB napisan s ekstenzijom S i odredišnim registrom PC, znači da treba obnoviti CPSR

FIQ

- Ako je u programu omogućeno prihvaćanje brzog prekida (bit F u CPSR je obrisan) procesor će na kraju izvođenja svake naredbe provjeriti da li je ulaz FIQ aktivan. Ako je na ulazu u procesor aktiviran signal brzog prekida (FIQ), procesor će nakon završetka trenutne naredbe obaviti sljedeći niz operacija:

```
R14_fiq = address of next instruction to be executed + 4
SPSR_fiq = CPSR
CPSR[4:0] = 0b10001      /* Enter FIQ mode */
CPSR[5] = 0                /* Execute in ARM state */
CPSR[6] = 1                /* Disable fast interrupts */
CPSR[7] = 1                /* Disable normal interrupts */
PC = 0x0000001C
```

Primjer obrade iznimke (verzija za ADS)

Glavni program treba povećavati registar r1. Kada dođe do prekida IRQ, treba postaviti oznaku da treba završiti glavni program.

```
GLAVNI    MRS r0, CPSR      ; pročitaj CPSR
          BIC r0, r0, #0x80   ; pobriši bit I (za bit F koristi se maska 0x40)
          MSR CPSR_c, r0     ; upiši promjenu u CPSR (dozvoli IRQ)

          MOV r0, #1          ; oznaka da treba ponavljati petlju

PETLJA    ADD r1, r1, #1      ; povećavaj r1 dok ne dođe prekid
          CMP r0, #0          ; provjeri oznaku kraja
          BNE PETLJA

          MOV r0, #0x18        ; angel_SWIreason_ReportException
          LDR r1, =0x20026    ; ADP_Stopped_ApplicationExit
          SWI 0x123456         ; ARM semihosting SWI

OBR_IRQ    ; Potprogram za obradu iznimke
          MOV R0, #0            ; oznaka da treba završiti petlju
          SUBS PC, R14, #4      ; povratak iz iznimke
```

; Na adresu 0x18 upisati 02 20 00 EA = B OBR_IRQ

Detalji...



- Zapis narebe B

[cond] [101] [L] [24-bitni podatak]

- U našem programu treba skočiti na labelu OBR_IRQ koja se nalazi na adresi 8028
- Naredba B se nalazi na adresi 0x18, no zbog protočne strukture procesora u trenutku izvođenja ove naredbe PC je već uvećan za 8 pa je PC=0x20
- Da bi skočili na 8028, trebamo PC-u dodati 0x8008, a kako se u naredbi zapisuje odmak podijeljen sa 4, tada je vrijednost koja se upisuje u binarni kod naredbe 0x2002.
- Iz toga slijedi da binarni zapis naredbe izgleda ovako:
[1110] [101] [0] [0x2002] ili 0xEA002002
- kada se to zapise u little endian formatu:
02 20 00 EA

FIQ (brzi prekid)

- Brzi prekid namjenjen je primjenama gdje je bitno brzo reagirati i obaviti niz operacija
- FIQ zato ima dovoljan broj 'privatnih' registara tako da ne treba obavljati operacije pohranjivanja i vraćanja konteksta na stog
- Pored toga, adresa prekidnog potprograma FIQ je namjerno zadњa na listi tako da se potprogram može napisati odmah od te adrese bez potrebe za skokom na drugo mjesto u memoriji (izbjegnuto je kašnjenje zbog takvog skoka)

FIQ

- Za povratak iz FIQ-potprograma treba izvesti naredbu (isto kao za IRQ):

SUBS PC, R14, #4

- Ova naredba će obnoviti PC (iz R14_fiq) i CPSR (iz SPSR_fiq) te nastaviti izvođenje programa na mjestu gdje je prekinut

Detalji...



- Zašto je na adresi 0x18 upisano 02 20 00 EA?
 - Zato što je to strojni kod naredbe B OBR_IRQ
- Format naredbe Branch (B) je
B <target address>
gdje je <target address> adresa koja se dobije iz 24 bitovne neposredne vrijednosti s predznakom upisane u samoj naredbi koja kada se pomnoži sa 4 predstavlja odmak od trenutne vrijednosti PC
- množenje sa 4 je zbog toga jer se sve naredbe uvijek nalaze na adresi djeljivoj s 4 pa se time postiže veći opseg mogućih skokova
- prema tome naredba B skače na adresu
PC + <odmak>*4

Primjer obrade iznimke (verzija za ATLAS)

```
'ORG 0
B GLAVNI
`ORG 18
B OBR_IRQ' } glavna razlika u odnosu na ADS

GLAVNI    MRS r0, CPSR      ; pročitaj CPSR
          BIC r0, r0, #80     ; pobriši bit I (za bit F koristi se maska 40)
          MSR CPSR_c, r0     ; upiši promjenu u CPSR (dozvoli IRQ)

          MOV r0, #1          ; oznaka da treba ponavljati petlju

PETLJA    ADD r1, r1, #1      ; povećavaj r1 dok ne dode prekid
          CMP r0, #0          ; provjeri oznaku kraja
          BNE PETLJA

          SWI 123456

OBR_IRQ    ; Potprogram za obradu iznimke
          MOV R0, #0            ; oznaka da treba završiti petlju
          SUBS PC, R14, #4      ; povratak iz iznimke
```

Računalni sustav s procesorom ARM

© Kovač, Basch, FER, Zagreb

16

AMBA

- AMBA, kratica za Advanced Microcontroller Bus Architecture
- Trenutno aktualne specifikacije su AMBA 4 (ACE, ACE-Lite, AXI4, AXI4-Lite, AXI4-Stream, AHB, APB, ATB)
- Mi ćemo u okviru ovog predmeta proučiti samo načelno dvije sabirnice koje su definirane još u AMBA2 specifikacijama:
 - AHB (Advanced High-performance Bus)
 - APB (Advanced Peripheral Bus).

© Kovač, Basch, FER, Zagreb

17

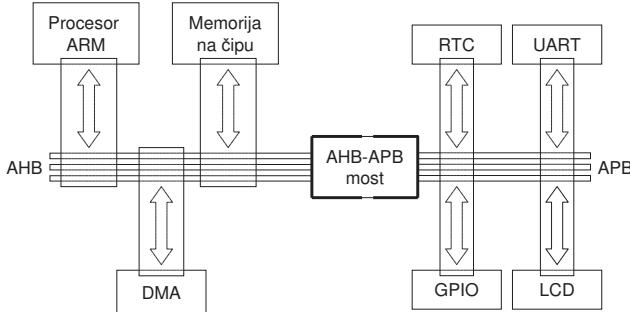
AMBA

- **AMBA AHB** je sabirnica namijenjena za sustave visokih performansi i visokih frekvencija signala vremenskog upravljanja i koristi se isključivo kao brza središnja (memorijska) sabirnica. AHB omogućuje efikasno povezivanje procesora, memorije koja se nalazi na čipu kao i vanjske memorije.
- **AMBA APB** je sabirnica za povezivanje vanjskih uređaja u sustav. Karakteristike ove sabirnice su mala potrošnja i jednostavnija izvedba u usporedbi sa središnjim sabirnicama, s ciljem što jednostavnijeg povezivanja vanjskih jedinica u cjelovit sustav. Na APB sabirnicu se povezuju vanjski uređaji koji ne zahtijevaju visoke performanse i nemaju kompleksna sučelja. Primjeri takvih uređaja su serijski kontroler (UART), LCD-kontroler, vremenski sklop (RTC) i paralelni sklop (GPIO).

© Kovač, Basch, FER, Zagreb

18

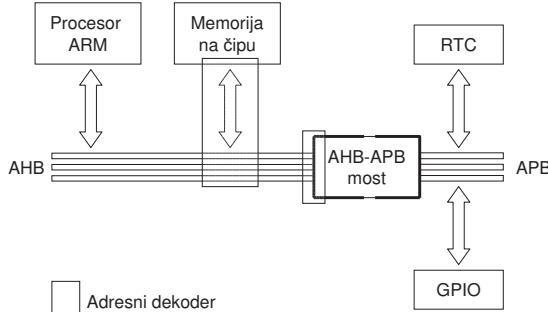
ARM sustav



© Kovač, Basch, FER, Zagreb

19

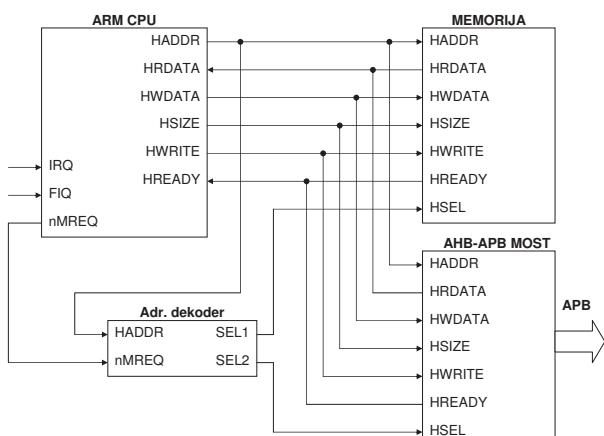
... i naš sustav u ATLAS-u



© Kovač, Basch, FER, Zagreb

20

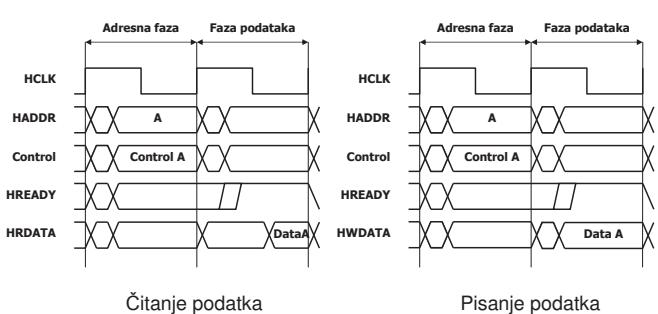
Središnji sustav sa sabirnicom AHB



© Kovač, Basch, FER, Zagreb

21

Adresna i podatkovna faza na AHB

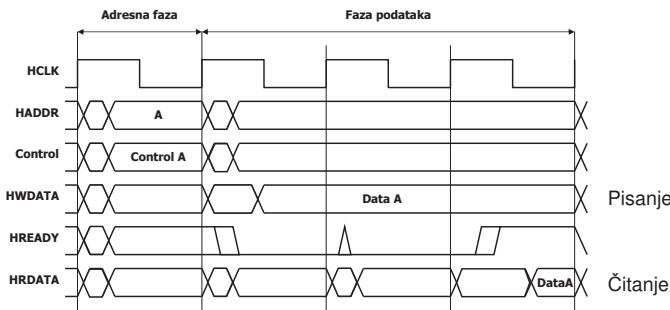


© Kovač, Basch, FER, Zagreb

22

© Kovač, Basch, FER, Zagreb

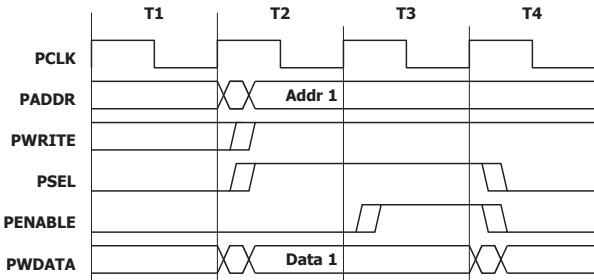
23



Sabirnički periodi na AHB

nMREQ	SEQ	Tip perioda	Opis
0	0	N-period	Neslijedni period (engl. Nonsequential)
0	1	S-period	Slijedni period (engl. Sequential)
1	0	I-period	Interni period (engl. Internal)
1	1	C-period	Period prijenosa sadržaja koprocesorskog registra (engl. Coprocessor register transfer)

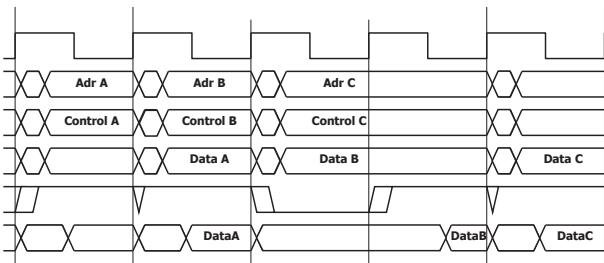
Period pisanja na APB



AHB APB most

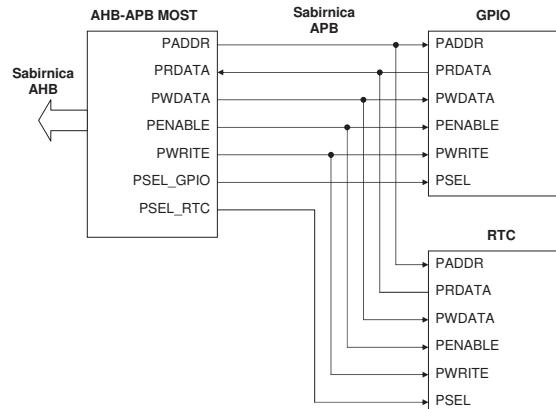
- Sabirnice AHB i APB predviđene su za različite tipove sklopova:
 - AHB je središnja, brza sabirница,
 - APB je jednostavnija sabirница namijenjena vanjskim sklopovima koji su obično sporiji.
- Most AHB-APB je sklop koji omogućuje povezivanje ovih sabirnica i uređaja na njima u cijelovit sustav te prijenos podataka između uređaja sa sabirnicama AHB i APB.

- Kod sabirnice AHB postoji **preklapanje** između adresne faze jednog pristupa i podatkovne faze prethodnog pristupa* čime se ubrzava komunikacija

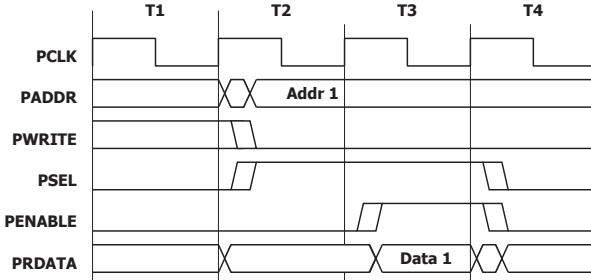


* slično ideji preklapanja faza u protočnoj strukturi

Sabirnica APB

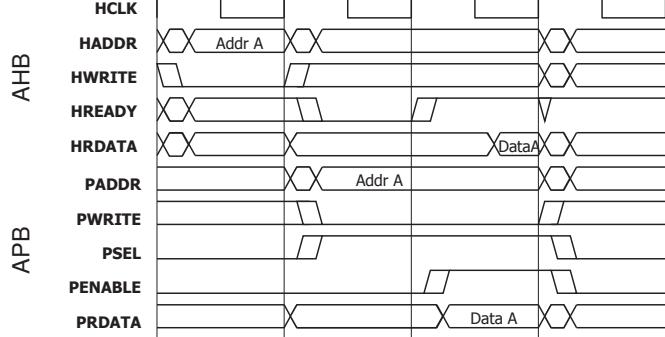


Period čitanja na APB



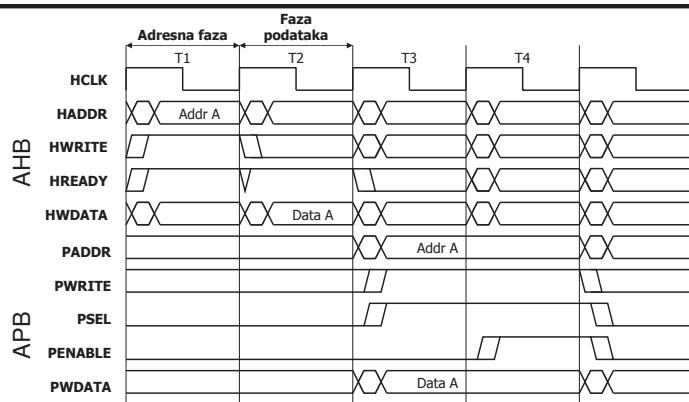
AHB APB most

- Funkcije mosta su sljedeće:
 - uzima adresu i održava je valjanom tijekom cijelog prijenosa
 - dekodira adresu i generira signal PSEL_x kojim se izabire jedna od vanjskih jedinica kojom se izvodi prijenos podataka
 - postavlja podatke na sabirnicu APB kod perioda pisanja
 - postavlja podatke sa sabirnice APB na sabirnicu AHB tijekom perioda čitanja
 - generira vremenski signal PENABLE kojim se omogućuje prijenos



© Kovač, Basch, FER, Zagreb

32

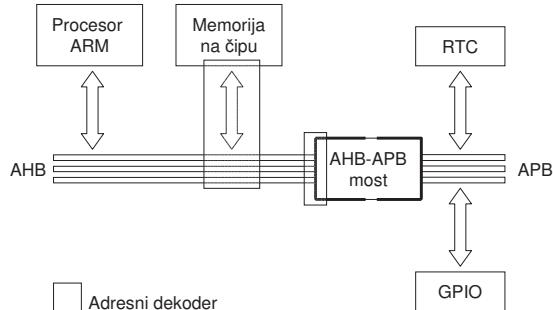


© Kovač, Basch, FER, Zagreb

33

Vanjske jedinice za procesor ARM

GPIO i RTC



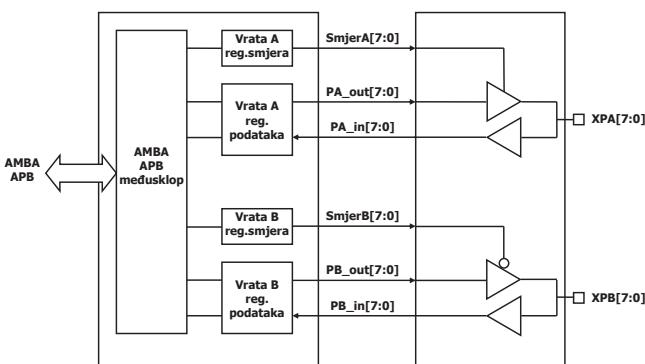
© Kovač, Basch, FER, Zagreb

34

© Kovač, Basch, FER, Zagreb

35

Sklop GPIO (General Purpose Input Output)



© Kovač, Basch, FER, Zagreb

36

Registri sklopa GPIO

Adresa	Naziv registra	Opis
GPIO_bazna_adr	GPIOPADR	8-bitni registar podataka, vrata A
GPIO_bazna_adr + 0x4	GPIOPBDR	8-bitni registar podataka, vrata B
GPIO_bazna_adr + 0x8	GPIOPADDR	8-bitni registar smjera podataka za vrata A
GPIO_bazna_adr + 0xC	GPIOPBDDR	8-bitni registar smjera podataka za vrata B

© Kovač, Basch, FER, Zagreb

37

Registri GPIO

- GPIOPADR (GPIO Port A Data Register)
 - 8-bitni registar podataka za vrata A.
 - Podatci upisani u ovaj registar postavljeni su na izlazne priključke ako je pripadni bit u registru smjera podataka (GPIOPADDR) postavljen u logiku nulu.
 - Čitanje ovog registra vraća trenutačno stanje priključaka koji su programirani kao ulazni. Za priključke programirane kao izlazne, vraća se vrijednost zadnjeg podatka upisanog u ovaj registar.
- GPIOPBDR (GPIO Port B Data Register)
 - 8-bitni registar podataka za vrata B.
 - Podatci upisani u ovaj registar postavljeni su na izlazne priključke ako je pripadni bit u registru smjera podataka (GPIOPBDDR) postavljen u logiku nulu.
 - Čitanje ovog registra vraća trenutačno stanje priključaka koji su programirani kao ulazni. Za priključke programirane kao izlazne, vraća se vrijednost zadnjeg podatka upisanog u ovaj registar.

© Kovač, Basch, FER, Zagreb

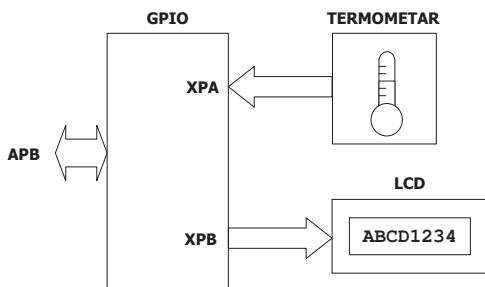
38

Registri GPIO

- GPIOPADDR (GPIO Port A Data Direction Register)
 - 8-bitni registar smjera podataka za vrata A. Vrijednost bita u ovom registru zadaje smjer odgovarajućeg priključka na vratima A:
 - 0 – ulazni smjer
 - 1 – izlazni smjer
- GPIOPBDDR (GPIO Port B Data Direction Register)
 - 8-bitni registar smjera podataka za vrata B. Vrijednost bita u ovom registru zadaje smjer odgovarajućeg priključka na vratima B:
 - 0 – izlazni smjer
 - 1 – ulazni smjer
- Početna vrijednost registara
 - Svi registri unutar GPIO-a nakon inicijalizacije (resetiranja) se postavljaju u logiku nulu. Ovime se inicijalno vrata A postavljaju kao ulazna, a vrata B kao izlazna. Sadržaji obaju registara podataka su nula.

© Kovač, Basch, FER, Zagreb

39



Temperaturni sklop

- Dva najviša bita (povezana na GPIO-priklučke XPA[6] i XPA[7]) služe za sinkronizaciju*. Bitovi 6 i 7 su aktivni u visokoj razini.
- Bit 6 ulazi u GPIO i pomoću njega temperaturni sklop signalizira da je izmjerio temperaturu i postavio njezinu vrijednost na nižih šest bitova. Kad procesor ustanovi da je stanje na bitu 6 u jedinici, može očitati temperaturu s nižih šest bitova.
- Nakon što je pročitao podatke, procesor pomoću izlaznog bita 7 dojavljuje temperaturnom sklopu da je vrijednost temperature pročitana i da treba izbrisati bit 6, sve do trenutka dok temperaturni sklop ne postavi novu vrijednost temperature te ponovo ne aktivira bit 6.

* GPIO nema priključke za sinkronizaciju pa je treba obaviti programski

Primjer: Čitanje temperature

Temperaturni sklop spojen je na vrata B sklopa GPIO. Na najniži bit vrata A spojena je tipka koja daje pozitivni impuls kad je pritisнута. GPIO je na adresi FFFF1000(16).

Napisati program koji očitava temperaturu svaki put kada se pritisne tipka i sprema je u memoriju na adresu 1000(16). Nakon 100(16) očitavanja zaustaviti procesor.

```

`ORG 0
GLAVNI
LDR R1, GPIO ; R1 = GPIO bazna adresa

; inicijalizacija GPIO
MOV R0, #0B 00000000 ; smjer vrata A, bit 0 je ulazni (tipka)
STR R0, [R1, #08]
MOV R0, #0B 01111111 ; smjer vrata B, bit 7 je izlazni, ostali su ulazni
STR R0, [R1, #0C]

; priprema varijabli za glavnu petlju
MOV R0, #10<4 ; brojač očitavanja
MOV R4, #10<8 ; adresa za spremanje temperature

```

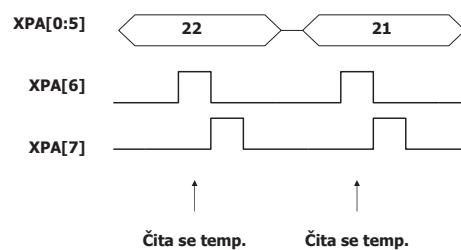
LCD

- LCD-prikaznik je izlazna vanjska jedinica koja omogućuje ispisivanje do osam ASCII-znakova na zaslonu LCD-a. LCD je jednostavna jedinica preko koje se rezultati operacija izvedenih u procesoru mogu prikazati korisniku (ispis znakova tijekom simulacije bit će prikazan u prozoru simulatora ATLAS).
- Komunikacija s LCD-om objašnjena je na primjeru kad je on spojen na vrata B sklopa GPIO.
- Sabirnica LCD-a je 8-bitna i organizirana tako da se nižih sedam bitova (bitovi 0 do 6) koriste za podatke
- Najviši bit (bit 7) je upravljački bit (WR) kojim se podatci upisuju u interni registar LCD-a.
- Registar LCD-a pamti do osam ASCII-zнакова.

Temperaturni sklop

- Temperaturni sklop je jednostavna vanjska jedinica koja omogućuje mjerjenje i očitavanje vanjske temperature. Komunikacija s temperaturnim sklopm objašnjena je na primjeru kad je on spojen na vrata A sklopa GPIO.
- Temperaturni sklop periodički mjeri temperaturu i izmjerenu vrijednost u formatu 6-bitnog cijelog broja postavlja na ulaz GPIO i to na priključke XPA[0] do XPA[5].
- Čitanjem vrata A može se pročitati trenutačna vrijednost temperature (u nižih 6 bitova).

Vremenski dijagram očitanja temperature



Primjer: Čitanje temperature

```

PETLJA
LDR R2, [R1, #0] ; glavna petija
ANDS R2, R2, #00000001 ; čitaj stanje tipke (najniži bit na portu A)
BEQ PETLJA ; ako nije pritisnuta, čekaj

CEKAJ_TEMP
LDR R2, [R1, #4] ; čekaj spremnost temperaturnog sklopa
ANDS R2, R2, #01000000
BEQ CEKAJ_TEMP

CITAJ_TEMP
LDR R2, [R1, #4] ; čitaj temperaturu (nižih 6 bitova)
AND R2, R2, #00111111
STRB R2, [R4], #1 ; spremi temperaturu u memoriju

ORR R2, R2, #00000000 ; postavi bit 7 u jedan
STR R2, [R1, #4]
AND R2, R2, #01111111 ; postavi bit 7 u nulu
STR R2, [R1, #4]

SUBS R0, R0, #1 ; smanji brojač za 100 očitanja
BHI PETLJA
HALT

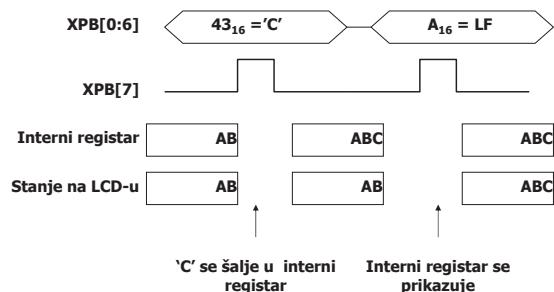
GPIO DW FFFF1000 ; GPIO bazna adresa

```

Način ispisu znakova na LCD

- Dok je bit WR neaktivovan (XPB[7] u logičkoj nuli), na bitove XPA[0:6] treba postaviti ASCII-kód znaka koji se želi ispisati.
- Kad se postavi vrijednost znaka, na bit WR treba poslati pozitivan impuls (stanje mu se iz nule postavi na jedinicu i ponovo vrati u nulu). Impuls uzrokuje upisivanje znaka (postavljenog na XPA[0:6]) na krajnje desno mjesto u interni registar LCD-a.
- Svi ostali do tada ispisani znakovi pomiču se za jedno mjesto ulijevo, a prethodni krajnje lijevi znak se gubi.
- Da bi se znakovi pohranjeni u internom registru prikazali na zaslonu LCD-a, programer treba poslati specijalan ASCII-znak LF (line feed) s kódom 0A(16).
- Kada LCD primi znak LF, on ne biva spremljen u interni registar, već uzrokuje ispis svih znakova iz internog registra na zaslon LCD-a.
- Postoji još jedan specijalan znak, ASCII-znak CR (carriage return) s kódom 0D(16), koji briše sadržaj internog registra (na svih osam mesta upisuje se ASCII kód 20(16) odnosno znak praznine ili razmaka).

Vremenski dijagram ispisa na LCD



© Kovač, Basch, FER, Zagreb

48

Primjer: Ispis teksta 'FER' na LCD

; potprogram za ispis jednog znaka na LCD spojen na port B sklopa GPIO
; R0 = ASCII kód znaka koji treba ispisati
; R2 = bazna adresa sklopa GPIO

```
LCDWR STMFD R13!, {R0}

    AND R0, R0, #7F          ; postavi bit 7 u nulu (za svaki slučaj) i pošalji znak
    STRB R0, [R2, #4]

    ORR R0, R0, #80          ; postavi bit 7 u jedan (podigni impuls)
    STRB R0, [R2, #4]

    AND R0, R0, #7F          ; postavi bit 7 u nulu (spusti impuls)
    STRB R0, [R2, #4]

LDMFD R13!, {R0}
MOV PC, LR                 ; povratak
```

© Kovač, Basch, FER, Zagreb

50

Dijelovi arhitekture

- AMBA APB sučelje
 - 32-bitno brojilo
 - 32-bitni register usporedbe (match register)
 - 32-bitni sklop za usporedbu.
- 32-bitno brojilo je glavni dio sklopa RTC. Ovo brojilo uvećava svoj sadržaj za jedan na svaki rastući brid signala CLK1HZ. Ako brojilo dođe do vrijednosti 0xFFFFFFFF, tada se na sljedeći brid vrijednost brojila postavlja na 0x00000000 i nastavlja dalje normalno brojiti.

© Kovač, Basch, FER, Zagreb

52

RTC: Registri

- RTCDR (Real Time Clock Data Register)
 - RTCDR je 32-bitni register podataka. Čitanjem ovog registra dobiva se trenutačna vrijednost brojila. Pisanje u ovaj register nije dozvoljeno.
- RTCMR (Real Time Clock Match Register)
 - RTCMR je 32-bitni register usporedbe. Upisivanjem podatka u ovaj register postavlja se nova vrijednost koja služi za usporedbu s brojilom. Čitanjem ovog registra dobiva se zadnja vrijednost upisana u register usporedbe.

© Kovač, Basch, FER, Zagreb

Primjer: Ispis teksta 'FER' na LCD

LCD je spojen na vrata B sklopa GPIO. GPIO je na adresi FFFFFF00(16). Treba ispisati tekst "FER". Slanje pojedinog znaka na vrata B treba riješiti potprogramom. Potprogram preko R0 prima ASCII znak, a preko R2 baznu adresu sklopa GPIO.

```
'ORG 0
MOV R13,#1<16
MOV R1, #1<8           ; stog
LDR R2, [R1]            ; R1=100(16), jer se tu nalazi adresa GPIO
                        ; u R2 se učitava bazna adresa GPIO-a

START MOV R0, #0D        ; znak 0xD, briše se interni registar
BL LCDWR                ; piše se na LCD
MOV R0, #46               ; ASCII kod znaka 'F'
BL LCDWR                ; piše se na LCD
MOV R0, #45               ; ASCII kod znaka 'E'
BL LCDWR                ; piše se na LCD
MOV R0, #52               ; ASCII kod znaka 'R'
BL LCDWR                ; piše se na LCD
MOV R0, #0A               ; znak 0xA, ispis znakova na zaslon
BL LCDWR                ; piše se na LCD

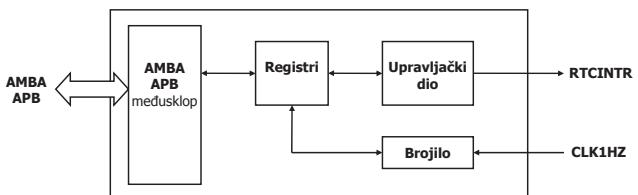
SWI 123456

'ORG 100
DW FFFFFF00              ; adresa sklopa GPIO
```

© Kovač, Basch, FER, Zagreb

49

Sklop RTC (Real Time Clock)



© Kovač, Basch, FER, Zagreb

51

Programski model

Adresa	Naziv registra	Opis
RTC_bazna_adr	RTCDR	32-bitni register podataka (može se samo čitati)
RTC_bazna_adr + 0x4	RTCMR	32-bitni register usporedbe
RTC_bazna_adr + 0x8	RTCSTAT/RTCEOI	1-bitni register stanja prekida (ako se čita) 0-bitni register za brisanje prekida (ako se piše)
RTC_bazna_adr + 0xC	RTCLR	32-bitni register za punjenje brojila
RTC_bazna_adr + 0x10	RTCCR	1-bitni upravljački register

© Kovač, Basch, FER, Zagreb

53

RTC: Registri

- RTCSTAT/RTCEOI (Real Time Clock Interrupt STATus Register/Real Time Clock Interrupt Clear Register)
 - RTCSTAT/RTCEOI je virtualni register bez fizičkog sklopovlja za pohranjivanje podataka. Pisanjem bilo kojeg podatka na ovu adresu čisti se prekidni signal RTCINTR i pripadni registar*. Čitanjem s ove adrese dobiva se podatak koji na bitu 0 (najniži bit) ima trenutačnu vrijednost RTCINTR. Ako je bit 0 postavljen na jedinicu, to znači da je prekidni signal aktiviran.
- RTCLR (Real Time Clock Load Register)
 - RTCLR je 32-bitni register koji služi za upis vrijednosti u brojilo ili čitanje zadnje vrijednosti koja je upisana. Upisana vrijednost se upisuje u brojilo (koje se kasnije mijenja prilikom odbrojavanja), a vrijednost u ovom registru ostaje nepromijenjena.

* interni status-bistabil

54

© Kovač, Basch, FER, Zagreb

55

RTC: Registri

• RTCCR (Real Time Clock Control Register)

• RTCCR je 1-bitni upravljački registar kojim programer može omogućiti ili onemogućiti generiranje prekida. Ako se na bit 0 (najniži bit) ovog registra upiše logička nula, tada se RTC-u onemogućuje generiranje prekida. Ako se upiše jedinica, tada se omogućuje generiranje prekida. Čitanjem ovog registra na bitu 0 dobiva se zadnja upisana vrijednost prekidnog bita. Ostali bitovi u ovom registru ne postoje.

Primjer

Treba napisati program za izmjenično ispisivanje znaka 0 i znaka 1 na zaslonu LCD-a i to tako da se znakovi mijenjaju svakih 5 sekundi. Mjerenje perioda valja riješiti korištenjem RTC-a. Program napisati za ATLAS.

LCD je spojen na vrata B sklopa GPIO (GPIO je na adresi FFFFFF00).

Na ulaz CLK1HZ od RTC-a je spojen signal frekvencije 1 Hz. RTC je na adresi FFFFE00 i spojen je na IRQ

Primjer

; potprogram koji dohvaca znak, mijenja ga iz '0' u '1' (ili obratno) te ga ispisuje na LCD
PISI_ZNAK
STMFD R13!, {R0, R2, R14}

```
LDR R2, GPIO          ; u R2 dohvati adresu sklopa GPIO
MOV R0, #0D           ; znak 0xD, briše se interni registar
BL LCDWR             ; šalje se na LCD

LDR R0, ZNAK          ; dohvati znak iz memorije
EOR R0, R0, #1         ; mijenja znak '0' <--> '1'
STR R0, ZNAK          ; spremi znak natrag u memoriju

BL LCDWR             ; šalje se znak '0' ili '1' na LCD

MOV R0, #0A           ; znak 0xA, ispis znaka na zaslonu
BL LCDWR             ; šalje se na LCD

LDMFD R13!, {R0, R2, R14}
MOV PC, LR
```

; potprogram za ispis znaka iz R0 na LCD (spojen na vrata B sklopa GPIO na baznoj adresi R2)
LCDWR ... (potprogram kao u primjeru za LCD na slajdu 51)

Razni primjeri programa za GPIO+RTC

RTC: način rada

- Brojilo se povećava na svaki rastući brid na signalu CLK1HZ
- Kad brojilo dosegne vrijednost upisanu u registru usporedbe (match register), onda RTC postaje spreman i postavlja prekid (ako je omogućen u registru RTCCR)
- Brojilo nastavlja s brojenjem i ne vraća se automatski na ništicu*
 - Ako se želi ponoviti ciklus brojenja, onda programski treba u brojilo upisati ništicu
- Nakon što postane spreman (ili nakon što postavi prekid) RTC-u treba obrisati stanje (tj. prekid ako ga je postavio)

* Za razliku od FRISC-CT-a u kojem ciklus brojenja automatski započinje iznova

Primjer

```
'ORG 0
B GLAVNI

'ORG 18          ; adresa za obradu iznimke IRQ

PREKDINI
; druge iznimke se ne koriste pa možemo odmah ovdje napisati cijeli prekidni potprogram
STMFD R13!, {R0, R3, R14}

LDR R3, RTC      ; dohvati adresu sklopa RTC
; reinicijaliziraj RTC
STR R0, [R3, #8]
MOV R0, #0
STR R0, [R3, #0C]
MOV R0, #5
STR R0, [R3, #4] ; napuni ponovo brojač s ništicom
; napuni ponovo RTCMR - nije nužno (već u njemu piše 5)

BL PISI_ZNAK    ; ispiši sljedeći znak

LDMFD R13!, {R0, R3, R14}
SUBS PC, R14, #4 ; povratak iz obrade iznimke
```

Primjer

```
GLAVNI
MOV R13, #1<16      ; stog
; inicijalizacija RTC-a
LDR R3, RTC          ; u R3 učitaj adresu RTC-a
MOV R4, #5
STR R4, [R3, #4]
MOV R4, #1
STR R4, [R3, #10]    ; omogući prekid (RTCCR)

BL PISI_ZNAK          ; inicijalno pisanje znaka (GPIO ne treba inicijalizirati)

MRS R0, CPSR          ; pročitaj CPSR u R0
BIC R0, R0, #80        ; na mjestu bita I se stavljaju nula
MSR CPSR_c, R0         ; i to se ponovo zapisuje u CPSR

PETLJA
B PETLJA             ; beskonačna petlja

GPIO    DW 0FFFFF000   ; adresa sklopa GPIO
RTC     DW 0FFFFE00    ; adresa sklopa RTC
ZNAK    DW 030          ; trenutačni znak za ispis (inicijalno ASCII kod znaka '0')
```

Primjer 1

Računalni sustav sastoji se od procesora ARM, jedinice GPIO (0xFFFF0100) i uređaja za ispis računa koji je spojen na port A (8 podatkovnih priključaka) i na port B (izlazni signal SEND (bit 0) i ulazni signal ACK (bit 1)). Uredaj radi tako da u trenutku aktiviranja impulsa na signalu SEND počne ispis 8-bitnog podatka koji je prisutan na podatkovnim priključcima (port A). Dovršetak ispisa svakog 8-bitnog podatka uređaj za ispis dojavljuje procesoru generiranjem impulsa na signalu ACK.

Potrebno je napisati potprogram SEND koji inicijalizira sklop GPIO, te dani niz podataka ispisuje na papirnu traku. Adresa niza prenosi se preko registra R0. Duljina niza nije unaprijed poznata, ali se zna da je niz završen podatkom 0xAA. Adresa GPIO-a zapisana je na fiksnoj memorijskoj lokaciji (na adresi 0x100).

Glavni program treba korištenjem potprograma SEND poslati na ispis podatke na adresi 0x200.

Rješenje primjera 1

```
'ORG 0
; glavni program
MOV R13,#1<16          ; stog
MOV R0, #2<8             ; parametar za SEND (adresa niza)
BL SEND
HALT

; bazna adresa GPIO-a
`ORG 100
DW 0FFFF0100

; podaci za ispis
`ORG 200
DB 0F0, 14, 12, 05, 0C4, 0AA
```

© Kovač, Basch, FER, Zagreb

64

Rješenje primjera 1

```
; petlja za ispis podatka na traku
PISI LDRB R1, [R0], #1    ; dohvati podatak iz niza
CMP R1, #0AA              ; provjeri kraj
BEQ KRAJ

STRB R1, [R2]; šalji podatak na port A

; slanje impulsa SEND (port B, bit 0)
MOV R1, #1
STRB R1, [R2, #4]
MOV R1, #0
STRB R1, [R2, #4]

; čekanje impulsa ACK (port B, bit 1)
ACK LDRB R1, [R2, #4]
CMP R1, #0
BEQ ACK
B PISI                   ; ponovi za sljedeći znak

KRAJ LDMFD R13!, {R0, R1, R2}
MOV PC, LR
```

© Kovač, Basch, FER, Zagreb

66

Rješenje primjera 2

```
'ORG 0
LDR R1, GPIO
; GPIO Port A inicijalizacija
MOV R0, #%B 1000
STRB R0, [R1, #8]

PETLJA
; dohvati niža 3 bita podatka s porta A
LDR R0, [R1]
AND R3, R0, #%B 0111

; ako je udio svih sirovina OK, onda će
; u R2 biti broj %b111
CMP R3, #%B 111

; bazna adresa GPIO-a
GPIO DW 0FFFF0000
```

Komentar: uočite da za komunikaciju sa GPIO-om možemo koristiti naredbe load/store i za riječi i za bajtove

© Kovač, Basch, FER, Zagreb

68

Rješenje primjera 3

```
'ORG 0
B GLAVNI

'ORG 1C          ; Prekidni potprogram za FIQ
; ne spremamo kontekst jer koristimo registre od načina rada FIQ

MOV R8, #1<8
LDR R9, [R8], #4      ; R9 = RTC bazna adresa
LDR R10, [R8]         ; R10 = GPIO bazna adresa

LDRB R8, [R10]        ; promijeni stanje
EOR R8, R8, #1        ; pravokutnog signala na
STRB R8, [R10]         ; GPIO-u

STR R8, [R9, #8]       ; prihvati prekida RTC-a

MOV R8, #0             ; reinicijalizacija RTC-a

SUBS PC, LR, #4
```

© Kovač, Basch, FER, Zagreb

Rješenje primjera 1

```
; potprogram SEND
SEND STMFD R13!, {R0, R1, R2}

MOV R1, #1<8
LDR R2, [R1]           ; stavi adresu GPIO-a u R2
; inicijalizacija GPIO-a

; port A: izlazni - 8-bitni podatak
MOV R1, #%B 11111111
STRB R1, [R2, #8]

; port B: bit 0 izlazni, bit 1 ulazni
MOV R1, #%B 00000010
STRB R1, [R2, #OC]
```

© Kovač, Basch, FER, Zagreb

65

Primjer 2

Računalni sustav sastoji se od procesora ARM, sklopa GPIO (0xFFFF0000) i kontrolne jedinice CTRL koja služi za mjerjenje udjela triju sirovina u proizvodnom procesu.

Jedinica CTRL spojena je na GPIO na port A i to sa 4 priključka. Prva tri priključka su ulazni i spojeni su na bitove 0, 1 i 2 (XPA[0]-XPA[2]). CTRL na ovim priključcima daje logičku 1 za svaku od 3 sirovine kada je njezin udio u proizvodnom procesu zadovoljavajući, a inače daje logičku 0. Četvrti priključak je izlazni i spojen je na bit 3 (XPA[3]). Pomoću ovog priključka procesor ARM zaustavlja i pokreće proizvodni proces.

Ako udio bilo koje od sirovina nije zadovoljavajući, potrebno je ugasiti proces slanjem logičke 0 na XPA[3]. Kada udio svih sirovina postane zadovoljavajući, treba uključiti proces slanjem logičke 1 na XPA[3]. Upravljanje procesom ponavljati beskonačno.

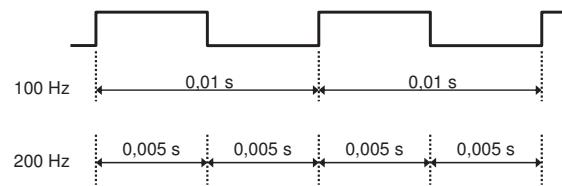
© Kovač, Basch, FER, Zagreb

67

Primjer 3

U računalnom sustavu nalaze se ARM, sklop RTC (na adresi 0xFFFF0000) i sklop GPIO (na adresi 0xFFFF1000). RTC je spojen na FIQ.

Potrebno je na sklopu GPIO, na priključku 0 porta A (XPA[0]) generirati pravokutni signal frekvencije 100 Hz. Na ulaz sklopa RTC doveden je signal frekvencije 10 kHz.



© Kovač, Basch, FER, Zagreb

69

Rješenje primjera 3

```
GLAVNI

MOV R13, #81<8      ; inicijalizacija SP-a
MOV R0, #1<8
LDR R1, [R0], #4      ; R1 = RTC bazna adresa
LDR R2, [R0]           ; R2 = GPIO bazna adresa

; inicijalizacija GPIO port A - XPA[0] je izlazni
MOV R0, #%B 00000001
STRB R0, [R2, #8]

; inicijalizacija RTC-a
MOV R0, #%D 50         ; konstanta brojenja
STR R0, [R1, #4]        ; RTCMR = 50 = 10 kHz / 200Hz = 10000 / 200
; enable interrupt u RTC-u
MOV R0, #1
STR R0, [R1, #10]

MOV R0, #0              ; briši brojilo u RTC-u
STR R0, [R1, #0C]        ; RTCLR = 0
```

70

© Kovač, Basch, FER, Zagreb

71

Rješenje primjera 3

; omogućavanje prekida FIQ

MRS R0, CPSR

BIC R0, R0, #40

MSR CPSR_c, R0

PETLJA B PETLJA

; bazne adrese RTC-a i GPIO-a

```
`ORG 100
DW 0FFFF0000 ; RTC
DW 0FFFF1000 ; GPIO
```

© Kovač, Basch, FER, Zagreb

72

Rješenje primjera 4

`ORG 0
B GLAVNI

`ORG 18 ; adresa prekidnog potprograma
B PREKIDNI

GLAVNI

MOV R13, #1<16 ; inicijalizacija SP-a

; GPIO je inicijalno dobro postavljen

; inicijalizacija RTC-a

LDR R2, RTC ; R2 = bazna adresa RTC-a

MOV R3, #0

STR R3, [R2, #0C] ; RTCLR - brojilo

MOV R3, #%d100

STR R3, [R2, #4] ; RTCMR - konstanta

MOV R3, #1

STR R3, [R2, #10] ; RTCCR - prekidanje

; omoguciti prekid IRQ

MRS R0, CPSR

BIC R0, R0, #80

MSR CPSR_c, R0

PETLJA B PETLJA

GPIO DW FFFF1000 ; bazna adresa GPIO-a
RTC DW FFFF0000 ; bazna adresa RTC-a

© Kovač, Basch, FER, Zagreb

74

Primjer 4

U računalnom sustavu nalaze se ARM, GPIO (0xFFFF1000) i RTC (0xFFFF0000).

GPIO preko vrata A prima 8-bitni NBC podatak koji predstavlja temperaturu nekog procesa (u Celsijevim stupnjevima). Pinovi 0, 1 i 2 od vrata B služe za signalizaciju - pomoću njih se pale i gase lampice: crvena (XPB[0]), žuta (XPB[1]) i zelena (XPB[2]). Lampice se pale slanjem logičke 1, a gase slanjem logičke 0 na odgovarajući priključak.

Ako je temperatura veća od 128°C, treba upaliti crvenu lampicu, a ugasiti žutu i zelenu. Isto treba učiniti sa žutom lampicom, ako je temperatura u rasponu od 55°C do 128°C, odnosno sa zelenom ako je temperatura manja od 55°C.

Svake sekunde potrebno je mjeriti temperaturu i obaviti navedenu signalizaciju. Na RTC je spojen signal frekvencije 100 Hz.

© Kovač, Basch, FER, Zagreb

73

Rješenje primjera 4

PREKIDNI

STMFD R13!, {R2, R4, R7}

LDR R2, RTC ; R2 = RTC bazna adresa

STR R7, [R2, #8] ; brisanje int-zastavice u RTC-u

MOV R7, #0

STR R7, [R2, #0C] ; RTCLR - brisanje brojila

; dohvati temperaturu sa porta A

LDR R2, GPIO ; R2 = bazna adresa GPIO-a

LDR R4, [R2]

; ispitaj u kojem je opsegu temperatura

CMP R4, #%D 128

MOVHI R7, #%B 001 ; CRVENA = bit 0

BHI SIG

CMP R4, #%D 55

MOVLO R7, #%B 100 ; ZELENA = bit 2

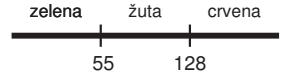
BLO SIG

MOV R7, #%B 010 ; ŽUTA = bit 1

SIG STR R7, [R2, #4] ; signalizacija lampicama

LDMFD R13!, {R2, R4, R7}

SUBS PC, LR, #4



© Kovač, Basch, FER, Zagreb

75

Primjer signala na običnoj statickoj memoriji (SRAM) od 64Mb (2Mx32b) tj. ($2^{21} \times 32b$)



Dio ARM8: Memorijski sustav

Ovaj dokument namijenjen je isključivo za osobnu upotrebu studentima Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

Svaka druga upotreba ili umnožavanje bilo kojeg dijela ovog dokumenta nije dozvoljena bez pismene dozvole autora.

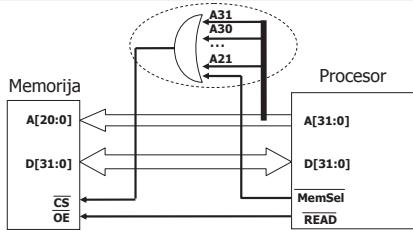
© Mario Kovač, Danko Basch



© Kovač, Basch, FER, Zagreb

1

Povezivanje memorije i procesora



- /CS** (Chip Select) ili **/CE** (Chip Enable) je signal koji omogućuje dekodiranje adresne sabirnice i ostalih upravljačkih signala unutar memorije
- /OE** (Output Enable) je signal koji podatke iz internog spremnika podataka u memoriji proslijeđuje na vanjsku sabirnicu podataka (obično se spaja na procesorov priključak /RD)
- /WE** (Write Enable) je signal koji inicira pisanje podataka sa sabirnice podataka na izabranu lokaciju u memoriji (obično se spaja na procesorov priključak /WR)
- /MemSel** (Memory Select) je signal kojim procesor definira da je na adresnoj sabirnici stabilna memoriska adresa (postoje različite izvedbe ovog signala)

© Kovač, Basch, FER, Zagreb

3

Načini povezivanja u stvarnosti

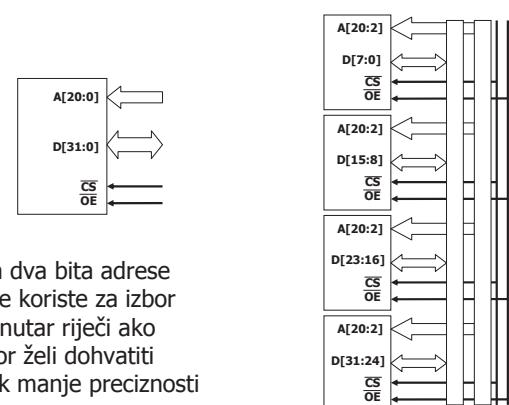
- Na dosadašnjim slikama memorija je prikazivana kao jedan čip s potrebnim (tj. velikim) brojem adresnih i podatkovnih priključaka
- U stvarnosti se ne proizvode memorijski čipovi s velikim brojem adresnih priključaka i podatkovnih priključaka
- Cijena memorije s manje priključaka je povoljnija
- Memorija se formira kao blok memorijskih čipova manjeg kapaciteta i manje širine riječi*

* manji kapacitet znači manju širinu adresnih priključaka, a manja širina riječi znači manju širinu podatkovnih priključaka

© Kovač, Basch, FER, Zagreb

5

Memorijski blok s čipovima manje širine riječi



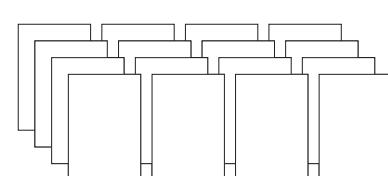
- Najniža dva bita adrese često se koriste za izbor bajta unutar riječi ako procesor želi dohvatiti podatak manje preciznosti

© Kovač, Basch, FER, Zagreb

6

Memorijski blok većeg kapaciteta

- Kombinacijom prethodna dva načina povezivanja ostvaruju se memorijski blokovi (memorijski moduli) većeg kapaciteta



- Veći memorijski blok dobiva se povezivanjem nekoliko mem. čipova
- adresni dekoder obavlja odabir pojedine čipa unutar grupe

© Kovač, Basch, FER, Zagreb

7

© Kovač, Basch, FER, Zagreb

8

Primjeri stvarnih memorija i njihovih cijena

- U stvarnosti se jako rijetko koriste memorijski čipovi sa širinom riječi od npr. 64 bita, već se uglavnom koriste širine od 8 ili 16 bita koje se povezuju u veći memorijski modul kao što je upravo pokazano
- Ukupna cijena memorijskog modula (načelno):
 - Što je kapacitet memorijskog čipa veći, to mu je cijena po Mb veća (pinovi, kućište, površina silicija, itd.)
 - Cijena memorijskog modula (pločica, montiranje, prostor, itd.) raste s porastom broja memorijskih čipova
- UKUPNA CIJENA: **mora se tražiti optimum zbroja gornje dvije funkcije**

© Kovač, Basch, FER, Zagreb

9

Ubrzanje memorijskog sustava

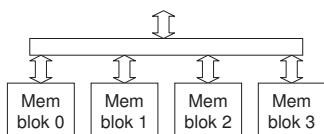
- Zbog cijene se u stvarnim računalnim sustavima primjenjuju određene metode organizacije memorijskog sustava kojima se poboljšavaju performanse memorijskog sustava (postiže se ubrzanje)
- Rezultat tih metoda je da se korištenjem jeftinije tehnologije postižu performanse memorijskog sustava koje su bliske izvedbi s brzim, ali i znatno skupljim memorijama
- Ovakve metode organizacije memorije **stvaraju dojam** velike količine brze memorije
- U nastavku ćemo objasniti neke osnovne organizacije memorijskog sustava kojima se poboljšavaju performanse

© Kovač, Basch, FER, Zagreb

11

Preplitanje - načelo rada

- Povećanje propusnosti mem. sustava može se postići **preplitanjem** (engl. interleaving) memorija
- Adresni prostor dijeli se u **n** memorijskih blokova (engl. memory bank) tako da memorijska lokacija s adresom **a** bude smještena u memorijski blok (**a mod n**). Operacija **mod** jednostavno se izvodi dekodiranjem zadnjih bitova adrese

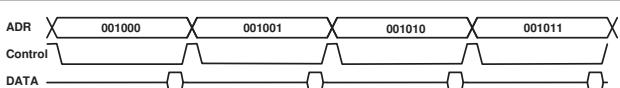


- Takva memorija sa **n** blokova se naziva **memorija s n-terostrukim preplitanjem** (engl. n-way interleaved)

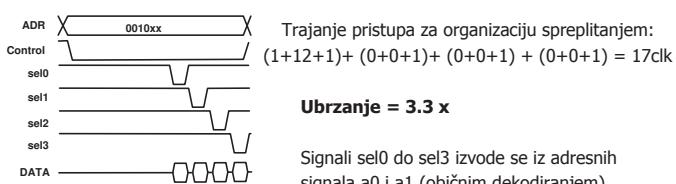
© Kovač, Basch, FER, Zagreb

13

Primjer uštede za čitanje 4 slijedna podatka



Trajanje običnog pristupa za 4 podatka (uz vrijeme pristupa = 12 clk):
4*(1clk (slanje adrese) + 12clk (pristup) + 1clk (učitavanje podatka))=56clk



Zahtjevi na memoriju

- Pored zahtjeva za što većim kapacitetom, u primjeni je isto tako važan i zahtjev za željenom brzinom memorije
- Brzina memorije značajno utječe na performanse sustava
- Pri razvoju programa uvijek postoji želja ili potreba za velikom količinom **brze** memorije
- Brza memorija je **skupa**, a u nekom trenutku koristi se samo mali dio ukupne memorije u sustavu te je teško opravdati stavljanje velike količine brze memorije u računalni sustav

© Kovač, Basch, FER, Zagreb

10

Ubrzanje memorijskog sustava

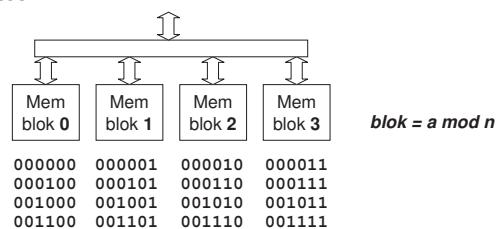
Preplitanje

© Kovač, Basch, FER, Zagreb

12

Preplitanje - načelo rada

- Na slici su prikazane adrese u pojedinim blokovima (binarno) za **n=4**
- Na taj način je osigurano da se susjedne adrese ne nalaze u istom bloku te se mogu paralelno dohvataći
- Sustav radi na načelu da se uklanja čekanje na dekodiranje adrese (engl. latency) tako da se dekodiranje obavlja jednom za četiri uzastopne adrese



© Kovač, Basch, FER, Zagreb

14

Preplitanje

- Vrlo dobro rješenje za sustave s čestim pristupom slijednim lokacijama (npr. programski kod, pristup poljima podataka, ...)
- Efektivno vrijeme pristupa, a time i propusnost, povećava se približno za **n**, gdje je **n** broj memorijskih **blokova**
- Sustav izvrsno radi i za čitanje i pisanje !!!

Hijerarhijska organizacija memorije

- Primjer problema: učenje za ispit iz ARH1 1 u knjižnici

- **Način 1:**

- Sve knjige su na policama knjižnice
- Svaki put kad trebamo neki podatak iz knjige, uzmememo knjigu, pročitamo podatak i knjigu vratimo na policu
- **SPORO !!!**

- **Način 2:**

- Kad zatrebamo neku knjigu, uzmememo je s police i stavimo na stol
- Ako na stolu više nema mjesta, onda za svaku novu knjigu prvo moramo jednu knjigu sa stola spremiti na policu, a tek onda možemo uzeti novu knjigu za čitanje
- S obzirom da za proučavanje neke teme ne trebamo sve knjige, već da većinu vremena provedemo čitajući neki manji skup knjiga, nećemo trebati ustajati od stola tako često

- **BRŽE !!!**

Ubrzanje memoriskog sustava

Hijerarhijska organizacija memorije

Hijerarhijska organizacija memorije

- Prethodni primjer opisuje metodu poboljšanja performansi korištenjem hijerarhijske organizacije
- Često korištena metoda za poboljšanje performansi memoriskog sustava upravo je hijerarhijska organizacija
- Načelo hijerarhijske organizacije memorije određuje da se podaci koji se češće koriste smještaju u manju, ali brzu memoriju, dok se ostali, rjeđe korišteni podaci čuvaju u većoj i sporijoj memoriji
- Kako se može znati koji će se podaci češće koristiti? Podaci u stvarnim primjenama imaju karakteristiku nazvanu **lokalnost podataka**:
 - **Prostorna lokalnost**: ako prepostavimo da je neki podatak bio potreban, onda je vjerojatno da će i njemu **susjedni podaci uskoro** biti potrebni (npr. naredbe u nekom programu, elementi polja, ...)
 - **Vremenska lokalnost**: ako prepostavimo da je neki podatak bio potreban, onda je vjerojatno da će **isti podatak uskoro** biti opet potreban (npr. varijable u nekom programu, ...)

Hijerarhijska organizacija memorije

- U današnje vrijeme koriste se **tri osnovne tehnologije** za izvedbu memoriske hijerarhije:
 - SRAM (statička memorija – static random access memory)
 - DRAM (dinamička memorija – dynamic random access memory)
 - HD (tvrdi diskovi – hard disk)

Tehnologija	Karakteristike (red veličine)		
	Kapacitet	Vrijeme pristupa	Cijena po GB
SRAM	10 KB do 1 MB	<1 ns	1000 EUR
DRAM	100 MB do 1 GB	<10 ns	100 EUR
HD	više od 100 GB	<10 ms	<10 EUR

Hijerarhijska organizacija memorije

- S obzirom da hijerarhijski memoriski sustavi predstavljaju veoma široko područje i da se nove efikasnije metode korištenja pojavljuju gotovo svakodnevno u stručnoj literaturi i industriji, mi ćemo u okviru Arhitekture računala 1 objasniti samo neka osnovna načela
 - stvarne izvedbe mogu biti složenije, koristiti brojne podvarijante i poboljšanja itd.
- Slijedi objašnjenje osnovnih načela rada priručne memorije...

Hijerarhijska organizacija memorije

- Hijerarhijska organizacija memoriskog sustava može se simbolički prikazati piramidom:



- Svaka razina izvedena je u drugoj tehnologiji >>>

Hijerarhijska organizacija memorije

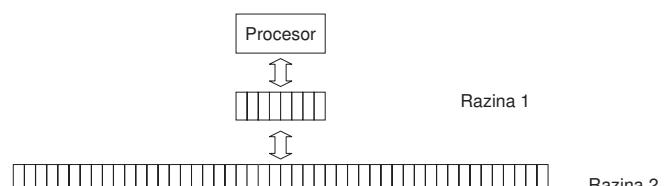
- **Glavna ili radna memorija** (druga razina) ostvarena je korištenjem DRAM-a, koji predstavlja kompromis između kapaciteta, brzine i cijene
- Između glavne memorije i procesora*, na prvu razinu, može se staviti manja količina brže statičke memorije SRAM. Takva memorija naziva se **priručna memorija** (engl. cache)
- Na kraju, u sustav se na treću razinu obično stavlja magnetska memorija u obliku diskova. Ona ima malu cijenu i velik kapacitet, ali je spora



* možemo promatrati kao da su registri procesora još viša razina od prve razine

Kako radi sustav s priručnom memorijom

- Memorija se dijeli u **blokove**, minimalne količine podataka koji se mogu naći u susjednim razinama



Kako radi sustav s priručnom memorijom

- Ako procesor želi pročitati neki podatak, tada se **pretražuje** memorija koja je najbliža procesoru, a to je **priručna memorija**
 - Ako se podatak **nalazi** u priručnoj memoriji, tada se to naziva **pogodak** (engl. hit)
 - Ako se podatak **ne nalazi** u priručnoj memoriji, tada se to naziva **promašaj** (engl. miss). Tada se pristupa glavnoj memoriji kako bi se dohvatio podatak
- **Omjer pogodaka** (engl. hit-rate): broj pristupa memoriji kada je podatak pronađen u višoj razini u odnosu na ukupni broj pristupa
- **Omjer promašaja** (engl. miss-rate): $miss_rate = 1 - hit_rate$, tj. broj pristupa memoriji kada podatak nije pronađen u višoj razini u odnosu na ukupni broj pristupa
- Budući da procesor stalno pristupa podacima, **pretraživanje mora biti iznimno brzo** (ostvaruje se sklopovski)
 - Pretraživanje čemo detaljnije objasniti malo kasnije

© Kovač, Basch, FER, Zagreb

25

Obrada pisanja

- Pisanje novog podataka, odnosno promjena postojećeg podatka u priručnoj memoriji predstavlja poseban problem, bez obzira što se on već u njoj nalazi, tj. bez obzira što je došlo do pogotka*
- Problem je osiguravanje istovjetnosti (ili koherentnosti) podataka u različitim razinama memorije:
 - Ako procesor mijenja podatak u priručnoj memoriji, onda je potrebno osvježiti i originalni podatak u glavnoj memoriji

* slučaj promašaja kod pisanja obraditi čemo malo kasnije

© Kovač, Basch, FER, Zagreb

27

Obrada pisanja

- Napredniji algoritam je **pisanje pri povratku** (engl. write-back)
 - Procesor mijenja podatak SAMO u priručnoj memoriji
 - Podatak se zapisuje u glavnu memoriju SAMO onda kada se blok izbacuje iz priručne memorije
 - Efikasniji algoritam od prethodnog, ali teži za izvedbu:
 - U glavnu memoriju se zapisuju samo oni podaci iz bloka priručne memorije koji su bili mijenjani
 - Za svaki podatak u priručnoj memoriji imamo dodatnu zastavicu u kojoj se pamti je li podatak mijenjan ili nije (engl. dirty bit)
 - Zastavica se postavlja kod svake operacije pisanja. Kod punjenja novog bloka u priručnu memoriju brišu se sve zastavice za taj blok

© Kovač, Basch, FER, Zagreb

29

Pretraživanje podataka i preslikavanje

- Problem: **kako efikasno obaviti pretraživanje**, tj. kako odrediti je li podatak u priručnoj memoriji i ako jeste, gdje se nalazi?
 - Prilikom pristupa podatuču zadaje se njegova normalna adresa u radnoj memoriji. Potrebno je odrediti nalazi li se taj podatak u priručnoj memoriji i u kojem bloku.
 - Zapravo treba normalnu adresu podatka **pretvoriti (preslikati)** u adresu podatka/bloka u priručnoj memoriji
 - Budući da se između priručne i glavne memorije ne premještaju pojedini podaci, nego blokovi, onda promatramo adrese blokova
- Postoje mnogi načini organizacije preslikavanja podataka iz niže razine u priručnu memoriju:
 - direktno preslikavanje
 - skupovna asocijativnost
 - puna asocijativnost

© Kovač, Basch, FER, Zagreb

31

Obrada promašaja kod čitanja

- Ako traženi podatak jeste u priručnoj memoriji, onda ga se jednostavno čita (pogodak), a operacija čitanja je vrlo brza
- Ako traženi podatak nije u priručnoj memoriji (promašaj), tada jedinica za upravljanje memorijom mora zahtijevati taj podatak iz glavne memorije (zapravo, cijeli blok)
- U tom trenutku rad procesora se mora zaustaviti dok se traženi podatak (tj. blok) ne dohvati i spremi u priručnu memoriju. To dovodi do kašnjenja zbog promašaja (engl. miss penalty)
- Postoje mnogi načini kako se pokušava izbjegći ovaj negativni utjecaj promašaja na efikasnost rada procesora, ali bez obzira na organizaciju, promašaj uvek degradira performanse

© Kovač, Basch, FER, Zagreb

26

Obrada pisanja

- Najjednostavniji algoritam je **pisanje s proslijedivanjem** (engl. write-through)
 - Svaki put kad procesor promijeni podatak u priručnoj memoriji, taj isti podatak se odmah proslijedi i zapisuje i u glavnu memoriju
 - Osnovni nedostatak: znatan gubitak vremena na pisanje u glavnu memoriju
 - Efikasnost se može povećati tako da procesor ne čeka dovršetak upisa u glavnu memoriju, već nastavlja raditi, a pisanje u glavnu memoriju se obavlja preko posebnih međuspremnika neovisno o radu procesora

© Kovač, Basch, FER, Zagreb

28

Obrada promašaja kod pisanja

- **Promašaj pri pisanju**
 - Ako se blok u kojeg se treba upisati vrijednost ne nalazi u priručnoj memoriji, tada postoje razna rješenja
 - Najjednostavnije rješenje je da se traženi blok prvo dohvati u priručnu memoriju, a zatim obrađuje nekim od prije navedenih algoritama

© Kovač, Basch, FER, Zagreb

30

Direktno preslikavanje

- Jeden od najjednostavnijih načina preslikavanja je **direktno preslikavanje**
 - Podatak s određene adrese iz glavnoj memoriji može se preslikati **samo na jednu točno definiranu adresu** u priručnoj memoriji.
 - Formula koja obavlja direktno preslikavanje je*:
$$ADRBLK_u_priručnoj_mem = ADRBLK_u_glavnoj_mem \bmod Broj_blokova_u_priručnoj_mem$$
 - To znači da se više blokova iz glavne memorije preslikavaju u jedan blok priručne memorije
 - Na primjer, ako priručna memorija ima 8 blokova, onda se u blok 3 u priručnoj memoriji preslikavaju blokovi 3, 11, 19, ..., glavne memorije (jer vrijedi: $3 = 3 \bmod 8 = 11 \bmod 8 = 19 \bmod 8 = \dots$)
 - Posljedica je da u priručnoj memoriji ne mogu istovremeno biti neki od blokova iz glavne memorije (npr. 3 i 11, 3 i 19, 11 i 19, itd.)

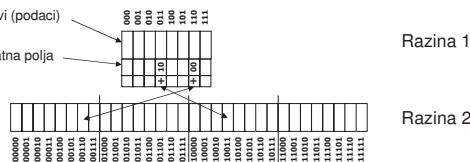
* ADRBLK je kratica za *adresa bloka*

© Kovač, Basch, FER, Zagreb

32

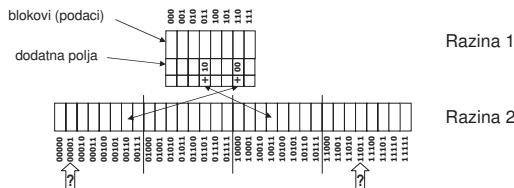
Direktno preslikavanje

Primjer priručne memorije kapaciteta 8 blokova i glavne memorije kapaciteta 32 bloka:



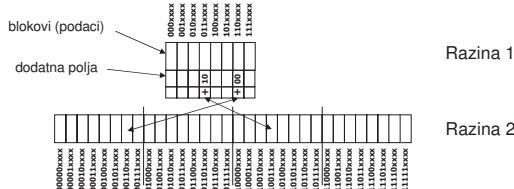
- Zeleni blokovi u glavnoj memoriji mogu se preslikati samo na položaj zelenog bloka u priručnoj memoriji (isto vrijedi i za crvene blokove)
- Na slici je prikazana situacija gdje se u priručnoj memoriji trenutno nalaze samo dva bloka iz glavne memorije: 00110 i 10011 koji su označeni strelicama (ostali su blokovi priručne memorije neiskorišteni)
- Na slici su prikazane adrese blokova, a ne pojedinih podataka, jer pri preslikavanju radimo s adresama blokova

Direktno preslikavanje



- Na primjer:
 - Ako se zatraži blok 00001, onda on (zbog direktnog preslikavanja) može biti u priručnoj memoriji na adresi 001. Za blok 001 se u dodatnom polju prvo provjerava oznaka valjanosti pomoću koje se ustanovljava da blok 00001 sigurno nije u priručnoj memoriji, jer na toj poziciji nema niti jednog bloka (promašaj)
 - Ako se zatraži blok 11011, onda on može biti u priručnoj memoriji na adresi 011 pa se prvo provjerava oznaka valjanosti pomoću koje se ustanovljava da u tom bloku priručne memorije postoji neki blok (ima oznaku +), ali se ne zna koji. Zato se dalje provjerava dodatno polje, ali u tom polju piše 10 što nije isto kao viši dio adrese traženog bloka 11 pa znači da se tu ne nalazi traženi blok (promašaj)

Direktno preslikavanje



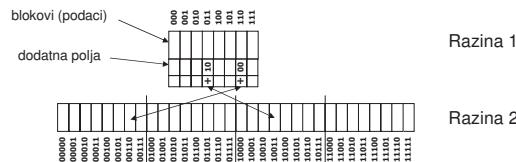
- Kad bi umjesto adresa blokova htjeli prikazati adrese pojedinih podataka, to bi izgledalo kao na ovoj slici, gdje je pretpostavljeno da svaki blok sadrži 16 podataka (za čije adresiranje trebaju 4 bita)
- Oznake xxxx predstavljaju moguće adrese (od 0 do 15) pojedinih podataka unutar bloka od 16 podataka: vidimo da se iz adrese podatka vrlo lako dobiva adresa bloka – potrebitno je samo odbaciti određen broj najnižih bitova (u ovom primjeru treba odbaciti 4 najniža bita)

Skupovna asocijativnost

- Radi smanjenja broja promašaja i poboljšanja performansi uvodi se **skupovna asocijativnost**

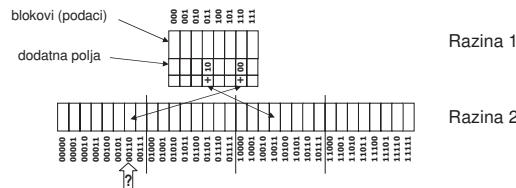
- To je mogućnost da priručna memorija može unutar jednog **skupa blokova** pohraniti **nekoliko blokova** iz glavne memorije (koji bi u direktnom preslikavanju spadali u isti skup)
- Formula koja obavlja skupovnu asocijativno preslikavanje je:
Adresa_skupa_priručna_memorija =
ADRBLK_glavna_memorija mod Broj_skupova_u_priručnoj_mem
- Blok glavne memorije uvijek se preslikava u isti skup, ali u skupu ima mesta za više blokova
- U isti skup preslikava se više blokova glavne memorije, ali u skupu ima mesta za više blokova
- Radi efikasnosti izvedbe skupovi obično sadrže 2, 4, 8, ... blokova
 - Na primer, ako skup ima dva bloka onda se za priručnu memoriju kaže da ima dvostruku asocijativnost (engl. two-way set-associative cache)

Direktno preslikavanje



- Na temelju nižih bita adrese bloka (tri smeđa bita na slici - to je zapravo operacija $mod\ 8$) zna se gdje bi podatak mogao biti u priručnoj memoriji
- Svakom bloku u priručnoj memoriji pridruženo je dodatno polje (engl. tag). U njima su informacije o višim bitovima adrese (dva plava bita na slici) te o prisutnosti/valjanosti bloka (znak + na slici). Ispitivanje je li traženi podatak prisutan ili nije obavljaju se provjerom dodatnih polja za blok odabran pomoću preslikavanja

Direktno preslikavanje



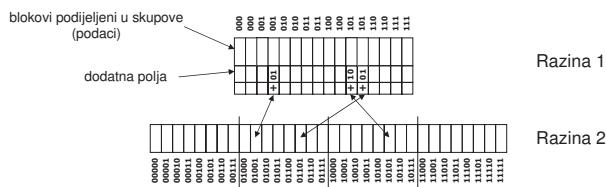
- Na primjer:
 - Ako se zatraži blok 00110, onda on može biti u priručnoj memoriji na adresi 110 pa se prvo provjerava oznaka valjanosti pomoću koje se ustanovljava da u tom bloku priručne memorije postoji neki blok, ali se ne zna koji. Zato se dalje provjerava dodatno polje, ali u tom polju piše 00 što je isto kao viši dio adrese traženog bloka 11 pa znači da se traženi blok nalazi u priručnoj memoriji (pogodak)

Direktno preslikavanje - svojstva

- Prednost: direktno preslikavanje je jednostavno
 - Pretraživanje podatka/bloka je brzo
 - Za svaki blok iz glavne memorije jednostavno se i jednoznačno određuje gdje se u priručnu memoriju treba smjestiti
 - Neki drugi blok, koji se tu već nalazi, treba izbaciti iz priručne memorije
 - Jednostavna sklopovska izvedba
- Nedostatak: nema izbora gdje će se u priručnu memoriju smjestiti blok iz glavne memorije
 - Postoji relativno velika vjerojatnost da je potrebno istodobno koristiti dva ili više blokova iz glavne koji se preslikavaju u isti blok priručne memorije
 - Postljedica su učestalo izbacivanje i učitavanje blokova uslijed čestih promašaja što može znatno narušiti performanse
 - Za poboljšanje performansi može se primijeniti drugačiji mehanizam preslikavanja >>>

Skupovna asocijativnost

Primjer priručne memorije s dvostrukom asocijativnošću (tj. sa po dva bloka u skupu) i sa 8 skupova



- Zeleni blokovi u glavnoj memoriji mogu se preslikati na jedan od dva položaja u zelenom skupu priručne memorije (isto vrijedi za crvene blokove)
- Na slici je prikazana situacija gdje se u priručnoj memoriji trenutno nalaze tri bloka iz glavne memorije: 01001 u crvenom skupu te 01101 i 10101 u zelenom skupu koji su označeni strelicama (ostali su blokovi priručne memorije neiskorišteni)
- Za glavnu memoriju su prikazane adrese blokova, a za priručnu memoriju adrese skupova

Skupovna asocijativnost

- Kad se traži neki blok, onda se prvo obavi preslikavanje njegove adrese u adresu skupa
- Za razliku od direktnog preslikavanja gdje je preslikavanje dalo jednoznačni položaj bloka, ovdje moramo ispitati sve blokove u skupu, jer traženi blok može biti u bilo kojem od njih
 - Ovo ispitivanje se odvija slično kao kod direktnog preslikavanja (valjanost i viši bitovi adrese), ali se to radi za svaki blok u skupu pomoću specijalnih sklopova (sklopovi za usporedbu i multipleksori)
- Što je veći broj blokova u skupu, to je manja vjerojatnost da će istodobno trebati upravo oni blokovi iz glavne memorije koji su svi u istom skupu i da će ih biti toliko puno da ne stanu u skup
 - Posljedica je smanjenje broja promašaja i znatno povećanje performansi memorijskog sustava

© Kovač, Basch, FER, Zagreb

41

Skupovna asocijativnost - svojstva

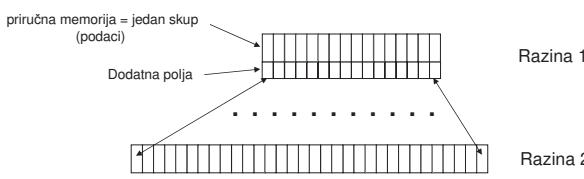
- Prednost: veća efikasnost zbog manjeg broja promašaja
 - Položaj blokova je fleksibilniji nego kod direktnog preslikavanja pa je manja vjerojatnost da će trebati u isti skup staviti više blokova koji su istovremeno potrebni
 - Ako je za neku primjeru veća vjerojatnost da istovremeno trebaju u priručnoj memoriji biti blokovi iz istog skupa, može se upotrijebiti organizacija s većim brojem blokova u skupu
- Nedostatak: složenija i skuplja izvedba
 - Potrebiti su dodatni algoritmi izbacivanja koji ne postoje u direktnom preslikavanju
 - Što ima više blokova u skupu, izvedba sklopova za traženje i zamjenu postaje sve komplikiranija, složenija i skuplja
 - Što ima više blokova u skupu, postupci traženja postaju sve sporiji. Time se povećava vrijeme pristupa priručnoj memoriji (vrijeme pogotka, engl. hit time) što smanjuje njene performanse

© Kovač, Basch, FER, Zagreb

43

Puna asocijativnost

- Svaki blok iz glavne memorije može doći na bilo koji položaj u priručnoj memoriji

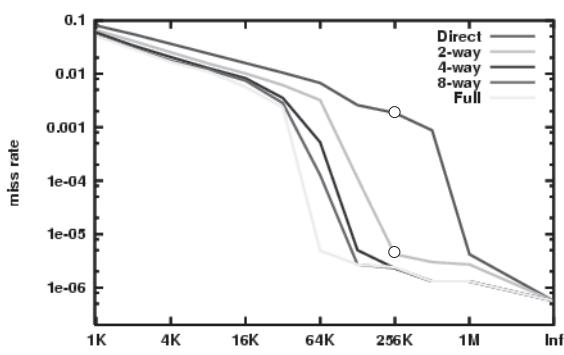


© Kovač, Basch, FER, Zagreb

45

Usporedba organizacija priručne memorije

- Izvor: <http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/>
- Npr. miss rate za 256K cache: direct~0.002 2way~0.000003



© Kovač, Basch, FER, Zagreb

47

Skupovna asocijativnost

- Ako traženi podatak/blok nije u priručnoj memoriji, mora se izbaciti jedan od postojećih blokova iz skupa kojemu traženi blok pripada
- Kod direktnog preslikavanja je postojao samo jedan blok, pa se on izbacivao i zamjenjivao traženim blokom. Ovdje postoji više potencijalnih blokova koji se mogu izbaciti što komplicira zamjenu blokova.
- Postoje brojni algoritmi zamjene kojima se izabire koji će se blok izbaciti da bi se ubacio blok s traženim podatkom:
 - Najjednostavniji takav algoritam zamjenjuje blok koji se najdulje vrijeme nije koristio (engl. LRU - Least Recently Used)
 - Izvedba algoritma je jednostavna za skupove s 2 bloka. Dodaje se samo jedan bit u dodatna polja bloka. Taj bit se postavlja kad procesor zatraži podatak iz tog bloka, a bit u susjednom bloku se briše. Za izbacivanje se odabire blok u kojemu bit nije postavljen.
 - Za skupove s više blokova izvedba se komplicira
- Alternativni algoritam je zamjena slučajno odabranog bloka. Ovo je jednostavan algoritam, ali daje veći omjer promašaja

© Kovač, Basch, FER, Zagreb

42

Puna asocijativnost

- Ako proširimo ideju skupovne asocijativnosti na cijelokupnu priručnu memoriju, tako da cijela priručna memorija predstavlja samo jedan skup u koji možemo pohranjivati blokove, dolazimo do najfleksibilnijeg i najefikasnijeg načina pohranjivanja blokova*
- Ovo se naziva puno asocijativno preslikavanje
- Na ovaj način broj promašaja maksimalno se smanjuje
- Na žalost, cijena sklopova za provjeru dodatnih polja je izuzetno velika, kao i njihova složenost pa vrijeme traženja bloka također raste
- Ova organizacija koristi se zato samo kod manjih priručnih memorija koje trebaju maksimalne performanse

* barem teorijski, a u praksi postoje neka ograničenja

© Kovač, Basch, FER, Zagreb

44

Rekapitulacija preslikavanja

- Direktno preslikavanje
 - Jedan blok samo na jedno točno određeno mjesto
 - Velik broj promašaja
 - Jednostavna i jeftina organizacija
- Skupovna asocijativnost s n-blokovima
 - Jedan blok može se pohraniti na n mesta unutar skupa
 - Značajno smanjenje broja promašaja
 - Potreban je algoritam zamjene blokova
 - Dodatac cijena sklopova za usporedbu
- Puna asocijativnost
 - Jedan blok bilo gdje u priručnu memoriju
 - Maksimalno smanjenje broja promašaja
 - Potreban je algoritam zamjene blokova
 - Vrola visoka složenost sklopova i visoka cijena
 - Praktično samo za priručne memorije manjeg kapaciteta

© Kovač, Basch, FER, Zagreb

46

Višerazinska organizacija priručne memorije

- Kod modernih procesora razlika između brzine priručne memorije i brzine glavne memorije je izuzetno velika te se uvode dodatne razine priručne memorije
- Tako se npr. osim priručne memorije prve razine (L1 cache), može staviti i druga razina priručne memorije (L2 cache) koja je nešto sporija od razine L1, no još uvjek brža od glavne memorije. Zatim se može staviti i priručna memorija treće razine (L3 cache) itd.

© Kovač, Basch, FER, Zagreb

48

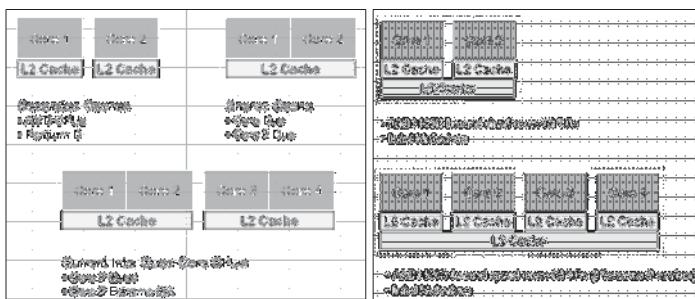
- Intelova arhitektura Itanium 2 (2006.)
 - 32KB L1, 256KB L2, 3MB L3
- Intel Pentium 4 Processor Extreme Edition supporting Hyper-Threading Technology (2006.)
 - 12-K micro-op trace cache + 16-KB L1 data cache, 512KB L2 cache, 2MB L3 cache
- Nehalem (2010.)
 - L1: 32 kB za naredbe i 32 kB za podatke
 - Izbačen micro-TLB koji je postojao kod mikroarhitekture Core
 - Reducirana asocijativnost (u odnosu na mikroarhitekturu Core) sa 8-strike na 4-struktu kako bi kašnjenje (latency) bilo manje
 - L2: 256 kB
 - L3: 8MB
 - Sadrži kopije L1 i L2
- Ivy Bridge (2012.)
 - L1: 32 kB za naredbe i 32 kB za podatke
 - L2: 256 kB
 - L3: 8MB

© Kovač, Basch, FER, Zagreb

49

Primjeri

- Promjena u hijerarhiji priručne memorije između mikroarhitekture Core i mikroarhitekture Nehalem

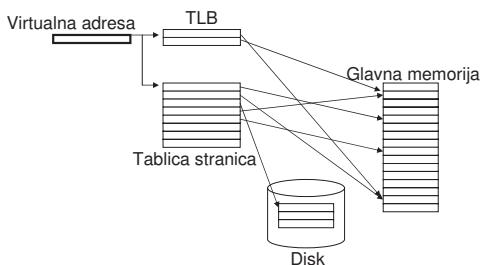


© Kovač, Basch, FER, Zagreb

51

Virtualna memorija

- Iz povijesnih razloga blok virtualne memorije naziva se stranica (page)
- Ako stranica nije u glavnoj memoriji, dolazi do greške na stranici (page fault), što je analogno promašaju kod priručnih memorija
- Tablica stranica (page table) sadrži translacije virtualnih adresa u fizičke. Čuva se u memoriji za svaki program
- Procesori imaju i posebnu priručnu tablicu (translation lookaside buffer, TLB) koja sadrži najčešće referencirane virtualne adrese (16-512 adresa) tako da se ne troši vrijeme na pristup tablici stranica



© Kovač, Basch, FER, Zagreb

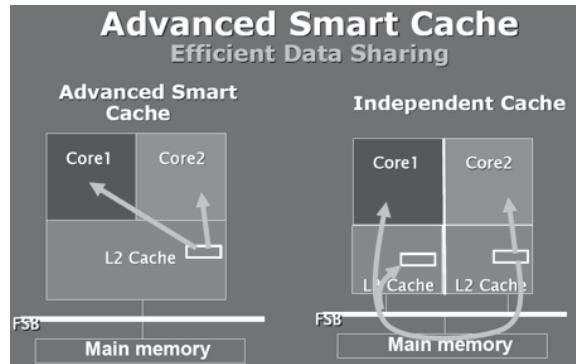
53

Osnovna pitanja o priručnoj memoriji

- Kako se osigurava istovjetnost podataka u različitim razinama memorije prilikom pisanja?
 - Pisanje s proslijedivanjem ili pisanje pri povratku.
- Koje su prednosti asocijativne organizacije?
 - Značajno smanjenje broja promašaja.
- Koji su nedostaci asocijativne organizacije?
 - Porast cijene zbog sklopolja za provjeru, povećanje vremena pogotka, kompleksnost zamjene bloka.

© Kovač, Basch, FER, Zagreb

- Reklama za memorijsku hijerarhiju Intel Core 2 microarchitecture



© Kovač, Basch, FER, Zagreb

50

Virtualna memorija

- Povijesno su postojala dva razloga za uvođenje virtualne memorije:
 1. Glavna memorija može se gledati kao neka vrsta "priručne" memorije u odnosu na znatno sporiju masovnu memoriju (posebno važno u nekadašnje vrijeme kad su glavne memorije bile relativno malog kapaciteta)
 2. Efikasno dijeljenje i upravljanje memorijom za više programa koji se istovremeno izvode (današnja funkcija virtualne memorije)
- Svakom programu dodjeljuje se zaseban **adresni prostor** (niz virtualnih memorijskih adresa rezerviranih samo za taj program)
- Memorijski sustav obavlja **translaciju** programskega adresnog prostora (virtualnih adresa) u stvarne, fizičke adrese
- Na taj način se obavlja **zaštita** adresnog prostora nekog programa ili OS-a

© Kovač, Basch, FER, Zagreb

52

Osnovna pitanja o priručnoj memoriji

- Kako se određuje gdje blok može biti smješten?
 - Jedno mjesto (direktno preslikavanje), nekoliko mjesta (skupovna asocijativnost) ili bilo koje mjesto (potpuna asocijativnost).
- Kako se pronalazi blok?
 - Direktno preko nižih bitova adrese bloka (direktno preslikavanje), ograničenim pretraživanjem dodatnih polja (skupovna asocijativnost) ili pretraživanjem svih dodatnih polja (potpuna asocijativnost).
- Koji se blok zamjenjuje u slučaju promašaja?
 - Najmanje korišteni ili slučajnim odabirom.

© Kovač, Basch, FER, Zagreb

54

55