

UVOD U ARHITEKTURU RAČUNALA

⇒ rad procesora:

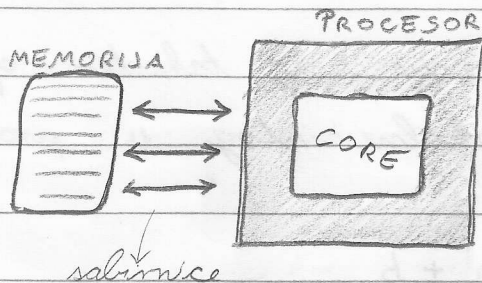
- 1.) dohvat naredbe
- 2.) dekodiranje naredbe
- 3.) izvođenje naredbe

⇒ VON NEUMONOVA arhitektura memorije:

- ↳ i podaci i naredbe se nalaze u istoj memoriji
- ↳ jeftinija, ali sporija

⇒ HARVARDSKA arhitektura memorije:

- ↳ podaci i naredbe se nalaze u različitim memorijama
- ↳ brža, ali skuplja memorija



⇒ postoje ADRESNE
i PODATKOVNE
sabirnice

⇒ memorija se ne mora sastojati od samo jedne fizičke komponente

⇒ **STOG** ⇒ struktura podataka koja radi po načelu LIFO ("last in, first out")

↳ na engleskom: **STACK**

↳ ta memorija se ne navigira pomoću adresa varijabli, već ima jedan "Stack Pointer" (SP) koji uvijek pokazuje na vrh stoga

↳ stog se jako često koristi kod procesora

↳ važno je znati koliko je stog velik kako nebi prekril neke važne podatke

⇒ **ARHITEKTURE S OBZIROM NA DOHVAT OPERANDA:**

a) **STOGOVNA**

↳ upotrebljava stog na pripremanje operanda

↳ to je mali stog na procesoru koj je jako brz

↳ na procesoru se nalazi i pokazivač stoga

↳ rezultat operacije se također sprema na taj mali, lokalni stog u procesoru

Pr: računanje $x = a + b$

push a

→ 4 naredbe

push b

→ 7 pristupa memoriji

add

pop x

↳ stogovna arhitektura je vrlo jednostavna

↳ danas se u izvornom obliku više ne koristi

↳ brzo dekodiranje i brzo izvrštavanje

↳ izvrštavanje nije efikasno (velik broj naredbi) te velik broj pristupa memoriji

↓) AKUMULATORSKA

↳ ima samo jedan registar koji se naziva akumulator

↳ jedan operand se uvijek nalazi u tom registru

PR: računanje $x = a + b$

load a → 3 naredbe

add b → 6 pristupa memoriji

store x

↳ jednostavna struktura

↳ danas se rijetko koristi

↳ naredbe, uvođenja i preraditelji su vrlo jednostavni

↳ loše je što čuva samo jedan međurezultat u procesoru, te izvrštavanje nije efikasno

c) REGISTAR - MEMORIJA (slajd 9)

d) REGISTAR - REGISTAR (slajd 12)

⇒ pristup memoriji je uvijek sporiji a "cache" memorija procesora, stoga nam je cilj da varijable pohranimo u registar procesora, a tek na kraju izvođenja vraćamo rezultat

⇒ ARHITEKTURE S OBZIROM NA NAREDBE:

a) CISC

↳ procesori su brzo napredovali a
druhom na naredbe

više tranzistora → više naredbi

↳ tako su se procesori sve više
prilagođivali višim programskim
jezikima, pa je izvršavanje programa
bilo brže, lakše, ali je izvođenje
bilo sporije

↳ specijalizirani registri

b) RISC

↳ zbog kompleksnost CISC-a, razvijem je RISC kako bi se naredbe izvršavale jako brzo, ali bi ih bilo nešto više

↳ naredbe su jako jednostavne

↳ registri su međusobno ravnopravni

↳ jeftiniji nego CISC

⇒ REDOSLED ZAPISA:

↳ naš podatak je 1234

a) little-endian

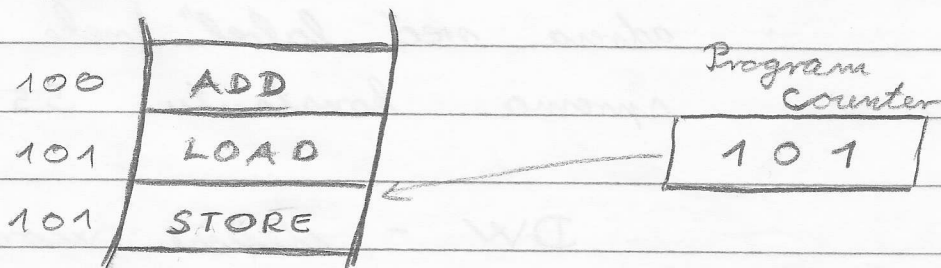
b) big-endian

1000	34
1001	12

1000	12
1001	34

⇒ PROGRAMSKO BROJILO:

↳ registar unutar procesora koji u sebi pamti adresu sledeće naredbe koju procesor treba izvršiti



⇒ STROJNI KOD:

↳ niz nula i jedinica koj procesoru govori koji naredbu treba izvršiti

↳ strojni kod je obično širine riječi tako da za svaku naredbu trošimo jedan pristup memoriji (ovo vrijedi samo za RISC procesore)

↳ kod CISC procesora, duljina strojnog koda razlikuje se od duljine riječi (fleksibilniji, ali kompleksniji)

⇒ NAREDBE PROCESORA:

↳ ima ih jako puno

↳ neke skupine naredbi:

a) aritmetičko-logičke

b) memorijske

LOAD register, (adresa)

STORE register, (adresa)

- korisna naredba:

↳ BROJ1 DW 55

na adresu BROJ1 (ovo nije adresa već "label" nako adrese)
sprema konstantu 55

DW - define word

c) upravljačke

- ↳ koriste nam na "petljane" programa
- ↳ te naredbe mijenjaju stanje programskog broja
- ↳ najčešća naredba:

NAREDBA SKOKA (GRANANJA)

i) bezuvjetni skok - preskače dio koda

ii) uvjetni skok - preskače dio koda ako je uvjet zadovoljen

JP adresa

JP-uvjet adresa

↳ napomena: procesor ne može uspoređivati dva broja, već se uvjet bit vrlo primitivni

⇒ KAKO USPOREDITI DVA BROJA?

↳ dva broja najjednostavnije uspoređujemo pomoću naredbe oduzimanja i u skladu s rezultatom postavljamo određene zastavice

↳ neke zastavice: (jako, jako sesto)

C - prenos

V - preliv (overflow)

Z - nula

N - predznak

↳ procesor uvijek ima ta 4 bitakula na zastavice

↳ kako odrediti rastoricu V ? (slajd 13)

$$V = C_{m-1} \oplus C_m$$

⇒ na svakom procesoru, rastorice su spremjene u STATUS REGISTER-u (kratice SR)

⇒ pogledaj uzete na slajdu 17, slajdu 18, slajdu 19

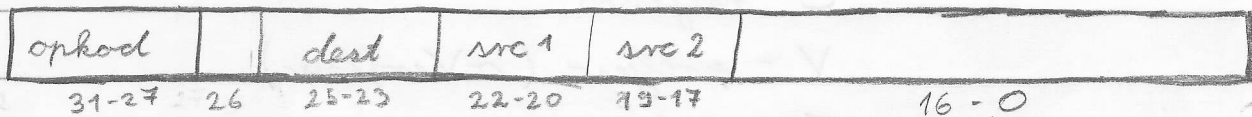
⇒ broj naredbi skoka koristiti s labelama:

JP NP DALJE

⋮
DALJE

d) općenite

⇒ ako imamo operacijski kod od 32-bita, onda najčešće nemamo 5 bitova na definiranje koje naredbu izvršavamo



⇒ vidimo da nam je ostalo samo 16 bitova na adresu, pa ne možemo adresirati

svih 2^{32} memorijskih lokacija

↳ taj problem ćemo kasnije riješiti
pomoću ostalih procesorskih naredbi

⇒ INSTRUKCIJSKI REGISTAR:

↳ služi za čuvanje instrukcijskog koda
naredbe

⇒ DATA REGISTAR:

↳ za memorijske naredbe, privremeno
sprema podatke prije spremanja u
memoriju

⇒ ADRESNI REGISTAR:

↳ čuva bitove adrese za naredbe

PROŠIRENJE NAREDBI

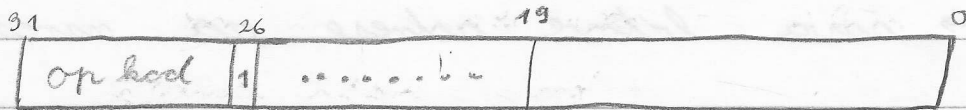
⇒ RADVA S KONSTANTAMA:

↳ preko ADD naredbe:

- 26. bit govori je li u pitanju klasična ADD naredba ili naredba s konstantom (ako je 1 onda računamo s konstantom)

PR: ADD R1, 55, R0

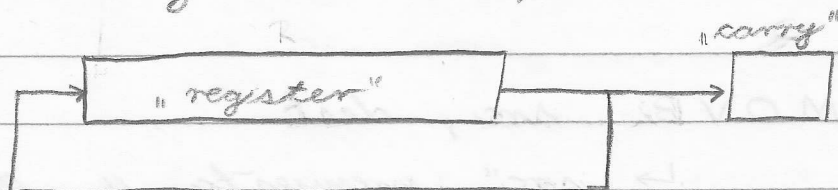
↳ mi možemo razprati broj od -2^{19} do $2^{19}-1$ jer nam zapis konstante izgleda ovako:



↳ procesor automatski proširuje 20 bitni broj na 32 bitni tako da na preostala mjesta kopira bit predznaka

⇒ POSMACHI I ROTACIJE:

- ↳ logički pomak ulijeva i udesno
- ↳ aritmetički pomak udesno
- ↳ rotacije ulijeva i udesno (koja ne ide kroz carry zastavicu)



⇒ drugi operand govori za koliko mjesta pomičemo, odnosno rotiramo register

⇒ PROŠIRENJA ZA VIŠESTRUKU PRECIZNOST:

- ↳ ADC - add with carry
- ↳ SBC - subtract with carry

⇒ NAREDBA USPOREDBE:

- ↳ CMP src1, src2
- ↳ primjetiti da je isto kao i naredba odumiranja samo što nikad ne sprema rezultat već samo modificira zastavice

REGISTARSKÉ

NAREDBE

⇒ naredba **MOVE**:

MOVE src, dest

↳ „src“ premješta u „dest“

↳ „src“ može biti SR, R0-R7 ili 20-bitna konstanta (prednosažno proširena)

⇒ dobro za podatke:

↳ ako u registru želimo neku konstantu veću od 2^{19} ili manji od 2^{-19} , možemo tu konstantu definirati u memoriji (DW), povratiti pomoću naredbe **LOAD**

⇒ strojni kod registarskih naredbi:



0 - registar
1 - konstanta
1 - rad sa SR (source-register)
1 ili 0 → kopira se u/iz status registra

PROŠIRENJE MEMORIJSKIH NAREDBI

⇒ u strojnom kodu imamo 20 bitova koje možemo koristiti za adresiranje

↳ time možemo direktno adresirati
1 MB od 4096 MB (4 GB)

↳ problem rješavamo ovako:

- koristiti čemu jedan od
registara opće namjene za
adresiranje

LOAD R3, (R1)

↳ u registar R3 upiši podatak
iz memorije koji se nalazi na
adres koji je upisan u R1

↳ stoga, R1 se u nekom smislu
ponaša kao pokazivač

⇒ u strojnom kodu memorijskih naredbi, imamo
slobodne bitove od 0 do 19

↳ to koristimo za izražavanje konstante
nekej adresi

LOAD R3, (R1 + 20)

"odmah"

⇒ zbog kompatibilitet sa starijim procesorima, imamo naredbe sa šifrom 8 bitova i 16 bitova iz memorije:

↳ ove naredbe se ponašaju analogno ostalim (normalnim) 32-bitnim memorijskim naredbama

LOADB R1, (300)

↳ "učita jedan bajt sa adrese 300"

STOREH R1, (300)

↳ "spremi dva bajta iz R1 na adresu 300"

⇒ dobro na radatihe:

↳ riječ imaju adrese djeljive s 4

↳ poluriječ imaju adrese djeljive s 2

↳ bajtovi nemaju ograničenja adresa

PR: Kada napišemo `LOAD R1, (103)` procesor čita 32-bita, pa zna da riječ ima adresu djeljivu s 4 pa maskira zadnja dva bita adrese u nulu:

0001 0000 0011

AND 1111 1111 1100

0001 0000 0000

⇒ 100₍₁₀₃₎

LOAD R1, (103)

isto kao:

LOAD R1, (100)

⇒ naredbe za rad sa stogom:

PUSH - stavlja podatke iz registra na stog

POP - uzima podatke iz registra i stavlja ga na stog

↳ stog se nalazi unutar onog našeg memorijskog prostora od 4 GB

↳ kako bi ovaj gođo je stog, moramo imati SP (stack - pointer)

↳ kako bi manipulirali stogom, trebamo i naredbe za rad sa stack - pointerom

↳ kako nebi dodatno komplicirali naredbe, mi ćemo registar R7 proglašiti stack - pointerom

⇒ stack - pointer pokazuje na zadnji upisan podatak

PUSH src

↳ radi ovako:

$R7 - 4 \rightarrow R7$

$src \rightarrow (R7)$

POP dest

↳ radi ovako:

$(R7) \rightarrow dest$

$R7 + 4 \rightarrow R7$

⇒ vidimo da je naš stog padajuć

⇒ SHUFFLER - sklop koji vrši poravnavanje
podataka u slučajju čitanja
8-bitova ili 16-bitova

⇒ naredba relativnog skoka:
JR - učitaj label

label DW 5

↳ ova naredba povećava program
counter na vrijednost koja se nalazi
u memoriji označenoj sa label (u
ovom slučaju je to broj 5)

POTPROGRAMI

⇒ za pozivanje potprograma:

CALL label

↳ rad ovako:

$$\left. \begin{array}{l} SP - 4 \rightarrow SP \\ PC \rightarrow (SP) \end{array} \right\} PC \rightarrow \text{stog}$$

ADRESA $\rightarrow$ PC

⇒ za vraćanje iz potprograma:

RET

↳ uzme povratnu adresu sa stoga
(ustvari, uzme podatak sa vrha stoga)

↳ rad ovako:

$$\left. \begin{array}{l} (SP) \rightarrow PC \\ SP + 4 \rightarrow SP \end{array} \right\} \text{stog} \rightarrow PC$$

⇒ ove naredbe se ustvari nazivaju JP naredbe

⇒ ove naredbe mogu imati i usjet:

CALL - usjet label

RET - usjet

ASSEMBLER

⇒ postopek assembleri:

- a) JEDNOPROLAZNI
 - b) DVOPROLAZNI
 - c) TROPROLAZNI
 - d) ČETVEROPROLAZNI
- } makroassembleri

↳ jednoprolazni assembler ne more prevesti ero:

JP B

§§

B

→ kada assembler pride do ove linije, ne zna na kojoj je adresi B, pa u strojni kod JP naredbe ne more unjeti adresu

↳ dvoprolazni procesor radi internu tablico na prvom prolazu o adresama na koji pokazuje pojedina labela

⇒ PSEUDONAREDBE:

- ↳ to su naredbe za assemblyski prevoditelj
- ↳ uvodi ih assemblyski prevoditelj, a ne procesor
- ↳ one govore prevoditelju kako treba obavljati prevodenje
- ↳ jedna od tih naredbi je DW (Define - Word)

LINUX KOMANDE

- ⇒ `pwd` - print working directory
- ⇒ `cd` - change directory
- ⇒ `ls` - list directory
- ⇒ `kill -9` - gas program pod željenim brojem
- ⇒ `rm` - brisanje željene datoteke
- ⇒ ako na kraj naredbe dodamo znak `&`,
radimo funkcionalnost komandne linije
- ⇒ relativan put je isto kao u DOS-u
- ⇒ `more` - ispisuje sve o željenoj datoteci
- ⇒ `ps` - ispisuje sve programe koje procesor izvršava

NAČINI ADRESIRANJA

a) REGISTRARSKO

↳ podatak ili rezultat su u registru

b) NEPOSREDNO

↳ podatak se račuje neposredno kao broj (ili labela)

c) APSOLUTNO

↳ podatak se račuje brojem koji predstavlja adresu podatka

d) RELATIVNO

↳ kod naredbi koje su napravljene za relativno adresiranje (npr. JR)

e) REGISTRARSKO-INDIREKTNO

↳ u registru se nalazi adresa koju koristimo

f) REGISTRARSKO-INDIREKTNO S ODMAKOM

↳ u registru se nalazi adresa te njoj dodajemo željeni odmak

↳ FRISC ima samo ovo, a ne registrarsko-indirektno jer je tamo odmak nula

g) IMPLICITNO

↳ kod rada sa stogom

↳ ne navodimo adresu, već se ona postaramijem zbog same naredbe

POTPROGRAMI

⇒ stack se nalazi na kraju adresnog prostora od 64k

⇒ prije korištenja stacka, moramo inicijalizirati stack - pointer, to na 10000 što je:

FFFC + 4

↳ to je zbog toga što STORE umanjuje stack - pointer na 4, a mi prvo moramo na stack staviti nešto kako bi nešto uzeli

⇒ NAPOMENA: potprogram moraju čuvati vrijednost registara koje ne želimo izmijeniti

⇒ ako trebamo više registara, onda spremamo podatke iz tih registara na stog te ih po povratku samo uzimamo sa stoga

⇒ PRIJENOS STOGOM:

↳ ako vraćamo ili šaljemo vrijednost u potprogram preko stoga, morat ćemo prelagirati stog jer CALL, RET stavljaju (uzimaju) nešto sa stoga

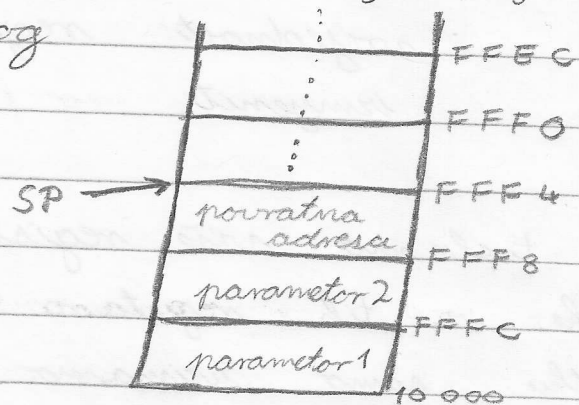
⇒ NAPOMENA: naredba POP ne briše podatak sa stoga, već samo nujenja stack - pointer

⇒ ako želimo učit nek podatak sa stoga koj nije na vrhu stoga, morat ćemo "kalkulirat" koliko treba dodati stack - pointeru

⇒ DOBRO ZA PROGRAMIRANJE:

↳ najčešće se podaci u potprogram šalju pomoću stoga, a vrijednost iz potprograma se vraćaju preko nekog registra

↳ na kalkuliranje sa stogom je dobro crtati sam stog



↳ na kraju potprograma, dobro je "ukloniti" parametre sa stoga kako bi "počistili smeće", a to se najbolje učiniti tako da uvedemo stack - pointer na odredeni broj (npr. `ADD SP, 8, SP`)

⇒ REKURZIVJE - pogledaj malo po prezentaciji,
ali neće biti na ispitu

⇒ MAKRONAREDBE:

- ↳ imaju istu namjenu kao i potprogrami
- ↳ assembler koji podržavaju MACRO-e moraju biti višepolarni

- ↳ isto kao i u C-u, assembler samo prekopira dio koda na mjesto naziva neke makronaredbe pod tim imenom

- ↳ MACRO-i su nešto brži od potprograma (jedina prednost), ali zauzimaju više mjesta ako ga više puta pozivamo / kopiramo

⇒ dolazi na programiranje:

- ↳ kod oblikovanja NBC brojeva, ako je CARRY restarica postavljena, izlazi smo iz sprega

- ↳ kod oduravanja NBC brojeva, imamo grešku ako je BORROW restarica postavljena - to zapravo znači da CARRY nije postavljen jer je $B = \bar{C}$

PROCESOR FRISC KAO KOMPONENTA

⇒ naš procesor trebamo poveriti sa vanjskim svijetom kako bi vidjeli rezultate programiranja

⇒ SABIRNICE:

- a) adresna
- b) podatkovna
- c) upravljačka

⇒ sabirnice mogu biti:

- a) ulazne
- b) izlazne
- c) dvosmjerne

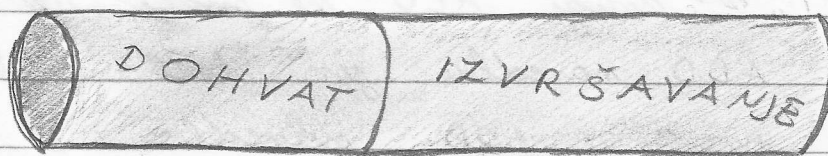
⇒ SABIRNICA - spozna put koj poveruje više uređaja, a sastoji se od više vodiča

⇒ PROTOKOL - redoslijed svih koraka u komunikaciji

PROTOČNA ARHITEKTURA

⇒ protočna arhitektura daje vremensku dimenziju izvodenju naredb

⇒ PROTOČNA ARHITEKTURA FRISC-a:



⇒ naredbe po trajanju djelovanja:

- a) jednosklusne
- b) dvosklusne (memorijske i upravljačke)

⇒ tipovi hazarda:

- a) strukturni - javlja se zbog strukture procesora (npr. Von-Neumannova arhitektura)

↳ naredbe koje ga uzrokuju:
STORE, LOAD, PUSH, POP

- b) upravljački - javlja se zbog upravljačkih naredbi jer se dohvat naredba koja se ne treba izvršiti

↳ naredbe koje ga uzrokuju:

JP, JR, CALL, RET, HALT

↳ broj ciklusa kod naredbi govori o izvršenju ujeta

c) ~~podatkovni~~ - javlja se ako dve naredbe
nisi neovisne jedna o drugoj

↳ NAPOMENA: ne javlja se kod procesora

FRISC

PR: ako imamo dva ALU-a:

ADD R1, R2, R3

ADD R3, R5, R6

↳ drugi ALU ne može izvršiti druge

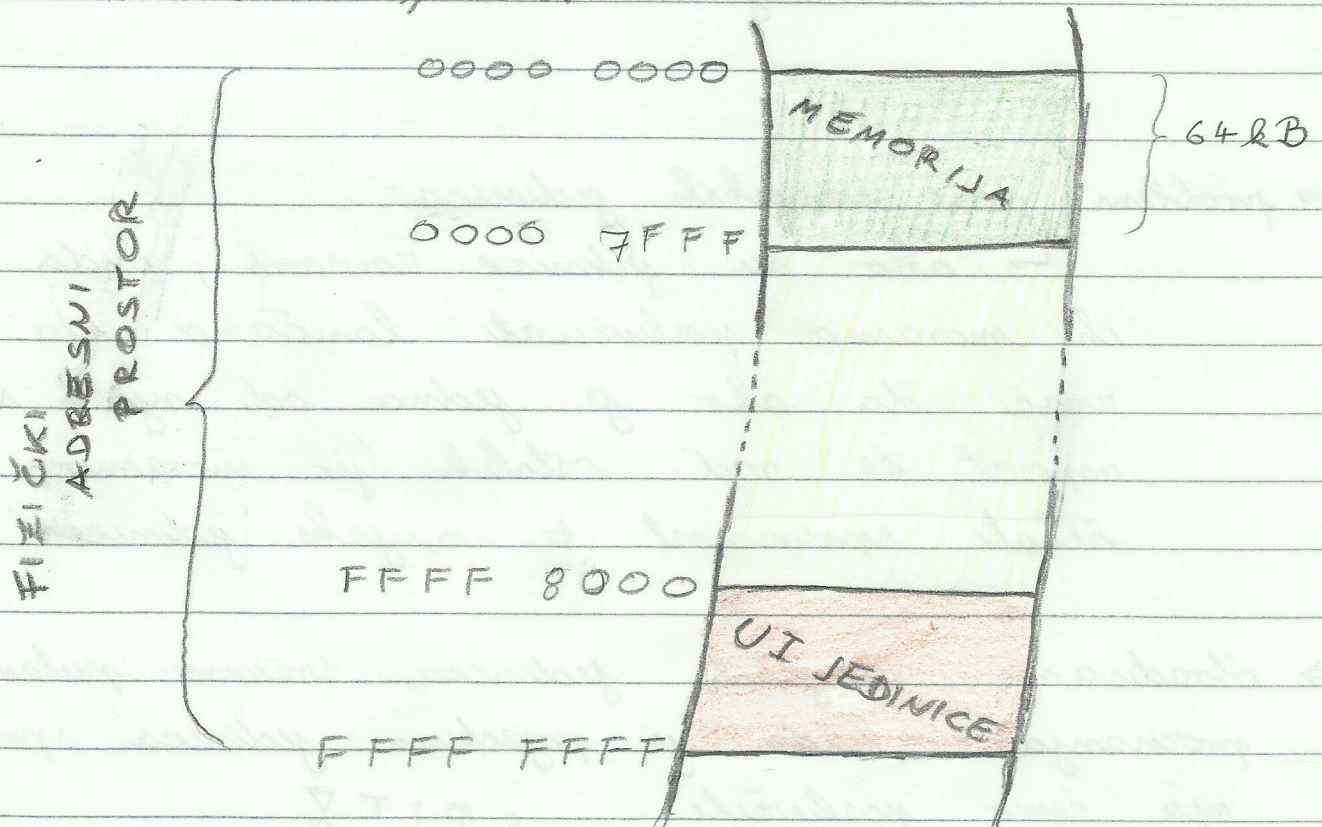
ADD naredbu jer treba čekati rezultat
prve ADD naredbe

POVEZIVANJE RAČUNALA

S OKOLINOM

→ ulazno - izlazne jedinice stavit ćemo u prostor fizičkog adresnog prostora zbog jednostavnijeg adresiranja, a mi isključivo koristimo samo 64 kB memorije od mogućeg 4 GB što možemo adresirati

↳ naše LOAD / STORE naredbe stoga mogu pristupiti ulazno - izlaznim jedinicama te ne trebamo posebne naredbe kao kod procesora koji ne smještaju ulazno - izlazne jedinice u fizički adresni prostor



⇒ PRIJENOS PODATAKA:

a) besurjetni - jednostavan i brz

- ne provjerava je li vanjska jedinica spremna primiti podatke
- većina prijenosa podataka s vanjskim jedinicama je besurjetna

b) urjetni - ima listu adresa stanja koji govori je li vanjska jedinica spremna za komunikaciju s procesorom.

- listu adresa se nalazi na barok adresi + 14

↳ stoga, urjetna vanjska jedinica razumna s adresnih lokacija

⇒ problem više vanjskih jedinica:

↳ ako su jedinice razvijene, onda ih moramo posluživati lančano što znači da ako je jedna od njih spora usporit će rad ostalih jer moramo čekati spremnost te vanjske jedinice

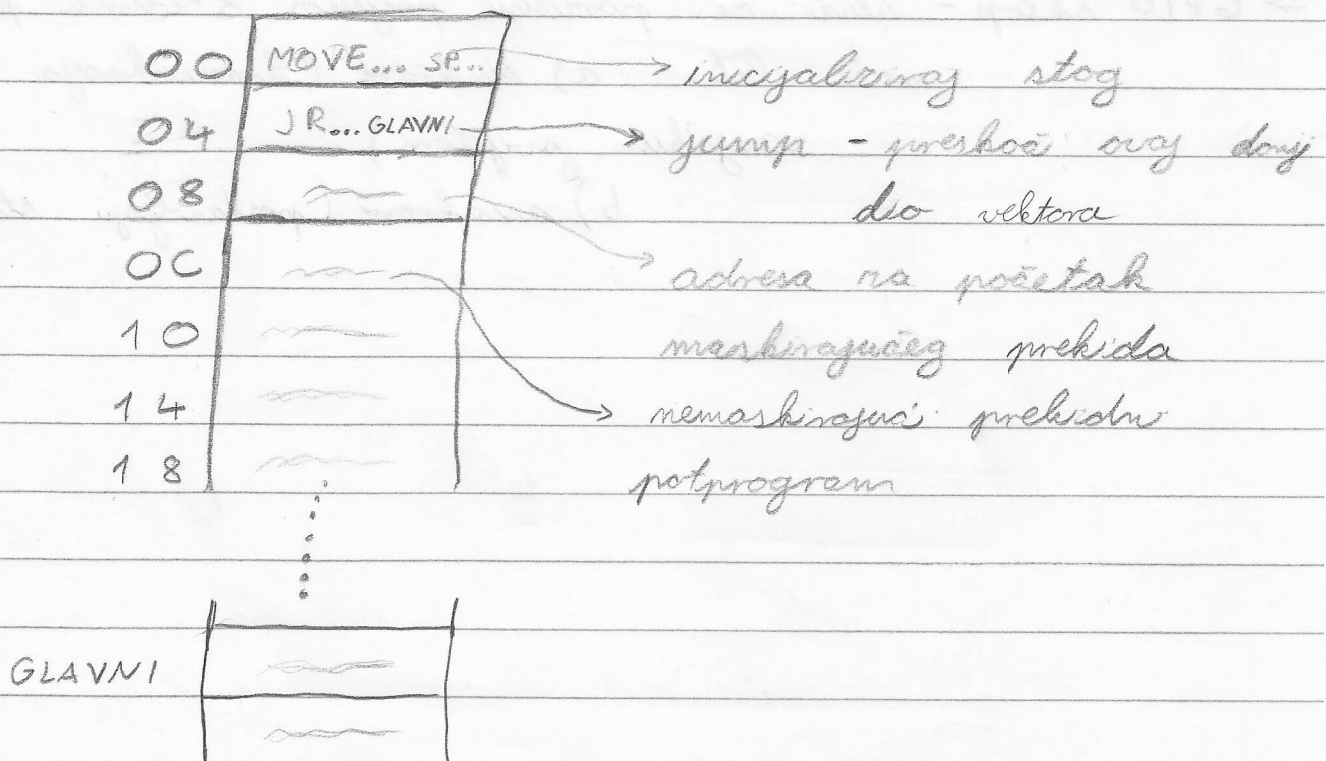
⇒ održavanje vanjskih jedinica vršimo putem programiranja - kada je pojedina jedinica spremna mi ćemo poslužiti

ISPIT!

PREKIDNI PRIJENOS

- ⇒ vanjska jedinica obavljajući procesoru da je ona spremna za prijenos podataka to procesor prekida izvođenje glavnog programa, posluživši vanjsku jedinicu, to nastavlja izvođenje programa
- ↳ ovakav način rada je vrlo čest u današnjim sustavima
 - ↳ prioritet prekida su vrlo bitni jer nam određuje brzinu rada našeg računalnog sustava

- ⇒ prekidan potprogram mora spremati registre i tastaricu



DODATNO O VANJSKIM JEDINICAMA

⇒ CT sklop - sklop koji je sličan vanjskoj jedinici, ali se on može programirati

- on, na primjer, može imati ulogu generiranja prekida svakih 5 sekundi i tako možemo izvršavati nešto točno u određenom vremenu (umjesto petlje koja se zove "m"-puta)

CR - računi kod naredbi koje CT izvršava
↳ koristimo najniža tri bita

⇒ GPIO sklop - služi za paralelni prijem 8-bitnih podataka
- može bit:
a) sinkroni (komunikacija s vanjskim svijetom)
b) asinkron (postavljeni stanje)

DIRECT MEMORY ACCESS

⇒ króće DMA

⇒ imaju ga ne jedinice koje prenose velik broj podataka

⇒ ideja je odvojiti procesor od trošenja vremena na takav stalni prijenos podataka

⇒ DMA kontroleri komuniciraju s procesorom pomoću signala:

a) BREQ - bus request

b) BACK - bus acknowledged

⇒ procesor je za vrijeme prijenosa neaktivan

⇒ VRSTE PRIJENOSA:

a) zaustavljanje procesora

↳ dok se prijenos izvršava, procesor čeka

b) kratak ciklus

↳ jedan ciklus procesor koristi samoinicijaciju, a jedan DMA kontroler

↳ ima guljenje vremena

c) blokovski

↳ prenosi se jedan blok, pa se vraća upravljanje procesoru itd.

↳ najčešći koristeći

d) multiplexiran

↳ DMA ima kontrolu sve dok procesor ne treba memoriju s podacima

↳ najčešći kod Harvardske arhitekture

⇒ DMA kod FRISC-a može posluživat samo
berunjetne vanjske jedinice jer ne pružava
spremnost

⇒ DMA kontroler treba na početku programa inicijalizirati

ARM PROCESORI

⇒ skraćenica: Acorn RISC Machine

↳ prvi procesor projektiran od skupine istraživača na Cambridgeu

↳ kasnije se dio tih istraživača odvojio u kompaniju nazvanu: Advanced RISC Machines

↳ ta kompanija ne proizvodi procesore već samo prodaje arhitekturu (intelektualno vlasništvo)

⇒ mi ćemo se baviti ARM7 porodicom što je jedan od najpopularnijih ARM procesora

ARM 7 - PROGRAMSKE KARAKTERISTIKE

⇒ RISC procesor (s malo CISC karakteristika)

⇒ "LOAD-STORE" arhitektura

↳ samo "load" i "store" pristupaju memoriji
(nema PUSH, POP, CALL, ...)

⇒ sve naredbe su iste dužine od 32-bita

⇒ ima ujetno izrođenje svih naredbi

↳ što za ubrzanje manjih odvajaka koda
(nema remetenja protočne arhitekture u neim slučajevima)

⇒ aritmetičko - logičke naredbe ne mijenjaju redoslijed po
"default-u"

⇒ mogućnost proširenja skupa naredbi

↳ kada ARM naleti na nepoznatu operaciju,
poziva se podprogram i izvršava se "nova"
naredba

⇒ protočna arhitektura od tri razine

⇒ memorijska arhitektura je fleksibilna

⇒ ima "Barrel Shifter" na drugom operandu koji ulazi
u ALU, pa ARM7 nema posebne naredbe
za rotaciju i pomak

↳ inovacija jer vrlo često trebamo
množiti ili dijeliti s potencijom broja dva
kod kretanja kroz nekakvo polje

⇒ može raditi s bajtovima, polubajtovima, riječima

⇒ mogućnost predviđenog proširivanja i proširivanja s nulom

⇒ ARM ima ukupno 37 registara širine 32 bita

↳ 31 registara opće namjene od kojih može vidjeti 16 registara u isto vrijeme

↳ 6 registara su statusni (posebno ra-
vnan način rada)

⇒ registar R15 je programsko brojilo

⇒ registri u drugom načinu rada imaju kopije
(najčešće R13 i R14)

R13 → priklon prelaska načina rada
čuva nam pokazivač stoga

R14 → povratna adresa priklon
prelaskom načina rada

⇒ priklon „fast-interrupt“ načina rada, registri
od R8 - R14 su rezervirani za spremanje
konteksta što je brže od „ručnog“ spremanja
konteksta, još dodatno brže od spremanja
u memoriju

NAREDBE PROCESORA ARM7

- ⇒ naredbe Multiple LOAD/STORE uvijek registre s manjim indeksima učitava/sprema s/na nove memorijske lokacije bez obzira na redoslijed navođenja u naredbi
- ⇒ konstante koje upisujemo moraju se moći dobiti parnim brojem pomaka u jednu ili drugu stranu
- ⇒ Iz potprograma se vraćamo:
MOV PC, RL

VANJSKE JEDINICE PROCESORA ARM

⇒ SKLOP GPIO:

- ↳ vrlo složna funkcija kao kod FRISC-a
- ↳ razoma 4 nivoitne memorijske lokacije
- ↳ razlika je što GPIO sklop može raditi istovremeno sa 2 vanjska procesa, a na njih je spojen preko 2 GATE-a:

GATE A:

GATE B:

0 → ulazni način

0 → izlazni način

1 → izlazni način

1 → ulazni način

- ↳ u ATLAS-u, naš GPIO sklop je spojen na termometar (sage vrijednost na bitovima $XPA[0:5]$, a $XPA[6]$, $XPA[7]$ se koriste za sinkronizaciju)

$XPA \rightarrow$ oznaka na sklopu GATE-a A

- ↳ prije korištenja GPIO sklopa, treba izvršiti inicijalizaciju
- ↳ GPIO sklop kod ARM-a ne može generirati prekid (kod FRISC-a je mogao)
- ↳ u ATLAS-u, GPIO je spojen i na LCD prikaznik ($XPB[0:6]$ su podatkovni bitovi, a $XPB[7]$ je "Write Signal")
- ↳ podaci se spremaju u Buffer, te se prikazuju priklon primanja podataka A_{16} (line feed)

⇒ SKLOP RTC:

↳ "Real Time Clock"

↳ ra razliku od FRISC-a, RTC trebamc
nah put ponovno inicializirat kada brojlo
dođe do željene vrijednosti:

↳ brojlo broj od nule prema
višim vrijednostima te se ciklički resetira

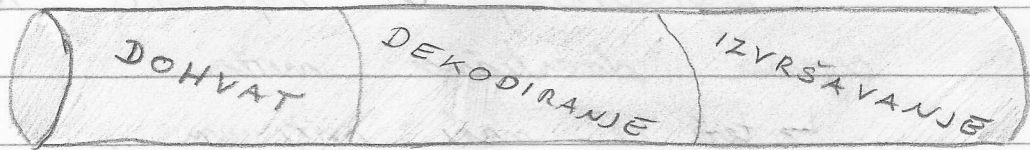
↳ zauzima 5 memorijskih lokacija

↳ nemamo mogućnost resetiranja brojila

↳ nije nužno nah put reinitializirat registar
usporedbe jer njegova vrijednost ostaje nepromijenjena
iako se brojlo došlo do željene vrijednosti

PROTOČNE ARHITEKTURE PROCESORA ARM

⇒ ARM 7 ima protočnu arhitekturu u tri razme:



⇒ naredbe za obradu podataka su jednosobne

⇒ naredbe za prijenos podataka:

↳ uzrokuju strukturni hazard (isto kao kod procesora FRISC)

↳ npr. STR naredba je dvostrukna

LDR naredba je trostrukna

STM naredba traje "n+1" ciklusa, ako je "n" broj podataka koje spremamo

⇒ naredbe grananja:

↳ uzrokuju upravljački hazard (isto kao kod procesora FRISC)

↳ predviđanje grananja:

a) STATIČKO - isloporški kažemo procesoru koje naredbe učitava (npr. ako je skok prema gore, onda je to "for" petlja)

b) DINAMIČKO - na temelju look-up tablice i statistike predviđamo hoće li se izvršiti skok ili ne

MEMORIJSKI SUSTAVI

⇒ iz manjih memorijskih modula (npr. 64 kB) možemo dobiti veliku memoriju ako ih povežemo

↳ naravno, to usporava rad, ali je cijena drastično niža

↳ teži se naći optimum

⇒ ubrzanje memorije:

a) PREPLITANJE ("interleaving")

↳ ako imamo npr. 4 bloka memorije, podaci se nalaze u svim 4 bloka

↳ naravno, mi čitamo iz svih 4 bloka istovremeno, pa nam podaci koji trebamo moraju biti sljedivi (to je u velikoj većini slučajeva istina jer procesori najviše trebaju sljedive podatke)

b) HIJERARHIJSKA ORGANIZACIJA MEMORIJE

↳ podaci koji često trebamo spremaju se u brzu memoriju (npr. "cache")

↳ naravno, koristi se i logika koja govori da ako smo učitali podatke s adrese "X", i vrlo vjerojatno ćemo trebati i podatke s adrese "X+1" (ili "X+4"), pa se i taj podatak smjesti u brzu djelove memorije

BRZINA ↑	SRAM	⇒ priručna memorija („cache“)	CIJENA ↑
	DRAM	⇒ glavna radna memorija	
	HD	⇒ masovna („hard-disk“) memorija	

↳ kako procesoru dojaviti gdje se nalaze podatak koji neka naredba treba?

↳ ovo je vrlo kompleksno i time se bavi MEMORY MANAGEMENT UNIT, to je taj „MMU“ puno kompleksniji i zahtjevniji od samog procesora

↳ „MMU“ koristi statistiku: hit/miss

hit → ako je podatak koj trebam u priručnoj memoriji

miss → ako je podatak koj trebam u nekoj sporij memoriji (npr. „hard-disk“)

↳ daljnji problem je: „kako nešto zapisati iz priručne memorije u masovnu ili čak glavnu radnu memoriju?“

↳ danas se najčešće memorija još više djeluje po „LEVEL“-ima te MMU ima još više posla, ali sve funkcionira (vrlo kompleksno)