Увод у организацију и архитектуру рачунара 2

Александар Картељ

kartelj@matf.bg.ac.rs

Напомена: садржај ових слајдова је преузет од проф. Саше Малкова

Процесор

Архитектура скупа инструкција (ISA)

Архитектура скупа инструкција

- Један од основних аспеката архитектуре процесора је скуп инструкција

- Архитектура скупа инструкција има неколико основних аспеката:
  - пројектовање скупа инструкција
  - имплементација
  - перформансе

Број и сложеност инструкција

- По броју и сложености инструкција процесори се деле у три групе:
  - *CISC* – процесори са сложеним скупом инструкција (енгл. *complex instruction set computing*)
  - *RISC* – процесори са редукованим скупом инструкција (енгл. *reduced instruction set computing*)
  - векторски процесори

CISC процесори

- Циљеви:
  - сложена архитектура скупа инструкција (*ISA*)
    - кодирање што сложенијих инструкција у што мање меморије
  - разноврсност операција
  - разноврсност начина адресирања

CISC процесори

- Последице:
  - нови модели процесора уводили су све више и више нових начина адресирања и нових инструкција
  - подржан превелики број сложених инструкција
    - чак се додају неке иснтрукције које се *никада* не користе при програмирању на асемблеру, али омогућавају ефикасније превођење програма писаних на вишим програмским језицима
  - отежано декодирање инструкција

RISC процесори

- Циљеви:
  - једноставна архитектура скупа инструкција
  - обезбеђивање минималног скупа инструкција и начина адресирања
  - повећан број регистара који се могу користити за рачунање

RISC процесори

- Последице:
  - сталнији скуп инструкција
  - једноставно декодирање инструкција
  - скраћивање трајања извршавања операција
    - већина операција у једном или два циклуса
  - једноставнија имплементација процесора

RISC процесори

- Основни принципи дизајна:
  - једноставне операције
  - операције регистар-у-регистар
  - једноставни начини адресирања
    - углавном се користи регистарско адресирање
  - већи број регистара
  - фиксна дужина и једноставан формат инструкција
    - често су инструкције фиксне дужине, поравнате на дужину речи

RISC процесори

- Друге одлике:
  - удвајање магистрале – посебно за податке и инструкције
    - тзв. Харвард архитектура
    - најчешће само на нивоу кеша
  - сви регистри су равноправни по могућностима и перформансама
  - преклапање извршавања инструкција
    - извршавање бар једне инструкције по циклусу
  - висока пропусност системске магистрале

Однос *RISC* и *CISC* процесора

- *RISC* концепти се развијају од 1975. године
- У актуелном стању технологије производње процесора (*CISC*):
  - било је скупо подржавати сложене операције на већем броју регистара, па је зато постојало мало регистара
    - *RISC* приступ омогућава повећавање броја регистара
  - већина инструкција је захтевала сложену имплементацију, па и дугачке циклусе
    - *RISC* приступ поједностављује имплементације и скраћује извршавање инструкција у циклусима
  - сложена имплементација отежава подизање радне фреквенције
    - *RISC* омогућава значајно подизање радне фреквенције

Однос *RISC* и *CISC* процесора

- Анализе програма показују да већина инструкција обавља преношење података између процесора и меморије
- Имплементација сложених инструкција које се ретко извршавају значајно усложњава имплементацију процесора а доноси скромне добитке у перформансама

Однос *RISC* и *CISC* процесора

- Од 1975. па до средине 1990. постоји тенденција да се произвођачи процесора приклањају *RISC* концептима
- Касније постепено усавршавање производних технологија смањује значај *RISC* архитектуре
- Долази до спајања елемената архитектура

Однос *RISC* и *CISC* процесора

- Данас су уобичајене архитектуре које се одликују *CISC* односом према скупу инструкција, а имају већину осталих одлика *RISC* архитектура:
  - велики број регистара, који су практично равноправни
  - преклапање извршавања инструкција
  - напредне архитектуре кеш меморија

Однос *RISC* и *CISC* процесора

	CISC		RISC
Characteristic	VAX 11/780	Intel 486	MIPS R4000
Number of instructions	303	235	94
Addressing modes	22	11	1
Instructions size (bytes)	2–57	1–12	4
Number of general-purpose registers	16	8	32

Пример кода

- У случају *VAX*-а, једна инструкција је могла да
  - прочита податак из меморије, сабере га са вредношћу регистра, упише назад у меморију и повећа вредност показивача:

    (R2) = (R2) + R3; R2 = R2 + 1
- У случају *RISC* процесора, за то су потребне 4 инструкције:

  R4 = (R2)

  R4 = R4 + R3

  (R2) = R4

  R2 = R2 + 1

Векторски процесори

- Векторски процесори су процесори који оперишу на низовима података
  - инструкције су пројектоване тако да раде са низовима података
  - може се рећи да су низови података елементарни облик података векторских процесора
  - представљају контраст тзв. скаларним процесорима, који раде са појединачним подацима

Број адреса у инструкцијама

- Бинарне операције захтевају два, а унарне један аргумент
- Операције најчешће имају један излаз, али их може бити и више
  - на пример, дељење даје количник и остатак
- Уобичајена бинарна операција захтева три адресе:
  - две адресе аргумената
  - једну адресу резултата

Број адреса у инструкцијама

- Постоје процесори:
  - са 3 адресе
  - са 2 адресе
  - са 1 адресом
  - без адреса

- Процесор који подржава неки број адреса, обично може да подржи и инструкције са мањим бројем адреса

Процесори са 3 адресе

- “Троадресни” процесори експлицитно адресирају два аргумента и резултат операције
- Већина савремених процесора је троадресна

Примери троадресних инструкција

add dest, src1, src2	Сабирају се вредности адресиране са *src1* и *src2* и резултат се уписује у *dest*
sub dest, src1, src2	Одузимају се вредности адресиране са *src1* и *src2* и резултат се уписује у *dest*
mult dest, src1, src2	Множе се вредности адресиране са *src1* и *src2* и резултат се уписује у *dest*

Пример кода

- На троадресном процесору израз:

  A = B + C * D – E + F + A

- може да се имплементира као:

```
mult    T,C,D     ; T = C*D
add  T,T,B        ; T = B + C*D
sub  T,T,E        ; T = B + C*D – E
add  T,T,F        ; T = B + C*D – E + F
add  A,T,A        ; A = B + C*D – E + F + A
```

Карактеристике

- Из примера се виде неке карактеристике:
  - у пракси већина инструкција садржи поновљену једну од адреса аргумената као адресу резултата
  - скуп инструкција одговара операцијама које процесор може да извршава

Процесори са 2 адресе

- “Двоадресни” процесори експлицитно адресирају један аргумент и резултат операције
- Мотивација потиче из чињенице да се релативно ретко употребљавају три различите адресе

- Процесори фамилије *Intel x86* су двоадресни

Примери двоадресних инструкција

load dest, src	Садржај податка адресираног са *src* се преписује у *dest*
add dest, src	Сабирају се вредности адресиране са *dest* и *src* и резултат се уписује у *dest*
sub dest, src	Одузима се вредност адресирана са *src* од вредности адресиране са *dest* и резултат се уписује у *dest*
mult dest, src	Множе се вредности адресиране са *src* и *dest* и резултат се уписује у *dest*

Пример кода

- На двоадресном процесору израз:

  A = B + C * D – E + F + A

- може да се имплементира као:

```
load  T,C ; T = C
mult  T,D ; T = C*D
add   T,B ; T = B + C*D
sub   T,E ; T = B + C*D – E
add   T,F           ; T = B + C*D – E + F
add   A,T ; A = B + C*D – E + F + A
```

Карактеристике

- Из примера се виде неке карактеристике:
  - у пракси већина инструкција понавља једну исту циљну адресу
  - скуп инструкција одговара операцијама које процесор може да извршава
  - додаје се инструкција за преписивање податка

Процесори са 1 адресом

- “Једноадресни” процесори експлицитно адресирају један аргумент
- Резултат се увек уписује у *акумулатор*
  - акумулатор је (углавном) једини регистар на коме могу да се извршавају операције
- Мотивација потиче из чињенице да се за већину операција употребљава понављање адресе циља

Примери једноадресних инструкција

load addr	Садржај податка адресираног са *addr* се преписује у акумулатор
add addr	Сабирају се вредност акумулатора и вредност адресирана са *addr* и резултат се уписује у акумулатор
sub addr	Одузима се од вредности акумулатора вредност адресирана са *addr* и резултат се уписује у акумулатор
mult addr	Множе се вредност акумулатора и вредност адресирана са *addr* и резултат се уписује у акумулатор
store addr	Вредност акумулатора се преписује на адресу *addr*

Пример кода

- На једноадресном процесору израз:

  A = B + C * D – E + F + A

- може да се имплементира као:

```
load    C       ; acc = C
mult    D       ; acc = C*D
add     B       ; acc = B + C*D
sub     E       ; acc = B + C*D – E
add     F       ; acc = B + C*D – E + F
add     A       ; acc = B + C*D – E + F + A
store   A       ; A = B + C*D – E + F + A
```

Карактеристике

- Из примера се виде неке карактеристике:
  - редукован број регистара
    - само један има пуну оперативну функционалност
  - скуп инструкција одговара операцијама које процесор може да извршава
  - додају се инструкције за преписивање података у акумулатор и из акумулатора

Процесори са 0 адреса

- “Безадресни” процесори не адресирају ниједан аргумент експлицитно
  - осим у посебним инструкцијама које стављају податке на стек и узимају податке са стека
- И аргументи и резултат се увек налазе на стеку
- Мотивација потиче из чињенице да је за већину операција потребно релативно мало података

Примери безадресних инструкција

push addr	Садржај податка адресираног са *addr* се ставља на врх стека
pop addr	Податак са врха стека се склања са стека и уписује на адесу *addr*
add	Два податка са врха стека се склањају са стека и сабирају. Резултат се ставља на врх стека.
sub	Два податка са врха стека се склањају са стека и одузимају. Резултат се ставља на врх стека.
mult	Два податка са врха стека се склањају са стека и множе. Резултат се ставља на врх стека.

Пример кода

- На безадресном процесору израз:

  A = B + C * D – E + F + A

- може да се имплементира као:

```
push    E       ; <E>
push    C       ; <C, E>
push    D       ; <D, C, E>
mult            ; <D*C, E>
push    B       ; <B, D*C, E>
add             ; <B + D*C, E>
sub             ; <B + D*C – E>
push    F       ; <F, B + D*C – E>
add             ; <B + D*C – E + F>
push    A       ; <A, B + D*C – E + F>
add             ; <B + D*C – E + F + A>
pop     A       ; <>
```

Карактеристике

- Из примера се виде неке карактеристике:
  - не постоје именовани регистри
  - скуп инструкција одговара операцијама које процесор може да извршава
  - додају се инструкције за преписивање података на стек и са стека

Имплементација

- Обично се имплементирају тако да се неколико последњих података на стеку налази у тзв. *стек регистрима* процесора
  - број стек регистара се назива *дубина стека*
  - на тај начин се значајно убрзавају операције са стеком, зато што није потребно приступати меморији

Поређење начина адресирања

- Сваки од представљених приступа има предности и мане
- Што се више адреса наводи у инструкцијама
  - број приступа меморији је већи
  - запис инструкција је већи
  - програми се састоје од мање инструкција

Пример процене ефикасности

- Троадресни рачунар:
  - свака инструкција захтева 4 приступа меморији
    - један за инструкцију, два за податке, један за резултат
  - пет инструкција
    - укупно 20 приступа меморији
- Двоадресни рачунар:
  - свака операција и даље захтева 4 приступа меморији
    - један за инструкцију, два за податке, један за резултат
    - инструкција *load* захтева три приступа
  - пет операција и једно преписивање
    - укупно 23 приступа меморији

Пример процене ефикасности

- Једноадресни рачунар:
  - свака операција захтева 2 приступа меморији
    - један за инструкцију и један за податке
    - акумулатор је регистар, а не меморија
    - инструкција *load* захтева два приступа
  - седам инструкција
    - укупно 14 приступа меморији
- Безадресни рачунар:
  - свака операција захтева 1 приступ меморији
    - један за инструкцију
    - претпостављамо да је стек довољно дубок да је пример стао у стек регистре
    - инструкције *push* и *pop* захтевају по два приступа
  - пет операција по 1 и седам инструкција *push* и *pop*
    - укупно 19 приступа меморији

Пример процене ефикасности

- Пример сугерише да је у претходном примеру једноадресни приступ најефикаснији, међутим:
  - поређење једноадресног и безадресног рачунара је фер, зато што се у оба случаја претпоставља постојање регистара
  - са друге стране, и неки од адресираних података у случају дво- и троадресних рачунара могу бити регистри

- Ако претпоставимо да дво- и троадресни рачунар имају на располагању један регистар *T*, онда се однос мења:
  - двоадресни има 13 приступа меморији
  - троадресни има свега 12 приступа меморији

Пример процене ефикасности

- У претходним примерима није узета у обзир величина инструкција:
  - што се више адреса наводи, то је инструкција већа
  - величина није увек једноставно предвидива

Ограничавање врста адреса

- Постоји већи број различитих начина адресирања података
- Код *RISC* процесора је уобичајено да се у већини инструкција могу адресирати само различити регистри процесора
- Процесор *Intel Pentium* је двоадресни, али највише један од операнада сме бити у меморији

Архитектура *load / store*

- Концепт:
  - све операције се извршавају искључиво над регистрима процесора
  - само операције *load* и *store* могу да приступају меморији

Архитектура *load / store* (2)

- *RISC* и векторски процесори често користе овакву архитектуру
  - значајно се смањује величина инструкција
  - значајно се редукује сложеност декорирања и имплементирања инструкција
  - омогућава се висок степен преклапања инструкција
    - дужина извршавања није непосредно пропорционална броју инструкција и приступа меморији

Пример кода

- На троадресном проц. са арх. *load / store* израз:

  A = B + C * D – E + F + A

- може да се имплементира као:

```
load    R1, B              ; R1 = B
load    R2, C              ; R2 = C
load    R3, D              ; R3 = D
load    R4, E              ; R4 = E
load    R5, F              ; R5 = F
load    R6, A              ; R6 = A
mult    R2, R2, R3 ; R2 = C*D
add     R2, R2, R1 ; R2 = B + C*D
sub     R2, R2, R4 ; R2 = B + C*D – E
add     R2, R2, R5 ; R2 = B + C*D – E + F
add     R2, R2, R6 ; R2 = B + C*D – E + F + A
store   A, R6
```

Архитектура регистара

- Регистри процесора служе за чување
  - података
  - инструкција
  - стања процесора

- Регистри се деле на
  - регистре опште намене и
  - посебне регистре (или регистре посебне намене)
    - посебни регистри доступни корисничким програмима
    - посебни регистри резервисани за системске потребе

Архитектура регистара опште намене

- Број и врста регистара опште намене су обично повезани са архитектуром адресирања
  - безадресни процесори не захтевају регистре опште намене
    - мада имају имплицитне стек-регистре
  - код дво- и троадресних процесора регистри опште намене нису неопходни
    - уводе се због подизања перформанси
  - *RISC* процесори по правилу имају већи број регистара опште намене

Архитектура регистара посебне намене

- Пример регистара посебне намене су:
  - регистри за вођење стека
  - бројач инструкција
  - интерни регистар инструкције (који садржи текућу инструкцију)

Контрола тока програма

- Бројач инструкција (или *програмски бројач*) има улогу контролора тока
  - садржи адресу наредне инструкције
  - чим се инструкција прочита, бројач се повећава тако да показује на наредну инструкцију
  - код архитектура са фиксном величином инструкције, увек се увећава за фиксан број
    - нпр. код процесора *MIPS* и *SPARC*, све инструкције су 32-бита

- Програм се у начелу извршава секвенцијално
- Секвенцијално извршавање се по потреби може изменити
  - гранањем и
  - петљама

Гранање

- Гранање се имплементира инструкцијама гранања
  - Оне експлицитно мењају вредност бројача инструкција
- Постоје две врсте инструкција гранања:
  - безусловне (или *експлицитне*) и
  - условне
- Гранање може бити
  - тренутно или
  - одложено

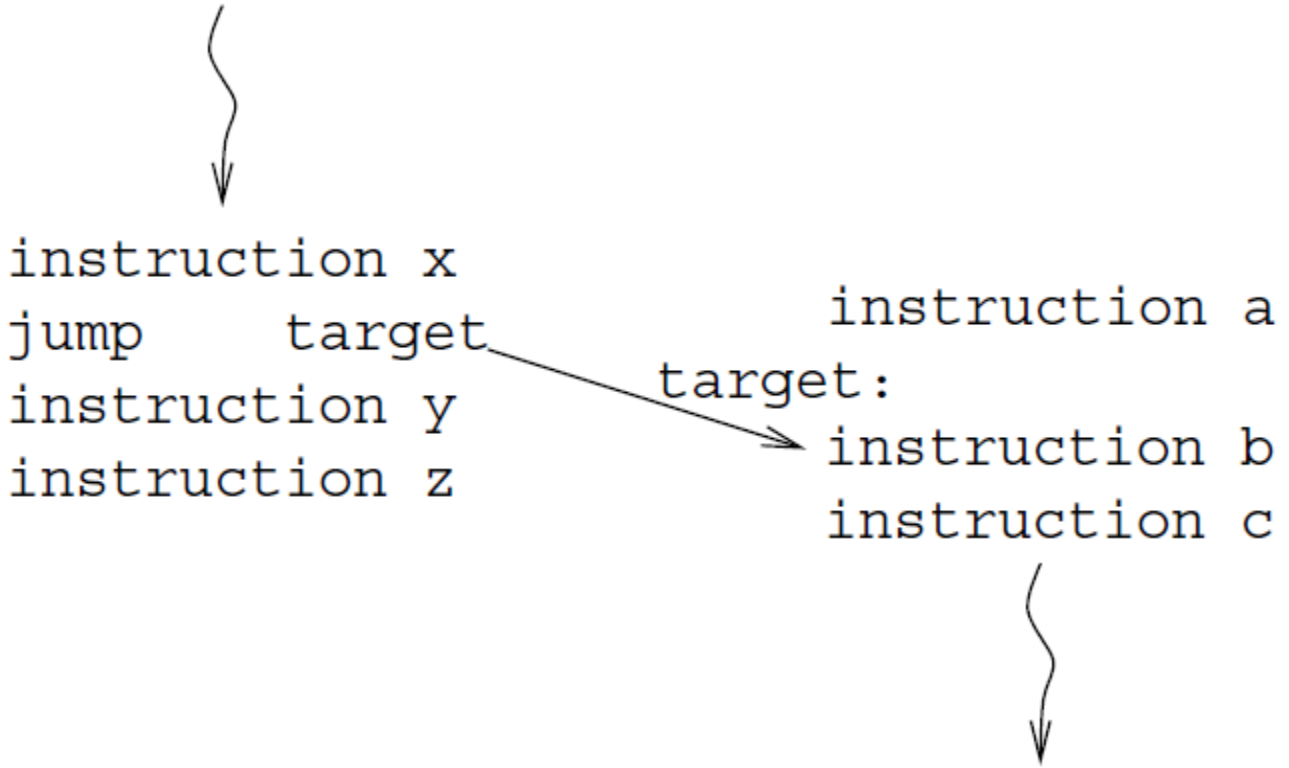
Начин навођења нове адресе

- Нова адреса извршавања се наводи као
  - апсолутна
    - наводи се пуна нова адреса
    - подржавају практично сви процесори
  - релативна
    - наводи се разлика између нове адресе и текуће адресе
    - не подржавају сви процесори
    - предност је у померљивости кода
      - без обзира на локацију у меморији гранање је једнако исправно
    - ако разлика није велика, може имати краћи запис него навођење апсолутне адресе

Безусловно гранање

- Безусловно гранање је експлицитна и безусловна промена тока извршавања програма

Безусловно гранање - пример



Условно гранање

- Условно гранање је експлицитна промена тока извршавања програма у случају важења неког наведеног услова
- Постоје две основне врсте условног гранања:
  - постави-па-скочи и
  - провери-и-скочи

Гранање постави-па-скочи

- Основна идеја је да се раздвоје проверавање услова и гранање
  - најпре се посебним инструкцијама проверају услови или другачије поставља одговарајуће стање процесора
  - затим се инструкцијама гранања само проверава стање процесора и по потреби извршава промена бројача инструкција

- Гранање постави-па-скочи је примењено код фамилије процесора *Intel x86*

Пример – постави-па-скочи

- Пример у случају процесора *Intel Pentium*

```
    cmp AX,BX       ; поређење вредности AX и BX
    je target       ; ако су једнаке, контрола се преноси на target
    sub AX,BX       ; иначе се наставља од ове инструкције
    . . .
target:
    add AX,BX       ; у случају једнакости се наставља одавде
```

Гранање провери-и-скочи

- Основна идеја је да једна иста инструкција проверава услов и промени адресу
- Овакав вид гранања примењен је код већег броја процесора, укључујући и *MIPS*

Пример – провери-и-скочи

- Пример у случају процесора *MIPS*
  - наредна инструкција пореди вредности регистара *$t0* и *$t1* и прелази на дату адресу *target* ако су вредности једнаке

```
beq $t1,$t0,target
```

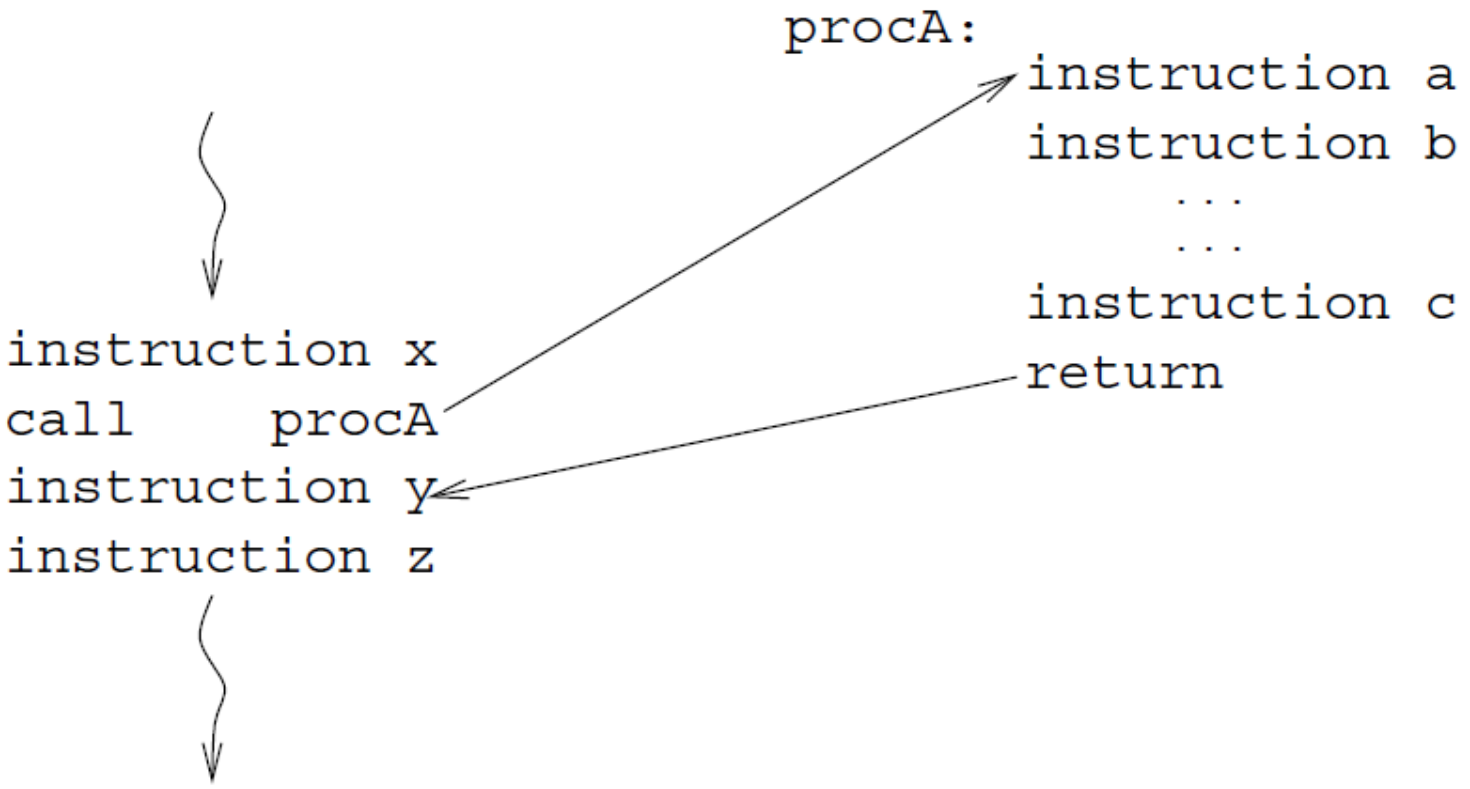
Регистри стања

- Да би било могуће имплементирати гранање постави-па-скочи, неопходно је да процесор има *регистар стања* који чува резултате поређења и других операција
  - овакви регистри се називају и *регистри кодова поређења*
- Овакве регистре
  - имају *Intel x86*, *SPARC*, *PowerPC*
  - нема *MIPS*

Позивање процедура

- Гранања су једносмерне промене тока извршавања
  - дејство је ограничено на једну промену тока извршавања
- Позивања процедура су двосмерне промене
  - након иницијалног позивања,
    касније следи повратак на место одакле је извршено позивање
- Да би повратак био могућ неопходне су две ствари:
  - експлицитна ознака краја процедуре
    - за то служи инструкција *return*
      - *Intel Pentium* има инструкцију *ret*
      - *MIPS* има инструкцију *jr*
  - адреса повратка
    - мора бити сачувана при позивању процедуре

Пример позивања процедуре



Чување адресе повратка

- Може се чувати
  - у регистру
    - намењеном за то
      - шта је са рекурзијом или другим позивањима?
    - било ком регистру
      - на пример *MIPS*
  - на стеку
    - нпр. *Intel x86*

- Чува се
  - адреса инструкције позивања
    - *SPARC*
  - адреса прве инструкције након инструкције позивања
    - већина процесора, укључујући *Intel x86*, *MIPS*

Чување адресе повратка у регистрима

- Може се користити
  - регистар посебно намењен за то
  - било који регистар
    - на пример *MIPS*
    - пример повратка, уз претпоставку да је адреса повратка у регистру $*ra*

```
jr  $ra
```

- Проблеми
  - шта је са рекурзијом?
  - шта је са позивањем потпроцедура?
  - у оба случаја је неопходно додатно старање о адресама повратка

Преношење параметара

- Две основне технике су:
  - помоћу регистара
    - параметри се записују у регистрима пре позивања
    - овај метод је бржи
    - не омогућава рекурзију
    - често захтева чување претходних вредности регистара
    - коришћен код *RISC* процесора
      - нпр. *MIPS*, *PowerPC*
  - помоћу стека
    - параметри се постављају на стек пре позивања
    - флексибилнији начин
    - мање ефикасан
    - коришћен код *CISC* процесора
      - нпр. *Intel x86*

Пројектовање скупа инструкција

- Пројектовање скупа инструкција има неколико важних аспеката:
  - типови операнада
  - начини адресирања
  - типови инструкција
  - формати инструкција

Типови операнада

- Уобичајено је да процесори препознају само елементарне типове података
  - *char, int, float*
- Често исте инструкције раде са подацима различите величине, у зависности од начина навођења операнада
  - на пример (*Intel x86*):

```
mov AL, addr    ; преписује 8-битни податак
mov AX, addr    ; преписује 16-битни податак
mov EAX, addr   ; преписује 32-битни податак
```

Типови операнада

- У случају *RISC* процесора уобичајено је да се из нотације инструкције препознаје величина операнда, на пример:

```
lb    Rdest, address    ; преписује 8-битни податак (бајт)
lh    Rdest, address    ; преписује 16-битни податак (пола речи)
lw    Rdest, address    ; преписује 32-битни податак (реч)
ld    Rdest, address    ; преписује 64-битни под. (двострука реч)
```

Начини адресирања

- Начини адресирања описују како се одређује операнд инструкције
- Операнди могу бити
  - константе
    - режим непосредног адресирања
  - у регистрима
    - режим регистарског адресирања
  - у меморији
    - режим меморијског адресирања
    - постоји много различитих начина адресирања података у меморији

Начини адресирања (2)

- Сви процесори подржавају бар два основна начина адресирања:
  - Режим непосредног адресирања
    - навођење константне вредности операнда
    - не постоји приступање меморији
      - осим читања инструкције
    - назива се и *непосредно адресирање*
  - Режим регистарског адресирања
    - навођење регистра који садржи вредност операнда
    - не постоји приступање меморији
      - осим читања инструкције
    - назива се и *регистарско адресирање*

Начини адресирања (3)

- Разлика између *RISC* и *CISC* процесора је у подржаним начинима адресирања података у меморији
  - *RISC* процесори користе архитектуру *load / store*
    - све инструкције, осим *load* и *store*, подржавају само непосредно и регистарско адресирање
    - подаци који су у меморији могу се адресирати само у оквиру инструкција *load* и *store*
    - број начина адресирања је обично сасвим скроман
  - *CISC* процесори подржавају мноштво начина адресирања
    - убичајено је да све инструкције подржавају адресирање података у меморији
    - број начина адресирања је обично велики

Врсте инструкција

- Инструкције за премештање података
- Аритметичке и логичке инструкције
- Инструкције за контролу тока
- Улазно / излазне инструкције

Инструкције за премештање података

- Подржавају их сви процесори
- Деле се на инструкције које премештају податке:
  - између меморије и регистара
    - посебна подврста за рад са стеком
  - између регистара

Инструкције за премештање података (2)

- Код *RISC* процесора је премештање података између процесора и меморије строго ограничено на инструкције:
  - *load*
  - *store*
  - неки од *RISC* процесора не омогућавају непосредно премештање података између регистара, већ само у оквиру других инструкција (нпр. сабирање), на пример:
    - *add Rdest, Rsource, 0 /* Rdest = Rsource + 0 */*

Инструкције за премештање података (3)

- Код *CISC* процесора се уместо две обично имплементира само једна инструкција за премештање података која равноправно третира регистре и меморију
  - на пример, код *Intel x86*:
    - ***mov dest, src***
    - највише један од аргумената може бити у меморији
    - MOV CX,20
    - MOV CX, [BX]
    - MOV CX, [50000]

Аритметичке инструкције

- Аритметичке инструкције обухватају како целобројне тако и операције у покретном зарезу

- Већина процесора подржава бар 4 основне аритметичке операције
  - сабирање и одузимање захтевају по једну инструкцију
  - множење и дељење захтевају посебне операције за означене и неозначене аргументе
  - неки процесори не подржавају дељење у потпуности
    - потпуна подршка је рачунање количника и остатка
    - *Intel x86, MIPS* пружају пуну подршку
    - *PowerPC, Sparc* рачунају само количник

Логичке инструкције

- Логичке операције подразумевају скуп операција на нивоу битова
  - практично сви процесори подржавају *and* и *or*
  - већина процесора подржава *not* и *xor*

Контролни битови

- Скоро све аритметичке и логичке инструкције постављају контролне битове процесора при свом извршавању
  - називају се и *заставице* или *условни кодови*

- Уобичајени контролни битови су:
  - *S* – бит знака (0=позитиван, 1=негативан)
  - *Z* – бит нуле (0=резултат није нула, 1=резултат је нула)
  - *O* – бит прекорачења (0=нема пр., 1=прекорачење)
  - *C* – бит преноса (0=нема преноса, 1=има преноса)

Контролни битови (2)

- Контролни битови се употребљавају
  - као улазни подаци за неке операције
    (нпр. сабирање са преносом, померање са преносом и сл.)
  - у инструкцијама гранања
- пример за *Intel x86*:

```
cmp count, 25
je target
```

Инструкције за контролу тока

- Инструкције за контролу тока програма су
  - инструкције гранања
  - инструкције за позивање процедура
    - овде спадају и инструкције за враћање из процедура

- Размотрено раније...

Улазно / излазне инструкције

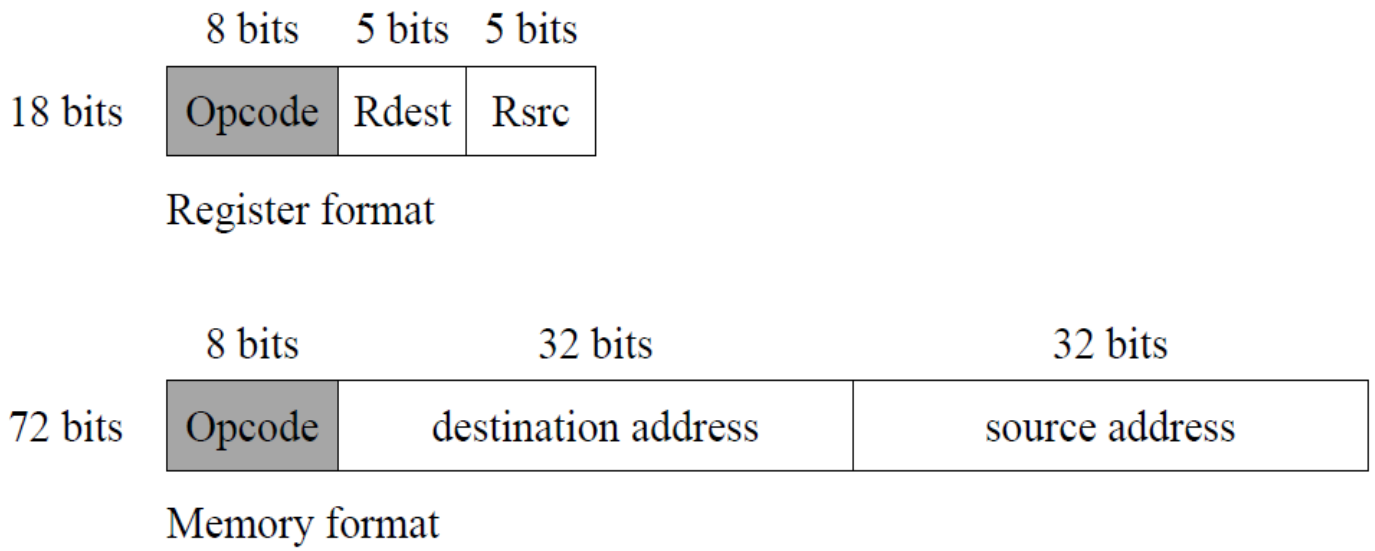
- Улазно / излазне инструкције се значајно разликују између процесора
- Оне постоје само код процесора који подржавају изоловано пресликавање улаза и излаза
  - ако процесор подржава само меморијско пресликавање У/И, онда нема ове инструкције
- Уобичајене су две инструкције:
  - ***in Reg, io_port***
  - ***out io_port, Reg***
- Величина инструкције зависи од подржане дужине адресе порта

Формат инструкција

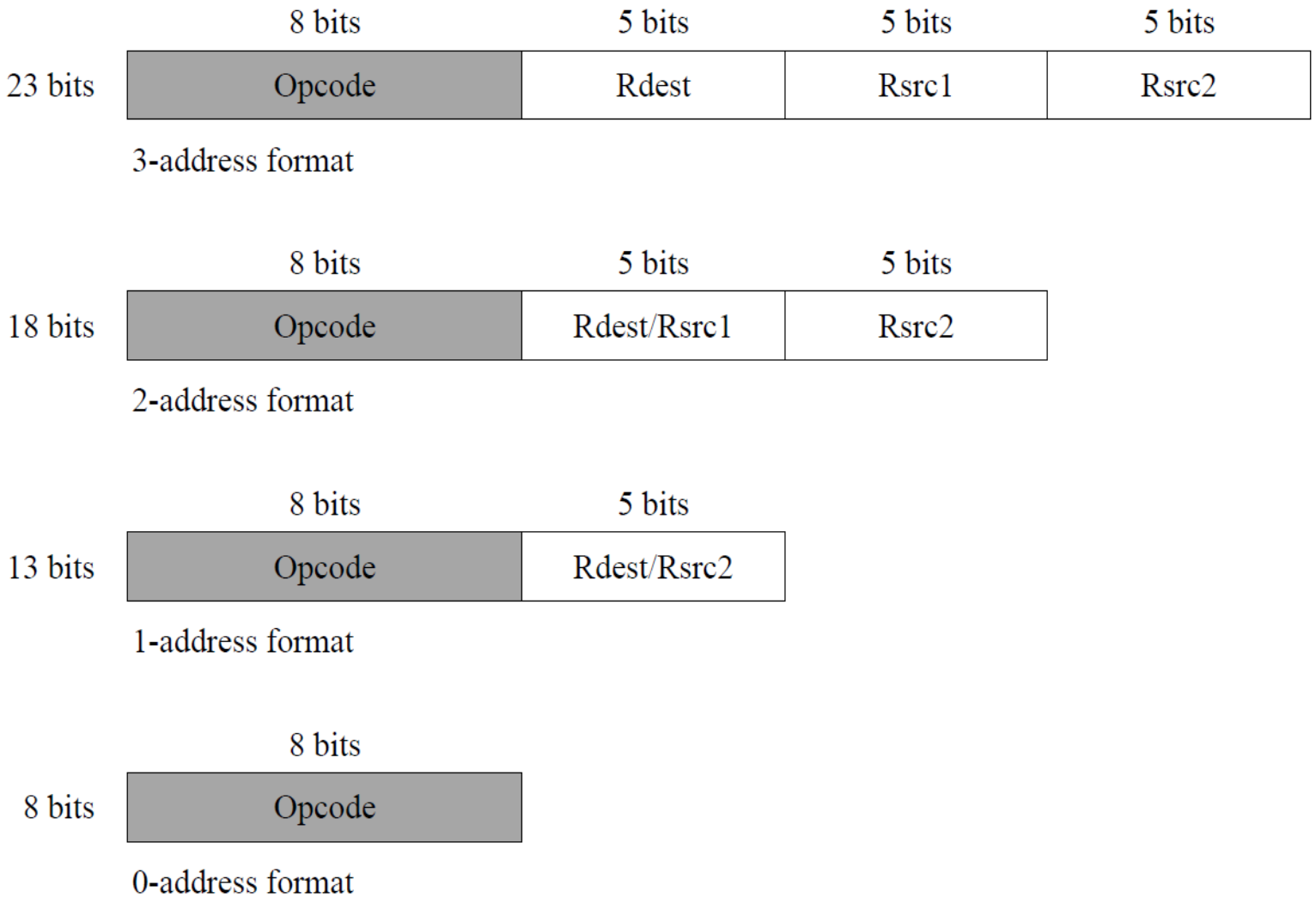
- Формат инструкције подразумева начин кодирања (тј. бинарног записивања) инструкције

- Два основна типа формата инструкција су:
  - формат фиксне дужине инструкција
    - уобичајен за *RISC* процесоре
    - *MIPS, PowerPC, Sparc* имају исинтрукције дужине 32 бита
  - формат променљиве дужине инструкција
    - уобичајен за *CISC* процесоре
    - нпр. *Intel x86*

Врста аргумената и формат инструкција



Фиксан формат инструкција



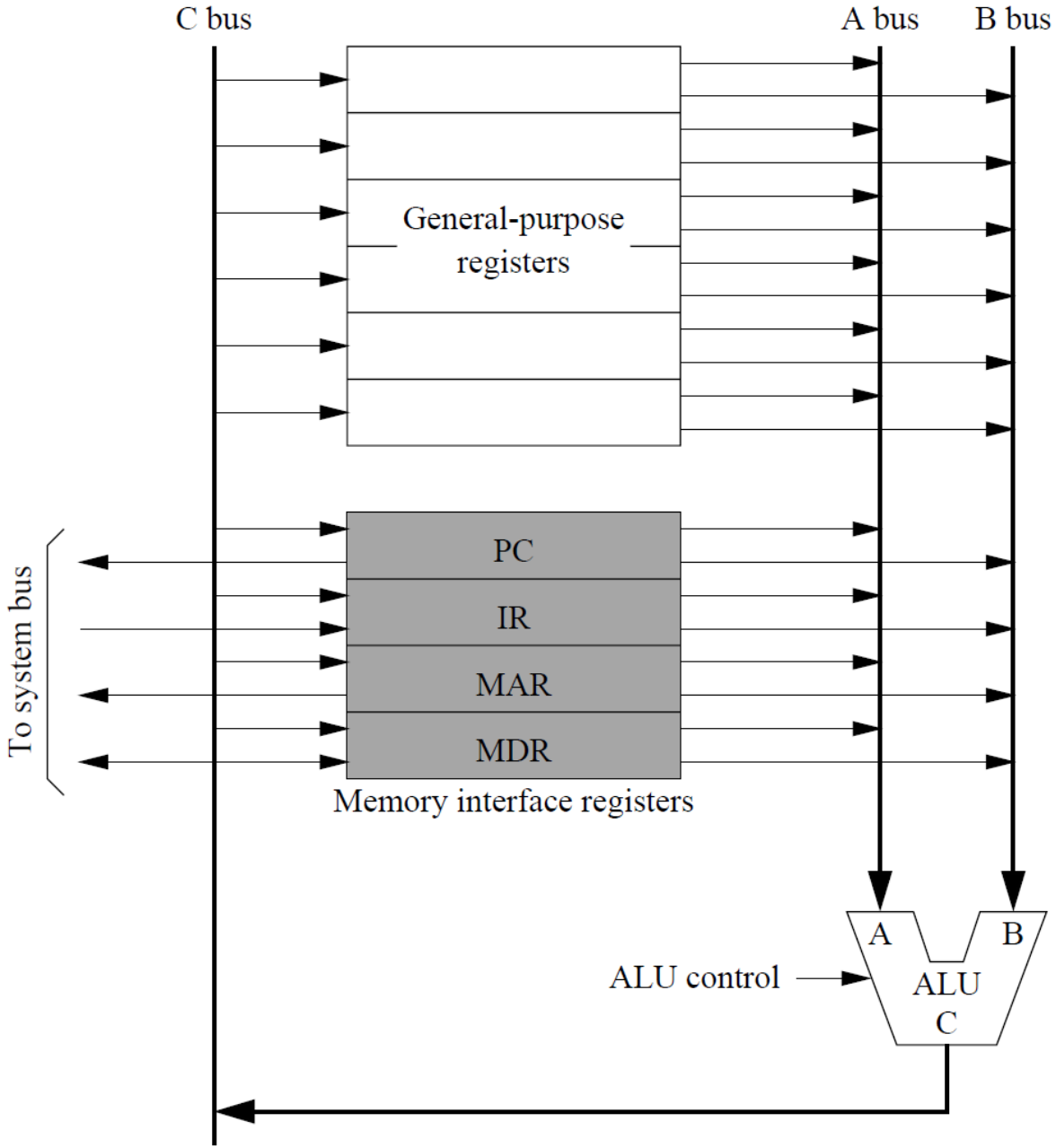
Процесор

Имплементација инструкција

Извршавање инструкција

- Да би процесор могао да изврши инструкцију потребно је да:
  - адресира и прочита инструкцију из меморије
  - декодира инструкцију
  - адресира и прочита потребне аргументе
  - изврши одговарајућу операцију
  - адресира и запише израчунат резултат

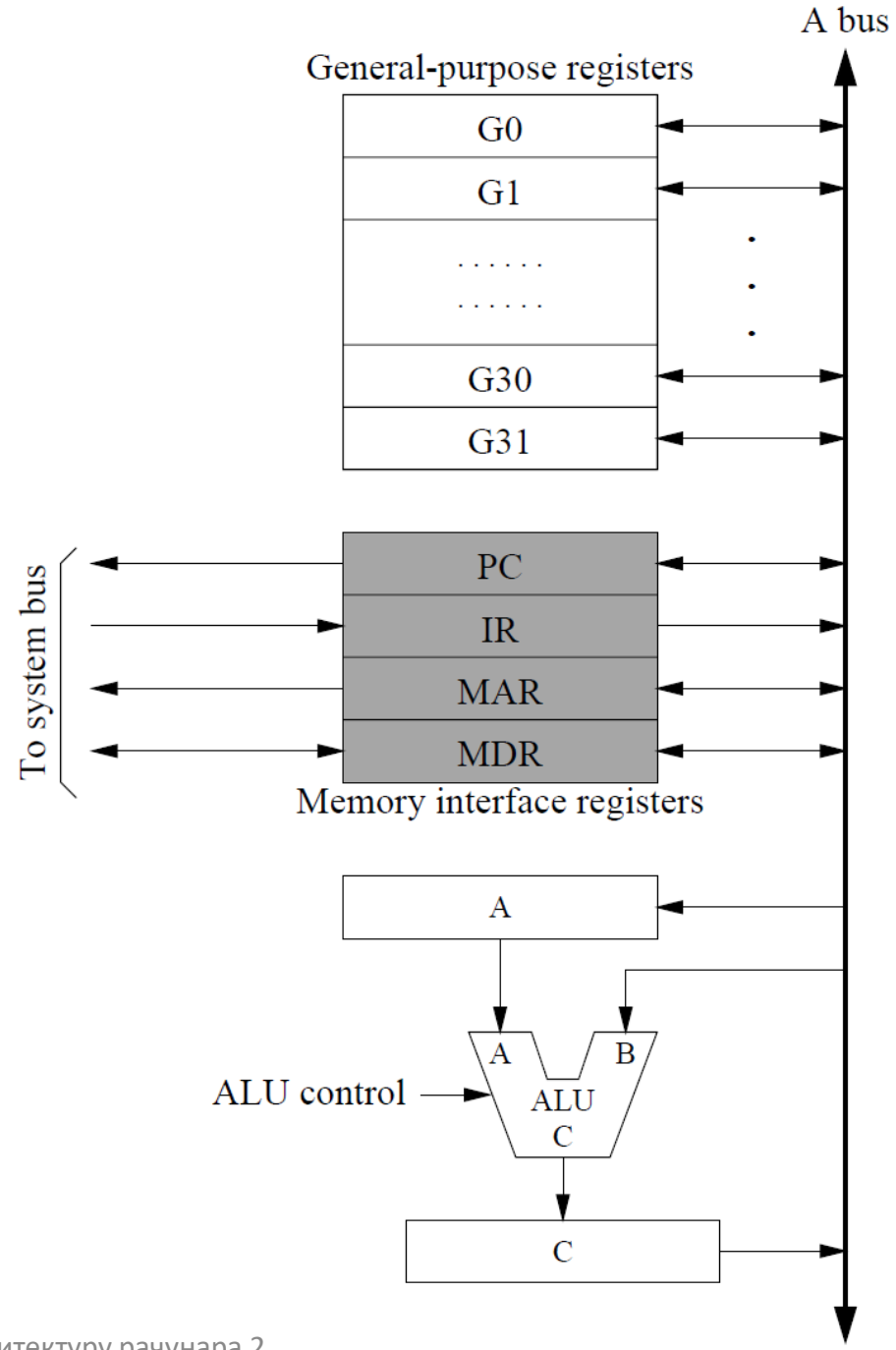
Уопштени пут података



Конкретнији пример

- Претпоставимо да процесор има:
  - јединствену магистралу (A) ширине 32 бита кроз коју пролазе све адресе и подаци у оквиру процесора
  - 32 регистра опште намене (G1-G32)
  - могућност обраде само 32-битних података

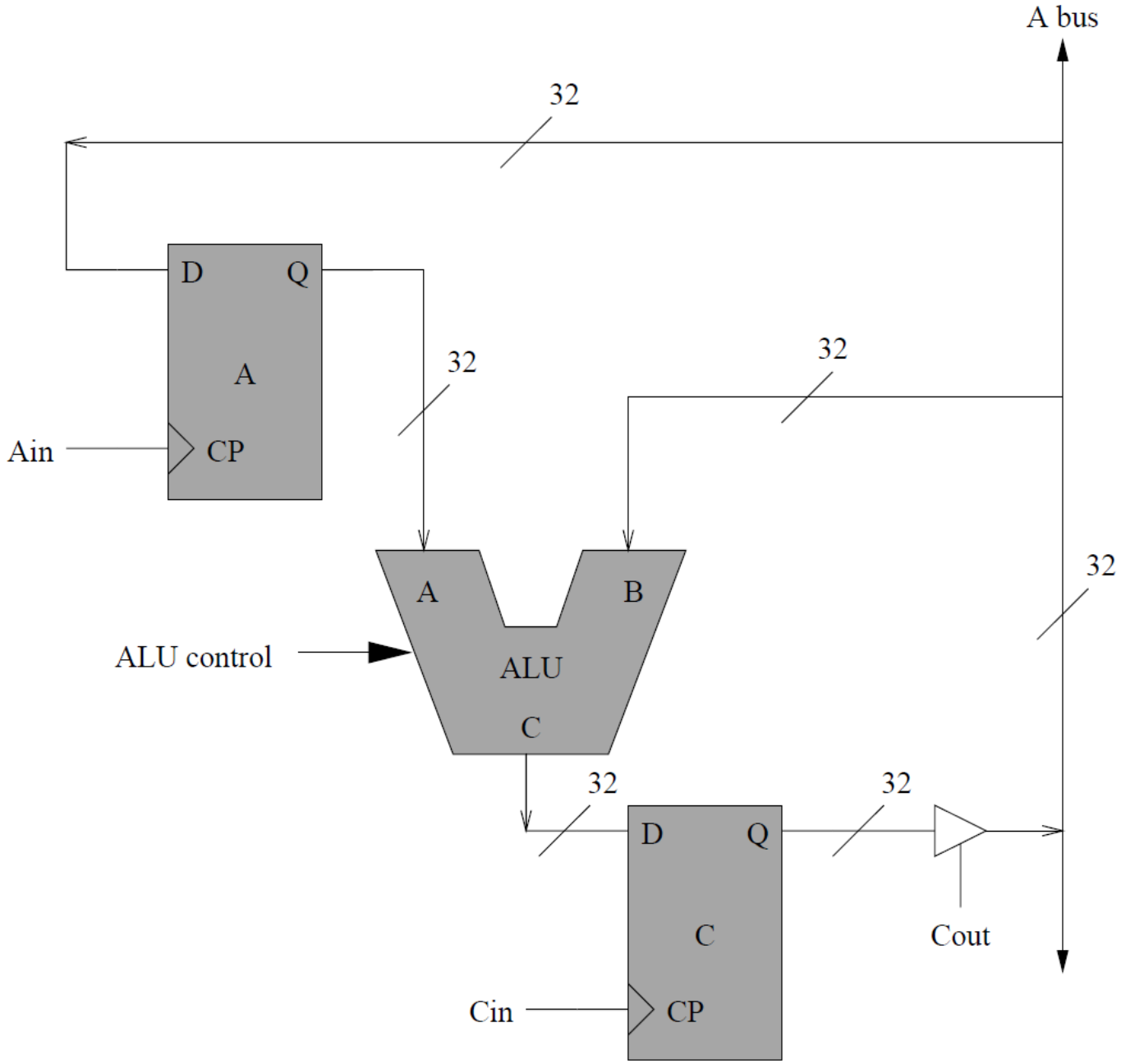
Конкретнији пример пута података



Додатни регистри

- Због тога што постоји јединствена интерна магистрала А, потребни су помоћни регистри:
  - регистар *A* чува вредност првог операнда док се други адресира
    - увек је доступан за читање
  - регистар *C* чува резултат док се не пренесе даље
    - увек је доступан претходни резултат за писање
- Имплементирају се помоћу *D* флип-флопова

Имплементација помоћних регистара

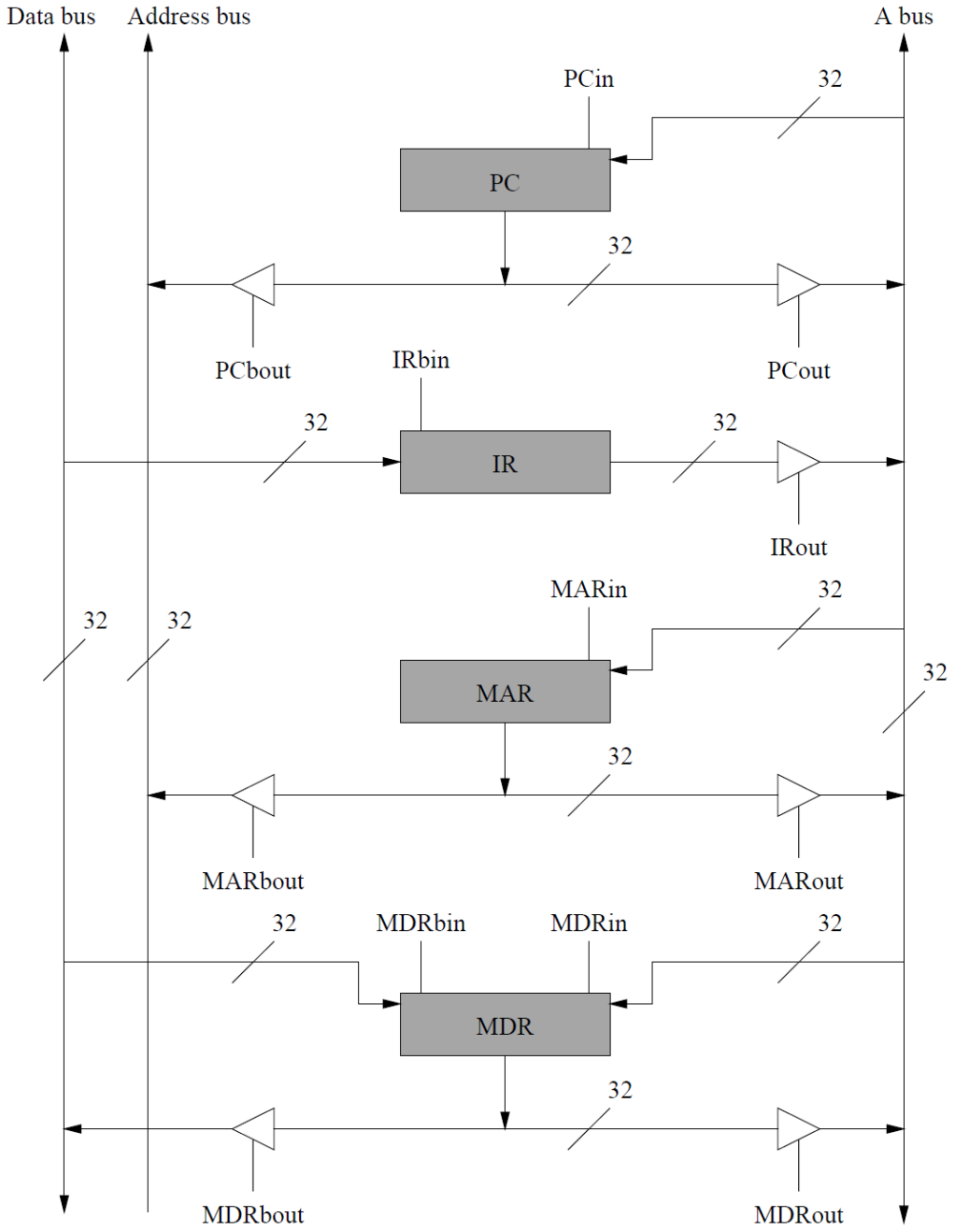


Меморијски интерфејс

- Меморијски интерфејс користи четири помоћна регистра
  - ови регистри посредују између системске магистрале и интерне процесорске магистрале A

- Регистар *PC* је бројач инструкција
- Регистар *IR* је регистар инструкције
- Регистар *MAR* је регистар меморијске адресе
- Регистар *MDR* је регистар меморијског податка

Имплементација пута података



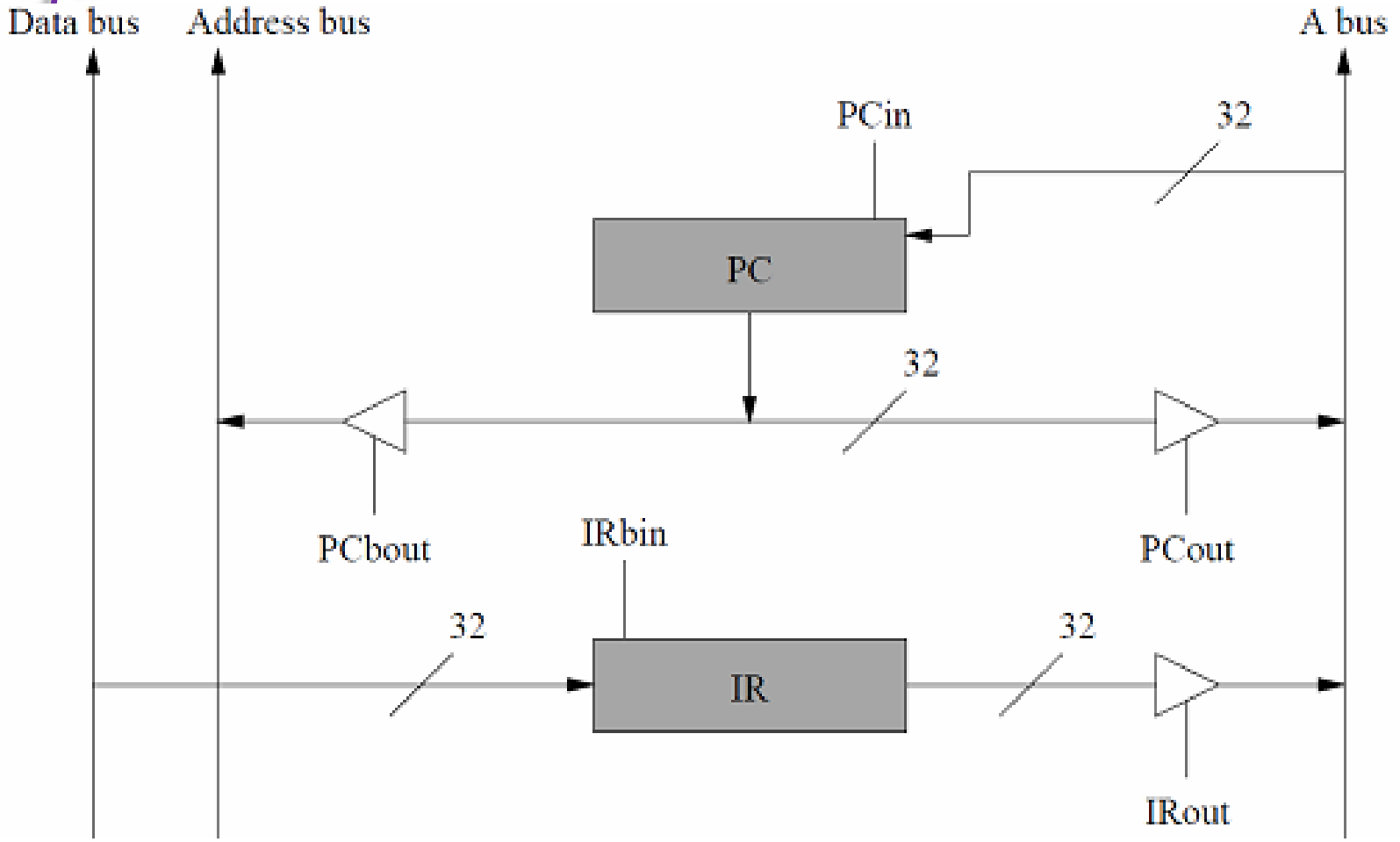
Регистар *PC*

- Бројач инструкција
  - садржи адресу наредне инструкције
    - контролни сигнал *PCin* поставља вредност регистра
  - садржај ставља на системску адресну магистралу ради читања инструкције из меморије
    - контролни сигнал *PCbout*
  - садржај ставља на магистралу A ради омогућавања релативних скокова и позивања процедура
    - контролни сигнал *PCout*
  - може симултано да иде на обе магистрале

Регистар *IR*

- Регистар инструкције
  - садржи инструкцију која се трнутно извршава
    - контролни сигнал *IRbin* поставља вредност регистра
    - контролни сигнал *IRout* ставља вредност регистра на магистралу A

Имплементација пута података



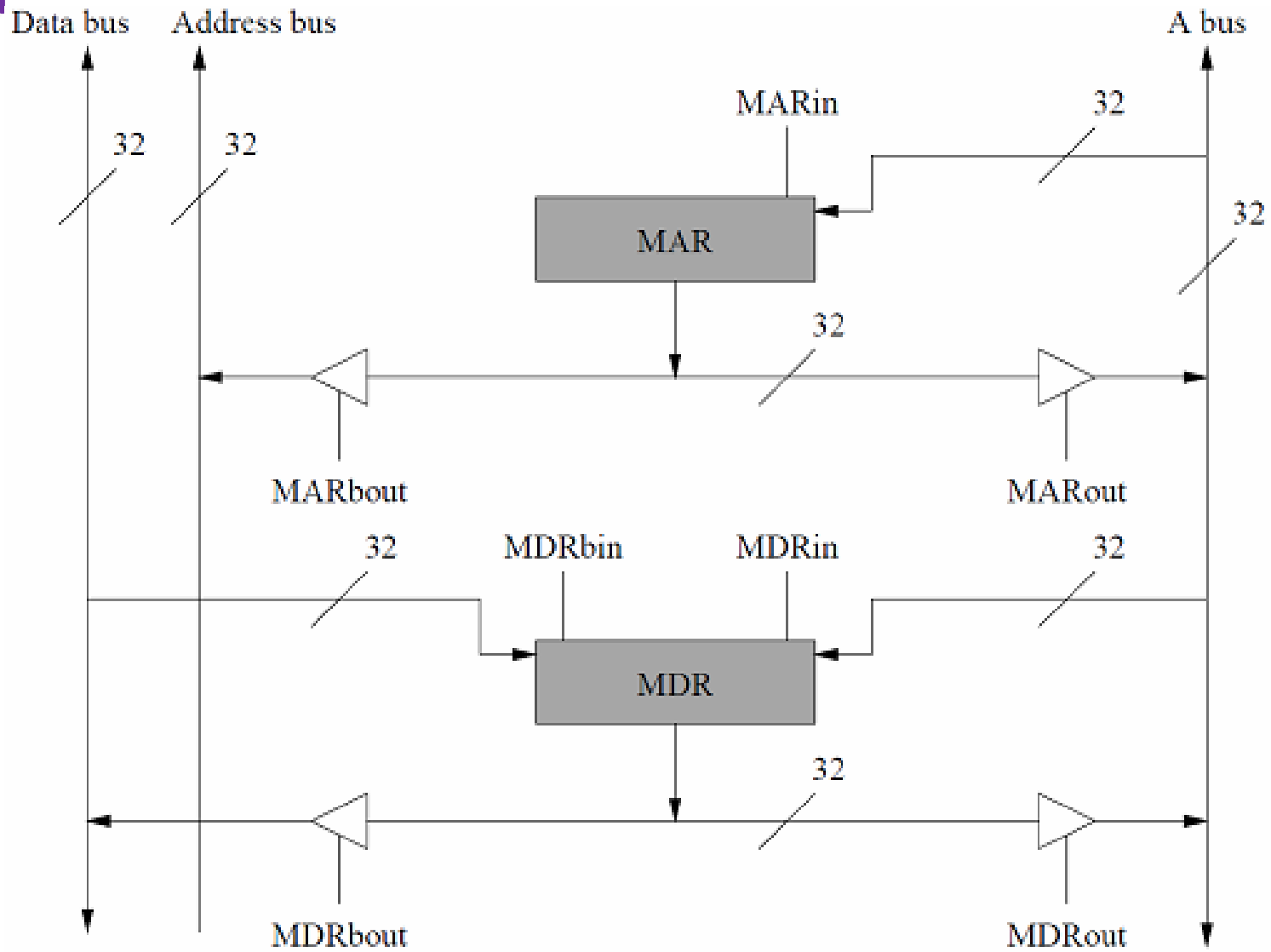
Регистар *MAR*

- Регистар меморијске адресе
  - садржи адресу операнда који је у меморији
  - користи се при адресирању података који су у меморији
  - ради слично регистру *PC:*
    - контролни сигнал *MARin* поставља вредност регистра
    - контролни сигнал *MARout* ставља вредност регистра на магистралу А
    - контролни сигнал *MARbout* ставља вредност регистра на системску адресну магистралу

Регистар *MDR*

- Регистар меморијског податка
  - садржи вредност операнда који је у меморији
  - користи се при читању операнда из меморије
  - адреса операнда је у регистру *MAR*
  - има двосмеран интерфејс*:*
    - контролни сигнал *MDRin* поставља вредност регистра са магистрале A
    - контролни сигнал *MDRbin* поставља вредност регистра са системске магистрале података
    - контролни сигнал *MDRout* ставља вредност регистра на магистралу A
    - контролни сигнал *MDRbout* ставља вредност регистра на системску магистралу података

Имплементација пута података



Регистри опште намене

- Регистри опште намене су везани само на инстерну магистралу A
- Сваки од регистара *Gx* има по два контролна сигнала
  - *Gxin*
  - *Gxout*

Пример инструкције

- Пратимо контролне сигнале на примеру инструкције:

  add %G9, %G5, %G7 / G9 = G5 + G7 */*

- Најпре се садржај једног регистра мора сачувати у помоћном регистру A, а затим сабирати са другим...
  - корак 1: преписујемо *G5* у помоћни регистар *A*
  - корак 2: стављамо *G7* на магистралу *A* и рачунамо
  - корак 3: резултат уписујемо у регистар *G9*

- Нећемо експлицитно наглашавати искључивање сигнала између корака

Корак 1:

- Преписујемо *G5* у помоћни регистар *A*
  - поставља се сигнал *G5out* да би се садржај регистра *G5* ставио на магистралу A
  - поставља се сигнал *Ain* да би се подаци са магистрале A уписали у помоћни регистар *A*

Корак 2:

- Стављамо *G7* на магистралу *A* и рачунамо
  - поставља се сигнал *G7out* да би се садржај регистра *G7* ставио на магистралу A
  - (садржај регистра A је увек доступан АЛ јединици)
  - поставља се сигнал *Cin* да би се резултат уписао у регистар *C*
  - АЛ јединици се налаже да израчуна резултат

Корак 3:

- Резултат уписујемо у регистар *G9*
  - поставља се сигнал *G9in* да би се садржај регистра *G9* прочитао са магистрале A
  - поставља се сигнал *Cout* да би се садржај регистра *C* ставио на магистралу A

Трајање циклуса

- На основу дужине трајања корака се дефинише трајање циклуса
  - У идеалном случају би сваки корак требало да траје једнако дуго
  - По потреби се може доделити већи број циклуса захтевнијим корацима
    - у претходном примеру су кораци 1 и 3 слични, али корак 2 може бити захтевнији (зависно од операције)

Читање инструкције

- У претходном опису је намерно прескочено представљање читања и декодирања инструкције

- Читање инструкције обухвата:
  - корак 1: адресирање инструкције
  - корак 2: уписивање нове адресе инструкције
  - корак 3: читање инструкције

Корак 1:

- Адресирање инструкције
  - поставља се сигнал *PCbout* да би се садржај регистра *PC* ставио на системску адресну магистралу
  - поставља се сигнал *PCout* да би се садржај регистра *PC* ставио на магистралу A
  - АЛ јединици се даје инструкција *add4* да би сабрала вредност на свом улазу *B* (вредност регистра *PC*) са 4
  - поставља се сигнал *Cin* да би се резултат (*PC+4*) уписао у помоћни регистар *C*

Корак 2:

- Уписивање нове адресе инструкције
  - чекамо (бар) један циклус да меморија прочита и достави инструкцију
  - поставља се сигнал *Cout* да би се резултат (*PC+4*) ставио на магистралу A
  - поставља се сигнал *PCin* да би се резултат (*PC+4*) са магистрале A уписао у регистар *PC*

Корак 3:

- Читање инструкције
  - поставља се сигнал *IRbin* да би се прочитана инструкција ставила у помоћни регистар *IR*

Декодирање инструкција

- Декодирање инструкције је заједничко за све врсте инструкција
- Следи непосредно после читања инструкције

- У фази декодирања се препознају
  - код инструкције
  - број и типови операнада
  - начини адресирања операнада

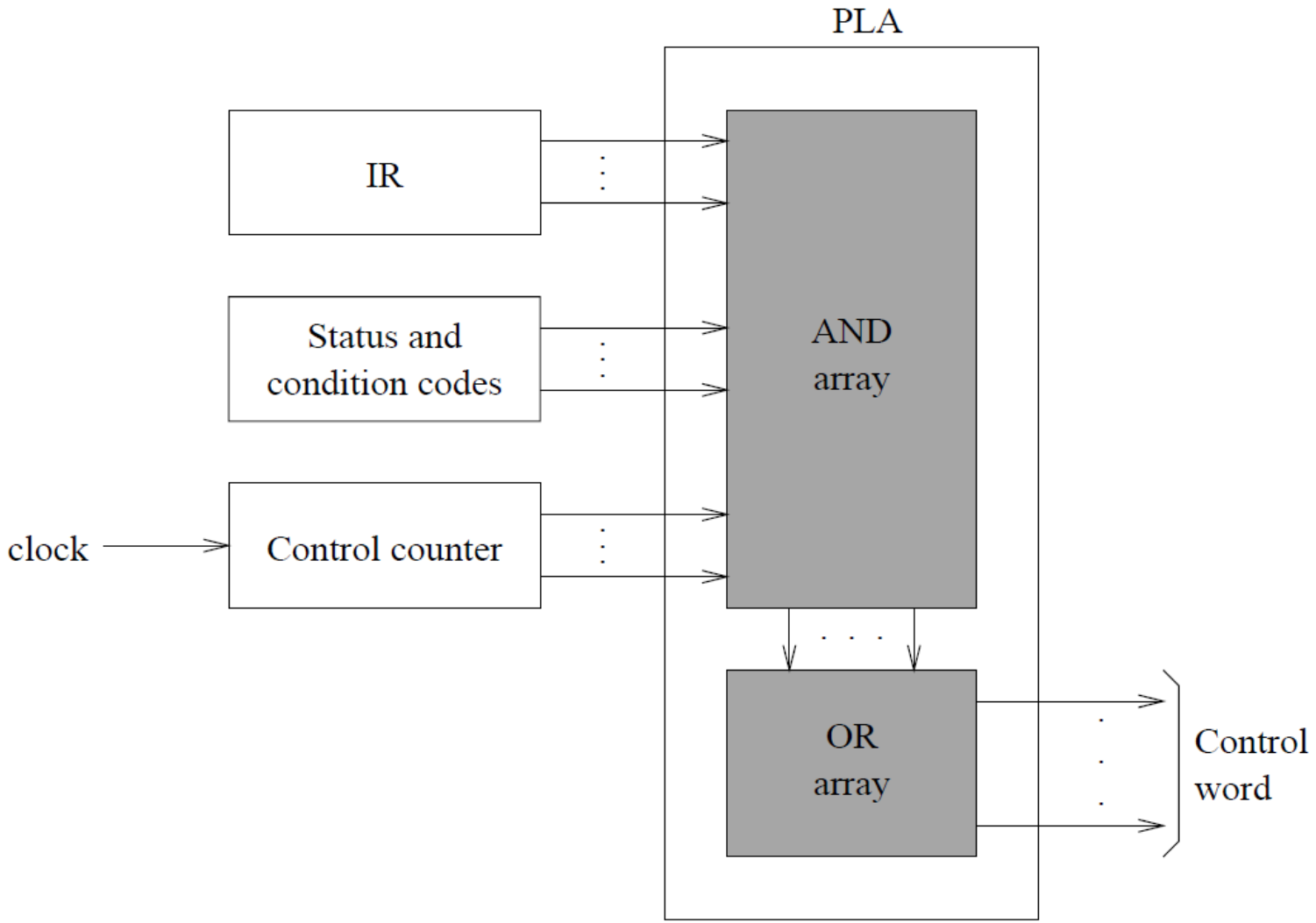
Микропрограмски контролер

- Инструкције се могу описати коначним аутоматима
- Сваки од коначних аутомата може имплементирати хардверски или софтверски

Хардверска имплементација

- Улазе у *PLA* (програмибилни низови логичких елемената) чине три групе сигнала:
  - код операције (*opcode*)
    - операција која се имплементира
  - статусни и условни кодови
    - неопходни за имплементацију условних гранања и неких АЛ операција
  - часовник
    - служи за бројање корака и синхронизацију активности

Дијаграм хардверске имплементације коначног аутомата



Избор имплементације

- Ако су инструкције једноставне, онда је хардверска имплементација прихватљивија варијанта
  - *RISC*
- У случају сложених инструкција хардверска имплементација није добар избор
  - *CISC*
  - тада се сигналима управља програмски

Софтверска имплементација

- Почетком 1950-их је први пут предложено да се контрола извршавања инструкција имплементира софтверски [*Wilkes*, *Stinger*]
  - тзв. *микропрограмски контролери*
  - програмски код инструкције се назива *микро-код*
- Идеја је да се коначни аутомат трансформише у *микро-инструкције* које управљају постављањем сигнала који управљају радом процесора

Секвенца контролних сигнала

Instruction	Step	Control signals
Instruction fetch	S1	`PCbout: read: PCout: ALU=add4: Cin;`
	S2	`read: Cout: PCin;`
	S3	`read: IRbin;`
	S4	Decodes the instruction and jumps to the appropriate execution routine
`add %G9,%G5,%G7`	S1	`G5out: Ain;`
	S2	`G7out: ALU=add: Cin;`
	S3	`Cout: G9in: end;`

* PCbout поставља сигнал на системску магистралу, а PCout на интерну магистралу A

Секвенца контролних сигнала (2)

- Ако претпоставимо да се сваки корак израчунава у једном циклусу:
  - потребно је 3 циклуса за читање
  - бар један циклус за декодирање
  - три циклуса за сабирање

- Већина сигнала је из претходног описа, осим:
  - *read* – постављање сигнала за читање на системску магистралу
  - *ALU* – је група контролних сигнала АЛЈ која укључује одговарајућу операцију
  - *end* – сигнал који означава крај и иницира циклус читања наредне инструкције

Други пример

- Разматрамо само извршавање инструкције
  - регистарско индиректно адресирање једног сабирка

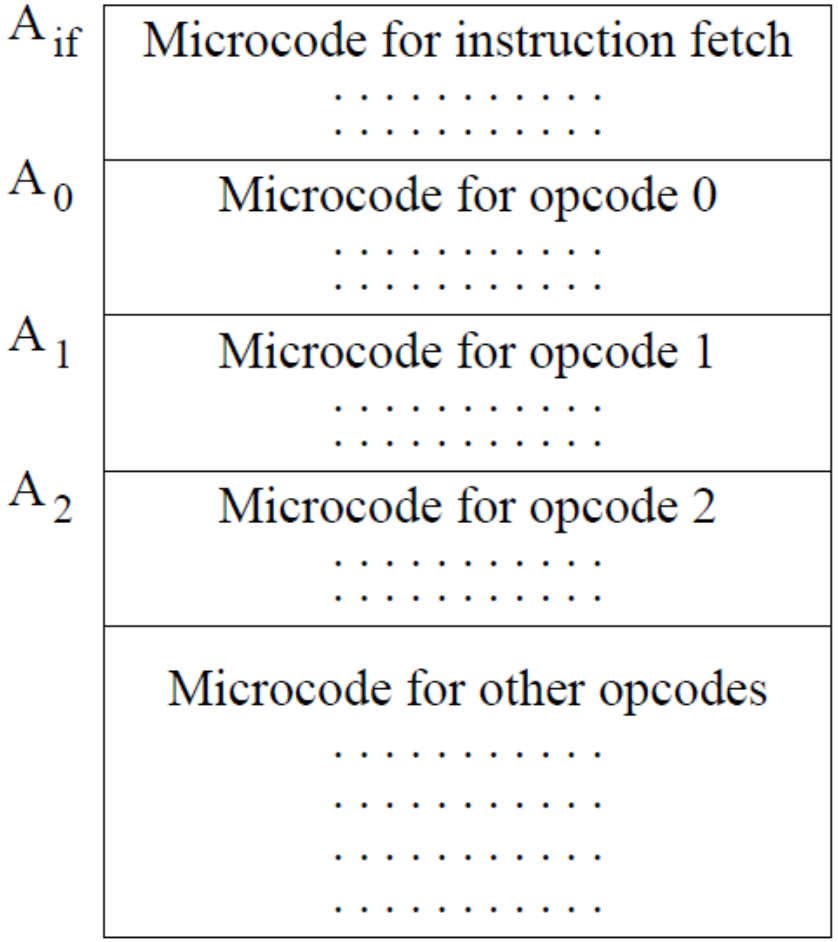
Instruction	Step	Control signals
`add %G9,[%G5],%G7`	S1	`G5out: MARin: MARbout: read;`
	S2	`read;`
	S3	`read: MDRbin: MDRout: Ain;`
	S4	`G7out: ALU=add: Cin;`
	S5	`Cout: G9in: end;`

Концепт микро-кода

- Идеја:
  - сви сигнали једног корака се кодирају у тзв. *кодну реч*
  - добијена кодна реч представља *микроинструкцију*
  - низ микроинструкција који одговара једној инструкцији процесора гради *микрорутину*
  - помоћу микроинструкција се пишу *микропрограми*

Упрошћена организација микрокода

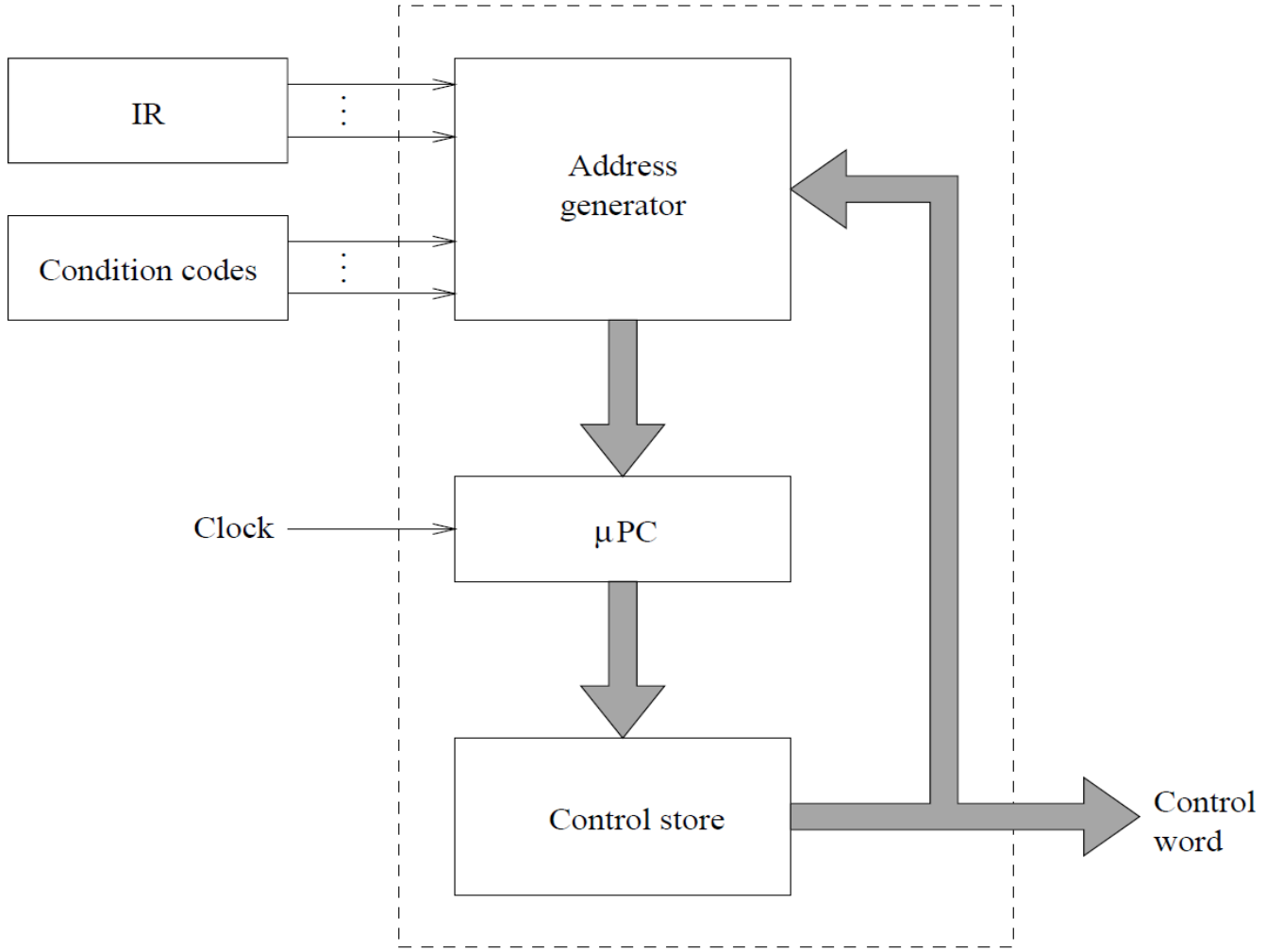
- Линеарна организација микрокода почиње од читања инструкције
- Након декодирања инструкције наставља се од микрорутине која одговара коду операције
- Извршавање микрорутине је секвенцијално
- Када се дође до сигнала *end* понавља се читање (наредне) инструкције



Микропрограмски контролер

- Микропрограм се извршава од стране микропрограмског контролера
- Извршавање почива на микропрограмском бројачу (μPC) – слично бројачу инструкција
- Коло за генерисање адресе
  - иницира почетну адресу (тј. адресу микрорутине за читање инструкције)
  - имплементира микропрограмске скокове
  - користи се и да на основу прочитане инструкције условног гранања и стања контролних битова одабере одговарајући микрокод

Имплементација микроконтролера



Сложена организација микрокода

- Линеарна организација микрокода има за последицу да се неки заједнички делови микрокода понављају више пута
- Сложена организација микрокода подразумева да се заједнички делови ефикасније организују у микрорутине са гранањима
  - свака микроинструкција се проширује адресом наредне микроинструкције
  - због тога се повећава контролна реч
  - али се штеди на дужини микропрограма

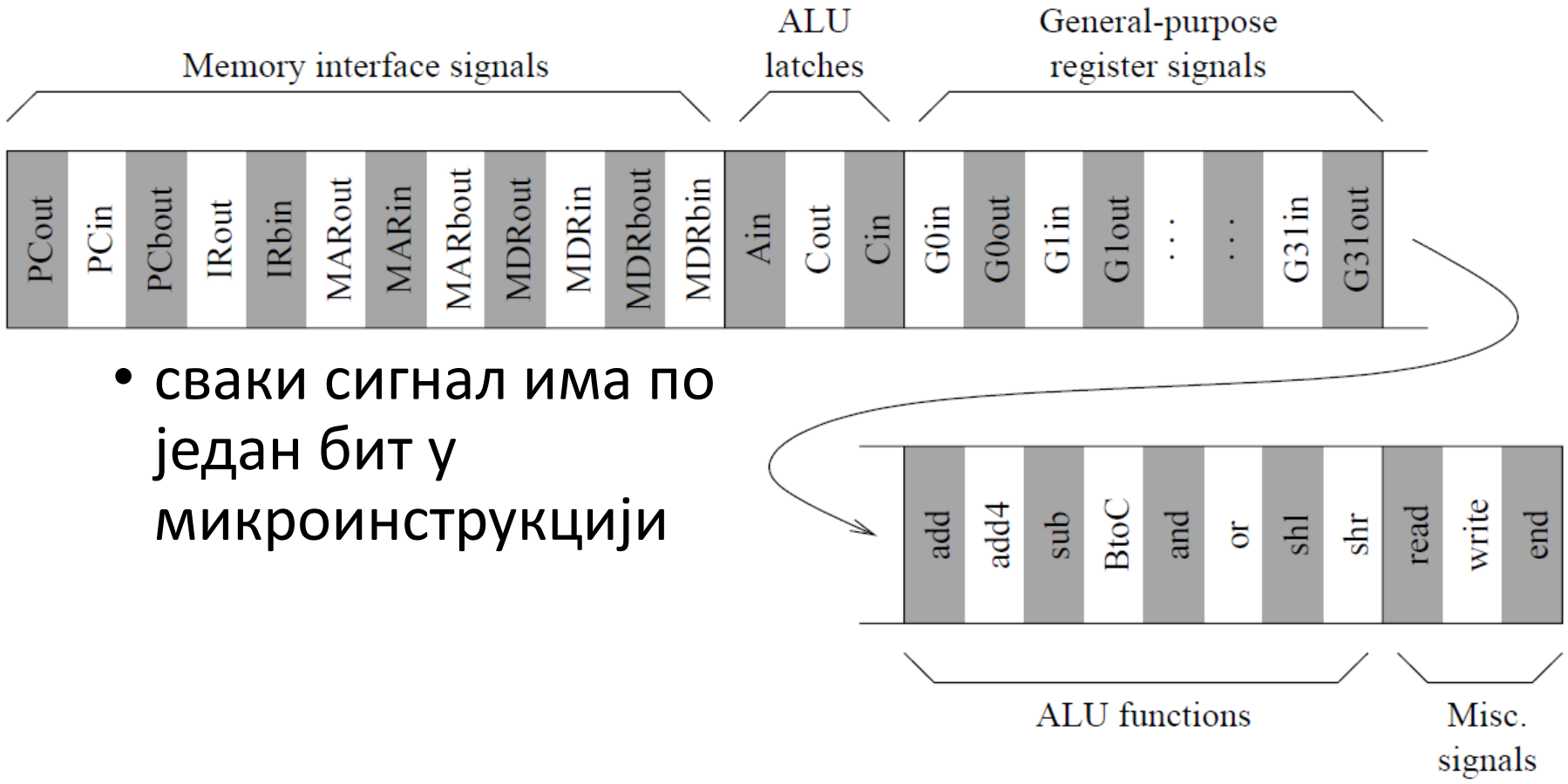
Формат микроинструкција

- Свака микроинструкција се састоји од одговарајућих контролних сигнала
  - првих 12 контролних сигнала долази из имплементације пута података:
    - *PCin, PCbout, PCout, IRbin, IRout, MARin, MARbout, MARout, MDRbin, MDRin, MDRbout, MDRout*
  - наредних 3 бита контролишу резе A и C
    - *Ain, Cin, Cout*
  - 64 сигнала контролишу регистре опште намене
    - *Gxin, Gxout* – по 2 за сваки регистар *Gx*
  - потребан број битова за операције процесора

Формат микроинструкција (2)

- Потребан број битова за операције процесора зависи од скупа подржаних операција
  - претпоставимо да су подржане операције:
    - *add, sub, and, or*
    - *shl, shr* – померање за један бит улево и удесно
    - *add4* – за повећавање програмског бројача
    - *BtoC* – преписивање из једног у други регистар (или меморију)
      - може и без тога, сигналима: *Gxout: Gyin:*

Хоризонтален формат



- сваки сигнал има по један бит у микроинструкцији

Хоризонталан формат (2)

- Сваком сигналу одговара по један бит
- Нема кодирања и декодирања сигнала

- Предности:
  - омогућава навођење великог броја сигнала у једној инструкцији
    - нпр: преписивање *G0* у више регистара:
      - *G0out: G2in: G3in: G4in: G5in: G6in: end;*
- Мане:
  - величина микроинструкције
    - у претходном примеру читавих 90 битова

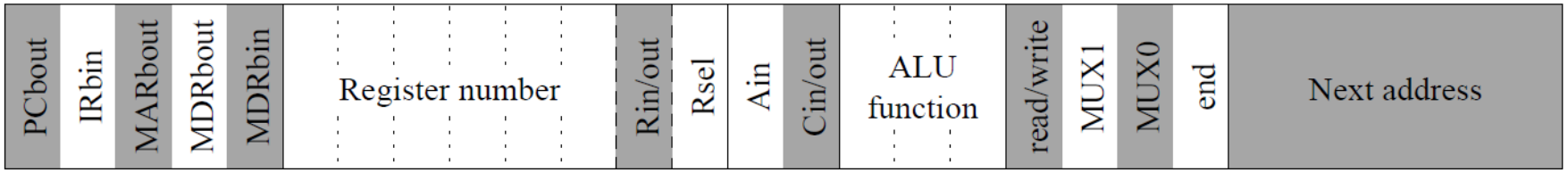
Вертикалан формат

- На дужини се може уштедети ако се битови кодирају
  - на пример, уместо 64 битова за контролисање 32 регистра је довољно 5 битова за идентификовање регистра и 1 бит за избор сигнала
- Предности:
  - краће микроинструкције
- Мане:
  - у једном кораку се може навести само један регистар

Вертикалан формат (2)

- У наставку је пример где ова мана долази до изражаја
- Неопходно је нпр. преписати садржај једног регистра у неколико других
  - Операција *BtoC* у АЛЈ
  - на пример, преписивање *G0* у *G2-G6*:
    - *G0out: ALU=BtoC: Cin;*
    - *Cout: G2in;*
    - *Cout: G3in;*
    - *Cout: G4in;*
    - *Cout: G5in;*
    - *Cout: G6in: end;*

Вертикалан формат (3)

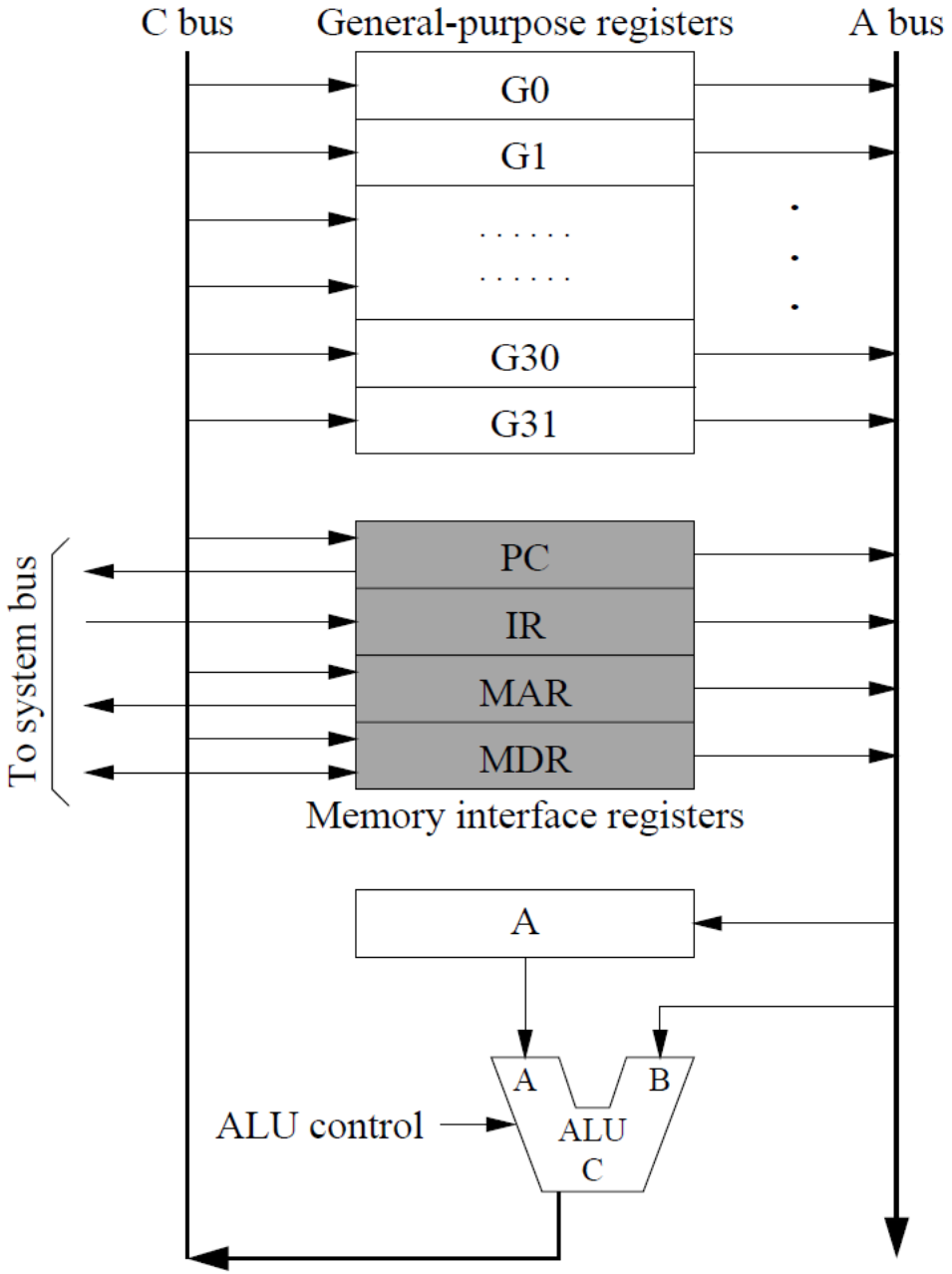


- Још штедљивија вертикална организација:
  - 32 регистра опште намене и 4 специјална регистра (*PC, IR, MAR, MDR*) се кодирају са 6 битова
  - 8 операција се кодирају са 3 бита

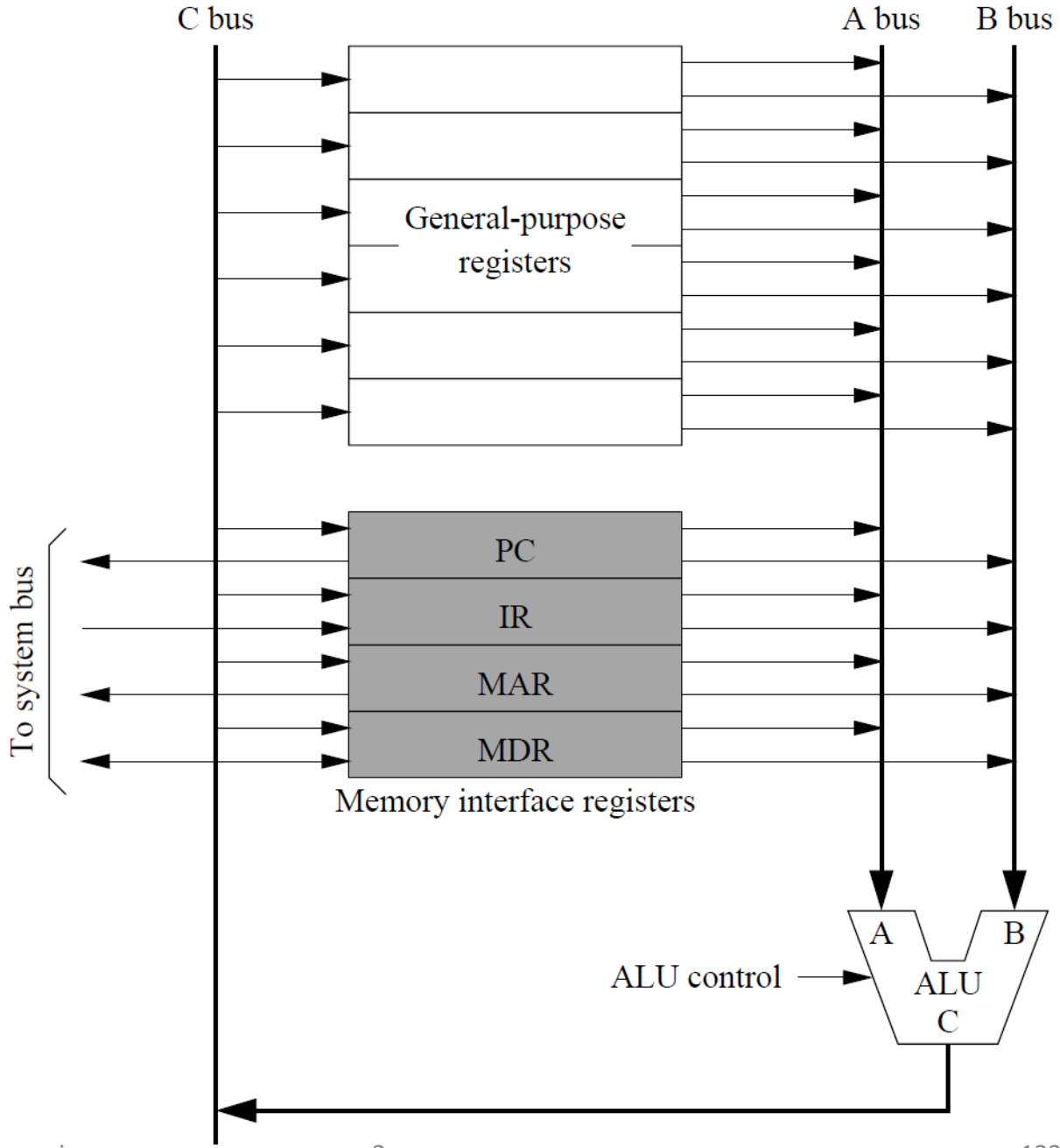
Путеви података са више магистрала

- Представљен пут података има само једну интерну магистралу А
  - основна слабост је милтиплексирање те магистрале и већи број корака у имплементацији инструкција
- Пут података може да има и више интерних магистрала
  - на пример две:
    - магистрала А – за један операнд
    - магистрала С – за резултат
  - или три:
    - магистрала А – за један операнд
    - магистрала В – за други операнд
    - магистрала С – за резултат

Пут података
са две магистрале



Пут података са три магистрале



Примери са више магистрала

- у случају једне магистрале неопходна су три корака
- у случају две магистрале довољна су два корака:

Instruction	Step	Control signals
`add %G9,%G5,%G7`	S1	`G5out: Ain;`
	S2	`G7out: ALU=add: G9in;`

- у случају три магистрале довољан је један корак:

Instruction	Step	Control signals
`add %G9,%G5,%G7`	S1	`G5outA: G7outB: ALU=add: G9in;`