Архитектура и организација рачунара

Милан Банковић

3. септембар 2025.

Садржај

I Основи дигиталне логике **7**

1 Логичке функције и логички изрази **9**

1.1 Булова алгебра 9

1.1.1 Аксиоме и основни закони Булове алгебре 9

1.2 Логички изрази и њихове нормалне форме 11

1.2.1 Конјунктивна и дисјунктивна нормална форма 11

1.2.2 Савршена конјунктивна и дисјунктивна форма 13

1.3 Логичке функције 14

1.3.1 Савршена дисјунктивна (конјунктивна) нормална форма функције 14

1.3.2 Потпуни скупови везника 16

1.3.3 n-арни везници 17

1.4 Минимизација логичких израза 18

1.4.1 Метод алгебарских трансформација 18

1.4.2 Метод Карноових мапа 20

1.4.3 Метод Квин-Мекласког 27

1.4.4 Минимизација у присуству небитних вредности 35

1.4.5 Минимална КНФ форма 38

2 Логичка кола **41**

2.1 О логичким колима 41

2.2 Вредност високе импедансе 42

2.3 Логичке капије 43

2.4 Кашњење логичког кола 44

2.5 Имплементација логичких капија у савременим рачунарима . 46

2.5.1 НЕ коло 48

2.5.2 НИ и И коло 50

2.5.3 НИЛИ и ИЛИ коло 50

2.5.4 ЕИЛИ коло 50

2.5.5 Бафер 51

2.5.6 Бафер са три стања 52

2.5.7 Пропусни транзистори и преносне капије 53

2.5.8 Бафер са три стања и преносне капије 54

2.5.9 ЕИЛИ коло и преносне капије 54

2.5.10 Вишеулазне логичке капије 55

3 Комбинаторна кола **59**
3.1 Основна комбинаторна кола 60
3.1.1 Мултиплексер 60
3.1.2 Демултиплексер 65
3.1.3 Декодер 67
3.1.4 Кодер 68
3.2 Аритметичко-логичка кола 70
3.2.1 Битовске операције 70
3.2.2 Померачи 71
3.2.3 Сабирачи и одузимачи 72
3.2.4 Компаратори 83
3.2.5 Аритметичко-логичка јединица 85
3.3 Општа комбинаторна кола 87
3.3.1 Неизмењиве меморије 87
3.3.2 PLA кола и PAL кола 89

4 Секвенцијална кола **91**
4.1 Резе 94
4.2 Синхрона и асинхрона секвенцијална кола 97
4.3 Флип-флопови 99
4.3.1 SR флип-флоп 100
4.3.2 D флип-флоп 102
4.3.3 JK флип-флоп 103
4.3.4 T флип-флоп 105
4.3.5 Проблем „хватања јединице” 106
4.3.6 Време поставке и време задржавања 108
4.4 Регистри 109
4.5 Меморије 111
4.5.1 Синхроне меморије 112
4.5.2 Конструкција већих меморија помоћу мањих 113
4.5.3 Асинхроне меморије 114
4.5.4 Оптимизација синхроних меморија 118
4.5.5 О произвољном приступу 120
4.5.6 Динамичке меморије 121
4.6 Бројачи 126
4.7 Бројачи са произвољним редоследом стања 129
4.8 Коначни аутомати 132

5 Принцип рада рачунара **137**
5.1 Рачунари са фиксираним програмом 138
5.1.1 Програмирање рачунара са фиксираним програмом . . 144
5.2 Рачунари са ускладиштеним програмом 152
5.2.1 Алгоритам контролне јединице 158
5.2.2 Аутомат контролне јединице 163
5.2.3 Програмирање рачунара са ускладиштеним програмом 164

II Архитектура и организација рачунара 167

6 Централни процесор 169
6.1 Архитектура централног процесора . . . 172
6.1.1 Врсте архитектура према броју операнада . . . 172
6.1.2 Начини адресирања операнада . . . 177
6.1.3 Врсте машинских инструкција . . . 184
6.1.4 Позивање потпрограма и системски стек . . . 187
6.1.5 RISC и CISC архитектуре . . . 198
6.2 Организација централног процесора . . . 201
6.2.1 Путања података . . . 201
6.2.2 Регистри специјалне намене процесора . . . 204
6.2.3 Фазе извршавања инструкције . . . 207
6.2.4 Имплементација контролне јединице . . . 212

7 Оперативна меморија 219
7.1 Меморијска магистрала . . . 219
7.2 Ефикасна организација меморије . . . 226
7.3 Вишебајтни трансфер података . . . 229
7.4 Меморијско поравнање . . . 231
7.5 Испреплетени приступ . . . 233
7.6 Контролер меморије . . . 238
7.7 Перзистентни садржај у меморији . . . 243

8 Кеш меморија 245
8.1 Принципи локалности . . . 246
8.2 Организација кеша . . . 249
8.3 Претрага кеша . . . 251
8.3.1 Потпуно асоцијативни кеш . . . 251
8.3.2 Кеш са директним мапирањем . . . 253
8.3.3 Скуп-асоцијативни кеш . . . 255
8.4 Модификација података у кешу . . . 258
8.4.1 Политика директног уписа . . . 258
8.4.2 Политика одложеног уписа . . . 259
8.5 Замена линија кеша . . . 260
8.5.1 LRU политика . . . 261
8.5.2 Псеудо-LRA политика . . . 262
8.5.3 Политика случајног избора . . . 264
8.6 Раздвојени и унификовани кеш . . . 264
8.7 Нивои кеша . . . 266
8.7.1 Инклузивни и ексклузивни кеш . . . 268
8.7.2 Упис података у случају кеша на више нивоа . . . 270

Део I

Основи дигиталне логике

Глава 1

Логичке функције и логички изрази

У овој глави разматрамо основне градивне елементе који се користе у изградњи савремених рачунарских система. Најпре ћемо се упознати са *Буловом алгебром* која представља логички оквир на коме се заснива рад савремених рачунара. Увешћемо појам логичких функција које су погодне за изражавање операција над подацима записаним у бинарном облику. Разматраћемо представљање логичких функција помоћу логичких израза, као и нормалне форме логичких израза. Главу ћемо завршити разматрањем техника за поједностављивање логичких израза (тј. техника *минимизације логичких израза*).

1.1 Булова алгебра

Булова алгебра, настала средином 19. века, представља једну од најзначајнијих алгебарских структура. Њен првобитни творац је енглески математичар Џорџ Бул (1815-1864), а настала је као резултат Булових напора да логичке законе разматра у оквирима алгебарских система. Због тога се Булова алгебра често назива и *алгебра логике*, иако је савремена формулација Булове алгебре знатно општија и обухвата и многе друге математичке структуре.[1]. Ми ћемо у даљем излагању најпре дати једну општу аксиоматику Булових алгебри, а затим ћемо се фокусирати на тзв. *двоелементну Булову алгебру* у којој постоје само две вредности (*тачно* и *нетачно*) и на којој се управо и заснива рад савремених рачунара.

1.1.1 Аксиоме и основни закони Булове алгебре

Булова алгебра је уређена шесторка $(S, \cdot, +, ^-, 1, 0)$, где је S непразан скуп, $\cdot$ и $+$ две бинарне операције на скупу S, $^-$ унарна операција на скупу S, а 1 и 0 два издвојена елемента скупа S, при чему важе следеће *аксиоме*:

[1] Формулација Булове алгебре каква се данас може наћи у савременој литератури је заправо нешто другачија од оригиналне Булове формулације и резултат је рада других математичара с краја 19. и почетка 20. века. Међутим, и даље се користи назив Булова алгебра, у част Џорџа Була који се сматра пиониром у овој области.

- $(x \cdot y) \cdot z = x \cdot (y \cdot z)$, $(x + y) + z = x + (y + z)$ (асоцијативност)
- $x \cdot y = y \cdot x$, $x + y = y + x$ (комутативност)
- $x \cdot (y + z) = x \cdot y + x \cdot z$, $x + y \cdot z = (x + y) \cdot (x + z)$ (дистрибутивност)
- $x + 0 = x$, $x \cdot 1 = x$ (неутрални елемент)
- $x + \overline{x} = 1$, $x \cdot \overline{x} = 0$ (комплементарност)

Изрази над Буловом алгебром називају се *буловски изрази*. Приликом записивања буловских израза подразумевамо да оператор $^-$ има највиши приоритет, за којим следи оператор $\cdot$, док најнижи приоритет има оператор $+$. Отуда је израз $x+y\cdot z$ еквивалентан изразу $x+(y\cdot z)$, док израз $x\cdot(y+z)$ није еквивалентан изразу $x \cdot y + z$.

Може се доказати да из горњих аксиома следе следећи важни *закони Булове алгебре*[2]

- $x \cdot x = x$, $x + x = x$ (закони идемпотенције)
- $x \cdot 0 = 0$, $x + 1 = 1$ (закони нуле и јединице)
- $x \cdot (x + y) = x$, $x + x \cdot y = x$ (закони апсорпције)
- $\overline{\overline{x}} = x$ (закон двојне негације)
- $\overline{x + y} = \overline{x} \cdot \overline{y}$, $\overline{x \cdot y} = \overline{x} + \overline{y}$ (де-Морганови закони)

У најједноставнијем моделу Булове алгебре скуп S се састоји само из елемената 0 и 1 (које називамо, редом, *логичком нулом* и *логичком јединицом*), при чему су операције $^-$, $\cdot$ и $+$ дефинисане на начин дат у табели 1.1.

x	$\overline{x}$
0	1
1	0

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

Табела 1.1: Таблице операција у алгебри логике

Операцију $+$ називаћемо операцијом *дисјункције* (*ИЛИ* операција, енгл. *OR*). Операцију $\cdot$ називаћемо операцијом *конјункције* (*И* операција, енгл. *AND*). Операцију $^-$ називаћемо операцијом *негације* или *комплемента* (*НЕ* операција, енгл. *NOT*). Овај модел одговара стандардној семантици класичне исказне логике. Често се назива и *двоелементна Булова алгебра* или *алгебра логике*. Отуда ћемо буловске изразе над овако дефинисаном Буловом алгебром називати и *логичким*

[2]Приметимо да сви наведени закони, изузев закона двојне негације, имају две форме, при чему се једна добија од друге тако што се $+$ замени са $\cdot$, а 1 са 0 и обратно. Ово својство је познато и као *принцип дуалности* у Буловој алгебри. Овај принцип је једноставна последица чињенице да то својство имају и горе наведене аксиоме, па се свако извођење неког идентитета из аксиома може заменити њему дуалним извођењем.

изразима. Управо ова двоелементна Булова алгебра је логички оквир који ћемо користити за опис функционисања дигиталних кола која се користе у савременим рачунарима.[3] За овако нешто постоје два главна разлога. Први разлог је то што је имплементација уређаја који имају два стабилна стања релативно једноставна, што омогућава имплементацију уређаја који израчунавају логичке изразе у савременој електронској технологији (тзв. *логичких кола*) на релативно једноставан, јефтин и поуздан начин. Други разлог је то што је стандардне аритметичке операције над бинарним бројевима могуће једноставно описати на језику алгебре логике, што омогућава имплементацију бинарне аритметике помоћу логичких кола.

1.2 Логички изрази и њихове нормалне форме

Логички изрази се састоје из логичких константи (0 и 1) и логичких променљивих (које означавамо са $x, y, z, \dots$) које су повезане логичким везницима $\cdot$, $+$ и $^-$ на произвољан начин. Изрази могу садржати и заграде којима се може променити уобичајени приоритет оператора (највиши приоритет има негација, затим конјункција, па дисјункција). Везник $\cdot$ ћемо често изостављати при писању, као што је уобичајено и у стандардној алгебри (нпр. уместо $x \cdot y$ писаћемо xy).

Свака логичка променљива која учествује у логичком изразу може узети вредност 0 или 1. Придруживање логичких вредности променљивама називамо *валуацијом*. Формално, под валуацијом над скупом логичких променљивих P подразумевамо било коју функцију $v : P \longrightarrow \{0,1\}$. Оваквих функција има $2^{|P|}$ (дакле, коначно много). Јасно је да, на основу дефиниције логичких везника, за сваку унапред фиксирану валуацију v једнозначно можемо израчунати *вредност* израза E, коју ћемо означавати са $I_v(E)$, и која је такође из скупа $\{0,1\}$.[4] За два логичка израза E_1 и E_2 кажемо да су *еквивалентна* ако имају једнаке вредности у свакој валуацији.

Изрази могу бити произвољне сложености и произвољне форме (тј. могу садржати произвољан број логичких везника који могу бити распоређени на произвољан начин). Нама је обично у интересу да изрази са којима радимо буду што једноставнији, као и да буду у некој нама погодној форми. Због тога ћемо често имати потребу да дате изразе трансформишемо (применом логичких закона) у њима еквивалентне изразе који су у некој жељеној форми. У наставку уводимо тзв. *нормалне форме логичких израза*.

1.2.1 Конјунктивна и дисјунктивна нормална форма

Литерал је логички израз који је или логичка променљива или негација логичке променљиве (нпр. x, $\overline{y}$, z). *Елементарна конјункција* је израз који се састоји из конјункције литерала (нпр. $x\overline{y}z\overline{u}\,\overline{v}$). За израз кажемо да је

[3] Напоменимо још да постоји велики број других математичких структура које задовољавају аксиоме Булове алгебре. На пример, ако посматрамо партитивни скуп $\mathbb{P}X$ било ког непразног скупа X и операције уније, пресека и комплемента, тада ће таква структура такође бити модел Булове алгебре. Наравно, овакве Булове алгебре немају примене у дигиталним рачунарима.

[4] Дакле, валуација $v : P \longrightarrow \{0,1\}$ индукује функцију $I_v : \mathcal{E}(P) \longrightarrow \{0,1\}$ која сваком изразу придружује његову вредност одређену том валуацијом.

у *дисјунктивној нормалној форми* (ДНФ), ако се састоји из дисјункције елементарних конјункција (нпр. $x\overline{y}z + \overline{x}\,\overline{y}\,\overline{z} + xy\overline{z}$).

За сваки израз E постоји израз E' у ДНФ који је еквивалентан изразу E. Ово тврђење следи из чињенице да постоји ефективан поступак за трансформацију произвољног израза у еквивалентан ДНФ израз. Он се састоји из следећих корака:

1. Најпре се полазни израз упрошћава тако што се из њега елиминишу све примене логичких везника над логичким константама (0 и 1), ако постоје. Ово се постиже исцрпном применом следећих логичких закона:

$$\overline{0} = 1 \quad \overline{1} = 0 \quad e \cdot 0 = 0 \quad e \cdot 1 = e \quad e + 0 = e \quad e + 1 = 1$$

   где је e произвољан подизраз израза који трансформишемо. Након овог корака, полазни израз се своди или на логичку константу (0 или 1), или на израз који не садржи логичке константе.

2. У другом кораку се израз трансформише тако да се негације примењују искључиво на појединачне логичке променљиве. Ово се постиже исцрпном применом закона двојне негације и де-Морганових закона:

$$\overline{\overline{e}} = e \quad \overline{e_1 + e_2} = \overline{e_1} \cdot \overline{e_2} \quad \overline{e_1 \cdot e_2} = \overline{e_1} + \overline{e_2}$$

   где су e, e_1 и e_2 произвољни подизрази израза који трансформишемо. Након овог корака, израз се састоји из литерала који су повезани којункцијама и дисјункцијама на произвољан начин.

3. у трећем кораку се дисјункције „извлаче" из конјункција, тако што се исцрпно примењује дистрибутивни закон:

$$e \cdot (e_1 + e_2) = e \cdot e_1 + e \cdot e_2$$

   где су e, e_1 и e_2 произвољни подизрази израза који трансформишемо. Након овог корака, добијамо израз који је у ДНФ-у.

Како смо у свим корацима примењивали логичке законе који чувају еквивалентност, следи да ће и коначни ДНФ израз бити еквивалентан са полазним изразом.

Напоменимо да се током примене горњег поступка понекад јавља потреба за применом и других логичких закона, ради даљег упрошћавања израза. На пример, ако након примене дистрибутивног закона добијемо конјункцију која садржи и x и $\overline{x}$, тада је та конјункција еквивалентна са 0 (јер је $x \cdot \overline{x} \cdot e = 0 \cdot e = 0$) и треба је обрисати (јер је $0 + e = e$). Вишеструке појаве истог литерала у конјункцији се могу обрисати (јер важи закон идемпотенције $x \cdot x = x$). Слично, вишеструке појаве истих (до на редослед литерала) конјункција у ДНФ-у се могу обрисати (јер важи закон идемпотенције $e + e = e$). Најзад, уколико имамо две конјункције K_1 и K_2, такве да је скуп литерала прве подскуп литерала друге (тј. $K_2 = K_1 \cdot K'$, где је K' конјункција литерала који се налазе у K_2 а не налазе се у K_1), тада важи $K_1 + K_2 = K_1 + K_1 \cdot K' = K_1$ на основу закона апсорпције (тј. „дужа" конјункција се може обрисати).

Пример 1. *Нека је дат израз* $\overline{\overline{x+\overline{y}\cdot(z+\overline{wy})}}\cdot(\overline{y}+w+(z\cdot 0))$. *Применом првог корака, добијамо израз* $\overline{\overline{x+\overline{y}}\cdot(z+\overline{wy})}\cdot(\overline{y}+w)$. *Затим, применом другог корака добијамо* $(\overline{\overline{x+\overline{y}}}+\overline{z+\overline{wy}})\cdot(\overline{y}+w)$, *односно* $(x+\overline{y}+\overline{z}\cdot\overline{\overline{wy}})\cdot(\overline{y}+w)$, *и најзад* $(x+\overline{y}+\overline{z}wy)\cdot(\overline{y}+w)$. *У трећем кораку примењујемо дистрибутивни закон и добијамо* $x\overline{y}+xw+\overline{y}\overline{y}+\overline{y}w+\overline{z}wy\overline{y}+\overline{z}wyw$. *Даљим упрошћавањем добијамо* $x\overline{y}+xw+\overline{y}+\overline{y}w+\overline{z}wy$. *Најзад, конјункције* $x\overline{y}$ *и* $\overline{y}w$ *се могу обрисати, јер дисјункција садржи* $\overline{y}$. *Коначни ДНФ је* $xw+\overline{y}+\overline{z}wy$.

Аналогно дисјунктивној, можемо дефинисати и *конјунктивну нормалну форму*. Под *елементарном дисјункцијом* подразумевамо израз који се састоји из дисјункције литерала (нпр. $x+\overline{y}+z$). За израз кажемо да је у *конјунктивној нормалној форми* (КНФ) ако се састоји из конјункције елементарних дисјункција (нпр. $(x+\overline{y}+z)\cdot(\overline{x}+\overline{y}+\overline{z})\cdot(y+z)$).

Поступак трансформације у КНФ је аналоган поступку трансформације у ДНФ. Заправо, прва два корака су идентична. Разлика је у трећем кораку, где се исцрпно примењује други дистрибутивни закон:

$$e+e_1\cdot e_2=(e+e_1)\cdot(e+e_2)$$

где су e, e_1 и e_2 произвољни подизрази израза који трансформишемо. Овим се све конјункције „извлаче" из дисјункција. Као и код ДНФ-а, и овде се на крају могу применити додатна упрошћавања, применом дуалних логичких закона идемпотенције, апсорпције и комплементарности.

Пример 2. *Размотримо поново исти израз као у примеру 1. Након другог корака, као и тамо добијамо израз* $(x+\overline{y}+\overline{z}wy)\cdot(\overline{y}+w)$. *Даље, у првој загради примењујемо закон дистрибуције:* $(x+\overline{y}+\overline{z})\cdot(x+\overline{y}+w)\cdot(x+\overline{y}+y)\cdot(\overline{y}+w)$. *Даље можемо приметити да је трећа дисјункција* $(x+\overline{y}+y)$ *еквивалентна са* $x+1$ *(јер је* $\overline{y}+y=1$*), што је даље еквивалентно са* 1. *Како је* $1\cdot e=e$, *ова дисјункција се може изоставити из израза, па добијамо* $(x+\overline{y}+\overline{z})\cdot(x+\overline{y}+w)\cdot(\overline{y}+w)$. *Најзад, применом закона апсорпције, закључујемо да се друга дисјункција може изоставити (јер је њен скуп литерала надскуп скупа литерала треће дисјункције), па добијамо коначни КНФ израз:* $(x+\overline{y}+\overline{z})\cdot(\overline{y}+w)$.

1.2.2 Савршена конјунктивна и дисјунктивна форма

За елементарну конјункцију (дисјункцију) кажемо да је *савршена* у односу на дати коначни скуп променљивих P ако садржи тачно по један литерал за сваку од променљивих из P. На пример, ако је скуп $P=\{x,y,z,w\}$, тада је $x+\overline{y}+\overline{z}+w$ савршена елементарна дисјункција, док $x+y$ то није. Слично, $\overline{x}yz\overline{w}$ је савршена елементарна конјункција (у односу на P), а $x\overline{z}$ то није.

За конјуктивну (дисјунктивну) нормалну форму кажемо да је *савршена* (или *канонска*), ако су све њене елементарне дисјункције (конјункције) савршене. Може се показати да за сваки израз постоји савршена КНФ и ДНФ која му је еквивалентна. Значај савршених КНФ и ДНФ је у томе што се помоћу њих једноставно могу формирати изрази које одговарају произвољним логичким функцијама. Лоша особина ових форми је њихова сложеност, јер често постоје еквивалентне КНФ и ДНФ које су једноставније. На срећу, постоје и поступци помоћу којих се дата савршена

КНФ (ДНФ) може трансформисати у једноставнију, а еквивалентну КНФ (ДНФ) форму. О свему овоме говоримо у наставку.

1.3 Логичке функције

Под *логичком функцијом реда* n подразумевамо било које пресликавање $f : \{0,1\}^n \to \{0,1\}$ које свакој n-торци логичких вредности $(x_1, \ldots, x_n) \in \{0,1\}^n$ придружује логичку вредност $y \in \{0,1\}$. Записиваћемо и $y = f(x_1, \ldots, x_n)$. Логичке променљиве $x_1, \ldots, x_n$ називаћемо *улазима* (или *аргументима*) функције f, а променљиву y *излазом* (или *вредношћу*) функције f. С обзиром да је домен функције f кардиналности 2^n, а кодомен кардиналности 2, следи да је укупан број логичких функција реда n једнак 2^{2^n}. На пример, свих логичких функција реда 0 (тј. функција без аргумената) има укупно $2^{2^0} = 2^1 = 2$ и то су управо константе 0 и 1. Функција реда 1 (са једним аргументом) има укупно $2^{2^1} = 2^2 = 4$, и оне су дате у табели 1.2.

Назив функције	**Вредност функције**
Нула функција	$f(x) = 0$
Јединична функција	$f(x) = 1$
Идентичка функција	$f(x) = x$
Негација	$f(x) = \overline{x}$

Табела 1.2: Логичке функције реда 1

Функција реда 2 (тј. са два аргумента) има $2^{2^2} = 2^4 = 16$, и дате су у табели 1.3.

Оно што примећујемо из ових табела је да се и стандардни логички везници (негација, конјункција и дисјункција) могу разумети као логичке функције реда 1 (негација) односно 2 (конјункција и дисјункција). Важи и обратно: произвољна логичка функција реда 2 се може сматрати бинарним везником. Тако ћемо ексклузивну дисјункцију (енгл. *XOR*) означавати везником $x \oplus y$, Шеферову функцију (познату као *НИ*, енгл. *NAND*) означаваћемо везником $x \uparrow y$, а Пирсову функцију (познату као *НИЛИ*, енгл. *NOR*) означаваћемо везником $x \downarrow y$. Над овако уведеним бинарним везницима се могу формирати логички изрази на уобичајен начин.

Приметимо још и да сваки логички израз дефинише једну логичку функцију чији су улази управо променљиве које се појављују у изразу. Кажемо и да израз *израчунава* ову функцију. Лако се види да су два израза еквивалентна акко израчунавају исту логичку функцију.

1.3.1 Савршена дисјунктивна (конјунктивна) нормална форма функције

Ако погледамо табеле у претходном одељку, видећемо да је вредности свих логичких функција реда 1 и 2 било могуће представити изразима који су изграђени над улазним променљивама функције уз помоћ везника конјункције, дисјункције и негације. Лако се може показати да ово важи и за

Назив функције	**Вредност функције**
Нула функција	$f(x,y) = 0$
Јединична функција	$f(x,y) = 1$
Прва пројекција	$f(x,y) = x$
Друга пројекција	$f(x,y) = y$
Негација прве пројекције	$f(x,y) = \overline{x}$
Негација друге пројекције	$f(x,y) = \overline{y}$
Конјункција	$f(x,y) = x \cdot y$
Дисјункција	$f(x,y) = x + y$
Шеферова функција (НИ) $x \uparrow y$	$f(x,y) = \overline{xy} = \overline{x} + \overline{y}$
Пирсова (Лукашиевичева) функција (НИЛИ) $x \downarrow y$	$f(x,y) = \overline{x + y} = \overline{x} \cdot \overline{y}$
Импликација $x \Rightarrow y$	$f(x,y) = \overline{x} + y$
Импликација $y \Rightarrow x$	$f(x,y) = x + \overline{y}$
Негација импликације $\overline{x \Rightarrow y}$	$x\overline{y}$
Негација импликације $\overline{y \Rightarrow x}$	$\overline{x}y$
Ексклузивна дисјункција $x \oplus y$	$f(x,y) = x\overline{y} + \overline{x}y$
Еквиваленција	$f(x,y) = xy + \overline{x}\overline{y}$

Табела 1.3: Логичке функције реда 2

логичке функције већег реда. На пример, претпоставимо да имамо логичку функцију реда 3 (са улазима x, y и z), дату табелом 1.4.[5] На основу дате табеле увек можемо формирати израз у савршеној дисјунктивној нормалној форми који израчунава дату функцију. Поступак је следећи:

x	y	z	$f(x,y,z)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

Табела 1.4: Пример функције реда 3

- за сваку комбинацију улазних вредности за коју функција има вредност 1 формирамо савршену елементарну конјункцију која је *тачна* само у тој комбинацији (нпр. за тројку $(0,0,0)$ на улазу имаћемо савршену елементарну конјункцију $\overline{x}\,\overline{y}\,\overline{z}$). Дакле, ако је вредност променљиве x у тој комбинацији 1, узимамо литерал x, а ако је вредност 0, узимамо литерал $\overline{x}$ (слично и за друге две променљиве).

[5] Будући да су домени логичких функција коначни, увек их можемо задати и табеларно. Међутим, повећавањем реда функције величина табеле експоненцијално расте, па то није увек погодан начин за задавање логичких функција.

- направимо дисјункцију тако добијених савршених елементарних конјункција (тј. савршену дисјунктивну нормалну форму). Овај израз ће имати вредност 1 акко је бар једна од његових елементарних конјункција тачна, а то ће бити тачно у оним комбинацијама за које функција треба да има вредност 1 (јер смо тако конструисали савршене елементарне конјункције).

У горњем примеру, имаћемо следећи ДНФ:

$$f(x,y,z) = \overline{x}\,\overline{y}\,\overline{z} + \overline{x}yz + x\overline{y}z$$

Дуално, могуће је формирати и израз у савршеној конјунктивној нормалној форми који израчунава дату функцију. Поступак је следећи:

- за сваку комбинацију улазних вредности за коју функција има вредност 0 формирамо савршену елементарну дисјункцију која је *нетачна* само у тој комбинацији (нпр. за тројку $(0,0,1)$ имаћемо дисјункцију $x + y + \overline{z}$). Дакле, ако је вредност променљиве x у тој комбинацији 0 узимамо литерал x, а ако је вредност 1, узимамо литерал $\overline{x}$ (слично и за друге две променљиве).
- формирамо конјункцију овако добијених савршених елементарних дисјункција (тј. савршену конјунктивну нормалну форму). Овај израз ће имати вредност 0 акко је бар једна од његових елементарних дисјункција нетачна, а то је тачно у оним комбинацијама за које функција има вредност 0 (јер смо тако конструисали савршене елементарне дисјункције).

У горњем примеру, имаћемо следећи КНФ:

$$f(x,y,z) = (x + y + \overline{z}) \cdot (x + \overline{y} + z) \cdot (\overline{x} + y + z) \cdot (\overline{x} + \overline{y} + z) \cdot (\overline{x} + \overline{y} + \overline{z})$$

Описани поступци за конструкцију савршене дисјунктивне (конјунктивне) нормалне форме се могу лако уопштити на логичке функције произвољног реда, из чега следи да се било која логичка функција произвољног реда може представити изразом у савршеној дисјунктивној (конјунктивној) нормалној форми.

1.3.2 Потпуни скупови везника

У претходном одељку смо видели да се свака логичка функција произвољног реда може представити логичким изразом који је изграђен над улазним променљивама функције, користећи везнике конјункције, дисјункције и негације (штавише, може бити представљена изразом у ДНФ или КНФ). С обзиром да смо у претходном излагању видели да можемо уводити и друге бинарне везнике, поставља се питање да ли постоје и други скупови везника помоћу којих је могуће изразити све логичке функције. Скупове везника са овом особином називамо *потпуним скуповима везника*.

Уколико је неки скуп везника C потпун скуп везника, тада је и сваки његов надскуп C' такође потпун скуп везника. Отуда се поставља питање минималности потпуног скупа везника у односу на релацију инклузије. На

пример, поменути скуп $C = \{\cdot, +, \overline{\ }\}$ (тј. скуп основних логичких везника) није минималан потпун скуп везника, јер је и његов прави подскуп $C^{\cdot} = \{\cdot, \overline{\ }\}$ такође потпун скуп везника. Заиста, применом закона двојне негације и де-Моргановог закона свака појава везника $+$ се може елиминисати из израза:

$$x + y = \overline{\overline{x + y}} = \overline{\overline{x} \cdot \overline{y}}$$

Потпуно аналогно, може се показати и да је скуп $C^{+} = \{+, \overline{\ }\}$ потпун скуп везника. Са друге стране, скупови $C^{\cdot}$ и C^{+} јесу минимални потпуни скупови везника. Доказ ове чињенице остављамо читаоцу за вежбу.

Друго питање које се поставља је који је најмањи могући број везника који могу чинити неки потпуни скуп везника. Већ смо видели да постоје такви скупови са по два елемента. Природно је поставити питање да ли постоје једночлани потпуни скупови везника? Одговор на ово питање је такође потврдан, јер су скупови $\{\uparrow\}$ и $\{\downarrow\}$ потпуни системи везника. Заиста, из $\overline{x} = \overline{x \cdot x} = x \uparrow x$ и $x \cdot y = \overline{\overline{x \cdot y}} = \overline{\overline{x \cdot y} \cdot \overline{x \cdot y}} = (x \uparrow y) \uparrow (x \uparrow y)$ следи да је скуп $\{\uparrow\}$ потпун скуп везника.[6] Доказ потпуности скупа $\{\downarrow\}$ је аналоган.

1.3.3 n-арни везници

Бинарни логички везници се могу уопштити и посматрати као n-арни везници. Дефинишимо формално n-арне верзије за нас најзначајнијих бинарних везника:

- *n-арна конјункција*: $x_1 \cdot x_2 \cdot x_3 \cdot \ldots \cdot x_n \equiv (\ldots((x_1 \cdot x_2) \cdot x_3) \cdot \ldots) \cdot x_n$. Може се показати да ће n-арна конјункција дати вредност 1 акко су сви x_i једнаки 1.

- *n-арна дисјункција*: $x_1 + x_2 + x_3 + \ldots + x_n \equiv (\ldots((x_1 + x_2) + x_3) + \ldots) + x_n$. Може се показати да ће n-арна дисјункција дати вредност 1 акко је бар једно x_i једнако 1.

- *n-арна ексклузивна дисјункција*: $x_1 \oplus x_2 \oplus x_3 \oplus \ldots \oplus x_n \equiv (\ldots((x_1 \oplus x_2) \oplus x_3) \oplus \ldots) \oplus x_n$. Може се показати да ће n-арна ексклузивна дисјункција дати вредност 1 акко је непаран број вредности x_i једнако 1.

- *n-арни Шеферов (НИ) везник*: $x_1 \uparrow \ldots \uparrow x_n \equiv \overline{x_1 \cdot \ldots \cdot x_n}$. Јасно је да ће n-арни НИ везник дати вредност 1 акко бар једно x_i има вредност 0.

- *n-арни Пирсов (НИЛИ) везник*: $x_1 \downarrow \ldots \downarrow x_n \equiv \overline{x_1 + \ldots + x_n}$. Лако се види да ће n-арни НИЛИ везник дати вредност 1 акко су сви x_i једнаки 0.

Приметимо да се n-арне верзије везника конјункције, дисјункције и ексклузивне дисјункције по дефиницији своде на бинарне верзије ових везника, при чему су термови груписани (асоцирани) на лево. Притом,

[6] И везник $+$ се може представити коришћењем везника $\uparrow$. Наиме, важи $x + y = \overline{\overline{x + y}} = \overline{\overline{x} \cdot \overline{y}} = \overline{\overline{x \cdot x} \cdot \overline{y \cdot y}} = (x \uparrow x) \uparrow (y \uparrow y)$. Међутим, за доказ потпуности довољно је показати да се везници скупа $C^{\cdot}$ могу представити помоћу везника $\uparrow$, с обзиром да је скуп $C^{\cdot}$ потпун скуп везника.

груписање на лево није суштински битно, имајући у виду закон асоцијативности који важи за ове бинарне везнике. Приликом израчунавања n-арних варијанти ових везника изрази се могу груписати и на другачији начин, а не само на лево, као што је наведено у дефиницији. На пример, израз $xyzu$ се може израчунати као $((xy)z)u$ (у складу са формалном дефиницијом), али и као нпр. $x((yz)u)$. Са друге стране, за бинарне везнике НИ и НИЛИ не важи закон асоцијативности. На пример, важи да је $(1 \uparrow 1) \uparrow 0 = 1$, као и да је $1 \uparrow (1 \uparrow 0) = 0$. Отуда није природно n-арне варијанте ових везника дефинисати груписањем и свођењем на одговарајуће бинарне везнике, јер би се поставило питање начина груписања. Најприродније је n-арне НИ и НИЛИ везнике дефинисати као негације n-арних И и ИЛИ везника, јер су на аналоган начин биле дефинисане и бинарне верзије ових везника. Приметимо, притом, да $x \uparrow y \uparrow z \not\equiv (x \uparrow y) \uparrow z$, као и да $x \uparrow y \uparrow z \not\equiv x \uparrow (y \uparrow z)$. Дакле, никавко груписање није дозвољено, јер n-арни НИ везник уопште није дефинисан на тај начин. Исто важи и за n-арни НИЛИ везник.

1.4 Минимизација логичких израза

У овом поглављу бавимо се проблемом минимизације логичких израза. Циљ минимизације је проналажење логичког израза минималне сложености који израчунава неку логичку функцију, или, еквивалентно, проналажење израза минималне сложености који је еквивалентан датом изразу (у случају да је функција задата табеларно, израз који се минимизује је одговарајућа савршена ДНФ (или КНФ) која се добија директно на основу таблице функције). Формално, *сложеност израза* дефинишемо као број везника који се у изразу појављују. Проблем минимизације логичких израза је од великог значаја у процесу дизајна логичких кола која у савременим рачунарима имплементирају логичке изразе, јер се тиме добија значајна уштеда у процесу производње, као и у потрошњи електричне енергије приликом експлоатације уређаја. На жалост, проблем проналажења израза (произвољне форме) минималне сложености који је еквивалентан датом изразу је NP-тежак проблем. Проблем минимизације не постаје лакши ни ако се ограничимо на изразе у ДНФ-у (или КНФ-у). Наиме, доказано је да је проблем проналажења израза у ДНФ-у који израчунава дату логичку функцију, а који садржи минимални број елементарних конјункција такође NP-тежак. Због тога је примена егзактних алгоритама минимизације од користи само у случају функција релативно малог реда. У случају функција великог реда се могу користити разни неегзактни алгоритми засновани на хеуристикама који не гарантују да ће добијени израз бити заиста минималан, али у пракси дају доста добре резултате.[7] У наставку приказујемо неке егзактне методе минимизације логичких израза.

1.4.1 Метод алгебарских трансформација

Метод алгебарских трансформација подразумева примену одређених логичких закона на ДНФ израз у циљу смањивања његове сложености.

[7] Други приступ који се често користи је да се изрази који израчунавају сложеније функције конструишу хијерархијски, полазећи од једноставнијих функција.

Основна идеја методе алгебарских трансформација се заснива на следећим принципима:

- уколико у ДНФ-у имамо две елементарне конјункције облика xK и $\overline{x}K$, где је K произвољна конјункција литерала (другим речима, имамо две конјункције које садрже супротне литерале по једној променљивој, а сви остали литерали су им исти), тада имамо $xK + \overline{x}K = (x + \overline{x}) \cdot K = 1 \cdot K = K$. Овај корак зовемо *груписање* елементарних конјункција. На пример, ако имамо конјункције $xy\overline{z}$ и $x\overline{y}\,\overline{z}$, тада груписањем од ове две конјункције добијамо једну конјункцију $x\overline{z}$ (дакле, уклањамо променљиву по којој се разликују, а задржавамо оно што им је заједничко).

- уколико у ДНФ-у једну те исту којункцију K можемо груписати на два различита начина са две конјункције K_1 и K_2, тада се применом закона идемпотенције ($K = K + K$) конјункција може *удвојити*, тј. могу се направити две копије исте конјункције, при чему се једна групише са K_1, а друга са K_2.

Применом горња два правила на одговарајући начин може се доћи до минималног ДНФ израза. Илуструјмо то следећим примерима.

Пример 3. *Претпоставимо да имамо функцију задату савршеним ДНФ изразом:*

$$F(x, y, z) = \overline{x}\,\overline{y}\,\overline{z} + \overline{x}\,\overline{y}z + \overline{x}yz + x\overline{y}z$$

Приметимо да се прва и друга конјункција могу груписати. Међутим, друга којункција се може груписати и са трећом, али и са четвртом. Због тога ћемо најпре два пута удвојити другу конјункцију:

$$F(x, y, z) = \overline{x}\,\overline{y}\,\overline{z} + (\overline{x}\,\overline{y}z + \overline{x}\,\overline{y}z + \overline{x}\,\overline{y}z) + \overline{x}yz + x\overline{y}z$$

Сада по једну копију друге конјункције групишемо са сваком од преосталих конјункција, тј. добијамо:

$$F(x, y, z) = (\overline{x}\,\overline{y}\,\overline{z} + \overline{x}\,\overline{y}z) + (\overline{x}yz + \overline{x}\,\overline{y}z) + (x\overline{y}z + \overline{x}\,\overline{y}z)$$

одакле следи:

$$F(x, y, z) = \overline{x}\,\overline{y} + \overline{x}z + \overline{y}z$$

Понекад се правило груписања може примењивати и на следећем нивоу, тј. на елементарне конјункције које су већ добијене груписањем. Ову појаву илуструјемо следећим примером.

Пример 4. *Претпоставимо да имамо функцију задату следећим савршеним ДНФ изразом:*

$$F(x, y, z) = \overline{x}\,\overline{y}\,\overline{z} + \overline{x}\,\overline{y}z + \overline{x}yz + x\overline{y}z + \overline{x}y\overline{z}$$

Приметимо да је у питању израз сличан изразу у претходном примеру – једина разлика је у још једној додатној конјункцији $\overline{x}y\overline{z}$. Ова додатна конјункција се може груписати са трећом конјункцијом, због чега нећемо

правити две нове копије друге конјункције, већ само једну. Дакле, након удвајања имамо:

$$F(x,y,z) = \overline{x}\,\overline{y}\,\overline{z} + (\overline{x}\,\overline{y}z + \overline{x}\,\overline{y}z) + \overline{x}yz + x\overline{y}z + \overline{x}y\overline{z}$$

Затим групишемо прву и другу, другу и четврту, као и трећу и пету:

$$F(x,y,z) = (\overline{x}\,\overline{y}\,\overline{z} + \overline{x}\,\overline{y}z) + (x\overline{y}z + \overline{x}\,\overline{y}z) + (\overline{x}yz + \overline{x}y\overline{z})$$

одакле следи:

$$F(x,y,z) = \overline{x}\,\overline{y} + \overline{y}z + \overline{x}y$$

Сада се над добијеним ДНФ-ом може даље вршити груписање (прва и трећа конјункција), одакле добијамо:

$$F(x,y,z) = \overline{x} + \overline{y}z$$

Лоша страна ове методе је то што њена примена није увек тако једноставна, јер није увек могуће тако лако уочити шта се са чим може груписати и шта је потребно удвојити пре груписања. Због тога је ову методу тешко ручно примењивати, а још теже аутоматизовати. Да би се процес груписања и удвајања учинио прегледнијим, као и да би се цео поступак лакше аутоматизовао, развијене су друге методе минимизације које приказујемо у наставку.

1.4.2 Метод Карноових мапа

Метод Карноових мапа је назван по аутору Маурису Карноу (енгл. *Maurice Karnaugh*) који је овај метод први пут увео у употребу 1953. године. У питању је графички метод који поступак груписања чини прегледнијим и омогућава брже препознавање поједностављених елементарних конјункција које чине ДНФ. Нарочито је погодан за ручну примену, јер се ослања на човекову способност да препозна визуелне обрасце.

Карноова мапа је таблица правоугаоног облика чији је укупан број поља једнак 2^n, где је n број променљивих у ДНФ изразу (тј. ред функције). За $n = 3$ имамо правоугаону таблицу димензије 2×4, док за $n = 4$ имамо таблицу димензије 4×4. Свако поље таблице одговара једној валуацији, тј. једној n-торци вредности променљивих (или једној савршеној елементарној конјункцији над улазним променљивама функције). На пример, уколико имамо функцију по три улазне променљиве x, y и z, имаћемо облик мапе приказан на слици 1.1.

	$\overline{x}\overline{y}$	$\overline{x}y$	xy	$x\overline{y}$
$\overline{z}$				
z				

Слика 1.1: Изглед Карноове мапе реда 3

Дакле, по хоризонтали се мењају вредности променљивих x и y тако да поља редом одговарају вредностима (по xy): 00, 01, 11 и 10, док је

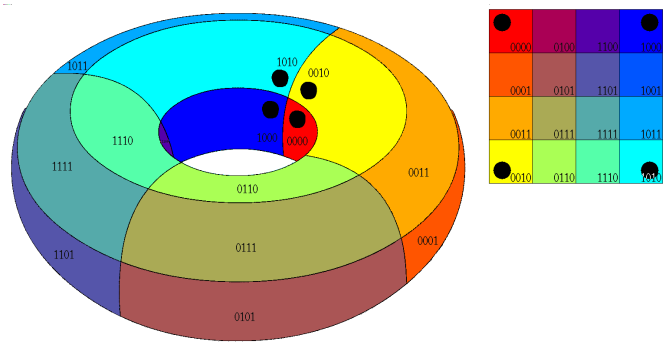
вредност променљиве z фиксирана. По вертикали се мења само вредност променљиве z. Другим речима, свака два суседна поља мапе (по вертикали или хоризонтали) се разликују само по вредности једне променљиве. Приметимо, притом, да се прво и последње поље произвољне врсте такође разликују само по једној променљивој (по променљивој x), па их можемо сматрати суседним пољима, иако визуелно то нису.

У случају да имамо функцију по четири улазне променљиве x, y, z и u, имаћемо облик мапе приказан на слици 1.2.

	$\overline{x}\,\overline{y}$	$\overline{x}y$	xy	$x\overline{y}$
$\overline{z}\,\overline{u}$				
$\overline{z}u$				
zu				
$z\overline{u}$				

Слика 1.2: Изглед Карноове мапе реда 4

Овог пута се по вертикали такође мењају две променљиве, z и u, тако да поља редом одговарају вредностима (по zu): 00, 01, 11 и 10. Дакле, и у овој мапи два суседна поља (по хоризонтали или вертикали) се разликују само по вредности једне променљиве. Као и у претходном случају мапе са три променљиве, и овде се прво и последње поље произвољне врсте (колоне) разликују само по једној променљивој (x, односно z), па се могу сматрати суседним пољима. Другим речима, Карноова мапа се може посматрати и као *торус*, при чему су горња и доња ивица мапе спојене у унутрашњости торуса, док спој леве и десне ивице мапе чини попречни пресек торуса (слика 1.3).



Слика 1.3: Торусни приказ Карноове мапе 4×4

На почетку поступка минимизације, у поља Карноове мапе се упишу одговарајуће вредности функције, односно израза који се минимизује. Уколико два суседна поља (при чему суседност разматрамо у уопштеном,

торусном смислу) садрже јединице, то значи да у савршеној ДНФ форми дате функције имамо две савршене елементарне конјункције које се разликују у поларитету само једне променљиве, па се могу груписати. Слично, ако у мапи имамо четири јединице које формирају (уопштени) правоугаоник, тада се тај правоугаоник заправо састоји из два пара суседних јединица, при чему су та два пара суседна међусобно (тј. омогућавају даље груписање и поједностављивање елементарних конјункција). На пример, у Карноовој мапи 4×4, квадрат 2×2 у горњем левом углу садржи поља која одговарају следећим савршеним елементарним конјункцијама: $\overline{x}\,\overline{y}\,\overline{z}\,\overline{u}$, $\overline{x}\,\overline{y}\,\overline{z}u$, $\overline{x}y\overline{z}\,\overline{u}$, $\overline{x}y\overline{z}u$. Груписањем прве две (лева два поља тог квадрата) и друге две (десна два поља тог квадрата) добијамо конјункције $\overline{x}\,\overline{y}\,\overline{z}$ и $\overline{x}y\overline{z}$. Ове две конјункције су такође „суседне“, јер се разликују само по променљивој y, па њиховим груписањем добијамо $\overline{x}\,\overline{z}$. Литерали $\overline{x}$ и $\overline{z}$ су управо литерали који су заједнички за све четири полазне савршене елементарне конјункције, тј. за сва четири поља овог квадрата. Ова конјункција, дакле, „покрива” ове четири јединице у мапи и обезбеђује да функција заиста има вредност 1 за те вредности улазних променљивих.

Имајући ово у виду, поступак минимизације се састоји у томе да извршимо груписање јединица у мапи, тако да свака јединица буде бар у једној од група. Групе се визуелно означавају заокруживањем. Правила заокруживања су следећа:

- Заокружују се само јединице. Нуле се не смеју заокруживати.
- Свака јединица мора да буде заокружена бар једном. Дозвољено је вишеструко заокруживање јединица.
- Могу се заокруживати искључиво групе од по 2^k поља (уопштеног) правоугаоног облика.
- У циљу минимизације, увек се заокружују што веће групе, чак и ако се том приликом неке јединице поново заокружују (што је, као што смо рекли, дозвољено).
- Након што се све јединице заокруже, треба проверити да ли је неко од заокруживања постало сувишно, јер свако његово поље припада и неком другом заокруживању. Таква сувишна заокруживања се елиминишу.

Сваком од добијених заокруживања одговара једна елементарна конјункција која садржи управо оне литерале који су „заједнички“ за сва поља која обухвата то заокруживање. Што је заокруживање веће, то има мање заједничких литерала, па су конјункције једноставније. Поступак ћемо илустровати следећим примерима.

Пример 5. *Посматрајмо логичку функцију дату табелом 1.5.*

Овој функцији одговара савршена ДНФ:

$$F(x, y, z) = \overline{x}\,\overline{y}\,\overline{z} + \overline{x}\,\overline{y}z + \overline{x}yz + x\overline{y}z$$

(иста као у примеру 3). За дату функцију имамо Карноову мапу дату на слици 1.4.

x	y	z	$F(x,y,z)$
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

Табела 1.5: Табела функције из примера 5

	$\overline{x}\,\overline{y}$	$\overline{x}y$	xy	$x\overline{y}$
$\overline{z}$	1	0	0	0
z	1	1	0	1

Слика 1.4: Карноова мапа за функцију из примера 5

У овом примеру, најбољи начин да се покрију све јединице је да се употребе три заокруживања са по две јединице (јер је очигледно да није могуће заокружити четири јединице једним заокруживањем правоугаоног облика). Овакво заокруживање дато је на слици 1.5.

	$\overline{x}\,\overline{y}$	$\overline{x}y$	xy	$x\overline{y}$
$\overline{z}$	1	0	0	0
z	1	1	0	1

Слика 1.5: Решење примера 5

Приметимо да једно од заокруживања групише поља која су суседна у уопштеном смислу. Вертикалном заокруживању одговара конјункција $\overline{x}\,\overline{y}$, левом хоризонталном заокруживању одговара конјункција $\overline{x}z$, док заокруживању које групише крајње јединице друге врсте одговара конјункција $\overline{y}z$. Отуда је минимални ДНФ израз:

$$F(x,y,z) = \overline{x}\,\overline{y} + \overline{y}z + \overline{x}z$$

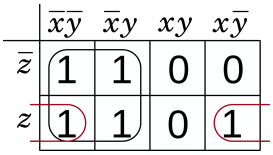
Пример 6. *Посматрајмо функцију дату у табели 1.6.*
Овој функцији одговара савршена ДНФ:

$$F(x,y,z) = \overline{x}\,\overline{y}\,\overline{z} + \overline{x}\,\overline{y}z + \overline{x}y\overline{z} + x\overline{y}z + \overline{x}yz$$

(иста као у примеру 4). За дату функцију имамо Карноову мапу дату на слици 1.6.

x	y	z	$F(x,y,z)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

Табела 1.6: Функција из примера 6



Слика 1.6: Карноова мапа за функцију из примера 6

Дакле, овде имамо једно заокруживање величине 4. *Преостала незаокружена јединица се може груписати са већ груписаном јединицом у доњем левом углу. Добијени минимални ДНФ израз је:*

$$F(x,y,z) = \overline{x} + \overline{y}z$$

Пример 7. *Претпоставимо да је логичка функција дата табелом 1.7.*

Овој функцији одговара Карноова мапа на слици 1.7. Наиме, лако се види да не постоји правоугаоно заокруживање величине 8. *Због тога ћемо најпре заокружити* 4 *поља прве врсте, а након тога и квадрат од* 4 *поља у средишњем делу горње половине мапе. Две јединице у левој половини последње врсте се могу заокружити заједно са две јединице у левој половини прве врсте (иако су већ заокружене, не заборавимо да нам је увек циљ да имамо што већа заокруживања). Остаје да се заокружи још јединица у доњем десном углу. За ову јединицу имамо једно, на први поглед веома чудно, заокруживање. Оно обухвата сва четири угла мапе. Заиста, ако се сетимо торусне интерпретације Карноових мапа, лако се види да ова четири поља заправо чине квадрат* 2×2 *у унутрашњости торуса.*

Након што смо заокружили све јединице, можемо приметити да је заокруживање које обухвата четири јединице прве врсте постало сувишно, с обзиром да су све ове четири јединице касније заокружене поново. Отуда се ово заокруживање може избацити, па добијамо коначну мапу, приказану на слици 1.8.

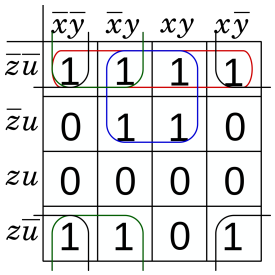
Израз у ДНФ-у који одговара добијеној мапи је:

$$F(x,y,z,u) = y\overline{z} + \overline{x}\,\overline{u} + \overline{y}\,\overline{u}$$

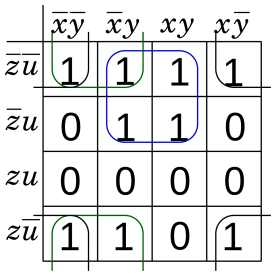
У претходном примеру видели смо да се може догодити да након што заокружимо све јединице, нека заокруживања остану сувишна. У том

x	y	z	u	$F(x,y,z,u)$
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

Табела 1.7: Функција из примера 7



Слика 1.7: Карноова мапа за пример 7



Слика 1.8: Коначно решење примера 7

случају се та сувишна заокруживања уклањају. Са друге стране, може се догодити да се заокруживање јединица може постићи на више различитих начина који дају минималне, али различите ДНФ изразе. Ову појаву

илуструјемо следећим примером.

Пример 8. *Нека је функција дата табелом 1.8.*

x	y	z	u	$F(x,y,z,u)$
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Табела 1.8: Функција из примера 8

Овој таблици одговара Карноова мапа на слици 1.9.

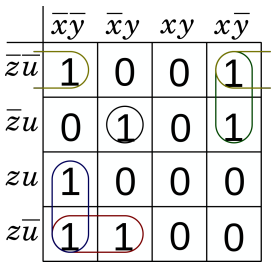
	$\overline{x}\overline{y}$	$\overline{x}y$	xy	$x\overline{y}$
$\overline{z}\overline{u}$	1	0	0	1
$\overline{z}u$	0	1	0	1
zu	1	0	0	0
$z\overline{u}$	1	1	0	0

Слика 1.9: Карноова мапа за функцију из примера 8

Једно могуће заокруживање је дато на слици 1.10. Приметимо да у овом примеру није било могуће наћи заокруживање величине 4*. Јединица у другој врсти и другој колони мапе нема других јединица у суседству, па је зато морамо заокружити саму (ово је најлошија ситуација у минимизацији, јер то значи да ћемо на том месту имати савршену елементарну конјункцију која одговара том пољу). Израз у ДНФ-у који одговара оваквом заокруживању је:*

$$F(x,y,z,u) = \overline{x}z\overline{u} + \overline{x}\,\overline{y}z + x\overline{y}\,\overline{z} + \overline{x}y\overline{z}u + \overline{y}\,\overline{z}\,\overline{u}$$

Међутим, јединица у горњем левом углу мапе је могла бити груписана

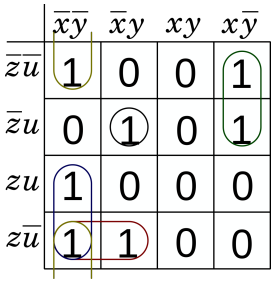


Слика 1.10: Прво решење примера 8

и са јединицом у доњем левом углу. У том случају бисмо имали решење дато на слици 1.11, које даје следећи ДНФ израз:

$$F(x,y,z,u) = \overline{x}z\overline{u} + \overline{x}\,\overline{y}z + x\overline{y}\,\overline{z} + \overline{x}y\overline{z}u + \overline{x}\,\overline{y}\,\overline{u}$$

Ова два израза су једнаке сложености, па је свеједно који ћемо изабрати. Дакле, видимо да поступак минимизације не даје увек једнозначан резултат.



Слика 1.11: Друго решење примера 8

Проблем методе Карноових мапа је у томе што је њена примена на логичке функције реда већег од 4 веома отежана. Наиме, по свакој димензији Карноове мапе могуће је мењати вредности највише две променљиве, док су остале променљиве фиксиране. То значи да у случају дводимензионе мапе можемо имати највише 4 променљиве. Уколико желимо да минимизујемо функцију реда већег од 4, потребно је разматрати вишедимензионе Карноове мапе, које су у општем случају k-димензиони паралелотопи, где је $k = \lceil n/2 \rceil$ (за $n \leq 4$ имамо 2-димензиони паралелотоп, тј. правоугаоник, док за $4 < n \leq 6$ имамо 3-димензиони паралелотоп, тј. квадар, итд.). Ово отежава визуелизацију и смањује прегледност, чиме се губи главна добра особина Карноових мапа, а то је могућност једноставног уочавања груписаних јединица.

1.4.3 Метод Квин-Мекласког

Метод Квин-Мекласког је метод који су развили Вилард Квин (*Willard Quine*) и Едвард Мекласки (*Edward McCluskey*). Ова метода је

функционално идентична претходним двема методама, али је погоднија за аутоматизацију, тј. имплементацију у рачунару. Такође, примењива је на функције произвољног реда.[8]

Идеја алгоритма је да се најпре систематски изврши груписање на све могуће начине. Овај поступак се оптимизује тако што се најпре све савршене елементарне конјункције класификују по броју неинвертованих литерала, како би се смањио број парова конјункција за које треба проверити да ли се могу груписати. Груписање се обавља у више итерација: најпре групишемо по две савршене конјункције, затим по четири, па по осам, итд. Веће групе покривају мање, тако да на крају остају само максималне групе. Након што се груписање заврши, разматрају се конјункције које треба укључити у финални ДНФ израз. С обзиром да је груписање извршено на све могуће начине, међу издвојеним конјункцијама (које се у овој методи називају *прости импликанти* (енгл. *prime implicants*), а које одговарају заокруживањима код Карноових мапа) обично има сувишних, па их је потребно елиминисати. То се ради тако што се најпре идентификују тзв. *битни прости импликанти* (енгл. *essential prime implicants*), а то су оне прости импликанти који морају да буду присутни у ДНФ-у јер су једини прости импликанти који покривају неку од почетних савршених којункција. Након што се издвоје битни импликанти, морамо проверити да ли су њима покривене све полазне савршене конјункције. Ако нису, тада је међу преосталим простим импликантима потребно издвојити најмањи могући подскуп оних који покривају преостале непокривене савршене конјункције.

Поступак се може прецизно описати алгоритмом. Улаз у алгоритам је израз у савршеној ДНФ форми који представља задату функцију. Најпре се савршене елементарне конјункције овог израза сортирају растуће по броју неинвертованих литерала, након чега се деле у класе: i-ту класу чине оне конјункције које садрже тачно i неинвертованих литерала.

У првој фази алгоритма врши се груписање. Ова фаза је подељена у итерације. У првој итерацији групишу се парови савршених конјункција. С обзиром да се две савршене конјункције могу груписати само ако се разликују у поларитету тачно једног литерала, јасно је да такве две конјункције морају бити у суседним класама. Зато се разматрају парови суседних класа i и $i+1$ (за $i = 0, 1, \ldots, n-1$). За сваки пар суседних класа се разматрају сви могући парови конјункција, при чему је прва из i-те, а друга из $(i+1)$-ве класе. Ако се две конјункције могу груписати, тада се резултат њиховог груписања (а то је елементарна конјункција са $n-1$ литерала) преноси у следећу итерацију, а полазне конјункције се означавају као *покривене*.

У следећој итерацији се идентичан поступак груписања примењује над конјункцијама које су пренете из претходне итерације, а добијене конјункције се преносе у наредну итерацију, итд. Прва фаза алгоритма се завршава онда када у текућој итерацији није могуће извршити ни једно груписање, тј. ни једна конјункција се не преноси у следећу итерацију. Све конјункције које су остале непокривене у свим итерацијама прве фазе чине тзв. *просте импликанте* који се преносе у другу фазу алгоритма.

У другој фази алгоритма се формира *табела простих импликаната*.

[8] Ипак, њена сложеност је у општем случају експоненцијална, што је и за очекивати, с обзиром да решавамо NP-тежак проблем.

Колоне ове табеле означене су савршеним конјункцијама из почетног ДНФ израза који се минимизује (тј. конјункције које морамо покрити простим импликантима). Врсте ове табеле означене су простим импликантима пренетим из прве фазе. Најпре означавамо (нпр. симболом +) сва поља табеле која имају особину да је одговарајући прости импликант те врсте садржан у савршеној конјункцији те колоне. Овим смо обележили који импликанти покривају које конјункције. Након тога идентификујемо *битне просте импликанте*: посматрамо колоне у којима постоји само једно обележено поље (што значи да за те савршене конјункције постоји само по један прост импликант који их покрива). Импликанти из одговарајућих врста су битни прости импликанти. Затим се посматра да ли постоје конјункције које нису покривене битним простим импликантима (тј. колоне у којима ни једно од означених поља не припада врстама које одговарају битним простим импликантима). Уколико има таквих конјункција, тада покушавамо да пронађемо *додатне просте импликанте*, тј. тражимо најмањи могући подскуп преосталих простих импликаната који покривају преостале савршене конјункције.

Пример 9. *Размотримо поново функцију из примера 3:*

$$F(x,y,z) = \overline{x}\,\overline{y}\,\overline{z} + \overline{x}\,\overline{y}z + \overline{x}y\overline{z} + x\overline{y}z$$

У првој фази алгоритма, најпре ћемо разврстати савршене елементарне конјункције према броју неинвертованих литерала:

0	$\overline{x}\,\overline{y}\,\overline{z}$
1	$\overline{x}\,\overline{y}z$
	$\overline{x}y\overline{z}$
2	$\overline{x}yz$
	$x\overline{y}z$

Груписањем у првој итерацији добијамо:

0	$\overline{x}\,\overline{y}\,\overline{z}\surd$	$\overline{x}\,\overline{y}$
1	$\overline{x}\,\overline{y}z\surd$	$\overline{x}z$
		$\overline{y}z$
2	$\overline{x}yz\surd$	
	$x\overline{y}z\surd$	

Симболом $\surd$ означене су конјункције које су покривене, тј. које су груписане на бар један начин. Конјункције које су резултат груписања и које се преносе у следећу итерацију су записане у следећој колони горње таблице. Приметимо да се конјункције које добијамо за следећу итерацију једноставно разврставају на исти начин, по броју неинвертованих литерала, с обзиром да груписањем конјункција из i-те и $(i+1)$-ве класе добијамо конјункцију која има i неинвертованих литерала, па ће бити у i-тој класи у следећој итерацији. Даље груписање у нашем примеру није могуће, па све конјункције из друге итерације остају непокривене (тј. то су управо прости импликанти).

У другој фази формирамо табелу простих импликаната:

	$\overline{x}\,\overline{y}\,\overline{z}$	$\overline{x}\,\overline{y}z$	$\overline{x}yz$	$x\overline{y}z$
$\overline{x}\,\overline{y}$	+	+		
$\overline{x}z$		+	+	
$\overline{y}z$		+		+

Дакле, за сваку врсту, симболом + означена су поља која одговарају савршеним конјункцијама које садрже одговарајући импликант. Сада идентификујемо битне импликанте (тј. тражимо плусеве који су једини у својој колони, као и импликанте који им одговарају).

	$\overline{x}\,\overline{y}\,\overline{z}$	$\overline{x}\,\overline{y}z$	$\overline{x}yz$	$x\overline{y}z$
$\overline{x}\,\overline{y}$	$\oplus$	$+$		
$\overline{x}z$		$+$	$\oplus$	
$\overline{y}z$		$+$		$\oplus$

Затим треба одредити које све савршене конјункције покривају битни импликанти (ове плусеве ћемо уоквирити, да бисмо их разликовали од заокружених плусева који идентификују битне импликанте):

	$\overline{x}\,\overline{y}\,\overline{z}$	$\overline{x}\,\overline{y}z$	$\overline{x}yz$	$x\overline{y}z$
$\overline{x}\,\overline{y}$	$\oplus$	$\boxplus$		
$\overline{x}z$		$\boxplus$	$\oplus$	
$\overline{y}z$		$\boxplus$		$\oplus$

Дакле, уоквирујемо све плусеве који су у истој врсти са неким заокруженим плусом. Након тога, проверавамо да ли постоји нека непокривена савршена конјункција (тј. колона у којој ни један плус није ни заокружен, ни уоквирен). У нашем примеру таквих конјункција нема, па битни импликанти чине управо минимални ДНФ израз:

$$F(x,y,z) = \overline{x}\,\overline{y} + \overline{x}z + \overline{y}z$$

Пример 10. *Размотримо сада израз из примера 4:*

$$F(x,y,z) = \overline{x}\,\overline{y}\,\overline{z} + \overline{x}\,\overline{y}z + \overline{x}yz + x\overline{y}z + \overline{x}y\overline{z}$$

У првој фази имамо:

0	$\overline{x}\,\overline{y}\,\overline{z}\surd$	$\overline{x}\,\overline{y}\surd$	$\overline{x}$
		$\overline{x}\,\overline{z}\surd$	
1	$\overline{x}\,\overline{y}z\surd$	$\overline{x}z\surd$	
	$\overline{x}y\overline{z}\surd$	$\overline{y}z$	
		$\overline{x}y\surd$	
2	$\overline{x}yz\surd$		
	$x\overline{y}z\surd$		

Дакле, у овом примеру је било могуће груписати и конјункције у другој итерацији (друга колона горње таблице), из чега је проистекла конјункција $\overline{x}$ који се преноси у трећу итерацију (трећа колона горње табеле). Даље груписање није могуће, а прости импликанти су $\overline{y}z$ и $\overline{x}$. У другој фази формирамо табелу простих импликаната:

	$\overline{x}\,\overline{y}\,\overline{z}$	$\overline{x}\,\overline{y}z$	$\overline{x}yz$	$x\overline{y}z$	$\overline{x}y\overline{z}$
$\overline{x}$	$\oplus$	$\boxplus$	$\oplus$		$\oplus$
$\overline{y}z$		$\boxplus$		$\oplus$	

Како нема непокривених колона, минимални ДНФ је:

$$F(x,y,z) = \overline{x} + \overline{y}z$$

Пример 11. *Посматрајмо поново функцију из примера 7 (табела 1.7). Овој функцији одговара савршена ДНФ форма:*

$$F(x,y,z,u) = \overline{x}\,\overline{y}\,\overline{z}\,\overline{u}+\overline{x}\,\overline{y}z\overline{u}+\overline{x}y\overline{z}\,\overline{u}+\overline{x}y\overline{z}u+\overline{x}yz\overline{u}+x\overline{y}\,\overline{z}\,\overline{u}+x\overline{y}z\overline{u}+xy\overline{z}\,\overline{u}+xy\overline{z}u$$

у првој фази имамо:

0	$\overline{x}\,\overline{y}\,\overline{z}\,\overline{u}\surd$	$\overline{x}\,\overline{y}\,\overline{u}\surd$	$\overline{x}\,\overline{u}$
		$\overline{x}\,\overline{z}\,\overline{u}\surd$	$\overline{y}\,\overline{u}$
		$\overline{y}\,\overline{z}\,\overline{u}\surd$	$\overline{z}\,\overline{u}$
1	$\overline{x}\,\overline{y}z\overline{u}\surd$	$\overline{x}z\overline{u}\surd$	$y\overline{z}$
	$\overline{x}y\overline{z}\,\overline{u}\surd$	$\overline{y}z\overline{u}\surd$	
	$x\overline{y}\,\overline{z}\,\overline{u}\surd$	$\overline{x}y\overline{z}\surd$	
		$\overline{x}y\overline{u}\surd$	
		$y\overline{z}\,\overline{u}\surd$	
		$x\overline{y}\,\overline{u}\surd$	
		$x\overline{z}\,\overline{u}\surd$	
2	$\overline{x}y\overline{z}u\surd$	$y\overline{z}u\surd$	
	$\overline{x}yz\overline{u}\surd$	$xy\overline{z}\surd$	
	$x\overline{y}z\overline{u}\surd$		
	$xy\overline{z}\,\overline{u}\surd$		
3	$xy\overline{z}u\surd$		

Дакле, прости импликанти су четири двочлане конјункције из последње итерације. Сада формирамо табелу простих импликаната:

	$\overline{x}\,\overline{y}\,\overline{z}\,\overline{u}$	$\overline{x}\,\overline{y}z\overline{u}$	$\overline{x}y\overline{z}\,\overline{u}$	$\overline{x}y\overline{z}u$	$\overline{x}yz\overline{u}$	$x\overline{y}\,\overline{z}\,\overline{u}$	$x\overline{y}z\overline{u}$	$xy\overline{z}\,\overline{u}$	$xy\overline{z}u$
$\overline{x}\,\overline{u}$	⊞	⊞	⊞		⊕				
$\overline{y}\,\overline{u}$	⊞	⊞				⊞	⊕		
$\overline{z}\,\overline{u}$	+		+			+		+	
$y\overline{z}$			⊞	⊕				⊞	⊕

Дакле, имамо три битна импликанта који покривају све колоне. Отуда, минимални ДНФ је:

$$F(x,y,z,u) = \overline{x}\,\overline{u} + \overline{y}\,\overline{u} + y\overline{z}$$

Приметимо да је конјункција $\overline{z}\,\overline{u}$ сувишна. Она управо одговара заокруживању свих четири поља прве врсте у примеру 7 које се на крају такође показало као сувишно.

Пример 12. *Нека је дата функција као у примеру 8 (табела 1.8). Овој функцији одговара следећа савршена ДНФ:*

$$F(x,y,z,u) = \overline{x}\,\overline{y}\,\overline{z}\,\overline{u} + \overline{x}\,\overline{y}z\overline{u} + \overline{x}\,\overline{y}zu + \overline{x}y\overline{z}u + \overline{x}yz\overline{u} + x\overline{y}\,\overline{z}\,\overline{u} + x\overline{y}\,\overline{z}u$$

У првој фази имамо:

0	$\overline{x}\,\overline{y}\,\overline{z}\,\overline{u}\surd$	$\overline{x}\,\overline{y}\,\overline{u}$
		$\overline{y}\,\overline{z}\,\overline{u}$
1	$\overline{x}\,\overline{y}z\overline{u}\surd$	$\overline{x}\,\overline{y}z$
	$x\overline{y}\,\overline{z}\,\overline{u}\surd$	$\overline{x}z\overline{u}$
		$x\overline{y}\,\overline{z}$
2	$\overline{x}\,\overline{y}zu\surd$	
	$\overline{x}y\overline{z}u$	
	$\overline{x}yz\overline{u}\surd$	
	$x\overline{y}\,\overline{z}u\surd$	

У другој итерацији није могуће даље груписање, па су прости импликанти све конјункције из друге итерације, уз једну непокривену конјункцију из почетне итерације. Сада је табела простих импликаната:

	$\overline{x}\,\overline{y}\,\overline{z}\,\overline{u}$	$\overline{x}\,\overline{y}z\overline{u}$	$\overline{x}\,\overline{y}zu$	$\overline{x}y\overline{z}u$	$\overline{x}yz\overline{u}$	$x\overline{y}\,\overline{z}\,\overline{u}$	$x\overline{y}\,\overline{z}u$
$\overline{x}y\overline{z}u$				$\oplus$			
$\overline{x}\,\overline{y}\,\overline{u}$	$+$	$+$					
$\overline{y}\,\overline{z}\,\overline{u}$	$+$					$+$	
$\overline{x}\,\overline{y}z$		$\boxplus$	$\oplus$				
$\overline{x}z\overline{u}$		$\boxplus$			$\oplus$		
$x\overline{y}\,\overline{z}$						$\boxplus$	$\oplus$

Из табеле се види да су битни прости импликанти $\overline{x}y\overline{z}u$, $\overline{x}\,\overline{y}z$, $\overline{x}z\overline{u}$ и $x\overline{y}\,\overline{z}$. Такође, видимо да битни прости импликанти не покривају савршену конјункцију $\overline{x}\,\overline{y}\,\overline{z}\,\overline{u}$. Због тога је потребно изабрати додатне просте импликанте који ће покрити ову конјункцију. Од два преостала проста импликанта $\overline{x}\,\overline{y}\,\overline{u}$ и $\overline{y}\,\overline{z}\,\overline{u}$ оба могу покрити конјункцију $\overline{x}\,\overline{y}\,\overline{z}\,\overline{u}$, па можемо изабрати било који од та два. Отуда имамо две могуће минималне ДНФ форме:

$$F(x,y,z,u) = \overline{x}y\overline{z}u + \overline{x}\,\overline{y}z + \overline{x}z\overline{u} + x\overline{y}\,\overline{z} + \overline{x}\,\overline{y}\,\overline{u}$$

и

$$F(x,y,z,u) = \overline{x}y\overline{z}u + \overline{x}\,\overline{y}z + \overline{x}z\overline{u} + x\overline{y}\,\overline{z} + \overline{y}\,\overline{z}\,\overline{u}$$

Овај резултат одговара резултату примера 8. Дакле, одабир додатних простих импликаната често није једнозначан и одговара различитим заокруживањима код Карноових мапа.

Иако је у претходном примеру одабир додатних простих импликаната деловао као једноставан корак, у општем случају то није тако. Подсетимо се да је нама циљ да од преосталих простих импликаната изаберемо најмањи могући скуп[9] додатних простих импликаната који покривају преостале савршене конјункције. Уколико имамо велики број простих импликаната који нису битни прости импликанти, тада постоји и велики број могућих подскупова простих импликаната које треба разматрати, што чини овај последњи корак алгоритма тешким. Заправо, баш овај последњи корак

[9] Кад кажемо *најмањи могући*, мислимо скуп простих импликанта са најмањим бројем конјункција. Уколико два скупа простих импликаната имају једнак број конјункција, тада преферирамо онај који има мањи укупан број литерала, тј. чије су конјункције једноставније.

алгоритма у најгорем случају има експоненцијалну сложеност (с обзиром да скуп преосталих простих импликаната има експоненцијално много подскупова) док се претходни кораци алгоритма увек могу извршити у полиномијалном времену. Један од метода који се обично користи за проналажење траженог најмањег подскупа је тзв. *Петриков метод* (*Stanley Petrick*). Он се састоји у следећем:

- Најпре се из табеле простих импликаната обришу врсте које одговарају битним простим импликантима, као и колоне које одговарају савршеним конјункцијама које су покривене битним простим импликантима (другим речима, остају само преостали прости импликанти и непокривене савршене конјункције).
- i-тој врсти упрошћене табеле се придружује новоуведена логичка променљива p_i.
- За j-ту колону формирамо дисјункцију $D_j = p_{i_1} + p_{i_2} + \ldots + p_{i_{k_j}}$, при чему су $p_{i_1}, p_{i_2}, \ldots, p_{i_{k_j}}$ логичке променљиве које одговарају врстама чији прости импликанти покривају j-ту савршену конјункцију. Сада формирамо КНФ формулу $K = D_1 \cdot D_2 \cdot \ldots \cdot D_m$, где је m број колона табеле.
- Избор подскупа преосталих простих импликаната сада одговара избору подскупа логичких променљивих p_i које ће имати вредност 1. Изабрани подскуп ће покривати све преостале савршене конјункције акко одговарајућа валуација променљивих p_i задовољава формулу K.
- Формула K се сада сведе на ДНФ применом закона дистрибуције. Притом, да би се смањила сложеност овог поступка (која је у најгорем случају експоненцијална), примена закона дистрибуције се комбинује са законом апсорпције где год је то могуће ($X + XY = X$).
- Добијени ДНФ чине конјункције логичких променљивих p_i које одговарају минималним (у смислу инклузије) скуповима преосталих простих импликаната који покривају све преостале савршене конјункције. Међу њима бирамо ону са најмањим бројем логичких променљивих (тј. најмањим бројем простих импликаната). Уколико има више таквих, онда међу њима бирамо ону која одговара скупу простих импликаната са најмањим укупним бројем литерала.

Пример 13. *Претпоставимо да имамо логичку функцију дату у табели 1.9.*

У првој фази методе Квин-Мекласког добијамо:

0	$\overline{x}\,\overline{y}\,\overline{z}\,\overline{u}\surd$	$\overline{x}\,\overline{y}\,\overline{z}$	
		$\overline{x}\overline{z}\,\overline{u}$	
1	$\overline{x}\,\overline{y}\,\overline{z}u\surd$	$\overline{y}\,\overline{z}u$	
	$\overline{x}y\overline{z}\,\overline{u}\surd$	$\overline{x}y\overline{u}$	
		$y\overline{z}\,\overline{u}$	
2	$\overline{x}yz\overline{u}\surd$	$\overline{x}yz$	
	$x\overline{y}\,\overline{z}u\surd$	$x\overline{z}u$	
	$xy\overline{z}\,\overline{u}\surd$	$xy\overline{z}$	
3	$\overline{x}yzu\surd$		
	$xy\overline{z}u\surd$		

x	y	z	u	$F(x,y,z,u)$
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

Табела 1.9: Функција из примера 13

Табела простих импликаната је:

	$\overline{x}\,\overline{y}\,\overline{z}\,\overline{u}$	$\overline{x}\,\overline{y}\,\overline{z}u$	$\overline{x}y\overline{z}\,\overline{u}$	$\overline{x}yz\overline{u}$	$\overline{x}yzu$	$x\overline{y}\,\overline{z}u$	$xy\overline{z}\,\overline{u}$	$xy\overline{z}u$
$\overline{x}\,\overline{y}\,\overline{z}$	+	+						
$\overline{x}\overline{z}\,\overline{u}$	+		+					
$\overline{y}\,\overline{z}u$		+				+		
$\overline{x}y\overline{u}$			+	+				
$y\overline{z}\,\overline{u}$			+				+	
$\overline{x}yz$				⊞	⊕			
$x\overline{z}u$						+		+
$xy\overline{z}$							+	+

У овом случају имамо само један битан прост импликант који покрива две савршене конјункције. Остале савршене конјункције морају бити покривене додатним простим импликантима које морамо изабрати из скупа преосталих простих импликаната. Формирајмо, најпре, упрошћену табелу уклањањем врсте која одговара битном простом импликанту, као и колона које он покрива:

		$\overline{x}\,\overline{y}\,\overline{z}\,\overline{u}$	$\overline{x}\,\overline{y}\,\overline{z}u$	$\overline{x}y\overline{z}\,\overline{u}$	$x\overline{y}\,\overline{z}u$	$xy\overline{z}\,\overline{u}$	$xy\overline{z}u$
p_1	$\overline{x}\,\overline{y}\,\overline{z}$	+	+				
p_2	$\overline{x}\overline{z}\,\overline{u}$	+		+			
p_3	$\overline{y}\,\overline{z}u$		+		+		
p_4	$\overline{x}y\overline{u}$			+			
p_5	$y\overline{z}\,\overline{u}$			+		+	
p_6	$x\overline{z}u$				+		+
p_7	$xy\overline{z}$					+	+

Притом, свакој врсти смо придружили једну новоуведену логичку променљиву. Формирамо КНФ формулу на основу табеле:

$$K = (p_1 + p_2) \cdot (p_1 + p_3) \cdot (p_2 + p_4 + p_5) \cdot (p_3 + p_6) \cdot (p_5 + p_7) \cdot (p_6 + p_7)$$

Сада формулу K треба превести у ДНФ:

$$\begin{aligned} K &= (p_1 + p_2) \cdot (p_1 + p_3) \cdot (p_2 + p_4 + p_5) \cdot (p_3 + p_6) \cdot (p_5 + p_7) \cdot (p_6 + p_7) \\ &= (p_1 + p_2p_3) \cdot (p_2 + p_4 + p_5) \cdot (p_3 + p_6) \cdot (p_7 + p_5p_6) \\ &= (p_1p_2 + p_1p_4 + p_1p_5 + p_2p_3) \cdot (p_3p_7 + p_6p_7 + p_5p_6) \\ &= p_1p_2p_3p_7 + p_1p_2p_6p_7 + p_1p_2p_5p_6 + p_1p_4p_3p_7 + p_1p_4p_6p_7 + p_1p_4p_5p_6 \\ &+ p_1p_5p_3p_7 + p_1p_5p_6p_7 + p_1p_5p_6 + p_2p_3p_7 + p_2p_3p_6p_7 + p_2p_3p_5p_6 \\ &= p_1p_2p_6p_7 + p_1p_4p_3p_7 + p_1p_4p_6p_7 + p_1p_5p_3p_7 + p_1p_5p_6 + p_2p_3p_7 + p_2p_3p_5p_6 \end{aligned}$$

Притом смо у последњем кораку елиминисали конјункције $p_1p_2p_3p_7$, $p_1p_2p_5p_6$, $p_1p_4p_5p_6$, $p_1p_5p_6p_7$ и $p_2p_3p_6p_7$, јер имамо конјункције $p_1p_5p_6$ и $p_2p_3p_7$ које их апсорбују. Добијене конјункције представљају минималне подскупове преосталих простих импликаната који покривају све преостале савршене конјункције. Међу њима бирамо оне са најмањим бројем елемената, а то су $p_1p_5p_6$ и $p_2p_3p_7$. Првој одговара скуп простих импликаната $\{\overline{x}\,\overline{y}\,\overline{z}, y\overline{z}\,\overline{u}, x\overline{z}u\}$, а другој $\{\overline{x}\overline{z}\,\overline{u}, \overline{y}\,\overline{z}u, xy\overline{z}\}$. Оба скупа имају укупно по 9 литерала, па је отуда свеједно који ћемо изабрати. Отуда је једна минимална ДНФ форма наше функције:

$$F(x, y, z, u) = \overline{x}yz + \overline{x}\,\overline{y}\,\overline{z} + y\overline{z}\,\overline{u} + x\overline{z}u$$

а друга:

$$F(x, y, z, u) = \overline{x}yz + \overline{x}\overline{z}\,\overline{u} + \overline{y}\,\overline{z}u + xy\overline{z}$$

У оба случаја смо скупу додатних простих импликаната придодали битни прости импликант $\overline{x}yz$.

1.4.4 Минимизација у присуству небитних вредности

Понекад се дешава да нам је логичка функција само парцијално задата, тј. да су вредности функције задате само за неке комбинације вредности на улазу. Остале вредности функције које нису задате називамо *небитне вредности* (енгл. *don't-care*), због тога што нам није битно које ће вредности функција узети за те комбинације вредности на улазима. Уколико при задавању функције имамо небитне вредности, оне се могу додефинисати на произвољан начин. Ово нам даје додатну флексибилност приликом минимизације, јер небитне вредности можемо додефинисати тако да добијени минимални ДНФ израз буде што мање сложености. У наставку овог одељка описујемо на који начин се третирају небитне вредности приликом минимизације методом Карноових мапа и методом Квин-Мекласког.

Метод карноових мапа. У случају да се минимизација врши методом Карноових мапа, тада ћемо небитне вредности третирати на следећи начин:

- небитна вредност у неком пољу мапе биће третирана као вредност 1, уколико та јединица омогућава већа заокруживања

- у супротном, небитна вредност у том пољу биће третирана као вредност 0

Другим речима, јединице се и даље морају заокружити, нуле се не смеју заокружити, а небитне вредности се могу, али не морају заокружити, па ћемо заокруживати само оне које нам омогућавају да имамо већа заокруживања и, самим тим, изразе мање сложености.

Пример 14. *Посматрајмо логичку функцију дату у табели 1.10.*

x	y	z	u	$F(x,y,z,u)$
0	0	0	0	0
0	0	0	1	–
0	0	1	0	0
0	0	1	1	1
0	1	0	0	–
0	1	0	1	1
0	1	1	0	–
0	1	1	1	–
1	0	0	0	–
1	0	0	1	–
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	–
1	1	1	0	1
1	1	1	1	–

Табела 1.10: Функција из примера 14

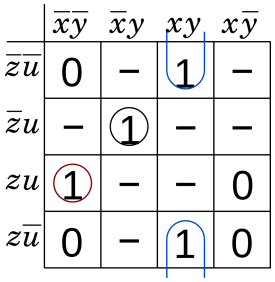
Овој функцији одговара Карноова мапа приказана на слици 1.12.

	$\bar{x}\bar{y}$	$\bar{x}y$	xy	$x\bar{y}$
$\bar{z}\bar{u}$	0	–	1	–
$\bar{z}u$	–	1	–	–
zu	1	–	–	0
$z\bar{u}$	0	–	1	0

Слика 1.12: Карноова мапа из примера 14

Уколико бисмо као и раније заокруживали само јединице (тј. када бисмо све небитне вредности третирали као нуле), имали бисмо заокруживање приказано на слици 1.13 које даје ДНФ:

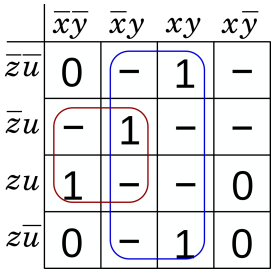
$$F(x,y,z,u) = \bar{x}\,\bar{y}zu + \bar{x}y\bar{z}u + xy\bar{u}$$



Слика 1.13: Погрешно решење примера 14

Функција коју смо добили свуда где су у табели биле небитне вредности има вредност 0. *Међутим, уколико дозволимо да се неке (погодно одабране) небитне вредности третирају као јединице, можемо добити заокруживање дато на слици 1.14 које даје ДНФ:*

$$F(x, y, z, u) = y + \bar{x}u$$



Слика 1.14: Исправно решење примера 14

Приметимо да две небитне вредности у горњем десном углу нисмо заокружили, јер није постојала могућност да се неко од заокруживања учини већим укључивањем ових поља. Због тога смо их третирали као нуле, док смо остале небитне вредности третирали као јединице.

Метод Квин-Мекласког. У случају да се минимизација врши методом Квин-Мекласког, тада се небитне вредности третирају на следећи начин:

- У првој фази алгоритма се све небитне вредности третирају као јединице, тј. учествују у груписању. Овим се омогућава да се потенцијално пронађу веће групе и да се тиме смањи сложеност израза.
- У другој фази алгоритма се небитне вредности третирају као нуле, тј. не наводе се у табели простих импликаната. Ово је зато што те улазне комбинације не морамо покрити, јер функција у њима не мора да буде једнака јединици.

Пример 15. *Посматрајмо исту функцију као у претходном примеру. У првој фази узимамо оне савршене конјункције за које функција или има вредност* 1 *или је вредност небитна. Имамо следећа груписања:*

1	$\overline{x}\,\overline{y}\,\overline{z}u\surd$	$\overline{x}\,\overline{y}u\surd$	$\overline{x}u$	y
	$\overline{x}y\overline{z}\,\overline{u}\surd$	$\overline{x}\,\overline{z}u\surd$	$\overline{z}u$	
	$x\overline{y}\,\overline{z}\,\overline{u}\surd$	$\overline{y}\,\overline{z}u\surd$	$\overline{x}y\surd$	
		$\overline{x}y\overline{z}\surd$	$y\overline{z}\surd$	
		$\overline{x}y\overline{u}\surd$	$y\overline{u}\surd$	
		$y\overline{z}\,\overline{u}\surd$	$x\overline{z}$	
		$x\overline{y}\,\overline{z}\surd$		
		$x\overline{z}\,\overline{u}\surd$		
2	$\overline{x}\,\overline{y}zu\surd$	$\overline{x}zu\surd$	$yu\surd$	
	$\overline{x}y\overline{z}u\surd$	$\overline{x}yu\surd$	$yz\surd$	
	$\overline{x}yz\overline{u}\surd$	$y\overline{z}u\surd$	$xy\surd$	
	$x\overline{y}\,\overline{z}u\surd$	$\overline{x}yz\surd$		
	$xy\overline{z}\,\overline{u}\surd$	$yz\overline{u}\surd$		
		$x\overline{z}u\surd$		
		$xy\overline{z}\surd$		
		$xy\overline{u}\surd$		
3	$\overline{x}yzu\surd$	$yzu\surd$		
	$xy\overline{z}u\surd$	$xyu\surd$		
	$xyz\overline{u}\surd$	$xyz\surd$		
4	$xyzu\surd$			

Дакле, прости импликанти су $\overline{x}u$, $\overline{z}u$, $x\overline{z}$ *и* y. *Сада табела простих импликаната изгледа овако:*

	$\overline{x}\,\overline{y}zu$	$\overline{x}y\overline{z}u$	$xy\overline{z}\,\overline{u}$	$xyz\overline{u}$
$\overline{x}u$	$\oplus$	$\boxplus$		
$\overline{z}u$		$+$		
$x\overline{z}$			$+$	
y		$\boxplus$	$\boxplus$	$\oplus$

Приметимо да се у табели разматрају само оне савршене конјункције које одговарају јединицама у табели, али не и оне које одговарају небитним вредностима. Битни прости импликанти су y *и* $\overline{x}u$. *Како они покривају све колоне табеле, минимални ДНФ је:*

$$F(x,y,z,u) = \overline{x}u + y$$

1.4.5 Минимална КНФ форма

Иако се у рачунарској техници ДНФ изрази чешће користе, понекад је потребно пронаћи минимални КНФ. Проблем проналажења минималног КНФ-а неке функције F се може једноставно свести на проблем проналажења минималног ДНФ-а њене негације (тј. функције $\overline{F}$). Наиме, приметимо да се негацијом ДНФ израза и применом Де-Морганових закона добија КНФ израз и обратно. Ово значи да уколико најпре пронађемо минимални ДНФ функције $\overline{F}$, а затим га негирамо, добићемо минимални КНФ функције F, што је и требало пронаћи. Отуда се било која од

раније описаних метода за минимизацију ДНФ израза може користити и за минимизацију КНФ израза, уз претходно инвертовање таблице функције, као и негацију добијеног ДНФ израза на крају.

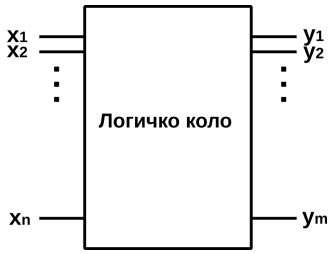
Глава 2

Логичка кола

Логичко коло (енгл. *logic circuit*) је уређај који имплементира неки скуп логичких функција у датој технологији. У овој глави бавимо се пре свега елементарним логичким колима која имплементирају унарне и бинарне логичке везнике. Ова кола представљају основне градивне елементе у конструкцији сложенијих кола, којима се бавимо у наредне две главе.

2.1 О логичким колима

Логичко коло у општем случају има n улаза $(x_1, x_2, \ldots, x_n)$ и m излаза $(y_1, y_2, \ldots, y_m)$ (слика 2.1). Сваки од улаза и излаза представља логичку вредност, тј. може имати вредност 0 или 1. Притом, вредности на излазима зависе од вредности на улазима, тј. могу се изразити као логичке функције од улаза кола.



Слика 2.1: Шематски приказ општег логичког кола

Вредности улаза (па самим тим и вредности излаза) се мењају током времена, па можемо сматрати да је сваки улаз x_i (односно излаз y_j) функција од времена[1], тј. $x_i = x_i(t)$ $(y_j = y_j(t))$. Ово најчешће нећемо посебно наглашавати, када говоримо о вредностима улаза и излаза у

[1]Притом, време може бити посматрано као континуална (непрекидна) или као дискретна величина. У овом другом случају време се посматра као низ дискретних временских тренутака $t_0, t_1, \ldots$. Ова дискретна интерпретација времена је много чешћа, имајући у виду да се свако израчунавање у рачунару изражава низом елементарних корака који се извршавају у дискретним временским тренутцима. Континуална интерпретација времена се користи само када анализирамо понашање кола током прелаза

неком фиксираном временском тренутку. Са друге стране, када говоримо о вредностима улаза и излаза у различитим временским тренутцима, то ће бити посебно наглашено (нпр. „вредност улаза x_i у тренутку t_0", „вредност излаза y_j у тренутку t_1" и сл.).

Уколико вредности на излазу неког логичког кола у неком временском тренутку зависе само од вредности улаза у том истом тренутку, тада такво логичко коло зовемо *комбинаторно коло*. Дизајн комбинаторних кола је са теоријске тачке гледишта једноставан, јер се своди на одређивање и минимизацију логичких израза који израчунавају логичке функције на излазима кола у зависности од тренутних вредности улаза. У пракси, овај поступак ипак није увек тако једноставан, због комбинаторне експлозије која настаје код кола која имају велики број улаза и излаза. Дизајном комбинаторних кола опширније се бавимо у глави 3. Уколико, са друге стране, вредности излаза у неком тренутку зависе не само од вредности улаза у том тренутку, већ и од вредности улаза у претходним временским тренутцима, тада такво коло називамо *секвенцијално коло*. Дизајн секвенцијалних кола је суштински тежи од дизајна комбинаторних кола, зато што је потребно разматрати и оне вредности улаза које више нису присутне, већ су постојале у неким претходним временским тренутцима. Дизајном секвенцијалних кола опширније се бавимо у глави 4.

2.2 Вредност високе импедансе

Сваки од излаза логичког кола, поред тога што у неком тренутку може имати вредност 0 или 1, може и да нема никакву вредност. Другим речима, логичко коло може да, просто, у неком тренутку „искључи" неки од излаза и да на њему не производи ни нулу ни јединицу. Иако овакво понашање није дефинисано у оквиру алгебре логике, омогућено је из практичних разлога. Наиме, претпоставимо ситуацију у којој се излази y' кола C' и y'' кола C'' оба повезују на исти улаз x неког кола C (слика 2.2). Уколико излази y' и y'' имају различите вредности (нпр. $y' = 0$ и $y'' = 1$), тада ће вредност на улазу x бити недефинисана.[2][3] Оваква нежељена ситуација се може спречити тако што обезбедимо да у сваком тренутку *највише један* од излаза који су повезани на исти улаз x има вредност 0 или 1, док су остали излази искључени.

Да бисмо формално дефинисали понашање „искључених" излаза, уводимо тзв. *вредност високе импедансе*[4]. Ову вредност означаваћемо са **Z** и сматраћемо да је имају сви излази који су искључени, тј. који не производе никакву вредност. Уколико на неком излазу имамо вредност **Z**,

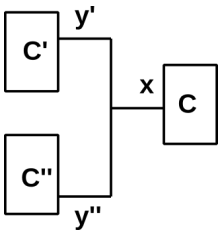
---

из једног у друго стање: ти прелази се у конкретним технологијама не могу обавити тренутно, па је за боље разумевање понашања потребно анализирати и оно што се дешава између два стабилна стања. У наставку овог текста, осим ако није другачије наглашено, увек ћемо сматрати да је време дискретна величина.

[2]За вредност на улазу/излазу кола кажемо да је *недефинисана* уколико је није могуће једнозначно одредити. Оваква појава је увек знак лошег дизајна кола или неке грешке у повезивању кола.

[3]У уобичајеној електронској технологији, не само да ћемо имати логички недефинисану вредност, већ ћемо имати и „кратак спој", што може довести до квара.

[4]Назив потиче из савремене електронске технологије, где се вредност **Z** реализује прекидом везе са извором напајања, тј. имамо „прекинуту жицу" која, отуда, има бесконачно велики отпор (импедансу).



Слика 2.2: Потенцијална колизија сигнала y' и y'' који су повезани на исти улаз x

тада тај излаз не утиче на вредност улаза на који је повезан. На пример, ако доведемо два излаза y' и y'' на исти улаз x, и уколико је, рецимо, $y' = \mathbf{Z}$, тада ће вредност на улазу x бити иста као и вредност излаза y''. Са друге стране, ако су и y' и y'' једнаки $\mathbf{Z}$, тада ће и на улазу x бити вредност $\mathbf{Z}$.[5]

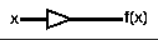
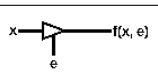
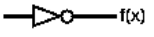
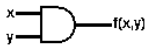
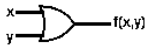
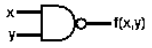
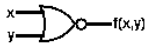
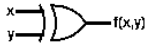
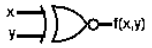
2.3 Логичке капије

У глави 1 смо видели да се произвољна логичка функција може представити логичким изразом који је изграђен над улазним променљивама користећи неки потпуни скуп логичких везника. Отуда следи да ћемо и произвољно логичко коло моћи да имплементирамо уколико на располагању имамо логичка кола која имплементирају логичке везнике из изабраног скупа. Оваква елементарна логичка кола називамо *логичке капије* или *гејтови* (енгл. *logic gates*). Табела 2.1 приказује основне гејтове и њихове шематске ознаке.

У табели 2.1, поред кола која реализују уобичајене логичке везнике, имамо и два додатна кола: *бафер* и *бафер са три стања*. Бафер је коло које реализује идентичку функцију $f(x) = x$ и са логичке тачке гледишта нема никакву функцију. Разлог за постојање оваквог кола је техничке природе, јер се њиме омогућава „појачавање" вредности сигнала приликом преноса између различитих делова система (више о томе у наредном поглављу). Бафер са три стања додатно има и контролни улаз e: уколико је $e = 1$, понашање је исто као и код обичног бафера, док у случају када је $e = 0$, тада је на излазу вредност високе импедансе. Другим речима, бафер са три стања се понаша као прекидач који у зависности од контролног улаза пропушта или не пропушта улаз на излаз.

Понашање логичких капија у случају да неки од улаза има вредност високе импедансе зависи од типа логичке капије. На пример, у случају И кола, довољно је да један од улаза има вредност 0, излаз ће такође бити 0, чак и да је на другом улазу вредност $\mathbf{Z}$. Са друге стране, уколико је, на пример, $x = 1$, а $y = \mathbf{Z}$, тада ће излаз имати недефинисану вредност. Понашање ИЛИ кола је дуално: уколико је бар један од улаза јединица,

[5] Неформално, вредност $\mathbf{Z}$ можемо разумети и као одсуство вредности. Уколико су сви излази који су повезани на неки улаз x искључени, тада до тог улаза неће долазити никаква вредност, па ћемо и на улазу x имати вредност $\mathbf{Z}$. Сличну ситуацију имаћемо и ако на улаз x не повежемо ни један излаз (ово такође логички на први поглед нема много смисла, али је у пракси могуће).

Назив кола	**Функција кола**	**Шематска ознака**
Бафер (енгл. *buffer*)	$f(x) = x$	
Бафер са три стања (енгл. *three-state buffer*)	$f(x,e) = \begin{cases} x, & \text{за } e = 1 \\ \mathbf{Z}, & \text{за } e = 0 \end{cases}$	
НЕ коло (енгл. *NOT*)	$f(x) = \overline{x}$	
И коло (енгл. *AND*)	$f(x,y) = x \cdot y$	
ИЛИ коло (енгл. *OR*)	$f(x,y) = x + y$	
НИ коло (енгл. *NAND*)	$f(x,y) = x \uparrow y$	
НИЛИ коло (енгл. *NOR*)	$f(x,y) = x \downarrow y$	
ЕИЛИ коло (енгл. *XOR*)	$f(x,y) = x \oplus y$	
НЕИЛИ коло (енгл. *XNOR*)	$f(x,y) = x \sim y$	

Табела 2.1: Логичке капије

тада ће и на излазу бити јединица, чак и да је други улаз једнак **Z**. Са друге стране, комбинација $x = 0, y = \mathbf{Z}$ ће дати недефинисану вредност на излазу. Кола НИ и НИЛИ се понашају слично колима И и ИЛИ, уз додатну негацију. Сва остала кола ће увек имати недефинисану вредност на излазу кад год је неки од улаза једнак **Z**.

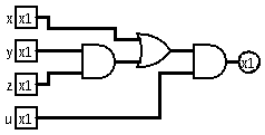
Подсетимо се да се бинарни логички везници могу уопштити и посматрати као n-арни везници (одељак 1.3.3). Отуда уместо двоулазних можемо разматрати и вишеулазне логичке капије које имплементирају семантику одговарајућих n-арних везника.

2.4 Кашњење логичког кола

Под *кашњењем* логичког кола (енгл. *propagation delay*) подразумевамо време које је потребно да се након промене вредности на улазима кола излази стабилизују на новим вредностима. Ово кашњење зависи од структуре логичког кола које имплементира сваку од излазних функција (дакле, кашњење не мора бити једнако за све излазе логичког кола). У основи, свака логичка капија има своје кашњење које зависи од технологије израде и у данашње време се обично мери у наносекундама или пикосекундама. Ово кашњење називамо *кашњење капије* (енгл. *gate delay*). Кашњење капије не мора бити исто за све типове логичких капија. На пример, у CMOS технологији (поглавље 2.5) И и ИЛИ кола типично имају веће кашњење од НИ и НИЛИ кола (супротно интуицији). Такође, кашњење капије може да буде другачије када је у питању прелаз са 0 на 1 на излазу, у односу на прелаз са 1 на 0. Кашњење такође може зависити од кола које је повезано на излаз капије (нпр. ако на исти излаз

повежемо улазе више различитих капија, тада ће тај излаз бити више оптерећен и биће потребно више времена да он успостави своју вредност). Најзад, кашњење вишеулазних капија може бити знатно веће у односу на кашњење двоулазних капија. Због свега овога, анализа кашњења није ни мало једноставна и у великој мери је зависна од технологије која се користи. Због једноставности анализе, ми ћемо у даљем тексту претпостављати да све логичке капије имају исто кашњење које ћемо обично означавати са Δ (изузетак су вишеулазне капије, о чему ћемо детаљније дискутовати у одељку 2.5.10).

Уколико се сада две логичке капије надовежу једна на другу (тј. излаз прве се повеже на улаз друге), тада ће укупно кашњење тако повезаних кола бити једнако збиру кашњења појединачних капија. Уопште, уколико се логичко коло конструише према неком логичком изразу, тада ће кашњење одговарајућег логичког кола у најгорем случају бити пропорционално дубини[6] тог израза. На пример, посматрајмо коло на слици 2.3.



Слика 2.3: Логичко коло дубине 3

Ако претпоставимо да двоулазна конјункција и дисјункција имају исто кашњење Δ, тада ће кашњење овог кола у најгорем случају бити 3Δ јер је дубина израза $(x + y \cdot z) \cdot u$ који ово коло реализује једнака 3.[7]

На крају напоменимо да и жице којима се логичка кола повезују такође имају своје кашњење. Наиме, савремени рачунари су по правилу електронски, што значи да се заснивају на преношењу електричних сигнала кроз проводнике. Ако узмемо да се струја кроз проводник креће брзином светлости (300000km/s, што је теоријски максимум), добијамо да у свакој наносекунди сигнал може прећи највише 30cm. У пракси се електрични сигнал кроз проводник креће брзином и до два пута мањом од брзине светлости, па отуда кашњење на 30cm жице може бити и до 2ns. Због тога је за повећање брзине логичких кола, поред побољшавања технологије израде самих логичких капија, потребно смањивати растојања међу њима, а то се постиже тако што се на веома малој површини смешта велики број логичких капија. Зато је повећање степена интеграције[8] у последњих неколико деценија довело до драматичног убрзавања рада савремених рачунара.

[6] Под дубином израза подразумевамо висину синтаксног стабла тог израза.

[7] Приметимо да кашњење може бити и мање, у зависности од тога које вредности на улазу су промењене. На пример, ако променимо само вредност улаза u, тада та промена треба да прође само кроз последње И коло, што даје кашњење 1Δ. У случају да је промењен само улаз x, кашњење ће бити 2Δ, док ће у случају промене улаза y или z кашњење бити 3Δ, јер тада промена мора да прође кроз све три логичке капије.

[8] Степен интеграције је одређен бројем прекидача (транзистора) које је могуће сместити на јединицу површине интегрисаног кола (чипа).

2.5 Имплементација логичких капија у савременим рачунарима

Савремени рачунари се другачије зову и електронски рачунари због тога што се логичка кола у њима реализују помоћу електронских компоненти. Уређај је прикључен на извор електричне струје, а логичке вредности се представљају одређеним напонским нивоима. Претпоставимо, на пример, да је напон батерије која напаја уређај једнак $5V$. Ово значи да се потенцијали позитивног и негативног пола батерије разликују за $5V$. Обично се претпоставља да је негативан пол батерије на потенцијалу $0V$ и да је то референтна тачка у односу на коју одређујемо напонске нивое осталих тачака у колу (ова тачка се обично назива и *маса*, енгл. *ground (GND)*). Отуда је потенцијал позитивног пола батерије једнак $+5V$ (ову тачку називамо и *напајање*, енгл. *supply*). Вредности потенцијала које су блиске нули (нпр. мање од $+1.5V$) сматрају се логичком нулом, док се вредности потенцијала блиске $+5V$ (нпр. веће од $+3.5V$) сматрају логичком јединицом. Напонски нивои који се налазе између (нпр. у интервалу од $1.5V$ до $3.5V$) не представљају валидну логичку вредност. Због тога се сва кола морају дизајнирати тако да у стабилном стању ни једна тачка у колу није на потенцијалу из овог средишњег напонског опсега.[9] Са друге стране, у току транзиције између два стабилна стања (која се не може тренутно остварити, већ захтева извесно време), напон пролази и кроз овај опсег вредности. На пример, приликом транзиције из логичке нуле у логичку јединицу, напон се увећава од вредности која је блиска нули до вредности која је блиска $+5V$, пролазећи притом кроз све вредности напона које се налазе између.

Имајући наведено у виду, да бисмо обезбедили да на излазу неког логичког кола буде логичка нула, потребно је повезати тај излаз са масом, док ако желимо да на излазу имамо логичку јединицу, треба га повезати са напајањем. Ово се остварује помоћу *прекидача* којима се одговарајуће електричне везе успостављају и прекидају. Ови прекидачи су контролисани напонима који долазе са излаза других логичких кола, чиме се успоставља зависност једних логичких променљивих од других.

Различите генерације електронских рачунара се разликују управо по технологији израде прекидача. Први електронски рачунари су били засновани на *вакуумским цевима*, али се од средине педесетих година 20. века ова технологија замењује *полупроводничком технологијом* која се затим кроз деценије усавршавала и у одређеној форми је и данас у употреби. Зато ћемо се у наставку текста бавити искључиво полупроводничком технологијом.

Полупроводничка технологија је заснована на хемијском елементу *силицијуму*. Овај елемент, након што се на њега примене одговарајући технолошки поступци, добија занимљиве електричне особине на којима је заснован рад различитих полупроводничких компоненти међу којима је свакако најзначајнији *транзистор*. На транзисторима је, између осталог, заснован и рад савремених рачунара, с обзиром да у одређеном режиму рада

[9] У случају лошег дизајна, може се догодити да се потенцијал неке тачке стабилизује или у дужем временском периоду задржи на нивоу који не представља валидну логичку вредност. Оваква појава се назива *метастабилност* и сматра се нежељеном последицом лошег дизајна.

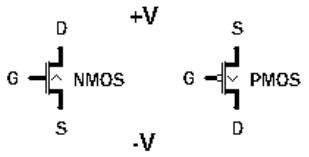
транзистор може функционисати као прекидач. Постоје две основне врсте транзистора: *биполарни* и *униполарни*. Савремени рачунари су углавном засновани на једној посебној врсти униполарних транзистора – на *MOS транзисторима*.[10]

MOS транзистори представљају полупроводничку компоненту која има три прикључка: *сорс*, *дрејн* и *гејт*. По свом понашању, MOS транзистор функционише као прекидач – струја може протицати од сорса ка дрејну под условом да се одговарајући напон доведе на гејт. Кроз сам гејт не протиче струја, јер је он изолатором одвојен од остатка транзистора. Његова улога је да својим електричним потенцијалом створи одговарајуће електрично поље које ће у самом транзистору привући носиоце наелектрисања и тако обезбедити проток струје од сорса ка дрејну. Постоје два типа MOS транзистора: NMOS и PMOS транзистор.

Код NMOS транзистора сорс мора бити прикључен на негативан, а дрејн на позитиван напон. Да би дошло до провођења струје, потребно је на гејт довести довољно велики позитиван напон (у односу на сорс). Транзистор у потпуности проводи када је напон на гејту у зони логичке јединице (типично изнад 2/3 напона напајања). Уколико је напон у зони логичке нуле (испод 1/3 напона напајања), тада је транзистор у потпуности затворен и не проводи струју од сорса кад дрејну.

Са друге стране, код PMOS транзистора, сорс се прикључује на позитиван, а дрејн на негативан напон. Да би струја потекла од сорса ка дрејну, потребно је на гејт довести негативан напон (у односу на напон сорса). Дакле, PMOS транзистор потпуно проводи када је напон на гејту у зони логичке нуле, док је потпуно затворен и не проводи струју када је напон на гејту у зони логичке јединице.

На слици 2.4 приказане су типичне ознаке NMOS и PMOS транзистора. Стрелица означава смер од сорса ка дрејну. Средишњи прикључак је гејт. Ознака PMOS транзистора садржи кружић на гејту и по томе се разликује од ознаке NMOS транзистора.

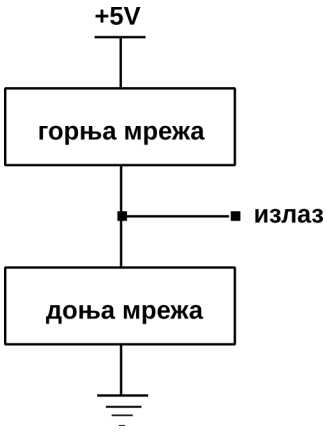


Слика 2.4: Ознаке NMOS и PMOS транзистора

Типична структура логичке капије у MOS технологији дата је на слици 2.5. *Горња мрежа* (енгл. *pullup network*) се састоји из елемената који успостављају везу излаза са напајањем када је потребно да на излазу буде логичка јединица. *Доња мрежа* (енгл. *pulldown network*) се састоји из елемената који успостављају везу са масом када је потребно да на

[10]Енглеска скраћеница MOS је од *Metal-Oxide-Semiconductor*, тј. *метал-оксид-полупроводник*, а потиче од структуре самог транзистора – испод металне електроде налази се слој силицијум-оксида који представља изолатор, а испод њега се налази силицијумски полупроводнички слој.

излазу имамо логичку нулу. Горња мрежа може садржати искључиво PMOS транзисторе, с обзиром да њихов сорс мора бити повезан на напајање. Слично, доња мрежа може садржати само NMOS трензисторе, јер њихов сорс мора бити повезан на масу.

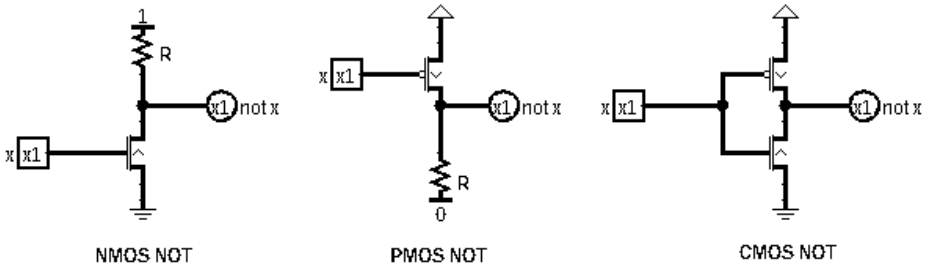


Слика 2.5: Општа структура логичке капије

Уколико се у изради логичких кола користе само NMOS (PMOS) транзистори, тада ту варијанту MOS технологије називамо NMOS (PMOS) технологија. Са друге стране, уколико се користе и NMOS и PMOS транзистори, тада говоримо о CMOS (енгл. *Complementary MOS*) технологији. CMOS технологија је данас готово искључиво у употреби, због значајне уштеде струје и смањеног загревања. Међутим, CMOS је технолошки најзахтевнији, јер је потребно обезбедити да на истом чипу коегзистирају и PMOS и NMOS транзистори.

2.5.1 НЕ коло

Разлику између NMOS, PMOS и CMOS технологије илустроваћемо на примеру НЕ кола на слици 2.6.



Слика 2.6: Реализација НЕ кола у NMOS, PMOS и CMOS технологији

Код NMOS имплементације (лева слика) у доњој мрежи се налази један NMOS транзистор чија је улога да отвара и затвара везу излаза са масом.

С обзиром да нам у NMOS технологији нису доступни PMOS транзистори (а NMOS транзистори не могу да стоје у горњој мрежи), горња мрежа се састоји из једног *отпорника* чија је улога да успостави слабу везу са напајањем. Уколико је NMOS транзистор у доњој грани затворен (када је на улазу логичка нула), тада ова слаба веза омогућава пренос потенцијала напајања на излаз, па имамо логичку јединицу на излазу. Када је доња грана отворена (логичка јединица на улазу), тада се успоставља директна (јака) веза излаза са масом, па је на излазу низак потенцијал (логичка нула).[11] Отуда, ово коло функционише као инвертер (НЕ коло).

Ситуација је аналогна код PMOS имплементације (средње коло на слици), с тим што је сада PMOS транзистор у горњој мрежи, а у доњој мрежи имамо отпорник који успоставља слабу везу са масом. Логичка нула на улазу отвара транзистор и повезује излаз са напајањем (логичка јединица). Логичка јединица на улазу затвара транзистор и прекида проток струје, па се преко отпорника на излаз доводи потенцијал масе (логичка нула).

Проблем са NMOS и PMOS технологијом је у томе што оваква кола у једном од два стабилна стања континуирано проводе струју (NMOS у случају јединице на улазу, а PMOS у случају нуле на улазу) што доводи до загревања отпорника. Повећано загревање и повећана потрошња струје постају веома значајан фактор када имамо велики број транзистора на чипу. Због тога се у неком тренутку морало прећи на скупљу CMOS технологију која ове проблеме нема.

CMOS имплементација НЕ кола је дата на трећој слици. Како су нам сада доступни и NMOS и PMOS транзистори, отпорници више нису потребни. У случају логичке нуле на улазу доњи транзистор је затворен, а горњи отворен, па имамо везу излаза са напајањем (логичка јединица). У случају логичке јединице на улазу доњи транзистор је отворен, а горњи затворен, па имамо везу излаза са масом (логичка нула). У сваком тренутку отворен је тачно један од транзистора, док је други затворен, па нема протицања струје ни у једном стабилном стању. Струја протиче само у периодима транзиције између стања док се не успостави стабилан потенцијал на излазу кола. Због тога је код CMOS технологије потрошња струје и количина ослобођене топлоте драстично мања.[12]

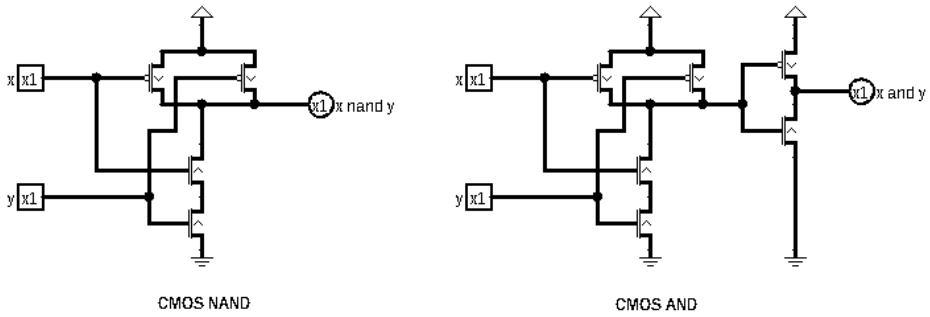
---

[11] Овај феномен се може прецизније описати ако се сетимо *Омовог закона*: $U = I \cdot R$. Дакле, напон (разлика потенцијала) између тачака на крајевима отпорника је једнак производу отпорности отпорника и јачине струје која кроз отпорник протиче. Када је NMOS транзистор затворен и струја не протиче кроз отпорник, напон на отпорнику је $U = 0 \cdot R = 0V$, па је потенцијал доњег краја отпорника (тј. потенцијал излаза кола) једнак потенцијалу горњег краја отпорника, а то је потенцијал напајања. Отуда, на излазу имамо логичку јединицу. Када транзистор проводи струју, та струја креира напон између крајева отпорника. Укупан отпор у колу је једнак $R + R_t$, где је R_t отпорност транзистора када је потпуно отворен (редна веза). Јачина струје која протиче кроз отпорник ће бити једнака $I = U_d/(R + R_t)$, где је U_d напон напајања, Сада је напон између крајева отпорника једнак $U_R = I \cdot R = U_d \cdot (R/(R + R_t))$. Како је отпорност потпуно отвореног транзистора R_t занемарљива у односу на R, следи да је $U_R \approx U_d$, па је потенцијал доње тачке отпорника $U_d - U_R$ близак нули. Отуда на излазу имамо логичку нулу.

[12] Заправо, потрошња струје и ослобођена топлота је пропорционална броју транзиција у секунди. У модерним рачунарима, овај број је одређен фреквенцијом часовника. Отуда, што је већи такт процесора, то се он више загрева и троши више струје.

2.5.2 НИ и И коло

Лево коло на слици 2.7 представља CMOS имплементацију НИ кола. У доњој мрежи налази се редна веза два NMOS транзистора, док се у горњој мрежи налази паралелна веза два PMOS транзистора. Када су на улазима кола две логичке јединице, оба доња транзистора проводе, па проводи и њихова редна веза, те имамо везу излаза са масом. Са друге стране, оба горња транзистора не проводе, па не проводи ни њихова паралелна веза, те немамо везу излаза са напајањем. Отуда је на излазу логичка нула. У свим другим комбинацијама на улазу бар један од доњих транзистора неће проводити, а бар један од горњих транзистора ће проводити, па ћемо имати везу излаза са напајањем, али не и са масом. Отуда ћемо на излазу имати логичку јединицу. Ово је управо НИ функција.



Слика 2.7: НИ и И коло у CMOS-у

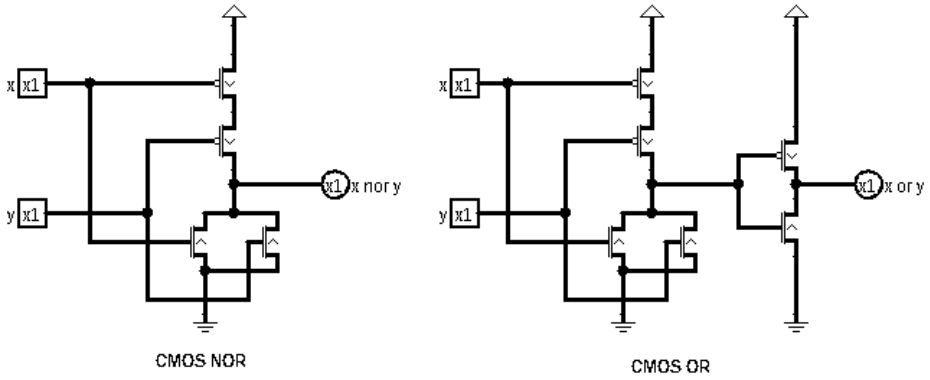
Десна слика приказује И коло у CMOS технологији. Оно се добија надовезивањем једне негације (НЕ кола) на излаз НИ кола. Дакле, иако ми интуитивно НИ доживљавамо као „негирано И", у CMOS технологији је заправо једноставније направити НИ коло, док се И коло представља као „негирано НИ".

2.5.3 НИЛИ и ИЛИ коло

Слика 2.8 приказује CMOS имплементацију ИЛИ и НИЛИ кола. Имплементација НИЛИ кола је аналогна имплементацији НИ кола, с том разликом што су NMOS транзистори у паралелној вези, а PMOS транзистори у редној вези. Отуда ће сада јединица бити на излазу само ако су две нуле на улазу, што одговара функцији НИЛИ кока. ИЛИ коло се поново добија негацијом НИЛИ кола.

2.5.4 ЕИЛИ коло

Једна могућа имплементација ЕИЛИ кола дата је на слици 2.9. У доњем делу слике видимо две негације: једна производи $\overline{x}$, а друга $\overline{y}$. У горњем делу слике видимо коло чија горња мрежа садржи четири PMOS транзистора у комбинованој редно-паралелној вези, а доња мрежа садржи четири NMOS транзистора који су повезани на исти начин.Транзистори у левој грани



Слика 2.8: НИЛИ и ИЛИ коло у CMOS-у

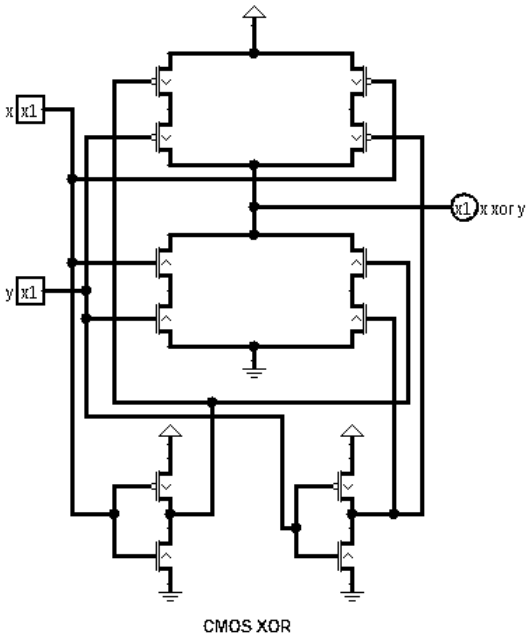
горње мреже повезани су на y и $\overline{x}$, док су транзистори у десној грани горње мреже повезани на x и $\overline{y}$. Отуда горња мрежа проводи струју ако је или $x = 1, y = 0$ (лева грана) или $x = 0, y = 1$ (десна грана). У доњој мрежи ситуација је комплементарна: транзистори леве гране су повезани на x и y, док су транзистори десне гране повезани на $\overline{x}$ и $\overline{y}$. Отуда доња мрежа проводи када је $x = 0, y = 0$ (десна грана) или када је $x = 1, y = 1$ (лева грана). Дакле, горња мрежа проводи када су улази x и y различити, а доња мрежа када су једнаки. Ово значи да ћемо на излазу имати логичку јединицу акко улази x и y имају различите вредности, док ћемо у супротном имати логичку нулу, што одговара функцији ЕИЛИ кола.

2.5.5 Бафер

Бафер се обично имплементира као две надовезане негације (слика 2.10). Овим се логички не постиже ништа. Међутим, бафер има улогу појачавача снаге сигнала који се преноси кроз коло. Наиме, услед отпорности компоненти кроз које електрични сигнал пролази може доћи до значајног пада напона, што понекад може довести до тога да напон који стигне до жељене тачке у колу више не буде у зони одговарајуће логичке вредности. До сличне појаве може доћи и у ситуацијама када је потребно излаз датог кола повезати на више улаза других кола, или на улаз неког већег електричног потрошача, при чему излазна снага кола није довољно велика. У таквим ситуацијама се могу додати бафери који, захваљујући томе што имају сопствено напајање, појачавају ослабљени сигнал и прослеђују га даље.[13]

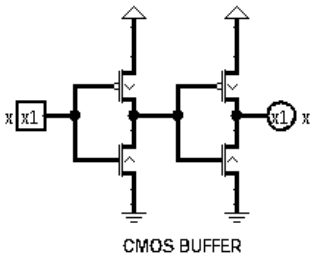
Приметимо да постојање бафера, иако нема логичког ефекта, уводи додатно кашњење. Ово кашњење је резултат тога што је потребно извесно време да се вредност са улаза бафера пропагира на излаз (тј. да прође кроз две негације). Иако се кашњење углавном посматра као негативна појава, има ситуација када је увођење додатног кашњења пожељно. На пример, понекад је потребно обезбедити да неки сигнал стигне на своју

[13] Уобичајено је да се друга негација у баферу реализује помоћу већих транзистора који могу да обезбеде већу излазну снагу.



Слика 2.9: ЕИЛИ КОЛО У CMOS-у

дестинацију пре другог сигнала. Један од начина да се тај други сигнал „успори" приликом пропагације кроз коло је да се пропусти кроз додатне бафере.

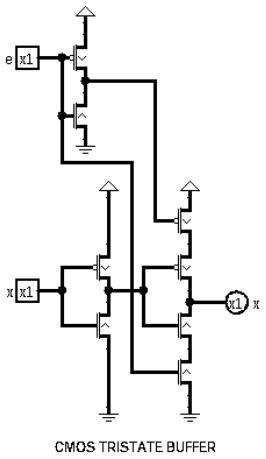


Слика 2.10: Бафер у CMOS-у

2.5.6 Бафер са три стања

Пример реализације бафера са три стања у CMOS технологији дат је на слици 2.11. У десном делу шеме имамо четири транзистора, при чему унутрашња два транзистора чине класичан инвертер, док спољашња два контролишу доток напајања у инвертер. Ако је $e = 0$, тада су горњи PMOS и доњи NMOS транзистор искључени, те нема везе унутрашњих транзистора ни са масом ни са напајањем, па на излазу имамо **Z**. Ако је $e = 1$, тада

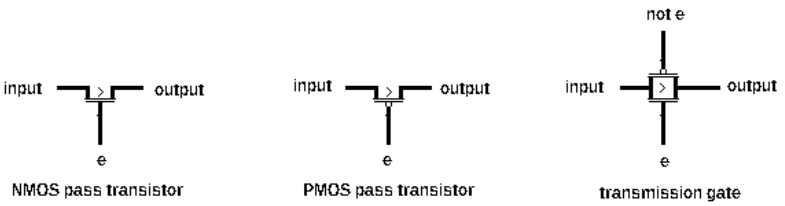
су оба спољашња транзистора потпуно отворена, те се унутрашњи пар транзистора понаша као инвертер и даје на излазу x.



Слика 2.11: Бафер са три стања у CMOS-у

2.5.7 Пропусни транзистори и преносне капије

Понекад је потребно контролисано пропуштање неког сигнала од једне тачке кола ка другој у зависности од вредности неког другог сигнала. Типична примена је у реализацији бафера са три стања, али има и других ситуација где је оваква функционалност потребна. Најједноставнији начин да се ово постигне је коришћење тзв. *пропусног транзистора* (енгл. *pass transistor*).



Слика 2.12: Пропусни транзистори и преносне капије

NMOS пропусни транзистор је приказан у левом делу слике 2.12. Сигнал који се пропушта се доводи на сорс транзистора, а излаз се налази на дрејну. Контролни сигнал e се налази на гејту транзистора. Транзистор се отвара када је на e улазу логичка јединица и вредност са сорса се преноси ка дрејну. Уколико се на улазу e налази логичка нула, тада је транзистор затворен и на излазу (дрејну) немамо никакву вредност (односно, имамо **Z**).

Проблем са оваквим пропусним транзистором је у томе што он, будући да је у питању NMOS транзистор, добро проводи само ако је на сорсу

низак напон (тј. логичка нула), а на гејту логичка јединица. То значи да ће логичка нула са улаза (сорса) бити добро пропуштена према излазу (дрејну транзистора). Са друге стране, логичка јединица са улаза неће бити добро пропуштена ка излазу, јер су тада и сорс и гејт на високом потенцијалу, па не постоји довољно велика разлика потенцијала између гејта и сорса да би се транзистор потпуно отворио. Уколико би уместо NMOS транзистора ту стајао PMOS транзистор (средишња шема на слици 2.12), тада би ситуација била обрнута: логичка јединица са сорса би била добро пропуштана ка дрејну (када је $e = 0$), док би логичка нула била слабо пропуштена ка излазу. Овај недостатак пропусних транзистора се решава CMOS варијантом пропусних транзистора – *преносном капијом* (енгл. *transmission gate*).

Преносна капија (на десној страни слике 2.12) се састоји из пара комплементарних транзистора чији су сорсови повезани у једну тачку (улаз капије) а дрејнови у другу тачку (излаз капије). На гејт NMOS транзистора се доводи контролни сигнал e, док се на гејт PMOS транзистора доводи негација контролног улаза $\overline{e}$. Уколико је $e = 0$, тада не проводи ни један транзистор, па на излаз не пролази ништа (имамо **Z**). Уколико је $e = 1$, тада оба транзистора проводе. Притом, NMOS транзистор добро пропушта логичку нулу, док PMOS транзистор добро пропушта логичку јединицу. Недостатак пропусне капије је у томе што је потребно да на располагању имамо и негацију контролног сигнала $\overline{e}$ – у супротном, неопходно је увести још једно НЕ коло које ће да инвертује контролни улаз.

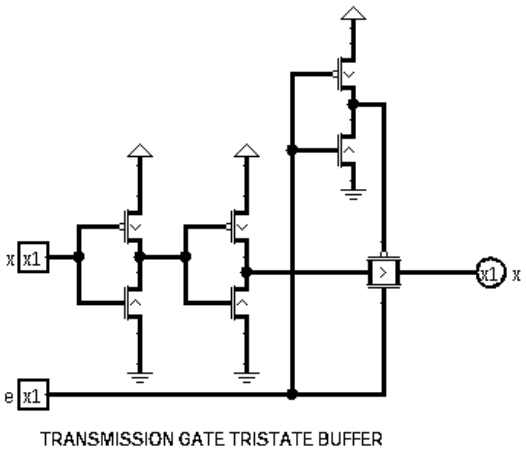
2.5.8 Бафер са три стања и преносне капије

Типична примена преносних капија је у ефикасној реализацији бафера са три стања. Да бисмо направили бафер са три стања, потребно је на излазу обичног бафера поставити преносну капију која ће бити контролисана додатним контролним сигналом e (слика 2.13). Да бисмо произвели негацију контролног улаза e, потребно је додати још једно НЕ коло (у горњем десном углу шеме).

Напоменимо да се, у случају да не постоји потреба за појачавањем снаге сигнала, инвертери у левом делу шеме могу изоставити, па се бафер са три стања може свести на једну преносну капију и један инвертер који производи негацију сигнала e. Уколико нам је негација контролног сигнала e већ доступна, тада једна преносна капија може сама играти улогу бафера са три стања. Оваква реализација бафера са три стања је најјефтинија, јер садржи само два транзистора.

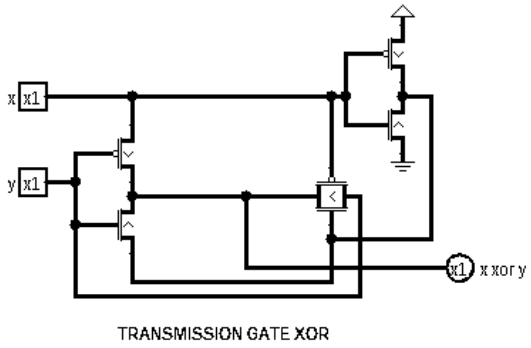
2.5.9 ЕИЛИ коло и преносне капије

Једна другачија, мање интуитивна, али једноставнија имплементација ЕИЛИ кола помоћу преносне капије дата је на слици 2.14. Основна идеја ове имплементације је да се ЕИЛИ функција декомпонује на две функције: ако је $x = 0$ тада је $x \oplus y = y$, па на излаз треба пропустити y; ако је $x = 1$, тада је $x \oplus y = \overline{y}$, па на излаз треба пропустити инвертовано y. У горњем десном углу шеме налази се једно НЕ коло који производи $\overline{x}$ на свом излазу. На левој страни шеме се налазе два комплементарна транзистора који функционишу као негација улаза y, али само ако је на x улазу јединица



Слика 2.13: Бафер са три стања реализован помоћу преносне капије

(тада је сорс горњег PMOS транзистора повезан на висок потенцијал, док је сорс доњег NMOS транзистора повезан на низак потенцијал, па цео склоп функционише као НЕ коло које производи $\overline{y}$). У случају да је на улазу x логичка нула, тада ова два транзистора не раде и на излазу тог дела кола се не производи ништа. Преостала два транзистора су повезани као преносна капија која пропушта вредност y са десна на лево (ка излазу) само ако је на улазу x логичка нула. Ово се постиже тако што се на гејт PMOS транзистора доводи x, а на гејт NMOS транзостора доводи $\overline{x}$.

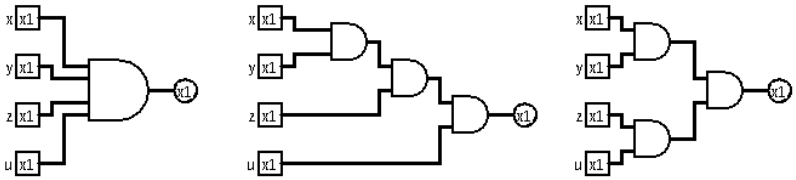


Слика 2.14: ЕИЛИ коло реализовано помоћу преносне капије

2.5.10 Вишеулазне логичке капије

Вишеулазне логичке капије И, ИЛИ и ЕИЛИ се могу имплементирати као композиције одговарајућих двоулазних логичких капија, имајући у виду да се одговарајући n-арни везници по дефиницији своде на бинарне везнике истог типа, груписањем подизраза (одељак 1.3.3). Груписање се може

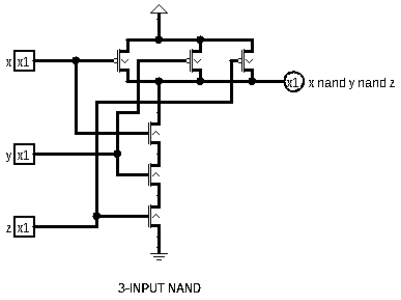
извршити на произвољан начин, имајући у виду закон асоцијативности који важи за бинарне везнике конјункције, дисјункције и ексклузивне дисјункције. На пример, израз $x \cdot y \cdot z$ је еквивалентан изразу $(x \cdot y) \cdot z$, па се ова конјункција три вредности увек може реализовати надовезивањем две двоулазне И капије. Међутим, оваква реализација повећава кашњење кола, па је потребно груписање извршити тако да добијени израз има што мању дубину. На пример, четвороулазна конјункција $xyzu$ се може посматрати као $(((x \cdot y) \cdot z) \cdot u)$, али и као $(x \cdot y) \cdot (z \cdot u)$. Овим изразима одговарају реализације помоћу двоулазних И кола прилазане на слици 2.15.



Слика 2.15: Четвороулазно И коло и две његове различите реализације помоћу двоулазних И кола

Лева слика представља четвороулазну логичку конјункцију, док преостале две одговарају наведеним реализацијама уз помоћ двоулазних И кола. Ако кашњење двоулазне конјункције означимо са Δ, тада ће кашњење кола на средњој слици бити 3Δ у најгорем случају, док ће кашњење десног кола бити 2Δ. Дакле, пожељно је да се двоулазна И кола групишу тако да је стабло добијеног израза балансирано. У општем случају, кашњење овако реализоване n-улазне конјункције ће бити $\lceil \log_2(n) \rceil \cdot \Delta$ (док би у случају реализације као на левој слици кашњење било $(n-1) \cdot \Delta$).

Алтернатива овом приступу је да се логичке капије реализују директно као вишеулазне, уколико то технологија дозвољава. На слици 2.16 дат је пример имплементације троулазног НИ кола у CMOS технологији.



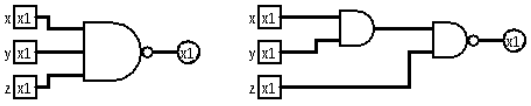
Слика 2.16: Директна реализација четвороулазне конјункције у CMOS-у

Дакле, у доњој мрежи сада имамо три редно повезана NMOS транзистора, а у горњој мрежи имамо три паралелно повезана PMOS транзистора. Додавањем негације на излаз овог кола добили бисмо троулазно И коло. На сличан начин се могу имплементирати и капије са

већим бројем улаза.

Иако на први поглед делује да смо на овај начин решили питање повећаног кашњења које настаје уланчавањем двоулазних И кола, ово заправо није тачно. Наиме, на кашњење оваквог кола негативно утиче отпорност редне везе транзистора (која расте линеарно са бројем улаза), као и електрична капацитивност читаве капије (која се такође линеарно повећава са бројем транзистора у капији). Кашњење ће, грубо говорећи, бити пропорционално производу отпорности и капацитивности, што значи да ће кашњење квадратно расти са бројем улаза. Отуда се оваква реализација асимптотски понаша знатно лошије од балансираног уланчавања двоулазних И капија, па се не исплати користити је за веће вредности n. [14] Због тога ћемо ми у наставку претпостављати да се вишеулазне капије увек реализују балансираним уланчавањем двоулазних капија.

Приметимо да уланчавање капија није могуће за везнике НИ и НИЛИ, с обзиром да n-арне верзије ових везника нису дефинисане свођењем на одговарајуће бинарне везнике, већ као негације И и ИЛИ везника (видети одељак 1.3.3). Због тога се троулазно НИ коло не може добити композицијом два двоулазна НИ кола (јер $x \uparrow y \uparrow z = \overline{xyz} \neq \overline{\overline{xy}z} = (x \uparrow y) \uparrow z$). Ипак, троулазно НИ коло се може реализовати уланчавањем једног двоулазног И кола и једног двоулазног НИ кола (јер је $x \uparrow y \uparrow z = \overline{xyz} = \overline{(xy)} \cdot z = (xy) \uparrow z$). Оваква реализација је приказана на слици 2.17.



Слика 2.17: Реализација троулазног НИ кола уланчавањем двоулазног И и двоулазног НИ кола

Као и код вишеулазних И, ИЛИ и ЕИЛИ кола, и овде се најефикаснија реализација добија балансираним уланчавањем двоулазних кола, при чему се код реализације вишеулазног НИ кола користе двоулазна И кола, изузев последњег кола у ланцу које је двоулазно НИ коло. Аналогна ситуација је са вишеулазним НИЛИ колом. Кашњење овакве реализације је, као и раније, логаритамска функција од броја улаза.

[14] Осим кашњења, вишеулазне капије имају и додатно технолошко ограничење које се тиче улазног напона. Наиме, што је већи број транзистора у редној вези, потребно је довести већи напон на гејтове транзистора како би они проводили. Како је напон у колу обично ограничен напоном батерије која напаја уређај (типично $5V$), број транзистора у редној вези је такође ограничен.

Глава 3

Комбинаторна кола

Комбинаторна кола (енгл. *combinatorial circuits*) су логичка кола код којих се вредности на излазима у сваком тренутку могу изразити као логичке функције од вредности улаза у том истом тренутку. Дакле, вредности које су биле на улазима у претходним тренутцима не утичу на вредности излаза у датом тренутку. Одавде следи да ова кола немају могућност памћења стања које је одређено претходним вредностима на улазима. Помоћу комбинаторних кола се могу имплементирати основне аритметичке и логичке операције над бинарним бројевима, будући да се логичке вредности 0 и 1 могу разумети и као бинарне цифре – *битови* (енгл. *BInary digiT*). Дизајн комбинаторних кола је са теоријске стране једноставан: потребно је само задати функције које описују зависност улаза и излаза, те функције представити изразима и евентуално те изразе минимизовати. У пракси, проблем је у томе што је број улаза и излаза најчешће велики, па је функције тако великог реда тешко директно представити изразима. На пример, ако бисмо имали комбинаторно коло које имплементира бинарни 32-битни сабирач, тада би то коло имало 64 улаза (два пута по 32 бита) и 33 излаза (32битни збир и додатни бит за пренос). Сваки од излаза је сада функција реда 64, па би за директно представљање оваквих функција било потребно формирати таблицу са 2^{32} врста (или ДНФ израз са исто толико елементарних конјункција у себи у најгорем случају), што није једноставно урадити са постојећим ресурсима. Због тога се комбинаторна кола дизајнирају хијерархијски[1] – полазећи од елементарних логичких кола (логичких капија) најпре се формирају једноставна комбинаторна кола која имплементирају једноставне функције. Комбиновањем ових једноставних функција формирају се сложеније, и тако даље. На овај начин се поједностављује сам дизајн, а такође се смањује број потребних гејтова за реализацију функције, као и простор потребан за реализацију кола на чипу. Са друге стране, овакав начин реализације логичког кола повећава његове кашњење, с обзиром да се повећава дубина кола. Нека од најчешће коришћених комбинаторних кола разматрамо у наставку ове главе.

Напоменимо још да ћемо у наставку логичке вредности на улазима и

[1] Овакав приступ се често у литератури назива и *логика на више нивоа* (енгл. *multi-level logic*).

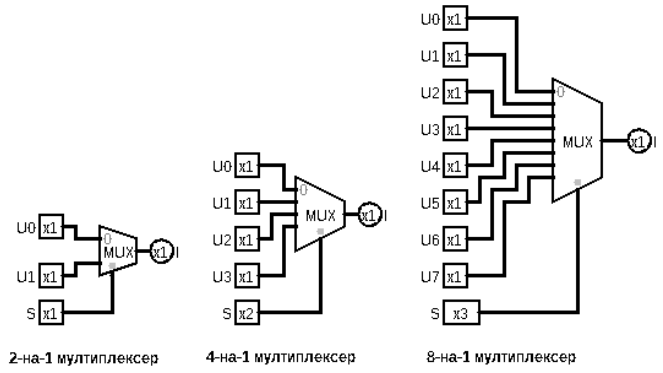
излазима кола поистовећивати са бинарним цифрама, па ћемо их често називати и *битовима*. Такође, поједине групе улаза (или излаза) ћемо често посматрати као целине, тј. као вишебитне бинарне бројеве. Такве улазе и излазе ћемо називати *вишебитним*.

3.1 Основна комбинаторна кола

У овом поглављу разматрамо основна комбинаторна кола која се користе за изградњу сложенијих комбинаторних кола.

3.1.1 Мултиплексер

Мултиплексер (енгл. *multiplexer* или *mux*) је комбинаторно коло које омогућава избор једне од више понуђених вредности. Мултиплексер има 2^k улаза и један излаз, као и додатних k селекционих улаза помоћу којих се врши избор једног од 2^k улаза који ће се проследити на излаз. Овакав мултиплексер зовемо 2^k-*на*-1 *мултиплексер*. Шематске ознаке мултиплексера 2-на-1, 4-на-1 и 8-на-1 приказане су на слици 3.1.



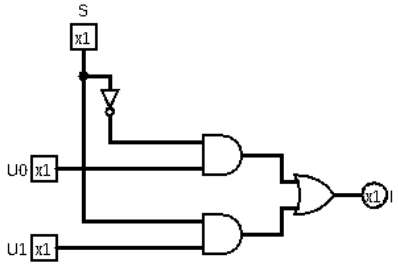
Слика 3.1: Шематска ознака мултиплексера

Селекционе улазе 2^k-на-1 мултиплексера можемо посматрати као један k-битни цео број који представља индекс улаза који желимо да проследимо на излаз. На пример, ако у случају 8-на-1 мултиплексера на селекционим улазима имамо комбинацију 010, тада ће се вредност улаза U_2 проследити на излаз.

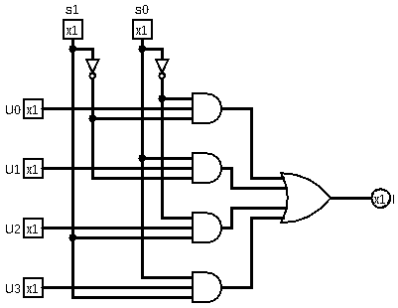
Имплементација мултиплексера. Пример имплементације 2-на-1 мултиплексера дат је на слици 3.2.

Уколико је $S = 0$, тада ће горња конјункција на свом излазу имати вредност U_0, док ће доња конјункција на излазу имати 0. Отуда ће вредност на излазу дисјункције бити једнака U_0. Ако је $S = 1$, тада ће горња конјункција на свом излазу имати вредност 0, док ће доња конјункција имати вредност U_1, па ће на излазу дисјункције бити вредност U_1.

Пример имплементације 4-на-1 мултиплексера дат је на слици 3.3.



Слика 3.2: Имплементација 2-на-1 мултиплексера



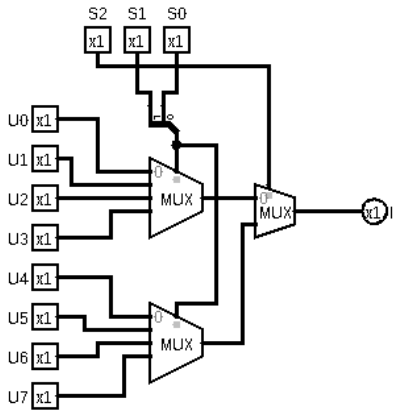
Слика 3.3: Имплементација 4-на-1 мултиплексера

Овога пута имамо троулазне конјункције уместо двоулазних. Сваку од конјункција активира одговарајућа комбинација селекционих улаза. На пример, ако на селекционим улазима имамо комбинацију 00, тада ће прва конјункција одозго на свом излазу имати U_0, док ће на излазу осталих конјункција бити 0, па ће на излазу дисјункције бити U_0. Слично је и за остале комбинације на селекционим улазима.

Повећавањем броја селекционих улаза имплементација мултиплексера се компликује, зато што је потребно имати конјункције и дисјункцију са великим бројем улаза. У општем случају, мултиплексер 2^k-на-1 ће се састојати из 2^k $(k+1)$-улазних конјункција и једне (2^k)-улазне дисјункције. Ако претпоставимо да се вишеулазне конјункције и дисјункције реализују балансираним уланчавањем двоулазних кола, број гејтова за реализацију овог мултиплексера биће једнак $k \cdot 2^k + 2^k - 1 = 2^k \cdot (k+1) - 1$, што за велико k постаје превелик број гејтова.

Због тога се сложенији мултиплексери често имплементирају тако што се своде на једноставније мултиплексере. Пример имплементације 8-на-1 мултиплексера помоћу 4-на-1 и 2-на-1 мултиплексера дат је на слици 3.4.

Селекциони улаз S_2 контролише 2-на-1 мултиплексер којим се врши избор између ниже групе улаза $U_0 - U_3$ и више групе улаза $U_4 - U_7$. Селекциони улази S_0 и S_1 контролишу два мултиплексера 4-на-1 који су задужени за избор једног од четири улаза из своје групе. Тако ће у случају комбинације 101 мултиплексер 2-на-1 одабрати да на свој излаз проследи вредност која му долази са доњег мултиплексера 4-на-1, а тај мултиплексер



Слика 3.4: Имплементација 8-на-1 мултиплексера помоћу 2-на-1 и 4-на-1 мултиплексера

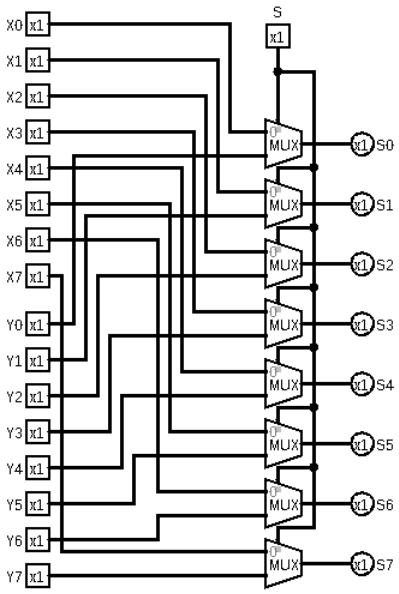
ће на свој излаз прослеђивати улаз U_5 (јер на његове селекционе улазе долази комбинација 01). Дакле, избор улаза који ће се проследити на излаз се врши у два корака: најпре се бира група од четири улаза, а затим се из те групе бира један од улаза. Овај приступ се може примењивати на више нивоа, чиме се могу добити још сложенији мултиплексери.

Овакав дизајн значајно смањује број гејтова. У екстремном случају, ако бисмо мултиплексер 2^k-на-1 реализовали композицијом 2-на-1 мултиплексера на k нивоа, укупан број таквих мултиплексера био би $2^k - 1$, при чему сваки од њих садржи по 3 двоулазне капије, што значи да ће укупан број гејтова бити $3 \cdot 2^k - 3$. Ово је приближно $(k+1)/3$ пута мање гејтова него у директној реализацији 2^k-на-1 мултиплексера. Нпр. за $k = 8$ (мултиплексер 256-на-1) имаћемо 3 пута мање гејтова, док ћемо за $k = 32$ имати 11 пута мање гејтова. Ова уштеда није занемарљива.

Са друге стране, кашњење овакве реализације ће бити нешто веће у односу на директну реализацију. Наиме, код директне реализације 2^k-на-1 мултиплексера имамо кашњење $(\lceil \log_2(k+1) \rceil + k) \cdot \Delta$, (где је Δ кашњење двоулазног гејта), док ће у случају реализације овог мултиплексера композицијом 2-на-1 мултиплексера кашњење бити $2k \cdot \Delta$. На пример, за $k = 7$ у првом случају имаћемо кашњење 10Δ, док ћемо у другом случају имати кашњење 14Δ. За $k = 31$ имаћемо кашњења 36Δ и 62Δ, респективно. Ипак, за велико k пресудну улогу игра број потребних гејтова за реализацију, док је повећано кашњење нужно зло које се мора прихватити.

Вишебитни мултиплексери. Постоје и мултиплексери са вишебитним улазима. На пример 8-битни мултиплексер 2-на-1 ће вршити избор између два 8-битна улаза. У зависности од вредности једнобитног селекционог улаза, један од два 8-битна улаза биће прослеђен на 8-битни излаз. Овакав мултиплексер се може једноставно имплементирати помоћу 8 обичних, једнобитних мултиплексера 2-на-1. Имплементација је приказана на слици 3.5.

Улази x_0 до x_7 чине један осмобитни улаз, док улази y_0 до y_7 чине други

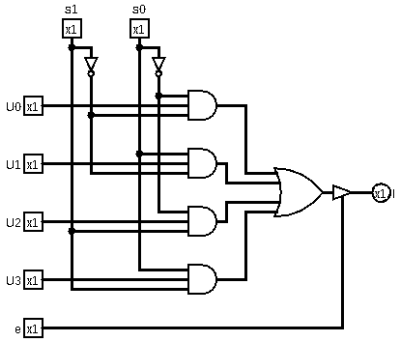


Слика 3.5: Имплементација 8-битног 2-на-1 мултиплексера

осмобитни улаз. Улази x_i и y_i се прослеђују на улазе i-тог мултиплексера. Сви мултиплексери су контролисани од стране истог селекционог улаза S. Ако је $S = 0$, тада ће i-ти мултиплексер прослеђивати x_i на излаз, а за $S = 1$ прослеђиваће y_i. Вишебитни мултиплексери имају исту шематску ознаку као и једнобитни, с тим што се обично нагласи да су њихови улази (и излаз) вишебитни.

Мултиплексери са додатним контролним улазом. Уколико постоји потреба да мултиплексер у неким случајевима на излаз не пропушта ништа (односно да на излазу има вредност **Z**), тада се у имплементацији може на излазу додати још један бафер са три стања који такву функционалност омогућава. Пример имплементације 4-на-1 мултиплексера са додатним контролним улазом дат је на слици 3.6.

Примена мултиплексера. Основна примена мултиплексера је одабир једне од више могућих вредности. Таква функционалност нам је потребна, на пример, када желимо да изаберемо вредност коју ћемо послати преко магистрале, или када желимо да одаберемо операцију коју ће израчунавати аритметичко-логичка јединица. Интуитивно, мултиплексер се може посматрати као еквивалент гранању у програмима. На пример, можемо имати два комбинаторна кола која израчунавају вредности два израза E_1 и E_2, и те две вредности се прослеђују на улазе 2-на-1 мултиплексера. Такође, имамо и треће комбинаторно коло које израчунава да ли је испуњен неки услов C. Излаз овог кола (0 ако услов није испуњен, 1 ако јесте) се повезује на селекциони улаз 2-на-1 мултиплексера. Ако је испуњен услов, на излазу мултиплексера ће бити вредност израза E_2, а



Слика 3.6: Мултиплексер 4-на-1 са додатним контролним улазом

ако није, биће вредност израза E_1. Дакле, овакво коло заправо израчунава C-израз `(C ? E2 : E1)`. Мултиплексери са више улаза се, слично, могу користити за имплементацију семантике вишеструког гранања (попут вишеструке `if-else-if` наредбе или `switch` наредбе у C-у).

Мултиплексери се могу користити за декомпозицију логичке функције на више једноставнијих функција (мањег реда). Ово техника је корисна приликом имплементације функција великог реда које је, као што је раније речено, тешко директно имплементирати (помоћу ДНФ израза) због комбинаторне експлозије. Претпоставимо, на пример, да имамо логичку функцију реда три, дату у табели 3.1.

x	y	z	$F(x,y,z)$
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

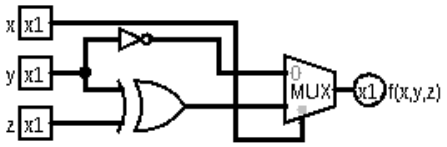
Табела 3.1: Функција реда 3

Ову функцију можемо декомпоновати на различите начине. Први начин је да је посматрамо овако:

$$f(x,y,z) = \begin{cases} f_0(y,z), & \text{за } x = 0 \\ f_1(y,z), & \text{за } x = 1 \end{cases}$$

Дакле, фиксирамо вредност улаза x и разматрамо добијену функцију реда два по y и z. У нашем примеру је $f_0(y,z) = \overline{y}$, а $f_1(y,z) = y \oplus z$. Сада се функција f може имплементирати користећи мултиплексер 2-на-1 за избор између f_0 и f_1 (слика 3.7).

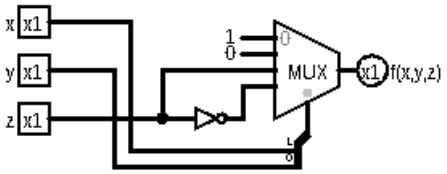
Други начин је да функцију f посматрамо овако:



Слика 3.7: Декомпозиција функције реда 3 на две функције реда 2

$$f(x,y,z) = \begin{cases} f_{00}(z), & \text{за } x=0, y=0 \\ f_{01}(z), & \text{за } x=0, y=1 \\ f_{10}(z), & \text{за } x=1, y=0 \\ f_{11}(z), & \text{за } x=1, y=1 \end{cases}$$

Овог пута фиксирамо вредности променљивих x и y, па разматрамо добијену функцију по z. У нашем примеру је $f_{00}(z) = 1$, $f_{01}(z) = 0$, $f_{10}(z) = z$ и $f_{11}(z) = \overline{z}$. Коло које имплементира функцију на овај начин дато је на слици 3.8. Мултиплексер 4-на-1 се користи за избор између функција f_{00}, f_{01}, f_{10} и f_{11}.



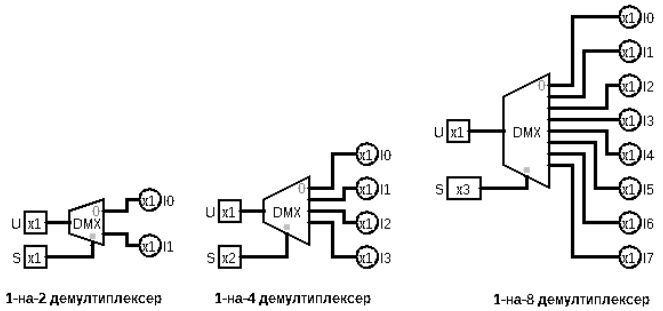
Слика 3.8: Декомпозиција функције реда 3 на четири функције реда 1

У случају функција великог реда, декомпозиција се може вршити на више нивоа. На пример, функција реда 10 се може декомпоновати на 4 функције реда 8 које се даље могу декомпоновати на по 4 функције реда 6 и сл. Опет наглашавамо да се у случају овакве имплементације логичких функција повећава кашњење добијеног логичког кола, али је то у случају функција великог реда неминовно.

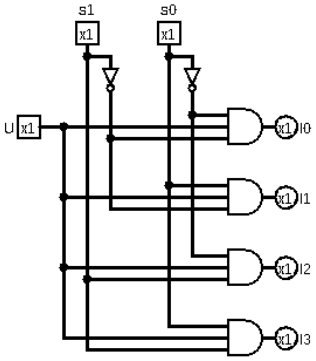
3.1.2 Демултиплексер

Демултиплексер (енгл. *demultiplexer* или *demux*) врши обрнуту функцију од мултиплексера. Демултиплексер има један улаз и 2^k излаза, при чему се улаз преусмерава на тачно један од излаза, у зависности од вредности k-битног селекционог улаза. Другим речима, селекциони улаз тумачимо као бинарни број чија вредност одређује индекс излаза на који треба преусмерити улаз. Овакав демултиплексер се зове и 1-*на*-2^k *демултиплексер*. Шематске ознаке 1-на-2, 1-на-4 и 1-на-8 демултиплексера су дате на слици 3.9.

Имплементација демултиплексера. Пример имплементације демултиплексера 1-на-4 је дат на слици 3.10.



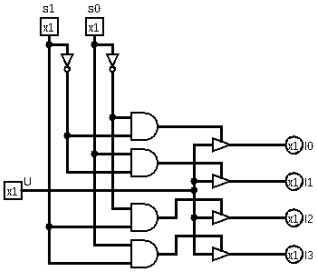
Слика 3.9: Шематске ознаке демултиплексера



Слика 3.10: Пример имплементације 1-на-4 демултиплексера

Слично као и код мултиплексера, свака комбинација на селекционим улазима активира одговарајућу конјункцију која онда пропушта улаз U на свој излаз. Вредност на осталим излазима је 0.

Понекад је пожељно да вредност на осталим излазима не буде 0, већ да буде **Z**. Пример имплементације таквог демултиплексера 1-на-4 је дат на слици 3.11.



Слика 3.11: Демултиплексер 1-на-4 са вредностима високе импедансе на неселектованим излазима

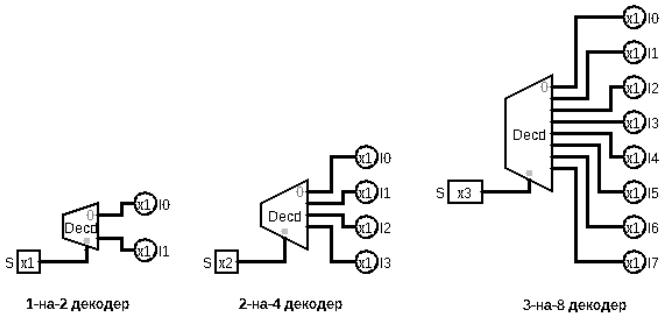
У овој имплементацији се користе бафери са три стања. Свака комбинација на селекционим улазима даје јединицу на одговарајућој конјункцији која активира одговарајући бафер, чиме се пропушта улаз на жељени излаз. Остали бафери су искључени и на њиховим излазима се налази **Z**.

Напоменимо да се демултиплексери већег реда могу реализовати и композицијом демултиплексера мањег реда, на сличан начин као и код мултиплексера, што остављамо читаоцу за вежбу. Предност овакве реализације је, као и тамо, знатно мањи број гејтова потребних за реализацију, док је мана нешто веће кашњење.

Примена демултиплексера. Демултиплексер се користи када је потребно изабрати одредиште одговарајуће вредности која се преноси неком магистралом. Одговарајућим контролним сигналима који се доводе на селекциони улаз демултиплексера вредност са магистрале се пропушта до жељеног кола које ту вредност користи као свој улаз.

3.1.3 Декодер

Декодер (енгл. *decoder*) је комбинаторно коло које декодира бинарно записани број и на основу његове вредности активира одговарајући сигнал на излазу. Ово коло има k-битни селекциони улаз и 2^k излаза. Улаз се тумачи као k-битни бинарни број који представља индекс излаза који треба укључити (тј. поставити на 1). Остали излази имају вредност 0. Овакав декодер се назива и *k-на-2^k декодер*. Шематске ознаке 1-на-2, 2-на-4 и 3-на-8 декодера дате су на слици 3.12.

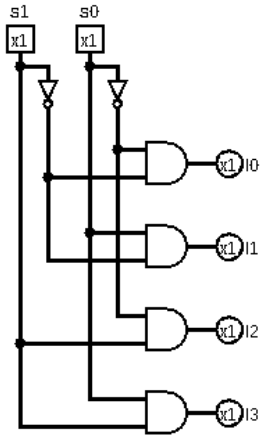


Слика 3.12: Шематске ознаке декодера

Имплементација декодера. На слици 3.13 дат је пример имплементације 2-на-4 декодера.

Као и код демултиплексера, одговарајућа комбинација селекционих битова активира одговарајуће И коло које даје 1 на излазу. Остали излази имају вредност 0.

Декодери већег реда се могу реализовати композицијом демултиплексера мањег реда, чиме се као и раније, постиже значајна



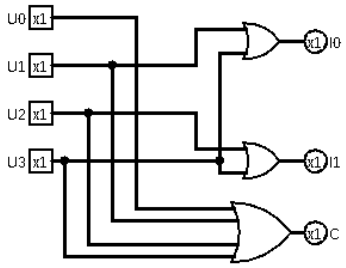
Слика 3.13: Пример имплементације 2-на-4 декодера

уштеда у броју потребних гејтова, али се у извесној мери повећава кашњење.

Примена декодера. Декодер је коло које омогућава одређивање вредности броја који је дат својим бинарним записом. Ефекат који декодер производи је да се укључује одговарајући контролни сигнал који активира одређену акцију. На пример, претпоставимо да имамо машинску инструкцију која као операнд има неки од регистара процесора. Претпоставимо да процесор има 16 регистара и да сваки од њих може бити операнд дате инструкције. Приликом кодирања машинске инструкције мора се на неки начин кодирати који регистар желимо да користимо као операнд. Типично, регистри имају своје индексе $(R_0, R_1, \ldots, R_{15})$ па је најједноставнији начин да се као део машинске инструкције наведе индекс регистра који желимо да користимо као операнд. За ово је потребно 4 бита. Када се инструкција учита у процесор, ова 4 бита инструкције се користе као селекциони улаз декодера који ће укључити одговарајући излаз. Сваки од излаза декодера се користи као контролни сигнал којим се активира одговарајући регистар са тим индексом.

3.1.4 Кодер

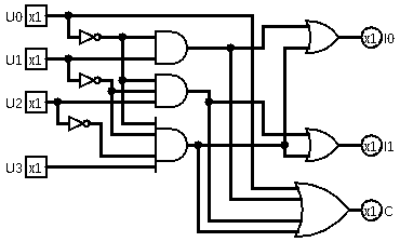
Кодер (енгл. *encoder*) је коло које врши обрнуту функцију од декодера. Кодер има 2^k улаза и k излаза (такав кодер називамо 2^k*-на-*k *кодер*). Претпоставка је да је у сваком тренутку највише један од улаза једнак 1, док су остали 0. Излази кодера се могу тумачити као k-битни бинарни број чија је вредност индекс улаза који има вредност 1. Уколико дозволимо могућност да ни један од улаза нема вредност 1, тада је потребно да можемо и такву ситуацију да детектујемо. У том случају се додаје још један контролни излаз C који ће имати вредност 1 ако је неки од улаза 1, а 0 ако су сви улази 0. Пример имплементације кодера 4-на-2 са контролним излазом је дат на слици 3.14.



Слика 3.14: Пример имплементације 4-на-2 кодера

Контролни излаз C се налази на излазу дисјункције, па ће бити једнак 1 када је било који од улаза 1, а нула ако су сви улази 0. Излаз I_0 ће бити једнак 1 уколико је укључен U_0 или U_3, док ће излаз I_1 бити једнак 1 ако је укључен U_2 или U_3. Уколико пар битова I_1I_0 тумачимо као двобитни бинарни број, вредност на излазу ће управо бити једнака индексу улаза који је укључен.

Недостатак оваквог једноставног кодера је то што његово понашање није добро дефинисано у случају када више улаза истовремено има вредност 1. Овај недостатак се отклања тако што се улазима придружују приоритети. На пример, претпоставимо да највиши приоритет има улаз U_0, а најнижи улаз U_3. Уколико истовремено више улаза има вредност 1, тада ће вредност на излазу бити индекс улаза са највишим приоритетом (тј. са најмањим индексом) који има вредност 1. Пример имплементације кодера са приоритетом дат је на слици 3.15.



Слика 3.15: Кодер 4-на-2 са приоритетом

Разлика у односу на претходну имплементацију је у додатним конјункцијама које спречавају да до дисјункција пролазе улази са већим индексима уколико постоји улаз са мањим индексом који је једнак 1.

Примена кодера. Претпоставимо да имамо неколико регистара и знамо да се нека задата вредност налази у највише једном од регистара. Потребно је одредити индекс регистра који садржи тражену вредност. У том случају бисмо за сваки од регистара имали по један компаратор (видети следеће поглавље) који упоређује вредност тог регистра са траженом вредношћу. Уколико су вредности једнаке, компаратор на излазу даје 1,

а у супротном 0. Како је претпоставка да се тражена вредност налази у највише једном регистру, следи да ће највише један од компаратора дати јединицу на излазу. Излази компаратора се повезују на улазе кодера који израчунава индекс регистра у коме се налази тражена вредност. Оваква функционалност је присутна, на пример, у кеш меморијама. Ако ни један од регистара не садржи тражену вредност, тада ће контролни излаз кодера бити 0, што значи да тражена вредност није пронађена. Уколико бисмо уместо обичног кодера користили кодер са приоритетом, тада би тражена вредност могла да се налази и у више од једног регистра – кодер би нам израчунао најмањи индекс регистра који садржи дату вредност.

Други пример примене кодера са приоритетом је код обраде прекида. Улазно-излазни уређаји шаљу сигнал за прекид када желе пажњу процесора. Потребно је да процесор зна који уређај је послао сигнал за прекид. Уколико их више истовремено захтева прекид, тада је потребно да пажњу процесора добије онај уређај који има највиши приоритет. Кодер се у том случају може користити да одредити индекс уређаја који је захтевао прекид, а који има највиши приоритет.

3.2 Аритметичко-логичка кола

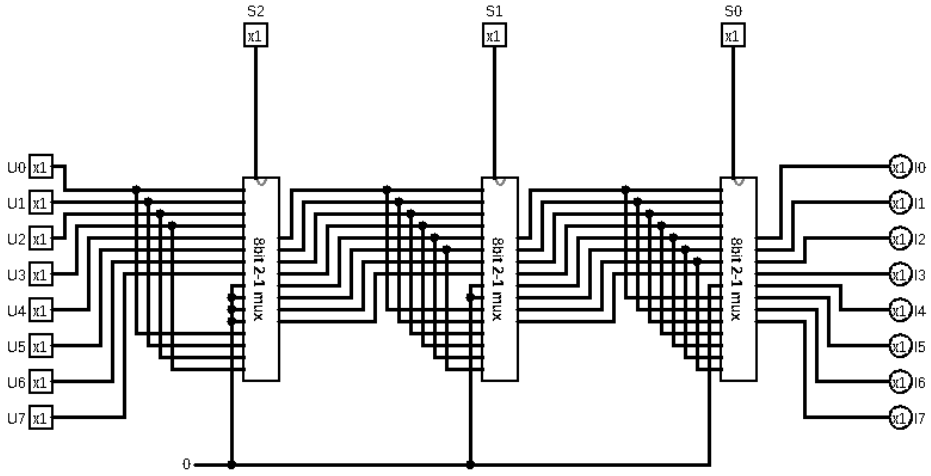
3.2.1 Битовске операције

Под битовским операцијама подразумевају се логичке операције над појединачним битовима вишебитних података. Овакве операције су хардверски подржане од стране већине модерних процесора, а подршка за битовске операције постоји чак и појединим програмским језицима високог нивоа (попут *C*-а и њему сродних језика). Операције које су типично подржане су битовска конјункција, битовска дисјункција, битовска ексклузивна дисјункција и битовска негација. Прве три наведене битовске операције подразумевају да имамо два n-битна податка, а добијени резултат је такође n-битни податак. Притом, i-ти бит резултата се формира тако што се одговарајућа логичка операција примени на i-те битове у полазним подацима. У случају битовске негације, врши се комплементирање сваког од битова полазног податка.

Битовске операције се имплементирају једноставно, помоћу одговарајућих логичких капија. За сваку битску позицију имамо по једну логичку капију која израчунава вредност бита на одговарајућој позицији. Нпр. за имплементацију битовске конјункције над 32-битним подацима потребна су нам 32 двоулазна И кола. Напоменимо да се због једноставности често у шемама кола уместо n логичких капија приказује само једна капија код које се подразумева да су улази n-битни сигнали, а да је излаз такође n-битни сигнал добијен одговарајућом битовском операцијом. Ово се обично посебно нагласи на одговарајући начин (нпр. изнад линија које су повезане на улазе и излазе напишу се бројеви битова које те линије садрже).

3.2.2 Померачи

Поред операција наведених у претходном одељку, у битовске операције се обично убрајају и операције померања (енгл. *shift*). Овим операцијама се бинарни садржај вишебитног податка помера у лево или десно за жељени број позиција. Приликом померања у лево, битови на левом крају бивају истиснути (тј. губе се), а упражњена места на десном крају се попуњавају нулама. Код померања у десно постоје две варијанте – логичко и аритметичко померање. Логичко померање је аналогно левом померању: садржај се помера у десно, битови на десном крају бивају истиснути, а упражњене позиције на левом крају се попуњавају нулама. Код аритметичког померања у десно разлика је у томе што се упражњена места на левом крају попуњавају *битом знака* (највишим битом полазног податка).



Слика 3.16: 8-битни померач у лево

На слици 3.16 приказан је осмобитни померач у лево. На улазу кола се налази осмобитни податак $\mathbf{x} = x_7x_6x_5x_4x_3x_2x_1x_0$, као и тробитни улаз $\mathbf{S} = S_2S_1S_0$ који одређује за колико се битова податак $\mathbf{x}$ помера у лево (осмобитни податак се може померити за највише 7 позиција, те је тробитни улаз $\mathbf{S}$ довољан да се њиме изрази жељени број позиција за померање). Имплементација користи три осмобитна 2-на-1 мултиплексера. Први мултиплексер (са лева у десно) на горњем улазу има управо податак $\mathbf{x}$, док на доњем улазу има $\mathbf{x}$ померен за 4 позиције у лево (нижа 4 бита су нуле, док се на виша 4 бита доводе нижа 4 бита податка $\mathbf{x}$). Уколико је бит S_2 нула, тада ће се горњи улаз проследити на излаз, па неће бити померања, док ће у случају да је S_2 јединица, излаз из првог мултиплексер бити $\mathbf{x}$ померен за 4 бита у лево. Други мултиплексер на горњем улазу има излаз претходног мултиплексера, док на доњем улазу има излаз претходног мултиплексера померен за две позиције у лево (на нижа два бита имамо нуле, док на виших 6 бита имамо нижих 6 бита излаза првог мултиплексера). Уколико је сада $S_1 = 0$, тада ће се на излаз другог мултиплексера прослеђивати

неизмењен излаз првог мултиплексера (дакле, нема додатног померања), док ће за $S_1 = 1$ садржај бити додатно померен за 2 позиције у лево. Најзад, последњи мултиплексер на горњем улазу има излаз другог мултиплексера, а на доњем има излаз другог мултиплексера померен за једну позицију у лево. У случају да је $S_0 = 0$ излаз другог мултиплексера ће непромењен ићи на излаз трећег, а у случају $S_0 = 1$ имаћемо додатно померање за једну позицију у лево. Укупно, број померања у лево биће $4S_2 + 2S_1 + S_0$ што одговара вредности бинарног броја $\mathbf{S} = S_2S_1S_0$.

Померачи у десно се могу имплементирати на сличан начин, с тим што се у случају аритметичког померања на улазе мултиплексера који одговарају „упражњеним" битовима доводи највиши бит x_7 улаза померача, док се код логичког померања у десно доводе нуле.

Анализирајмо сада кашњење овог кола. Ако узмемо да је кашњење 2-на-1 мултиплексера 2Δ, онда је кашњење целог кола 6Δ. У општем случају, кашњење ће бити $2\Delta \cdot log_2(n)$, где је n број битова података $\mathbf{x}$.

3.2.3 Сабирачи и одузимачи

Сабирачи и одузимачи представљају основна аритметичка кола, јер се на њих могу свести и све остале аритметичке операције. На пример, множење се може свести на узастопно сабирање, а дељење на узастопно одузимање. Такође, поређење два броја се може свести на њихово одузимање и разматрање знака вредности добијене разлике. Отуда је од великог значаја ефикасна имплементација сабирања и одузимања у хардверу рачунара.

n-битни сабирач је логичко коло које има два n-битна улаза x и y који представљају сабирке (у бинарном запису) и један n-битни излаз S који представља збир. Поред тога, сабирач обично има још један једнобитни излаз C који представља индикатор прекорачења (јер збир два n-битна сабирка може имати $n + 1$ бит у најгорем случају, па се тај додатни бит обично на излазу представља као прекорачење). Такође, сабирачи обично имају и један додатни једнобитни улаз pc који представља тзв. *претходни пренос* и који се сабира са x и y (другим речима, сабирач на излазу заправо даје вредност $x + y + pc$, при чему се нижих n битова овог збира добијају на излазу S, а највиши $(n + 1)$-ви бит се добија на излазу C). Овим се омогућава уланчавање сабирача. Наиме, n-битним сабирачем ми не можемо сабрати било која два природна броја, већ само природне бројеве који имају највише n цифара у бинарном запису. Ово ограничење се може надокнадити тако што се у случају потребе за сабирањем бројева са већим бројем битова најпре изврши сабирање најнижих n битова датих бројева, након чега се саберу следећих n битова (уз узимање у обзир прекорачења са нижих n битова), и тако даље. Ово уланчавање се може извршити било хардверски (физичким надовезивањем више сабирачких кола, при чему се C излаз сваког сабирача повезује на pc улаз следећег сабирача у низу), било софтверски (тако што се у више корака врши сабирање n по n битова помоћу истог хардверског сабирача). На савременим архитектурама ово софтверско уланчавање се реализује помоћу *ADDC* (енгл. *add-with-carry*) инструкције.

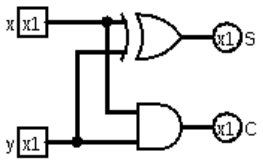
Једнобитни сабирач

Приказ имплементације сабирача започињемо једнобитним сабирачем. Он се обично реализује у две фазе. У првој фази се реализује сабирање два једнобитна податка без узимања у обзир претходног преноса (тј. без pc улаза). Овакво коло се обично назива *полусабирач* (енгл. *half adder*). У другој фази се коришћењем два полусабирача реализује тзв. *потпуни сабирач* (енгл. *full adder*) који узима у обзир и претходни пренос.

x	y	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Табела 3.2: Функција полусабирача

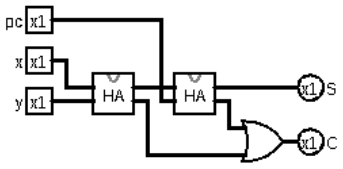
Да бисмо имплементирали полусабирач, посматрајмо најпре његову функцију задату таблично (табела 3.2). Дакле, полусабирач сабира два бита x и y и као резултат даје њихов збир S и пренос C (који се може разумети и као виши бит збира). Из таблице се види да је излаз $S = x \oplus y$, док је $C = x \cdot y$. Шема полусабирача дата је на слици 3.17.



Слика 3.17: Полусабирач

Уколико желимо да наше коло узима у обзир и претходни пренос, потребно је, заправо, сабрати три бита: x, y и pc. Узимајући у обзир асоцијативност операције сабирања, јасно је да можемо најпре помоћу једног полусабирача сабрати x и y, а затим на добијени збир другим полусабирачем додати pc. Вредност збира ће тада бити на излазу S другог полусабирача. Остаје још питање израчунавања преноса C. Пренос C ће бити једнак јединици ако је збир $x + y + pc$ двоцифрен, тј. ако су бар два од ова три бита једнака 1. У том случају ће или x и y оба бити јединице, па ћемо имати пренос на првом полусабирачу (без обзира на вредност улаза pc), или ће x и y бити различитих вредности (па ће збир на првом полусабирачу бити 1), а pc ће такође бити 1, одакле ће се на другом полусабирачу појавити пренос. Дакле, коначни пренос ће заправо бити дисјункција два парцијална преноса на првом и другом полусабирачу. Отуда добијамо коло потпуног сабирача, приказано на слици 3.18 (са HA су означени полусабирачи).

Под претпоставком да је кашњење двоулазних И и ЕИЛИ кола једнако Δ, кашњење полусабирача ће бити Δ за оба излаза. Надовезивањем два полусабирача на описани начин добијамо кашњење од 2Δ за излаз S, односно 3Δ за излаз C у најгорем случају.



Слика 3.18: Потпуни сабирач

x	y	pc	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Табела 3.3: Функција потпуног сабирача

Алтернативни приступ је да потпуни сабирач конструишемо директно, минимизацијом одговарајућих логичких функција за S и C (табела 3.3). За функцијау S имамо Карноову мапу дату на слици (3.19). У питању је „шаховска табла", што је најгори могући случај – ништа се не може груписати, па је резултат израз у савршеној ДНФ форми:

$$S = x\overline{y}\,\overline{pc} + \overline{x}y\overline{pc} + \overline{x}\,\overline{y}\cdot pc + xy\cdot pc$$

	$\overline{x}\overline{y}$	$\overline{x}y$	xy	$x\overline{y}$
$\overline{pc}$	0	①	0	①
pc	①	0	①	0

Слика 3.19: Карноова мапа функције S

Карноова мапа функције C дата је на слици 3.20. Из ове мапе добијамо израз:

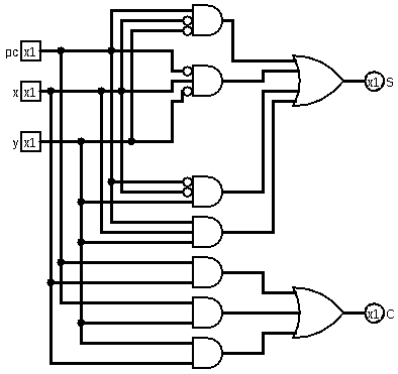
$$C = xy + y\cdot pc + x\cdot pc$$

Имплементација[2] потпуног сабирача према добијеним логичким изразима дата је на слици 3.21.

[2]Иако на први поглед није лако уочити логичку везу између ове две имплементације потпуног сабирача, она ипак постоји. Наиме, уколико пођемо од тога да је $x\oplus y = \overline{x}y + x\overline{y}$, тада имамо да је (из прве имплементације) $C = x\cdot y + (x\oplus y)\cdot pc = xy + (\overline{x}y + x\overline{y})\cdot pc =$

	$\overline{x}\overline{y}$	$\overline{x}y$	xy	$x\overline{y}$
$\overline{pc}$	0	0	1	0
pc	0	1	1	1

Слика 3.20: Карноова мапа функције C



Слика 3.21: Директна имплементација потпуног сабирача

Кашњење излаза S ће у овом случају бити једнака $(\lceil\log_2(3)\rceil+\lceil\log_2(4)\rceil)\cdot\Delta = 4\Delta$, док ће кашњење на излазу C бити $(\lceil\log_2(2)\rceil+\lceil\log_2(3)\rceil)\cdot\Delta = 3\Delta$. Уз то, број гејтова је знатно већи, па је закључак[3] да је исплативије користити имплементацију засновану на надовезивању два полусабирача.

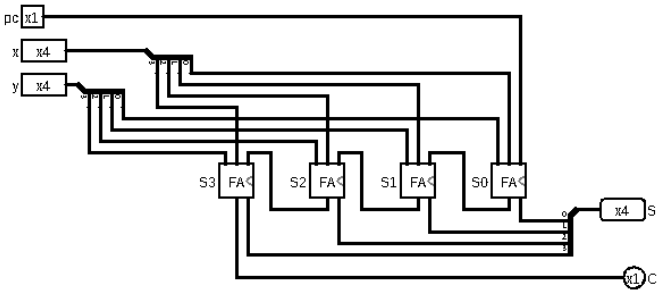
Вишебитни сабирач

n-битни сабирач можемо имплементирати хардверским уланчавањем једнобитних потпуних сабирача. Пример такве имплементације 4-битног сабирача дат је на слици 3.22.

На горњој слици, потпуни сабирачи су означени са *FA*. Четворобитни улази x и y представљају бинарне бројеве које сабирамо, и њихови се битови доводе на одговарајуће улазе потпуних сабирача (на x и y улазе i-тог сабирача доводе се битови x_i и y_i, респективно). Излаз C сваког сабирача је повезан на pc улаз сабирача на следећој битској позицији. На овај начин се преноси са сваке од позиција урачунавају у збир на следећој позицији. Излаз C сабирача на највишој позицији представља индикатор

$xy+\overline{x}y\cdot pc+x\overline{y}\cdot pc$. Ово је даље једнако $xy+\overline{x}y\cdot pc+x\overline{y}\cdot pc+xy\cdot pc+xy\cdot pc$ на основу закона апсорпције и идемпотенције. Груписањем друге и четврте, као и треће и пете конјункције добијамо израз као у другој имплементацији. Слично се може урадити и за излаз S.

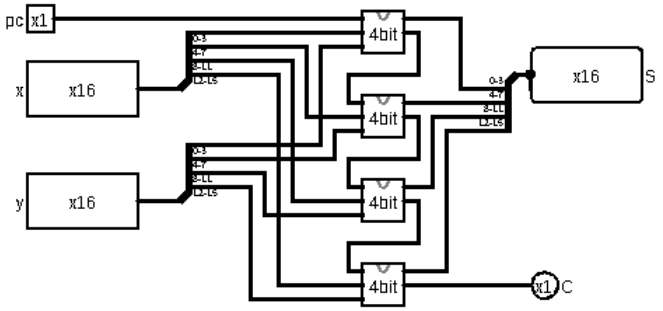
[3] Овај закључак важи под наведеним претпоставкама о кашњењима гејтова, тј. да сви двоулазни гејтови имају једнако кашњење, као и да се вишеулазни гејтови реализују балансираним уланчавањем двоулазних. Уколико ово није случај, закључак може бити другачији. На пример, ако технологија не подржава директну имплементацију ЕИЛИ кола, или је његово кашњење превелико, тада реализација излаза S као у другој имплементацији може бити исплативија.



Слика 3.22: 4-битни сабирач

прекорачења 4-битног сабирача (означен као излаз C целог кола), док улаз pc сабирача на најнижој позицији представља улаз за претходни пренос 4-битног сабирача (означен као улаз pc целог кола). Дакле, наш сабирач заправо израчунава вредност $x + y + pc$, где су x и y четворобитни, а pc једнобитни број. Резултат сабирања (који има највише 5 битова) се добија на четворобитном излазу S (нижа четири бита) и једнобитном излазу C (највиши бит збира, тј. индикатор прекорачења).

Улаз pc као и излаз C n-битног сабирача омогућавају даље уланчавање. Тако помоћу четири 4-битна сабирача можемо добити један 16-битни сабирач (слика 3.23).



Слика 3.23: Имплементација 16-битног сабирача надовезивањем 4 4-битна сабирача

На горњој слици смо 4-битне сабираче повезали на исти начин као што смо на претходној слици повезивали једнобитне сабираче, с тим што се сада шеснаестобитни улази x и y деле у групе од по четири бита које се прослеђују одговарајућим 4-битним сабирачима. Дакле, принцип уланчавања је увек исти.

Проблем са описаном имплементацијом вишебитног сабирача је у превеликом кашњењу кола, нарочито за велико n. Наиме, претпоставимо да се у тренутку t_0 на улазе n-битног сабирача доведу вредности x, y и pc. Прелиминарне вредности S излаза (без урачунатих претходних преноса) биће израчунате са кашњењем 2Δ. Сам пренос ће се пропагирати кроз коло

у облику таласа са једног на други сабирач са кашњењем од 2Δ по биту.[4] Тек када до неког потпуног сабирача дође одговарајући пренос, он ће моћи да ту вредност урачуна и коригује прелиминарно израчунати бит збира, упоредо рачунајући пренос за следећу позицију. Да би се комплетан збир израчунао на исправан начин, потребно је да се пренос пропагира кроз цело коло, за шта је потребно укупно $n \cdot 2\Delta$ (упоредо са пропагацијом преноса вршиће се и корекција битова збира, па за то неће бити потребно додатно кашњење). На пример, за 4-битни сабирач кашњење ће бити 8Δ, док ће за 32-битни сабирач кашњење бити чак 64Δ. Дакле, кашњење расте линеарно са бројем битова. Оваква имплементација сабирача се, због пропагације преноса у облику таласа често назива и *таласасти сабирач* (енгл. *ripple carry adder*).

Вишебитни сабирачи са израчунавањем преноса унапред

Описани проблем са кашњењем таласастих вишебитних сабирача се у пракси решава на различите начине. Једна од најчешће коришћених техника је тзв. *израчунавање преноса унапред* (енгл. *carry lookahead adder (CLA)*). Основна идеја је да се преноси на свакој од битских позиција изразе као непосредне функције од улаза кола. Нека су $C_0, C_1, \ldots, C_{n-1}$ редом преноси на излазима сваког од n потпуних сабирача. Tada је:

$$C_0 = x_0y_0 + x_0pc + y_0pc = x_0y_0 + (x_0 + y_0) \cdot pc = x_0y_0 + (x_0 \oplus y_0) \cdot pc$$

при чему ова последња једнакост важи зато што ће у случају да су оба бита x_0 и y_0 јединице цео израз и даље бити 1 због конјункције x_0y_0. Ову формулу напишимо у облику:

$$C_0 = G_0 + P_0 \cdot pc$$

где је $G_0 = x_0y_0$, а $P_0 = x_0 \oplus y_0$. На сличан начин се могу формулисати и преноси на осталим битским позицијама:

$$C_i = G_i + P_i \cdot C_{i-1}$$

где је $G_i = x_iy_i$, а $P_i = x_i \oplus y_i$. Вредност G_i нам говори да ли се на битској позицији i *генерише пренос* ка следећем биту, док нам вредност P_i говори да ли битска позиција i *пропагира претходни пренос* C_{i-1} ка следећем биту. Генерисање преноса значи да пренос настаје баш на тој позицији. То ће се десити само ако су оба бита x_i и y_i једнаки 1, јер ће у том случају збир бити двоцифрен, чак и ако нема претходног преноса. Са друге стране, пропагација преноса значи да се пренос не генерише на тој позицији, већ је генерисан на некој претходној позицији, али се пропагира (преноси) кроз текућу позицију ка вишим позицијама. То ће се догодити уколико је један од битова x_i или y_i једнак 1, а други 0: у том случају нема генерисања

[4] Приметимо да ово није у колизији са раније изнетом тврдњом да је кашњење C излаза код потпуног сабирача једнако 3Δ. Наиме, ако је потпуни сабирач реализован помоћу два полусабирача, тада pc од излаза C дели једна конјункција и једна дисјункција. Први полусабирач ће свој посао обавити са кашњењем Δ у односу на почетни тренутак, јер он не зависи од преноса, док ће се кроз друге полусабираче и дисјункције пренос пропагирати са кашњењем 2Δ по биту.

преноса, па ће пренос C_i бити 1 ако и само ако постоји претходни пренос C_{i-1}. Распишимо сада ове формуле за C_1, C_2, C_3:

$$C_1 = G_1 + P_1 \cdot C_0 = G_1 + P_1 \cdot (G_0 + P_0 \cdot pc) = G_1 + P_1G_0 + P_1P_0pc$$

Интуитивно, ово значи да ћемо имати пренос C_1 ако је или генерисан на позицији 1 (G_1), или је генерисан на позицији 0, а пропагира се кроз позицију 1 (P_1G_0), или пренос постоји на pc улазу, а пропагира се кроз позиције 0 и 1 (P_1P_0pc). Слично, имамо:

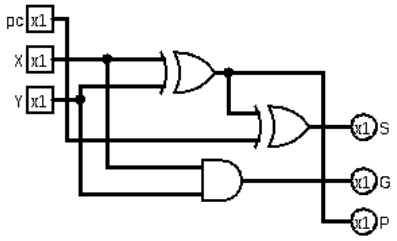
$$C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0pc$$

као и:

$$C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0pc$$

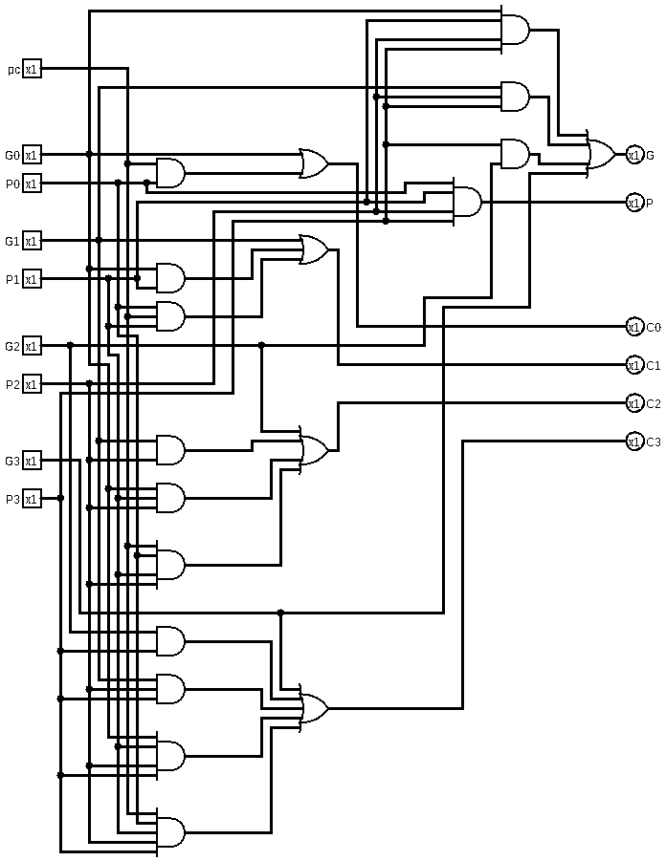
Приметимо да вредности G_i и P_i зависе само од x_i и y_i, па се (за свако $i = 0, 1, 2, 3$) могу израчунати са кашњењем 1Δ, где је Δ кашњење једне двоулазне капије. Отуда се вредност C_i може израчунати са укупним кашњењем $(1 + 2 \cdot \lceil log_2(i + 2) \rceil) \cdot \Delta$ (имамо И кола са највише $i + 2$ улаза, као и ИЛИ коло са $i + 2$ улаза). Последњи пренос C_{n-1} биће израчунат са кашњењем $(1 + 2 \cdot \lceil log_2(n + 1) \rceil) \cdot \Delta$. Након израчунавања, преноси C_i се користе за корекцију битова збира, што се израчунава са додатним кашњењем од 1Δ. Дакле, имамо логаритамски раст кашњења са повећањем броја битова у сабирачу, што је знатно боље него код уобичајеног таласастог сабирача. На пример, за $n = 32$ имаћемо кашњење 13Δ (у односу на 32Δ код таласастог сабирача), док ћемо за $n = 64$ имати кашњење 15Δ (уместо 128Δ). За $n = 256$ имаћемо кашњење од 19Δ, у односу на чак 512Δ код таласастог сабирача.

Илуструјмо сада реализацију четворобитног сабирача са рачунањем преноса унапред. Основно коло од кога полазимо је нешто модификовани једнобитни сабирач на слици 3.24.



Слика 3.24: Једнобитни сабирач са P и G излазима

Ово коло рачуна збир $S_i = x_i \oplus y_i \oplus pc$ (са кашњењем 2Δ), као и P_i и G_i (са кашњењем 1Δ). Приметимо да ово коло не рачуна C_i као код обичних потпуних сабирача. Тај посао преузима посебно коло које ће на основу вредности P_i, G_i и pc у складу са претходним изразима израчунати вредности C_i. Ово коло је познато и под називом *јединица за рачунање преноса унапред* (енгл. *lookahead carry unit (LCU)*). Њена имплементација дата је на следећој слици 3.25.

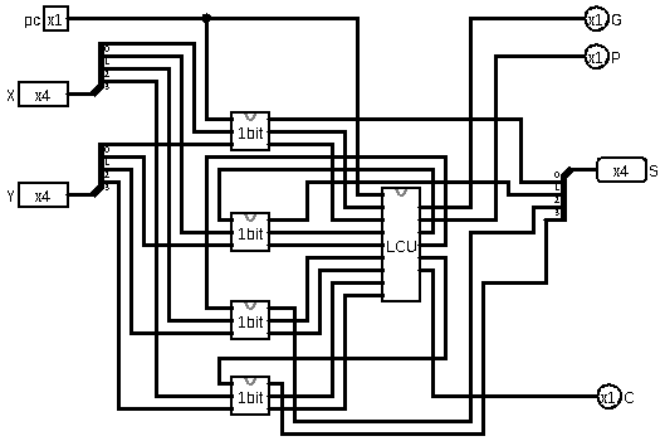


Слика 3.25: Имплементација LCU јединице

Ово коло на својим улазима има вредности P_i и G_i које генеришу модификовани једнобитни сабирачи, као и улаз pc. На излазу имамо вредности C_i које се израчунавају према претходним формулама. Ове вредности ће бити прослеђене назад на pc улазе модификованих једнобитних сабирача, који ће уз додатно кашњење од 1Δ израчунати кориговане вредности S_i. Шема 4-битног сабирача дата је на слици 3.26.

Анализирајмо сада број потребних капија за реализацију израза за израчунавање преноса. Под претпоставком да се вишеулазне капије реализују балансираним уланчавањем двоулазних капија, број капија за израчунавање C_i биће једнак $(1{+}2{+}\ldots{+}i{+}1){+}i{+}1 = (i{+}1){\cdot}(i{+}2)/2{+}(i{+}1) = (i+1)\cdot(i+4)/2$. Може се показати да је за све преносе потребно укупно $n\cdot(n+1)\cdot(n+5)/6 = \Theta(n^3)$ двоулазних капија, уз додатних $2n$ капија за израчунавање P_i и G_i.

Како би се број гејтова смањио, обично се иде на хијерархијски приступ. Претпоставимо да смо на претходно описани начин креирали четворобитни сабирач са рачунањем преноса унапред. Приметимо да LCU коло има још два додатна излаза G_G и P_G који се рачунају по следећим формулама:



Слика 3.26: 4-битни сабирач са рачунањем преноса унапред

$$G_G = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3$$

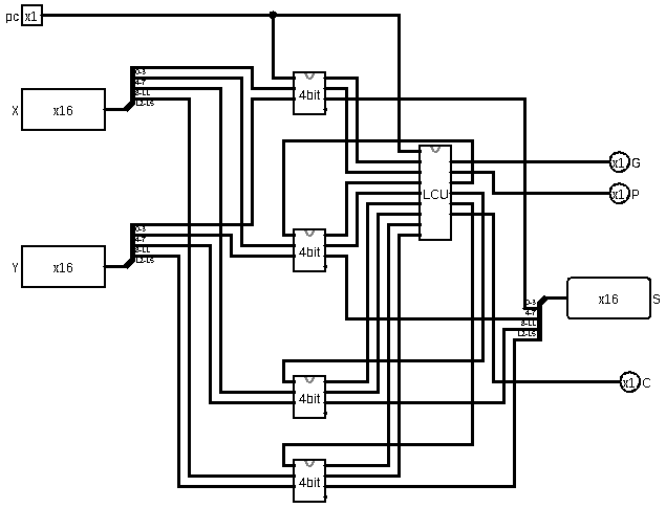
и

$$P_G = P_0P_1P_2P_3$$

Ове две вредности представљају исто што и вредности P_i и G_i, али овог пута на нивоу четворобитног сабирача. Интуитивно, четворобитни сабирач ће генерисати пренос (тј. прекорачење) уколико се или пренос генерише на највишој битској позицији (G_3), или се пренос генерише на позицији 2, а пропагира се кроз позицију 3 (G_2P_3), или се генерише на позицији 1, а пропагира се кроз позиције 2 и 3 ($G_1P_2P_3$), или се генерише на позицији 0, а пропагира се кроз позиције 1, 2 и 3 ($G_0P_1P_2P_3$). Слично, четворобитни сабирач пропагира пренос *pc* са улаза уколико га пропагирају све четири битске позиције. Вредности G_G и P_G су кључне, јер оне омогућавају да се сада четири четворобитна сабирача на исти начин групишу у један шеснаестобитни сабирач. Ово се постиже тако што се на следећем нивоу вредности G_G и P_G које производе четири четворобитна сабирача, уз помоћ додатног LCU кола, користе за брзо рачунање преноса између ових четворобитних блокова. Начин комбиновања четири 4-битна сабирача у 16-битни сабирач приказан је на слици 3.27.

Приметимо да је слика готово идентична као претходна, с тим што се сада уместо 1-битних користе 4-битни сабирачи. Добијени 16-битни сабирач на исти начин рачуна своје G_G и P_G, како би се надаље четири 16-битна сабирача могла на исти начин комбиновати у један 64-битни сабирач на следећем хијерархијском нивоу, итд. Са сваким новим хијерархијским нивоом број битова сабирача се увећава четири пута, али се, извесно, увећава и кашњење.

Покушајмо сада да одредимо кашњење овако добијеног сабирача. Претпоставимо да имамо k хијерархијских нивоа (нумерисаних бројевима од 0 до $k-1$), где се на сваком нивоу четири сабирача са претходног нивоа групишу помоћу једног *LCU* кола. Како се на нултом нивоу налазе



Слика 3.27: 16-битни сабирач са рачунањем преноса унапред на два нивоа

једнобитни сабирачи, укупно ћемо имати $n = 4^k$ битова. Претпоставимо да се вредности на улазе кола x_i, y_i и *pc* доводе у тренутку t_0. Прелиминарне вредности битова збира S_i (без урачунатих преноса) биће израчунате са кашњењем 2Δ, док ће P_i и G_i вредности на нултом нивоу бити израчунате са кашњењем 1Δ. LCU кола на нултом нивоу израчунаће вредности P_G и G_G са додатним кашњењем 2Δ и 4Δ респективно, што значи да ће LCU коло на нивоу 1 имати спремне улазе након 5Δ од почетног тренутка t_0. На сличан начин ће свако следеће *LCU* коло са додатним кашњењем 4Δ израчунати своје G_G и P_G излазе које ће прослеђивати *LCU* колу на следећем нивоу. Приметимо да вредности G_G и P_G, за разлику од излаза C_i, зависе искључиво од вредности P_i и G_i са претходног нивоа, али не и од вредности *pc*. Ово је кључно, јер ће вредности *pc* бити израчунате тек накнадно, помоћу LCU кола на следећем нивоу. LCU коло на последњем $(k-1)$-вом нивоу ће добити своје P_i и G_i улазе након $(4\cdot(k-1)+1)\cdot\Delta$ времена од почетног тренутка t_0. У том тренутку ово коло може израчунати своје C_i излазе са кашњењем 4Δ,[5] чиме започиње пропагација ових вредности уназад ка *pc* улазима четири LCU кола на нивоу $k-2$, где се са додатним кашњењем 4Δ израчунавају вредности C_i на том нивоу и преносе назад на ниво $k-3$ итд. Након укупног кашњења од $4k\cdot\Delta$ вредности C_i на нултом нивоу биће израчунате. Сада је потребно још додатно кашњење 1Δ за корекцију битова збира S_i у једнобитним сабирачима. Укупно кашњење је, дакле, $(4\cdot(k-1)+1+4k+1)\cdot\Delta = (8k-2)\cdot\Delta$. Како је $n = 4^k$, следи да је $k = \log_4(n)$, па је укупно кашњење, изражено у функцији од броја битова сабирача једнако $(8\log_4(n)-2)\cdot\Delta = (4\log_2(n)-2)\cdot\Delta$. Дакле, кашњење је нешто веће него раније, али је и даље у питању логаритамски раст. Што се тиче броја потребних гејтова, свака 4-битна LCU јединица има 30 гејтова за рачунање преноса, уз додатних 12 за рачунање вредности P_G и G_G, што

[5] Кашњење 4Δ има пренос C_2, који има највеће кашњење, ако изузмемо пренос C_3 који није битан јер се он не враћа назад

укупно даје 42 гејта. На k хијерархијска нивоа имамо укупно $(4^k - 1)/3$ LCU јединица. Најзад, имамо 4^k једнобитних сабирача, од којих сваки има по 3 гејта. Сада је укупан број гејтова $42 \cdot (4^k - 1)/3 + 3 \cdot 4^k = 17 \cdot 4^k - 14$. Имајући у виду да је $k = \log_4(n)$, следи да је укупан број гејтова једнак $17n - 14 = \Theta(n)$. Дакле, број гејтова је значајно мањи у хијерархијској имплементацији.

Имплементација одузимача

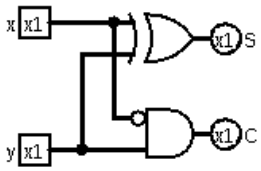
n-битни бинарни одузимач има два n-битна улаза – умањеник x и умањилац y. Вредност разлике добија се на n-битном излазу S. Поред тога, постоји још један једнобитни излаз C који, као и код сабирача, представља индикатор прекорачења. Овај излаз у ствари представља позајмицу на последњој позицији и биће једнак 1 уколико је умањеник мањи од умањиоца. Како би се обезбедила могућност уланчавања одузимача, као и код сабирача постоји додатни једнобитни улаз pc који представља *претходну позајмицу*. Другим речима, одузимач израчунава вредност $x - y - pc$, при чему се вредност разлике добија на излазу S. Уколико је $x < y + pc$, излаз C имаће вредност 1.

Као и код сабирача, полазимо од једнобитног одузимача који се конструише у две фазе. У првој фази формирамо *полуодузимач* који не узима у обзир претходну позајмицу. Функцију полуодузимача задајемо таблично (табела 3.4).

x	y	S	C
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

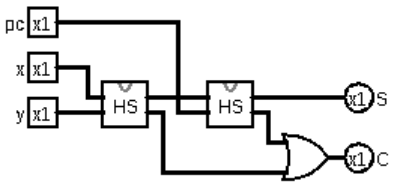
Табела 3.4: Функција полуодузимача

На основу ове таблице, закључујемо да је разлика $S = x \oplus y$, а позајмица $C = \overline{x} \cdot y$. Одговарајуће коло дато је на слици 3.28.



Слика 3.28: Полуодузимач

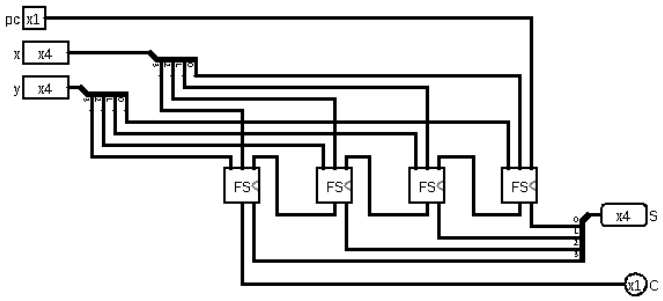
У поређењу са полусабирачем, видимо да је ово коло веома слично, осим што је улаз конјункције на који се доводи x негиран. *Потпуни одузимач* који узима у обзир претходну позајмицу се може добити од два полуодузимача на идентичан начин као у случају потпуних сабирача (слика 3.29).



Слика 3.29: Потпуни одузимач

На горњој слици, са *HS* означили смо полуодузимаче. Најпре од x одузмемо y једним полуодузимачем, а затим од те разлике одузмемо pc другим полуодузимачем. Позајмица C ће постојати уколико је $x < (y + pc)$, а то ће бити или ако је $x = 0$ а $y = 1$ (у ком случају ће се јавити позајмица на првом полуодузимачу), или уколико је $x = y$ и $pc = 1$ (у ком случају ће позајмица бити детектована на другом полуодузимачу). Због тога ће излаз C поново бити дисјункција парцијалних позајмица на излазима полуодузимача. Напоменимо да се и у случају потпуних одузимача може разматрати другачија имплементација заснована на минимизацији функција добијених из таблице потпуног одузимача. Детаље остављамо читаоцу за вежбу.

Надаље се од једнобитних одузимача могу конструисати вишебитни одузимачи уланчавањем, на потпуно исти начин као у случају сабирача. Пример имплементације таласастог 4-битног одузимача дат је на слици 3.30 (са *FS* су означени потпуни једнобитни одузимачи).



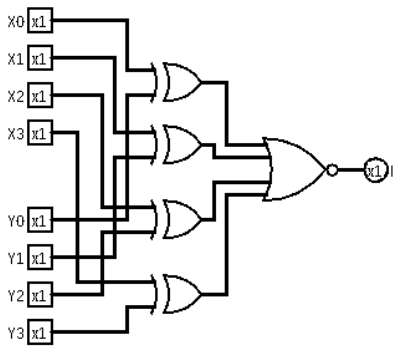
Слика 3.30: 4-битни одузимач

Као и код таласастих сабирача, и овде ће кашњење линеарно расти са повећањем броја битова, па се такође могу разматрати и сличне технике оптимизације, засноване на рачунању позајмице унапред. Поступак је аналоган поступку конструкције сабирача са рачунањем преноса унапред и остављамо га читаоцу за вежбу.

3.2.4 Компаратори

Компаратори су кола која упоређују два податка. У најједноставнијој варијанти, компаратор упоређује два податка и на излазу нам даје

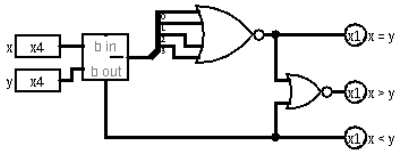
информацију да ли су та два податка једнака или не. Пример имплементације 4-битног компаратора дат је на слици 3.31.



Слика 3.31: 4-битни компаратор на једнакост

ЕИЛИ кола дају нулу на излазу ако и само ако су одговарајући битови једнаки. Уколико једнакост важи на свим битским позицијама, НИЛИ коло ће имати све нуле на улазима, па ћемо на излазу имати јединицу. У свим другим случајевима, излаз кола ће бити нула.

Уколико желимо да, у случају да подаци на улазу нису једнаки, добијемо и додатну информацију који је од њих већи, а који мањи (посматрани као неозначени цели бројеви), тада компаратор постаје сложенији. Један начин да се овакав компаратор имплементира је да се искористи одузимач. Уколико се при одузимању $x - y$ појави прекорачење, тада је $x < y$. У супротном, ако су сви битови резултата нуле (што се може проверити довођењем свих битова резултата на улазе једног НИЛИ кола), тада је $x = y$. У супротном, важи $x > y$. Имплементација је приказана на слици 3.32.

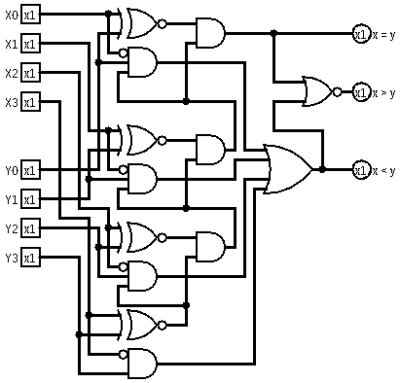


Слика 3.32: 4-битни потпуни компаратор

Коло означено квадратом на горњој слици је четворобитни одузимач. Битови разлике се шаљу на улазе НИЛИ кола које нам даје информацију да ли су подаци једнаки. Излаз који означава прекорачење нам даје информацију да ли је $x < y$. Ако су оба ова излаза нуле, тада је $x > y$.

Нешто једноставнија имплементација (у смислу броја гејтова) дата је на слици 3.33.

На свакој битској позицији имамо једно НЕИЛИ коло (негирано ЕИЛИ) које утврђује да ли су одговарајући битови података x и y на тој позицији једнаки. Резултат поређења на једнакост се акумулира помоћу конјункција (у десној колони на горњој слици). Свака од ових конјункција утврђује да ли истовремено важи једнакост како на текућој позицији, тако и на свим



Слика 3.33: Директна имплементација потпуног 4-битног компаратора

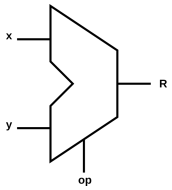
вишим позицијама. Такође, на свакој битској позицији имамо и по једну конјункцију са негираним улазом за x бит (лево на слици, у истој колони са НЕИЛИ колима), која испитује да ли је x бит једнак 0, а y бит једнак 1, под условом да је на свим вишим битским позицијама важила једнакост. Јасно је да уколико било која од ових конјункција да јединицу на излазу, то ће значити да је $x < y$ (јер је на некој битској позицији x бит једнак 0, а y бит једнак 1, док на свим вишим позицијама важи једнакост). Зато се излази ових конјункција повезују на улазе једног ИЛИ кола. Уколико подаци нису једнаки, нити је $x < y$, тада је $x > y$, што се утврђује додатним двоулазним НИЛИ колом.

Напоменимо да се описани поступци упоређивања могу примењивати и на друге типове података, не само за упоређивање неозначених целих бројева. За упоређивање на једнакост довољно је да за дати тип података важи да су два податка једнака ако и само ако су им бинарни записи идентични (ово нпр. важи за означене целе бројеве у потпуном комплементу, али не и за реалне бројеве у покретном зарезу, због двоструког записа нуле, што се мора додатно испитати). За утврђивање који је податак већи, а који мањи, довољно је да за дати тип података постоји дефинисан потпуни поредак, као и да се мање вредности у том поретку записују бинарним записом који је мањи у лексикографском смислу (ово важи за целе бројеве у потпуном комплементу, под условом да упоређујемо бројеве истог знака, што се мора додатно испитати, а слично је и за реалне бројеве у покретном зарезу).

3.2.5 Аритметичко-логичка јединица

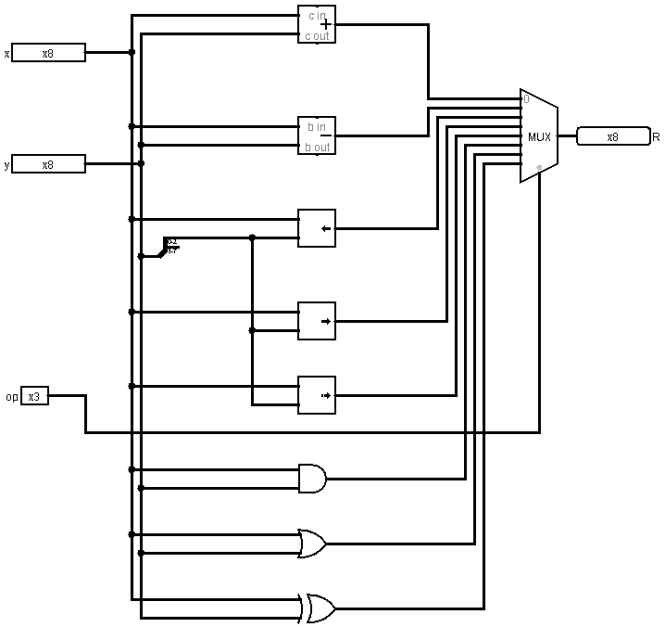
Аритметичко логичка јединица (енгл. *arithmetic logic unit (ALU)*) је један од најважнијих делова савремених рачунарских система и саставни је део сваког модерног процесора. У питању је комбинаторно коло које може да извршава различите аритметичке и логичке операције, у складу са захтевом корисника. Ово коло типично има два улаза x и y на које се доводе подаци над којима се операција извршава, као и улаз *op* на који се доводи кôд операције коју желимо да ALU коло изврши. На излазу R се добија тражени

резултат. Типична ознака ALU јединице је дата на слици 3.34.



Слика 3.34: ALU јединица

ALU јединица конкретног рачунара може подржавати већи или мањи број рачунских операција, од чега зависи и њена сложеност. По структури, ALU јединица се састоји из засебних комбинаторних кола која рачунају конкретне рачунске операције (сабирачи, одузимачи, померачи, компаратори и сл.) као и једног мултиплексера који на основу задатог кôда операције одговарајућу вредност прослеђује на излаз. Пример имплементације једне једноставне 8-битне ALU јединице дат је на слици 3.35.



Слика 3.35: Пример имплементације једноставне ALU јединице

ALU јединица са слике 3.35 подржава сабирање, одузимање, померање у лево и десно (аритметички и логички), као и битовске операције И, ИЛИ и ЕИЛИ. Улаз *op* је тробитни и одређује једну од осам подржаних операција. Мултиплексер на основу вредности *op* улаза тражени резултат прослеђује на излаз кола.

Поред излаза R, ALU јединица може имати и додатне излазе који ближе

одређују статус извршене операције и добијеног резултата (на пример, да ли је резултат нула, који је знак резултата, да ли постоји прекорачење и сл.). Такође, ALU јединица може имати и додатне улазе који могу да утичу на резултат (на пример, приликом софтверског уланчавања код сабирања, од значаја нам је да знамо да ли је било претходног преноса).

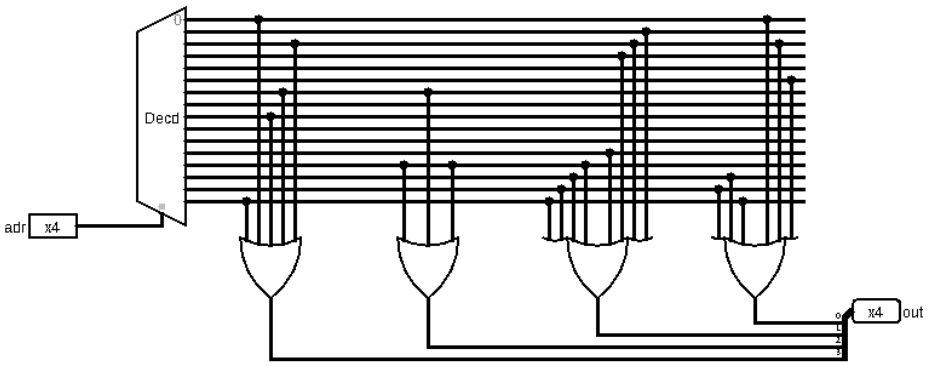
3.3 Општа комбинаторна кола

Поред раније описаних комбинаторних кола која имају специфичну намену, често постоји потреба и за општим комбинаторнм колима која реализују произвољне логичке функције. Иако је теоријски увек могуће да та кола конструишемо помоћу гејтова, након што минимизујемо одговарајуће изразе и одредимо ДНФ, у пракси је често згодније да постоје генеричка комбинаторна кола која се могу „програмирати" тако да реализују жељену функцију. Оваква кола се производе серијски, у облику чипова. У наставку описујемо неке најчешће типове општих комбинаторних кола.

3.3.1 Неизмењиве меморије

Неизмењиве меморије (енгл. *read-only memory (ROM)*) су меморије чији је садржај фиксиран и не може се променити. На улаз ове меморије доводи се *адреса* која је неозначени бинарни број, а на излазу се добија вредност која се у меморији налази на тој адреси. Суштински, ROM меморија је комбинаторно коло, јер нема могућност памћења, већ се понаша као функција која свакој адреси придружује вредност на тој адреси (која је увек иста за фиксирану адресу).

Пример реализације једне ROM меморије дат је на слици 3.36.

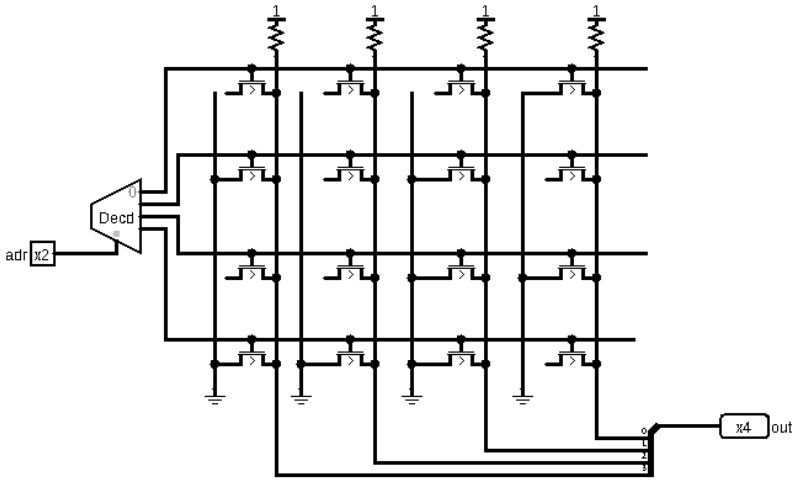


Слика 3.36: Концептуална шема ROM меморије

Декодером се најпре декодира адреса и активира се одговарајући излаз који преко дисјункција активира одговарајуће битове на излазу. Свака од дисјункција се повезује само са оним излазним линијама декодера које одговарају адресама на којима тај бит треба да буде једнак 1. Отуда се садржај ROM меморије одређује тако што одредимо пресеке излазних

линија декодера и улазних линија дисјункција на којима ћемо имати спојеве. Код класичних ROM меморија, овај поступак се обавља у фабрици, тако што се приликом производње чипа одговарајућом маском означе позиције у матрици пресека на којима је потребно обезбедити спој. Са друге стране, PROM меморије (енгл. *programmable ROM*) се производе као генерички чипови који се на лицу места могу програмирати, од стране самог корисника. У случају ових меморија, иницијално (фабрички) постоје сви спојеви, али су они реализовани преко осигурача (ослабљених тачака) који се могу „прегорети" пуштањем нешто јаче струје. Овај поступак се обавља помоћу посебног уређаја који се зове *PROM програматор*. Након што се PROM једном програмира (тј. у њега се упише садржај), више није могуће тај садржај променити. Програмирани PROM чип се уграђује у уређај у коме врши жељену функцију.

На слици 3.37 приказана је нешто ефикаснија имплементација ROM меморије која се чешће јавља у пракси.



Слика 3.37: Реална имплементација ROM меморије

Сваки бит меморије представљен је једним транзистором. Сви транзистори су распоређени у облику матрице код које свака врста одговара једној меморијској адреси, а свака колона једној битској позицији на излазу. Гејтови транзистора у фиксираној врсти су повезани на исти излаз декодера, па се активирају избором одговарајуће адресе. Дрејнови транзистора у фиксираној колони су повезани на одговарајући бит излаза. Преко отпорника, сваки излазни бит је повезан на напајање, па је вредност сваког излазног бита подразумевано јединица, осим ако се отварањем неког од транзистора у колони не успостави веза са нулом. Приликом фабрикације (или приликом накнадног програмирања) неки сорсеви транзистора су повезани на нулу, а неки не. Уколико постоји веза са нулом, тада ће приликом активације одговарајуће адресе на тој битској позицији на излазу бити нула, а у супротном ће бити јединица.

3.3.2 PLA кола и PAL кола

ROM меморије су кола која су идеална за чување неког фиксираног садржаја (програма и података). Са друге стране, ова кола се могу користити и као генеричка комбинаторна кола, јер сваки бит излаза заправо представља логичку функцију од улазних битова (битова адресе), при чему ту функцију можемо сами одабрати, избором одговарајућег садржаја меморије. Међутим, коришћење ROM меморија у ову сврху је најчешће прескупо. Наиме, веома често је случај да функције које су нам потребне буду прилично „ретке", тј. у њиховој таблици доминирају нуле, док се јединице јављају релативно ретко. Ово значи да се СДНФ такве функције састоји из релативно малог броја савршених конјункција. Са друге стране, ROM меморија поседује декодер који у својој имплементацији садржи по једно И коло за сваку савршену конјункцију над улазним променљивама. Већина ових савршених конјункција се уопште не јавља у излазним функцијама. Отуда је једна идеја за оптимизацију да уместо потпуног декодера имамо један релативно мали број конјункција које бисмо могли да „програмирамо" тако да израчунавају оне савршене конјункције које су нам заиста потребне у СДНФ излазних функција. Дакле, сада ћемо имати две матрице потенцијалних спојева: *„И" матрицу* у којој се избором одговарајућих спојева на улазе И кола доводе оне улазне променљиве или њихове негације које учествују у одговарајућој савршеној конјункцији, и *„ИЛИ" матрицу*, која, као и раније, бира који ће се излази И кола проследити на улазе сваког од ИЛИ кола. На овај начин, избором спојева у ове две матрице одређујемо ДНФ форме излазних функција. Оваква кола се обично називају PLA кола (енгл. *programmable logic array*). Ова кола се, попут PROM-а, израђују као генеричка кола, са свим спојевима присутним у облику осигурача који се затим у програматору спаљују по потреби, чиме се добијају тражене функције.

Додатна уштеда се може постићи фиксирањем ИЛИ матрице. У том случају имамо ИЛИ кола на чије су улазе повезани излази фиксираних И кола. Једина могућност избора је на нивоу И матрице: избором спојева можемо одредити променљиве (или њихове негације) које се доводе на улазе И кола. Овакав дизајн значајно смањује флексибилност, али чини коло једноставнијим и јефтинијим. Оваква кола су позната и под називом PAL кола (енгл. *programmable array logic*).

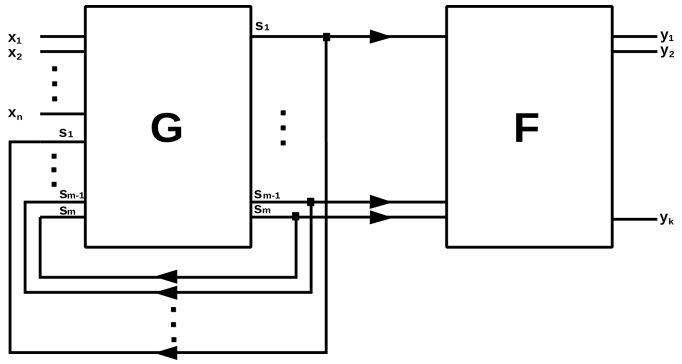
Глава 4

Секвенцијална кола

Код комбинаторних кола, вредност на излазу $Y = (y_1, \ldots, y_k)$ у неком тренутку t зависи искључиво од вредности улаза $X = (x_1, \ldots, x_n)$ у том истом тренутку t. Другим речима, веза између улаза и излаза се може представити на следећи начин:

$$Y = F(X)$$

где је F нека векторска[1] логичка функција по X. Дакле, оваква кола не поседују никакво унутрашње стање које би акумулирало утицај претходних вредности улаза. Под *стањем* логичког кола подразумевамо низ битова $S = (s_1, s_2, \ldots, s_m)$ чије се вредности могу одржавати унутар кола. Вредност стања се може мењати променом вредности на улазу кола, али нова вредност стања не зависи само од нове вредности улаза, већ и од претходне вредности стања. Да бисмо то постигли, потребно је обезбедити да стање на неки начин утиче само на себе. Ово се реализује *повратном спрегом*, као на слици 4.1.



Слика 4.1: Концепутална шема секвенцијалног кола

На овој слици, коло G представља комбинаторно коло које реализује векторску логичку функцију $G(X, S)$. Излаз овог кола представља *стање*

[1] Другим речима, F је вектор логичких функција $F = (F_1, F_2, \ldots, F_k)$, при чему је $y_i = F_i(x_1, \ldots, x_n)$.

овог кола и он се повратном спрегом враћа на његов S улаз (стога ћемо и стање такође означавати са S). *Стабилно стање* биће било која фиксна тачка функције G по S за дато фиксирано X, тј. било које S за које важи:

$$S = G(X, S)$$

Такво стање одржава само себе и неће се променити докле год се улаз X не промени. Излаз кола Y зависиће од текућег стања, тј. имамо:

$$Y = F(S)$$

где је $F(S)$ нека логичка функција по S. У пракси је често $Y = F(S) = S$, тј. излаз кола је често управо стање S.

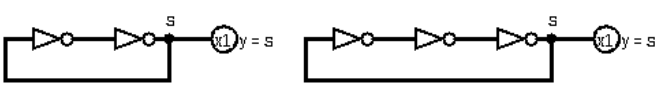
Уколико S није фиксна тачка функције G за дато X (тј. имамо да је $S \neq G(X, S)$), тада ће стање (тј. излаз кола G, па самим тим и улаз S) почети да се мења. Резултат те промене може бити нешто од следећег:

- Коло може стићи у неку фиксну тачку (тј. стабилно стање) S'. За такво S' важи $S' = G(X, S')$. Ово је пожељно понашање. Време потребно колу да дође у стабилно стање S' називамо *временом стабилизације* (енгл. *stabilization time*). Ово време зависи од структуре кола G, као и од кашњења појединачних компоненти из којих се ово коло састоји.

- Коло може осциловати између различитих стања, неуспешно покушавајући да пронађе стабилно стање. Ову појаву зовемо *нестабилност*.

- Коло се може стабилизовати у неком „међустању". Наиме, како се вредности 0 и 1 у електронским колима представљају напонским нивоима (рецимо $0V$ и $+5V$), а транзиција између ова два напонска нивоа није тренутна, може се догодити да се неки бит стања стабилизује на вредностима напона који су између ове две вредности (на пример, $+2.5V$) и које не представљају исправну логичку вредност. Оваква појава се назива *метастабилност*.

- Понекад за дати улаз X и стање S (за које је $S \neq G(X, S)$, тј. S није фиксна тачка за улаз X), имамо да коло може отићи у неко стабилно стање S', али и у неко друго стабилно стање S'', у зависности од разних непредвидивих физичких фактора. Оваква појава се назива *недетерминистичност*.



Слика 4.2: Пример једноставног стабилног и нестабилног кола

Један једноставан пример који илуструје неке од поменутих феномена дат је на слици 4.2. Кола на слици су сасвим једноставна и немају улазе, док су излази y једнаки стању s. Лево коло се састоји из две негације,

тј. $G(s) = \overline{\overline{s}} = s$, па су оба могућа стања $s = 0$ и $s = 1$ фиксне тачке ове функције. Отуда, у ком год да се почетном стању налази, то стање ће одржавати самог себе, тј. одговарајући бит биће запамћен у овом колу. Десно коло се састоји из три негације, тј. $G(s) = \overline{\overline{\overline{s}}} = \overline{s}$. Ова функција нема фиксну тачку, па коло неће моћи да се нађе у стабилном стању. У пракси су могућа два сценарија: или ће коло осциловати између два стања $s = 0$ и $s = 1$, или ће се стабилизовати у неком међустању (тј. имаћемо метастабилност).

Нестабилност, метастабилност и недетерминистичност су лоше појаве које желимо да избегнемо, па се о томе мора водити рачуна приликом дизајна кола. Другим речима, пожељно понашање кола се може описати на следећи начин:

Нека је X произвољна вредност на улазу и S стање такво да је $S = G(X, S)$ (тј. S је стабилно стање за улаз X). Ако у неком тренутку улаз добије нову вредност X', тада коло прелази у ново стабилно стање S' (тј. стање за које важи $S' = G(X', S')$), при чему је то стање S' детерминистички одређено претходним стањем S и новим улазом X', тј. важи да је:

$$S' = T(X', S)$$

где је T нека фиксирана векторска логичка функција. Ову функцију називамо ***функцијом преласка*** *кола G. За кола која имају овакво понашање кажемо и да су* ***добро дефинисана*** *или* ***стабилна****.*

Напоменимо да ће понекад описано својство важити само за неке вредности улаза X, али не за све. У таквим ситуацијама се за стабилни рад кола мора обезбедити да се на улазе кола доводе искључиво допустиве вредности, тј. оне које обезбеђују стабилно и детерминистичко понашање. Функција преласка тада неће бити тотална, већ ће бити само парцијално дефинисана. Већ у следећем одељку видећемо пример таквог кола.

Претпоставимо сада да имамо неко добро дефинисано коло G и да је T функција преласка тог кола. Претпоставимо да у почетном тренутку t_0 имамо на улазу вредност X_0 и стабилно стање S_0 (тј. имамо да је $S_0 = G(X_0, S_0)$). Такође, претпоставимо да је вредност на излазу кола $Y_0 = F(S_0)$. Ако у неком тренутку t_1 променимо вредност улаза на вредност X_1, тада добијамо ново стабилно стање: $S_1 = T(X_1, S_0)$, и сходно томе, нови излаз $Y_1 = F(S_1)$. Нека се, даље, у неком наредном тренутку t_2 улаз поново промени и добије вредност X_2. Сада ћемо имати ново стање $S_2 = T(X_2, S_1)$, као и нову вредност излаза $Y_2 = F(S_2)$. Уопште, ако у тренутку t_i на улаз доведемо допустиву вредност X_i, имаћемо ново стабилно стање $S_i = T(X_i, S_{i-1})$ и излаз $Y_i = F(S_i)$. Сада ће излаз кола у тренутку t_n бити:

$$\begin{aligned} Y_n &= F(S_n) = F(T(X_n, S_{n-1})) = F(T(X_n, T(X_{n-1}, S_{n-2}))) \\ &= F(T(X_n, T(X_{n-1}, T(X_{n-2}, S_{n-3})))) = \dots \\ &= F(T(X_n, T(X_{n-1}, T(X_{n-2}, \dots, T(X_1, S_0) \dots)))) \\ &= F(T(X_n, T(X_{n-1}, T(X_{n-2}, \dots, T(X_1, G(X_0, S_0)) \dots)))) \end{aligned}$$

Дакле, видимо да излаз Y_n у тренутку t_n зависи од *секвенце* свих претходних улаза $X_0, X_1, \dots, X_n$, а не само од X_n, као код комбинаторних кола. Отуда оваква кола називамо *секвенцијалним колима*.

Скуп свих стабилних стања секвенцијалног кола G дефинишемо на следећи начин:

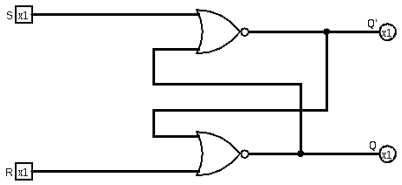
$$\mathcal{S}(G) = \{S \mid \exists X.S = G(X, S)\}$$

Другим речима, то је скуп свих стања S која су стабилна бар за неки улаз X. Приметимо да једно исто стање S може бити стабилно за више различитих улаза (тј. могу постојати различити улази X' и X'' за које је $S = G(X', S)$ и $S = G(X'', S)$. Уколико постоји улаз X_0 такав да за свако стабилно стање из $\mathcal{S}(G)$ важи да је $S = G(X_0, S)$ (тј. свако стабилно стање је стабилно и за улаз X_0), тада секвенцијално коло G називамо и *меморијско коло*. Интуитивно, улаз X_0 се користи за *памћење стања*: у које год стабилно стање да доведемо коло, то стање можемо на даље одржавати тако што ћемо на улазу држати вредност X_0. Ову вредност на улазу називаћемо *пасивном вредношћу*: за разлику од *активних вредности* на улазу које мењају вредност стања, тј. постављају стање на жељену вредност, пасивна вредност на улазу одржава оно што је претходно постављено, тј. обезбеђује могућност памћења стања.

У наставку ове главе разматрамо најједноставнија меморијска кола — *резе*.

4.1 Резе

Реза је меморијско коло које има могућност чувања једнобитног стања. Улази резе омогућавају постављање и одржавање вредности стања, док излази омогућавају очитавање тренутног стања кола. Најједноставнија реза је тзв. *SR-реза*, чија је шема дата на слици 4.3.



Слика 4.3: SR-реза

Дакле, ова реза се састоји из два НИЛИ кола. Коло има два једнобитна улаза R и S, стање се састоји из пара битова (Q, Q'), а излази су једнаки битовима стања. Притом, важи релација повратне спреге:

$$\begin{aligned} Q &= R \downarrow Q' \\ Q' &= S \downarrow Q \end{aligned}$$

Овом релацијом стање је изражено у функцији од улаза и самог себе. У табели 4.1 дата је анализа понашања овог кола (Q^{sled} и Q'^{sled} представљају нове вредности стања, које се израчунавају на основу горњих релација).

Стабилна стања су, дакле, она стања код којих је $(Q^{sled}, Q'^{sled}) = (Q, Q')$. У случају да је улаз $(S, R) = (0, 1)$, имамо само једно стабилно стање, $(Q, Q') = (0, 1)$, док сва остала стања воде у ово стање, уз одређено кашњење

S	R	Q	Q'	Q^{sled}	Q'^{sled}	
0	0	0	0	1	1	нестабилно (циклус)
0	0	0	1	0	1	← стабилно
0	0	1	0	1	0	← стабилно
0	0	1	1	0	0	нестабилно (циклус)
0	1	0	0	0	1	нестабилно
0	1	0	1	0	1	← стабилно
0	1	1	0	0	0	нестабилно
0	1	1	1	0	0	нестабилно
1	0	0	0	1	0	нестабилно
1	0	0	1	0	0	нестабилно
1	0	1	0	1	0	← стабилно
1	0	1	1	0	0	нестабилно
1	1	0	0	0	0	← стабилно
1	1	0	1	0	0	нестабилно
1	1	1	0	0	0	нестабилно
1	1	1	1	0	0	нестабилно

Табела 4.1: Функција SR резе

које је потребно да се коло стабилизује. У случају да на улазу имамо $(S, R) = (1, 0)$, тада такође имамо само једно стабилно стање, овога пута $(Q, Q') = (1, 0)$, док сва остала стања воде у ово стање, након времена стабилизације.

У случају да је $(S, R) = (0, 0)$, тада имамо два стабилна стања $(Q, Q') = (0, 1)$ и $(Q, Q') = (1, 0)$. Стања $(Q, Q') = (0, 0)$ и $(Q, Q') = (1, 1)$ и чине циклус, јер једно стање води у друго и обратно. Дакле, теоријски, имамо нестабилност. У пракси, с обзиром да се приликом преласка између ова два стања мењају оба бита Q и Q', један од битова ће, услед деловања различитих физичких фактора, нешто раније променити своју вредност, па ће се коло стабилизовати или у стању $(Q, Q') = (0, 1)$, или у стању $(Q, Q') = (1, 0)$, при чему је немогуће унапред одредити у ком. Другим речима, имаћемо недетерминистичност. Ово није добро понашање и треба га избећи.

Прецизнијом анализом можемо закључити да до циклуса може доћи једино ако се налазимо у стању $(Q, Q') = (0, 0)$ или $(Q, Q') = (1, 1)$, тј. ако оба бита стања имају исту вредност. На основу горње таблице, једини начин да се коло стабилизује у неком од ова два стања је да на улаз доведемо две јединице, тј. ако је $(S, R) = (1, 1)$. У том случају ће се коло стабилизовати у стању $(Q, Q') = (0, 0)$. Уколико након тога на улаз доведемо $(R, S) = (0, 0)$ имаћемо циклус. Дакле, да би описано коло било стабилно, потребно је да улаз $(R, S) = (1, 1)$ не буде допустив, тј. да обезбедимо да се ова комбинација улазних вредности никада не појављује. У том случају ће коло увек бити у једном од два стабилна стања $(Q, Q') = (0, 1)$ или $(Q, Q') = (1, 0)$. Имајући ово у виду, можемо да закључимо да ће у том случају коло суштински чувати само један бит информације, јер ће бит Q' увек бити комплемент бита Q. Можемо, дакле, надаље сматрати да је стање кола једнобитно, и да га представља вредност бита Q, као и да коло има два излаза – један чија

је вредност управо стање Q, а други чија је вредност комплемент стања $Q' = \overline{Q}$.

За улаз $(S, R) = (0, 0)$ оба ова стања ће бити стабилна, па се ова комбинација улаза користи за одржавање запамћене вредности бита Q (тј. ово је пасивна улазна комбинација). Улаз $(S, R) = (1, 0)$ ће довести до тога да се бит Q постави на 1 (овај поступак зовемо постављање или сетовање бита). Улаз $(S, R) = (0, 1)$ ће довести до тога да се бит Q постави на 0 (овај поступак зовемо искључивање или ресетовање бита). Заправо, ознаке улаза S и R и потичу од њихове описане функције (енгл. *Set* и *Reset*). Као што смо већ напоменули, комбинација $(S, R) = (1, 1)$ је недопустива и никада се не јавља на улазу.

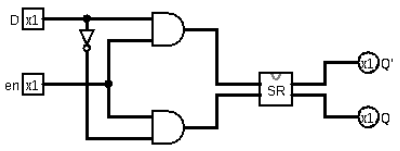
Функција преласка SR резе дата је у табели 4.2.

S	R	Q	Q^{sled}
0	0	0	0
0	0	1	1
0	1	-	0
1	0	-	1
1	1	-	?

Табела 4.2: Таблица преласка SR резе

У табели 4.2, Q^{sled} означава наредно стање: $Q^{sled} = T(S, R, Q)$. Цртице означавају било коју вредност, док упитник означава да је дата комбинација улаза недопустива.

Један од начина да се реши проблем недопустивог улаза $(S, R) = (1, 1)$ је да се ова два улаза повежу негацијом, тј. да имамо само један улаз (означен са D) који се директно повезује на S улаз SR резе, а преко негације на R улаз SR резе. Сада ћемо за $D = 1$ имати комбинацију $(S, R) = (1, 0)$ на улазу SR резе, а за $D = 0$ ћемо имати комбинацију $(S, R) = (0, 1)$ на улазу SR резе. Иако смо на овај начин избегли две јединице на улазима SR резе, изгубили смо и могућност да имамо две нуле на улазима. Отуда би овакво коло ефективно изгубило могућност памћења, јер би увек на излазу Q била вредност једнака улазу D. Овај проблем се решава увођењем додатног улаза e (од енглеске речи *enable*). Имплементација оваквог кола дата је на слици 4.4.



Слика 4.4: Имплементација D-резе

Улазом e се, помоћу два И кола, контролише пролаз вредности са D улаза до улаза SR резе. За $e = 0$ имаћемо две нуле на R и S улазима, па ће SR реза чувати претходно постављено стање. Када је $e = 1$, тада се стање SR резе поставља на вредност улаза D. Овако добијена реза назива се D

реза, и њена функција преласка дата је у табели 4.3.

D	e	Q	Q^{sled}
-	0	0	0
-	0	1	1
0	1	-	0
1	1	-	1

Табела 4.3: Таблица преласка D-резе

Назив улаза D потиче од енглеске речи *data*, јер је то улаз на који доводимо вредност једнобитног податка коју желимо да сачувамо у колу. Када је $e = 1$, тада D реза ради у *транспарентном режиму*, јер се свака промена вредности на улазу D директно преноси на излаз Q (уз одређено кашњење). За $e = 0$, излаз Q задржава вредност која је претходно сачувана и не реагује на промене на улазу D.

4.2 Синхрона и асинхрона секвенцијална кола

У претходном поглављу видели смо да се стања секвенцијалних кола мењају реагујући на промене на улазу. Како се ове промене могу десити у било ком тренутку, следи да секвенцијално коло може променити своје стање у било ком тренутку, независно од промене стања других секвенцијалних кола у систему. Зато оваква кола зовемо и *асинхрона секвенцијална кола*.

Проблем са оваквим приступом је у отежаној комуникацији између секвенцијалних кола. Наиме, у типичном сценарију, излази једног секвенцијалног кола се повезују на улазе другог секвенцијалног кола (директно, или преко неког комбинаторног кола). Другим речима, улази једног секвенцијалног кола представљају функције излаза (па самим тим и тренутног стања) другог секвенцијалног кола. Ово значи да ће тренутак промене стања неког секвенцијалног кола бити одређен тренутком промене стања неког другог секвенцијалног кола, као и временом пропагације сигнала кроз жице и одговарајућа комбинаторна кола. С обзиром да различита кола имају различита кашњења, веома је тешко предвидети тренутак када ће које секвенцијално коло променити своје стање. Додатно, кашњење појединачних улаза неког секвенцијалног кола може бити различито, што значи да ни сви улази секвенцијалног кола не морају стићи истовремено. Како су секвенцијална кола веома осетљива на редослед промена вредности на улазима, различит редослед промене улазних сигнала може довести коло у различита стабилна стања. Из свега овога следи да је веома тешко дизајнирати иоле сложеније асинхроно секвенцијално коло тако да његов рад буде поуздан и предвидив.

Посматрајмо, на пример, једноставну D резу. Она има два улаза: D и e. Да бисмо уписали неку вредност у резу, потребно је да поставимо ту вредност на улаз D, а затим да укључимо улаз e. Притом, веома је важно да улаз D остане стабилан дуже од трајања јединице на улазу e. Уколико би се вредност улаза D променила пре него што се улаз e искључи, вредност сачувана у рези би била нова вредност на улазу D, што вероватно није оно

што смо желели. Нарочито критична ситуација је када се вредности D и e улаза мењају у приближно истом временском тренутку – тада је тешко предвидети да ли ће у рези бити сачувана стара или нова вредност улаза D. Укратко, за стабилан и предвидив рад D резе потребно је обезбедити следеће услове: 1) да се улаз e укључи у тачно одређеном тренутку када се улаз D већ стабилизовао на вредности коју је потребно запамтити; 2) да улаз e буде укључен довољно дуго да би реза могла да се стабилизује у новом стању; 3) да улаз D остане стабилан нешто дуже након искључивања улаза e, како бисмо обезбедили детерминистичност рада резе.

Из претходног разматрања можемо закључити да је за поуздан и предвидив рад асинхроног секвенцијалног кола неопходно на неки начин синхронизовати редослед промене, као и трајање улазних сигнала. Код асинхроних кола ова синхронизација се мора имплементирати *експлицитно*, подсредством додатних синхронизационих сигнала које кола међусобно размењују, или подсредством неке централизоване контроле која би генерисала контролне сигнале (попут e улаза D резе) у тачно одређеним тренутцима и са одговарајућим трајањима.

Међутим, реализација експлицитне синхронизације није ни мало једноставна. Потребно је обезбедити да се одговарајући синхронизациони и контролни сигнали укључују у тачно одређеним тренутцима, за шта је потребна сложена логика. Много једноставнији начин за синхронизацију секвенцијалних кола је тзв. *имплицитна синхронизација*. Код имплицитне синхронизације не постоје никакви експлицитни синхронизациони сигнали. Уместо тога, постоји јединствен синхронизациони сигнал који се дистрибуира свим секвенцијалним колима у систему и који називамо *часовник* (енгл. *clock*). Часовник је сигнал који наизменично у једнаком ритму мења своју вредност са 0 на 1 и обратно. Прелазак са 0 на 1 назива се *узлазни руб* или *узлазна ивица*, док се прелазак са 1 на 0 назива *силазни руб* или *силазна ивица* часовника. Временски период између две узлазне ивице (или две силазне ивице) зове се *циклус* часовника. Време трајања јединице називамо *позитивни део циклуса*, а време трајања нуле називамо *негативни део циклуса*. Позитивни и негативни део циклуса могу трајати једнако (тзв. *симетрични часовници*), а могу бити и различитих трајања (*асиметрични часовници*). Број циклуса у једној секунди назива се *фреквенција часовника*. На пример, уколико је фреквенција часовника једнака 1MHz, то значи да имамо милион циклуса у свакој секунди. Часовник се обично генерише помоћу кварцног осцилатора који креира равномерне електричне импулсе који се затим, подсредством одговарајуће електронике, претварају у правоугаони облик.

Секвенцијална кола се сада могу имплементирати тако да поред осталих својих улаза имају и додатни улаз за сигнал часовника. Овај улаз ћемо означавати са *clk*. Притом, имплементацијом се обезбеђује да се стања секвенцијалног кола могу мењати само у тачно одређеним тренутцима, типично на узлазним (или силазним) рубовима часовника. Следеће стање се одређује на основу текућег стања, као и вредности улаза у тренутку наиласка одговарајућег руба часовника, тј. при преласку часовника са 0 на 1 (или са 1 на 0). У осталим тренутцима циклуса коло не реагује на промене на улазима. Оваква секвенцијална кола називају се *синхрона секвенцијална кола*.

Код синхроних секвенцијалних кола је, дакле, временски локализована

осетљивост на улазне вредности. За разлику од асинхроних кола код којих је време континуална величина (јер се промене могу дешавати у било ком тренутку), код синхроних кола време је дискретна величина (јер имамо низ временских тренутака у којима се очитавају вредности на улазу и евентуално мења стање). Самим тим је готово у потпуности искључена зависност од редоследа промене вредности на улазима – битно је само да све вредности буду спремне у тренутку наиласка одговарајућег руба часовника. На пример, ако бисмо имали синхрону варијанту D резе (коју ћемо називати и *D флип-флоп*), тада би једино било битно да у тренутку наиласка одговарајућег руба часовника улаз e буде укључен, а на улазу D буде вредност коју желимо да упишемо. Потпуно је небитно који је од улаза D и e први постављен (пре наиласка руба часовника), као и који ће први променити своју вредност (након проласка руба часовника), јер коло ни пре, а ни након проласка руба часовника не реагује на промене на улазима.

Ако претпоставимо да су сва секвенцијална кола у систему синхронизована истим часовником, то значи да ће сва кола мењати своја стања у исто време. Да би синхрони систем радио исправно, потребно је обезбедити да сва синхрона секвенцијална кола у њему у тренутку наиласка одговарајућег руба часовника имају стабилизоване вредности на улазима. Те вредности ће бити очитане у тренутку наиласка руба часовника и на основу њих ће бити израчунато ново стање. Ово значи да је за исправан рад синхроног система потребно да будемо сигурни да је трајање циклуса часовника дуже од кашњења свих комбинаторних кола и жица које их повезују. Уколико је овај услов испуњен, тада нам није потребна никаква додатна синхронизација, јер сада сва секвенцијална кола *имплицитно* претпостављају да су им вредности на улазима исправне у тренутку наиласка одговарајућег руба часовника.

Предност синхроних кола је у томе што се много лакше дизајнирају тако да раде поуздано. Теоријски недостатак је у брзини, јер је брзина рада одређена фреквенцијом часовника, а она је одозго ограничена кашњењем најспоријег кола у систему. То значи да бржа кола морају да чекају сигнал часовника, иако су можда и раније могла да обезбеде свој излаз. Асинхрона кола немају овакво теоријско ограничење, јер се код њих промена стања дешава чим се за то испуне услови, тј. чим се промене вредности на улазима. Међутим, неопходност експлицитне синхронизације значајно отежава имплементацију асинхроних кола, а често их може учинити и споријим. Такође, поступак дизајна асинхроних секвенцијалних кола је знатно компликованији, јер је модел израчунавања сложенији. Са друге стране, рад синхроних кола се може много једноставније формално описати и проучавати, што олакшава дизајн и верификацију исправности сложених система. Због тога се у пракси синхрона кола најчешће користе, а готово сви савремени рачунари су у највећој мери имплементирани користећи синхроне компоненте. У наставку овог текста ћемо разматрати искључиво синхрона секвенцијална кола, осим уколико није другачије наглашено.

4.3 Флип-флопови

Резе које смо раније упознали су примери најједноставнијих асинхроних кола. Синхрони аналогон резе је коло које се назива *флип-флоп* (енгл. *flip-*

flop). Флип-флоп, као и реза, чува један бит података. Међутим, у случају флип-флопа постоји и додатни *clk* улаз који синхронизује промену његовог стања. Промене стања могу се догодити само у тренутку наиласка одговарајућег руба часовника.[2] Притом, флип-флопови се могу имплементирати тако да реагују било на узлазни, било на силазни руб часовника. Такође, могуће је дизајнирати и флип-флопове који реагују на оба руба часовника, али се та варијанта ређе користи. Као и код реза, постоје различите врсте флип-флопова (RS флип-флоп, D флип-флоп, JK флип-флоп, T флип-флоп). У наставку разматрамо ове основне типове флип-флопова и дајемо примере њихових имплементација. Напоменимо да постоје различити начини да се флип-флопови имплементирају. Ми ћемо се овде задржати на једној стандардној варијанти имплементације флип-флопова познатој под називом *господар-слуга* (енгл. *master-slave*).

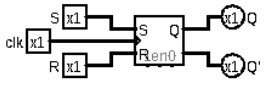
4.3.1 SR флип-флоп

SR флип-флоп има исте улазе и излазе као и SR реза, као и исту таблицу преласка[3] (табела 4.4).

S	R	Q	Q^{sled}
0	0	0	0
0	0	1	1
0	1	-	0
1	0	-	1
1	1	-	?

Табела 4.4: Таблица преласка SR флип-флопа

Једина разлика је у додатном синхронизационом *clk* улазу. На овај улаз се доводи сигнал часовника. Шематска ознака SR флип-флопа приказана је на слици 4.5 (обратити пажњу како је *clk* улаз означен троуглићем).



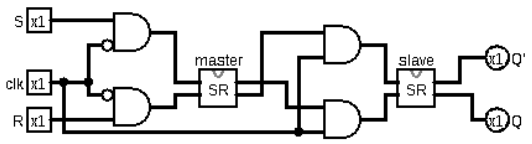
Слика 4.5: Шематска ознака SR флип-флопа

Када се на часовнику појави одговарајући руб, коло у том тренутку очитава вредности S и R улаза и на основу њихових вредности одређује да ли је потребно променити стање и на који начин. Вредности S и R улаза у осталим тренутцима циклуса се потпуно игноришу. На слици 4.6

[2] Напоменимо да у старијој литератури није постојала јасна дистинкција између реза и флип-флопова, већ се термин флип-флоп користио за сва ова кола. Притом, посебно се наглашавало да ли је у питању коло које реагује на ниво, тј. вредност синхронизационог сигнала (енгл. *level-triggered*) или коло које реагује на промену вредности синхронизационог сигнала (енгл. *edge-triggered*). У новијој литератури, јасно се разликују асинхроне варијанте кола, тј. резе (енгл. *latch*) и синхроне варијанте, тј. флип-флопови.

[3] И исти проблем — недозвољену комбинацију улаза $(S, R) = (1, 1)$.

приказујемо имплементацију SR флип-флопа који реагује на узлазној ивици часовника (господар-слуга имплементација).



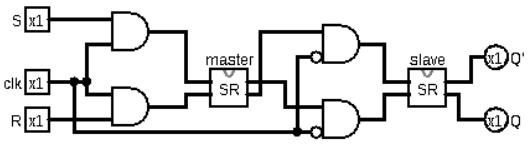
Слика 4.6: Имплементација SR флип-флопа

Имплементација се састоји из две SR резе. Лева реза је главна реза (или господар), док је десна реза подређена реза (или слуга). Резе су серијски повезане једна на другу, тако да се Q излаз главне резе шаље на S улаз подређене резе, док се излаз $\overline{Q}$ главне резе шаље на R улаз подређене резе. На овај начин се вредност која се чува у главној рези аутоматски прослеђује и уписује и у споредну резу (јер комбинација $(Q, \overline{Q}) = (1, 0)$ значи да на улазу подређене резе имамо $(S, R) = (1, 0)$, што значи да се у подређену резу уписује 1, слично и у обрнутој варијанти). Међутим, улази обе резе конролисани су конјункцијама које су повезане са сигналом часовника. Притом, конјункције на улазу главне резе повезане су на инвертовани сигнал часовника (обратити пажњу на кружиће на одговарајућим улазима конјункција), док су конјункције на улазу споредне резе повезане на неинвертовани сигнал часовника. Ово значи да је у току негативног дела часовника (када је $clk = 0$) главна реза „отворена", тј. улази S и R пролазе кроз конјункције и долазе на улазе главне резе, те се одговарајућа вредност уписује у главну резу и појављује се на њеним излазима. Међутим, конјункције на улазу споредне резе не пропуштају ове вредности, те је споредна реза „затворена", и чува раније уписану вредност (на њеним улазима су у том тренутку две нуле), тако да се вредност на излазу флип-флопа не мења. У тренутку наиласка узлазног руба (тј. промене часовника са 0 на 1) главна реза се затвара (њене конјункције престају да пропуштају улазе), али се споредна реза отвара, тј. претходно запамћена вредност у главној рези се уписује у споредну резу и појављује се на излазу флип-флопа. Током трајања позитивног дела циклуса часовника (када је $clk = 1$) све промене на улазима флип-флопа се игноришу, јер је главна реза затворена. Дакле, видимо да се стање флип-флопа (тј. оно што је на његовом излазу, а то је заправо стање подређене резе) може променити само у тренутку преласка часовника са 0 на 1, тј. на узлазном рубу.

Други начин да се опише принцип рада „господар-слуга" имплементације флип-флопа је да се примети да ни у једном тренутку не постоји директна веза улаза и излаза, јер је у сваком тренутку затворена или главна или споредна реза. Управо зато није могуће да флип-флоп реагује на промене вредности на улазима у произвољном тренутку, као што је то био случај код реза. Уместо тога, вредност која би требало да буде уписана у флип-флоп се акумулира у главној рези, а онда се пропагира на споредну резу у тренутку узлазног руба. На овај начин је обезбеђено да се евентуална промена стања дешава у тачно одређеном тренутку.

На слици 4.7 приказујемо варијанту SR флип-флопа који мења стање на силазној ивици. Разлика у односу на претходну имплементацију је само у

томе што се сада неинвертовани часовник доводи на конјункције на улазу главне резе, док се инвертовани часовник доводи на улазе конјункција испред споредне резе. Препуштамо читаоцу анализу рада овог кола.



Слика 4.7: SR флип-флоп који мења стање на силазној ивици часовника

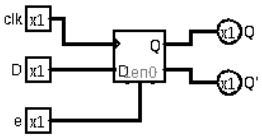
4.3.2 D флип-флоп

D флип-флоп има исте улазе и исту семантику као и D реза. Таблица преласка је иста као и у случају D резе (табела 4.5).

D	e	Q	Q^{sled}
-	0	0	0
-	0	1	1
0	1	-	0
1	1	-	1

Табела 4.5: Таблица преласка D флип-флопа

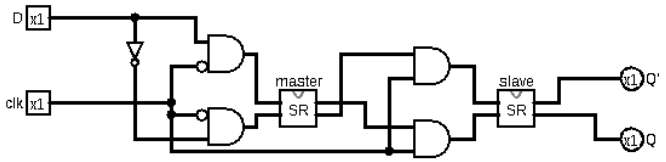
Додатни улаз *clk* омогућава синхронизацију помоћу часовника. Шематска ознака D флип-флопа је дата на слици 4.8.



Слика 4.8: Шематска ознака D флип-флопа

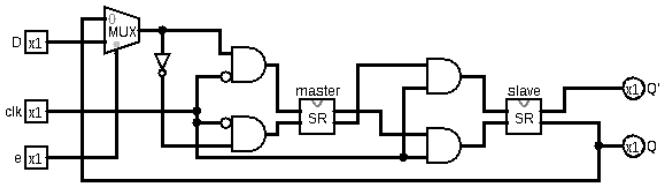
Као и у случају D резе, најједноставнији начин да се од SR флип-флопа добије D флип-флоп је да се дода негација између S и R улаза. Оваква „господар-слуга" имплементација је приказана на слици 4.9 и функционише на потпуно исти начин као и у случају SR флип-флопа, само што сада има само један D улаз чија се вредност уписује у стање флип-флопа.

Проблем са оваквом имплементацијом је што нема e улаз, па је такав флип-флоп принуђен да мења своје стање у сваком циклусу, у складу са вредношћу на улазу D. Другим речима, флип-флоп није у стању да чува своју вредност током већег броја циклуса. Имплементација која укључује и e улаз дата је на слици 4.10. Основна разлика је у додатном 2-1 мултуплексеру који обезбеђује да комбинација на S и R улазима буде она која одговара упису жељене вредности у флип-флоп. Када је на e улазу 0, тада је



Слика 4.9: Имплементација D флип-флопа без *enable* улаза

по таблици преласка потребно задржати претходну вредност. Због тога мултиплексер пропушта на излаз вредност која се на његов горњи улаз доводи повратном спрегом са излаза флип-флопа, чиме се у флип-флоп уписује иста вредност коју он тренутно чува (дакле, потврђујемо тренутну вредност). Када је на e улазу 1, тада је потребно вредност са улаза D сачувати у флип-флопу. Зато тада мултиплексер са свог доњег улаза пропушта вредност D улаза која се памти у флип-флопу.



Слика 4.10: Имплементација D флип-флопа

Напоменимо да се у литератури чешће јавља верзија D флип-флопа без e улаза. Додатни мултиплексер се у том случају увек може екстерно додати како би се обезбедила одговарајућа функционалност чувања вредности. Ми ћемо се, ипак, држати ове друге имплементације, тј. претпостављаћемо да, ако није другачије речено, D флип-флоп увек има и e улаз.

4.3.3 JK флип-флоп

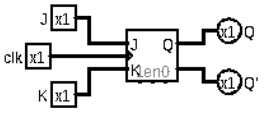
JK флип-флоп претставља другачији приступ решавању проблема непотпуне семантике SR флип-флопа. Сетимо се да је код SR флип-флопа главни проблем био то што је за две јединице на улазу понашање флип-флопа било недефинисано. Овде је идеја да покушамо да и за ову комбинацију на улазу дефинишемо неко смислено понашање флип-флопа. Код JK флип-флопа постоје два улаза, означени као J и K. Улаз J има исту улогу коју је имао улаз S код SR флип-флопа, а то је постављање стања на 1, док улаз K има исту улогу као R улаз код SR флип-флопа, а то је постављање стања на 0. Дакле, комбинације улаза $(J,K) = (0,0)$, $(J,K) = (0,1)$, $(J,K) = (1,0)$ имају потпуно исту семантику као код SR флип-флопа. Комбинација улаза $(J,K) = (1,1)$ која је код SR флип-флопа била недефинисана, овде има новодефинисану улогу – да инвертује вредност сачувану у флип-флопу. Таблица преласка JK флип-флопа је дата у табели 4.6.

Шематска ознака JK флип-флопа дата је на слици 4.11, а „господар-слуга" имплементација JK флип-флопа који реагује на узлазни руб

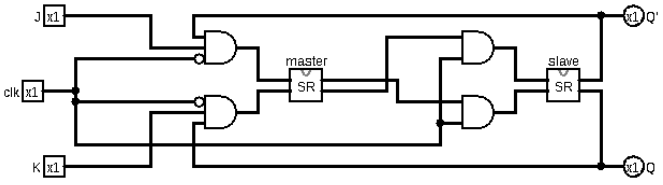
J	K	Q	Q^{sled}
0	0	0	0
0	0	1	1
0	1	-	0
1	0	-	1
1	1	0	1
1	1	1	0

Табела 4.6: Таблица преласка JK флип-флопа

часовника дата је на слици 4.12.



Слика 4.11: Шематска ознака JK флип-флопа



Слика 4.12: Имплементација JK флип-флопа

Као што видимо, имплементација JK флип-флопа је веома слична имплементацији SR флип-флопа. Једина разлика су две додатне повратне спреге које се са излаза Q и Q' враћају назад на конјункције које се налазе испред главне резе. Ове повратне спреге онемогућавају да се на S и R улазима главне резе појаве две јединице, јер су излази Q и Q' увек супротних вредности, па ће једна од повратних спрега увек бити 0. У случају да је стање флип-флопа (тј. вредност излаза Q) једнако 0, тада ће доња повратна спрега бити 0, па вредност улаза K неће моћи да прође до R улаза главне резе. То значи да ће сваки захтев за ресетовањем бити игнорисан, али то није суштинско ограничење, с обзиром да је флип-флоп већ ресетован. Са друге стране, вредност горње повратне спреге је једнака 1, па ће вредност J улаза моћи да прође до S улаза главне резе (наравно, у негативном делу циклуса часовника), тако да ће сетовање бити могуће. У случају да је стање флип-флопа $Q = 1$, ситуација је обрнута, тј. ресетовање је могуће, а сетовање не, што опет није суштинско ограничење, с обзиром да је флип-флоп и онако постављен на јединицу. Најзанимљивије је понашање у случају када су две јединице на улазима. Тада ће због поменутих повратних спрега само једна од те две јединице проћи кроз конјункције и доћи на одговарајући улаз главне резе. То ће управо бити она јединица која доводи

до промене вредности стања флип-флопа, с обзиром да повратне спреге дозвољавају пролаз само оној вредности која има тенденцију да промени вредност флип-флопа (J ако је $Q = 0$, а K ако је $Q = 1$). Ефекат је да ће се вредност стања флип-флопа инвертовати на првом следећем узлазном рубу часовника.

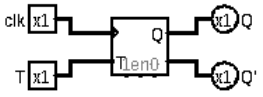
4.3.4 Т флип-флоп

Т флип-флоп у суштини и не представља посебну врсту флип-флопа, јер је у питању JK флип-флоп чији су улази спојени у један улаз (који обично означавамо са T). Уколико је овај улаз једнак нули, тада имамо две нуле на J и K улазима, па ће флип-флоп чувати текуће стање. У случају да је на T улазу јединица, тада имамо две јединице на J и K улазима, па ће се стање флип-флопа инвертовати. Дакле, спајањем улаза у један ми смо ограничили семантику JK флип-флопа тако да можемо или да чувамо претходно стање или да га инвертујемо. Таблица преласка Т флип-флопа дата је у табели 4.7.

T	Q	Q^{sled}
0	0	0
0	1	1
1	0	1
1	1	0

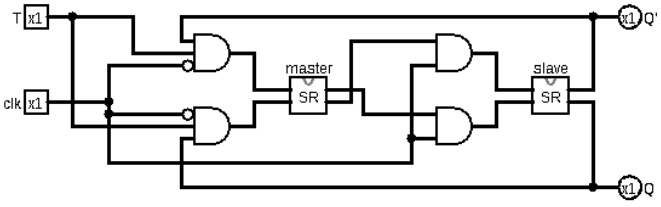
Табела 4.7: Таблица преласка Т флип-флопа

Шематска ознака Т флип-флопа дата је на слици 4.13.



Слика 4.13: Шематска ознака Т флип-флопа

Имплементација „господар-слуга” дата је на слици 4.14.



Слика 4.14: Имплементација Т флип-флопа

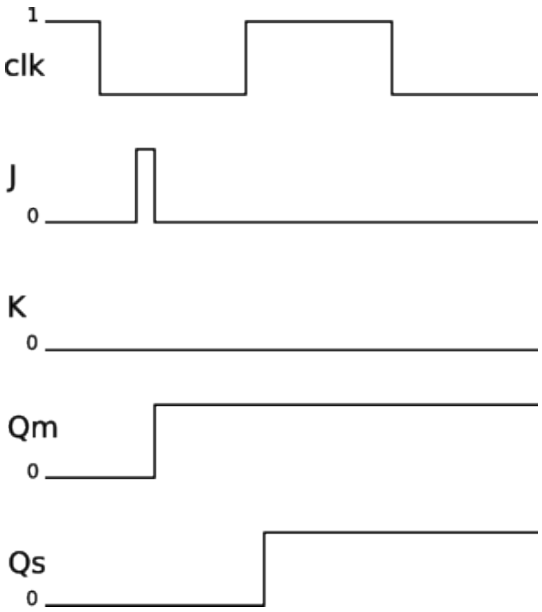
Приметимо да би асинхроне верзије JK и Т флип-флопова (тј. као резе) биле нестабилна кола, јер би у случају инверзије излаза (за $(J, K) = (1, 1)$,

односно за $T = 1$) непрекидно прелазили из стања у стање, без могућности да се стабилизују. Међутим, у случају синхроних кола, прелази се дешавају само на одговарајућем рубу часовника, тако да ови флип-флопови у тим случајевима раде у *toggle* режиму, тј. наизменично мењају своје стање у сваком циклусу часовника (ово је згодно, нпр. за бројаче, као што ћемо видети касније).

4.3.5 Проблем „хватања јединице”

Проблем „хватања јединице” (енгл. *1s catching problem*) је проблем који се у одређеним околностима може манифестовати код JK флип-флопа. Наиме, сетимо се да би синхрона кола требало да очитавају своје улазе само у тренутку наиласка узлазног (или силазног) руба часовника и на основу вредности улаза у том тренутку би требало одредити ново стање у које коло прелази. Са друге стране, вредности које се евентуално појављују на улазима у осталим тренутцима циклуса не би требало ни на који начин да утичу на рад кола. Међутим, хајде да мало боље анализирамо „господар-слуга” имплементацију JK флип-флопа дату на слици 4.12. Подсетимо се да ова имплементација реагује на узлазну ивицу, али то није суштински битно.

Посматрајмо временски дијаграм сигнала дат на слици 4.15.



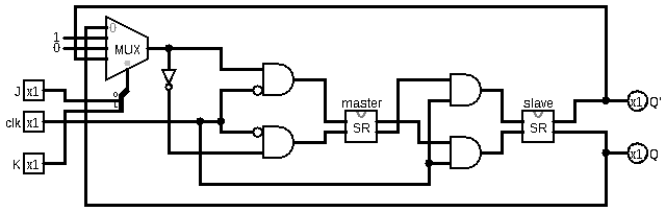
Слика 4.15: Хватање јединице приказано на временском дијаграму

Током негативног дела циклуса, док је главна реза отворена, на J улазу се појављује краткотрајна јединица. Ова јединица поставља вредност главне резе на јединицу (сигнал Q_m на дијаграму). Ова вредност је сада акумулирана у главној рези и на узлазној ивици часовника она пролази кроз конјункције и уписује се у споредну резу, чиме се стање флип-флопа

мења (сигнал Q_s на дијаграму). Међутим, то не би требало да се деси, с обзиром да у тренутку узлазне ивице часовника на улазима имамо комбинацију $(J, K) = (0, 0)$ која би требало да сачува претходно стање. Дакле, краткотрајна јединица је нехотице „ухваћена" што је утицало на вредност наредног стања, иако не би требало. Слична ствар би се десила и када би тренутно стање флип-флопа било 1, а када би се на улазу K појавила краткотрајна јединица током негативног дела часовника. Појава оваквих краткотрајних импулса на улазима секвенцијалних кола је веома честа, с обзиром да се улазне вредности добијају са излаза других, најчешће комбинаторних кола, којима је потребно извесно време да стабилизују своје излазне вредности.

Овај проблем се манифестује и код SR и T флип-флопова, али не и код D флип-флопа. Наиме, проблем настаје зато што се, након што се та краткотрајна јединица изгуби, улази враћају на $(J, K) = (0, 0)$, што главној рези (погрешно) говори да запамти то новоформирано стање, које се на силазном рубу само пропагира ка споредној рези. Са друге стране, код D флип-флопа, захваљујући негацији између улаза, увек имамо „активну" комбинацију на улазима главне SR резе: $(0, 1)$ или $(1, 0)$. Ово значи да током одговарајућег дела циклуса (позитивног у случају флип-флопа који реагује на силазну ивицу, а негативног у супротном) вредност главне резе D флип-флопа све време прати вредност D улаза: краткотрајна промена на њему ће само краткотрајно променити вредност главне резе, али ће се након стабилизације D улаза и вредност главне резе стабилизовати на исту вредност. Отуда ће једино бити важно шта се на улазу затекло у тренутку одговарајуће промене вредности часовника.

Отуда је један од начина да се проблем хватања јединице реши је да се JK (односно SR или T) флип-флоп имплементира по угледу на D флип-флоп (слика 4.16).

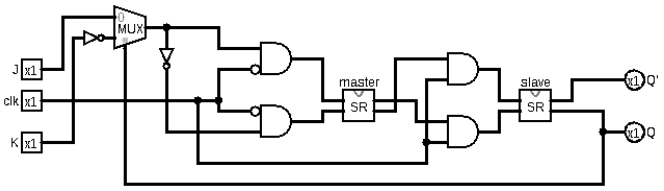


Слика 4.16: Елиминација хватања јединице код JK флип-флопа

У основи приказане имплементације имамо D флип-флоп који на улазу уместо 2-1 мултиплексера има 4-1 мултиплексер који разликује четири могуће комбинације на улазима J и K. За комбинацију $(J, K) = (0, 0)$ мултиплексер пропушта тренутну вредност флип-флопа (као и код D флип-флопа). У случају комбинације $(J, K) = (1, 0)$ мултиплексер пропушта јединицу која се уписује у флип-флоп (сетовање). Комбинација $(J, K) = (0, 1)$ врши ресетовање (јер се на излаз мултиплексера пропушта нула). Најзад, за комбинацију $(J, K) = (1, 1)$ мултиплексер пропушта Q', па се у флип-флоп уписује инвертована вредност. Како је десно од мултиплексера заправо SR флип-флоп чији су улази повезани негацијом, као и код D флип-

флопа вредност главне резе пратиће вредност која се налази на излазу мултиплексера, тако да привремене краткотрајне промене J и K улаза могу само привремено да промене стање главне резе, али ће се њена вредност вратити на старо када тај кракотрајни импулс прође. Као и код D флип-флопа, рачунаће се само оно што се на улазима затекло у тренутку наиласка одговарајућег руба часовника.

Нешто ефикаснија имплементација која уместо 4-1 мултиплексера користи 2-1 мултиплексер приказана је на слици 4.17. Овог пута се на селекциони улаз мултиплексера доводи вредност Q са излаза флип-флопа. Идеја је једноставна. Када је $Q = 0$, тада се на излаз мултиплексера прослеђује вредност J улаза, чиме се постиже да се за комбинације $(J, K) = (1, 0)$ (сетовање) и $(J, K) = (1, 1)$ (инвертовање) вредност флип-флопа поставља на 1, док се за комбинације $(J, K) = (0, 0)$ (чување тренутне вредности) и $(J, K) = (0, 1)$ (ресетовање) вредност флип-флопа поставља на 0. Ако је $Q = 1$, тада се кроз мултиплексер пропушта вредност $\overline{K}$, чиме се за комбинације $(J, K) = (0, 0)$ и $(J, K) = (1, 0)$ вредност флип-флопа поставља на 1, док се за комбинације $(J, K) = (0, 1)$ и $(J, K) = (1, 1)$ вредност флип-флопа поставља на 0. Овакво понашање одговара таблици преласка JK флип-флопа.



Слика 4.17: Елиминација хватања јединице код JK флип-флопа (са 2-1 мултиплексером)

На сличан начин се може решити проблем хватања јединице и код SR и код T флип-флопа. Цена оваквог решења је додатни мултиплексер. Постоје и други начини решавања овог проблема који подразумевају елиминацију појава нестабилности на улазима које би могле да узрокују овакво понашање, али њихово разматрање излази изван оквира овог текста.

4.3.6 Време поставке и време задржавања

Код комбинаторних кола, најзначајнији параметар је било кашњење, тј. време пропагације сигнала од улаза до излаза. Ову карактеристику имају и секвенцијална кола. Конкретно, код синхроних секвенцијалних кола, *време пропагације* је време које прође од тренутка наиласка одговарајућег руба часовника t_0 до тренутка када се нова вредност појави на излазима кола. Поред овог времена, од значаја су још два временска параметра: време поставке и време задржавања.

Време поставке (енгл. *setup time*) t_s је дужина временског интервала $(t_0 - t_s, t_0]$ пре тренутка наиласка руба часовника t_0 у коме улази не смеју да мењају своје вредности. Другим речима, улази кола морају бити стабилизовани најкасније до тренутка $t_0 - t_s$. У супротном би било тешко

детерминистички предвидети понашање кола. На пример, у господар-слуга имплементацији флип-флопа, ако би се улази променили прекасно, тј. недовољно пре тренутка t_0, постојала би могућност да те нове вредности не успеју да се „провуку" кроз конјункције које контролишу улазе у главну резу, јер ће оне врло брзо почети да се затварају. Да бисмо били сигурни да ће вредност коју желимо да упишемо у резу заиста бити запамћена, морамо обезбедити да се она стабилизује на време, пре него што конјункције почну да се затварају.

Време задржавања (енгл. *hold time*) t_h је дужина временског интервала $[t_0, t_0+t_h)$ након наиласка руба часовника у коме се вредности улаза не смеју мењати. Другим речима, вредности на улазима морају остати непромењене најраније до тренутка $t_0 + t_h$. На примеру господар-слуга имплементације флип-флопа, ако би вредности на улазима биле промењене прерано, могло би се догодити да се те нове вредности провуку кроз конјункције које контролишу главну резу (јер се оне не могу затворити тренутно) те да у флип-флопу буде запамћена нова вредност на улазу, уместо старе.

Сетимо се да смо раније рекли да би синхрона секвенцијална кола требало да узимају у обзир вредности улаза само у тачно одређеном тренутку, тј. у тренутку наиласка руба часовника t_0. У пракси видимо да ова кола никада нису идеална, тако да уместо једног тренутка t_0 постоји известан интервал $(t_0 - t_s, t_0 + t_h)$ у коме улазне вредности могу утицати на рад кола. Отуда није добро мењати вредности на улазима у том интервалу, јер би се у том случају поставило питање које ће вредности улаза бити узете у обзир (старе или нове). На срећу, код савремених електронских кола временски параметри t_s и t_h су обично веома мали. Уобичајено је да произвођачи чипова наводе ове параметре у спецификацији својих производа, како би их они који уграђују те чипове у своје уређаје могли узети у обзир.

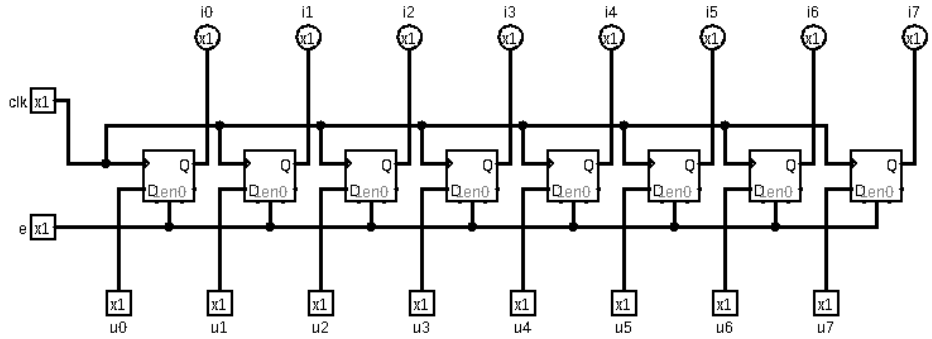
4.4 Регистри

Регистар дужине n (или n-битни регистар) је коло које чува један n-битни бинарни број. Регистри се обично праве од D флип-флопова. Основне операције са регистрима су читање и упис. Пример имплементације регистра дат је на слици 4.18.

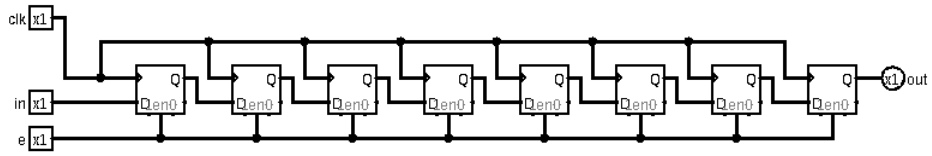
Приметимо да су сви D флип-флопови повезани на исте clk и e сигнале. Такође, сви флип-флопови су конструисани тако да реагују на исти руб часовника. Када је e сигнал нула, регистар чува вредност коју је имао и у претходном циклусу, док се за $e = 1$ у регистру памте вредности које се у тренутку одговарајућег руба часовника нађу на улазима $u0 - u7$. Читање вредности сачуване у регистру се у сваком тренутку може обавити очитавањем вредности на излазима $I0 - I7$.

Поред читања и уписа, регистри могу подржавати и друге операције. Једна од најчешћих операција је померање садржаја регистра за једно место у лево или у десно (померање у десно може бити логичко или аритметичко). Регистри који имају ову могућност зову се померачки регистри (енгл. *shift*). На слици 4.19 дат је једноставан померачки регистар.

Код овог једноставног померачког регистра излаз сваког флип-флопа се повезује на улаз следећег. На овај начин се обезбеђује да се у сваком



Слика 4.18: 8-битни регистар

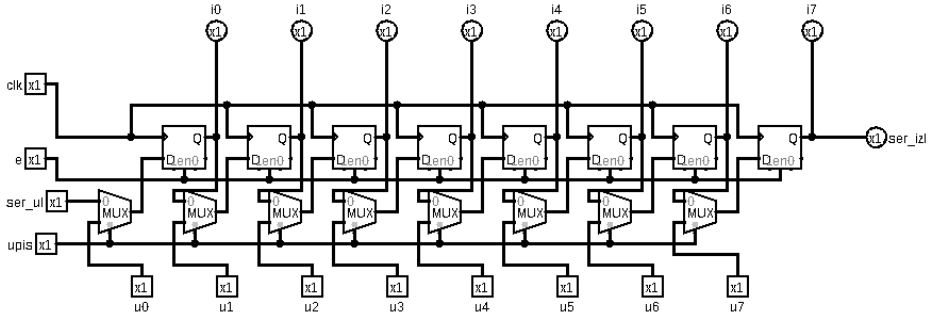


Слика 4.19: Померачки регистар

циклусу (на одговарајућем рубу часовника) текућа вредност неког флип-флопа премешта у флип-флоп десно од њега, док у сам тај флип-флоп долази вредност из флип-флопа лево од њега. С обзиром да је на нашој слици најлевљи бит бит најмање тежине, ово значи да овај регистар подржава операцију померања у лево (иако на слици делује као да се помера у десно, због обрнутог распореда битова). Упражњена места приликом померања попуњавају се вредношћу серијског улаза in, док се битови који се истискују појављују на серијском излазу out. Другим речима, овакав регистар подржава серијски улаз и серијски излаз, при чему излаз за улазом касни n циклуса, при чему је n дужина регистра.

У пракси, обично желимо да померачки регистри, поред операције померања (тј. серијског улаза и излаза) подржавају и уобичајени, паралелни упис и читање. Такав померачки регистар је приказан на слици 4.20.

Регистар има како серијске, тако и паралелне улазе и излазе. Посебан контролни сигнал $upis$ одређује да ли ћемо у том циклусу вршити померање (тј. упис са серијског улаза) или паралелни упис. У ту сврху се користе 2-1 мултиплексери који на улаз наредног флип-флопа могу преусмерити или излаз претходног (за $upis = 0$) или одговарајући бит паралелног улаза (за $upis = 1$). Овакви регистри омогућавају да се сви битови упишу одједном, а да се затим та вредност помера у регистру. Типична примена оваквих регистара је у имплементацији алгоритама множења (попут Бутовог алгоритма). Друга примена је у конверзији између серијског и паралелног трансфера. Наиме, код серијског трансфера, податак се преко једне линије преноси бит по бит. Код паралелног трансфера сви битови података се преносе одједном, путем одговарајућег броја паралелних линија.



Слика 4.20: Померачки регистар са паралелним улазом и излазом

Уколико је потребно да се податак који је до неке тачке преношен паралелно надаље преноси серијском линијом, можемо пристиглу реч уписати у померачки регистар (преко паралелног улаза), а затим у n наредних циклуса истиснути бит по бит ове речи кроз серијски излаз на одговарајућу серијску линију. Слично би било и у случају обрнуте конверзије.

Напоменимо да је код померачких регистра од изузетног значаја за исправан рад да време пропагације појединачног флип-флопа буде веће од времена задржавања наредног флип-флопа. У супротном би се нова вредност флип-флопа могла провући до улаза наредног флип-флопа пре његовог затварања, тј. било би могуће да бит у једном циклусу прескочи више битских места. Ово је најчешће испуњено, јер је време задржавања обично веома мало, док је време пропагације знатно веће.

4.5 Меморије

Меморија представља низ *меморијских локација*, при чему свака меморијска локација може да чува низ битова фиксне дужине. Уколико имамо m меморијских локација од којих се свака састоји од n битова, тада кажемо да је меморија димензије $m \times n$. Свака од m меморијских локација има своју *адресу*. Адресе су неозначени цели бројеви од 0 до $m - 1$. Дакле, меморијске локације су најмање јединице меморије којима се може независно непосредно приступати (навођењем адресе). Меморије омогућавају две основне операције: *читање* вредности из локације са дате адресе и *упис* дате вредности у локацију на датој адреси.

Капацитет меморије се мери у *бајтовима* (енгл. *byte*). Један бајт се састоји из осам битова (тј. представља један осмоцифрени бинарни број) и типично представља најмању јединицу мере капацитета. Један *килобајт* (у ознаци KB) садржи $2^{10} = 1024$ бајтова. Слично, један *мегабајт* (у ознаци MB) садржи 1024 килобајта. Даље, по истом принципу, један *гигабајт* (GB) садржи 1024 мегабајта, један *терабајт* (TB) садржи 1024 гигабајта, и тд.

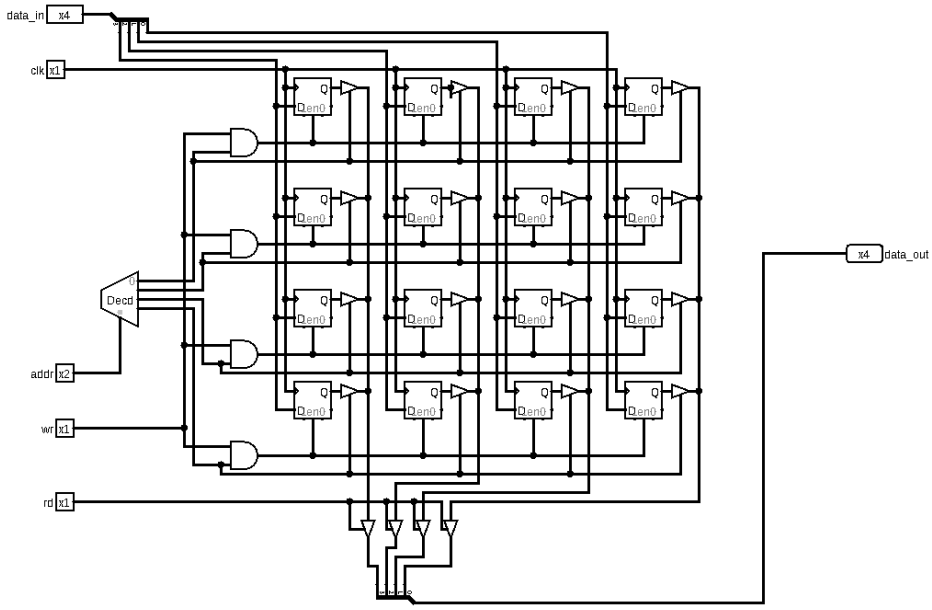
Напоменимо да адресибилне меморијске локације могу, али не морају одговарати бајтовима. Уколико то јесте случај, тј. ако је $n = 8$, тада кажемо да је меморија *бајт-адресибилна*, тј. да сваки бајт има посебну адресу. Ипак, ово не мора да буде случај, па могу постојати меморије чије су адресибилне

локације веће или мање од једног бајта.

Меморије могу бити *синхроне* и *асинхроне*. Асинхроне меморије нису синхронизоване часовником, већ се њихов рад мора синхронизовати на неки другачији начин. Структура асинхроних меморија је једноставнија, али је теже обезбедити њихов поуздан рад. Због тога се данас чешће користе синхроне меморије.

4.5.1 Синхроне меморије

Пример једноставне синхроне меморије 4×4 дат је на слици 4.21.



Слика 4.21: Пример синхроне меморије 4×4

На улаз *addr* доводи се адреса. Уколико имамо k адресних битова, тада имамо укупно $m = 2^k$ адреса. У нашем примеру, $m = 4$, па је број адресних линија $k = log_2(m) = 2$, тј. адресни улаз је двобитни. Такође, постоји n-битни (у нашем примеру 4-битни) улаз за податке *data_in*. На овај улаз доводи се податак који желимо да упишемо на дату адресу у случају операције уписа. За операцију читања постоји n-битни излаз *data_out* на који се, шаље вредност меморијске локације са адресе коју желимо да очитамо. Постоје и два контролна једнобитна улаза: улаз *rd* активира излаз за податке и омогућава очитавање вредности изабране меморијске локације (када је $rd = 0$, тада је на излазу вредност високе импедансе), док улаз *wr* омогућава упис вредности са улаза *data_in* у одговарајућу локацију на узлазном рубу часовника.

Меморија је структурно организована као матрица флип-флопова димензије $m \times n$, где свака врста представља једну меморијску локацију. Адреса се декодира уз помоћ декодера, при чему свака излазна линија декодера активира одговарајућу врсту матрице. У случају операције уписа,

излазна линија декодера се користи да активира e улазе свих флип-флопова у тој врсти матрице. Притом, e улази се активирају само ако је $wr = 1$, што је постигнуто додатним конјункцијама за сваку врсту матрице. Са друге стране, читање се контролише баферима са три стања који се налазе на излазима флип-флопова: свака излазна линија декодера активира бафере са три стања на излазима свих флип-флопова у одговарајућој врсти, те се управо те вредности шаљу на излаз. Додатни бафери са три стања при дну шеме омогућавају да се вредности битова са изабране адресе прослеђују на излаз само када је $rd = 1$.

Овај принцип се може уопштити на меморије произвољне величине. У случају меморије са $m = 2^k$ регистара потребно је, поред самих флип-флопова и један декодер k-на-2^k. Реализација овог декодера постаје све комплекснија како k расте, а повећава се и његово кашњење. Такође, повећава се величина саме матрице, што повећава растојања између компоненти и успорава пропагацију сигнала. Због свих ових разлога, веће меморије су по правилу спорије од мањих.

4.5.2 Конструкција већих меморија помоћу мањих

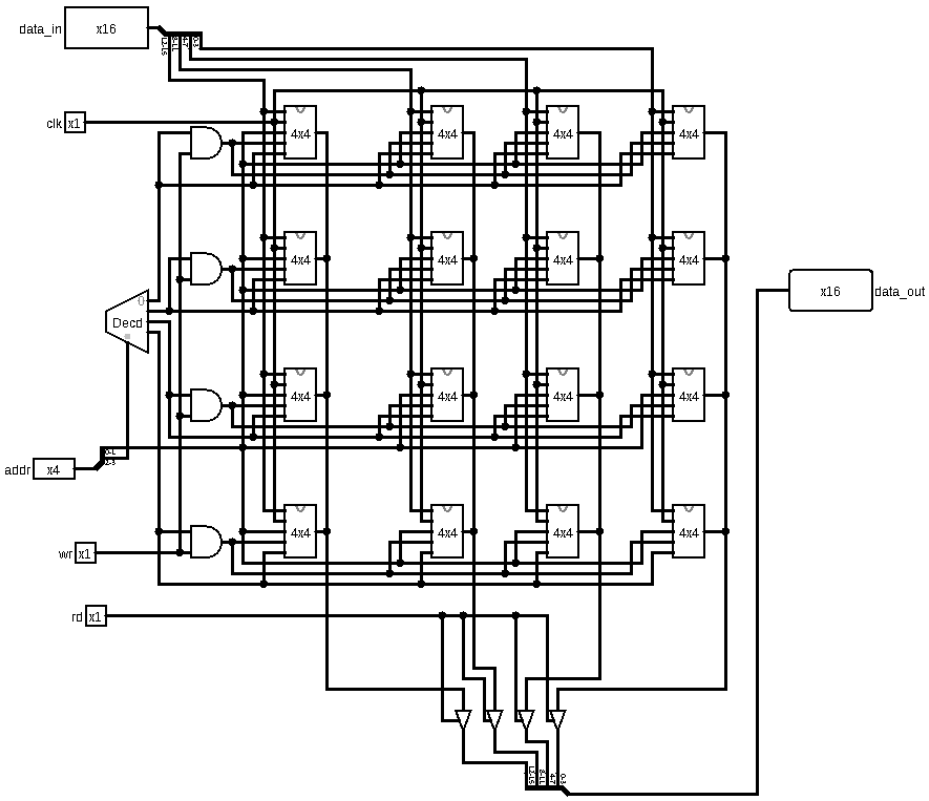
Уколико су нам доступни меморијски блокови неке дате величине $m \times n$, тада њих можемо искористити да конструишемо већу меморију. Пример оваквог дизајна дат је на слици 4.22.

Слика приказује меморију 16×16 која се састоји из матрице меморијских блокова 4×4. Матрица има 4 врсте и 4 колоне. Уопште, ако меморијске блокове $m \times n$ поређамо у матрицу $p \times q$, добићемо меморију $mp \times nq$. Свака врста матрице представља следећих m адреса, па се повећавањем броја врста добија меморија са више меморијских локација. Са друге стране, свака колона матрице продужава меморијске локације за n битова, па се повећавањем броја колона могу повећати дужине меморијских локација. Меморијске локације овакве меморије се састоје из низова одговарајућих меморијских локација у меморијским блоковима у истој врсти. Прецизније, све меморијске локације на адреси i у меморијским блоковима у врсти k чине једну меморијску локацију дужине nq са адресом $k \cdot m + i$.

Меморијски блокови се повезују на исти начин као што су се појединачни флип-флопови повезивали у основној структури меморије. nq-битни улаз in се дели на q делова од по n битова и сваки од њих се прослеђује на улазе блокова у одговарајућим колонама. На овај начин се битови одговарајућих тежина дистрибуирају ка одговарајућим деловима меморијских локација. На сличан начин се n-битни излази блокова из сваке колоне прослеђују на одговарајуће битове излаза, чиме добијамо један nq-битни излаз.

Адресирање се врши двостепено. Адреса се састоји из $\log_2(p) + \log_2(m)$ битова. Виших $\log_2(p)$ битова селектују једну од p врста (помоћу декодера на слици) док се нижих $\log_2(m)$ битова прослеђују на адресне улазе свих блокова у матрици и помоћу њих се врши избор једне од m локација у сваком од блокова у изабраној врсти. Приметимо да се унутар меморијских блокова налазе декодери са $\log_2(m)$ селекционих улаза, па се у суштини овде врши слагање једноставнијих декодера у сложеније.

Меморија се може слагати на више начина. На пример, меморија 16×16 се може добити и тако што се меморије 8×8 слажу у матрицу 2×2, али и тако што се меморије 4×1 слажу у матрицу 4×16. Начин слагања је у



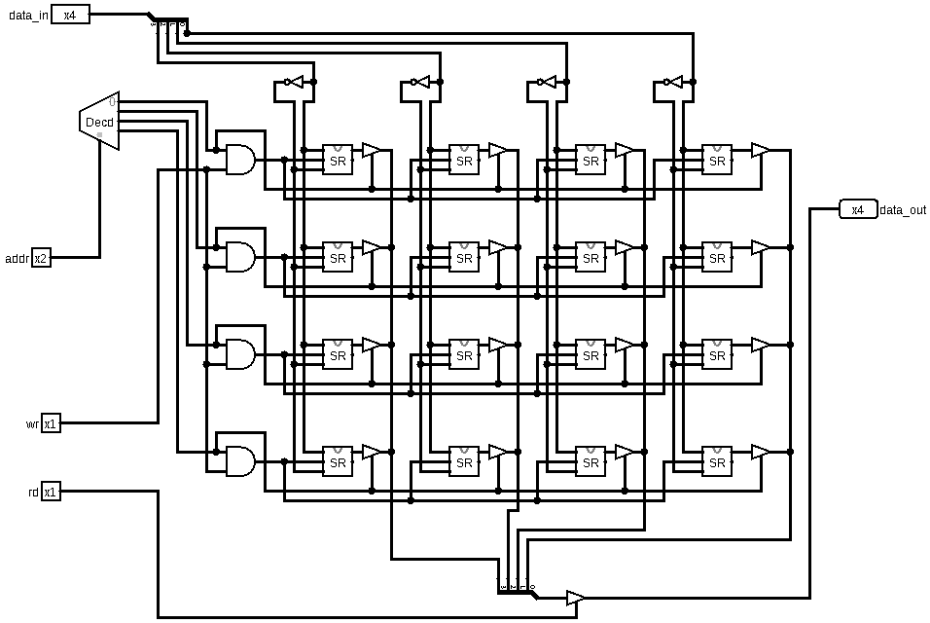
Слика 4.22: Сихнрона меморија 16×16 реализована помоћу меморија 4×4

пракси одређен тиме којих су димензија меморијски блокови који су нам доступни.

Уколико је потребно направити још веће меморије, могуће је вршити слагање на више нивоа, при чему се слагање увек врши по истом принципу као што је овде описано.

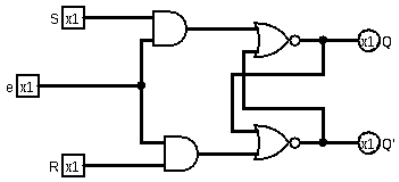
4.5.3 Асинхроне меморије

Асинхроне меморије се могу конструисати на потпуно исти начин као и синхроне, с тим што се уместо флип-флопова користе резе. Стога бисмо ово могли оставити читаоцу за вежбу. Ипак, у овом одељку ћемо се мало детаљније позабавити дизајном асинхроних меморија, из угла оптимизације по питању броја потребних капија и транзистора. Наиме, ако бисмо имали асинхрону меморију димензије $m \times n$, она би имала mn реза у себи. Ако узмемо да се свака D реза састоји из 5 капија (два NOR кола, два AND кола и једно NE коло), укупан број гејтова за матрицу меморије је $5mn$ (не рачунамо гејтове потребне за изградњу декодера). Прва оптимизација би се могла састојати у томе да се уместо D реза користе SR резе, при чему ћемо за сваку колону имати по једно NE коло које је заједничко за све резе у тој колони, као на слици 4.23.



Слика 4.23: Асинхрона меморија 4×4

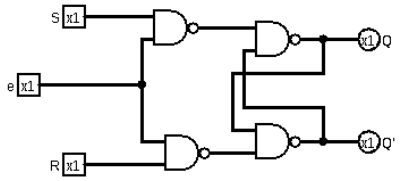
На овој слици, свака од SR реза има додатни e улаз, тј. изгледа као D реза из које је избачено NE коло (слика 4.24).



Слика 4.24: SR реза са додатним *enable* улазом

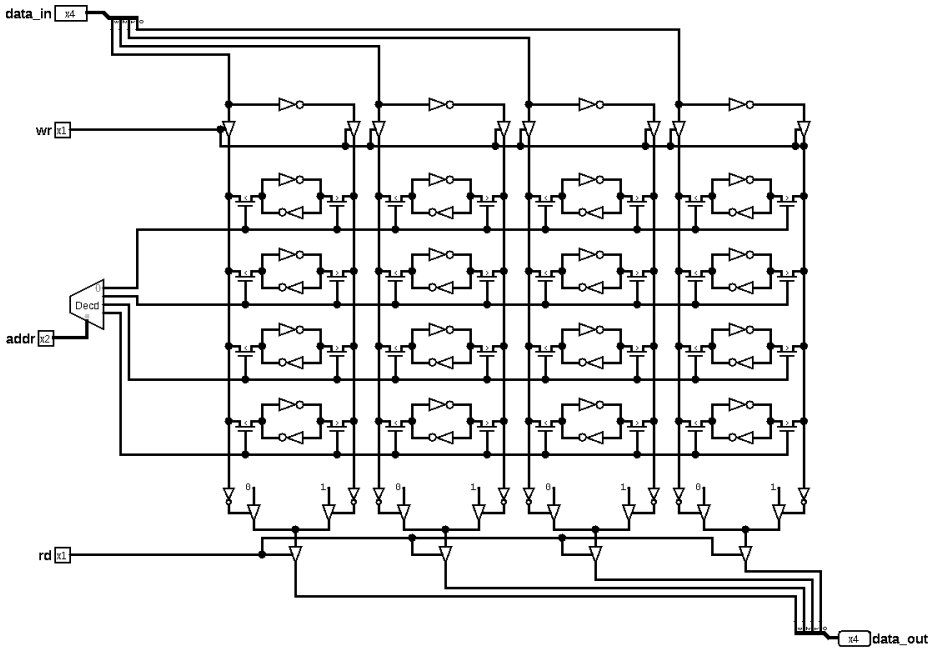
Уместо да свака реза има своје NE коло, имамо по једно NE коло у свакој колони. Вредност одговарајућег бита са улаза се једном негира, а онда се та вредност и њена негација спроводе респективно на S и R улазе SR резе у селектованој врсти. На овај начин уместо да имамо mn NE кола, имаћемо само n NE кола. Дакле, повећање броја меморијских локација не повећава број NE кола у меморији. Укупан број гејтова је сада $4mn + n$.

Додатна оптимизација на нивоу сваке SR резе се може постићи смањењем потребног броја транзистора за реализацију резе. Горња реализација резе захтева 20 транзистора (јер конјункција захтева 6 транзистора, а NOR коло захтева 4 транзистора у CMOS технологији). Алтернативно, SR резу смо могли реализовати и на начин приказан на слици 4.25. Читаоцу остављамо да провери да су ове две имплементације међусобно еквивалентне. Ова алтернативна реализација захтева 16



Слика 4.25: SR реза са додатним *enable* улазом реализована помоћу НИ кола

транзистора, јер се састоји из 4 NAND кола.



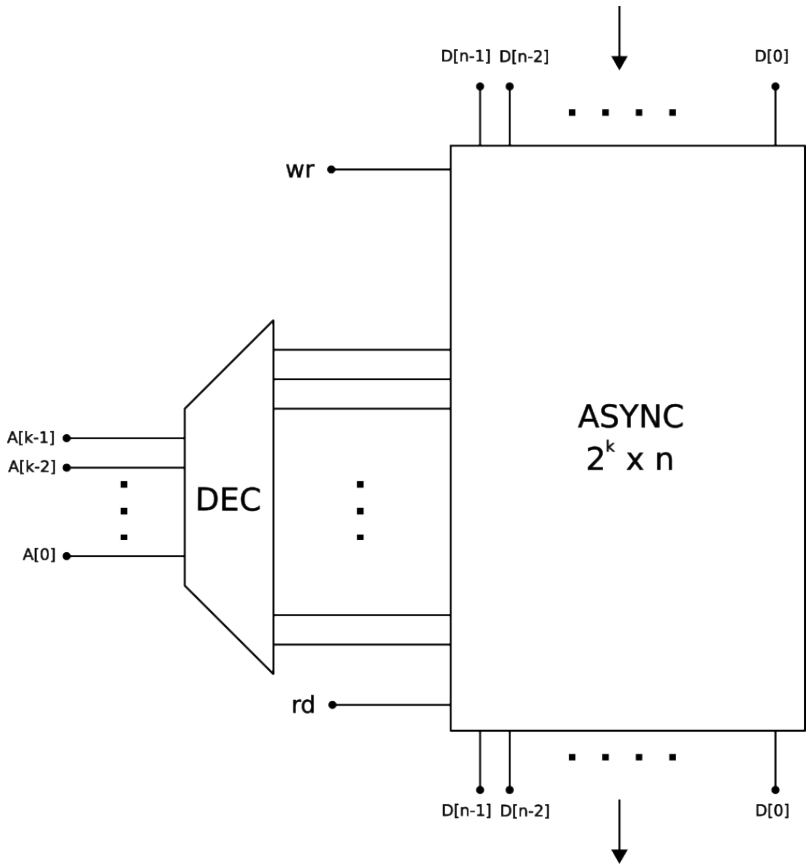
Слика 4.26: Ефикасна имплементација асинхроне меморије

Још ефикаснија реализација асинхроне меморије приказана је на слици 4.26. Меморија приказана на овој слици је величине 4×4. Њена структура је врло слична меморији датој на слици 4.23. Кључна разлика је у начину реализације меморијских ћелија које чувају појединачне битове. Свака врста представља једну меморијску локацију и активира се одговарајућом *линијом речи* (енгл. *word line*) која представља одговарајућу излазну линију меморијског декодера адресе (хоризонталне линије на слици). Са друге стране, свака колона се повезује на једну *линију бита* (енгл. *bit line*) којом се одговарајући бит са улаза доводи до меморијских ћелија (вертикалне линије на слици које иду са леве стране меморијских ћелија). Исте линије битова се користе и за повезивање на излаз. Приметимо да за сваку линију бита постоји и њој *негирана линија бита* (вертикална линија која иде са десне стране меморијских ћелија). Свака меморијска ћелија (реза) се састоји

из два НЕ кола повезаних тако да чине стабилни систем (излаз једног се повезује на улаз другог и обратно тако да један другом одржавају улаз и на тај начин чувају стање). Вредност на излазу доње негације (тј. вредност на левој страни резе) се сматра вредношћу која се чува у рези, док је вредност са десне стране негирана вредност чуване вредности. Када желимо да прочитамо сачувану вредност, декодером активирамо одговарајућу линију речи чиме се отварају пропусни транзистори са леве и десне стране реза у изабраној врсти. Вредности које се чувају у ћелијама те врсте се пропуштају на линије битова, а њихове негиране вредности се пропуштају на негиране линије битова. Како пропусни транзистори NMOS типа добро пропуштају нулу, а слабо пропуштају јединицу, само један од два транзистора у ћелији ће пропустити вредност (онај са чије стране је нула). На пример, ако ћелија чува вредност нула, тада је на левој страни нула, а на десној јединица. Нула ће проћи на битску линију. Обрнуто, ако је вредност коју ћелија чува јединица, тада ће са десне стране бити нула, па ће нула бити пропуштена на негирану битску линију. Негација на дну битске линије (или негиране битске линије) ће пропуштену нулу претворити у јединицу и довести је на контролни (*enable*) улаз одговарајућег бафера са три стања. Ако је на битској линији нула, тада ће на излаз бити пропуштена нула. Са друге стране, ако је на негираној битској линији нула, тада ће на излаз бити пропуштена јединица. Када желимо да извршимо упис, тада се вредности које хоћемо да упишемо доведу на улаз *data_in* и активира се *wr* сигнал чиме се улазне вредности пропуштају на битске линије (а њихове негације на негиране битске линије). Након тога се активира одговарајућа линија речи. Том приликом се отварају пропусни транзистори, па вредности на битским линијама могу доћи у колизију са сачуваним вредностима у ћелијама. Да би све функционисало како треба, транзистори у негацијама у меморијским ћелијама се намерно конструишу тако да буду мањи и слабији од пропусних транзистора. На тај начин, вредности које се преко пропусних транзистора са битских линија пренесу ка резама надјачаће вредности које се тренутно чувају у резама и натераће их да промене своја стања. Након искључивања линије речи, ова новоуспостављена стања ће остати сачувана у резама.

При овој имплементацији за сваки бит потребно је свега 6 транзистора, што је много ефикасније од раније приказаних реализација. Додатно, ова имплементација је веома брза, тј. има веома мало кашњење на нивоу меморијских ћелија (не рачунамо овде кашњење декодера које зависи од величине меморије). Колико је аутору познато, у CMOS технологији не постоји реализација меморијске ћелије засноване на резама која се састоји из мањег броја транзистора.

На слици 4.27 приказана је поједностављена блок шема асинхроне меморије, која сакрива детаље имплементације појединачних меморијских ћелија. Читава матрица меморијских ћелија приказана је апстрактним колом означеним са *ASYNC*. Саме меморијске ћелије могу бити имплементиране било као класичне резе, било као претходно описане оптимизоване резе са 6 транзистора, што је у пракси најчешћи случај. Подаци на битске линије долазе одозго (упис), а излазе на доле (читање). Хоризонталне линије речи се активирају помоћу декодера. Асинхрона матрица прихвата два контролна сигнала, *wr* и *rd* који омогућавају упис и читање путем битских линија, респективно. Приказана блок шема ће бити основа за разматрање сложенијих меморија у наставку.

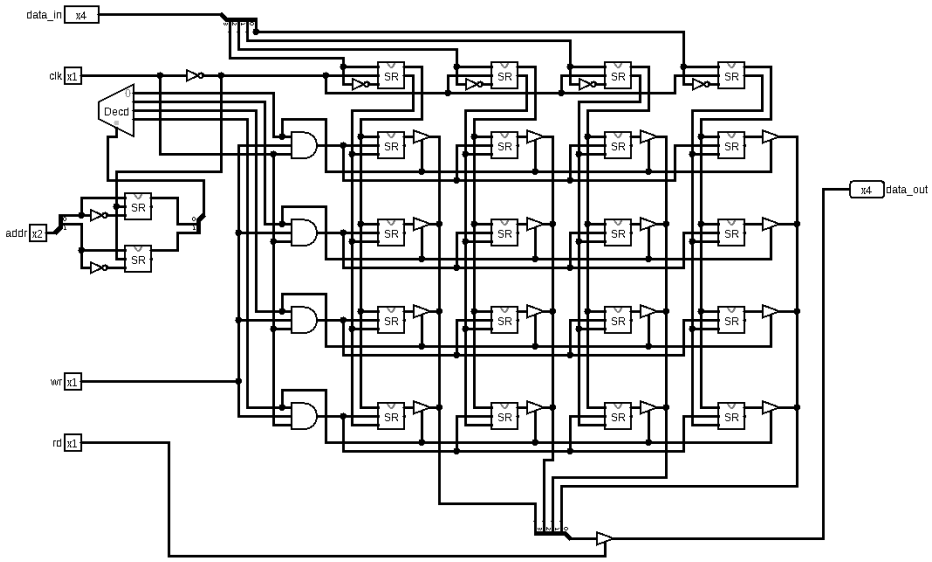


Слика 4.27: Блок шема асинхроне меморије

4.5.4 Оптимизација синхроних меморија

Раније смо приказали на који се начин синхроне меморије могу реализовати коришћењем флип-флопова. Меморија $m \times n$ је у себи садржала mn флип-флопова. Ако претпоставимо да се сваки флип-флоп састоји из две резе у господар-слуга организацији, тада је укупан број реза у меморији $2mn$. Ово можемо оптимизовати тако што, уместо да свака меморијска ћелија има главну (мастер) и подређену резу, за сваку колону имамо једну заједничку мастер резу, док се ћелије састоје само из подређених реза. У негативном делу часовника вредност бита са улаза се памти у мастер рези, док се при наиласку позитивног руба часовника ова вредност пропагира на ону подређену резу која се налази у врсти која је селектована декодером на основу адресе.

Пример овакве реализације је дат на слици 4.28. У питању је меморија величине 4×4. Прва врста на слици представља мастер резе, док остале четири врсте представљају подређене резе (свака врста је једна меморијска локација). Додатне две резе на левој страни слике служе за чување адресе. У негативном делу часовника су мастер резе отворене, јер је њихов e улаз повезан на часовник преко негације. Отуда се вредност са улаза уписује у

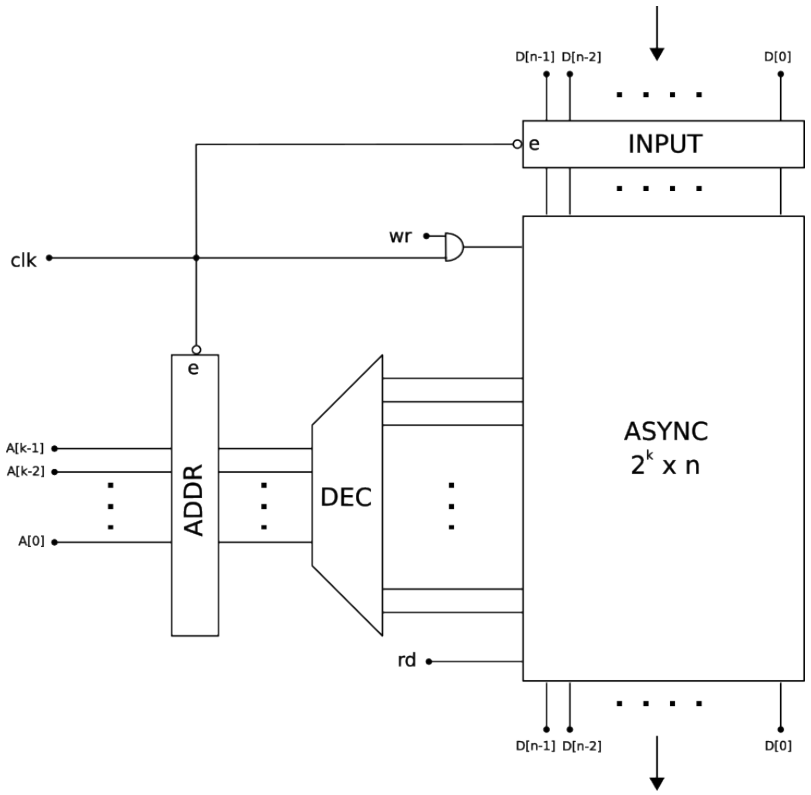


Слика 4.28: Оптимизована синхрона меморија

мастер резе. Такође, у негативном делу часовника се у резе за памћење адресе уписују адресни битови. Разлог за овако нешто је да се не би десило да се касније у позитивном делу часовника променом адресе на улазу вредност запамћена у мастер резама преусмери на неку другу адресу (другим речима, хоћемо да се и адресни битови као и сви остали улазни битови узимају у обзир само у тренутку наиласка узлазног руба, а касније промене не би смеле да утичу на рад кола). На узлазном рубу часовника се затварају мастер резе као и резе за чување адресних битова, а отварају се подређене резе у матрици меморије. Запамћена вредност у мастер резама се прослеђује ка подређеним резама у врсти чија је адреса запамћена у адресним резама. Притом, то се дешава само ако је wr сигнал укључен, с обзиром да се он доводи као додатни улаз у конјункције испред подређених реза.

На овај начин смо постигли да уместо $2mn$ реза имамо $mn + n + 2$ резе. За веће матрице овај однос тежи ка 2, тј. имаћемо приближно два пута мање реза у оптимизованој варијанти. Приметимо да ако ову оптимизовану имплементацију упоредимо са раније датом имплементацијом асинхроне меморије 4×4 (слика 4.23), можемо закључити да смо синхрону меморију добили тако што смо на асинхрону меморијску матрицу додали *синхронизациону логику*, која се у овом случају састоји из додатног реда реза за чување податка при упису и адресних реза. У сложенијим меморијама ова синхронизациона логика може бити нешто сложенија. Међутим, принцип је увек исти – најпре направимо матрицу асинхроне меморије која се састоји из асинхроних меморијских ћелија (типично оптимизованих, као у претходном одељку), а затим је синхронизујемо додавањем синхронизационе логике. Овај принцип је илустрован на слици 4.29 која приказује поједностављену блок шему типичне синхроне меморије.

Њена основа је блок шема асинхроне меморије приказана на слици 4.27 у претходном одељку. Додатно, шема садржи коло означено са *ADDR*, које представља асинхрони регистар који се састоји из реза за памћење адресних битова. Ове резе су отворене у негативном делу часовника, због тога што се на њихов e улаз доводи негирани сигнал часовника. Слично је и са колом означеним са *INPUT* које садржи резе за чување улазних битова података при упису. Коло *ASYNC* представља асинхрону матрицу меморијских ћелија која је имплементирана на начин који је описан у претходном одељку. Да би се извршио упис вредности са улаза у одговарајућу врсту матрице, потребно је активирати wr сигнал, као и одговарајућу линију речи. Линије речи се активирају декодером. Приметимо да је сигнал wr додатно ограничен конјункцијом, тако да „пролази" само у позитивном делу циклуса. На овај начин, обезбеђује се синхрони упис на узлазном рубу часовника, као што је раније објашњено.



Слика 4.29: Блок шема синхроне меморије

4.5.5 О произвољном приступу

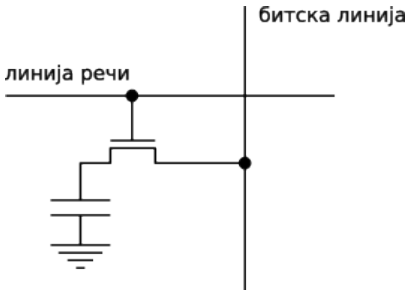
Меморије које смо приказали у претходним одељцима имају својство да се свакој меморијској локацији може приступити у приближно једнаком времену. Под *временом приступа* (енгл. *access time*) подразумевамо време од тренутка када се изда захтев за читање или упис појединачне локације до

тренутка када се та операција заврши. Меморије које имају ово својство се обично називају *меморије са произвољним приступом* (енгл. *random access memory* (RAM)). Насупрот меморијама са произвољним приступом, постоје и меморије код којих приступ појединим локацијама може захтевати више времена од приступа неким другим локацијама. Ово је типично случај код спољних меморија рачунара које садрже механичке делове (хард дискови, магнетне траке, компакт дискови, и тд.).

Доста конфузије у популарној литератури ствара однос између RAM и ROM меморије. Наиме, ROM меморија такође има својство произвољног приступа, па је на неки начин и она RAM меморија. Међутим, термин RAM меморија се обично користи за једну специфичну врсту меморије са произвољним приступом која се користи као главна, тј. *оперативна меморија* рачунара. Иако термини RAM и ROM оригинално нису најсрећније изабрани, они су се временом прилично одомаћили са овим веома специфичним значењем. Дакле, када данас кажемо RAM меморија, обично мислимо на оперативну меморију рачунара, а не на било коју меморију са произвољним приступом (попут ROM меморије).

4.5.6 Динамичке меморије

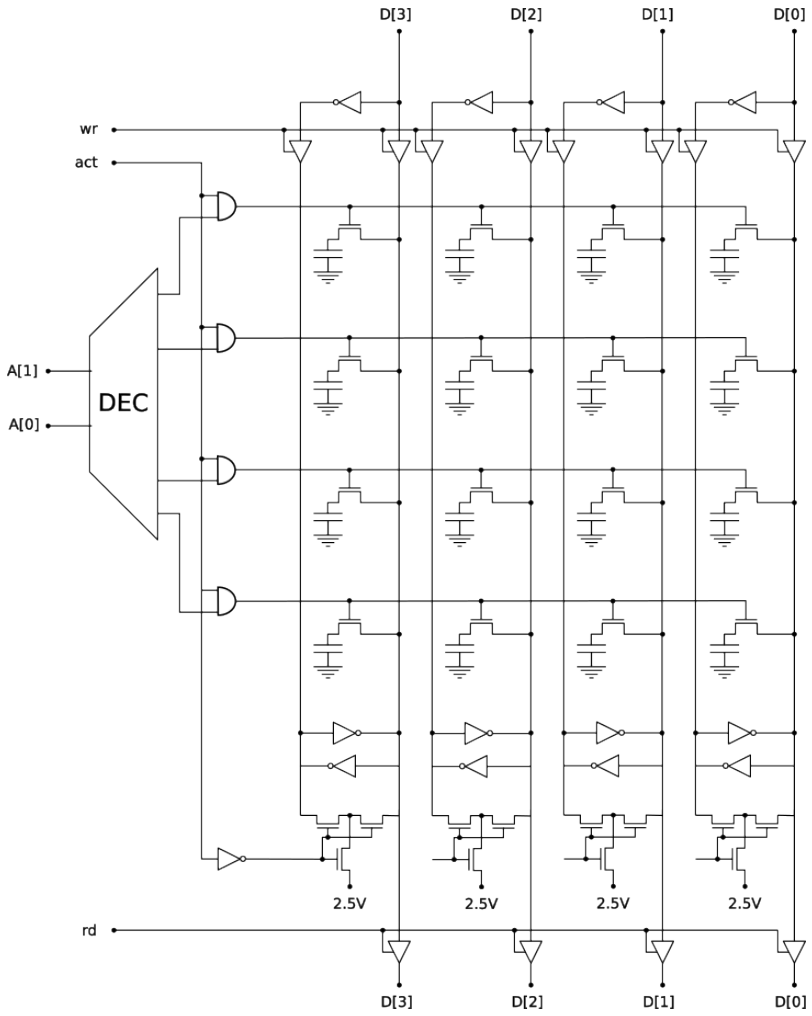
Меморије које смо до сада приказали (и асинхроне и синхроне) се обично називају *статичке меморије са произвољним приступом* (енгл. *static random access memory* (SRAM)). Њихова главна карактеристика је да се за чување појединачних битова користе резе, тј. секвенцијална кола која могу имати два стабилна стања. Насупрот ових меморија, постоје и *динамичке меморије са произвољним приступом* (енгл. *dynamic random access memory* (DRAM)). Код ових меморија, свака меморијска ћелија за чување једног бита се састоји од само једног транзистора и једног кондензатора (слика 4.30).



Слика 4.30: Ћелија динамичке меморије

Вредност бита чува се наелектрисањем кондензатора. Када је кондензатор напуњен, тада је потенцијал његове горње електроде висок, па је сачувана вредност 1. Када је кондензатор испражњен, тада је потенцијал његове горње електроде низак, па је сачувана вредност 0. Транзистор контролише везу кондензатора са спољним светом, тј. његово пуњење и пражњење.

Пример имплементације динамичке меморије величине 4×4 дат је на



Слика 4.31: Асинхрона динамичка меморија 4×4

слици 4.31. Као и код статичких меморија, и овде имамо битске линије (вертикалне линије на слици које стоје са десне стране ћелија), као и негиране битске линије (вертикалне линије које стоје лево од одговарајућих ћелија). На дну сваког пара битских линија налазе се *кола за појачавање опажаја* (енгл. *sense amplifier*). Наиме, како су капацитети кондензатора веома мали, не може се очекивати да простим отварањем транзистора наелектрисање из кондензатора успостави жељени потенцијал на битској линији ($5V$ за логичку јединицу, а $0V$ за логичку нулу). Оно што се може очекивати да наелектрисање из кондензатора мало промени потенцијал битске линије у жељеном смеру. Ову малу промену је потребно опазити и појачати до пуног потенцијала одговарајуће логичке вредности. Кола за појачавање опажаја су у суштини једноставне резе које се састоје из две негације које чине стабилно секвенцијално коло (попут ћелија у оптимизованој имплементацији статичких меморија). Суштина је да

их ми вештачки стављамо у стање метастабилности, тако што се на излазу обе негације успостави потенцијал од $2.5V$ (тј. аритметичка средина потенцијала логичке нуле и логичке јединице). Ово се постиже помоћу три пропусна $NMOS$ транзистора. Ови транзистори се активирају истовремено и обезбеђују кратку везу између леве и десне стране резе (чиме се гарантује исти потенцијал), као и њихову везу са извором напона од $2.5V$. Како су битска линија и негирана битска линија повезане на излазе ове резе, оне ће такође бити наелектрисане на потенцијал од $2.5V$. Овај поступак је познат као *предпуњење* (енгл. *precharge*).

Након завршетка предпуњења искључују се пропусни транзистори у колима за појачавање опажаја, а резе остају у метастабилном стању. Притом, као што знамо, метастабилно стање није природно стање за резу која тежи да буде у стабилном стању. Отуда је довољан мали подстицај ка нижем или вишем потенцијалу да реза „добровољно" оде у неко од два стабилна стања. Овај „подстицај" се добија од наелектрисања кондензатора које се пропушта на битску линију. Када се успостави стабилно стање у рези, одговарајућа вредност биће пропагирана на битску линију, чиме ће се добити ефакат „појачања" вредности очитане из кондензатора.

Операција се даље одвија на следећи начин. Истовремено са искључивањем пропусних транзистора у колима за појачавање опажаја, декодер активира одговарајућу линију речи којом се отварају транзистори у ћелијама у одговарајућој врсти. Овим се наелектрисање кондензатора пропушта на битске линије и нарушава крхки баланс који је успостављен на битској линији. На пример, ако је кондензатор био напуњен (логичка јединица), тада је његова горња електрода на потенцијалу од $5V$. Услед разлике потенцијала, кондензатор се делимично празни, а потенцијали горње електроде и битске линије се изједначавају на неком потенцијалу који је нешто виши од $2.5V$ (на пример, $2.6V$). Овим се реза у појачавачу опажаја из стања метастабилности гура ка стабилном стању у коме је на десној страни резе логичка јединица (тј. потенцијал $5V$). Дакле, реза која природно жели да дође у стабилно стање овим изливом наелектрисања из кондензатора бива „гурнута" ка логичкој јединици, чиме се успоставља логичка јединица на битској линији. Кондензатор, који је био делимично испражњен (и изгубио своју сачувану вредност), сада бива поново напуњен, чиме се поново успоставља логичка јединица у тој ћелији. Аналогно, ако је иницијално кондензатор био празан (логичка нула), тада ће његова горња електрода бити на потенцијалу $0V$. Отварањем транзистора, услед разлике потенцијала, битска линија ће почети да пуни кондензатор и њихови потенцијали ће се изједначити на некој вредности која је нешто нижа од $2.5V$ (на пример, $2.4V$). Овим ће реза из метастабилности бити гурнута у стабилно стање логичке нуле, а потенцијал $0V$ ће бити успостављен на битској линији. Такође, овим ће се делимично напуњен кондензатор поново испразнити, па ће и у њему бити поново успостављена логичка нула која је претходно била нарушена.

Цео овај поступак познат је под називом *отварање врсте* (енгл. *row opening*). Овим поступком се постиже да резе у појачивачима опажаја успоставе вредности које су очитане из ћелија изабране врсте. Приметимо да се током овог процеса вредности које су биле чуване у кондензатору најпре униште (делимичним пражњењем, односно, пуњењем кондезатора), а затим поново успоставе након појачавања сигнала. Овај феномен познат је и под

називом *деструктивно читање*. Како пуњење и пражњење кондензатора захтева извесно време, процес отварања врсте је по правилу веома спор.

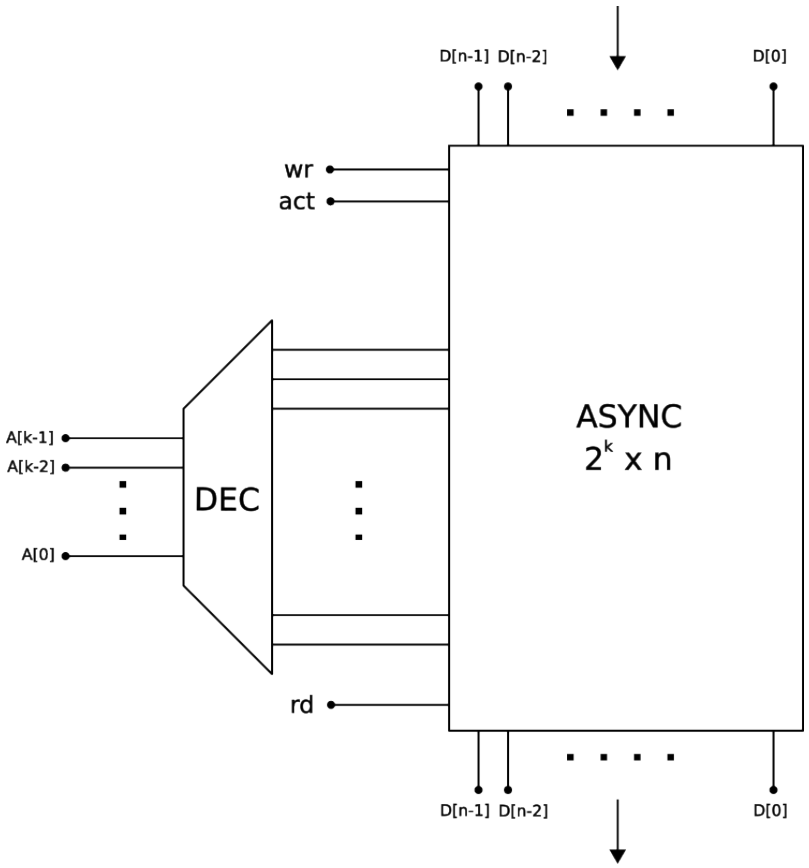
Активација процеса отварања врсте обавља се помоћу сигнала *act*. Овај сигнал са једне стране омогућава пролаз сигнала из декодера ка одговарајућој линији речи (јер контролише конјункције), а са друге стране постављањем *act* на вредност логичке јединице искључују се пропусни транзистори у колима за појачање опажаја, чиме се завршава поступак предпуњења и омогућава очитавање вредности из ћелија изабране врсте. Након што протекне довољно времена да се овај процес заврши, могуће је активирати сигнал *rd* или *wr* (уз задржавање *act* сигнала на вредности 1), у зависности од жељене операције. Активацијом сигнала *rd* се просто очитане вредности из реза у појачавачима опажаја пропуштају на излаз. Активацијом сигнала *wr* се, слично као и код статичких меморија, омогућава пропуштање вредности са улаза на битске линије, чиме се форсира промена стања реза у појачавачима опажаја. Наиме, слично као и код статичких меморија, транзистори у овим резама се праве да буду мале снаге тако да их је лако надјачати спољним сигналом и натерати их да промене своје стање. Када се у резама (и на битским линијама) успоставе нове вредности, кондензатори у отвореној врсти ће бити напуњени, односно испражњени, у складу са успостављеним вредностима. Када се операција заврши, сигнал *act* се искључује, чиме се транзистори у врсти затварају и кондензатори бивају одсечени од спољног света, чувајући запамћене вредности. Истовремено, поново се активира процес предпуњења, тј. битске линије се доводе на потенцијал $2.5V$, чиме је меморија спремна за следећу операцију.

Напоменимо да се кондензатори временом празне и сами од себе (с обзиром да је немогуће идеално их изоловати), те је потребно периодично вршити освежавање комплетне меморије (нпр. на сваких $50ms$), како се садржај не би изгубио. Овај процес додатно успорава рад целе меморије, јер је током процеса освежавања немогуће обављати читање и писање.

Поједностављена блок шема динамичке меморије дата је на слици 4.32. Може се приметити да је дијаграм готово идентичан оном на слици 4.27, уз додатни *act* сигнал који се испоручије матрици динамичких ћелија, као што је претходно описано.

Описани дизајн динамичких меморија је у основи асинхрон. Додавањем синхронизационе логике као што је раније описано добијају се *синхроне динамичке меморије са произвољним приступом* (енгл. *synchronous dynamic random access memory* (SDRAM)). Поједностављена блок шема синхроне динамичке меморије дата је на слици 4.33. Кључна разлика у односу на статичке синхроне меморије (слика 4.29) је у томе што овде постоји додатни *act* сигнал.

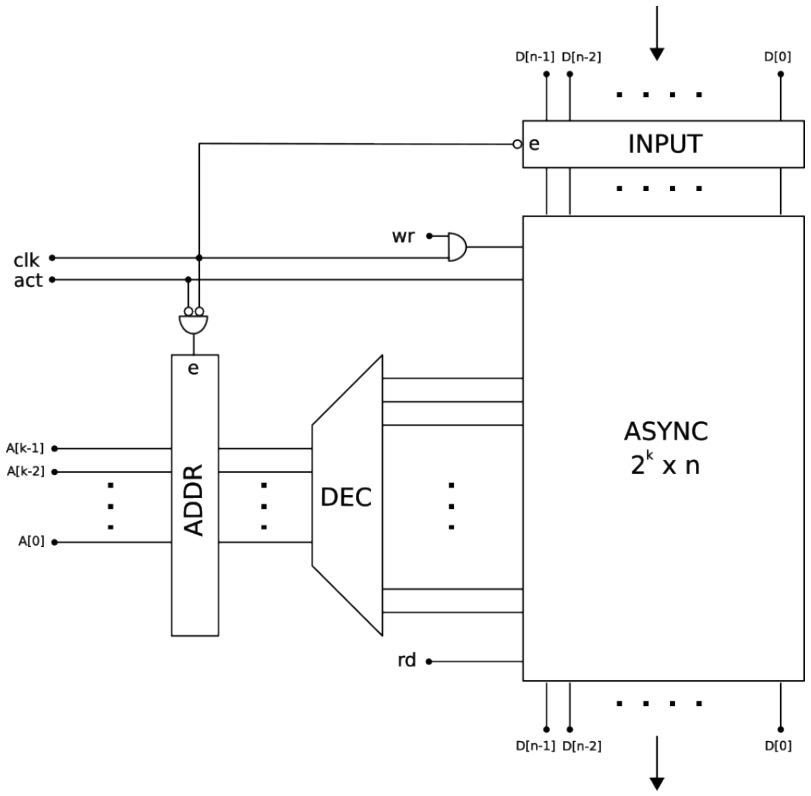
Операције са синхроном динамичком меморијом се обављају на сличан начин као и са асинхроном, с том разликом што се сада промене вредности сигнала синхронизују са часовником. Најпре се на адресни улаз доведе адреса локације која се памти у адресним резама. На узлазном рубу се активира *act* сигнал, а адресне резе се затварају. Приметимо да се негирана вредност *act* сигнала у конјункцији са негираном вредношћу часовника доводи на *e* улаз адресног регистра. Овим се спречава да се адреса мења док је укључен *act* сигнал, чак и у негативном делу часовника. Ово је од значаја зато што процес отварања врсте, у зависности од брзине



Слика 4.32: Блок шема асинхроне динамичке меморије

меморије и фреквенције часовника, може трајати више циклуса. Тиме се обезбеђује стабилна адреса током целог трајања овог процеса. Након тога се, у зависности од жељене операције, на неком од наредних узлазних рубова активира *rd* или *wr* сигнал. У првом случају се очитане вредности пропуштају на излаз. У другом случају је важно претходно на улазе за податке довести податак који желимо да упишемо. На узлазном рубу ће улазне резе за податке бити закључане, а *wr* сигнал ће бити пропуштен ка меморијској матрици, чиме ће се извршити упис улазних вредности у отворену врсту. Након тога можемо искључити *wr* и *act* сигнале.

Израда динамичких меморија је много јефтинија, због знатно мањег броја компоненти (уместо 6 транзистора имамо један транзистор и један кондензатор). Ово омогућава израду меморија веома великог капацитета по релативно ниској цени (савремене динамичке RAM меморије се мере у гигабајтима), па се динамичке меморије (у својој синхроној варијанти) данас по правилу користе као оперативне меморије. Са друге стране, статичке меморије су много брже, али су и скупље, па се обично користе за израду малих, али веома брзих меморија које су близу процесора (скуп регистара процесора и кеш меморија).

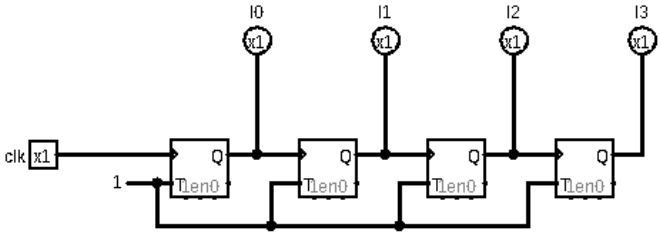


Слика 4.33: Блок шема синхроне динамичке меморије

4.6 Бројачи

Бројачи су посебна врста регистара који имају могућност да у сваком циклусу часовника своју вредност увећају (или умање) за један. На овај начин, ово коло може бројати циклусе часовника. Најједноставнија примена оваквог кола је за мерење протеклог времена у рачунару. Наравно, бројачи не морају увећавати своју вредност у сваком циклусу часовника, већ, на пример, само у оним циклусима у којима је укључен неки додатни контролни улаз (нпр. *inc* улаз). Такви бројачи се могу користити као бројачи инструкција у програму (нпр. *програмски бројач* у процесору), где се увећавање врши када прелазимо на следећу инструкцију. На сличан начин, бројачи се могу користити за бројање неких специфичних догађаја у рачунару (нпр. колико пута је притиснут неки тастер на тастатури).

Пример имплементације бројача дат је на слици 4.34. Имплементација се састоји из T флип-флопова који су конструисани тако да реагују на силазну ивицу часовника, а чији су сви T улази повезани на константну јединицу. Ово значи да ће при свакој транзицији са 1 на 0 на свом *clk* улазу флип-флопови мењати своје стање. Међутим, само флип флоп на позицији најмање тежине (крајњи десни на слици) је директно повезан на сигнал часовника. Сваком следећем флип-флопу се на улаз за часовник доводи вредност са излаза претходног флип-флопа. Ово значи да ће само најнижи



Слика 4.34: Асинхрони бројач

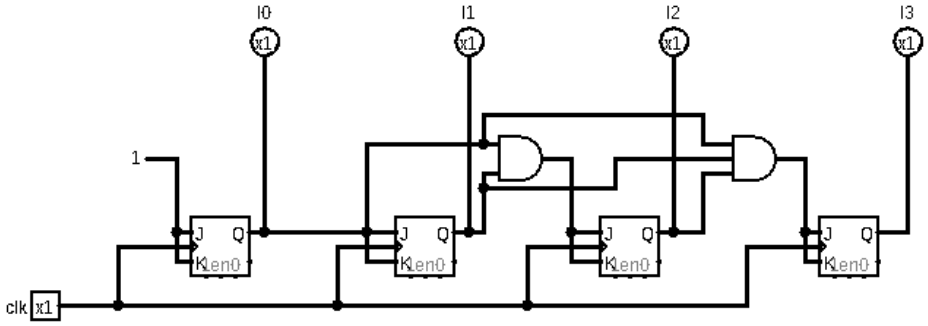
флип-флоп своје стање мењати на свакој транзицији са 1 на 0 сигнала часовника (најнижи бит бројача), док ће сваки следећи флип-флоп мењати своје стање при транзицији вредности претходног флип-флопа (претходног бита бројача) са 1 на 0. Ово одговара бинарном бројању: нпр. тробитно бројање представља секвенцу вредности $000 \to 001 \to 010 \to 011 \to 100 \to 101 \to 110 \to 111 \to 000$. Као што се може приметити, свака битска позиција мења своју вредност ако и само ако се претходна битска позиција (тј. позиција десно од ње у запису бинарног броја) мења са јединице на нулу у том кораку.

Приказана имплементација позната је и као *асинхрони бројач*. Назив потиче отуда што су T флип флопови из којих се регистар састоји повезани на различите синхронизационе сигнале. Иако је оваква имплементација веома једноставна, она има један важан недостатак који се огледа у акумулацији кашњења. Наиме, уколико је време пропагације појединачног флип-флопа једнако Δ, тада ће промена на улазу за часовник сваког следећег флип-флопа каснити за претходним управо за Δ, па ће за толико каснити и промена вредности тог флип-флопа. На пример, приликом транзиције $111 \to 000$ ће се најпре у тренутку t_0 променити вредност *clk* сигнала са 1 на 0. То ће изазвати промену на најнижем биту која ће се на излазу најнижег флип-флопа појавити у тренутку $t_0 + \Delta$. Ова промена ће бити са 1 на 0 што ће изазвати промену на другом флип-флопу која ће се на излазу овог флип-флопа појавити у тренутку $t_0 + 2\Delta$. Како је и ова промена са 1 на 0, то ће изазвати промену на трећем флип-флопу, а она ће се на излазу појавити у тренутку $t_0 + 3\Delta$. Дакле, промене битова ће се манифестовати у форми таласа, при чему свака за претходном касни за Δ. У случају n битова, цео регистар ће стабилизовати своју нову вредност тек након $n \cdot \Delta$. У случају великог n ово време може бити дуже од циклуса часовника, па бројач неће функционисати на очекивани начин.

Из наведених разлога се у пракси чешће користе *синхрони бројачи*. Пример имплементације синхроног бројача дат је на слици 4.35.

Имплементација се састоји из JK флип-флопова који су овог пута конструисани тако да своје стање мењају на узлазном рубу часовника.[4] Сада су улази за часовник свих флип-флопова повезани на стварни сигнал часовника, па се тиме гарантује да ће заиста сви променити своју вредност у

[4] у овој имплементацији користимо JK флип-флопове, мада смо могли да користимо и T флип-флопове, с обзиром да су на свим флип-флоповима J и K улази повезани на исти сигнал. Разлог за коришћење JK флип-флопова је лакше уопштавање на сложеније бројаче, што ћемо видети у следећем одељку.



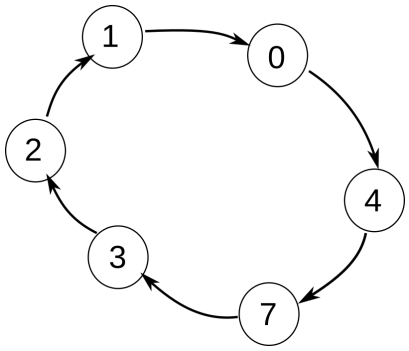
Слика 4.35: Синхрони бројач

истом тренутку. Са друге стране, у којим прелазима ће се мењати вредности којих флип-флопова зависи од тога шта доводимо на одговарајуће J и K улазе. Најнижи (крајњи десни) флип-флоп има оба улаза повезана на јединицу, па ће своје стање мењати на сваком узлазном рубу часовника. Следећи флип-флоп има J и K улазе повезане на излаз претходног флип-флопа, па ће своје стање мењати само ако је тренутно стање претходног флип-флопа јединица, што одговара ситуацији код бинарног бројања (имамо $00 \to 01$, $01 \to 10$, $10 \to 11$, $11 \to 00$, дакле, када је нижи бит у текућем стању јединица, у следећем стању се виши бит мења). Слично, J и K улази трећег флип-флопа се повезују на конјункцију излаза претходна два флип-флопа, па ће трећи флип-флоп мењати своје стање само када су оба претходна флип-флопа у текућем стању јединице (што опет одговара бинарном бројању, јер имамо, нпр. $011 \to 100$ и $111 \to 000$, дакле, трећи бит са десна се мења само када су претходна два бита десно од њега у текућем стању јединице). Уопште, J и K улази k-тог флип-флопа се повезују на излаз И кола које рачуна конјункцију излаза свих претходних $k-1$ флип-флопова. Отуда ће овај флип-флоп мењати своју вредност само ако су у текућем стању сви нижи битови јединице.

Сви флип-флопови који треба да промене своје стање у неком циклусу мењају стање у истом тренутку, тј. у тренутку узлазног руба часовника. Кашњење бројача сада зависи само од кашњења конјункција које рачунају вредности на J и K улазима, као и од кашњења самих флип-флопова, при чему нема акумулације кашњења. Зато су овакви бројачи по правилу бржи и могу радити на вишим фреквенцијама часовника. Са друге стране, њихова имплементација је нешто комплекснија. На свакој битској позицији поред флип-флопа имамо и једно И коло чији је број улаза једнак броју претходних битских позиција. У случају већег броја битова, конјункције на вишим позицијама због превеликог броја улаза неће бити могуће реализовати помоћу једног гејта, већ ће бити неопходно каскадно повезивање више И кола. Ово додатно повећава комплексност, али и кашњење (додуше, кашњење се увећава логаритамском брзином, што је и даље значајно боље него код асинхроних бројача).

4.7 Бројачи са произвољним редоследом стања

Бројачи се могу дизајнирати и тако да броје у неком произвољно задатом редоследу. На пример, можемо имати тробитни бројач који уместо да броји у редоследу $0 \to 1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 0$, он броји у редоследу $0 \to 4 \to 7 \to 3 \to 2 \to 1 \to 0$. Иако овако нешто на први поглед нема смисла (јер зашто би неко бројао на овакав начин), овакво коло можемо разумети и као коло које у одређеном редоследу пролази кроз неки низ стања, при чему ми можемо тај редослед одредити потпуно произвољно. Притом, стања за нас могу имати неку посебну семантику, нпр. могу представљати статус извршавања неке операције у рачунару. Редослед проласка кроз стања можемо приказати графом, као на слици 4.36.



Слика 4.36: Пример дијаграма преласка бројача

Да бисмо дизајнирали бројач који броји у неком произвољном унапред задатом редоследу, можемо приступити на исти начин као код дизајна синхроних бројача који броје у нормалном поретку. У сваком стању потребно је вредности J и K улаза свих флип-флопова поставити тако да се на наредном узлазном рубу часовника стање бројача промени на одговарајући начин, тј. да се пређе у наредно стање дато у спецификацији бројача. Код уобичајених бројача, да би се ово постигло било је довољно да на улазе J и K сваког флип-флопа доводемо конјункцију вредности свих претходних флип-флопова, јер је код нормалног бројања управо то био услов за промену вредности бита који се чува у том флип-флопу. Код произвољних бројача, J и K улази флип-флопова могу бити произвољне функције од текућег стања. Поставља се питање, како их одредити? Један начин је да се за сваки конкретан бројач посматрају законитости у променама вредности битова стања и да се на тај начин некако одреди функција за сваки од J и K улаза. Овакав *ad-hoc* приступ пролази у једноставнијим случајевима, али је у општем случају ипак потребно пронаћи систематски приступ. У наставку овог текста, изложићемо један такав систематски приступ.

Под *таблицом ексцитације* (енгл. *excitation table*) неког секвенцијалног кола подразумевамо инверз његове таблице преласка. Другим речима, ова таблица нам говори које вредности би требало да буду на улазу

како би се стање кола променило на одговарајући начин. На пример, инверзијом таблице преласка JK флип-флопа (табела 4.6), добијамо таблицу ексцитације приказану у табели 4.8.

Q	Q^{sled}	J	K
0	0	0	-
0	1	1	-
1	0	-	1
1	1	-	0

Табела 4.8: Таблица ексцитације JK флип-флопа

Ова таблица нам говори шта треба довести на улазе флип-флопа да би се стање променило на жељени начин. На пример, ако је тренутно стање 0 и желимо да остане 0, довољно је да је $J = 0$, док улаз K може бити било шта (ако је 0, имаћемо комбинацију $(0, 0)$ која одржава постојеће стање, док ћемо за $K = 1$ имати ресетовање, што нема ефекта, јер је стање већ 0). Због тога је вредност улаза K у таблици ексцитације небитна. Са друге стране, ако је тренутно стање 0, а желимо да у следећем циклусу буде 1, тада је довољно да се на J улаз доведе јединица, док је опет небитно шта ће бити на улазу K (ако је $K = 0$, тада ћемо имати сетовање, као што и желимо, док ћемо за $K = 1$ имати инвертовање, које ће имати исти ефекат промене вредности са 0 на 1). Слично се анализирају и остали случајеви.

Претпоставимо сада да желимо да направимо бројач који броји у редоследу: $0 \to 4 \to 7 \to 3 \to 2 \to 1 \to 0$, као на претходној слици. Како је највећи број који се појављује међу стањима 7, следи да је за кодирање стања довољно 3 бита, па ћемо имати тробитни бројач. Кодирајмо најпре стања бројача бинарно. Сада имамо низ прелазака $000 \to 100 \to 111 \to 011 \to 010 \to 001 \to 000$. Означимо битове бројача редом са A, B и C и запишимо ове прелазе у облику таблице (табела 4.9). Ову таблицу називаћемо *таблицом ексцитације бројача*. Она нам говори које вредности морамо довести на J и K улазе сваког од флип-флопова у сваком од стања, како би се стање променило на одговарајући начин на следећем узлазном рубу часовника. Ова таблица формирана је на основу таблице ексцитације JK флип-флопа: за сваки од битова стања A, B и C посматрамо у које је вредности A^{sled}, B^{sled} и C^{sled} респективно потребно да ови битови пређу у следећем кораку, на основу чега, према таблици ексцитације за JK флип-флоп, одређујемо на које вредности је потребно поставити J и K улазе одговарајућег флип-флопа, да би се такав прелаз десио.

Приметимо да бројач не мора пролазити кроз сва могућа тробитна стања. На пример, наш бројач не пролази кроз стања 5 и 6. У таблици се, зато, за ова стања за све J и K улазе користе небитне вредности, јер у тим стањима никада нећемо ни бити, па нам је свеједно шта би се у тим случајевима десило са улазима флип-флопова.

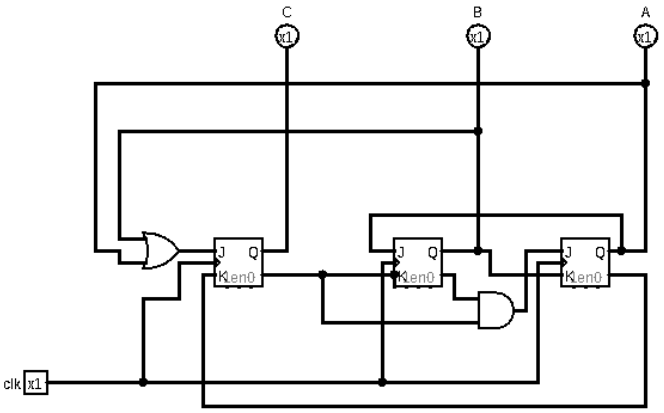
Након што смо формирали таблицу ексцитације бројача, сваку од колона J_A, K_A, J_B, K_B, J_C, K_C посматрамо као функције од тренутног стања, тј. од колона A, B и C. За ове функције одређујемо минимални ДНФ израз

A	B	C	A^{sled}	B^{sled}	C^{sled}	J_A	K_A	J_B	K_B	J_C	K_C
0	0	0	1	0	0	1	–	0	–	0	–
0	0	1	0	0	0	0	–	0	–	–	1
0	1	0	0	0	1	0	–	–	1	1	–
0	1	1	0	1	0	0	–	–	0	–	1
1	0	0	1	1	1	–	0	1	–	1	–
1	0	1	–	–	–	–	–	–	–	–	–
1	1	0	–	–	–	–	–	–	–	–	–
1	1	1	0	1	1	–	1	–	0	–	0

Табела 4.9: Таблица ексцитације бројача са слике 4.36

(нпр. помоћу Карноових мапа, што остављамо читаоцу за вежбу):

$$J_A = \overline{B}\,\overline{C} \quad J_B = A \quad J_C = B + A$$
$$K_A = B \quad K_B = \overline{C} \quad K_C = \overline{A}$$



Слика 4.37: Имплементација бројача са слике 4.36

Имплементација бројача дата је на слици 4.37. Бројач се састоји из три JK флип-флопа, као и из додатних гејтова којима се реализују логичке функције које израчунавају вредности J и K улаза. У нашем случају, имамо два додатна гејта, једну конјункцију и једну дисјункцију. У општем случају, број додатних гејтова може бити и већи, али најчешће добијена кола нису превише компликована. За то постоје два разлога. Први је у томе што су нам негације битова A, B и C већ доступне (јер сваки флип-флоп на излазу има и своје стање и његову негацију), па не морамо да уводимо додатна NE кола. Други разлог је то што су JK флип-флопови веома флексибилни – за сваки прелаз имамо по две могуће комбинације J и K улаза које реализују тај прелаз. На пример, да бисмо прешли из стања 0 у стање 1, на улазе треба довести комбинацију $(J, K) = (1, 0)$ или $(J, K) = (1, 1)$. Ово нам даје велики број небитних вредности у Карноовим мапама, што најчешће даје прилично једноставне ДНФ изразе.

4.8 Коначни аутомати

Основни недостатак бројача са произвољним редоследом стања је то што се на поредак промене стања ни на који начин не може утицати од споља. Другим речима, бројач увек броји у истом редоследу. Уколико би бројач имао улаз, тада би помоћу тог улаза могли да утичемо на то у које стање ће се прећи у следећем кораку. Овакво секвенцијално коло називамо *коначни аутомат*. Притом, коло може имати и излаз чија се вредност генерише приликом сваког преласка из стања у стање, а која, као и следеће стање, зависи од претходног стања и вредности улаза у тренутку преласка у ново стање.[5] Коначни аутомат представља најопштији модел синхроног секвенцијалног кола. Као што су флип-флопови, као најједноставнија синхрона секвенцијална кола имали своје улазе, своје стање и излазе, као и своју таблицу преласка, тако ће и произвољни аутомат имати своје улазе, своје стање и своје излазе, као и таблицу преласка. Притом, таблицу преласка ћемо моћи да дефинишемо потпуно произвољно, у складу са нашим потребама.

Посматрајмо, на пример, аутомат који има четири стања $(0, 1, 2, 3)$, као и један једнобитни улаз X и један једнобитни излаз Y. Нека је аутомат дефинисан таблицом преласка датом у табели 4.10.

Q	X	Q^{sled}	Y
0	0	1	0
0	1	2	0
1	0	2	1
1	1	3	0
2	0	0	1
2	1	3	1
3	0	1	1
3	1	1	0

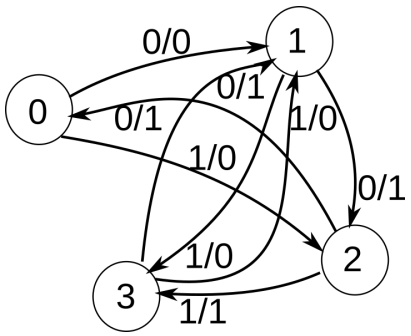
Табела 4.10: Пример таблице преласка аутомата

Овај аутомат можемо представити и графом, као на слици 4.38. У овом графу, чворови представљају стања, а гране представљају прелазе између стања. Свака грана је означена паром X/Y, где X означава за који улаз се тај прелаз врши, а Y представља вредност излаза која се генерише при том прелазу. На пример, када је аутомат у стању 0, а на улазу је вредност 1, тада аутомат на следећем узлазном рубу часовника прелази у стање 2, а на излазу се добија вредност 0.

Дизајн коначних аутомата се обавља на врло сличан начин као код дизајна бројача са произвољним редоследом стања. На основу таблице преласка генеришемо таблицу ексцитације (табела 4.11).

Таблица ексцитације изгледа сасвим слично као и раније, с тим што сада имамо две додатне колоне: колону X за улаз и колону Y за излаз. У случају вишебитних улаза и излаза, за сваки бит улаза и излаза бисмо имали по једну колону. Притом, упарујемо сва могућа стања са свим могућим вредностима на улазу, те ће број врста матрице бити једнак 2^{k+l}, где је

[5] Такав аутомат се у теорији често назива и *коначни трансдуктор*.



Слика 4.38: Дијаграм преласка коначног аутомата из табеле 4.10. Ознака X/Y на грани значи да се тај прелаз врши за улаз X, при чему се на излазу генерише вредност Y

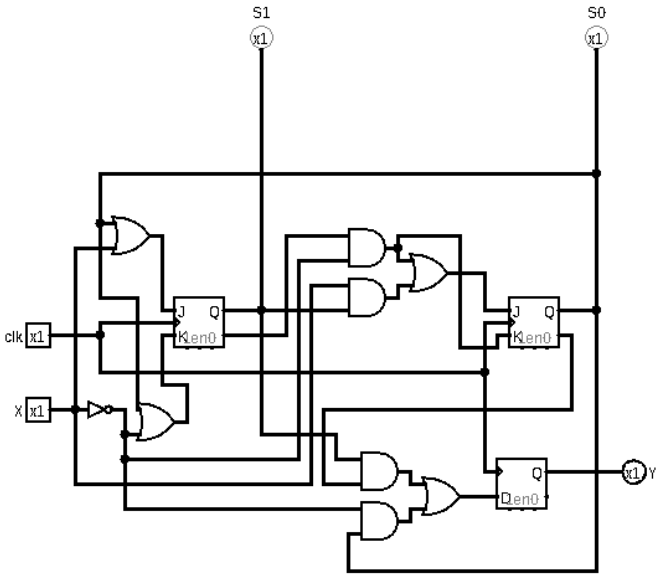
S_1	S_0	X	S_1^{sled}	S_0^{sled}	Y	J_1	K_1	J_0	K_0
0	0	0	0	1	0	0	–	1	–
0	0	1	1	0	0	1	–	0	–
0	1	0	1	0	1	1	–	–	1
0	1	1	1	1	0	1	–	–	0
1	0	0	0	0	1	–	1	0	–
1	0	1	1	1	1	–	0	1	–
1	1	0	0	1	1	–	1	–	0
1	1	1	0	1	0	–	1	–	0

Табела 4.11: Таблица експцитације аутомата из табеле 4.10

k број битова стања, а l број битова на улазу. Колоне J_1, K_1, J_0 и K_0 се попуњавају на основу таблица експцитације JK флип-флопа, а у зависности од тога који прелаз одговарајућег бита стања желимо да остваримо (тј. на основу односа старог стања S_1, S_0 и новог стања S_1^{sled} и S_0^{sled}). Након што се таблица попуни, колоне J_1, K_1, J_0, K_0 и Y се посматрају као функције од S_1, S_0 и X. Минимизацијом добијамо следеће изразе за ове функције:

$$\begin{array}{ll} J_1 = S_0 + X & K_1 = S_0 + \overline{X} \\ J_0 = \overline{S_1}\,\overline{X} + S_1 X & K_0 = \overline{S_1}\,\overline{X} \\ Y = S_1\overline{S_0} + S_0\overline{X} & \end{array}$$

Имплементација аутомата дата је на слици 4.39. Аутомат се састоји из два JK флип-флопа који чувају битове стања, као и одговарајуће комбинаторне логике која на основу горе наведених израза израчунава функције за J и K улазе флип-флопова. Вредности које ове функције израчунавају сада зависе не само од тренутног стања, већ и од тренутног улаза X. Такође, у доњем делу слике имамо коло које израчунава вредност излаза Y. Притом, приметимо да је вредност излаза Y која се израчунава на основу тренутног стања и тренутног улаза према горе наведеном изразу заправо вредност излаза које би требало да се појави на излазу у следећем циклусу (приликом преласка у следеће стање). Због тога се вредност



Слика 4.39: Имплементација аутомата из табеле 4.10

функције излаза не шаље директно на излазни прикључак кола, већ се доводи на улаз једног D флип-флопа у коме ће бити запамћена на следећем узлазном рубу. Након што флип-флоп запамти ту вредност (приликом преласка у следеће стање), она ће се појавити на излазу и биће важећа током читавог циклуса, до следећег преласка.

Имајући ово у виду, можемо сматрати да се стање аутомата заправо састоји из две компоненте: *унутрашњег стања* аутомата S, као и *стања излаза* Y, и обе ове компоненте се чувају у одговарајућим флип-флоповима. У том смислу, можемо рећи да је излаз Y заправо функција од тренутног стања – у питању је функција пројекције $(S, Y) \mapsto Y$. Уопште, аутомати код којих је тренутни излаз функција од тренутног стања[6] називају се *аутомати Мурового типа* (енгл. *Moore machine*). Насупрот њих, постоје *аутомати Милијевог типа* (енгл. *Mealy machine*), код којих тренутни излаз зависи од *тренутног стања* и *тренутног улаза*. На пример, када бисмо у горњем аутомату уклонили D флип-флоп који чува вредност излаза, добили бисмо Милијев аутомат. Код таквог аутомата, свака промена на улазу се одмах манифестује на излазу, па у том смислу овакво коло није у потпуности синхроно (једино се стање мења на синхрони начин, али не и излаз). Милијеви аутомати имају тенденцију да имају мање битова за чување стања (јер не чувају излаз, тј. излаз није део стања), па су зато једноставнији. Такође, вредност на излазу се пропагира брже, јер не чека сигнал часовника. Са друге стране, одсуство синронизације промена на излазу може понашање аутомата учинити мање предвидивим.

[6] Ово није у контрадикцији са претходно изнетом чињеницом да је излаз функција од *претходног стања* и вредности на улазу *у тренутку преласка* – јер тренутно стање је такође функција од претходног стања и улаза у тренутку преласка, па је, у посредном смислу, то случај и са тренутним излазом.

Ово се нарочито може манифестовати у ситуацијама у којима имамо циклично повезана секвенцијална кола – тада може доћи до нежељених и непредвидивих повратних спрега, јер се улази одмах пропагирају на излазе и циклично се крећу кроз систем без контроле часовника. Ми ћемо у наставку овог текста увек подразумевати да радимо са аутоматима Муровог типа.

Глава 5

Принцип рада рачунара

Сваки рачунарски програм се састоји из низа наредби. У свакој наредби се на основу тренутних вредности неких променљивих израчунава нека нова вредност која се затим памти у некој променљивој, како би се могла искористити у наредним наредбама. Самим тим, извршавање сваког програма се састоји из низа *израчунавања* и *памћења*. Израчунавање се у рачунару врши помоћу комбинаторних кола, док се памћење врши помоћу секвенцијалних кола.

Додатно, потребно је контролисати ток самог програма, тј. редослед извршавања појединих корака, као и изворište и одредиште свих података у току израчунавања. Другим речима, у сваком кораку извршавања програма потребно је одредити:

- у којим регистрима/меморијским локацијама се налазе вредности променљивих које се користе у том израчунавању
- коју операцију је потребно извршити, тј. кроз која комбинаторна кола је потребно „пропустити" те вредности
- у које регистре/меморијске локације је потребно уписати резултате израчунавања
- који је следећи корак програма који треба извршити

Притом, ово последње – одређивање следећег корака програма, може зависити од резултата израчунавања у текућем кораку. Ово је неопходно да би програм могао садржати гранање и петље. Отуда је поред чувања вредности променљивих које програм користи неопходно чувати и информацију о текућем кораку програма, као и о статусу претходно извршене операције, како би се могао одредити следећи корак програма.

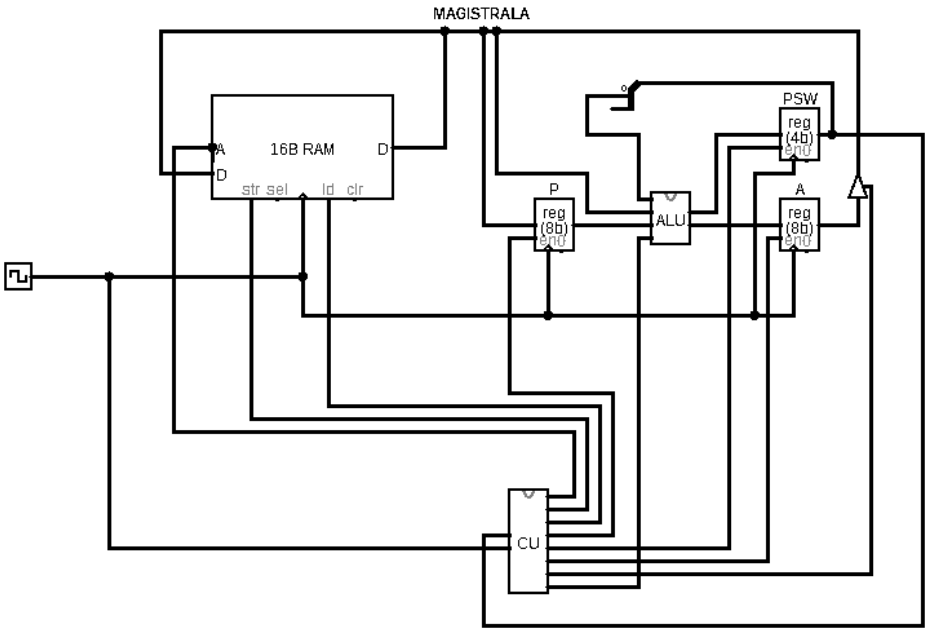
Јединица рачунара која је одговорна за управљање током програма назива се *контролна јединица*. У својој основној варијанти, контролна јединица представља коначни аутомат чије стање означава корак програма који се тренутно извршава. На улаз контролне јединице се доводе информације које утичу на ток програма (попут статуса претходно извршене операције), док се на излазу налазе контролни сигнали који управљају током података у том кораку (тј. одређују изворište, одредиште и операцију која ће бити примењена над подацима).

Имајући све ово у виду, делује да је знање које смо стекли у претходним главама овог текста довољно да конструишемо рачунар који може да извршава неки програм. Управо тиме се бавимо у наставку ове главе.

5.1 Рачунари са фиксираним програмом

Пођимо од најједноставније варијанте у којој рачунар извршава само један фиксирани, унапред задати програм. Овакви рачунари се називају *рачунари са фиксираним програмом* (енгл. *fixed program computer*). Тај фиксирани програм је одређен таблицом преласка коначног аутомата у контролној јединици рачунара и једини начин да се он промени је да се контролна јединица дизајнира из почетка, тј. да се логичка кола која израчунавају следеће стање коначног аутомата другачије повежу. Из данашњег угла, овакав принцип конструкције рачунара је веома нефлексибилан, јер се програмирање заснива на реконфигурацији хардвера. Ипак, први електронски рачунари (попут чувеног ENIAC-а) су функционисали управо на овакав начин.

У наставку овог поглавља описаћемо један једноставан рачунар са фиксираним програмом. Његова шема приказана је на слици 5.1, а његове појединачне компоненте описујемо у наставку.



Слика 5.1: Пример рачунара са фиксираним програмом

Меморија. Меморија рачунара се састоји из 16 8-битних (тј. једнобајтних) меморијских локација – регистара, које ћемо означавати са $R_0, R_1, \ldots, R_{15}$. У питању је синхрона меморија код које се упис врши

на узлазном рубу часовника, а чији интерфејс одговара опису из одељка 4.5.1. Улаз за податке (означен са D на левој страни), као и излаз за податке (означен са D на десној страни) су повезани на *магистралу* која представља осмобитну линију преко које се преносе подаци. Када је контролни сигнал *ld* укључен, тада се вредност са адресе дате на адресном улазу A пушта на магистралу, а у супротном је вредност на излазу D једнака вредности високе импедансе (тј. улаз *ld* одговара улазу *rd* који смо имали раније). Када је контролни сигнал *str* укључен, тада ће вредност на улазу D бити уписана у локацију на адреси A у тренутку наиласка улазног руба часовника (тј. улаз *str* одговара улазу *wr* који смо имали раније).

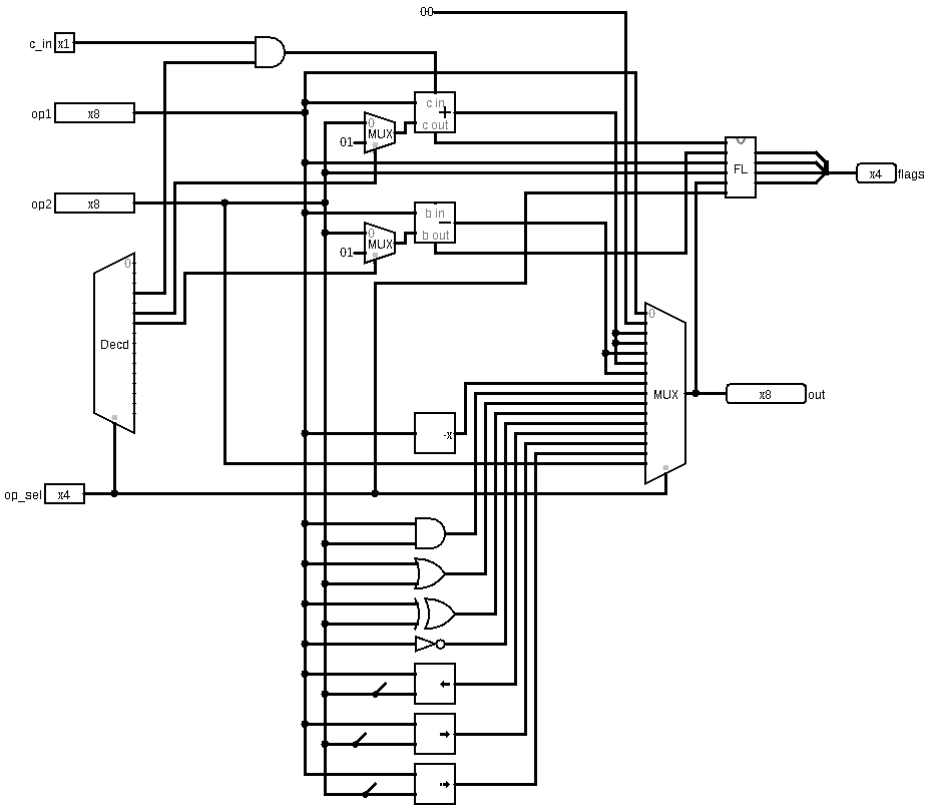
ALU јединица. Аритметичко логичка јединица (ALU) је осмобитна и подржава 16 различитих операција (отуда је улаз за избор операције четворобитни). Подржане операције дате су у табели 5.1.

Код операције	**Симболичка ознака**	**Опис операције**
0000	[no_op1]	$out = x$
0001	[zero]	$out = 0$
0010	[add]	$out = x + y$
0011	[addc]	$out = x + y + c_{in}$
0100	[sub]	$out = x - y$
0101	[inc]	$out = x + 1$
0110	[dec]	$out = x - 1$
0111	[neg]	$out = -x$
1000	[and]	$out = x\&y$
1001	[or]	$out = x\|y$
1010	[xor]	$out = x\hat{}y$
1011	[not]	$out = \tilde{}x$
1100	[shl]	$out = x << y$
1101	[shr]	$out = x >> y$
1110	[sar]	$out = x >>> y$
1111	[no_op2]	$out = y$

Табела 5.1: Операције ALU јединице

Ознака *out* означава вредност на излазу ALU јединице, и изражена је у опису операција као функција од улаза x, y и c_{in}. За опис операције коришћена је C-овска нотација, уз напомену да $x >> y$ означава логичко, а $x >>> y$ аритметичко померање у десно. Поред уобичајених операција, ова ALU јединица подржава и три „необичне" операције. Прве две су [no_op1] и [no_op2]. Ове операције омогућавају да се први, односно други улаз ALU јединице пропусти на излаз као резултат, без икакве измене. Ово је корисно приликом реализације операције премештања података. Трећа је операција [zero] која омогућава да се на излазу ALU јединице произведе нула, када је потребно иницијализовати неки регистар. Шема имплементације ове ALU јединице дата је на слици 5.2.

Приметимо да се операције [add], [addc] и [inc] изводе помоћу истог сабирача, с тим што се у случају [inc] операције на други улаз сабирача



Слика 5.2: Имплементација ALU јединице

доводи јединица (тај избор се постиже помоћу 2-1 мултиплексера), док се у случају [addc] операције на улаз за претходни пренос сабирача (означен са c_{in}) доводи вредност c_{in} улаза *ALU* јединице. Вредност c_{in} улаза ће бити једнака преносу (тј. прекорачењу) приликом претходне операције, чиме се операцијом [addc] ефективно омогућава софтверско уланчавање сабирача, у циљу сабирања бројева који не могу стати у 8 бита (као што је раније описано приликом проучавања сабирача). Слично, операције [sub] и [dec] се реализују уз помоћ истог одузимача.

Такође, приметимо да поред главног излаза (означеног са *out*) ALU јединица има и додатни четворобитни излаз који даје вредност израчунатих *флегова* (енгл. *flags*). Флегови су битови који описују статус извршене операције, тј. квалитативно описују резултат израчунавања. Притом, сваки бит кодира једну конкретну особину резултата. Четири основна флега који се јављају на већини модерних архитектура (можда не увек под тим именом) су:

- C или CF (carry flag): овај флег ће имати вредност 1 у следећим случајевима:
  - при операцијама [add], [addc], [inc], [sub], [dec], ако је дошло до неозначеног прекорачења

  - при операцијама [shl], [shr], [sar], ако је вредност последњег истиснутог бита једнака 1

  Иначе, вредност овог бита је 0.

- Z или ZF (zero flag): овај флег ће имати вредност 1 акко је резултат једнак 0, при свим операцијама.
- S или SF (sign flag): овај флег ће имати вредност највишег бита резултата, при свим операцијама.
- O или OF (overflow flag): овај флег ће имати вредност 1 у следећим случајевима:
  - при операцијама [add], [addc], [sub], [inc], [dec], ако је дошло до означеног прекорачења
  - при операцији [shl], ако је знак резултата (највиши бит) различит од знака улаза x
  - при операцији [neg], ако је $x = -128$ (тј. ако потпуни комплемент улаза x не може стати у 8 бита)

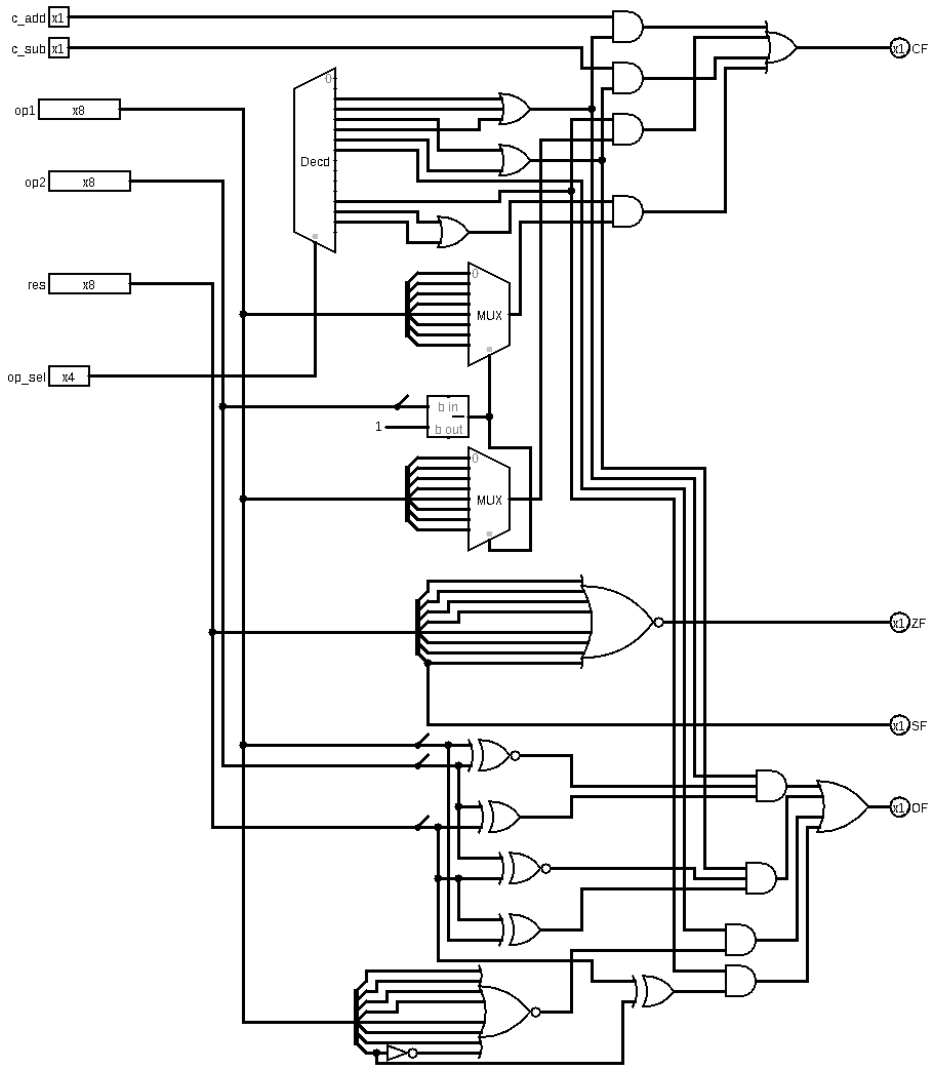
  Иначе, вредност овог бита је 0.

Због једноставности шеме, логичка кола за имплементацију израчунавања флегова издвојени су у посебно подколо (означено на горњој слици са *FL*). Његова имплементација дата је на слици 5.3.

Регистри специјалне намене. У питању су три регистра, означени са P, A и PSW. Сва три регистра своје стање мењају на узлазном рубу часовника. Осмобитни регистар P је помоћни регистар чији је улаз повезан на магистралу, док је његов излаз повезан на други улаз ALU јединице. Његова улога је да привремено сачува други операнд неке бинарне операције. Овако нешто је неопходно, с обзиром да имамо само једну магистралу преко које се у једном тренутку може преносити само један податак, па није могуће истовремено из меморије на улазе ALU јединице допремити оба операнда. Због тога ћемо у случају бинарних операција најпре други операнд путем магистрале доставити у P регистар, а затим ћемо у следећем циклусу часовника први операнд доставити путем магистрале директно на први улаз ALU јединице (док ће други операнд ALU јединица добијати из P регистра).

Осмобитни регистар A служи сличној сврси – за чување резултата ALU јединице. Наиме, како имамо само једну магистралу, није могуће резултат израчунавања одмах са излаза ALU јединице послати ка свом одредишту у меморији, јер је у том тренутку на магистрали вредност првог операнда. Како је ALU јединица комбинаторно коло, она не може запамтити то што је израчунала, па ће вредност са њеног излаза нестати чим вредности њених улаза буду уклоњене. Решење је да се израчуната вредност сачува у регистру A чији је улаз управо повезан на излаз ALU јединице. У наредном циклусу, када се магистрала ослободи, можемо послати вредност регистра A у своје одредиште у меморији.[1]

[1] Напоменимо да у реалним системима обично постоји више интерних магистрала унутар процесора којима се повезују његове компоненте, тако да је могуће у истом циклусу доводити и операнде на улазе ALU јединице и истовремено резултат израчунавања пребацивати у одредиште.



Слика 5.3: Имплементација израчунавања флегова

Четворобитни регистар *PSW* (енгл. *program status word*) служи за чување флегова које такође израчунава ALU јединица. Ови флегови се морају сачувати како би били на располагању контролној јединици, у циљу одређивања следећег корака програма. Због тога се вредност овог регистра спроводи на улаз контролне јединице. Најнижи бит овог регистра, који чува C флег, се такође доводи на улаз c_{in} ALU јединице, како би се могао користити за софверско уланчавање сабирања, како је раније описано.

Контролна јединица. Контролна јединица на улазу има вредност *PSW* регистра, док на излазу генерише *контролне сигнале* којима се управља осталим компонентама рачунара. У нашем примеру, у питању су следећи

контролни сигнали:

- adr: четворобитни сигнал који селектује меморијску локацију из које се врши читање или у коју се врши упис
- reg_{in}: сигнал који захтева упис податка са магистрале у меморију, на задату адресу
- reg_{out}: сигнал који захтева читање податка из меморије са задате адресе и постављање на магистралу
- p_{in}: сигнал који захтева од P регистра да прихвати и сачува вредност са магистрале
- alu: четворобитни сигнал који селектује операцију коју треба да изврши ALU јединица
- psw_{in}: сигнал који захтева од PSW регистра да сачува флегове које је израчунала ALU јединица
- a_{in}: сигнал који захтева од A регистра да сачува резултат који је на излазу ALU јединице
- a_{out}: сигнал који омогућава да се вредност A регистра постави на магистралу

Контролна јединица је у суштини један коначни аутомат чија стања одговарају корацима програма који се извршава, тј. говоре нам докле смо стигли у извршавању програма.

Сетимо се да сви регистри мењају своје стање на узлазном рубу часовника. Исто тако, и упис у меморију се врши на узлазном рубу часовника. Са друге стране, стање контролне јединице (и њен излаз, тј. контролни сигнали) се не може мењати на узлазном рубу часовника, јер у том случају комуникација између контролне јединице и осталих компоненти не би функционисала на исправан начин. Наиме, након што контролна јединица на свом излазу формира контролне сигнале који одређују следећу операцију коју треба извршити, потребно је извесно време да ти сигнали стигну до одговарајућих компоненти, као и да те компоненте на те сигнале реагују. На пример, ALU јединици је потребно извесно време да изврши захтевану операцију, а њен резултат мора бити на улазу A регистра пре наиласка одговарајућег руба часовника на ком A регистар врши промену стања (заправо, и нешто раније, јер треба узети у обзир и време постаке A регистра). Слично, потребно је одређено време да се податак прочита из меморије, постави на магистралу и преко магистрале пребаци, на пример, на улаз P регистра, и то се мора десити пре руба часовника на коме P регистар врши промену стања. Како сви ови регистри своје стање мењају на узлазном рубу, јасно је да контролни сигнали морају бити формирани нешто раније. Сличан проблем се јавља и у обрнутом смеру – контролна јединица своје ново стање и вредности контролних сигнала на излазу формира на основу тренутног стања и вредности на улазу, тј. тренутне вредности PSW регистра. Како је након уписа нове вредности у PSW регистар (што се дешава на узлазном рубу часовника) потребно извесно време да се та вредност појави на излазу PSW регистра и спроведе до

улаза контролне јединице, као и да комбинаторна логика за одређивање новог стања у аутомату контролне јединице изврши своја израчунавања, јасно је да контролна јединица не може мењати своје стање на узлазном рубу, већ нешто касније. Један начин да се ово постигне био би да имамо два различита часовника исте фреквенције, али са извесним фазним померајем. Други, једноставнији начин кога ћемо се ми држати у нашем примеру је да имамо јединствен часовник, али да контролна јединица своје стање мења на силазном рубу. Током позитивног дела циклуса часовника контролна јединица израчунава ново стање и нове вредности контролних сигнала које се на излазу формирају на силазном рубу часовника. Током негативног дела циклуса часовника компоненте рачунара реагују на контролне сигнале и извршавају захтеване операције, а резултати тих операција се уписују у одговарајуће регистре на узлазном рубу часовника. Како би рачунар функционисао исправно, потребно је да трајања позитивног и негативног дела циклуса часовника буду дужа од максималних кашњења одговарајућих компоненти које врше израчунавања у тим деловима циклуса, као и проводника кроз које се сигнали преносе. Отуда часовник може бити и асиметричан, уколико је нпр. кашњење контролне јединице мање од максималног кашњења ALU јединице.

5.1.1 Програмирање рачунара са фиксираним програмом

Елементарне операције. Под елементарним операцијама нашег рачунара подразумевамо операције које је могуће обавити у једном циклусу часовника. Ове операције се могу поделити у две категорије: *операције трансфера*, код којих се неки податак копира са једне локације на другу, и *операције израчунавања*, код којих се у ALU јединици извршава нека аритметичка или логичка операција, а њен резултат се чува у A регистру. Прецизније, имамо следеће елементарне операције:

- $R_i \longrightarrow P$: трансфер вредности из меморијског регистра R_i у регистар P
- $R_i \; op \; P \longrightarrow A, PSW$: израчунавање операције $R_i \; op \; P$ у ALU јединици и смештање резултата и флегова у регистре A и PSW, респективно
- $A \longrightarrow R_i$: трансфер вредности из регистра A у меморијски регистар R_i
- $A \longrightarrow P$: трансфер вредности из регистра A у регистар P
- $A \; op \; P \longrightarrow A, PSW$: израчунавање операције $A \; op \; P$ у ALU јединици и смештање резултата и флегова у регистре A и PSW, респективно. Приметимо да није проблем то што се регистар A користи и као операнд и као одредиште, јер ће се као операнд користити стара вредност регистра A, а након извршене операције у регистар A биће уписана нова вредност. Иако ова нова вредност може утицати на ALU јединицу, то ће се десити тек након проласка узлазног руба часовника, када је вредност регистра A већ безбедно сачувана.

Сваки алгоритам се може представити као низ ових елементарних операција које се извршавају у одговарајућем редоследу. Као илустрацију, покажимо на који начин се могу неке операције које се често срећу у језицима вишег нивоа реализовати помоћу наших елементарних операција:

- Наредба доделе $R_i = R_j$:

  1. R_j $[no_op1]$ $P \longrightarrow A, PSW$
  2. $A \longrightarrow R_i$

  Дакле, додела се врши у два циклуса. У првом се вредност изворишног регистра R_j пропушта кроз ALU јединицу без икакве операције, тј. непромењена се чува у регистру A. У другом циклусу се вредност регистра A пребацује у одредишни регистар R_i.

- Бинарне операције, нпр. $R_i = R_j + R_k$:

  1. $R_k \longrightarrow P$
  2. R_j $[add]$ $P \longrightarrow A, PSW$
  3. $A \longrightarrow R_i$

  У првом циклусу се други операнд пребацује из R_k у регистар P. У другом циклусу се извршава бинарна операција у ALU јединици над податком R_j који се налази на магистрали и податком у регистру P, а резултат се смешта у регистар A. У трећем циклусу се резултат пребацује из регистра A у одредишни регистар R_i.

- Унарне операције, нпр. $R_i = -R_j$:

  1. R_j $[neg]$ $P \longrightarrow A, PSW$
  2. $A \longrightarrow R_i$

  У првом циклусу се операнд преко магистрале доставља ALU јединици која извршава одговарајућу унарну операцију и резултат смешта у регистар A. Приметимо да се вредност регистра P у овом случају не користи, јер се код унарних операција увек користи операнд на првом улазу ALU јединице. У другом циклусу се резултат из регистра A смешта у R_i.

- Операција упоређивања, нпр. $R_i < R_j$:

  1. $R_j \longrightarrow P$
  2. R_i $[sub]$ $P \longrightarrow A, PSW$

  У првом циклусу се вредност регистра R_j пребацује у регистар P. У другом циклусу се врши одузимање $R_i - P$, а разлика и флегови се смештају редом у A и PSW. Дакле, поређење се своди на одузимање, с обзиром да већ имамо одузимач у ALU јединици, тј. није потребно имплементирати засебан компаратор. Сама разлика није битна и она остаје у регистру A, тј. не уписује се у меморију. Оно што је битно су флегови, јер њихово стање одређује однос података који се пореде. На пример, укључен C флег значи да је $R_i < R_j$, ако ове податке

тумачимо као неозначене целе бројеве. Са друге стране, уколико R_i и R_j тумачимо као означене бројеве у потпуном комплементу, тада ће важити $R_i < R_j$ акко је $S \oplus O = 1$. Остале релације се разматрају на сличан начин, а услови над флеговима који морају да важе за различите релације дати су у табели 5.2.

Релација	**Неозначени**	**Означени**
$=$	$ZF = 1$	$ZF = 1$
$\neq$	$ZF = 0$	$ZF = 0$
$<$	$CF = 1$	$SF \oplus OF = 1$
$>$	$CF + ZF = 0$	$(SF \oplus OF) + ZF = 0$
$\leq$	$CF + ZF = 1$	$(SF \oplus OF) + ZF = 1$
$\geq$	$CF = 0$	$SF \oplus OF = 0$

Табела 5.2: Услови над флеговима који морају важити након одузимања, за различите релације, за неозначене бројеве и означене бројеве у потпуном комплементу. Симбол + означава дисјункцију битова, а $\oplus$ ексклузивну дисјункцију

Програм као коначни аутомат. *Програм* нашег рачунара састојаће се из низа *корака*. Сваки корак програма се извршава у одређеном стању контролне јединице, током једног циклуса часовника. За опис корака програма користићемо следећу нотацију:

$$\textit{текуће_стање})\ \textit{услов}\ ?\ \textit{операција}\ (\textit{ново_стање})$$

при чему *текуће_стање* означава стање у коме се тај корак извршава, *услов* представља услов под којим се тај корак извршава (изражен у терминима флегова), *операција* представља елементарну операцију која се извршава у том кораку, а *ново_стање* представља стање у које прелази контролна јединица након извршења тог корака. Притом, одређени кораци се могу извршавати и безусловно – у том случају услов нећемо наводити. Такође, могуће је да у неком кораку не постоји операција која се извршава, јер циљ може бити само да се под одређеним условом пређе у неко одређено стање.

Да бисмо имплементирали аутомат контролне јединице који реализује неки програм, приметимо да се сваки од корака програма веома једноставно може описати у терминима прелаза коначног аутомата: *услов* представља улаз за који се врши тај прелаз у *текућем_стању*, а *операција* описује излаз при том прелазу (тачније, излаз ће бити контролни сигнали који су потребни да би се баш та операција извршила у том циклусу). *ново_стање* представља ново стање аутомата у које се одлази при том прелазу.

Као илустрацију, посматрајмо једноставан програм који израчунава минимум два неозначена броја који се налазе у регистрима R_0 и R_1, а резултат се смешта у регистар R_2. На језику високог нивоа, овај програм би се могао описати на следећи начин:

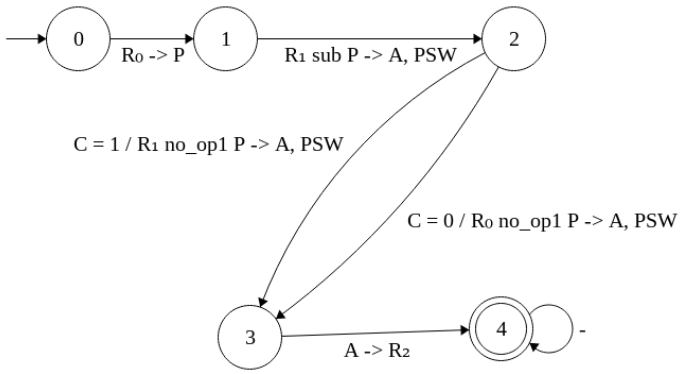
```
if(R0 > R1)
  R2 = R1;
```

```
else
  R2 = R0;
```

Превођењем овог програма на језик нашег рачунара, добијамо програм који се састоји из следећих корака:

- 0) $R_0 \longrightarrow P$ (1)
- 1) R_1 [sub] $P \longrightarrow A, PSW$ (2)
- 2) $C = 1$? R_1 [no_op1] $P \longrightarrow A, PSW$ (3)
- 2) $C = 0$? R_0 [no_op1] $P \longrightarrow A, PSW$ (3)
- 3) $A \longrightarrow R_2$ (4)
- 4) $-$ (4)

У почетном стању 0 аутомата врши се пребацивање податка из R_0 у регистар P и прелази се у стање 1, док се у стању 1 врши упоређивање (одузимањем) регистра R_1 са регистром P (тј. са вредношћу регистра R_0) и прелази се у стање 2. Оба ова корака извршавају се безусловно. У стању 2 се у зависности од вредности C флега врши пребацивање одговарајућег меморијског регистра у A (користећи [no_op1] операцију). Уколико је $C = 1$, то значи да је умањеник R_1 мањи од умањиоца R_0 (у неозначеном смислу), па се зато регистар R_1 пребацује у A, док се у супротном регистар R_0 пребацује у A. У сваком случају, након овог трансфера се прелази у стање 3, у ком се врши трансфер из регистра A у регистар R_2. Након тога се прелази у стање 4 које представља завршно стање. У овом стању се не врши никаква операција, а аутомат остаје у том истом стању заувек, чиме је извршавање програма завршено. Приказ графа аутомата контролне јединице дат је на слици 5.4.



Слика 5.4: Граф аутомата контролне јединице за рачунање минимума два броја

Имајући ово у виду, можемо формирати таблице ексцитације аутомата контролне јединице (табела 5.3). С обзиром да имамо пет стања, за чување стања довољна су три JK флип-флопа, а одговарајући битови

стања означени су са S_2, S_1 и S_0, почев од бита највише тежине. Битови новог стања означени су са S_2', S_1' и S_0'. Од улазних битова, једино је релевантан C флег, јер се остали флегови у програму не користе. Ово значајно поједностављује имплементацију, јер ће битови следећег стања, као и излазни битови бити функције од 4 бита (S_2, S_1, S_0 и C), уместо од 7 (ако бисмо узимали у обзир и остала три флега на улазу). Акције које је потребно извршити кодирају се одговарајућим вредностима контролних сигнала. На пример, да бисмо извршили операцију $R_0 \longrightarrow P$ у кораку 0, морамо укључити сигнал reg_{out}, а адресу adr поставити на 0000, како би вредност регистра R_0 била послата на магистралу. Истовремено, потребно је укључити сигнал P_{in}, како би регистар P на следећем узлазном рубу запамтио вредност која му долази на улаз са магистрале. Слично, операција у кораку 1 се кодира тако што се укључи сигнал reg_{out}, а адреса adr постави на 0001, како би се вредност регистра R_1 послала на магистралу. Истовремено, сигнал alu се поставља на вредност 0100, што је код операције [sub] (табела 5.1). Како би вредност коју израчунава ALU и флегови били сачувани у регистрима A и PSW респективно, укључују се сигнали a_{in} и psw_{in}. Слично се кодирају и операције у осталим корацима. Приметимо да у случају да се корак извршава безусловно, тада ће исти контролни сигнали бити активирани и за $C = 1$ и за $C = 0$, тј. две суседне врсте у таблици ће бити идентичне. Са друге стране, у кораку 2, контролни сигнали се разликују у зависности од вредности флега C на улазу. Најзад, у сваком кораку се кодира и следеће стање, а затим се на основу таблице ексцитације JK флип-флопа одређују вредности J и K улаза за сваки од битова стања.

Након попуњавања таблице ексцитације, све колоне се посматрају као функције од S_2, S_1, S_0 и C. Након минимизације, коју остављамо читаоцу за вежбу, добијају се следећи изрази:

$$J_2 = S_0S_1, K_2 = 0, J_1 = K_1 = S_0, J_0 = \overline{S_2}, K_0 = 1$$
$$adr[3] = adr[2] = 0, adr[1] = S_0S_1, adr[0] = S_0\overline{S_1} + \overline{S_0}S_1C$$
$$p_{in} = \overline{S_0}\,\overline{S_1}\,\overline{S_2}, a_{in} = psw_{in} = S_0\overline{S_1} + \overline{S_0}S_1, a_{out} = S_0S_1$$
$$reg_{in} = S_0S_1, reg_{out} = \overline{S_1}\,\overline{S_2} + \overline{S_0}\,\overline{S_2}$$
$$alu[3] = alu[1] = alu[0] = 0, alu[2] = S_0\overline{S_1}$$

Имплементација контролне јединице приказана је на слици 5.5. С обзиром да се функције које израчунавају поједине излазне битове поклапају, могуће су одређене уштеде у логичким колима, као и у флип-флоповима који чувају стање излаза (уместо 14, имамо 6 D флип-флопова за чување стања излаза).

Уколико бисмо желели да наш рачунар извршава неки други програм, тада би било потребно постојећу контролну јединицу заменити другом која имплементира тај други програм. Остатак рачунара би остао непромењен. Илустрације ради, посматрајмо један мало сложенији пример програма који израчунава највећи заједнички делилац (НЗД) два неозначена цела броја већа од нуле који се налазе у регистрима R_0 и R_1. Резултат треба сместити у R_0. На језику вишег нивоа, имали бисмо следећи програм:

```
while(R0 != R1)
{
```

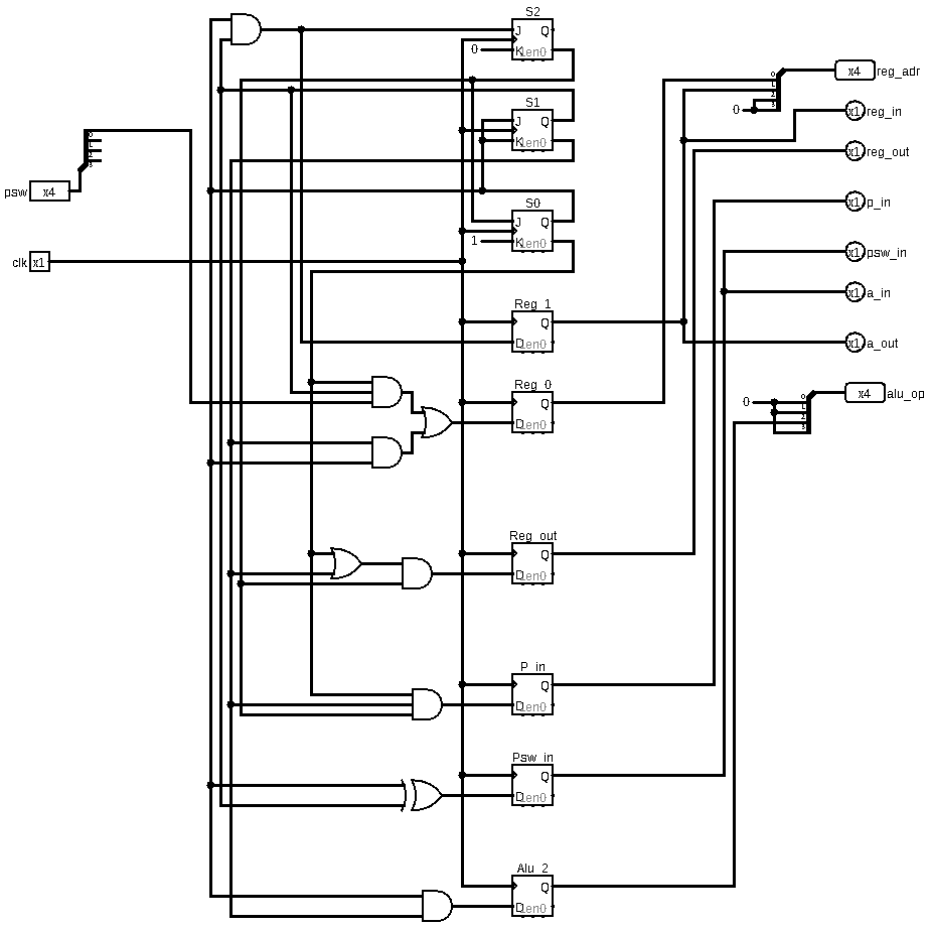
S_2	S_1	S_0	C	S_2'	S_1'	S_0'	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	0	0	0	1	0	-	0	-	1	-
0	0	0	1	0	0	1	0	-	0	-	1	-
0	0	1	0	0	1	0	0	-	1	-	-	1
0	0	1	1	0	1	0	0	-	1	-	-	1
0	1	0	0	0	1	1	0	-	-	0	1	-
0	1	0	1	0	1	1	0	-	-	0	1	-
0	1	1	0	1	0	0	1	-	-	1	-	1
0	1	1	1	1	0	0	1	-	-	1	-	1
1	0	0	0	1	0	0	-	0	0	-	0	-
1	0	0	1	1	0	0	-	0	0	-	0	-
1	0	1	0	-	-	-	-	-	-	-	-	-
1	0	1	1	-	-	-	-	-	-	-	-	-
1	1	0	0	-	-	-	-	-	-	-	-	-
1	1	0	1	-	-	-	-	-	-	-	-	-
1	1	1	0	-	-	-	-	-	-	-	-	-
1	1	1	1	-	-	-	-	-	-	-	-	-

S_2	S_1	S_0	C	$adr[3]$	$adr[2]$	$adr[1]$	$adr[0]$	reg_{in}	reg_{out}
0	0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	1
0	0	1	0	0	0	0	1	0	1
0	0	1	1	0	0	0	1	0	1
0	1	0	0	0	0	0	0	0	1
0	1	0	1	0	0	0	1	0	1
0	1	1	0	0	0	1	0	1	0
0	1	1	1	0	0	1	0	1	0
1	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0
1	0	1	0	-	-	-	-	-	-
1	0	1	1	-	-	-	-	-	-
1	1	0	0	-	-	-	-	-	-
1	1	0	1	-	-	-	-	-	-
1	1	1	0	-	-	-	-	-	-
1	1	1	1	-	-	-	-	-	-

S_2	S_1	S_0	C	p_{in}	psw_{in}	a_{in}	a_{out}	$alu[3]$	$alu[2]$	$alu[1]$	$alu[0]$
0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0	0	0	0
0	0	1	0	0	1	1	0	0	1	0	0
0	0	1	1	0	1	1	0	0	1	0	0
0	1	0	0	0	1	1	0	0	0	0	0
0	1	0	1	0	1	1	0	0	0	0	0
0	1	1	0	0	0	0	1	0	0	0	0
0	1	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0	0
1	0	1	0	-	-	-	-	-	-	-	-
1	0	1	1	-	-	-	-	-	-	-	-
1	1	0	0	-	-	-	-	-	-	-	-
1	1	0	1	-	-	-	-	-	-	-	-
1	1	1	0	-	-	-	-	-	-	-	-
1	1	1	1	-	-	-	-	-	-	-	-

Табела 5.3: Таблице експцитације за аутомат контролне јединице за рачунање минимума два броја (табела је, због прегледности, подељена на три дела)

```
if(R0 > R1)
  R0 = R0 - R1;
else
  R1 = R1 - R0;
```



Слика 5.5: Имплементација контролне јединице за пример рачунања минимума два броја

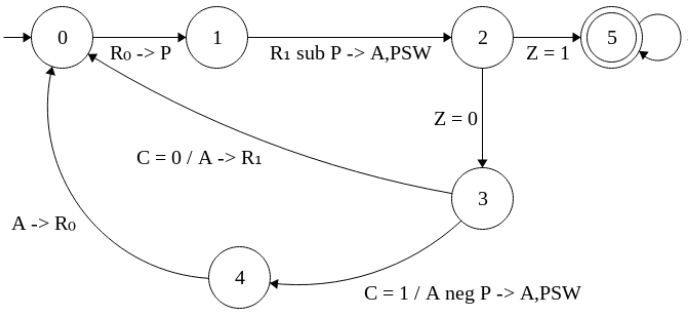
```
}
// Rezultat ostaje u R0
```

У питању је једна варијанта *Еуклидовог алгоритма*, код које се дељење симулира узастопним одузимањем. Уколико су R_0 и R_1 једнаки, тада је њихова заједничка вредност управо њихов НЗД. У супротном, већи од та два броја умањујемо за вредност мањег. Тиме оба броја остају позитивна, а њихов НЗД се не мења. Како се њихов збир неминовно смањује у свакој итерацији, овај ће се поступак завршити у коначном броју корака, а у регистру R_0 остаће НЗД полазних бројева. Овај програм, преведен на језик нашег рачунара изгледа овако:

- 0) $R_0 \longrightarrow P$ (1)
- 1) R_1 $[sub]$ $P \longrightarrow A, PSW$ (2)
- 2) $Z = 1$? $-$ (5)

- 2) $Z = 0$? $-$ (3)
- 3) $C = 0$? $A \longrightarrow R_1$ (0)
- 3) $C = 1$? A $[neg]$ $P \longrightarrow A, PSW$ (4)
- 4) $A \longrightarrow R_0$ (0)
- 5) $-$ (5)

Кораци у стањима 0 и 1 рачунају $R_1 - R_0$ и разлику смештају у A, а флегове у PSW. У стању 2 се испитује Z флег. Ако је $Z = 1$, тада је $R_0 = R_1$, па се прелази у стање 5, које је завршно стање (тј. програм се завршава, а у R_0 остаје израчунати НЗД). У супротном, прелази се у стање 3. У оба случаја, у стању 2 се не врши никаква операција (тј. сви контролни сигнали на излазу ће бити искључени). У стању 3 се проверава C флег. Ако је $C = 0$, то значи да је $R_1 > R_0$, па треба извршити *else* грану, тј. разлику $R_1 - R_0$ (која је већ у A регистру) треба пребацити у регистар R_1. Након извршења ове операције враћамо се у стање 0, ради поновне провере услова петље. У супротном, имамо да је $R_1 < R_0$, па је потребно разлику $R_0 - R_1$ (чија је вредност супротна од вредности A регистра) сместити у регистар R_0. Ово радимо тако што примењујемо операцију промене знака [neg] на вредност у регистру A, добијени резултат поново смештамо у A и прелазимо у стање 4, у коме ће вредност регистра A бити пребачена у R_0. Из стања 4 се прелази поново у стање 0, тј. на поновну проверу услова петље. Граф аутомата приказан је на слици 5.6.



Слика 5.6: Граф аутомата контролне јединице за рачунање НЗД два броја

Читаоцу за вежбу остављамо формирање таблице ексцитације аутомата, као и реализацију одговарајуће контролне јединице. За чување стања аутомата ће и овога пута бити довољно три JK флип-флопа, с обзиром да имамо 6 различитих стања. С обзиром да се у програму користе два флега, Z и C, сви контролни сигнали, као и J и K улази флип-флопова биће функције од 5 битова, што ће таблице ексцитације учинити дупло већим у односу на претходни пример. Такође, минимизација функција од 5 променљивих уз помоћ Карноових мапа је нешто тежа. Алтернативно, можемо користити метод Квин-Мекласког, а можемо применити и декомпозицију функција, а затим за њихову имплементацију користити мултиплексере, као што је описано у одељку 3.1.1.

Уопште, основни проблем описаног поступка је његова сложеност. Наиме, у случају иоле сложенијег програма имаћемо велики број корака програма, па самим тим и велики број стања аутомата, што ће таблице ексцитације учинити веома великим, а процес минимизације функција исувише компликованим. Наравно, ово стоји ако поступак спроводимо „ручно", на папиру. На срећу, данас постоји велики број готових алата који аутоматизују овај поступак и тиме га чине веома једноставним, јер се од дизајнера контролне јединице у том случају само очекује да опише прелазе аутомата (на неком симболичком језику, попут нашег описа горе).

5.2 Рачунари са ускладиштеним програмом

Основни проблем рачунара из претходног одељка је то што је програм био фиксиран конфигурацијом аутомата контролне јединице. Другим речима, програм је био уграђен у сам хардвер рачунара, док је меморија садржала искључиво податке са којима програм ради. Програмирање таквог рачунара се сводило на реконфигурацију аутомата, тј. на модификацију самог хардвера.

Идеја која се јавила врло брзо по настанку првих електронских рачунара је да се и програм, као и подаци, кодира у облику бинарних бројева и као такав смести у меморију рачунара. Како се сваки програм састоји из низа елементарних операција које се могу директно извршити на нашем хардверу (а то су, као што смо видели, операције копирања података и елементарне рачунске операције које подржава аритметичко-логичка јединица), свака од тих елементарних операција се мора засебно кодирати одговарајућим низом битова. Овако бинарно записана елементарна операција се назива и *машинска инструкција*, а програм састављен из низа оваквих елементарних операција назива се и *машински програм*. Задатак нашег рачунара је да сада узима инструкције машинског програма из меморије једну по једну, утврђује њихово значење (тј. декодира их) и извршава њихов ефекат користећи хардвер рачунара (аритметичко-логичку јединицу, регистре и магистрале које их повезују).

Подразумевано, инструкције које чине машински програм се извршавају једна за другом, у оном поретку у ком су наведене у меморији. Како би се обезбедила могућност сложеније контроле тока (гранање, петље), поред инструкција копирања података и аритметичко-логичких инструкција неопходне су и инструкције *контроле тока* којима се подразумевани редослед извршавања инструкција може по потреби променити. Ова промена тока извршавања програма се врши *скоком* на инструкцију од које желимо да наставимо извршавање програма. Ови скокови могу бити *безусловни* – када скачемо увек, или *условни* – када се скок врши само ако је испуњен неки услов изражен у терминима вредности флегова процесора. Скуп свих машинских инструкција које рачунар разуме и подржава чине *машински језик* датог рачунара.

Рачунари код којих се алгоритам који треба извршити кодира на описани начин у виду машинског програма називају се *рачунари са ускладиштеним програмом* (енгл. *stored program computers*). Код оваквих рачунара, процес програмирања се своди на изражавање алгоритма на машинском језику рачунара и унос одговарајућег машинског програма у

меморију рачунара. Након тога треба на одговарајуће меморијске локације унети и улазне податке и започети извршавање програма. Након што се извршавање програма заврши, у предвиђеним локацијама у меморији налазиће се резултати израчунавања које можемо прочитати. Када желимо да наш рачунар извршава неки други алгоритам, потребно је само да постојећи програм у меморији заменимо другим програмом. Дакле, процес програмирања и коришћења оваквог рачунара је далеко једноставнији него у случају рачунара са фиксираним програмом. С обзиром да је програме сада знатно лакше уносити у рачунар, брисати, модификовати и замењивати другим програмима, термин који се усталио за ускладиштене програме је – *софтвер* (енгл. *software*) који указује на то да је у питању нешто мекше и флексибилније у односу на *хардвер* (енгл. *hardware*) који представља скуп физичких компоненти рачунара које је знатно теже мењати, реконфигурисати и прилагођавати без одговарајућег техничког знања и потребних алата.

Постоје две варијанте рачунара са ускладиштеним програмом које се разликују по томе да ли су меморијски простори који чувају програм и податке одвојени или не. Уколико имамо само једну меморију у којој чувамо и програм и податке, тада такве рачунаре називамо рачунарима *Фон-Нојмановог типа*. Са друге стране, ако постоје одвојени меморијски простори за програм и за податке, тада такве рачунаре називамо рачунарима *Харвардског типа*. Већина модерних рачунара спада у Фон-Нојманове рачунаре, те ћемо у наставку подразумевати овакав тип рачунара и нећемо то посебно наглашавати.

Са становишта дизајна контролне јединице, суштинска разлика између рачунара са фиксираним и ускладиштеним програмом је у алгоритму који имплементира аутомат контролне јединице. Код рачунара са фиксираним програмом, аутомат контролне јединице реализује специфичан алгоритам којим решавамо неки конкретан проблем (нпр. израчунавамо НЗД два броја). Код рачунара са ускладиштеним програмом, аутомат контролне јединице имплементира алгоритам *интерпретације* машинског језика, тј. алгоритам који интерпретира машинске инструкције (декодира их и извршава њихов ефекат). Да бисмо ово боље разумели, можемо потражити аналогију у свету софтвера. Узмимо за пример програмски језик *Python*. Програми написани на овом програмском језику се *интерпретирају*. То значи да постоји програм који се зове *интерпретатор* који на улазу има *Python* програм и улазне податке тог програма. Интерпретатор узима једну по једну наредбу нашег *Python* програма, анализира је (тј. декодира њено значење), а затим ефекат те наредбе извршава над подацима програма. На крају, на излазу интерпретатор даје резултат извршавања датог *Python* програма над датим улазним подацима. Дакле, имамо један програм (написан на неком програмском језику, небитно ком) који имплементира алгоритам који *интерпретира* (*опонаша*) ефекат свих других алгоритама (изражених програмима написаним у језику *Python*). На исти начин, код рачунара са ускладиштеним програмом имамо контролну јединицу чији аутомат имплементира алгоритам који опонаша било који други алгоритам изражен на датом машинском језику. Дакле, контролна јединица рачунара са ускладиштеним програмом представља *хардверски интерпретатор*

машинског језика.[2]

Како би контролна јединица могла да интерпретира машински програм, неопходно је да у сваком тренутку зна до које машинске инструкције се стигло у извршавању програма. У ту сврху рачунари са ускладиштеним програмом имају један посебан регистар специјалне намене који се назива *програмски бројач* (енгл. *program counter*) који ћемо означавати са PC[3]. Вредност овог регистра представља меморијску адресу на којој се налази машинска инструкција која је на реду да се изврши. С обзиром да се инструкције машинског програма подразумевано извршавају редом којим су дате у меморији, вредност програмског бројача ће се подразумевано увећавати након сваке извршене инструкције, како би PC показивао на следећу инструкцију која је на реду да се изврши (ово објашњава назив *програмски бројач*, јер се овај регистар подразумевано понаша као бројачки регистар који се увећава након сваке извршене инструкције). Изузетак од овог понашања представљају инструкције скока, код којих се након њиховог извршења програм може наставити од произвољне инструкције у програму (чија адреса је обично задата као параметар инструкције који називамо и *операнд*). У том случају се вредност адресе инструкције на коју се скаче просто уписује у програмски бројач, чиме се извршење програма наставља од те инструкције.

Да би инструкција на коју указује програмски бројач била извршена, потребно је проследити је контролној јединици која ће на основу садржаја инструкције одлучити о даљим корацима. Дакле, рад коначног аутомата контролне јединице више не зависи само од стања флегова, већ и од тога која се инструкција тренутно извршава, с обзиром да различите инструкције захтевају различите кораке. Заправо, извршавање многих инструкција неће уопште ни зависити од стања флегова. Изузетак представљају инструкције условног скока, које врше скок у случају да је одређени услов (изражен у терминима флегова) испуњен, док у супротном не раде ништа. Ипак, постоје архитектуре код којих и друге инструкције, поред инструкција условног скока, могу бити условно извршене.[4] У том случају ће и код тих инструкција рад аутомата контролне јединице зависити и од садржаја инструкције, али и од стања флегова.

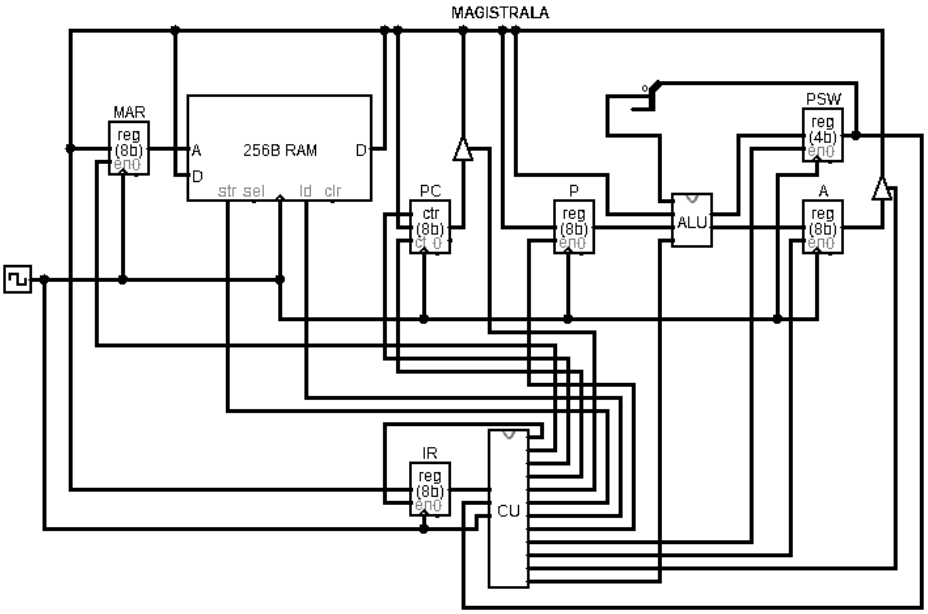
Када се инструкција на коју указује програмски бројач проследи контролној јединици, она мора да је негде запамти, како би у даљим корацима могла да је користи. У ту сврху постоји још један специјални

[2]Приметимо да у случају софтверске интерпретације нпр. *Python* програма, ми заправо имамо интерпретацију на два нивоа. Наиме, сам *Python* интепретатор је један машински програм (обично написан на неком вишем програмском језику, па затим преведен на машински помоћу компилатора). Тај машински програм тумачи наредбе Python програма и опонаша их одговарајућим машинским инструкцијама. Те машинске инструкције се на нижем нивоу интерпретирају помоћу хардверског интерпретатора – контролне јединице, а њихов ефекат се извршава на хардверу рачунара. Са друге стране, код програмских језика који се *компилирају* (попут C-а или C++-а), имамо интерпретацију само на нивоу машинског програма, с обзиром да се програми код ових програмских језика преводе директно на машински језик, а затим у том облику извршавају.

[3]У реалним рачунарима, овај регистар често има неко друго име, нпр. на архитектури *x86-64* овај регистар носи назив *RIP*. Ипак, ми ћемо се држати назива PC који се усталио у литератури.

[4]Екстремни пример тога је 32-битна верзија архитектуре *ARM32*, код које готово све инструкције могу бити извршене условно.

регистар који се назива *инструкцијски регистар*, а који ћемо означавати са IR. Излаз овог регистра повезан је на улаз контролне јединице, како би иста била све време свесна инструкције која се тренутно извршава.



Слика 5.7: Пример рачунара са ускладиштеним програмом

Пример једноставног рачунара са ускладиштеним програмом дат је на слици 5.7. У питању је рачунар добијен минималном модификацијом рачунара са фиксираним програмом датом на слици 5.1 (поглавље 5.1). Поред тога што су додати регистри PC и IR, повећана је меморија на 256 бајтова, те су сада меморијске адресе осмобитне (ово није суштинска промена, али јесте корисна, с обзиром да ће нам сада и програми бити у меморији, згодно је имати више меморије). Као што се може видети, на улаз контролне јединице се сада, поред излаза PSW регистра, доводи и излаз регистра IR. Са друге стране, вредност која се уписује у IR регистар долази са магистрале, како бисмо могли да инструкцију из меморије пребацимо у IR. Оно што такође можемо приметити је да сада адреса која се шаље меморији више није контролни сигнал који генерише контролна јединица. Ово је зато што адресе које се користе више нису уграђене у коначни аутомат контролне јединице, већ су задате програмом који се налази у меморији. Постоје две ситуације у којима нам је потребан приступ меморији:

- када је потребно учитати следећу инструкцију из меморије. У том случају се адреса налази у програмском бројачу PC.
- када је потребно учитати податак над којим је потребно извршити операцију, или сачувати резултат операције. У том случају је адреса дата као део инструкције, тј. представља операнд инструкције и налази се у меморији.

У оба ова случаја, потребно је адресу некако довести на адресни улаз меморије. У ту сврху се користи још један специјални регистар – *регистар меморијских адреса* (енгл. *memory address register*), означен са MAR. Улаз овог регистра повезан је на магистралу, а излаз је директно повезан на адресни улаз меморије, те је за адресирање меморије довољно жељену адресу путем магистрале ископирати у овај регистар.

Додатни контролни сигнали које генерише контролна јединица рачунара са слике 5.7 (поред оних који су постојали и код рачунара са фиксираним програмом са слике 5.1) су:

- *ir_in*: вредност са магистрале се уписује у регистар IR
- *mar_in*: вредност са магистрале се уписује у регистар MAR
- *pc_in*: вредност са магистрале се уписује у регистар PC
- *pc_inc*: вредност регистра PC се увећава за један
- *pc_out*: вредност регистра PC се пушта на магистралу

Машински језик. У нашем примеру, претпоставићемо да су инструкције величине 8 или 16 бита, тј. да заузимају један или два суседна бајта у меморији. Притом, бајт на нижој адреси представља тзв. *операциони код* инструкције који говори која операција је у питању (нпр. сабирање, копирање, упоређивање), а бајтом на вишој адреси се задаје *операнд*, тј. податак над којим се врши дата операција (у случају да инструкција нема операнд, тада ће се она састојати само из једног бајта).

У случају бинарних аритметичко-логичких инструкција (попут инструкције сабирања), операнд инструкције представља други податак над којим се врши операција (други сабирак, умањилац, множилац, делилац, и сл.), док ће се први податак (први сабирак, умањеник, множеник, дељеник) имплицитно увек налазити у A регистру. Резултат сабирања ће се поново смештати у регистар A. Унарне аритметичке операције (попут промене знака или битовске негације) немају операнд, већ се операција извршава над регистром A, и резултат се опет уписује у регистар A. Скуп аритметичко-логичких инструкција рачунара у нашем примеру дат је у табели 5.4. Приметимо да ове инструкције одговарају операцијама које подржава наша ALU јединица (виша четири бита операционог кода одговарају коду операције ALU јединице из табеле 5.1). Све аритметичко-логичке инструкције ажурирају флегове у PSW регистру на основу добијеног резултата операције.

У случају инструкција трансфера (које ћемо називати LOAD и STORE), операнд представља податак који се учитава у регистар A, или податак у који се уписује вредност регистра A. Инструкције трансфера дате су у табели 5.5. Инструкција LOAD ажурира флегове на основу вредности која је уписана у регистар A (конкретно, ZF ако је учитана нула, а SF ако је учитан негативан број).

Најзад, у случају инструкција скока, операнд инструкције представља адресу инструкције на коју треба скочити, тј. од које треба наставити извршавање програма. Инструкције контроле тока нашег рачунара дате су у табели 5.6. Поред инструкција безусловног и условног скока, у

Инструкција	Операциони код	Операција
ADD const	00100000	A = A + const
ADD addr	00100001	A = A + MEM[addr]
ADD [addr]	00100010	A = A + MEM[MEM[addr]]
ADDC const	00110000	A = A + const + CF
ADDC addr	00110001	A = A + MEM[addr] + CF
ADDC [addr]	00110010	A = A + MEM[MEM[addr]] +CF
SUB const	01000000	A = A - const
SUB addr	01000001	A = A - MEM[addr]
SUB [addr]	01000010	A = A - MEM[MEM[addr]]
INC	01010000	A = A + 1
DEC	01100000	A = A - 1
NEG	01110000	A = -A
AND const	10000000	A = A & const
AND addr	10000001	A = A & MEM[addr]
AND [addr]	10000010	A = A & MEM[MEM[addr]]
OR const	10010000	A = A | const
OR addr	10010001	A = A | MEM[addr]
OR [addr]	10010010	A = A | MEM[MEM[addr]]
XOR const	10100000	A = A^const
XOR addr	10100001	A = A^MEM[addr]
XOR [addr]	10100010	A = A^MEM[MEM[addr]]
NOT	10110000	A =~A
SHL const	11000000	A = A << const
SHL addr	11000001	A = A << MEM[addr]
SHL [addr]	11000010	A = A << MEM[MEM[addr]]
SHR const	11010000	A = A >> const
SHR addr	11010001	A = A >> MEM[addr]
SHR [addr]	11010010	A = A >> MEM[MEM[addr]]
SAR const	11100000	A = A >>> const
SAR addr	11100001	A = A >>> MEM[addr]
SAR [addr]	11100010	A = A >>> MEM[MEM[addr]]
CMP const	11110000	A - const
CMP addr	11110001	A - MEM[addr]
CMP [addr]	11110010	A - MEM[MEM[addr]]

Табела 5.4: Скуп аритметичко логичких инструкција рачунара са слике 5.7

Инструкција	Операциони код	Операција
LOAD const	00000000	A = const
LOAD addr	00000001	A = MEM[addr]
LOAD [addr]	00000010	A = MEM[MEM[addr]]
STORE addr	00010001	MEM[addr] = A
STORE [addr]	00010010	MEM[MEM[addr]] = A

Табела 5.5: Инструкције трансфера рачунара са слике 5.7

инструкције контроле тока убрајамо и посебну инструкцију HALT којом се контролна јединица рачунара одводи у завршно стање у коме остаје заувек. Овим се зауставља даљи рад рачунара. Инструкцијом HALT се мора завршити сваки програм, иначе би рачунар наставио даље да учитава инструкције са следеће адресе у меморији. Напоменимо да инструкције контроле тока не утичу на вредност флегова у PSW регистру.

Инструкција	Операциони код	Услов	Операција
JMP const	00001100	-	PC = const
JMP addr	00001101	-	PC = MEM[addr]
JMP [addr]	00001110	-	PC = MEM[MEM[addr]]
JE const	00011100	ZF = 1	PC = const
JE addr	00011101	ZF = 1	PC = MEM[addr]
JE [addr]	00011110	ZF = 1	PC = MEM[MEM[addr]]
JNE const	00101100	ZF = 0	PC = const
JNE addr	00101101	ZF = 0	PC = MEM[addr]
JNE [addr]	00101110	ZF = 0	PC = MEM[MEM[addr]]
JA const	00111100	ZF + CF = 0	PC = const
JA addr	00111101	ZF + CF = 0	PC = MEM[addr]
JA [addr]	00111110	ZF + CF = 0	PC = MEM[MEM[addr]]
JB const	01001100	CF = 1	PC = const
JB addr	01001101	CF = 1	PC = MEM[addr]
JB [addr]	01001110	CF = 1	PC = MEM[MEM[addr]]
JAE const	01011100	CF = 0	PC = const
JAE addr	01011101	CF = 0	PC = MEM[addr]
JAE [addr]	01011110	CF = 0	PC = MEM[MEM[addr]]
JBE const	01101100	ZF + CF = 1	PC = const
JBE addr	01101101	ZF + CF = 1	PC = MEM[addr]
JBE [addr]	01101110	ZF + CF = 1	PC = MEM[MEM[addr]]
JG const	01111100	(SF $\oplus$ OF) + ZF = 0	PC = const
JG addr	01111101	(SF $\oplus$ OF) + ZF = 0	PC = MEM[addr]
JG [addr]	01111110	(SF $\oplus$ OF) + ZF = 0	PC = MEM[MEM[addr]]
JL const	10001100	(SF $\oplus$ OF) = 1	PC = const
JL addr	10001101	(SF $\oplus$ OF) = 1	PC = MEM[addr]
JL [addr]	10001110	(SF $\oplus$ OF) = 1	PC = MEM[MEM[addr]]
JGE const	10011100	(SF $\oplus$ OF) = 0	PC = const
JGE addr	10011101	(SF $\oplus$ OF) = 0	PC = MEM[addr]
JGE [addr]	10011110	(SF $\oplus$ OF) = 0	PC = MEM[MEM[addr]]
JLE const	10101100	(SF $\oplus$ OF) + ZF = 1	PC = const
JLE addr	10101101	(SF $\oplus$ OF) + ZF = 1	PC = MEM[addr]
JLE [addr]	10101110	(SF $\oplus$ OF) + ZF = 1	PC = MEM[MEM[addr]]
HALT	11111111	-	$\longrightarrow halt_state$

Табела 5.6: Инструкције контроле тока рачунара са слике 5.7

5.2.1 Алгоритам контролне јединице

Претпоставимо да у датом тренутку регистар PC садржи адресу наредне инструкције коју треба извршити у нашем програму. Поступак интерпретације инструкције и фазе из којих се он састоји описујемо у наставку.

Дохватање инструкције. Прва фаза у извршењу инструкције је увек *дохватање инструкције* (енгл. *fetch*). Циљ дохватања је да се операциони кôд инструкције пребаци у регистар IR. Ово се на нашем рачунару може урадити на следећи начин:

- најпре се адреса текуће инструкције из регистра PC пребацује у регистар MAR путем магистрале
- затим се са ове адресе очитава операциони кôд текуће инструкције и пребацује магистралом у IR

Овај низ корака се може описати користећи нотацију из претходног поглавља на следећи начин:

- 0) $PC \longrightarrow MAR$, pc_inc (1)
- 1) $MEM \longrightarrow IR$ (2)

Притом, ознака $MEM \longrightarrow IR$ означава да се податак пребацује из меморије у IR, са адресе коју тренутно садржи MAR регистар, док ознака pc_inc означава да се активира посебан сигнал контролне јединице за инкрементацију PC регистра. Овим се PC увећава за један, тј. поставља се да указује на други бајт инструкције који описује операнд (или на наредну инструкцију у програму у случају да текућа инструкција нема операнд).

Описани кораци у фази дохватања инструкције се извршавају безусловно, јер они не зависе од инструкције и извршавају се увек на исти начин.

Декодирање инструкције. Оног тренутка када се операциони код инструкције нађе у IR регистру, започиње процес *декодирања инструкције* (енгл. *decode*). Овај процес тече тако што се вредност регистра IR прослеђује контролној јединици, чија комбинаторна кола на основу ове вредности одређују наредни корак у извршавању инструкције, као и наредно стање у које контролна јединица прелази.

У једноставним рачунарима, декодирање се обично обавља у току једног циклуса, тј. већ након наредног откуцаја часовника рачунар може наставити са наредним корацима у извршавању инструкције. У сложенијим рачунарима формати инструкције могу бити знатно комплекснији, те је и процес декодирања сложенији и може захтевати више циклуса часовника. На пример, у многим рачунарима инструкције могу бити различитих дужина, те је након учитавања почетног дела инструкције потребно утврдити колико још бајтова инструкције треба учитати и извршити дохватање остатка инструкције.

Декодирање такође обично укључује одређивање броја и врсте операнада које инструкција има, као и одређивање њихових адреса. У нашем примеру рачунара, инструкције могу да имају један или ниједан операнд. Уколико инструкција нема операнд, тада не треба радити ништа, већ само треба прећи у наредну фазу извршавања инструкције (која у случају нашег рачунара почиње у стању 5 и описана је у наредном одељку):

- 2) – (5)

Уколико се на основу операционог кода инструкције у IR регистру утврди да инструкција има операнд, тада је потребно одредити његову адресу у меморији и пребацити је у MAR регистар како би могао бити очитан у следећој фази извршавања инструкције.

Први и најједноставнији случај је када операнд представља део саме инструкције, тј. када други бајт инструкција садржи саму вредност податка који треба употребити у инструкцији. Овакав операнд називамо *непосредан операнд* (енгл. *immediate operand*). Адреса овог операнда је управо адреса другог бајта инструкције, тј. управо адреса на коју указује регистар PC након што је увећан у фази дохватања инструкције. Отуда је за

израчунавање адресе непосредног операнда довољно вредност регистра PC убацити у MAR:

- 2) $PC \longrightarrow MAR, pc_inc$ (5)

Приметимо да се након учитавања другог бајта инструкције регистар PC поново увећава, тако да сада показује на следећу инструкцију у програму. Стање у које прелазимо је поново стање 5, тј. настављамо са следећом фазом извршавања инструкције.

Други случај је када се операнд налази негде у меморији, на засебној адреси и није непосредно део инструкције. Овакве операнде називамо *меморијски операнди* (енгл. *memory operand*). У случају меморијских операнада, потребно је на неки начин задати адресу операнда у меморији. Начин на који ћемо то урадити одређује *начин адресирања* (енгл. *addressing mode*). Најједноставнији начин адресирања је *директно адресирање* (енгл. *direct addressing*). Код овог начина адресирања, други бајт инструкције садржи апсолутну адресу операнда у меморији. У том случају, израчунавање адресе операнда се може обавити на следећи начин:

- 2) $PC \longrightarrow MAR, pc_inc$ (3)
- 3) $MEM \longrightarrow MAR$ (5)

Као и малопре, најпре пребацујемо вредност PC регистра у MAR, како бисмо очитали други бајт инструкције. Њега у другом кораку такође пребацујемо у MAR регистар, како бисмо у MAR регистру имали спремну адресу операнда за даљи ток извршавања инструкције. Програмски бројач PC је након овог низа корака увећан за један и сада указује на следећу инструкцију у програму. Стање у које одлазимо на крају је поново стање 5, као и у претходном случају.

Директно адресирање подразумева да се операнд налази на фиксираној локацији у меморији која је позната у фази писања програма, како би могла бити наведена као део саме инструкције. У реалним програмским језицима вишег нивоа, то је случај са, нпр. глобалним променљивама чија се адреса одређује у фази превођења програма на машински језик. Са друге стране, постоје променљиве у нашим програмима чија апсолутна адреса није позната. Типичан пример су локалне променљиве које немају статички животни век, већ се креирају у меморији у тренутку када се уђе у функцију у којој су декларисане, те ће њихова адреса бити одређена у току извршавања програма. Самим тим, оваквим променљивама није могуће приступати директним адресирањем. Слично, постоје ситуације у којима се нека операција у нашем програму не обавља увек над истим податком. Типичан пример су операције које раде са низовима (не користи се исти елемент низа у свакој итерацији) или операције које користе показиваче (показивач не мора показивати на исти податак све време). У оваквим ситуацијама такође не можемо приступати подацима директним адресирањем. Отуда постоји потреба за другачијим, флексибилнијим начинима адресирања. Један такав начин адресирања познат је као *индиректно адресирање*. Код овог начина адресирања, није нам позната адреса податка над којим треба извршити операцију, али нам је позната адреса на којој се налази адреса тог податка. У терминологији показивача,

можемо рећи да наша инструкција садржи адресу показивачке променљиве у меморији, а та променљива садржи адресу стварног податка над којим је потребно извршити операцију. Како се вредност те показивачке променљиве може мењати, инструкција не мора увек приступати истом податку, нити локација тог податка мора бити позната у току писања програма (али адреса показивачке променљиве мора).

У случају нашег рачунара, код инструкција које користе индиректно адресирање други бајт инструкције ће садржати адресу адресе податка над којим се операција извршава. Ако декодирањем утврдимо да инструкција захтева индиректно адресирање, тада је најпре потребно извршити следеће кораке:

- 2) $PC \longrightarrow MAR, pc_inc$ (3)
- 3) $MEM \longrightarrow MAR$ (4)
- 4) $MEM \longrightarrow MAR$ (5)

Дакле, у односу на директно адресирање, разлика је само у једном додатном кораку (корак у стању 4), којим се учитава вредност са адресе која је садржана у другом бајту инструкције. Тиме се вредност показивачке променљиве из меморије довлачи у MAR, тако да у наставку извршавања инструкције можемо директно из меморије очитати операнд. Стање у коме се контролна јединица налази на крају овог поступка је поново стање 5.

Напоменимо да се у описаном поступку израчунавања адресе операнда кораци извршавају условно, само ако је инструкција таква да очекује операнд и притом користи одговарајући начин адресирања. Ипак, у жељи да поједноставимо нотацију, те услове нисмо експлицитно наводили.

У табелама 5.4, 5.5 и 5.6 је за све инструкције које имају операнде наведено више облика које могу да узму, у зависности од начина адресирања. Сваки облик има различит операциони кôд, што омогућава контролној јединици да препозна жељени начин адресирања. Ако је у питању непосредни операнд, он је у табели означен са *const*. У случају директног адресирања, адреса податка је означена са *addr*, док је код индиректног адресирања адреса адресе податка означена са [*addr*]. У опису операције инструкције, ознака MEM[X] означава локацију у меморији на адреси X.

Извршавање инструкције. Фаза извршавања инструкције (енгл. *execute*) у ужем смислу представља низ конкретних радњи којим се извршава ефекат текуће инструкције. Ова фаза се може извршити тек након што се инструкција дохвати и декодира, тј. након што контролна јединица закључи о којој се инструкцији ради као и над којим подацима се инструкција извршава.

У примеру нашег рачунара, ова фаза почиње од стања 5, и у њој бисмо, у зависности од конкретне инструкције, имали следеће кораке:

- ако је у питању инструкција LOAD, тада је потребно само вредност операнда пропустити кроз ALU без промене и сачувати га у регистру A (уз ажурирање флегова):

  - 5) $MEM\ [no_op1]\ P \longrightarrow\ A, PSW\ (0)$

- ако је у питању инструкција STORE, тада је потребно пребацити вредност регистра A у меморију на дату адресу операнда која се налази у MAR регистру:

  - 5) $A \longrightarrow\ MEM\ (0)$

- ако је у питању аритметичко-логичка инструкција која нема операнд (унарна операција), тада се она може извршити на следећи начин:

  - 5) $A\ op\ P \longrightarrow\ A, PSW\ (0)$

  где је *op* одговарајућа унарна операција ALU јединице. Дакле, операција се изводи над вредношћу регистра A и резултат се поново смешта у регистар A, уз ажурирање флегова.

- у случају аритметичко-логичке инструкције која има операнд (бинарна операција), можемо извршити следеће кораке:

  - 5) $MEM \longrightarrow P\ (6)$
  - 6) $A\ op\ P \longrightarrow\ A, PSW\ (0)$

  где је *op* кôд одговарајуће бинарне операције ALU јединице. Дакле, најпре је потребно меморијски операнд (који је увек други операнд операције) пребацити у P регистар, а затим извршити операцију користећи вредност регистра A као први операнд операције. Резултат се поново смешта у A, уз ажурирање флегова.

- у случају инструкција скока, уколико одговарајући услов (изражен у терминима флегова) није испуњен, тада се у овој фази не ради ништа, већ се враћамо на фазу дохватања (стање 0), како бисмо дохватили следећу инструкцију из меморије (то је управо она инструкција која следи за текућом инструкцијом, с обзиром да смо у фази дохватања и декодирања увећали PC):

  - 5) $-\ (0)$

  У супротном, ако услов за скок јесте испуњен, примењујемо следећи корак:

  - 5) $MEM \longrightarrow PC\ (0)$

  Овим кораком операнд инструкције (који представља адресу на коју треба скочити) копирамо у PC регистар и враћамо се у стање 0, ради дохватања следеће инструкције са те адресе.

- у случају инструкције HALT, контролна јединица се преводи у завршно стање 7:

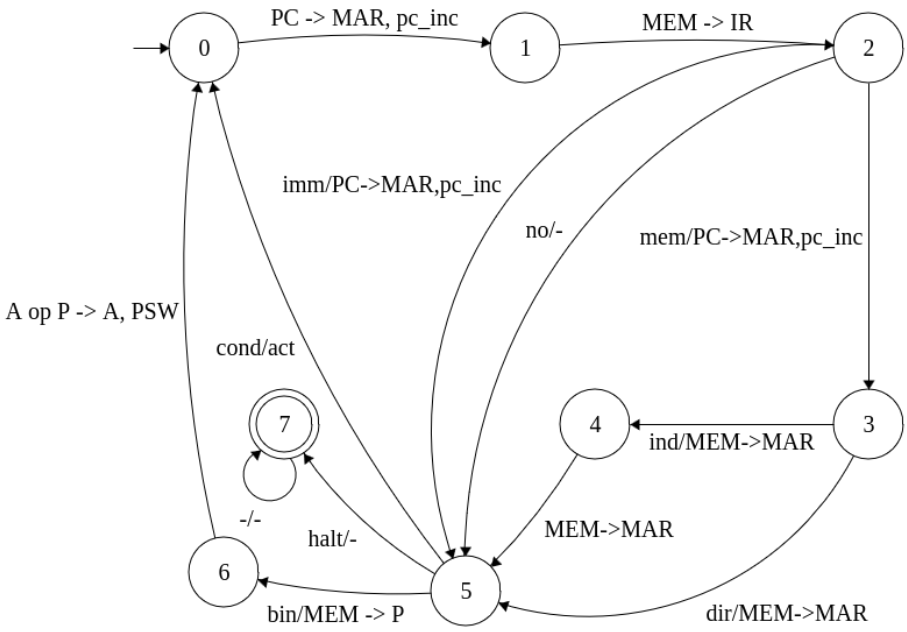
  - 5) $-\ (7)$

Као и раније, подразумевамо да се горе описани кораци извршавају условно (само ако је у питању одговарајућа инструкција), али те услове не наводимо експлицитно, због једноставности. Такође, приметимо да у свим случајевима на крају завршавамо у стању 0, чиме се поново започиње дохватање нове инструкције (осим у случају инструкције HALT, када завршавамо са извршавањем програма). У већини случајева, та следећа инструкција је управо она која непосредно следи у меморији након управо извршене. Једино у случају инструкција скока се може догодити да, ако је услов испуњен, извршавање настави од неке произвољне адресе коју уписујемо у програмски бројач.

5.2.2 Аутомат контролне јединице

Граф аутомата контролне јединице нашег рачунара дат је на слици 5.8. Као и обично, поред сваке гране стоји акција која се извршава при том преласку, а за оне кораке који се извршавају условно, пре знака '/' наведен је услов под којим се извршавају. Због једноставности, услови су записани симболички, а могу се детаљно расписати као функције од садржаја IR и PSW регистара. Фаза дохватања обухвата стања 0 и 1, а одговарајући преласци се обављају безусловно. У стању 2 започиње декодирање инструкције и простире се до стања 5. У случају да инструкција нема операнада (услов симболички записан као *no*), тада се само прелази у стање 5. Уколико инструкција има непосредни операнд (услов означен симболом *imm*), тада се поново прелази у стање 5, уз пребацивање адресе другог бајта инструкције у MAR. У случају да инструкција има меморијски операнд (услов означен симболом *mem*), прелази се у стање 3 како би се вредност другог бајта инструкције пребацила у MAR. Даље, у случају директног адресирања (симбол *dir*) одмах се прелази у стање 5, док је у случају индиректног адресирања (симбол *ind*) потребно прећи у стање 4, како би се вредност са адресе на коју указује други бајт инструкције учитала адреса операнда у MAR регистар, након чега се опет прелази у стање 5. У стању 5 започиње фаза извршавања инструкције. У случају бинарне аритметичко-логичке инструкције (услов симболички представљен ознаком *bin*), потребно је прећи у међустање 6, како би се најпре операнд пребацио у регистар P. Затим се у стању 6 врши одговарајућа операција и враћамо се у стање 0. У случају инструкције HALT (услов симболички представљен ознаком *halt*), одмах се из стања 5 прелази у завршно стање 7. У осталим случајевима, само се извршава одговарајућа операција која одговара инструкцији и стању флегова, а затим се прелази у стање 0. Ови „остали случајеви" су на слици 5.8 због недостатка простора представљени једном граном која је означена са *cond/act*, али у суштини је у питању скуп грана који укључује по једну грану за сваку од преосталих инструкција (а у случају инструкција скока имамо по две гране у зависности од тога да ли је услов испуњен или не).

Аутомат наше контролне јединице има 8 стања, па је стање могуће чувати помоћу три JK флип-флопа. Ипак, број улазних битова је знатно већи него код рачунара са фиксираним програмом, јер овде поред четири флега на улазу имамо и осам битова IR регистра. Отуда би конструкција горњег аутомата захтевала минимизацију функција реда 15, што није једноставно радити ручно. Као што је раније речено, у пракси се тај посао



Слика 5.8: Аутомат контролне јединице рачунара са слике 5.7

обично препушта аутоматским алатима.

5.2.3 Програмирање рачунара са ускладиштеним програмом

Програмирање рачунара са ускладиштеним програмом се састоји из изражавања алгоритама на машинском језику датог рачунара. Машинске инструкције су кодиране као бинарни бројеви (у нашем примеру, свака инструкција је један осмобитни или 16-битни бинарни број). Инструкције се смештају у меморију једна за другом, почев од неке локације од које рачунар подразумевано започиње рад када се укључи. У нашем примеру, претпоставићемо да је иницијална вредност програмског бројача по укључивању рачунара једнака 0. То значи да рачунар очекује да се програм у меморију унесе почев од те адресе. Подаци са којима програм ради се налазе у истој меморији, али на другим адресема (у нашем примеру, подаци ће бити на вишим адресама).

С обзиром да је кодирање машинских инструкција на бинарном језику напорно и нечитљиво, згодно је увести симболичке ознаке којима се записују инструкције. Овакве ознаке називамо *мнемоници*, и обично су такве да њихово значење буде интуитивно јасно. У нашем примеру, мнемоници су LOAD, STORE, ADD, SUB и тд. Такође, адресе операнада које инструкције користе се, уместо на бинарном језику, могу записивати симболички – овакве ознаке називамо *лабеле*. На пример, уместо да инструкцију

запишемо као:

```
00100001 11111110
```

једноставније је да је запишемо овако:

```
ADD x
```

при чему лабела x означава адресу 11111110. Овакви симболички еквиваленти машинских инструкција називају се *асемблерске инструкције*, а одговарајући симболички језик називамо *асемблерски језик*. Програми се сада уместо на машинском језику пишу на асемблерском језику, што олакшава поступак програмирања и чини програме читљивијим. Асемблерски програми се чувају у обичним текстуалним фајловима са одговарајућом екстензијом (типично `.s` или `.asm`). Превођење асемблерских програма на машински језик врши се помоћу посебног програма који се зове *асемблер*. Приликом превођења, лабелама се најпре придружују конкретне апсолутне адресе, а затим се инструкције преводе једна по једна, тако што се мнемоници замењују конкретним операционим кодовима, а лабеле придруженим адресама.[5] Машински програм се обично чува у бинарном облику и није читљив за човека.

У примеру нашег рачунара, један могући запис асемблерских инструкција је дат у табелама 5.4, 5.5 и 5.6. Непосредни операнди се записују као константе (нпр. `ADD 5`), директни меморијски операнди би били означени лабелом (нпр. `ADD x`), док би индиректни меморијски операнди били записани лабелом у угластим заградама (нпр. `ADD [x]`). У случају инструкција скока, навођење лабеле којом је означена инструкција тумачи се као непосредни операнд, док се навођење лабеле која означава податак тумачи као директно адресирање.

На пример, алгоритам за израчунавање највећег заједничког делиоца два броја који се налазе на адресама означеним лабелама x и y би на асемблерском језику нашег рачунара могао да изгледа овако:

```
start:
  LOAD x
  SUB y
  JE end
  JA x_above

x_below:
  NEG
  STORE y
  JMP start

x_above:
```

[5] У реалним асемблерским језицима, не мора увек постојати „1-1" придруживање између асемблерских и машинских инструкција. Наиме, постоје асемблерски језици у којима се понекад једна асемблерска инструкција може преводити у низ од неколико машинских инструкција. Овим се понекад програмирање на асемблерском језику додатно олакшава, јер се неке операције могу једноставније изразити. Ипак, тиме се процес превођења у извесној мери компликује.

```
  STORE x
  JMP start
end:
  HALT
```

Алгоритам у потпуности одговара алгоритму описаном у поглављу 5.1. Превођењем на машински језик, добили бисмо:

```
00000001 11111110
01000001 11111111
00011100 00010001
00111100 00001101
01110000
00010001 11111111
00001100 00000000
00010001 11111110
00001100 00000000
11111111
```

при чему су лабелама x, y, *start*, *end*, x_above и x_below редом додељене адресе 11111110, 11111111, 00000000, 00010001, 00001101 и 00001000, а програм се учитава почев од адресе 0. Почетне вредности података потребно је унети на адресе 11111110 и 11111111, а након завршетка алгоритма НЗД датих бројева се може прочитати из било које од ове две меморијске локације.

Читаоцу остављамо за вежбу да неке сложеније алгоритме имплементира на асемблерском језику нашег рачунара.

Део II

Архитектура и организација рачунара

Глава 6

Централни процесор

У претходној глави приказан је један модел једноставног Фон-Нојмановог рачунара и детаљно је објашњен његов начин рада. Са теоријског становишта, наш посао би ту могао бити завршен. Ипак, јасно је да приказаном моделу рачунара много тога недостаје да би почео да личи на савремене рачунаре. У наставку овог текста се управо бавимо тим питањима.

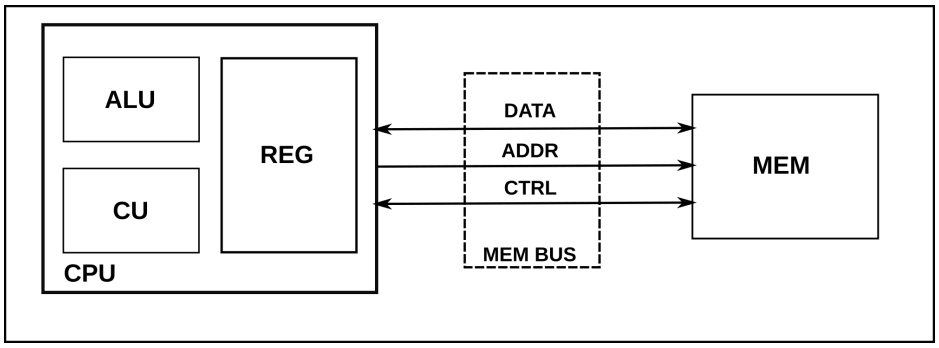
У моделу рачунара приказаном у претходној глави, сваки програм се, у крајњој линији, сводио на низ елементарних операција (попут операција трансфера података и ALU операција) које су се извршавале у једном циклусу часовника. Компоненте рачунара које директно учествују у извршавању ових елементарних операција обично се називају *путања података* (енгл. *datapath*). У случају нашег рачунара, путања података се састојала из меморије, регистара специјалне намене (програмски бројач, инструкцијски регистар, акумулатор и сл.), ALU јединице, као и жица којима се ове компоненте међусобно повезују. Путањом података управљала је контролна јединица, која је одређивала редослед елементарних операција које ће се на путањи података извршавати.

У нашем једноставном моделу рачунара, меморија која је садржала машински програм и податке (позната под називом *оперативна меморија*) била је део путање података, тј. била је директно повезана са ALU јединицом и регистрима специјалне намене, те је могла да са њима директно размењује податке. Овако нешто је могуће само за заиста мале меморије. У реалним рачунарима, оперативне меморије су обично знатно веће, што их чини значајно споријим од осталих компоненти на путањи података. Поред величине, на њихову спорост утиче и технологија израде – оперативне меморије се обично израђују као динамичке меморије како би се обезбедио што већи капацитет, али их то чини још споријим.

Из овог разлога се у реалним рачунарима путања података обично састоји из ALU јединице и регистара, док се меморија издваја у посебну компоненту рачунара. Са друге стране, контролна јединица мора бити близу путање података, како би њени контролни сигнали могли брзо да стижу до свог одредишта. Због тога се компоненте путање података и контролна јединица обично групишу у једну целину која се назива *централни процесор* (енгл. *central processing unit* (CPU)). Захваљујући развоју технологије и високом степену интеграције савремених чипова, све

компоненте централног процесора је могуће сместити на један чип, познат под називом *микропроцесор*. Рачунаре засноване на микропроцесорима називамо *микрорачунарима*.

Имајући све ово у виду, основни модел рачунара који ћемо разматрати у наставку овог текста је приказан на слици 6.1.



Слика 6.1: Модел рачунара са ускладиштеним програмом

Процесор се састоји из ALU јединице, скупа регистара и контролне јединице. Оперативна меморија је засебна компонента која се са процесором повезује *меморијском магистралом* (енгл. *memory bus*). Ово је посебна магистрала која се налази изван процесора и не треба је мешати са интерним магистралама које се користе за повезивање компоненти путање података унутар процесора. Састоји се из *магистрале података* (енгл. *data bus*) која се користи за пренос података (и инструкција) између процесора и меморије, *адресне магистрале* (енгл. *address bus*) која се користи за пренос адреса од процесора до меморије, као и контролне магистрале (енгл. *control bus*) која се састоји из скупа појединачних контролних сигнала којима меморија и процесор међусобно комуницирају како би синхронизовали свој рад. Типично, ово укључује сигнале којима процесор упућује меморији захтев за читањем или писањем, али и контролни сигнал којим меморија шаље процесору захтев за додатним чекањем (тзв. *wait* сигнал), уколико није у могућности да у предвиђеном времену обави тражену операцију. Ово је у пракси врло вероватно, с обзиром на знатно мању брзину меморије у односу на процесор. У том случају, процесор ће чекати већи број циклуса док се меморијска операција не заврши, пре него што може да настави свој рад. Отуда у овом моделу рачунара комуникација са меморијом представља главно „уско грло” које најдрастичније утиче на перформансе целог рачунара.

У раном периоду развоја рачунара, скуп регистара процесора се састојао искључиво из регистара специјалне намене, попут оних које смо упознали у претходној глави. Модерни процесори, поред регистара специјалне намене обично садрже и скуп регистара опште намене у које се могу пребацивати подаци из меморије и који се потом могу директно користити као операнди инструкција. Број регистара опште намене је обично релативно мали (нпр. 8, 16 или 32), што приступ таквим регистрима чини веома брзим. За разлику од оперативне меморије, регистри се

израђују као брза статичка меморија (помоћу флип-флопова). Отуда ови регистри представљају малу, али веома брзу меморију унутар процесора. Типично, ове регистре користимо да у њих сместимо податке са којима тренутно радимо. Циљ је да што више израчунавања извршимо унутар процесора, како би број меморијских трансфера био сведен на минимум. Имајући у виду да је у присуству регистара опште намене потребно омогућити програмеру да реферише на податке у тим регистрима у својим инструкцијама, поред непосредних и меморијских операнада са којима смо се упознали у претходној глави, модерни рачунари обично имају и трећу врсту операнада – *регистарске операнде*.

Централни процесор врши сва израчунавања из којих се наш програм састоји (јер садржи путању података), али и управља радом целог рачунара (с обзиром да садржи контролну јединицу). Програм по коме ради централни процесор, заједно са припадајућим подацима, налази се у оперативној меморији.

Постоје два основна аспекта централног процесора која обично разматрамо. Први аспект представља скуп свих оних својстава централног процесора која су од значаја за ефективно коришћење процесора и његову комуникацију са окружењем у које се уграђује. Ова својства се једним именом називају *архитектура централног процесора* и у њих спадају величина речи процесора, скуп регистара опште намене, скуп инструкција, формати инструкција, врсте операнада и начини њиховог адресирања, режими рада процесора, нивои привилегија извршавања инструкција, механизми управљања меморијом, величина адресног простора (виртуелног и стварног) и сл. Другим речима, у архитектуру рачунара спада све оно што је потребно програмеру да зна да би могао да пише програме на машинском (и асемблерском) језику тог процесора. Како централно место у архитектури процесора заузима скуп његових инструкција, често се користи и термин *архитектура скупа инструкција* (енгл. *instruction set architecture* (ISA)).

Други аспект представља унутрашња *организација процесора* која представља начин имплементације самог процесора (из којих се компоненти састоји, како су оне међусобно повезане и на који начин је реализована контролна јединица, као интерпретатор машинског језика). Једна те иста архитектура може бити реализована на више начина, па самим тим можемо имати више различито организованих процесора који имплементирају исту архитектуру, те су, са становишта програмера, потпуно еквивалентни.[1]

Архитектура процесора функционално описује процесор, тј. одговара нам на питање „шта процесор ради?". Са друге стране, организација процесора разматра структуру процесора и одговара нам на питање „како процесор ради?". Архитектура је старија од организације, јер прво морамо дефинисати шта желимо да процесор може да ради, а затим можемо прећи на разматрање како то да имплементирамо.

Аспекти архитектуре и организације постоје и код других компоненти рачунарског система, попут оперативне меморије, улазно-излазних контролера, контролера прекида и сл. о којима ћемо причати у наредним главама. Архитектура се увек односи на карактеристике компоненте када је

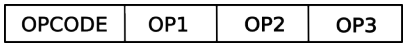
[1] Типичан пример су процесори компанија Intel и AMD који већ деценијама уназад производе процесоре који имплементирају исту архитектуру (познату као *x86* и *x86-64*), иако су интерно понекад доста другачије организовани.

посматрамо као „црну кутију" коју треба да користимо у датом окружењу. Организација се односи на начин имплементације те „црне кутије" изнутра. Најзад, често се говори и о архитектури целог рачунарског система, као скупу његових компоненти и начину на који су све његове компоненте повезане и међусобно комуницирају.

У наставку ове главе, детаљније се бавимо аспектима архитектуре и организације централног процесора.

6.1 Архитектура централног процесора

Као што је познато, машински програм се састоји из низа машинских инструкција. Свака машинска инструкција је низ битова који се структурно може поделити на *операциони код* који кодира операцију коју инструкција треба да изврши и *операнде* који кодирају на које податке се примењује та операција. Инструкција може имати нула или више операнада, али ретко више од три. Поред ових експлицитно наведених операнада, машинска инструкција може имати и *имплицитне* операнде, тј. операнде који се не наводе као део инструкције, већ се подразумевају (попут акумулатора у рачунару описаном у претходној глави).



Слика 6.2: Формат инструкције

Као што је раније речено, према локацији где се налазе, операнди могу бити регистарски, непосредни и меморијски. *Формат инструкције* представља начин кодирања горњих компоненти у бинарном облику у машинској инструкцији. Формат инструкције имплицитно одређује и дужину инструкције. Не морају све инструкције неке архитектуре бити истог формата, па самим тим, различите инструкције могу бити различитих дужина.

6.1.1 Врсте архитектура према броју операнада

Архитектуре се могу класификовати према броју експлицитно наведених операнада које инструкције могу имати. Рачунар описан у претходној глави је био пример *једноадресног рачунара*. Карактеристика ових рачунара је да немају регистре опште намене, већ само један специјални регистар који се назива *акумулатор* који може садржати податак са којим се тренутно ради. Све аритметичко-логичке инструкције имају овај регистар као имплицитни операнд, док се евентуални други операнд (ако га инструкција захтева) наводи као експлицитни операнд. Резултат се увек смешта поново у акумулатор. Отуда, највише један операнд се наводи експлицитно у инструкцији.

Као пример, размотримо извршавање наредбе доделе:

```
x = a*b + c*d
```

у програмском језику вишег нивоа. У случају једноадресног рачунара, асемблерски код који би био еквивалентан овој наредби би могао да изгледа овако:

```
LOAD a
MUL b
STORE t
LOAD c
MUL d
ADD t
STORE x
```

Дакле, најпре се `LOAD` инструкцијом вредност променљиве a учитава у акумулатор, а затим се акумулатор множи са променљивом b и резултат остаје у акумулатору. Како би се акумулатор ослободио за следеће израчунавање, његова тренутна вредност се привремено копира у променљиву t. Сада се на сличан начин у акумулатору израчунава производ c и d. На крају се на тренутну вредност акумулатора додаје претходно сачувана вредност променљиве t и добијена вредност израза се из акумулатора пребацују у одредишну променљиву x (инструкцијом `STORE`).

Како бисмо могли да разматрамо ефикасност једноадресних архитектура (и да их упоредимо са другим врстама архитектура), морамо се најпре одлучити за метрику коју ћемо у ту сврху користити. Како смо већ констатовали да приликом извршавања инструкција највише времена одлази на меморијске трансфере, делује разумно да број меморијских трансфера користимо као меру сложености горњег кôда. Притом, претпоставићемо да се учитавање саме инструкције може обавити у једном трансферу преко магистрале. Отуда је за извршавање инструкције `LOAD a` потребно два меморијска трансфера, први за учитавање инструкције, а други за учитавање податка из меморије у акумулатор. Слично, за инструкцију `STORE x` су потребна два меморијска трансфера, један за учитавање инструкције, а други за смештање вредности акумулатора у меморију. За аритметичке инструкције (`ADD` и `MUL`) потребна су такође два меморијска трансфера – први за учитавање инструкције, а други за учитавање операнда, након чега се унутар процесора (у ALU јединици) врши одговарајућа операција и резултат остаје у акумулатору. Дакле, за цео кôд потребно је $7 \cdot 2 = 14$ меморијских трансфера.

У случају једноадресних архитектура, један операнд је увек имплицитни, а он уједно служи и као одредиште за чување резултата. Програмер може да бира само шта ће бити други операнд, ако исти постоји. Додатна флексибилност се добија ако дозволимо да инструкција има два експлицитно задата операнда. Овакве рачунаре називамо *двоадресни рачунари*. У том случају се оба операнда бинарних операција могу експлицитно наводити (нпр. оба сабирка у случају сабирања). Ипак, због немогућности да се и одредиште експлицитно наведе, један од та два операнда ће уједно бити и одредишни операнд. Одређености ради, ми ћемо овде претпоставити да је први операнд уједно и одредишни операнд. У том случају би асемблерски кôд за израчунавање горњег израза могао изгледати овако:

```
MOV t, a
```

```
MUL t, b
MOV s, c
MUL s, d
MOV x, t
ADD x, s
```

Дакле, најпре се у помоћну променљиву t копира вредност a (инструкцијом `MOV`), а затим се множе променљиве t и b и добијени производ се уписује у t (дакле, код инструкције `MUL` променљива t се користи и као први чинилац и као одредиште резултата). На сличан начин се у променљивој s израчунава производ c и d. На крају се t копира у x, а затим се, инструкцијом `ADD` врши сабирање x и s, а резултат се поново смешта у први операнд x.

Како бисмо добијени кôд упоредили са претходним кôдом у случају једноадресних рачунара, претпоставимо најпре да су све наведене променљиве сачуване у меморији, тј. да инструкције имају искључиво меморијске операнде. Такође, претпоставићемо као и раније да се дохватање саме инструкције може обавити у једном меморијском трансферу преко магистрале. У том случају је за `MOV` инструкцију потребно три меморијска трансфера – први за дохватање инструкције, други за учитавање изворишног операнда (други операнд) и трећи за копирање учитане вредности у одредишни операнд (први операнд, који се такође налази у меморији). За аритметичке инструкције (`MUL` и `ADD`) потребна су четири меморијска трансфера. Првим трансфером се дохвата инструкција, другим и трећим се учитавају вредности, респективно, првог и другог операнда, а четвртим трансфером се израчуната вредност уписује у одредишни (први) операнд. Укупан број меморијских трансфера је, отуда, једнак $3{\cdot}3{+}3{\cdot}4 = 21$.

На први поглед, ово делује као доста лошије решење у односу на једноадресне архитектуре, где је број меморијских трансфера за израчунавање истог израза био једнак 14. Ово је зато што је код једноадресних рачунара један операнд увек био у акумулатору (који је регистар у процесору), а акумулатор је имплицитно био и одредиште. У случају двоадресног рачунара, сви операнди у горњем примеру били су меморијски, што је повећавало број трансфера. Ипак, сетимо се да модерни рачунари поседују регистре опште намене који се могу користити за чување међурезултата. Уколико у горњем кôду претпоставимо да су помоћне променљиве t и s у регистрима опште намене, а не у меморији, тада се број меморијских трансфера смањује на 12.

У пракси, регистри опште намене се не морају користити само за међурезултате, већ се и променљиве могу чувати у регистрима. Заправо, модерни преводиоци обично приликом превођења изворног програма са језика вишег нивоа на асемблер покушавају да што већи број променљивих све време чувају у регистрима, како би им програм могао брже приступати. Ако би, у нашем примеру, све променљиве биле у регистрима уместо у меморији, број трансфера би био свега 6, јер бисмо само морали да учитавамо инструкције из меморије, све остало би било у процесору.

Ако бисмо дозволили да инструкције имају три експлицитно наведена операнда, тада бисмо имали могућност да у случају бинарних операција оба изворишна операнда, као и одредишни операнд посебно задајемо. На пример, у случају сабирања, један операнд би одређивао одредиште у које треба уписати збир, а преостала два операнда би представљала

сабирке. Овакве рачунаре називамо *троадресни рачунари*. Одређености ради, претпоставићемо да је први операнд одредишни операнд. Сада би исти израз од малопре могао бити израчунат овако:

```
MUL t, a, b
MUL s, c, d
ADD x, t, s
```

Дакле, првом инструкцијом множимо a и b и производ смештамо у t, другом инструкцијом множимо c и d и производ смештамо у s, док трећа инструкција сабира t и s и збир смешта у x.

Ако претпоставимо да су све променљиве у меморији, тада је за сваку инструкцију потребно по четири меморијска трансфера (један за учитавање инструкције, два за учитавање изворишних операнада и један за смештање резултата на одредиште). Ово је укупно $3 \cdot 4 = 12$ трансфера. Међутим, ако опет претпоставимо да су помоћне променљиве t и s у регистрима, тада се број меморијских трансфера смањује на 8. У још екстремнијем случају, ако претпоставимо да су све променљиве у регистрима, тада ће број трансфера бити само 3 (за дохватање инструкција).

Број операнада које инструкције имају је ретко већи од три. Ипак, постоје и случајеви архитектура код којих већина инструкција уопште немају експлицитно наведене регистре. Такве архитектуре називамо *нулоадресне архитектуре*. Овакве архитектуре су обично реализоване као *стек машине*. Ово значи да овакви процесори у себи садрже одређени број регистара који су логички организовани као *стек*[2]. Постоје посебне инструкције за додавање новог податка на врх стека и за скидање податка са стека и пребацивање његове вредности у меморијску променљиву (означимо ове инструкције са `PUSH` и `POP`, респективно). Ове инструкције имају по један операнд који означава адресу меморијске променљиве која се користи. Са друге стране, аритметичко-логичке инструкције немају операнде, већ имплицитно користе вредности са врха стека. На пример, инструкција `ADD` ће скинути са врха стека две вредности, сабрати их и збир поново поставити на стек (ефективно, две вредности са врха стека смо заменили једном – њиховим збиром). Слично раде и друге аритметичко-логичке инструкције. На пример, израчунавање истог израза од малопре бисмо на једном нулоадресном рачунару могли да обавимо на следећи начин:

```
PUSH a
PUSH b
MUL
PUSH c
PUSH d
MUL
ADD
POP x
```

Дакле, сада најпре постављамо на врх стека вредност променљиве a, затим и вредност променљиве b, након чега их позивом `MUL` инструкције скидамо

[2]Стек је LIFO структура података (енгл. *last in – first out*) код које се онај податак који је последњи постављен на стек увек први са њега скида.

са стека и замењујемо њиховим производом. Затим на сличан начин на врх стека додајемо и променљиве c и d (не заборавимо да је вредност $a \cdot b$ остала на дну стека), а затим их скидамо и замењујемо њиховим производом. Сада на стеку имамо $c \cdot d$ на врху, а испод њега $a \cdot b$. Инструкцијом `ADD` се ове две вредности скидају са стека и замењују њиховим збиром. На крају, тај збир скидамо са стека (`POP` инструкцијом) и копирамо у променљиву x.

Број меморијских трансфера је у случају горњег кôда једнак 13 (инструкције `PUSH` и `POP` захтевају по два трансфера, а инструкције `ADD` и `MUL` само по један, за дохватање операционог кôда).

Оно што примећујемо је да употреба инструкција са већим бројем операнада има тенденцију да смањује број меморијских трансфера. Ово је пре свега због тога што се број инструкција смањује, тј. програм постаје краћи. Са друге стране, број меморијских трансфера по инструкцији може бити већи ако бисмо користили меморијске операнде. Ипак, овај негативни ефекат се успешно анулира употребом регистарских операнада који су доступни на модерним архитектурама. Притом, морамо имати у виду да смо горње поређење вршили под веома битном претпоставком да се дохватање инструкције може обавити у једном трансферу. Овако нешто је сасвим реално за нулоадресне и једноадресне рачунаре, јер су њихове инструкције сасвим кратке. Ипак, за двоадресне, а нарочито за троадресне рачунаре, овако нешто не мора бити реална претпоставка, поготово ако користимо меморијске операнде. Наиме, кодирање меморијских операнада може захтевати задавање апсолутних адреса (нпр. у случају директног адресирања), а апсолутне адресе могу бити прилично дуге (нпр. на савременим рачунарима оне могу бити дужине 32 или 64 бита). Отуда, коришћење меморијских операнада продужава формат инструкције, што може довести до тога да је за сâмо дохватање инструкције потребно више узастопних трансфера на магистрали. Са друге стране, регистарски операнди се кодирају веома компактно. На пример, ако архитектура има 16 регистара опште намене, довољно је 4 бита за кодирање редног броја регистра који се користи као операнд. Одавде произилази и начин на који се проблем предугачких формата инструкција обично решава у пракси – ограничавањем броја дозвољених меморијских операнада у инструкцији. На пример, архитектура *x86* (и *x86-64* као њен наследник) има ограничење да највише један операнд инструкције може бити меморијски (ово важи за све инструкције ове архитектуре). Још екстремнији пример су тзв. *load/store* архитектуре: код њих готово све инструкције не могу имати ни један меморијски операнд, већ искључиво раде са регистрима. Ово посебно важи за све аритметичко-логичке инструкције. Једине инструкције које користе меморијске операнде су инструкције за учитавање (*load*) података из меморије у регистре и инструкције за чување (*store*) вредности регистара у меморији. Ове инструкције имају по један меморијски операнд. Већина модерних троадресних архитектура се заснивају на овом принципу. На овај начин обезбеђујемо да инструкције не буду предугачке и да се у већини случајева могу у једном трансферу дохватити из меморије. Ово наше закључке из претходне анализе чини реалистичним у пракси.

Једноадресни рачунари су били доминантни у раном периоду развоја рачунара, када је императив био на једноставном дизајну процесора, тако да су формати инструкција морали бити једноставни, а регистри опште намене нису постојали. У модерно време, доминирају двоадресни

и троадресни рачунари, који, поред раније описаног смањења броја меморијских трансфера, нуде и знатно више флексибилности у смислу избора операнада који ће се користити у инструкцијама. Ово омогућава бољу оптимизацију кôда који генеришу компилатори. Пример двоадресне архитектуре је поменута *x86-64* архитектура, коју имплементирају модерни процесори компанија *Intel* и *AMD*. Са друге стране, већина новијих архитектура тзв. *RISC* типа (попут архитектура *ARM32*, *ARM64*, *SPARC*, *RISC-V*, *MIPS*) су реализоване као троадресне *load/store* архитектуре. Примери нулоадресних архитектура су неке старије архитектуре које се више не одржавају, попут архитектуре *HP3000*. Такође, добар пример је и *математички процесор* који постоји на рачунарима заснованим на *x86* архитектури. Овај копроцесор се традиционално користио[3] за израчунавања у покретном зарезу (најпре као физички одвојена компонента рачунара, а касније као интегрални део централног процесора). Архитектура овог копроцесора (позната и као *x87*) функционише по принципу стек-машине.

На крају, треба нагласити да горе описана подела није стриктна, у смислу да архитектура може садржати инструкције са различитим бројем експлицитно задатих операнада. На примеру нулоадресних рачунара, већ смо видели да нису све њихове инструкције без операнада – постојале су инструкције `PUSH` и `POP` које су имале по један меморијски операнд. Слично, на архитектури *x86-64*, постоји инструкција `mul` која има само један експлицитни операнд који представља други чинилац (тзв. множилац), док је први чинилац имплицитно садржан у регистру `rax`, који служи и као одредишни операнд. Примећујемо да описана инструкција функционише по принципу једноадресних рачунара, где регистар `rax` игра улогу акумулатора. Дакле, подела архитектура на двоадресне, троадресне, и сл. се врши према приступу који се доминантно користи у архитектури, уз могућност да постоје инструкције које представљају изузетак.

6.1.2 Начини адресирања операнада

Као што је раније речено, операнди инструкција могу бити имплицитни и експлицитни. Локација имплицитних операнада се подразумева (обично је то неки предефинисани регистар процесора), тако да се не наводи у инструкцији. Са друге стране, експлицитни операнди се задају као део инструкције, тј. инструкција мора садржати бинарни кôд који одређује вредност или локацију операнда. Информације о броју и типу операнада, као и на који начин се они кодирају је, по правилу, садржана у операционом кôду инструкције, тако да процесор након дохватања операционог кôда зна како да тумачи остатак бинарног записа инструкције.

У случају непосредног операнда, инструкција садржи саму вредност (константу) која се користи као операнд. У асемблеру ову константу обично записујемо декадно, мада је обично могуће задавати је и бинарном, окталном или хексадекадном облику. На пример, на архитектури *x86-64* то може изгледати овако:

```
add rax, 5  # dodajemo 5 (dekadno) na vrednost registra rax
```

[3] И данас се користи, али у мањој мери, с обзиром на постојање *SSE* скупа инструкција које такође врше израчунавања у покретном зарезу.

```
mov rax, 0b00000011  # kopiramo vrednost 3 (zapisanu binarno)
                       u rax
sub rax, 0777  # oduzimamo oktalni broj 777 od registra rax
cmp rax, 0xaf  # poredimo rax sa heksadekadnom konstantom af
```

Константа најчешће може имати и негативан предзнак, а бинарно се може кодирати у потпуном комплементу. Притом, колико ће битова бити широко поље за кодирање константе зависи од архитектуре. На пример, архитектура *x86-64* омогућава кодирање 8-битних, 16-битних или 32-битних целобројних константи у потпуном комплементу. Коришћење дугачких формата целобројних константи продужава и саму инструкцију, што може утицати на ефикасност њеног дохватања. С тим у вези, постоје архитектуре које инсистирају на униформној дужини инструкција, што омогућава њихово ефикасније дохватање и декодирање. Пример такве архитектуре је *ARM32* архитектура. Све инструкције ове архитектуре су дугачке 32 бита. Последица овог приступа је да не постоји могућност 32-битног кодирања непосредних операнада, с обзиром да је потребно одвојити одређени број битова и за операциони кôд и остале операнде. Архитектура *ARM32* за непосредне операнде одваја 12 битова, што за последицу има да не постоји могућност кодирања константи већих вредности.[4] Ово ограничење се обично разрешава тако што се бинарни код већих константи декомпонује на мање делове (нпр. на четири 8-битна поља), па се онда помоћу битовских инструкција ти делови слажу тако да се добије полазна вредност. Ово се обично аутоматски решава на асемблерском нивоу (тј. једна асемблерска инструкција која користи непосредни операнд велике вредности се преводи у неколико машинских инструкција којима се постиже жељени ефекат), тако да програмер не мора да се бави тим техничким детаљима.

У случају регистарских операнада, потребно је кодирати редни број регистра који се користи као операнд. Ако, на пример, архитектура има 16 регистара опште намене, тада је потребно одвојити 4-битно поље за кодирање редног броја регистра. С обзиром да је број регистара обично релативно мали, за њихово кодирање није потребно много битова. У асемблеру се регистарски операнди задају својим симболичким именима која зависе од архитектуре. Приликом превођења са асемблерског на машински језик, та симболичка имена се замењују одговарајућим бинарним кôдом.

Када су у питању меморијски операнди, потребно је на неки начин задати адресу у меморији на којој се операнд налази. Начини задавања ове адресе се обично називају *начини адресирања* или *адресни модови* (енгл. *addressing modes*). Неке најчешће коришћене начине адресирања описујемо у наставку.

Директно адресирање. У случају директног адресирања, инструкција садржи апсолутну адресу податка у меморији који се користи као операнд инструкције. Као што је дискутовано у поглављу 5.2, директно адресирање се користи за податке чија је адреса позната у фази писања програма (или у фази превођења, у случају програма писаних на вишим програмским

[4] Заправо, могуће је кодирати све 8-битне константе, уз додатну могућност кодирања и 32-битних константи које се могу добити проширењем 8-битне константе на 32-бита, а затим ротацијом за паран број позиција у десно.

језицима). У језицима *C* и *C++*, то су глобалне променљиве и статичке локалне променљиве, као и статички чланови класа. У асемблеру, директно адресирање се обично постиже помоћу симболичких ознака које називамо *лабеле*. Лабеле служе као идентификатори којима се придружују статички алоцирани подаци. На пример, на асемблерском језику архитектуре *x86-64*, то изгледа овако:

```
.data
  x: .int 15
  y: .int 12

.text
  mov eax, x
  add eax, y
```

Дакле, у `.data` секцији која садржи статички алоциране податке ми декларишемо два целобројна податка (и задајемо им почетне вредности) и том приликом им придружујемо лабеле `x` и `y`. Касније, у `.text` секцији која садржи кôд можемо референсирати на ове податке употребом њихових лабела. Приликом превођења на машински језик, асемблер ће подацима у `.data` секцији придружити апсолутне адресе, а затим ће у свим инструкцијама које користе одговарајуће лабеле заменити те лабеле додељеним апсолутним адресама (тј. користиће директно адресирање).

Директно адресирање се може користити и за задавање адресе скока у инструкцијама скока. У том случају, као део инструкције наводи се апсолутна адреса на коју треба скочити. Ипак, на већини савремених архитектура, директно адресирање се углавном користи за податке.

Главни проблем код директног адресирања је дужина бинарног записа апсолутне адресе – на модерним рачунарима, ове адресе су обично 32-битне или 64-битне. Ово значајно продужава формат саме инструкције. У случају рачунара са фиксним форматом инструкција, битско поље које се користи у инструкцији за запис апсолутне адресе меморијског операнда најчешће не може бити довољно велико тако да омогућава запис било које апсолутне адресе. На пример, на *ARM32* архитектури су све инструкције 32-битне, тако да није могуће кодирати апсолутну 32-битну адресу меморијског операнда као део инструкције. Отуда ова архитектура не подржава директно адресирање на уобичајен начин. Постизање ефекта директног адресирања може се симулирати тако што се у регистру формира 32-битна апсолутна адреса (слично као код формирања 32-битних константи), а затим се користи индиректно регистарско адресирање које ће бити описано касније у овом одељку. На срећу, асемблерски језик подржава директно адресирање, чиме се описана симулација препушта асемблерском преводиоцу.

Релативно адресирање. Релативно адресирање се типично користи за задавање адресе скока код инструкција условног и безусловног скока. Код релативног адресирања, као део инструкције наводи се *померај* (енгл. *offset*) у односу на вредност PC регистра (због тога се често назива и *PC-релативно*). Вредност помераја (која може бити позитивна или негативна) се додаје на вредност PC регистра и тако се добија апсолутна адреса

инструкције на коју треба скочити. Како вредност PC регистра представља апсолутну адресу текуће инструкције програма, на овај начин можемо једноставно референцирати на инструкције које се налазе „у околини" текуће инструкције. Ово је нарочито корисно када је потребно скочити на неку инструкцију у близини текуће (тзв. *кратки скокови*, енгл. *short jumps*). Код кратких скокова, адреса на коју скачемо се налази у некој малој околини текуће адресе, попут ±128 или ±256 бајтова. Заправо, анализом типичних програма утврђено је да преко 90% свих скокова спада у ову категорију. Отуда је за кодирање релативног помераја довољно 8 до 10 битова. Овим се постиже знатно краће кодирање инструкција скока, у поређењу са навођењем апсолутне адресе скока, што бисмо имали код директног адресирања.

Релативно адресирање се може користити и за адресирање удаљених инструкција (тзв. *дуги скокови*, енгл. *long jumps*) или за адресирање података на удаљеним адресама. У том случају, померај не може бити записан у малом броју битова, чиме се та главна корист употребе релативног адресирања губи. Ипак, постоји и друга корист од употребе релативног адресирања, а то је постизање да добијени кôд буде *независан од позиције* у адресном простору процесора (енгл. *position independant code* (PIC)). Ово значи да се програм може учитати почев од било које адресе у меморији и да ради исправно без икаквих модификација након учитавања, због тога што све адресе које се користе представљају релативне адресе у односу на текућу инструкцију.

У асемблерском језику, релативно адресирање се обично задаје на исти начин као и директно – навођењем лабеле која означава податак који користимо или инструкцију на коју желимо да скочимо. Асемблерски преводилац одређује релативне помераје аутоматски приликом превођења на машински језик. Ово често омогућава оптимизације приликом превођења, с обзиром да асемблерски преводилац може проценити колико му је битова потребно за запис помераја у односу на адресу текуће инструкције. Наиме, многе архитектуре дозвољавају више различитих формата исте инструкције, где постоји могућност да се релативни померај задаје са различитим бројем битова. На пример, инструкције скока на архитектури *x86-64* могу користити 8-битни или 32-битни померај, а асемблерски преводилац може одлучити да користи 8-битни, у случају кратких скокова, чиме се смањује дужина инструкције.[5] Са друге стране, архитектура *ARM32* (која такође користи релативно адресирање за инструкције скока) користи фиксирано 24-битно поље за померај. Овај 24-битни померај се означено проширује на 30 бита, а затим се додаје још 2 бита на десни крај (чиме се овај померај ефективно множи са четири). Ово је зато што су на архитектури *ARM32* све адресе инструкција увек поравнате тако да буду дељиве са четири. Овај померај се додаје на вредност PC регистра, чиме се добија апсолутна адреса на коју треба скочити.[6]

[5] Непостојање могућности 64-битног помераја ефективно спречава јако далеке условне скокове у 64-битном адресном простору. Овај проблем се може разрешити употребом инструкције безусловног скока (на коју можемо најпре скочити кратким условним скоком) која поред релативног допушта и регистарско индиректно адресирање, чиме јој је доступан цео адресни простор.

[6] На овај начин је могуће скакати на адресе у распону $\pm 32MB$ у односу на текућу инструкцију. За дуже скокове, *ARM32* допушта директни упис жељене вредности у PC

Индиректно адресирање. Код индиректног адресирања, адреса операнда се не наводи као део инструкције, већ се адреса налази на неком другом месту. Инструкција мора задати локацију места на ком се налази адреса, како би процесор могао да је дохвати и потом искористи за дохватање операнда. Традиционално индиректно адресирање подразумева да се адреса операнда налази у меморији, а да се апсолутна адреса локације у меморији на којој се налази адреса задаје као део инструкције (пример таквог начина адресирања је коришћен у опису рачунара у поглављу 5.2). Овај начин адресирања је био доминантан код раније рачунара који нису имали регистре опште намене. Главна мана овог начина адресирања је то што захтева додатни приступ меморији, за дохватање адресе. Са друге стране, модерне архитектуре подржавају тзв. *регистарско индиректно адресирање*, код кога се адреса операнда налази у неком од регистара опште намене. Отуда је за задавање операнда довољно као део инструкције навести код регистра који садржи адресу операнда. На пример, на архитектури *x86-64*, можемо на асемблерском језику имати нешто овако:

```
mov rax, [rsi]
```

Овом инструкцијом се из меморије, са адресе коју садржи регистар `rsi`, копира податак и уписује у регистар `rax`. Приметимо да је употреба заграда `[]` неопходна, иначе би асемблерски преводилац сматрао да се ради о регистарском операнду. Слично, на архитектури *ARM32*, можемо написати овако:

```
ldr r0, [r8]
```

Овим се меморијски операнд са адресе која је садржана у регистру `r8` копира у регистар `r0`.

Предност регистарског индиректног адресирања је то што се адреса налази у регистру, па није потребан додатни приступ меморији. Код модерних архитектура које поседују регистре опште намене, ово је готово искључива форма употребе индиректног адресирања.

Поједине архитектуре, попут *x86-64*, допуштају употребу регистарског индиректног адресирања и код инструкција скока. Прецизније, ово је допуштено само у случају инструкције безусловног скока `jmp`, док инструкције условног скока користе искључиво релативно адресирање. Са друге стране, архитектура *ARM32* за инструкције скока (и условног и безусловног) користи искључиво релативно адресирање, док се постизање ефекта регистарског индиректног адресирања може постићи простим копирањем адресе скока из датог регистра у PC.

Индиректно адресирање се користи у ситуацијама када адреса податка није позната у фази писања/превођења програма. Ово може бити случај са приступом локалним променљивама и параметрима потпрограма (попут функција и метода у језицима *C* и *C++*), динамички алоцираним подацима, елементима низова у петљама и сл. Такође, може се користити за позиве потпрограма који нису фиксирани у фази писања програма (нпр. када у зависности од захтева корисника у фази извршења програм може позвати један од више понуђених потпрограма). На пример, ово је случај

регистар.

са показивачима на функције у језику *C*, или са динамички повезаним (тзв. виртуелним) методама у језику *C++*.

База + померај. Даљим уопштавањем регистарског индиректног адресирања долазимо до адресирања *база + померај*, код кога се апсолутна адреса операнда добија сабирањем вредности тзв. *базног регистра* и означеног *помераја*. Као део инструкције сада наводимо кôд базног регистра, као и саму вредност помераја (у потпуном комплементу или некако другачије). Овај начин адресирања омогућава да адресирамо податке који се налазе у околини податка чија се адреса налази у регистру. Типична ситуација где је то потребно је приступ подацима на *системском стеку*, који ће детаљније бити описан у једном од следећих одељака. На пример, на архитектури *x86-64*, можемо навести нешто овако:

```
mov rax, [rsp + 8]
```

Регистар `rsp` се на овој архитектури користи као *показивач стека* (енгл. *stack pointer*), тј. садржи адресу у меморији на којој се налази податак који је на врху стека. Горња инструкција сада приступа податку који се налази 8 бајтова испод врха стека (јер стек на овој архитектури „расте" ка нижим адресама). Сличан ефекат би имала следећа инструкција на архитектури *ARM32*:

```
ldr r0, [sp,#8]
```

при чему је овде `sp` регистар који игра улогу показивача стека на *ARM32* архитектури. С обзиром да се на стеку често могу налазити параметри и локалне променљиве потпрограма, овакав начин адресирања се типично користи за приступ таквим подацима.

Померај се у машинској инструкцији може кодирати различитим бројем битова. На пример, на архитектури *x86-64* постоји могућност коришћења 8, 16 или 32 бита за кодирање помераја. Коришћење дугачких помераја продужава саму инструкцију, што успорава њено дохватање. Као и раније, асемблерски преводилац може сам да одреди минималну потребну ширину на основу вредности помераја који је задат у асемблерској инструкцији. На *ARM32* архитектури, померај је увек фиксиране ширине у машинској инструкцији – заузима 12 битова.

Индексно адресирање. Индексно адресирање се типично користи за приступ елементима низа. Идеја је да на располагању имамо адресу почетка низа у меморији, као и индекс елемента ком желимо да приступимо. С обзиром да се елементи низа у меморији налазе један за другим и да су сви исте величине, адресу i-тог елемента низа можемо добити тако што на адресу почетка низа додамо вредност i помножену са величином елемента низа израженом у бајтовима. Индекс i се налази у *индексном регистру*. У зависности од архитектуре, ово може бити фиксирани регистар који служи искључиво за ту сврху или произвољни регистар опште намене чији се кôд задаје као део инструкције. Са друге стране, адреса почетка низа се може задати или као апсолутна адреса (тзв. *апсолутно индексно адресирање* или помоћу *базног регистра* (тзв. *база + индекс* адресирање). У модерним

архитектурама, овај други приступ је много чешћи, јер омогућава краће кодирање инструкције (јер се уместо кодирања дугачке апсолутне адресе користи знатно краћи кôд базног регистра). На пример, на архитектури *x86-64* можемо имати следећу инструкцију:

```
mov rax, [rsi + rcx]
```

Овде регистар `rsi` има улогу базног регистра, док регистар `rcx` игра улогу индексног регистра. Сабирањем ова два регистра добија се апсолутна адреса податка чија се вредност користи као операнд. Сличан ефекат би на *ARM32* архитектури имала следећа инструкција:

```
ldr r0, [r8, r3]
```

Овакав једноставан начин адресирања омогућава адресирање елемената низа само у случају да су елементи низа величине једног бајта. Да бисмо могли да адресирамо и елементе низова већих величина, потребно је да омогућимо да се вредност индексног регистра најпре помножи величином елемента низа пре него што се дода на базни регистар. Оваква варијанта индексног адресирања се назива *скалирано индексно адресирање* (или *база + скалирање * индекс*). На пример, на архитектури *x86-64* можемо записати нешто овако:

```
mov rax, [rsi + 4*rcx]
```

Овде се користи *фактор скалирања* 4, што омогућава приступ низовима чији су елементи величине 4 бајта (попут типа `int` у језицима *C* и *C++*). Дозвољени фактори скалирања су 2, 4 или 8, како би се множење тим фактором могло ефикасно обавити померањем у лево. Сличан ефекат је могуће остварити и на *ARM32* архитектури:

```
ldr r0, [r8, r3, rsl #2]
```

Ознака `rsl #2` означава да се вредност индексног регистра `r3` најпре помера у лево за две позиције (што је еквивалентно множењу са четири) пре него што се дода на вредност базног регистра `r8`.

Скалирано индексно адресирање се у машинском кôду инструкције задаје на веома компактан начин, јер је потребно само кодирати два регистра опште намене, као и фактор скалирања (који се задаје бројем позиција за који се врши померање индексног регистра у лево). Све укупно, то је обично од 8 до 12 битова.

База + скалирање * индекс + померај. Комбинацијом скалираног индексног адресирања и адресирања „база + померај" добија се адресирање које уопштава оба ова адресирања. Код овог адресирања кодирамо два регистра, базни и индексни, фактор скалирања као и означену константу која представља померај. Апсолутна адреса операнда се добија тако што се на вредност базног регистра дода индексни регистар помножен фактором скалирања, а затим се на све то дода вредност помераја. Овакав начин адресирања представља најопштији начин адресирања на *x86-64* архитектури. На пример:

```
mov rax, [rsi + 4*rcx - 8]
```

Притом, сваки од елемената горњег израза се може изоставити, чиме добијамо разне претходно описане облике адресирања (на пример, ако изоставимо померај, имамо скалирано индексно адресирање; ако изоставимо и фактор скалирања имамо просто индексно адресирање, ако додатно изоставимо и индексни регистар, добијамо регистарско индиректно адресирање; ако задржимо само базни регистар и померај, добијамо „база + померај" адресирање).

Ажурирање базног регистра. Адресирања попут „база + померај" могу имати могућност ажурирања базног регистра на израчунату адресу. Овако нешто је подржано на *ARM32* архитектури:

```
ldr r0, [r8, #8]!
```

Додавање узвичника на крају описа меморијског операнда означава да се израчуната адреса (`r8 + 8`) уписује у регистар `r8`. Ово семантички одговара синтакси `*++p` у језицима *C* и *C++* и згодно је када желимо да користимо „показивачку синтаксу" за итерацију кроз низ. Слично, могуће је написати и нешто овако:

```
ldr r0, [r8]!, #8
```

Ова нотација представља тзв. постфиксно ажурирање базног регистра. То значи да се у инструкцији као адреса користи стара вредност базног регистра (као у регистарском индиректном адресирању), али се након завршетка инструкције базни регистар увећава за 8. Ово је еквивалентно изразу `*p++` у језицима *C* и *C++*. Слични ефекти су могући и у случају индексног и скалираног индексног адресирања на *ARM32* архитектури.

6.1.3 Врсте машинских инструкција

Према намени, машинске инструкције се могу поделити у неколико врста које разматрамо у наставку.

Инструкције трансфера. Ове инструкције служе за копирање података са једне локације на другу. Иако се често користе асемблерски мнемоници попут `mov` или `move`, ове инструкције не премештају податак, већ га копирају на нову локацију (тј. податак остаје и на својој оригиналној локацији). Оригинална локација податка, као и одредиште, могу бити регистар процесора или локација у меморији. Ипак, због ограничења броја меморијских операнада у једној инструкцији о ком смо говорили раније, на већини архитектура није могуће да и изворište и одредиште буду у меморији, тј. да једном инструкцијом вршимо трансфер податка из једне меморијске локације у другу (за то су нам неопходне две инструкције). Оно што је могуће је да вршимо трансфер између регистра процесора и меморије (у било ком смеру), или трансфер између два регистра процесора. Поједине архитектуре користе исту инструкцију за обе сврхе.[7] Нпр. на

[7] Заправо, исти се мнемоник користи у асемблеру, али су у суштини то различите варијанте исте инструкције, јер се операциони кôдови разликују у делу који дефинише врсту и начин адресирања операнада.

архитектури *x86-64*, инструкција `mov` се користи и за трансфер између регистара, и за трансфер између регистра и меморије. Са друге стране, неке архитектуре користе посебне инструкције за сваку од ове две врсте трансфера. Ово је посебно карактеристично за *load/store* архитектуре које имају посебне инструкције за комуникацију са меморијом. На пример, *ARM32* архитектура користи посебне инструкције `ldr` и `str` за трансфер из меморије у регистар и обрнуто. Са друге стране, за трансфер између регистара користи се инструкција `mov`.

Посебан вид инструкција трансфера су инструкције за рад са системским стеком, с обзиром да се он такође налази у меморији. О њима ће бити више речи у наредном одељку.

Аритметичко логичке инструкције. Ове инструкције обављају аритметичко-логичке операције, користећи ALU јединицу процесора. Већина процесора подржава операције сабирања, одузимања, поређења (која се своди на одузимање) и битовских операција конјункције, дисјункције, негације и ексклузивне дисјункције. Операције померања у лево и у десно, као и ротације битова су такође подржане на већини архитектура. Ово је зато што су логичка кола за реализацију наведених операција веома једноставна. У сложеније операције спадају множење и дељење. Множење може бити реализовано као комбинаторно коло, а може бити реализовано и кроз алгоритам коначног аутомата контролне јединице, свођењем на узастопно сабирање и померање (попут *Бутовог алгоритма*). У сваком случају, множење може захтевати више циклуса часовника. Слична је ситуација са дељењем. Треба имати у виду да множењем два n-битна чиниоца добијамо производ који има $2n$ битова. Због тога нам могу бити потребна два n-битна регистра за чување производа. На пример, на архитектури *x86-64*, инструкција `mul` множи 64-битни чинилац у регистру `rax` са другим 64-битним чиниоцем који је наведен као операнд инструкције. Нижи део производа се смешта у `rax`, док се виши део производа смешта у регистар `rdx`. Уколико су вредности чинилаца довољно мале, тада ће сви значајни битови производа стати у нижи део, тако да ће виши део садржати само нуле, те га можемо игнорисати. Ипак, регистар `rdx` ће бити анулиран, што може покварити његову претходну вредност. Код операције дељења је ситуација обрнута – делимо $2n$-битни дељеник n-битним делиоцем и добијамо два n-битна резултата – целобројни количник и остатак. На пример, на архитектури *x86-64*, инструкција `div` узима 128-битни дељеник чији се виши део налази у регистру `rdx`, а нижи у регистру `rax`, дели га 64-битним операндом инструкције и целобројни количник смешта у `rax`, а остатак у `rdx`. Такође, треба напоменути да, за разлику од сабирања и одузимања, код множења и дељења постоји разлика између неозначених и означених операција. Због тога постоје посебне инструкције за неозначено и означено множење, односно дељење (`mul` и `imul`, односно `div` и `idiv` на *x86-64* архитектури). Ипак, када је множење у питању, ако нас занима само нижи део производа, тада не постоји разлика између означеног и неозначеног множења, те можемо користити било коју инструкцију (обично се у том случају користи `imul` која врши множење Бутовим алгоритмом, који је нешто ефикаснији од алгоритма множења неозначених бројева).

Због своје сложености, инструкције множења и дељења не морају

бити подржане на свим архитектурама. На пример, архитектура *ARM32* подржава само инструкције множења, док се дељење мора емулирати софтверски, свођењем на узастопно одузимање. Када је множење у питању, *ARM32* подржава инструкцију `mul` која рачуна само нижи део производа (и стога се користи и у означеном и у неозначеном случају), као и инструкције `umull` и `smull` које рачунају цео производ, у неозначеном и означеном случају, респективно.

Инструкције контроле тока. У инструкције контроле тока спадају инструкције безусловног и условног скока. Операнд ових инструкција је адреса на коју треба скочити. У случају директног адресирања, наводи се апсолутна адреса скока, док се у случају релативног адресирања наводи означени померај у односу на адресу текуће инструкције (тј. саме те инструкције скока). Поједине архитектуре подржавају и регистарско индиректно адресирање, у ком случају се адреса скока налази у наведеном регистру.

Инструкција безусловног скока скаче увек, док инструкције условног скока испитују одређени услов и скачу само ако је тај услов испуњен, док у супротном програм наставља са извршавањем следеће инструкције у меморији. Услов који се испитује је најчешће формулисан у терминима вредности флегова у статусном регистру, као што је описано у примеру рачунара у поглављу 5.2. Понекад се услови могу формулисати и на другачији начин. На пример, на архитектури *x86-64* постоји инструкција `jrcxz` која проверава вредност регистра `rcx` и скаче ако и само ако је вредност тог регистра једнака нули.

Постоје и архитектуре код којих се не проверава услов над флеговима (који су претходно постављени нпр. инструкцијом поређења), већ се и поређење и сам скок врши у склопу инструкције условног скока. Такав је случај, на пример, са архитектуром *MIPS*:

```
beq $t0, $t1, label
```

Ова инструкција најпре пореди регистре `$t0` и `$t1` на једнакост (одузимањем), а затим, уколико јесу једнаки, скаче на дату лабелу, док у супротном програм наставља са наредном инструкцијом у меморији. Слично, инструкција:

```
beqz $t0, label
```

скаче на дату лабелу ако и само ако је вредност регистра `$t0` једнака нули. Дакле, уместо да користимо две инструкције, сада и поређење и скок вршимо у једној инструкцији.

У инструкције контроле тока спадају и инструкције које се користе за позивање потпрограма и враћање из њих. Иако ове инструкције формално нису неопходне (јер њихов ефекат можемо симулирати обичним инструкцијама скока уз претходно памћење повратне адресе), њихово постојање умногоме олакшава програмирање, те их већина савремених архитектура подржава. О овим инструкцијама говорићемо нешто више у наредном одељку.

Системске инструкције. Инструкције описане у претходним параграфима су довољне за имплементацију било ког алгоритма. Ипак, рад савремених рачунара захтева истовремено извршавање великог броја програма, што захтева пажљиву и ефикасну контролу ресурса који су додељени тим програмима. Ту улогу обавља *оперативни систем*, кога чини скуп програма посебно написаних тако да ефикасно управљају радом рачунара, како хардверских компоненти, тако и корисничких програма. Да би оперативни систем могао да обавља своју улогу, неопходна му је одређена хардверска подршка, која се састоји из одређених ресурса процесора, попут посебних системских регистара и придружене логике, као и одређеног скупа инструкција које омогућавају употребу тих ресурса на одговарајући начин. У ове инструкције убрајамо *инструкције за промену режима рада* (на пример, процесор може бити у 32-битном или 64-битном режиму), *инструкције за контролу нивоа привилегија* под којима се кôд извршава (како бисмо могли да обезбедимо да кориснички кôд нема приступа неким осетљивим ресурсима рачунара којима може приступати само оперативни систем), *инструкције за контролу прекида* (на пример, ако желимо да изазовемо софтверски прекид или да спречимо изазивање хардверских прекида током неког временског периода), и тд. Више детаља о системским инструкцијама и поменутим функционалностима процесора биће дато касније.

6.1.4 Позивање потпрограма и системски стек

Потпрограм је један од незаобилазних концепата у вишим програмским језицима. Као што сама реч каже, потпрограм представља издвојени део програма који обавља један специфични део целокупног посла који програм треба да обави. Потпрограм се може *позвати* из главног програма или другог потпрограма (који називамо *позивалац*, енгл. *caller*), како би обавио свој задатак. У том тренутку, извршавање позиваоца се зауставља на месту позива и контрола тока се преусмерава на *позвани* потпрограм (енгл. *callee*), који започиње своје извршавање. Након завршетка рада позваног потпрограма, врши се *повратак* контроле тока (енгл. *return*) позиваоцу, који наставља своје извршавање тамо где је стао.

Потпрограм се може позивати више пута, са исте локације или са различитих локација у програму. Он увек извршава исти задатак, али не увек над истим подацима. Наиме, потпрограм може имати *улазне параметре* који представљају податке над којима ће се извршавати кôд потпрограма. Ови параметри се приликом позива иницијализују *аргументима позива*, који представљају конкретне податке које потпрограму предаје позивалац. На тај начин, исти алгоритам се може позивати над различитим подацима.

Улазни параметри омогућавају пренос података из позиваоца у потпрограм. Са друге стране, да би позивалац могао да види резултате рада потпрограма, неопходно је обезбедити и пренос података у супротном смеру, од позваног потпрограма ка позиваоцу. Један начин да се то обезбеди је путем *излазних параметара*. Излазни параметри се приликом позива везују за конкретне податке позиваоца, тако да свако реферисање на излазне параметре унутар потпрограма заправо користи одговарајуће податке позиваоца. Отуда упис вредности у излазне параметре од

стране потпрограма постаје видљив позиваоцу, кроз промену вредности одговарајућих аргумената. Улазни параметри се обично преносе *по вредности*, тј. вредности аргумената се копирају локално и потпрограм на даље ради са тим копијама. Излазни параметри се преносе *по адреси* (или *по референци*), тј. потпрограму се прослеђују адресе оригиналних аргумената позиваоца, како би директно могао да их модификује. Поред излазних параметара, други начин на који потпрограм може саопштавати своје резултате позиваоцу је путем *повратне вредности*, која садржи оно што је потпрограм израчунао. Притом, синтакса враћања вредности у потпрограму и њеног очитавања у позиваоцу зависи од конкретног програмског језика.

Потпрограм може имати своје *локалне променљиве*. Оне су доступне само у потпрограму и не могу се користити у другим деловима програма. Потпрограм их користи за своја интерна израчунавања, чување међурезултата и сл.

Традиционално, потпрограми се деле на *функције* и *процедуре*. Функције представљају потпрограме који имају само улазне параметре, а резултат свог рада позиваоцу враћају искључиво кроз повратну вредност. Позив функције се обично сматра *изразом* чија је вредност управо оно што функција врати позиваоцу. Самим тим, позив функције се може користити као подизраз у сложенијим изразима (попут $f(x) + y$) у позиваоцу. Са друге стране, процедуре представљају потпрограме који немају повратну вредност, већ за комуникацију са позиваоцем (уколико је потребна) користе излазне параметре. Позив процедуре се не сматра изразом, те се он у позиваоцу увек наводи као посебна наредба. Оваква стриктна подела потпрограма је карактеристична за програмски језик *Pascal*, који је дуго био први избор приликом учења програмирања. Савремени језици су углавном засновани на програмском језику *C*. У програмском језику *C* (и већини језика изведених из њега, попут језика *C++*, *Java*, *JavaScript*, *C#*, и сл.), сви потпрограми се називају *функцијама*, с тим што је појам функције уопштен тако да обухвата и оно што се традиционално сматра процедуром. Тако функције у *C*-оликим језицима могу имати повратну вредност типа `void`, што представља *празан* тип, тј. тип који нема вредности. То заправо значи да такве функције не враћају ништа. Слично, *C*-олике функције могу преносити параметре и по адреси (коришћењем показивача у *C*-у, референци у *C++*-у, или `out` спецификатора у *C#*-у), што омогућава коришћење излазних параметара. Ми ћемо, стога, у наставку појмове *функција* и *потпрограм* сматрати еквивалентним. Посебан тип функција које се јављају у тзв. *објектно-оријентисаним* програмским језицима су *методе*. Методе су функције које су придружене одређеним типовима података (које називамо *класама*) и чији се позив увек везује за један податак одговарајуће класе (такве податке називамо *објектима*). Метода може очитавати и модификовати податке који су садржани у објекту и на тај начин може утицати на објекат за који је позвана. Ипак, са становишта имплементације позива метода у асемблеру, не постоје суштинске разлике у односу на обичне функције.

Главни разлог за поделу целокупног програма на потпрограме је лакше решавање задатка које програм треба да обави, у складу са девизом *подели па владај* (енгл. *divide-and-conquer*): проблем се дели на мање потпроблеме које је појединачно лакше решити. Сада се сваки од

потпроблема решава засебно, а затим се добијена парцијална решења склапају у једно интегрално решење. Такође, чест је случај да се нека функционалност захтева више пута током решавања главног проблема (нпр. потребно је више пута израчунати максимум два броја или извршити претрагу у низу). Уместо да се исти кôд копира на више места, сада га имамо само на једном месту и онда га позивамо тамо где нам је та функционалност потребна. На овај начин, програм се значајно скраћује, а такође се олакшава његово одржавање, јер се било која промена или надоградња може обавити само на једном месту. Најзад, више сродних потпрограма се може организовати у *библиотеку*, која се онда може увозити и делити од стране више различитих програма. На тај начин се обезбеђује *поновно коришћење кôда* (енгл. *code reusability*), где једном имплементиране функционалности могу бити коришћене изнова у новим програмима.

У овом одељку разматрамо начине имплементације механизма позивања потпрограма на нивоу асемблерског језика. Ово је значајно зато што се, на крају крајева, сваки програм на вишем програмском језику мора превести на асемблер, а затим и на машински језик. Познавање ових механизама је значајно из два разлога. Први је то што можемо боље разумети како наш програм функционише и увидети потенцијалне изворе неефикасности добијеног програма након превођења. На пример, преношење аргумената по адреси може бити много ефикасније него копирање вредности у локалне параметре функције, чак и ако не намеравамо да их користимо као излазне параметре, нарочито ако су у питању велики подаци (нпр. структуре и низови). Слично, враћање вредности из функције обично подразумева копирање вредности у простор који обезбеди позивалац. Отуда је и враћање великих података често врло неефикасно. Други разлог је то што је често потребно повезати кôд написан на асемблеру са остатком програма који је написан на вишем програмском језику. Тада је неопходно разумети које механизме користи преводилац приликом позивања програма, како би се повезивање извршило на одговарајући начин.

Постоји неколико сегмената у поступку позивања потпрограма које треба размотрити приликом имплементације тог поступка на асемблеру. Први међу њима је *чување адресе повратка*. Са становишта асемблерског (и машинског) програма, позив потпрограма је најобичнији безусловни скок, с том разликом што се пре самог скока запамти адреса повратка. У питању је адреса инструкције која следи непосредно након инструкције скока. Након што се програмски кôд потпрограма заврши, скаче се на запамћену адресу повратка како бисмо омогућили наставак извршавања кôда позиваоца.

Други сегмент је *пренос аргумената*, тј. копирање њихових вредности (или адреса) у локалне параметре, како би потпрограм могао да их користи. Трећи сегмент јесте реализација *враћања повратне вредности*. Она се мора копирати на локацију која ће бити доступна позиваоцу након повратка из потпрограма. Најзад, мора се извршити *алокација простора* за локалне променљиве.

У свим овим сегментима, кључну улогу игра *системски стек* или *стек позива* (енгл. *call stack*). За системски стек се резервише континуални простор у оперативној меморији. На пример, на архитектури *x86-64*, за стек се користи виши део адресног простора, тј. стек креће од највише адресе и „расте" ка нижим адресама приликом постављања нових елемената. Податак на врху стека се, стога, налази на најнижој адреси коју стек

тренутно заузима. Да бисмо могли да радимо са стеком, све што је потребно је да знамо на којој адреси се тренутно налази врх стека. У ту сврху обично користимо неки регистар у процесору који називамо *показивачем стека* (енгл. *stack pointer*). Да бисмо очитали вредност са врха стека, можемо користити регистарско индиректно адресирање над регистром показивача стека. Слично, да бисмо приступали другим подацима који се налазе „дубље" у стеку, можемо користити адресирање „база + померај", при чему се регистар показивача стека користи као базни регистар (као што је објашњено у једном од ранијих одељака). На пример, на архитектури *x86-64*, регистар `rsp` се обично користи као показивач стека, па се приступ подацима на стеку може обавити на следећи начин:

```
mov rax, [rsp]  # ocitavamo podatak na vrhu steka
mov rax, [rsp + 4] # pristupamo sledecem podatku na steku
```

Нарочито треба разјаснити другу инструкцију: овде претпостављамо да је податак на врху стека величине 4 бајта, па је отуда следећи податак испод врха стека на адреси `rsp + 4`, имајући у виду да стек „расте" ка нижим адресама.

Додавање новог податка на стек се може обавити на следећи начин:

```
sub rsp, 4
mov [rsp], eax
```

Дакле, најпре алоцирамо простор на врху стека тако што показивач врха стека смањимо за 4. Затим у тај алоцирани простор на врху стека упишемо жељену вредност (вредност 32-битног `eax` регистра), користећи регистарско индиректно адресирање.

Скидање податка са врха стека се обавља на следећи начин:

```
mov eax, [rsp]
add rsp, 4
```

Дакле, најпре копирамо 4-бајтни податак са врха стека у регистар `eax`, а затим увећавамо вредност показивача стека за 4, чиме показивач врха стека померамо да показује на следећи податак на стеку, испод управо скинутог. Приметимо да податак који смо скинули са стека није физички обрисан из меморије – он је остао ту, али више није на стеку, јер се зона меморије коју стек заузима (ограничена показивачем врха стека) померила ка вишим адресама. Овај податак ће извесно бити пребрисан приликом наредног постављања неког другог податка на стек. Приметимо да ако само желимо да скинемо вредност са стека, али нам није потребна у даљем програму, можемо изоставити прву инструкцију и само увећати показивач стека.

Како би се горе описани рад са системским стеком поједноставио, поједине архитектуре нуде посебне инструкције за постављање и скидање података са стека. Ове инструкције обично имплицитно раде са неким фиксираним регистром, што нас ограничава да као показивач врха стека морамо користити баш тај регистар. На пример, на архитектури *x86-64*, постоје инструкције `push` и `pop` које се користе редом за постављање и скидање података са стека:

```
push rax  # postavljamo vrednost registra rax na stek
pop rax   # skidamo 8-bajtnu vrednost sa steka i smestamo je u rax
```

Ове инструкције аутоматски ажурирају вредност `rsp` регистра, као што је горе описано.

На архитектури *ARM32*, за рад са стеком могу се користити инструкције `stmfd` и `ldmfd` (иако њихова намена није строго ограничена на рад са стеком). Ове инструкције захтевају да се експлицитно наведе регистар који се користи као врх стека. На пример:

```
stmfd sp!, {r0}  # postavljamo registar r0 na vrh steka sp
ldmfd sp!, {r0}  # skidamo sa steka sp vrednost i smestamo je u r0
```

У овом примеру, као показивач врха стека користи се регистар `sp` који се стандардно користи на *ARM32* архитектури у ову сврху. Ипак, архитектура допушта и употребу других регистара као показивача стека, што даје додатну флексибилност приликом имплементације механизма позивања потпрограма. Приметимо да је симбол `!` иза назива регистра (`sp`) неопходан да би се назначило да желимо да ажурирамо вредност регистра након извршене операције (тј. ажурирамо адресу врха стека након уписа, односно читања).

Да бисмо дочарали улогу стека приликом имплементације механизма позивања функција, посматрајмо следећи пример. Претпоставимо да приликом извршавања програма функција f позива функцију g, која даље позива функцију h. Позив функције h ће се завршити пре позива функције g, а позив функције g ће се завршити пре позива функције f. Другим речима, функција која је последња позвана прва ће се завршити. Како се параметри функције и њене локалне променљиве алоцирају приликом уласка у функцију, а деалоцирају приликом изласка из функције, следи да је потребно најпре деалоцирати податке функције која је последња позвана (а која ће се прва завршити), тј. најпре деалоцирамо податке који су последњи алоцирани. Исто важи и за повратне адресе – њих памтимо приликом позива функције, а заборављамо када се из функције вратимо (јер нам више нису потребне). То значи да ће повратна адреса функције која је прва позвана најдуже бити чувана (јер њен позив траје најдуже), док ће се повратна адреса последње позване функције прва заборавити. Отуда се стек природно намеће као структура која ће се користити за чување локалних променљивих и параметара функција, као и повратних адреса.

За илустрацију употребе стека у ову сврху, посматрајмо пример архитектуре *x86* (у питању је 32-битна претеча архитектуре *x86-64*). Ова архитектура је као показивач стека користила 32-битни регистар `esp`. За позивање функције коришћена је инструкција `call` која је као једини операнд имала адресу од које почиње кôд функције, тј. на коју треба безусловно скочити. Ова инструкција непосредно пре самог скока аутоматски израчунава адресу повратка (адреса инструкције која следи након инструкције `call`) и поставља је на стек (уз аутоматско ажурирање регистра `esp`). Приликом повратка из функције, позива се инструкција `ret`. Ова инструкција нема операнде. Она скида са врха стека 32-битну вредност, тумачи је као адресу повратка и скаче на ту адресу. Дакле, ове две инструкције увек иду у пару, а на програмеру (или преводиоцу) је

да обезбеди да у тренутку позива инструкције `ret` на врху стека заиста буде повратна адреса коју је инструкција `call` поставила (тј. све што је у међувремену стављано на стек мора се скинути пре позива инструкције `ret`).

Аргументи се такође преносе преко стека, а то је одговорност позиваоца. Пре самог позива, позивалац ће поставити на стек аргументе (`push` инструкцијом), и то у обрнутом редоследу (тј. последњи аргумент функције се поставља први на стек). Након постављања аргумената врши се позив функције `call` инструкцијом. Након што се контрола врати позиваоцу, он је дужан да аргументе скине са стека (увећавањем показивача врха стека за одговарајући број бајтова). На пример:

```
push eax
push ecx
call func
add esp, 8
```

Дакле, имамо два 4-бајтна аргумента, па је приликом скидања аргумената са стека након позива потребно увећати показивач стека за 8.

Унутар позване функције можемо приступати аргументима на стеку на раније описан начин, али и алоцирати додатни простор на стеку за локалне променљиве (смањивањем вредности показивача врха стека). Притом, ово померање врха стека утиче на померај који је потребно користити да би се приступало аргументима функције. Како би оријентација на стеку била једноставнија, уводи се појам *оквира стека* (енгл. *stack frame*). У питању је део стека који садржи локалне податке једне активне функције. Свака активна функција има свој оквир на стеку. Када се функција позове, прво што она ради јесте да алоцира нови оквир на врху стека за себе. У ову сврху се користи други регистар, тзв. *показивач оквира стека* (енгл. *stack frame pointer*). Овај регистар се приликом иницијализације оквира стека поставља на вредност показивача врха стека. Након тога се показивач врха стека смањује за одговарајући број бајтова, колико је потребно за локалне променљиве позване функције. Оквир стека текуће функције је, дакле, зона на стеку између показивача оквира стека и показивача врха стека. На архитектури *x86* као показивач оквира стека користи се регистар `ebp`. На пример:

```
func:
  push ebp
  mov ebp, esp
  sub esp, 8

  # ovde ide kod funkcije

  mov esp, ebp
  pop ebp
  ret
```

Горњи кôд захтева додатно појашњење. Наиме, како се исти регистар `ebp` користи као показивач оквира стека за све функције, његовим ажурирањем у позваној функцији бисмо изгубили информацију о адреси почетка оквира

стека претходне (позивајуће) функције. Да се то не би десило, најпре на стек постављамо тренутну вредност регистра `ebp`, тј. адресу оквира стека позивајуће функције. Након тога иницијализујемо наш (празан) оквир стека постављањем регистра `ebp` на адресу врха стека `esp`. Након тога врх стека „спуштамо" за 8 бајтова, чиме алоцирамо простор за две 32-битне локалне промењиве. Приметимо да ће након свих ових операција, први аргумент функције бити на адреси `ebp+8`, следећи на адреси `ebp+12` (под претпоставком да је први величине 4 бајта) и сл. Слично, прва локална промењива ће бити на адреси `ebp-4`, а друга на адреси `ebp-8`. С обзиром да се вредност регистра `ebp` током рада функције не мења, наведени помераји у односу на `ebp` остају исти све време, што олакшава приступ. Са друге стране, вредност регистра `esp` се може мењати, јер функција током свог рада може постављати нове податке на стек (нпр. приликом позива других потпрограма), што утиче на релативну позицију постојећих података у односу на врх стека. У томе је главна предност коришћења оквира стека.[8] На крају функције, пре извршења инструкције `ret`, потребно је најпре „испразнити" оквир стека (постављањем адресе врха стека на адресу почетка оквира стека), након чега на врху стека остаје претходно сачувана вредност адресе оквира стека позивајуће функције. Ми ту адресу скидамо `pop` инструкцијом и враћамо је у `ebp`, чиме смо поново активирали оквир стека позивајуће функције. На врху стека сада остаје повратна адреса (коју је поставила позивајућа функција `call` инструкцијом), коју ми скидамо `ret` инструкцијом и враћамо контролу позивајућој функцији. Горњи кôд се често краће записује на следећи начин:

```
func:
  enter 8,0

  # ovde ide kod funkcije

  leave
  ret
```

при чему инструкција `enter 0,0` замењује прве три инструкције у прологу функције, док инструкција `leave` замењује две инструкције које претходе `ret` инструкцији у епилогу функције.

Повратна вредност функције се прослеђује позиваоцу тако што се ископира у регистар `eax`. Када се вратимо у позивајућу функцију, она ће из тог регистра просто покупити враћену вредност. Овај приступ је могућ уколико је податак који враћамо целобројног или показивачког типа, па се може сместити у регистар. Са друге стране, поједини програмски језици (попут језика *C* и *C++*) допуштају да податак који се враћа из функције припада неком кориснички дефинисаном типу, попут структуре или класе. Овакви објекти могу бити веома велики и обично се не могу сместити у регистар. У том случају се прибегава следећој техници: позивалац најпре у свом оквиру стека алоцира простор довољно велики за чување повратне

[8] Наравно, коришћење оквира стека није неопходно. Штавише, модерни преводиоци могу да не креирају оквире стека приликом превођења, како би кôд био нешто краћи и ефикаснији. Са друге стране, када „ручно" пишемо функцију на асемблеру, препоручљиво је користити оквир стека, ради лакше оријентације и смањења могућности грешака. Такође, оквири стека олакшавају дебаговање програма.

вредности (смањењем вредности регистра `esp` за одговарајући број бајтова). Затим се на стек постављају аргументи у обрнутом поретку, као и раније, уз додавање још једног, „скривеног" аргумента који садржи адресу на којој почиње простор који је алоциран за повратну вредност. Помоћу тог скривеног аргумента, позвана функција може приступити том простору и у њега уписати повратну вредност. Након што се вратимо из функције, позивалац скида аргументе са стека и на врху стека остаје повратна вредност.

Напоменимо да су неке специфичности претходно описаног поступка фиксиране самом архитектуром, док су неке резултат нашег избора. На пример, адресе повратка су се морале чувати на стеку, јер инструкције `call` и `ret` просто тако функционишу. Слично, регистар `esp` се мора користити као показивач врха стека, јер инструкције `call`, `ret`, `push` и `pop` то подразумевају. Са друге стране, нигде није архитектуром фиксирано да се аргументи морају преносити преко стека, нити је специфицирано да се они на стек морају смештати у обрнутом регистру. Такође, избор регистра `eax` за чување повратне вредности је такође ствар нашег избора. Ипак, ако желимо да функције које ми пишемо у асемблеру можемо позивати из нпр. програма писаног у језику *C* или *C++*, морамо се држати *конвенција* које прате преводиоци које користимо за те језике. С тим у вези, поступак који је горе описан управо прати конвенције које користе преводиоци на оперативном систему *Linux* на *x86* архитектури. Други оперативни системи могу подразумевати другачије конвенције од горе описаних.

Алтернатива коришћењу стека приликом преноса аргумената и чувања адресе повратка је да се у те сврхе користе регистри процесора. Предност овог приступа је у брзини, јер се регистри налазе у процесору, док је стек у меморији. Са друге стране, овакав приступ захтева да процесор има довољан број регистара. На пример, архитектура *x86* је имала свега 8 регистара опште намене, па је такав приступ био непрактичан (ако све регистре искористимо за аргументе, нећемо имати у чему да рачунамо). Са друге стране, архитектура *x86-64* је проширила скуп регистара на 16 64-битних регистара[9], што даје простора да се примени овакав приступ. Слична ситуација је и са нпр. *ARM32* архитектуром, која такође поседује 16 регистара опште намене, као и са архитектурама *ARM64* (31 регистар) и *MIPS* (32 регистра).

Примера ради, илуструјмо начин позивања функција на *x86-64* архитектури. Као и код њеног претходника, архитектуре *x86*, за позивање функције и враћање из ње користи се пар инструкција `call` и `ret`, које функционишу на потпуно исти начин као и раније, с тим што је адреса повратка сада 64-битна. Ово значи да се и даље стек мора користити за чување повратних адреса. Са друге стране, стандардне конвенције на *Linux* оперативном систему на *x86-64* архитектури сада подразумевају употребу регистара за пренос аргумената, због ефикасности. Ако претпоставимо да

[9] Регистри архитектуре *x86* су били 32-битни и имали су имена: `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `esp`, `ebp`, од којих су последња два имала специјалну улогу у раду са стеком. Отуда смо у суштини имали само 6 регистара опште намене. Архитектура *x86-64* ове регистре замењује 64-битним, чији се називи добијају заменом слова `e` словом `r` (дакле, `rax`, `rbx`, ...). Њихове ниже половине се и даље могу користити као 32-битни регистри под старим називима. Додатно, архитектура *x86-64* уводи 64-битне регистре `r8` до `r15`. Ово је укупно 16 регистара, од којих се 14 могу користити у опште сврхе, док се регистри `rsp` и `rbp` користе за рад са стеком.

функција има до шест аргумената, тада се они смештају редом у регистре `rdi`, `rsi`, `rdx`, `rcx`, `r8` и `r9`. Уколико функција има више од шест аргумената (што није тако често), преостали аргументи се постављају на стек, у обрнутом поретку. Повратна вредност функције смешта се у регистар `rax`. За локалне променљиве се могу користити преостали регистри, а могу се алоцирати и на стеку, на сличан начин као и раније. У ту сврху, на стеку се може иницијализовати оквир стека, на исти начин као и на *x86* архитектури.

Занимљив је и пример *ARM32* архитектуре. На овој архитектури, позив функције врши се инструкцијом `bl`. Ова инструкција као операнд има адресу функције на коју треба скочити. Међутим, адреса повратка се не чува на стеку, већ у посебном регистру `lr` (енгл. *link register*). Према *Linux* конвенцијама, прва четири аргумента се преносе преко регистара `r0`, `r1`, `r2` и `r3`, редом. Остали аргументи, ако постоје, преносе се преко стека, у обрнутом поретку. Повратна вредност смешта се у регистар `r0`. Један позив функције која има два аргумента би могао да изгледа овако:

```
mov r0, #1
mov r1, #5
bl func
# sada se u r0 nalazi povratna vrednost
```

Архитектура *ARM32* такође допушта креирање оквира стека, за потребе чувања локалних променљивих. У ту сврху се користи регистар `fp`. Структура функције која користи овај механизам би могла да изгледа овако:

```
func:
  stmfd sp!, {fp}
  mov fp, sp
  sub sp, sp, #8

  # ovde ide kod funkcije

  mov sp, fp
  ldmfd sp!, {fp}
  mov pc, lr
```

Првом инструкцијом се на стек поставља вредност регистра `fp` (показивач оквира стека позивајуће функције). Након тога се тренутни врх стека проглашава за почетак новог оквира стека, а затим се у њему алоцира простор од 8 бајтова за две 32-битне локалне променљиве (приметимо да и овде стек расте ка нижим адресама). Прва локална променљива ће бити на адреси `fp - 4`, а друга на адреси `fp - 8`. Алтернативно, можемо користити и регистре опште намене за чување локалних променљивих. На крају функције се празни оквир стека (тј. враћамо показивач стека на почетак оквира), затим скидамо са стека претходно запамћену вредност показивача оквира стека и враћамо је у `fp`, након чега скачемо на адресу повратка, простим копирањем повратне адресе у програмски бројач.

Као што је већ речено, главна предност коришћења регистара за чување повратне адресе и/или пренос аргумената је у ефикасности. Ипак, постоји и један битан недостатак: регистри су заједнички ресурс свих

функција, тј. све функције користе исте регистре у поступку позивања других функција. Овај проблем се најбоље уочава у случају када имамо више функција у ланцу позивања: ако имамо ланац позивања где функција f позива функцију g, а ова даље позива функцију h, тада функција g приликом позивања функције h мора користити исте регистре за пренос аргумената и/или чување адресе повратка које је користила и функција f приликом позива функције g. Отуда функција g мора своје аргументе и повратну адресу изместити на неко друго безбедно место пре него што позове функцију h. То безбедно место могу бити неки други регистри опште намене. Међутим, у случају дужих ланаца позивања, просто ће понестати слободних регистара, па ћемо морати да користимо стек. Ово је нарочито карактеристично за рекурзивне функције (функције које позивају саме себе), код којих је неопходно да чувају своје аргументе и повратне адресе на стеку пре него што позову рекурзивни позив. Дакле, у овим екстремним ситуацијама, процес се своди на оно што бисмо имали и да смо одмах користили стек, само је нешто компликованије за програмера. Међутим, испостави се да су у пракси такви екстремни сценарији релативно ретки. Заправо, врло честе су функције које су сасвим једноставне и које не позивају ни једну другу функцију. Због тога се у пракси преношењем аргумената преко регистара значајно добија на ефикасности, па се тај приступ код модерних рачунара са већим бројем регистара по правилу и користи.

Употреба регистара у функцијама је нарочито критичан сегмент о коме треба водити рачуна. Као и у случају регистара који се користе за пренос аргумената (ако се користи тај приступ), регистри који се користе за чување локалних променљивих, као и регистри који се користе као привремени регистри приликом израчунавања у позваној функцији такође представљају дељени ресурс, заједнички за све функције. Отуда се лако може догодити да нека функција променом вредности неког регистра поквари неки драгоцен податак позивајућој функцији. Да се то не би догађало, конвенције о позивању потпрограма обично скуп регистара деле у два подскупа: регистри који припадају позваној функцији и регистри који припадају позивајућој функцији. Ако регистар припада позваној функцији, она има право да слободно користи тај регистар и не мора да претпоставља да се у њему налази било шта значајно за позивајућу функцију. Са друге стране, позивајућа функција мора сачувати на безбедном месту све податке из ових регистара пре него што позове неку другу функцију. То безбедно место може бити стек или неки регистар који припада позиваоцу. Са друге стране, ако регистар припада позиваоцу, тада је позвана функција дужна да сачува његову вредност за позиваоца. То значи да ако позвана функција жели да користи овакав регистар, она најпре мора сачувати његову вредност (типично на стеку). На крају позване функције, оригиналне вредности ових регистара се скидају са стека и враћају у регистре. Последица описаног понашања је да позивајућа функција сме да остави своје драгоцене податке у овим регистрима, без бојазни да ће позвана функција покварити њихову вредност. Посматрајмо пример на архитектури *x86* који ово илуструје:

```
func:
  push ebp
  mov ebp, esp
```

```
sub esp, 8

# registri esi, edi pripadaju pozivaocu
push esi
push edi
push ebx

# kod funkcije moze da koristi esi, edi
# kao i "svoje" registre eax, ecx, edx

# pozivamo drugu funkciju
# pretpostavimo da u ecx imamo nesto dragoceno

push ecx  # cuvamo registar ecx da nam ga gfunc ne pokvari

push eax  # postavljamo argument (u eax registru)
call gfunc
add esp, 4 # skidamo argument

pop ecx  # vracamo nasu vrednost registra ecx

# skidamo sacuvane vrednosti (u obrnutom poretku)
pop ebx
pop esi

# epilog funkcije
mov esp, ebp
pop ebp
ret
```

На архитектури *x86*, по конвенцији на оперативном систему *Linux*, регистри `esi`, `edi` и `ebx` припадају позивајућој функцији, док регистри `eax`, `ecx` и `edx` припадају позваној функцији. У горњем кôду, претпостављамо да функција користи регистре `esi` и `edi`, али не и `ebx`. Зато је прва два неопходно сачувати на стеку на почетку функције, а на крају их вратити на оригиналне вредности. Поред регистара `esi` и `edi`, функција може користити и регистре који припадају њој, као позваној функцији, и њих не мора чувати на стеку. Са друге стране, ако функција `func` позива неку другу функцију током свог рада, тада ће у контексту тог позива, функција `func` бити позивајућа функција. Претпоставимо да ова функција позива другу функцију `gfunc`, као и да у тренутку позива, функција има неку драгоцену вредност у регистру `ecx`. Како функција `gfunc` може променити овај регистар (јер је она сада позвана функција, па јој тај регистар припада), функција `func` га мора сачувати за себе пре позива, а затим га вратити на оригиналну вредност након позива (користећи стек). Са друге стране, ако у регистрима `esi` и `edi` имамо неке драгоцене податке, не морамо их чувати пре позива функције `gfunc`, јер ти регистри припадају нама као позиваоцу, те их позвана функција неће променити.

6.1.5 RISC и CISC архитектуре

У претходним одељцима смо разматрали различите аспекте архитектура. С обзиром на те аспекте, архитектуре се могу разврставати на различите начине. Нека груба подела која се током претходних деценија појавила у литератури је подела на архитектуре *RISC* и *CISC* типа. Ова подела је прилично груба, имајући у виду да многе архитектуре имају извесне карактеристике и једног и другог типа. Ипак, са академског становишта, битно је разликовати ова два типа архитектура и разумети њихове основне карактеристике.

Архитектуре CISC типа (енгл. *complex instruction set computer*) су настале током 60-тих и 70-тих година 20. века. Типични представници таквих архитектура у то време биле су архитектуре *IBM S/360*, *PDP-11*, *VAX*, као и архитектуре *8080* и *8086* компаније *Intel*. Ове две последње архитектуре су претеча архитектура *x86* и *x86-64* које су данас типични представници модерних CISC архитектура. Да бисмо разумели основне принципе дизајна CISC архитектура, потребно је најпре разумети контекст у ком су те архитектуре настајале. Наиме, крајем 60-тих и почетком 70-тих година 20. века долази до великог напретка у области софтверских технологија, пре свега оличеног у настанку нових програмских језика који су укључивали све више напредних концепата високог нивоа (структурни програмски језици, објектно оријентисани програмски језици и сл.). Ови концепти су омогућавали програмеру да много једноставније, концизније и апстрактније описује алгоритам. Са друге стране, то је стварало *семантичку празнину* (енгл. *semantic gap*) између онога што је оваквим језицима било могуће изразити и онога што је директно подржано од стране хардвера. Ово је отежавало процес превођења, јер су сложене функционалности виших програмских језима морале да се преводе на дугачке низове машинских инструкција елементарног нивоа. Са једне стране, то је компликовало конструкцију преводиоца, а са друге стране је продужавало програме и увећавало простор потребан за њихово смештање на диску и у оперативној меморији. Како су у то време меморије биле веома скупе и, последично, малих капацитета, компактније и краће изражавање алгоритама на нивоу машинског језика је постало императив. Такође, како су меморије биле веома споре, дугачки програми са великим бројем инструкција су захтевали велики број меморијских приступа (јер је много инструкција требало дохватити из меморије). Најзад, у време настанка CISC архитектура, регистри опште намене су били скупи и било их је мало. Отуда је било корисно да постоји могућност директног оперисања са подацима у меморији, уместо да их прво довлачимо у регистре процесора.

Да би се ови проблеми бар делимично превазишли, дизајнери процесора су дошли на идеју да у архитектуру процесора уграде што више концепата вишег нивоа. Тако настају CISC архитектуре. Основна карактеристика ових архитектура је да поседују моћне инструкције које су у стању да обављају више елементарних операција, или чак неке врло сложене операције (отуда оно *complex* у називу). На пример, једна од типичних карактеристика CISC архитектура је да њихове аритметичке инструкције могу имати меморијске операнде. Таква инструкција ће, на пример, учитати операнд из меморије, сабрати га са неким регистром, а затим добијени збир поново сместити у меморију. Приметимо да ова инструкција заправо ради три ствари – два

меморијска трансфера, и једну ALU операцију. Овако нешто је од изузетног значаја када имате ограничење у погледу величине и брзине меморије, као и броја доступних регистара. Такође, CISC архитектуре обично подржавају посебне инструкције за позиве процедура, постављање регистара на стек, имплементацију петљи, рад са сложеним типовима података попут стрингова и сл. Примера ради, архитектура *x86* поседује инструкције `pusha` и `popa` које истовремено на стек постављају све регистре опште намене, као и инструкције `enter` и `leave`, које се користе за компактнију имплементацију креирања и поништавања оквира стека. Наведене инструкције служе као подршка за ефикасно позивање потпрограма, што је једна од основних карактеристика модерних програмских језика. Слично, постоји инструкција `loop` која умањује регистар `ecx` за један, проверава да ли је након тог умањења регистар `ecx` анулиран и, ако није, скаче на дату адресу. Дакле, ова инструкција комбинује умањење регистра и условни скок, а користи се за ефикасну имплементацију бројачких петљи. Најзад, архитектуре CISC типа често имају подршку за неке врло сложене операције, попут претраге и упоређивања стрингова, израчунавања корена, синуса или логаритма. Овакве сложене операције израчунавају се у великом броју циклуса у аутомату контролне јединице, а програмер може да их једноставно позива у виду инструкција које му архитектура нуди.

Последица описаног дизајна архитектуре се огледа у великом броју различитих формата инструкција. С обзиром да већина инструкција могу имати било регистарске, било меморијске операнде, неопходно је обезбедити различите варијанте истих инструкција, што компликује операциони кôд. Како би приступ меморијским операндима био једноставнији за програмера, потребно је обезбедити различите начине адресирања (попут директног, индиректног, индексног, и сл.). Све ово ствара потребу за различитим, често врло неуједначеним форматима инструкција, и у погледу дужине и у погледу начина кодирања. На пример, инструкције на архитектури *x86-64* могу бити дугачке од 1 до 15 бајтова, у зависности од формата. Оваква разноликост формата инструкција доводи до знатно комплекснијег поступка дохватања и декодирања инструкције (наравно, и само извршавање инструкције је комплексније, због саме сложености операције коју инструкција обавља). Ово доводи до тога да је за већину инструкција (чак и оних врло једноставних) често потребно више циклуса да се изврше. Такође, модерне технике оптимизације рада процесора (попут технике *преклапања*, енгл. *pipelining*), се много теже имплементирају на тако неправилном скупу инструкција.

Основна претпоставка ефикасности CISC архитектура је била да ће програмски преводиоци искористити могућности сложених инструкција да једноставније генеришу кôд, као и да ће се тај код ефикасно извршавати, с обзиром да су многе функционалности директно имплементиране у процесору. Ипак, анализе су утврдиле да преводиоци нису увек тако вешти у коришћењу сложених инструкција које процесор нуди. Такође, испоставило се да у добијеним машинским програмима доминирају једноставне аритметичке инструкције, скокови и позиви потпрограма. Сложеније инструкције које CISC архитектуре нуде користе се знатно ређе.

Крајем 70-тих и почетком 80-тих година 20. века истраживачи долазе на идеју сасвим другачијег концепта архитектуре који би подразумевао да само најједноставније операције буду имплементиране директно у

процесору. Те могућности би биле дате програмеру (или преводиоцу) да помоћу њих у софтверу имплементира сложеније операције. Такве архитектуре су познате под називом *RISC* (енгл. *reduced instruction set computer*). Типични представници RISC архитектура су *ARM32*, *ARM64*, *MIPS*, *RISC-V*, *SPARC*, *IA-64* и сл. Основа карактеристика RISC архитектура је да аритметичко-логичке инструкције могу да раде само са регистарским и непосредним операндима. То значи да се подаци из меморије морају најпре довући у регистре (посебним инструкцијама које су за то намењене) пре него што се са њима нешто може радити. Раније смо рекли да се овакве архитектуре називају *load/store* архитектуре, а тај се термин често користи и као синоним за RISC архитектуре. Друга основна карактеристика је релативно велики број регистара опште намене, ретко мањи од 16, а обично и већи (нпр. архитектура *IA-64* компаније *Intel* има чак 128 регистра). Ово је неопходно на *load/store* архитектурама, како бисмо имали где да довлачимо операнде за аритметичке инструкције. Од аритметичко логичких инструкција, RISC архитектуре обично подржавају само једноставне операције, попут сабирања, одузимања, поређења, битовских операција, а понекад и инструкција множења и дељења. Сложеније операције нису директно подржане у процесору и морају се имплементирати софтверски, свођењем на ове једноставније. За RISC архитектуре је такође карактеристично да имају униформисани формат инструкција (типично су све инструкције исте дужине), као и ограничен скуп начина адресирања меморијских операнада (нпр. обично не подржавају директно адресирање, због ограничења дужине инструкције).

RISC архитектуре имају тенденцију да захтевају дуже програме, јер програмер мора да помоћу једноставних операција које процесор подржава реализује компликоване алгоритме. Са друге стране код CISC архитектура програми би у теорији требало да буду краћи, јер за многе сложене операције постоје инструкције које процесор директно подржава. Ипак, ово не значи да ће програми обавезно заузимати и мање меморије, имајући у виду дуже формате инструкција код CISC процесора.

Почев од 80-тих година RISC архитектуре постају све популарније, јер се околности које су раније ишле у прилог CISC архитектурама мењају: меморије постају све веће, меморијски трансфери захваљујући брзим *кеш* меморијама постају све бржи, а програмски преводиоци постају све бољи када је у питању оптимизација кôда, тако да имплементација сложених операција у софтверу више не заостаје у ефикасности у односу на хардверску реализацију. Такође, модерни процесори имају све већи број регистара, што је неопходно у реализацији *load/store* архитектура, као и других својстава која су карактеристична за RISC процесоре. Треба имати у виду и да једноставност RISC архитектура омогућава једноставну имплементацију процесора, што чини дохватање, декодирање и извршавање инструкција веома брзим. Такође, једноставност и униформисаност формата инструкција олакшава реализацију неких напредних техника паралелизације при извршавању инструкција, попут поменуте технике преклапања. Резултат свих тих побољшања је да се на RISC архитектурама у просеку извршава једна (или више) инструкција у сваком циклусу. Са друге стране, CISC архитектуре су гломазније, па чак

и извршавање једноставних инструкција обично захтева више циклуса.[10] Како ове једноставне инструкције доминирају у већини реалних програма, јасно је да ће ефикасност RISC архитектура доћи до изражаја у већини практичних апликација.

6.2 Организација централног процесора

Као што је раније речено, централни процесор се у организационом смислу састоји из путање података и контролне јединице. Притом, путању података чине ALU јединица, регистри процесора и линије које их повезују, а које ћемо називати *интерним магистралама* (треба их разликовали од меморијске магистрале која повезује процесор са оперативном меморијом и која се налази изван процесора). Наведене компоненте чине минималну организациону целину и присутне су у сваком процесору. Поред тога, модерни процесори могу садржати и бројне друге компоненте, попут меморијске јединице, контролера меморије, контролера магистрале и сл., али се ми тиме овде нећемо бавити.

6.2.1 Путања података

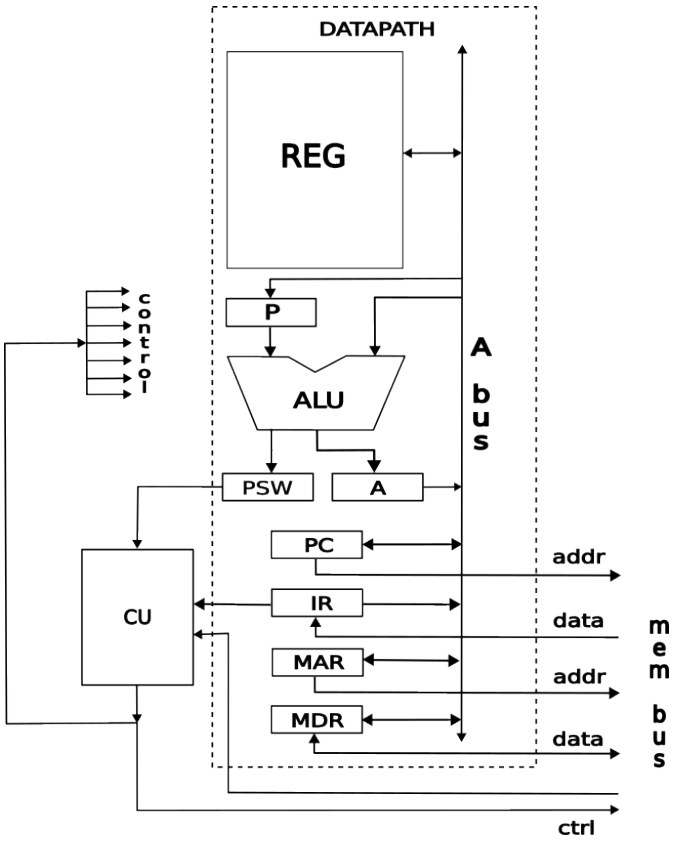
Структура путање података може бити различита, у зависности од тога како су компоненте међусобно повезане. Притом, основни параметар представља број интерних магистрала које су на располагању. У наставку разматрамо примере структура путања података са једном, две и три интерне магистрале.

На слици 6.3 приказана је структура процесора чија путања података има једну интерну магистралу. Путању података чине компоненте које су оивичене испрекиданом линијом, а интерна магистрала је означена са *A bus*. Пажљиви читалац ће приметити да је структуру попут ове имао рачунар који смо описали у поглављу 5.2. Основна карактеристика овакве структуре путање података је једноставност. Са друге стране, недостатак оваквог дизајна је у већем броју циклуса који је потребан да би се обавила нека операција. На пример, да бисмо сабрали вредности два регистра (означимо их, нпр., са R_0 и R_1) и њихов збир сместили у трећи регистар (означимо га са R_2), потребна су три циклуса:

1. $R_1 \rightarrow P$
2. R_0 $[add]$ $P \rightarrow A$
3. $A \rightarrow R_2$

Већи број интерних магистрала омогућава ефикаснију реализацију операција у оквиру путање података, јер је могуће вршити више трансфера у сваком циклусу. На пример, посматрајмо структуру путање података процесора датог на слици 6.4. У оквиру ове путање података имамо две магистрале – једну за читање (означену са *A bus*) и једну за упис (означену са *C bus*). Помоћу магистрале за читање можемо вредност било ког регистра

[10] Савремене CISC архитектуре овај недостатак обично надокнађују знатно већом фреквенцијом часовника, тј. броја циклуса у секунди.



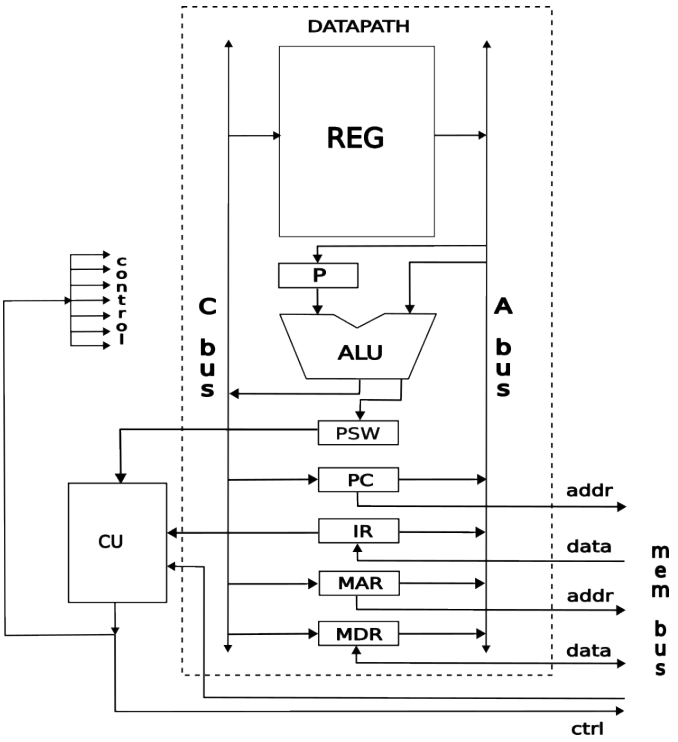
Слика 6.3: Процесор са путањом података са једном интерном магистралом

довести на улаз ALU јединице, док помоћу магистрале за упис можемо вредност са излаза ALU јединице пренети у било који регистар и тамо је сачувати. За разлику од путање података са једном магистралом, овде више није потрбан регистар A за чување вредности коју је ALU израчунала, јер ту вредност можемо одмах пренети на своје коначно одредиште, путем излазне магистрале. Са друге стране, регистар P је и даље потребан, с обзиром да имамо само једну магистралу за читање. Извршење операције $R_2 = R_0 + R_1$ из претходног примера би сада могло да се заврши у два циклуса:

1. $R_1 \rightarrow_A P$
2. R_0 $[add]$ $P \rightarrow_C R_2$

при чему ознака $\rightarrow_A$ (односно $\rightarrow_C$) означава траснфер путем A (односно C) магистрале.

На слици 6.5 дат је пример структуре процесора са три интерне магистрале. Путања података овог процесора има две магистрале за читање (означене са A *bus* и B *bus*) помоћу којих је могуће вредности два одабрана регистра истовремено довести на улазе ALU јединице, као и једну магистралу за упис (означену са C *bus*) помоћу које је резултат



Слика 6.4: Процесор са путањом података са две интерне магистрале

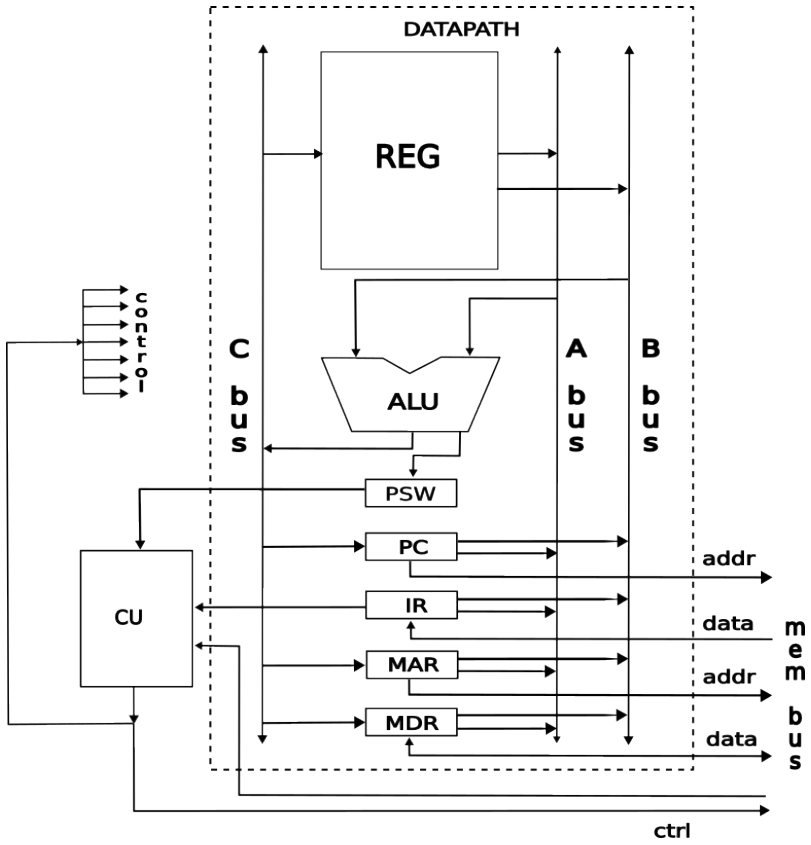
израчунавања ALU јединице могуће пребацити у жељени регистар и тамо га сачувати. Извршавање операције $R_2 = R_0 + R_1$ је сада могуће обавити у само једном циклусу:

1. R_0 $[add]$ $R_1 \rightarrow_C R_2$

при чему се подразумева да се први операнд ALU јединице преноси путем A магистрале, а други путем B магистрале.

Наравно, ефикасније извршавање операција на путањи података са већим бројем интерних магистрала има своју цену која се огледа у сложенијој имплементацији. Наиме, у случају путање података са једном магистралом, довољан је један декодер за избор регистра опште намене који ће се користити у одговарајућој операцији читања или уписа. Са друге стране, ако имамо две магистрале, тада истовремено можемо вршити и читање и упис, те је потребно истовремено реферисати на два регистра опште намене. За то су нам неопходна два декодера, који ће декодирати две адресе које генерише контролна јединица. Најзад, у случају путање података са три интерне магистрале, биће нам неопходна чак три декодера, с обзиром да је могуће у истом циклусу реферисати на три регистра опште намене истовремено (два за читање и један за упис).

На слици 6.6 приказан је начин повезивања појединачног регистра на магистрале у случају једне, две и три интерне магистрале. У првом случају, иста магистрала се користи и за читање и за упис, те је и улаз и излаз

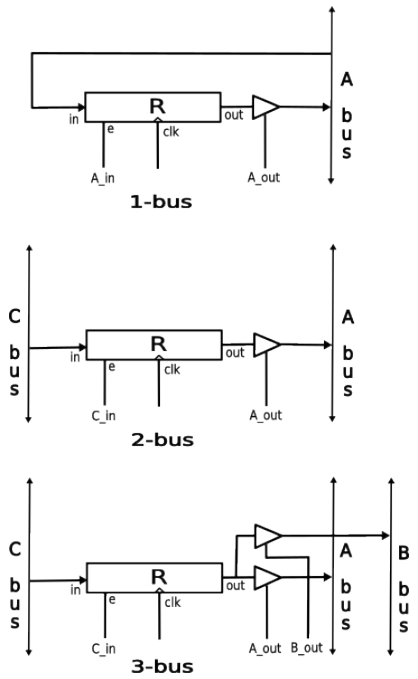


Слика 6.5: Процесор са путањом података са три интерне магистрале

регистра потребно повезати на њу. Два контролна сигнала (означени са A_{in} и A_{out}) се користе за одабир операције, при чему се A_{in} доводи на e улаз регистра (омогућивши упис), док се A_{out} користи као контролни сигнал бафера са три стања који контролише излаз вредности регистра на магистралу. Други случај је веома сличан, с тим што се за упис користи засебна C магистрала. У трећем случају имамо један контролни сигнал више (B_{out}), који контролише додатни бафер са три стања за излаз на B магистралу. У случају регистра опште намене (који се посматрају као скуп регистара у коме се избор регистара врши адресирањем), контролне сигнале за улаз и излаз генеришу декодери на основу одабраних адреса регистара.

6.2.2 Регистри специјалне намене процесора

Регистри опште намене (ако постоје) су по правилу део архитектуре процесора, тј. видљиви су програмеру и могу се користити у инструкцијама као регистарски операнди. Поред ових регистара, сваки процесор поседује и одређене регистре специјалне намене који служе за имплементацију интерних функционалности процесора. Ови регистри најчешће нису део архитектуре (уз ретке изузетке), те их није могуће користити у



Слика 6.6: Повезивање регистара на интерне магистрале (случај са једном, две и три магистрале)

инструкцијама као операнде. У наставку разматрамо најзначајније регистре специјалне намене који се јављају у већини процесора, а који су приказани на сликама 6.3, 6.4 и 6.5.

PC регистар. Овај регистар представља *програмски бројач* (енгл. *program counter*) и са његовом функцијом и улогом упознали смо се још у поглављу 5.2 – он садржи адресу инструкције коју следећу треба извршити. Путем ове адресе, процесор може да дохвати инструкцију из меморије и пребаци је у процесор. Вредност овог регистра се увећава након сваке инструкције тако да показује на следећу инструкцију у програму (ово је подразумевана семантика, с обзиром да се инструкције подразумевано извршавају једна за другом). У случају инструкције скока, вредност адресе инструкције на коју скачемо се директно уписује у овај регистар, чиме та инструкција постаје следећа коју треба извршити и која се дохвата у наредном циклусу. Регистар PC обично није директно доступан програмеру, већ се баратање овим регистром врши посредно, путем инструкција скока. Редак изузетак представља архитектура *ARM32*, код које регистар PC може бити операнд инструкције, као и сви други регистри опште намене. Зато се код ове архитектуре скок може извршити простим уписом жељене адресе у овај регистар (што се обично ради код повратка из потпрограма). Регистар PC је, поред интерних магистрала процесора, повезан и са адресном магистралом меморијске магистрале (што захтева додатни бафер са три стања за контролу излаза вредности овог регистра на адресну магистралу).

На овај начин, вредност регистра PC се може директно послати на адресни улаз оперативне меморије, што је неопходно код адресирања инструкције коју желимо да дохватимо.

IR регистар. Овај регистар представља *инструкциони регистар* (енгл. *instruction register*) и такође смо га већ упознали у поглављу 5.2. Његова улога је да чува операциони кôд и остале компоненте инструкције које су релевантне за рад контролне јединице. Вредност овог регистра се доставља на улаз контролне јединице, како би иста знала која се инструкција извршава и могла да, сходно томе, одреди наредне кораке у свом алгоритму. Улаз овог регистра је додатно повезан са магистралом података меморијске магистрале. На овај начин је омогућено да се приликом дохватања инструкције иста директно учита у IR регистар. Са друге стране, вредност овог регистра се може пустити и на интерне магистрале, како би се пренела у неки други регистар. Ово може бити потребно у случају да инструкција садржи апсолутну или релативну адресу меморијског операнда – тада је ту адресу потребно пребацити у MAR регистар или је пропустити кроз ALU јединицу ради израчунавања апсолутне адресе операнда.

MAR регистар. Још један регистар са чијом смо се улогом упознали раније. У питању је *адресни регистар меморије* (енгл. *memory address register*) и служи да се у њему формира адреса меморијских операнада које треба учитати из меморије или уписати у меморију. Због тога је овај регистар, осим са интерним магистралама, директно повезан на адресну магистралу меморијске магистрале. На овај начин, израчуната адреса се може директно слати на адресни улаз оперативне меморије, у циљу дохватања или чувања меморијског операнда. Приметимо да се на адресну магистралу повезују излази два регистра: MAR регистра и PC регистра. Избор која ће од ове две вредности бити послата на адресну магистралу врши се 2-на-1 мултиплексером (који није уцртан на сликама 6.3, 6.4 и 6.5 због једноставности).[11]

MDR регистар. У питању је *регистар меморије за податке* (енгл. *memory data registar*). Улога овог регистра је да се у њега смешта операнд који је управо дохваћен из меморије, као и да се у њему припрема податак који треба уписати у меморију као нову вредност меморијског операнда. Наиме, меморијски операнди се обично приликом дохватања уписују у неки од регистара опште намене. Слично, приликом уписа у меморијске операнде, обично у њих копирамо вредност неког регистра опште намене. Имплементирати директну везу свих ових регистара са магистралом података било би веома компликовано. Отуда се уводи посебан регистар који је повезан са магистралом података меморијске магистрале – то је управо MDR регистар. Приликом дохватања меморијског операнда, његова вредност се најпре учитава у овај регистар, а затим се из њега у

[11] Алтернатива овом приступу је била да само регистар MAR има везу са адресном магистралом, а да се приликом дохватања инструкције најпре вредност PC регистра пребаци у MAR. Тај приступ смо користили код модела рачунара у поглављу 5.2. Ипак, овај приступ захтева један циклус више приликом дохватања инструкције, те је у модерним рачунарима обично и PC регистар директно повезан на адресну магистралу.

наредном циклусу путем интерних магистрала пребацује у жељени регистар опште намене. Слично, приликом уписа, вредност жељеног регистра опште намене се пребацује интерним магистралама у MDR регистар, одакле се у наредним циклусима пребацује у меморију путем меморијске магистрале. Такође, сетимо се да у случају CISC архитектура аритметичке инструкције могу имати и меморијске операнде. У том случају се вредност меморијског операнда уопште не смешта у регистар опште намене, већ се користи као операнд ALU јединице. Ипак, он се не може директно довести на улаз ALU јединице, већ се најпре дохвата у MDR регистар, одакле се интерним магистралама доводи на улаз ALU јединице.

Веза MDR регистра са магистралом података је двосмерна, те је потребно и улаз и излаз овог регистра повезати са том магистралом (на сличан начин као што се A магистрала повезује са регистрима у случају путање података са једном интерном магистралом, видети горњи дијаграм на слици 6.6). Додатно, потребан је 2-на-1 мултиплексер за избор улазног податка (интерна магистрала или магистрала података).

PSW регистар. Овај регистар представља *статусни регистар процесора* (енгл. *program status word*). Битови овог регистра разматрају се засебно и представљају *заставице* или *флегове* (енгл. *flags*). Ови флегови се постављају од стране ALU јединице након извршене операције и служе да квалитативно опишу резултат операције. Са најзначајнијим статусним флеговима смо се упознали у поглављу 5.2. На неким архитектурама се вредности ових флегова постављају након сваке инструкције која подразумева употребу ALU јединице (попут аритметичко-логичких инструкција и инструкција трансфера). Примери таквих архитектура су *x86* и *x86-64*. Са друге стране, на неким архитектурама попут ARM32, флегови се ажурирају само у случају инструкције поређења (`cmp` инструкција), док остале аритметичко-логичке инструкције подразумевано не ажурирају флегове, осим ако није посебно назначено у операционом коду (у асемблеру се додаје суфикс `S` ако желимо такво понашање). Вредност PSW регистра се прослеђује на улаз контролне јединице, што омогућава условно извршавање инструкција (што је типично за инструкције условног скока). На неким архитектурама овај регистар може садржати и флегове који нису статусни, већ *контролни* – њихове вредности може постављати програмер посебним инструкцијама, чиме се контролише будуће понашање процесора. На пример, овим флеговима може се вршити избор режима рада процесора, начин контроле меморије, понашање појединачних инструкција и сл. Ово је случај са нпр. регистром RFLAGS на архитектури *x86-64*), који поред статусних, садржи и контролне флегове.

6.2.3 Фазе извршавања инструкције

Извршавање инструкције обавља се кроз алгоритам аутомата контролне јединице, а састоји се из неколико фаза са којима смо се оквирно упознали у поглављу 5.2. У овом одељку детаљније разматрамо фазе извршавања инструкције у контексту модела рачунара чије су путање података приказане на сликама 6.3, 6.4 и 6.5.

Дохватање инструкције. Фаза дохватања (енгл. *fetch*) има за циљ учитавање инструкције из оперативне меморије у процесор. Инструкција се дохвата у регистар IR, одакле постаје доступна контролној јединици која је даље тумачи и извршава. Примера ради, дохватање се може обавити на следећи начин:

- $MEM[PC] \rightarrow_{data} IR$

Дакле, врши се трансфер путем магистрале података из меморије, са адресе коју садржи програмски бројач у регистар IR. Због једноставности, овде претпостављамо да је за горњи трансфер довољан један циклус часовника. Ипак, у пракси то није увек тако. Наиме, како је меморија знатно спорија од процесора, врло често она није у стању да одговори на захтев процесора у истом циклусу. Због тога контролна јединица често мора да чека већи број циклуса док не добије сигнал од меморије да је садржај успешно постављен на магистралу података и да се може преузети. Ови циклуси се називају *циклуси чекања* (енгл. *wait cycle*). Ово важи за све трансфере између процесора и меморије, не само за дохватање инструкција (тј. важи и за дохватање и упис меморијских операнада). Други разлог због кога дохватање инструкције може захтевати више циклуса је то што поједине инструкције могу бити веома дугачке (ово је типично за CISC процесоре, код којих могу постојати различити формати инструкција, различитих дужина). Такве инструкције често садрже више битова од ширине магистрале података, те се физички не могу пренети у једном трансферу преко магистрале. Отуда се овакве инструкције дохватају део по део, у више циклуса.

Након што се инструкција пребаци у IR регистар, врши се увећање регистра PC тако да указује на следећу инструкцију у низу. Ово увећање се може извршити на више начина. Један начин је да се увећање имплементира у самом регистру. На пример, регистар PC се може имплементирати као бројачки регистар који се подсредством посебног контролног сигнала аутоматски увећава за одговарајући број бајтова (који је типично једнак неком степену двојке):

- $inc(PC)$

Ово је погодно уколико су све инструкције исте дужине, па се програмски бројач увек увећава за исту вредност. На пример, ако су све инструкције дугачке 4 бајта (као на, нпр. *ARM32* архитектури), тада је могуће део регистра PC без два најнижа бита[12] имплементирати као обичан бројачки регистар који се може увећати за један одговарајућим контролним сигналом (уз два најнижа бита која се не мењају, ово је еквивалентно увећању за четири). За овакво увећање је довољан један додатни циклус часовника.

У случају неравномерног формата инструкција, увећање није увек исто. У том случају се PC може увећати тако што се његова тренутна вредност сабере у ALU јединици са одговарајућом константом и након тога се добијени збир поново сачува у PC регистру. Сабирање са неком фиксираном константом је могуће имплементирати тако што се генерисана константа

[12] На *ARM32* архитектури, претпоставка је да су инструкције увек поравнате у меморији на адресама које су дељиве са 4. То значи да су два најнижа бита адресе инструкције (па самим тим и PC регистра) увек нуле.

од стране контролне јединице директно постави на једну од интерних магистрала за читање, чиме се доводи на улаз ALU јединице која онда врши уобичајену операцију сабирања. Сам PC регистар се у том случају може имплементирати као обичан регистар, без икаквих додатних бројачких функционалности. Број додатних циклуса часовника потребних за увећање програмског бројача у овом случају зависи од броја интерних магистрала. На пример, у случају путање података са једном интерном магистралом (слика 6.3), увећање PC регистра за константу n можемо обавити на следећи начин:

- $CONST(n) \rightarrow P$
- $PC\ [add]\ P \rightarrow A$
- $A \rightarrow PC$

при чему $CONST(n)$ значи да на магистралу постављамо вредност константе n. Ову константу генерише контролна јединица и посебним линијама је доводи на магистралу. У овом случају, инкрементација PC регистра захтева три додатна циклуса. У случају путање података са две интерне магистрале (слика 6.4), увећање програмског бројача је могуће урадити на следећи начин:

- $CONST(n) \rightarrow_A P$
- $PC\ [add]\ P \rightarrow_C PC$

Овог пута, потребна су нам два додатна циклуса. Најзад, ако користимо путању података са три интерне магистрале (слика 6.5), тада увећање програмског бројача можемо обавити у само једном додатном циклусу процесора:

- $PC\ [add]\ CONST(n) \rightarrow_C PC$

при чему се вредност регистра PC доставља ALU јединици путем магистрале A, а вредност константе n путем магистрале B.

Декодирање инструкције. Фаза декодирања (енгл. *decode*) врши тумачење битова инструкције која се налази у IR регистру и одређивање наредних корака у њеном извршавању. Из операционог кôда инструкције контролна јединица закључује о којој се инструкцији ради, који је њен формат, као и колико има операнада и које су они врсте. У случају да се утврди да је формат инструкције дугачак и да постоје делови инструкције који још увек нису учитани из меморије, приступа се њиховом учитавању. Ово је карактеристично за CISC процесоре. У таквим ситуацијама, фазе дохватања и декодирања инструкције су међусобно испреплетене. Уколико инструкција има меморијске операнде, тада се декодирањем утврђује и начин адресирања операнада, а затим се врши и израчунавање адресе. На пример, ако се испостави да инструкција користи индексно адресирање, тада је потребно сабрати вредности базног и индексног регистра, а затим тај збир пребацити у MAR регистар, ради дохватања операнда. Слично је и са другим начинима адресирања.

Број циклуса потребних за декодирање инструкције може варирати у зависности од архитектуре. На једноставним архитектурама (попут RISC архитектура), обично је један циклус довољан за декодирање операционог кода и евентуално срачунавање адресе меморијског операнда, у случају *load/store* инструкција. Код сложенијих архитектура, сложеност и разноврсност формата инструкција може захтевати више циклуса за потпуно декодирање инструкције. Такође, израчунавање адресе меморијских операнада може бити компликованије, у случају постојања сложенијих начина адресирања.

Извршавање инструкције. Фаза извршавања инструкције (енгл. *execute*) подразумева извршавање операције која одговара декодираној инструкцији. Ова операција зависи од саме инструкције, а може зависити и од испуњености услова (изражених у терминима флегова из PSW регистра), у случају условног извршавања инструкције.

Уколико инструкција има меморијски операнд, у овој фази се најпре врши дохватање операнда из меморије. На RISC архитектурама, меморијски операнд могу имати само *load/store* инструкције, док код CISC архитектура то може бити случај и са аритметичко-логичким инструкцијама. Операнд се дохвата из меморије на следећи начин:

- $MEM[MAR] \rightarrow_{data} MDR$

Дакле, вредност се из меморије са адресе која је садржана у MAR регистру пребацује путем магистрале података у регистар MDR. Као и код дохватања инструкције, овај трансфер се не може увек обавити у једном циклусу, због спорости меморије. Отуда извршење овог трансфера може подразумевати одређени број циклуса чекања.

Након тога се приступа обављању операције коју инструкција захтева:

- у случају инструкције трансфера из меморије у регистар опште намене (*load* инструкције код RISC архитектура), потребно је само учитану вредност операнда из MDR регистра пребацити у жељени регистар опште намене, што је у случају путање података са једном интерном магистралом (слика 6.3) могуће урадити на следећи начин:

  – $MDR \rightarrow R_0$

  У случају путање података са две или три интерне магистрале (слике 6.4 и 6.4), трансфер између регистара могуће је вршити само преко ALU јединице, с обзиром да су магистрале једносмерне. На пример, ако имамо две интерне магистрале, тада бисмо имали:

  – $MDR\ [no_op1]\ P \rightarrow_C R_0$

  Притом, операција no_op1 значи да се вредност првог операнда ALU јединице непромењена пропушта на излаз, а затим се преко C магистрале пребацује у жељени регистар. У случају путање података са три интерне магистрале, ситуација је скоро идентична, с тим што други операнд ALU јединице није регистар P, већ вредност B магистрале (што је, наравно, потпуно небитно, јер се други операнд ALU јединице и онако не користи).

- У случају аритметичко-логичке инструкције, врши се одговарајућа операција у ALU јединици. На пример, инструкција троадресног рачунара:

```
add R2 R0 R1
```

где је R_2 одредишни, а R_0 и R_1 изворишни операнди би могла у случају путање података са три интерне магистрале да се изврши у једном кораку на следећи начин:

  - R_0 $[add]$ $R_1 \rightarrow_C R_2$

Иста операција би, као што је раније објашњено, у случају мањег броја интерних магистрала захтевала два (са две магистрале), односно три корака (са једном магистралом). У случају CISC архитектура, аритметичко-логичке инструкције могу имати и меморијски операнд. На пример, претпоставимо да имамо инструкцију двоадресног рачунара:

```
add R0, mem
```

која сабира вредности регистра R_0 и меморијског операнда датог адресом mem и збир смешта поново у R_0. Ова операција се на путањи података са три интерне магистрале може обавити на следећи начин:

  - R_0 $[add]$ $MDR \rightarrow_C R_0$

уз претпоставку да смо претходно меморијски операнд дохватили у регистар MDR. Са друге стране, ако бисмо имали инструкцију:

```
add mem, R0
```

где је меморијски операнд уједно и одредишни, тада бисмо одговарајућу операцију могли да извршимо на следећи начин:

  - MDR $[add]$ $R_0 \rightarrow_C MDR$
  - $MDR \rightarrow_{data} MEM[MAR]$

Дакле, овде се вредност срачунава у MDR регистру, а затим се из њега пребацује у меморију на адресу која је раније, у фази декодирања, срачуната и налази се у регистру MAR.

Инструкција поређења (обично означена са `cmp` у многим архитектурама) се такође убраја у аритметичко логичке инструкције и врши одузимање, али не врши упис резултата. Самим тим, излаз ALU јединице се нигде не уписује. Једини ефекат је ажурирање флегова у PSW регистру.

- У случају инструкције трансфера из регистра опште намене у меморију (*store* инструкција на RISC архитектурама), потребно је најпре вредност регистра који се копира у меморију пребацити у MDR, а затим иницирати меморијски трансфер:

– R_0 [no_op1] $P/B \rightarrow_C MDR$
– $MDR \rightarrow_{data} MEM[MAR]$

при чему ознака P/B означава било регистар P у случају путање података са две интерне магистрале, или интерну магистралу B у случају три интерне магистрале (без икаквог значаја у случају no_op1 операције). У случају једне интерне магистрале, корак број 1 би просто пребацивао R_0 директно у MDR регистар, путем двосмерне A магистрале. У кораку број 2 поново претпостављамо да регистар MAR садржи адресу меморијског операнда која је, сходно задатом начину адресирања, срачуната у фази декодирања инструкције.

- У случају инструкције скока, извршавање може подразумевати копирање адресе на коју скачемо у регистар PC. У случају директног адресирања, сама инструкција у IR регистру садржи адресу скока, те се она може пребацити одатле у PC регистар:

  – IR [no_op1] $P/B \rightarrow_C PC$

  где P/B поново означава или регистар P или вредност B магистрале (и нема никаквог значаја у случају no_op1 операције). Напоменимо да се у регистар PC не пребацује цео регистар IR већ само његов део који садржи апсолутну адресу (ово се може постићи тако што се само одређени битови IR регистра повезују на C магистралу). У случају релативног адресирања, инструкција скока садржи релативни померај у односу на програмски бројач. У том случају, можемо извршити следећу операцију на путањи података са три интерне магистрале:

  – PC [add] $IR \rightarrow PC$

  при чему се, опет, на B магистралу доводе само битови IR регистра који садрже одговарајуће поље бинарног кôда инструкције које представља релативни померај.

  У случају да није потребно извршити скок (ако услов није испуњен), тада се не ради ништа, већ се само прелази на фазу дохватања следеће инструкције (са претходно увећане адресе у PC регистру).

6.2.4 Имплементација контролне јединице

Као што је од раније познато, контролна јединица рачунара са ускладиштеним програмом у суштини представља коначни аутомат који имплементира алгоритам интерпретације машинског језика користећи могућности путање података процесора. У зависности од сложености архитектуре и њеног машинског језика, имплементација одговарајућег коначног аутомата може бити мање или више компликована. У случају једноставних архитектура (попут RISC архитектура), коначни аутомат контролне јединице се може релативно једноставно имплементирати директно, у виду секвенцијалног кола, као што је то разматрано раније. Оваква имплементација је позната као *тврдо-ожичена* имплементација (енгл. *hardwired*).

Са друге стране, у случају сложенијих архитектура, директна реализација контролне јединице у виду секвенцијалног кола постаје сувише компликована. Због тога се прибегава *микропрограмираној* имплементацији (енгл. *microprogrammed*). Код микропрограмиране имплементације, алгоритам интерпретације машинског програма се реализује у виду посебног *микропрограма* који се чува у посебној меморији унутар контролне јединице коју називамо *контролна меморија* (енгл. *control store*). Контролна меморија се најчешће реализује као ROM меморија, тј. њен садржај је трајан и неизмењив[13]. Свака адресибилна локација ове меморије садржи једну *микроинструкцију* која описује један корак у извршењу машинске инструкције (тј. извршава се у једном циклусу процесора).

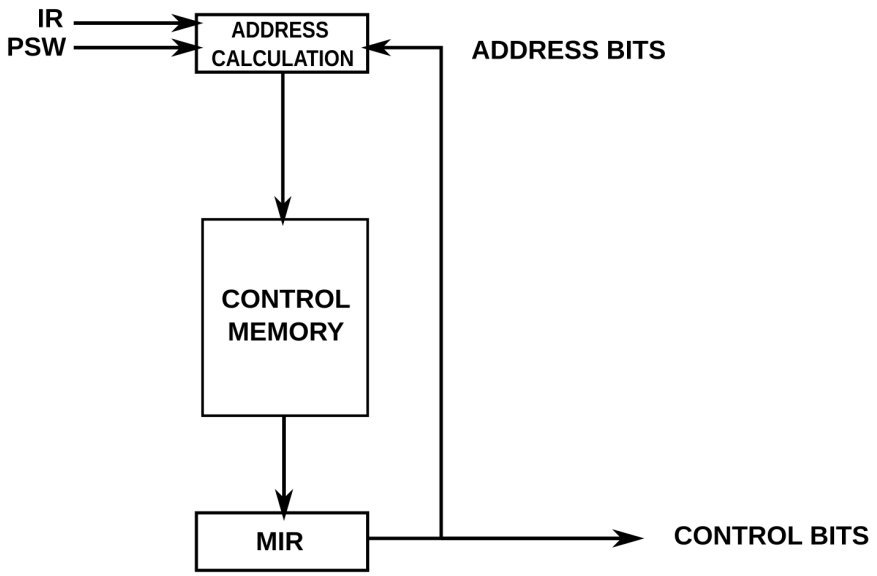
Слика 6.7: Структура микроинструкције

Микроинструкција се састоји из два дела (слика 6.7). Први део чине *контролни сигнали* који се приликом извршавање микроинструкције постављају на излаз контролне јединице и који управљају радом компоненти путање података у том циклусу. Други део микроинструкције чине *адресни битови* чија је улога да учествују у одређивању адресе следеће микроинструкције у контролној меморији која треба да буде извршена. Дакле, за разлику од машинског програма код кога се инструкције подразумевано извршавају једна за другом (осим у случају инструкције скока), у микропрограму свака микроинструкција мора да експлицитно наведе адресу наредне микроинструкције која треба да се изврши после ње. Ово пружа додатну флексибилност приликом реализације микропрограма, што је од посебног значаја, имајући у виду ограничену величину контролне меморије.

Имплементација дате архитектуре се сада своди на микропрограмирање функционалности њених инструкција, тј. описивање корака у извршењу машинских инструкција помоћу низова микроинструкција. Због тога се овакав дизајн контролне јединице често назива и *софтверски* (за разлику од тврдо-ожичене имплементације која се често назива и *хардверска*).

Пример организације микропрограмиране контролне јединице је дат на слици 6.8. Контролна меморија је, као и свака ROM меморија, комбинаторно коло које на улазу има адресу, а на излазу даје микроинструкцију на тој адреси. Добијена микроинструкција се прослеђује на улаз *микроинструкцијског регистра* (енгл. *microinstruction register* (MIR)). Вредност овог регистра се чува нпр. на силазном рубу часовника. Вредност овог регистра се на излазу дели на два дела. Контролни битови излазе ван контролне јединице и представљају њен излаз. Они управљају радом осталих компоненти процесора током трајања тог циклуса часовника. Адресни битови се, са друге стране, прослеђују комбинаторном колу за

[13] На модерним процесорима, ово не мора бити сасвим тачно, јер је могуће користити и EEPROM меморије, што омогућава да се одређени делови микропрограма накнадно надограде, у току експлоатације процесора.



Слика 6.8: Микропрограмска имплементација контролне јединице

израчунавање адресе наредне микроинструкције. У овом колу се на основу ових адресних битова, али и вредности IR и PSW регистара који су на улазу контролне јединице, израчунава адреса наредне микроинструкције и иста се прослеђује на улаз контролне меморије. Овим се на њеном излазу добија наредна микроинструкција коју треба извршити, а која ће бити уписана у MIR регистар тек на следећем силазном рубу часовника. На овај начин се вредности контролних битова на излазу одржавају стабилним током трајања циклуса (између два силазна руба), а за то време се одређује и припрема наредна микроинструкција за следећи циклус. Нагласимо да ће регистри процесора у путањи података у том случају реаговати на супротни, узлазни руб. То значи да се циклус дели на два дела:

- **негативни део** (између силазног и узлазног руба): на основу контролних сигнала текуће микроинструкције у MIR регистру, врше се одговарајући трансфери путем интерних магистрала путање података као и евентуално израчунавање у ALU јединици. На узлазном рубу се пренете/израчунате вредности чувају у одговарајућим регистрима.

- **позитивни део** (између узлазног и силазног руба): на основу вредности IR и PSW регистра (које су евентуално ажуриране на узлазном рубу) као и адресних битова текуће микроинструкције, врши се израчунавање адресе наредне микроинструкције. Ова адреса се прослеђује контролној меморији која на излазу производи наредну микроинструкцију. На силазном рубу се ова микроинструкција смешта у MIR, чиме започиње следећи циклус.

Низ микроинструкција у контролној меморији назива се *микропрограм* или *микрокôд*. Свакој машинској инструкцији обично одговара посебан микропрограм у контролној меморији који имплементира функционалност те инструкције. Такође, може постојати посебан микропрограм који врши дохватање и декодирање инструкције – овај микропрограм обично не зависи од конкретне инструкције, а његова улога је да дохвати инструкцију из меморије у IR регистар, одреди њен формат и даље евентуално учита преостале делове инструкције, израчуна адресе операнада, и тд.

Вредност PSW регистра такође може утицати на ток извршавања инструкције, јер се поједине инструкције могу извршавати на различите начине у зависности од флегова у овом регистру. Отуда је вредност овог регистра такође значајна у израчунавању адресе наредне микроинструкције.

Микропрограмирана имплементација контролне јединице омогућава да се дизајн контролне јединице сведе на програмирање (тј. микропрограмирање). Ово дизајн чини једноставнијим у случају сложених скупова инструкција. Такође, микропрограмирана контролна јединица је флексибилнија, јер се многе промене архитектуре (нпр. додавање нових инструкција, исправљање неких грешака) могу обавити ажурирањем микропрограма. Са друге стране, микропрограмирана имплементација има тенденцију да буде спорија од тврдо-ожичене имплементације. Због тога се у случају једноставних архитектура (попут RISC архитектура) обично прибегава тврдо-ожиченој имплементацији. Постоји и хибридни приступ: да се неке основне инструкције које се најчешће користе имплементирају на тврдо ожичени начин како би се брзо и ефикасно извршавале, а да постоји и контролна меморија у оквиру које је могуће кроз микропрограм имплементирати неке сложеније инструкције.

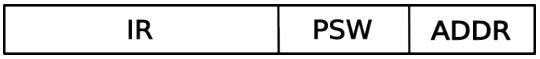
Формат микроинструкције приказан на слици 6.7 је познат и као *хоризонтални формат*. У овом формату су сви контролни сигнали представљени посебним битовима и у таквом, неизмењеном облику се прослеђују на излаз контролне јединице и даље спроводе до одговарајућих компоненти које контролишу. Овај приступ је веома једноставан, али има једну веома великуману – овакви формати микроинструкција су веома дугачки, с обзиром на огроман број контролних сигнала у модерним процесорима. Ово повећава потребан простор за чување микропрограма и самим тим повећава величину и цену контролне меморије. Алтернативни приступ је *вертикални формат*. Идеја је да се на неки начин изврши компресија хоризонталног формата, како би микроинструкција била краћа. Ово се ради тако што се препознају типичне комбинације битова које се јављају у микроинструкцијама, које се затим компактније кодирају. Овако кодиране микроинструкције се морају декодирати на излазу, помоћу декодера, како би се добили одговарајући „распаковани" контролни сигнали.

На пример, претпоставимо да у оквиру путање података имамо неких осам регистара, при чему сваки од њих има одговарајући *enable* сигнал за упис који генерише контролна јединица. У хоризонтално формату, имаћемо експлицитно задатих ових осам контролних сигнала. Са друге стране, у вертикалном формату можемо кодирати само редни број регистра у који се врши упис, за шта су нам потребна само три бита. Посебним 3-на-8 декодером можемо „распаковати" ове контролне сигнале и проследити их

одговарајућим регистрима.

Овај процес декодирања донекле успорава рад контролне јединице, али је чини знатно јефтинијом. Са друге стране, губи се на флексибилности, јер се ограничава могућност задавања произвољних комбинација контролних сигнала. У горњем примеру, вертикални формат нас ограничава да у једној микроинструкцији вредност са излазне C магистрале можемо уписати у само један од тих осам регистара. Понекад нам може бити корисно да израчунату вредност истовремено упишемо у више регистара. Ово је могуће код хоризонталног формата, јер просто можемо укључити више *enable* сигнала истовремено. Са друге стране, код вертикалног формата бисмо морали да вредност упишемо у само један од регистара, а да је касније у наредним циклусима копирамо у остале регистре. Ово продужава микропрограм и чини његово извршавање споријим.

Пример рачунања адресе микроинструкције. Претпоставимо да се микроинструкција састоји из 36 битова од којих су виша 32 бита контролни сигнали, а нижа 4 бита су адресни битови. Претпоставимо даље да је IR регистар 8-битни, као и да је PSW регистар 4-битни (флегови O,S,Z и C). Нека је контролна меморија састављена из 2^{16} 36-битних меморијских локација од којих свака може да чува једну микроинструкцију. Адреса сваке микроинструкције (која је 16-битна) се може логички поделити на три дела: виших 8 битова чине битови IR регистра, наредних 4 бита чине битови PSW регистра, а најнижих 4 чине адресни битови (слика 6.9). У том случају, коло за рачунање адресе би било веома једноставно: оно би само требало да сложи битове које добија на својим улазима на овај начин.



Слика 6.9: Једноставан начин формирања адресе микроинструкције

Последица оваквог начина рачунања адресе микроинструкције је да би контролна меморија у том случају била логички подељена на блокове од по 16 суседних локација: битови IR и PSW би одређивали редни број блока, а битови ADDR би одређивали конкретну локацију унутар блока. Сваки такав блок би одговарао микропрограму за конкретну инструкцију (одређену IR регистром) и за конкретну комбинацију флегова (одређену PSW регистром). Све што би микропрограмер требало да уради је да за сваку инструкцију и сваку комбинацију флегова напише микропрограм (не дужи од 16 микроинструкција). Свака микроинструкција би, поред контролних сигнала који се шаљу осталим компонентама процесора, морала да садржи и редни број следеће микроинструкције унутар блока (ADDR битови). Типично би почетак сваког блока био исти и одговарао би корацима дохватања инструкције. Након што се инструкција дохвати и декодира (стигне у IR регистар), тог тренутка се IR улаз у контролну јединицу мења, што за последицу има „скок” из текућег блока контролне меморије у блок који одговара управо декодираној инструкцији (при чему би микрокод у том новом блоку био извршаван не од почетка, већ од локације одређене ADDR битовима, што је управо микроинструкција

након декодирања, тј. микроинструкција којом почиње фаза извршавања инструкције). Последња микроинструкција у микрокоду сваке инструкције би у ADDR пољу садржала вредност 0000, што значи да бисмо се враћали на почетак блока и започели ново дохватање и декодирање инструкције (које би нас бацило у неки други блок контролне меморије, и тд.).

Овакав начин организације је веома једноставан, али прилично неефикасан у смислу употребе контролне меморије. Наиме, микроинструкције за дохватање и декодирање инструкције (које су увек исте) би се непотребно налазиле у сваком блоку контролне меморије. Такође, за сваку инструкцију бисмо имали 16 блокова у зависности од стања флегова, иако већина инструкција не зависи од флегова и увек се извршавају на исти начин. То значи да би код већине инструкција садржај свих 16 блокова био идентичан, што је расипање скупе контролне меморије.

Због тога се прибегава компликованијим начинима израчунавања адресе, са циљем да се микроинструкције не дуплирају, односно да постоји јединствен микропрограм за дохватање и декодирање, као и да инструкције које не зависе од флегова имају само једну копију свог микропрограма. Ово компликује коло за рачунање адреса, али зато много ефикасније користи контролну меморију, што је од изузетног значаја.

Глава 7

Оперативна меморија

Оперативна меморија (или *главна меморија*) је друга најзначајнија компонента рачунарског система, поред централног процесора. Њена улога је да чува програме који се тренутно извршавају, као и њихове податке. Пре покретања, сваки програм се мора учитати у оперативну меморију како би могао да се извршава. Оперативна меморија се често у пракси назива и *RAM меморија*, иако смо раније дискутовали због чега такав термин није најпрецизнији.

Оперативна меморија се у данашње време по правилу израђује као синхрона динамичка меморија (овакве меморије се често означавају скраћеницом SDRAM). Ово значи да се као меморијске ћелије користе кондензатори чије се пуњење и пражњење контролише транзистором. Овакав дизајн омогућава веома велике капацитете по релативно малој цени. Са друге стране, ово чини меморију знатно споријом у односу на много брже статичке меморије, засноване на резама и флип-флоповима.

Оперативне меморије су по правилу *бајт-адресибилне*, што значи да се сваки бајт може засебно адресирати, тј. адресибилне локације меморије су величине једног бајта. Другим речима, оперативне меморије су величине $n \times 8$, где је n број меморијских адреса који, самим тим, одговара броју бајтова у меморији. Отуда n одређује меморијски капацитет који се у данашње време обично изражава у гигабајтима ($1GB = 2^{30}$ бајтова).

У овој глави разматрамо употребу оперативних меморија у контексту савремених рачунара.

7.1 Меморијска магистрала

Централни процесор и оперативна меморија повезани су подсредством *меморијске магистрале*. Структура ове магистрале је разматрана у претходној глави. Подећања ради, меморијска магистрала се састоји из:

- *магистрале података* којом се машинске инструкције преносе из меморије ка процесору, као и подаци (меморијски операнди) у оба смера
- *адресна магистрала* којом се адресе преносе од процесора ка меморији

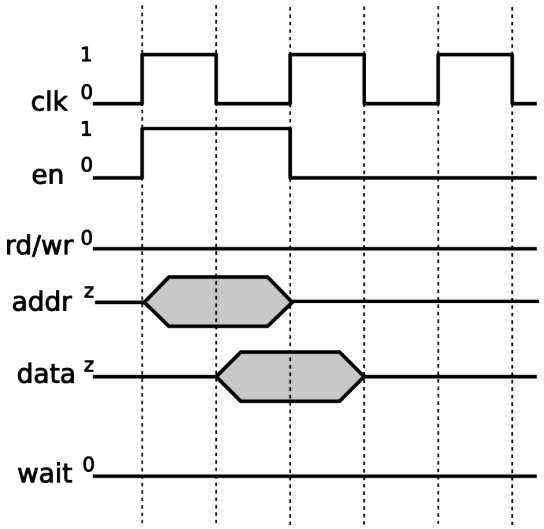
- *контролна магистрала* која се састоји из скупа контролних сигнала помоћу којих процесор и меморија размењују команде и статусне информације.

Меморијска магистрала спада у *паралелне магистрале* с обзиром да омогућава пренос вишебитних података у једном трансферу, тако што се сви битови вишебитног податка истовремено преносе путем скупа паралелних линија. Број ових паралелних линија називамо *ширина магистрале*. Ширина магистрале података одређује величину бинарне речи, тј. величину бинарно кодираног податка или инструкције који се могу пренети путем магистрале у једном трансферу. Типична ширина магистрала података на савременим рачунарима је 64 бита. То значи да се подаци величине до 64 бита могу пренети у једном трансферу путем магистрале. То укључује већину типова података у модерним програмским језицима. Слично, инструкције које нису дуже од 64 бита се могу дохватити у једном трансферу путем магистрале. Како су оперативне меморије бајт-адресибилне, пренос 64 бита преко магистрале података подразумева пренос осам бајтова који се у меморији налазе на суседним адресама, почев од бајта на адреси која је задата на адресној магистрали. Већи подаци и дуже инструкције морају се поделити на фрагменте и слати путем магистрале у више циклуса.[1] Са друге стране, ширина адресне магистрале одређује величину *адресног простора* процесора, тј. опсег адреса које процесор може да адресира. На пример, ако је адресна магистрала ширине 32 бита, тада је могуће задавати адресе у опсегу од 0 до $2^{32} - 1$, што је $4GB$ адресног простора.

Основне операције на магистрали су *читање* и *писање* појединачног податка са задате адресе. Поред тога, често су подржане и сложеније операције, попут *читања блока података* (енгл. *burst read*). Ова операција подразумева читање више суседних података у више циклуса путем магистрале. На пример, на модерним *Intel*-овим процесорима, ова операција подразумева читање осам суседна 64-битна података из меморије, почев од задате адресе (укупно 512 битова, тј. 64 бајта). На овај начин се убрзава читање, ако знамо да ће програм подацима (или инструкцијама) приступати секвенцијално, што је често случај. Убрзање долази отуд што није потребно иницирати засебне команде за сваки од података, већ се једном командом покреће читање целог блока. Такође, свака операција са меморијом подразумева одређено *кашњење* (енгл. *latency*), које представља време које је потребно да операција уопште почне са извршавањем након што је одговарајућа команда издата, а које је узроковано временом потребним да меморија приступи траженим локацијама (ово укључује време декодирања, као и време приступа одговарајућој врсти у матрици меморије). У случају читања блока података, ово кашњење ће бити присутно само на почетку, приликом читања првог податка у блоку, док ће се преостали подаци читати један за другим у суседним циклусима. Са друге стране, ако бисмо те податке читали једног по једног, у засебним операцијама на магистрали, извесно кашњење бисмо имали при сваком читању, што би повећало укупно кашњење за читање исте количине података.

[1] Сетимо се да инструкције на неким CISC процесорима, попут *x86-64*, могу бити дугачке и до 15 бајтова, што је 120 битова.

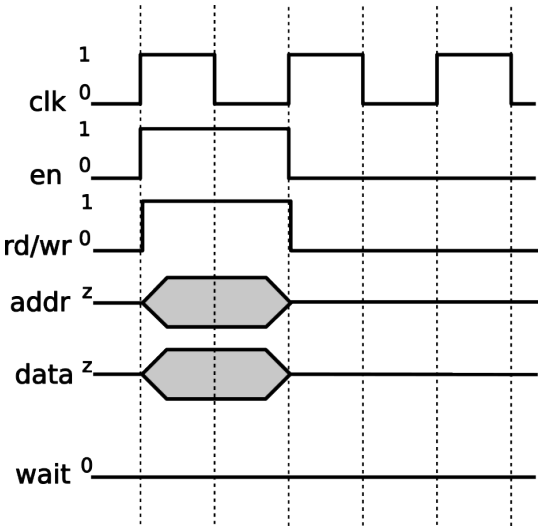
Комуникација преко магистрале остварује се у складу са одговарајућим *протоколом*, који одређује редослед размене сигнала на магистрали у циљу извршења задате операције. У наставку овог одељка, приказаћемо пример једног једноставног протокола магистрале, осмишљеног за потребе објашњења комуникације на магистрали у овом тексту. Иако протокол који ћемо описати не представља протокол неке постојеће магистрале, веома је сличан стварним протоколима, те сасвим добро осликава њихову суштину.



Слика 7.1: Временски дијаграм операције читања без циклуса чекања

На слици 7.1 приказан је пример временског дијаграма операције читања једног податка из меморије, која се обавља у складу са дефинисаним протоколом магистрале. Сигнал *clk* представља часовник – ово значи да је магистрала синхрона, те се размена сигнала између процесора и меморије усклађује са рубовима часовника. На првом узлазном рубу, процесор укључује *en* сигнал, којим се активира меморија (када је *en* једнак нули, меморија не реагује на сигнале са магистрале). Сигнал *rd/wr* одређује да ли је у питању операција читања или писања (вредност 0 означава читање, а 1 писање). На истом улазном рубу, процесор на адресну магистралу поставља вредност адресе у меморији са које се врши читање (сигнал означен са *addr*). Сива зона на временском дијаграму сигнала *addr* на слици означава да се на адресној магистрали налази активна вредност, док је изван те зоне на адресној магистрали вредност високе импедансе. Ради ефикасније комуникације, меморија је конструисана тако да вредности очитава и поставља на силазном рубу часовника. Отуда ће на првом следећем силазном рубу меморија очитати сигнале који стижу од процесора (укључен сигнал *en* којим се сигнализира захтев за операцијом, вредност *rd/wr* једнака нули, што значи да се захтева читање, као и вредност адресе на адресној магистрали). Меморија ће у складу са овим захтевом прочитати вредност са дате адресе и поставити је на магистралу података (сива зона на дијаграму *data* сигнала, који представља магистралу података). На следећем узлазном рубу, процесор ће очитати вредност са магистрале

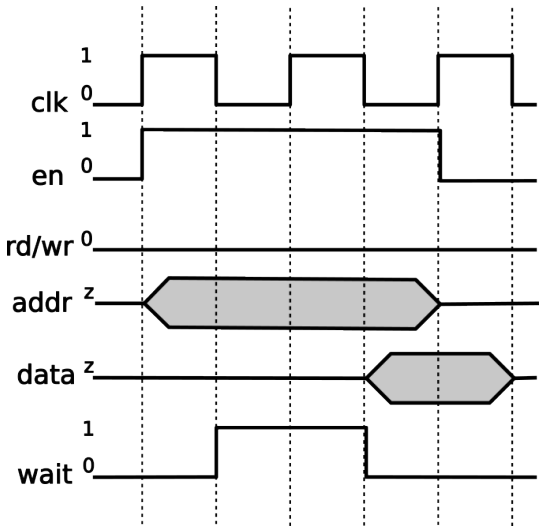
података и сачувати је у одговарајућем регистру (MDR или IR, у зависности од тога да ли чита податак или инструкцију). Истовремено ће деактивирати своје сигнале, јер је за њега операција завршена. На следећем силазном рубу, меморија ће видети да су сигнали процесора деактивирани и повући ће вредност податка са магистрале података (вредност *data* сигнала се враћа на вредност високе импедансе), чиме је операција читања завршена. Већ на следећем узлазном рубу, могуће је започети следећу операцију на магистрали. Отуда, читава операција читања према овом протоколу захтева два циклуса на магистрали.



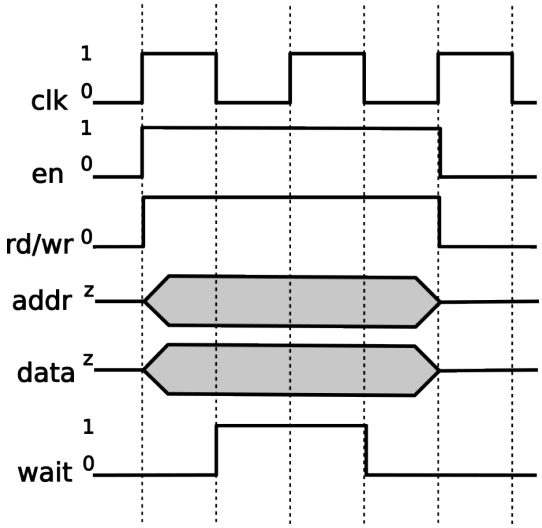
Слика 7.2: Временски дијаграм операције уписа без циклуса чекања

На слици 7.2, приказан је временски дијаграм операције уписа појединачног податка у меморију. Операција се обавља на врло сличан начин као и претходна, с том разликом што се сада сигнал *rd/wr* поставља на јединицу (операција писања), а у исто време се на магистралу поставља и адреса и податак који треба уписати (на адресну магистралу и магистралу података, респективно).

У оба претходна случаја, подразумевало се да меморија може обавити тражену операцију у једном циклусу. Отуда је процесор очекивао да већ на следећем узлазном рубу може да деактивира своје сигнале, сматрајући да је операција успешно завршена. У пракси, ово често неће бити случај, зато што меморија неће моћи тако брзо да обави тражену операцију. У том случају, меморија некако мора да сигнализира процесору да је потребно да сачека још неко време пре него што деактивира своје сигнале и настави са својим даљим радом. Ово се постиже помоћу сигнала *wait*. Када меморија постави овај сигнал на вредност 1, процесор ће знати да операција још увек није завршена, те ће чекати у „празном ходу" (тзв. *стање чекања*) док се овај сигнал не постави на вредност 0. Тек када процесор на неком узлазном рубу очита вредност 0 сигнала *wait*, моћи ће да заврши са операцијом и настави са даљим радом. Примери временских дијаграма операција читања и уписа са једним циклусом чекања дати су на сликама 7.3 и 7.4, респективно. У



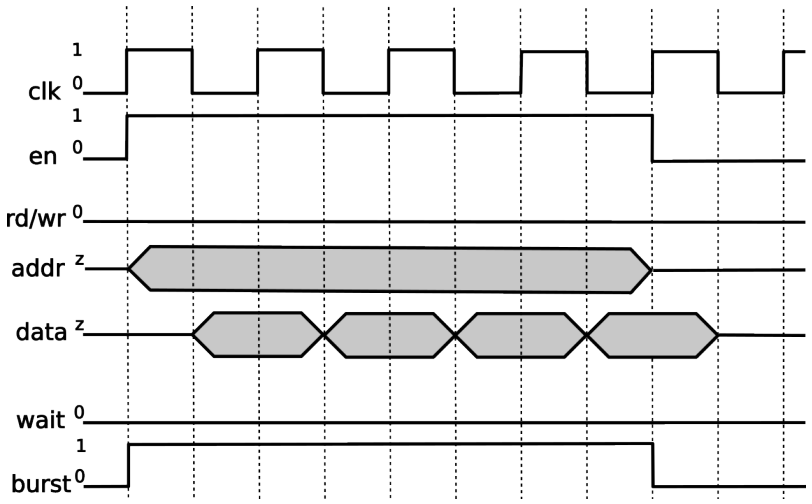
Слика 7.3: Временски дијаграм операције читања са циклусом чекања



Слика 7.4: Временски дијаграм операције уписа са циклусом чекања

пракси, број циклуса чекања може бити и већи од један.

Уколико желимо да поред читања и уписа појединачног податка омогућимо и операцију читања блока података, потребно је обезбедити још један контролни сигнал. На слици 7.5 на којој је приказан временски дијаграм операције читања блока података овај сигнал је означен са *burst*. Уколико желимо да активирамо операцију читања блока података, потребно је поставити *en* сигнал на 1 (шаљемо захтев), сигнал *rd/wr* на 0 (операција читања) и сигнал *burst* на 1 (читање блока података уместо појединачног податка). У нашем примеру, блок податка се састоји из

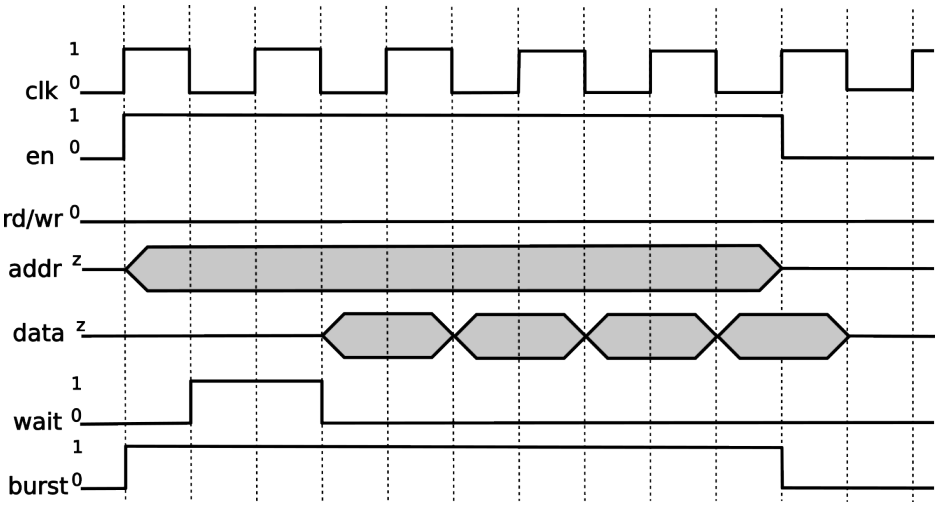


Слика 7.5: Временски дијаграм операције читања блока података без циклуса чекања

четири податка на узастопним меморијским локацијама, почев од адресе која је задата на адресној магистрали. У случају реалних магистрала, блок може садржати и већи број података (нпр. 8 на модерним *Intel*-овим архитектурама). Приметимо да је у нашем примеру за целокупну операцију трансфера четири суседна податка из меморије у процесор потребно 5 циклуса. Када бисмо, уместо операције читања блока података, применили четири узастопне операције читања појединачних података, требало би нам 8 циклуса (за сваку операцију по два). Ово илуструје ефикасност употребе операције читања блока података у пракси.

На слици 7.6 приказана је операција читања блока података са циклусом чекања. У питању је реални сценарио у коме постоји неко кашњење док меморија не почне да испоручује податке путем магистрале. Као што смо раније поменули, ово кашњење је присутно на почетку, пре него што започне трансфер података. Са друге стране, када подаци једном крену да теку магистралом, они се испоручују процесору у суседним циклусима, без икаквог додатног кашњења. У нашем примеру, са само једним циклусом чекања, за операцију трансфера четири суседна податка из меморије у процесор потребно је 6 циклуса. Када бисмо уместо тога користили четири узастопне операције читања појединачних података, са само једним циклусом чекања по операцији било би потребно чак 12 циклуса (по три за сваку операцију). У случају већег броја циклуса чекања по операцији (што је реалан сценарио у пракси), ова разлика у ефикасности постаје још драстичнија.

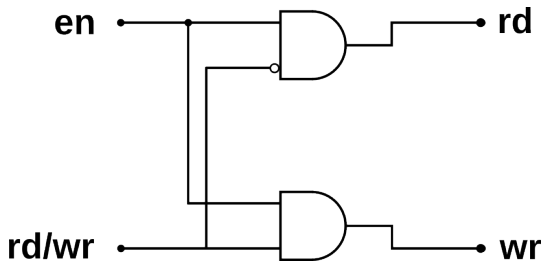
Описани протокол магистрале подразумева да се у сваком циклусу часовника може пренети само један податак преко магистрале. Модерне меморијске магистрале се обично дизајнирају тако да податке преносе на оба руба часовника. Тиме се у сваком циклусу могу пренети по два податка, чиме се дуплира брзина преноса. Наравно, за овако нешто је потребно да и меморија буде конструисана тако да може да користи оба руба часовника за



Слика 7.6: Временски дијаграм операције читања блока података са циклусом чекања

слање података. Ово је случај са тзв. DDR RAM меморијама (енгл. *double data rate* (DDR)).

Напомена о сигнализацији. У поглављу 4.5 смо обично подразумевали да меморија има два одвојена контролна сигнала *rd* и *wr* који активирају операцију читања, односно уписа. У пракси је знатно чешћа варијанта сигнализације која је коришћена у овом поглављу у опису протокола магистрале, а која подразумева један сигнал *rd/wr* за избор операције и додатни *en* сигнал за активацију меморијског кола. Предност овог приступа је то што није могуће (грешком) испоручити захтеве за читање и писање истовремено, што не би било могуће остварити, с обзиром да се иста магистрала користи и за читање и за писање. Наравно, ово нема никаквог суштинског значаја, јер се ова друга сигнализација увек може једноставно превести у ону претходну, као што је приказано на слици 7.7. У наставку текста то нећемо посебно наглашавати, тј. подразумеваћемо да је то читаоцу јасно.



Слика 7.7: Трансформација *rd/wr* и *en* сигнала у *rd* и *wr* сигнале

7.2 Ефикасна организација меморије

Као што је описано у глави 4, меморија је интерно организована као матрица ћелија које чувају појединачне битове. Типично се у једној врсти матрице налазе битови једне меморијске локације. С обзиром да су оперативне меморије адресибилне на нивоу бајта, то значи да ће свака меморијска локација садржати 8 битова. Отуда ће број колона овакве матрице бити осам. Са друге стране, број врста одговара броју меморијских локација, што ће у случају великих меморија бити огроман број (на пример, за 1GB меморије имали бисмо 2^{30} врста). Меморијске матрице са овако неравномерним димензијама (мали број колона, а огроман број врста) непрактичне су из више разлога:

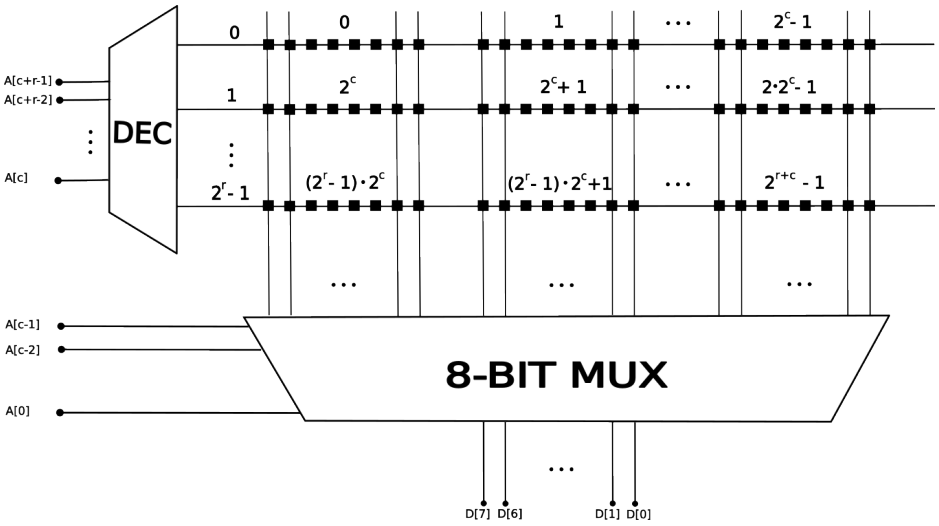
- просторно је много једноставније и ефикасније распоредити меморијске ћелије у матрицу равномернијих димензија (приближно квадратну). Квадратна организација за резултат има краћа растојања унутар матрице. Са друге стране, велики број врста подразумева и већу дужину битских линија, што успорава приступ појединачним локацијама.

- број излазних линија декодера адресе мора бити једнак броју врста матрице. Отуда, ако имамо огроман број врста у матрици, треба нам декодер огромне величине, што је веома скупо за направити. Такође, тако велики декодери по правилу имају и велико кашњење, што успорава приступ појединачним меморијским локацијама.

- приступ свакој меморијској локацији је једнако спор, па оваква организација меморијске матрице не омогућава, на пример, брз приступ суседним меморијским локацијама, што је често потребно, јер се подаци (и инструкције) углавном читају секвенцијално.

Из свих ових разлога, савремене меморије се организују тако да врсте матрице не садрже битове једне меморијске локације, већ више узастопних меморијских локација. Пример тако организоване матрице дат је на слици 7.8. У овом примеру, меморија се састоји из 2^{r+c} 8-битних меморијских локација, при чему свака од 2^r врста садржи по 2^c меморијских локација (тј. по $2^c \cdot 8$ битова). Адреса меморијске локације се сада може поделити на два дела: виших r битова (означени са $A[c+r-1]$ до $A[c]$ на слици) који одређују број врсте и нижих c битова (означени са $A[c-1]$ до $A[0]$ на слици) који одређују број 8-битне колоне у матрици (тј. конкретну 8-битну локацију у изабраној врсти). Виши део адресе се декодира помоћу r-на-2^r декодера, на основу чега се активира одговарајућа линија речи (хоризонтална линија на слици). Овим се очитава цела врста, тј. вредности свих меморијских локација из изабране врсте се појављују на битским линијама (вертикалне линије на слици). Помоћу 8-битног 2^c-на-1 мултиплексера се затим од очитаних 2^c локација бира једна, на основу нижег дела адресе.

Предност овакве организације, поред просторне ефикасности, је и јефтиније и једноставније декодирање адресе. Укупан број логичких кола у r-на-2^r декодеру и 2^c-на-1 мултиплексеру је значајно мањи него што би био у једном декодеру са 2^{r+c} излаза. Такође, када се једном декодира адреса врсте и очита читава врста (што укључује извесно кашњење), веома се брзо

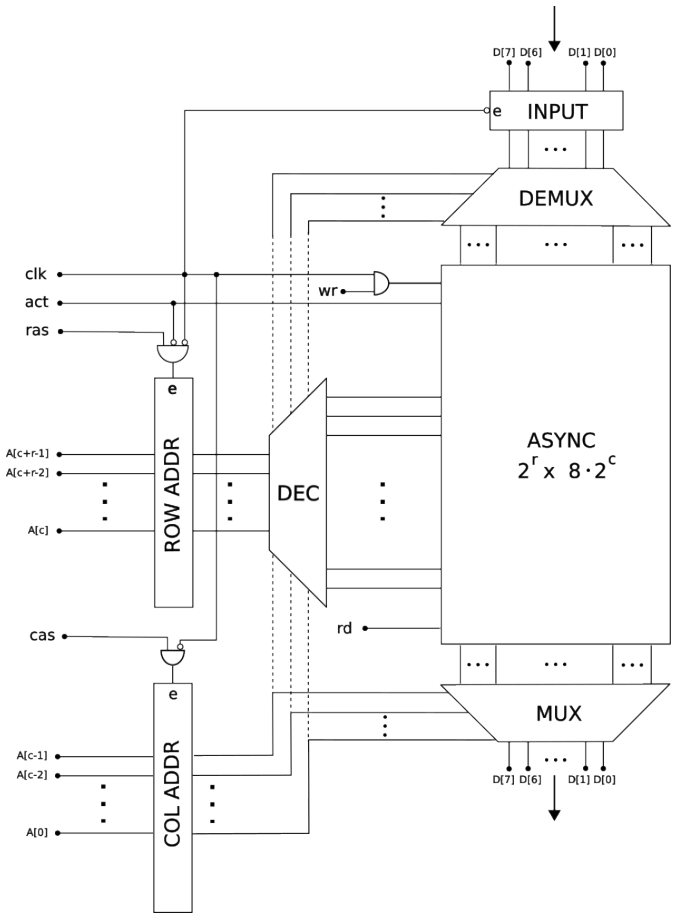


Слика 7.8: Меморијска матрица организована по врстама и колонама. Састоји се из 2^r врста, при чему свака врста садржи 2^c 8-битних меморијских локација.

могу очитавати све локације из изабране врсте. За то је само потребно одржавати виши део адресе непромењеним, а мењати нижи део адресе. Мултиплексер ће на тај начин узимати различите локације из већ очитане врсте и слати их на излаз. На овај начин се веома ефикасно могу очитавати локације на суседним адресама у меморији, што је од нарочитог значаја, нпр. у случају операције читања блока података (јер не желимо додатно кашњење приликом очитавања сваког појединачног податка).

Уобичајени начин примене описане организације је у виду синхроне динамичке меморије (SDRAM) чија је поједностављена шема приказана на слици 7.9. Структура ове меморије је веома слична оној која је приказана на слици 4.29 у одељку 4.5.4. Главна разлика је што сада користимо матрицу која у свакој врсти садржи 2^c 8-битних локација. Виши део адресе (адреса врсте) се доставља меморији у првом циклусу и чува се у одговарајућим резама (коло означено са *ROW ADDR*). Да би то било могуће, потребно је активирати посебан контролни сигнал означен са *RAS* (енгл. *row address strobe*). При наиласку узлазног руба, резе које чувају адресу врсте се закључавају, декодер активира одговарајућу излазну линију која се пропушта ка асинхроној матрици, а активира се сигнал *act* којим се покреће процес отварања врсте и врши се деструктивно читање. Овај процес може захтевати неко време, тако да се, у зависности од карактеристика меморије, обично мора сачекати одређени број циклуса пре наставка операције.

Након што се битске линије стабилизују на очитаним вредностима ћелија изабране врсте, у следећем циклусу се доставља нижи део адресе (адреса колоне), који се такође памти у одговарајућим резама (коло означено са *COL ADDR*). За ову сврху се активира контролни сигнал означен са *CAS* (енгл. *column address strobe*). Такође, у овом циклусу се, у случају операције читања, активира и *rd* сигнал. Адреса врсте омогућава мултиплексеру да



Слика 7.9: Синхрона DRAM меморија са RAS и CAS сигналима

изабере одговарајућу 8-битну вредност из отворене врсте и проследи је ка излазу. У случају операције уписа, активира се сигнал *wr* којим се омогућава промена вредности битских линија помоћу сигнала који излазе из демултиплексера. Такође, путем магистрале података доставља се 8-битни податак који треба уписати, а који се памти у улазним резама (коло означено са *INPUT*). Запамћени податак се путем демултиплексера преусмерава на одговарајућу колону асинхроне матрице, на основу задате адресе колоне. Претпоставка је да демултиплексер на излазима који нису одабрани има вредност високе импедансе. То значи да ће битске линије које нису изабране остати непромењене, док ће битске линије које одговарају изабраној колони бити постављене на вредности које долазе са улаза. На овај начин се врши упис података у одабрану меморијску локацију.

Једна од последица овакве организације меморије је и то што се виши и нижи део адресе преносе меморији у два одвојена циклуса. То значи да се исте физичке адресне линије могу користити и за виши и за нижи део адресе, што смањује неопходну ширину адресне магистрале, чинећи је јефтинијом. Прецизније, уместо да адресна магистрала има ширину $r + c$,

имаће ширину $max(r, c)$. На пример, ако је $r = c = 16$, тада је $r + c = 32$ (адресе су нам ефективно 32-битне), док је $max(r, c) = 16$, што значи да ће нам бити потребно свега 16 адресних линија на магистрали. У првом циклусу ћемо пренети 16 битова вишег дела адресе, а у наредном ћемо путем истих линија пренети 16 битова нижег дела адресе.

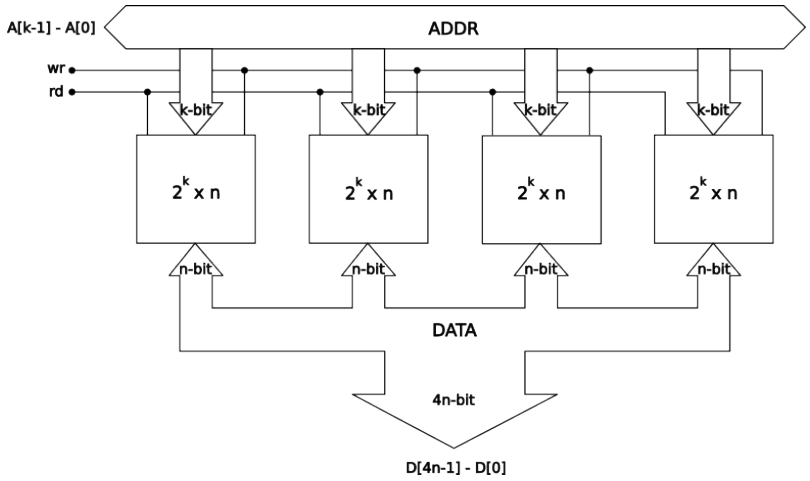
Оваква организација меморије је нарочито ефикасна у случају секвенцијалног читања података на суседним меморијским локацијама. Наиме, подаци на суседним меморијским локацијама ће по правилу имати исти виши део адресе, па ће се налазити у истој врсти матрице (изузетак је ситуација у којој податак који следи прелази у следећу врсту, али то се дешава релативно ретко, с обзиром да дужина врсте обично није тако мала). Ово значи да ћемо моћи да најпре проследимо заједничку адресу врсте у првом циклусу и отворимо врсту (што подразумева извесно иницијално кашњење). Након тога ћемо моћи да у суседним циклусима часовника читамо један по један податак из отворене врсте тако што у сваком циклусу достављамо следећу адресу колоне и активирамо *CAS* сигнал. Кашњење је у овом случају веома мало и своди се на кашњење мултиплексера и време трансфера податка преко магистрале, што је обично занемарљиво у односу на време потребно за отварање врсте. Описани механизам ефикасног читања суседних података у меморији је познат и као *FPM* режим (енгл. *fast page mode*).[2]

7.3 Вишебајтни трансфер података

С обзиром да су оперативне меморије адресибилне на нивоу појединачних бајтова, у претходном поглављу смо, једноставности ради, претпоставили да меморијско коло има осмобитни улаз и излаз за податке, те да се трансфер података врши путем осмобитне магистрале података. Ипак, као што је раније речено, у пракси су магистрале података обично знатно шире, типично 64 бита (8 бајтова). То значи да се бајтови из меморије преносе у пакетима од по 8 бајтова. Најједноставнији начин да се ово постигне је да се меморијска кола израђују тако да њихове меморијске локације буду 64-битне уместо 8-битних. Ово можемо постићи тако што у матрици свака врста садржи $64 \cdot 2^c$ битова уместо $8 \cdot 2^c$ битова, уз коришћење 64-битних (де)мултиплексера. Ипак, у пракси се чешће користи алтернативни приступ код кога се најпре конструишу меморијски блокови мање ширине (нпр 8 или 16 битова), па се онда такви блокови слажу у једну врсту да би се добила ширина од 64 битова, по принципу описаном у одељку 4.5.2. На пример можемо сложити осам 8-битна или четири 16-битна меморијска блока и тако добити 64-битне меморијске локације. Пример таквог слагања меморијских блокова дат је на слици 7.10, где су четири n-битна меморијска блока сложена у једну $4n$-битну меморију са истим бројем меморијских локација. Сигнали wr и rd (као и остали сигнали који су изостављени са слике због једноставности) се испоручују истовремено свим блоковима у врсти. Такође, свим блоковима се испоручује

[2]Заправо, термин *FPM* је коришћен раније за сличан механизам који се користио код асинхроних динамичких меморија, све до средине деведесетих година 20. века. Савремене синхроне динамичке меморије и даље користе овај механизам, али се више у литератури не користи тај назив.

иста адреса. Слагањем изабраних n-битних података на излазу се добија $4n$-битни податак.



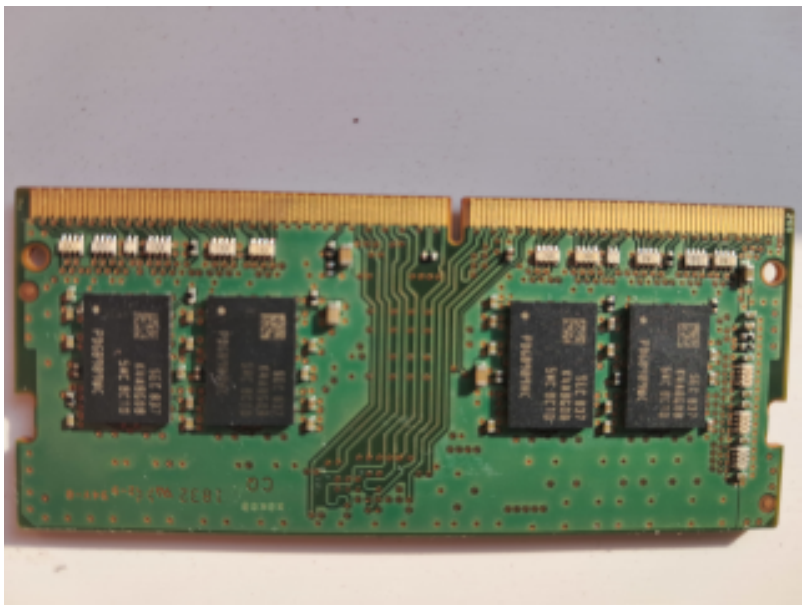
Слика 7.10: Слагање меморијских блокова ради постизања жељене ширине меморијске локације

У пракси се најчешће појединачни n-битни меморијски блокови израђују као засебни чипови, а затим се више таквих чипова истог капацитета слажу на описани начин у оквиру интегрисаног кола меморијског модула. Скуп овако организованих чипова се назива *меморијски ранг* (енгл. *memory rank*, слика 7.11).

Како бисмо задржали адресибилност на нивоу бајтова, можемо применити следећи приступ: адресу сваког бајта у меморији можемо логички поделити на три дела – адресу врсте (највиших r битова), адресу колоне (следећих c битова) и адресу бајта (најнижа три бита). Дакле, цела адреса ће бити величине $r + c + 3$ бита, а меморија ће имати 2^{r+c+3} бајтова. Притом, 8-бајтне меморијске локације у меморији ће увек почињати бајтом чија је адреса дељива са осам. На пример, нулта локација ће садржати бајтове на адресама од 0 до 7, следећа локација ће садржати бајтове на адресама од 8 до 15 и тд. Када желимо да приступимо бајту на некој адреси, примењујемо следећи поступак:

- помоћу адресе врсте отварамо одговарајућу врсту
- помоћу адресе колоне приступамо одговарајућој 8-бајтној локацији у меморији и преносимо је ка процесору путем 64-битне магистрале података
- унутар процесора, од очитаних осам бајтова бирамо онај који нам је потребан, користећи најнижа три бита адресе (адресу бајта)

Дакле, путем адресне магистрале се не шаљу најнижа три бита адресе, они се користе за издвајање жељеног бајта унутар процесора. Последица оваквог начина рада је да се подаци из меморије увек шаљу у пакетима од



Слика 7.11: Меморијски модул са скупом чипова који чине један ранг

по осам бајтова који су поравнати на адреси која је дељива са осам. Овакав пакет називаћемо *меморијска реч* (енгл. *memory word*).

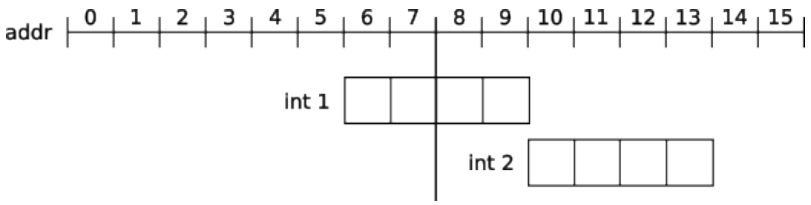
7.4 Меморијско поравнање

Опште је познато да многи типови података у стандардним програмским језицима нису једнобајтни, већ су вишебајтни. На пример, стандардни целобројни тип (`int` тип у *C*-оликим језицима) је 32-битни, тј. заузима четири суседна бајта у меморији. Уколико је потребно очитати један 32-битни податак из меморије, могу наступити два сценарија:

- сва четири бајта тог податка се налазе у оквиру исте меморијске речи. На пример, претпоставимо да желимо да очитамо 32-битни целобројни податак са адресе 10. То значи да он окупира бајтове на адресама 10, 11, 12 и 13. Сетимо се да нулта меморијска реч садржи бајтове са адресама од 0 до 7, а да следећа реч садржи бајтове са адресама од 8 до 15. Очигледно, ова друга реч садржи све бајтове нашег целобројног податка, па ће након што тих осам бајтова буду пребачени у процесор, бити једноставно да се из њих издвоје ових четири који нас занимају.

- четири бајта тог податка су расподељени у две суседне речи у меморији. На пример, претпоставимо да је 32-битни целобројни податак који желимо да прочитамо на адреси 6. То значи да он окупира бајтове на адресама 6, 7, 8 и 9. Сада бајтови на адресама 6 и 7 припадају речи која садржи бајтове са адресама од 0 до 7, док бајтови на адресама 8 и 9 припадају речи која садржи бајтове на адресама од 8 до 15. У овом сценарију ће процесор морати да прочита

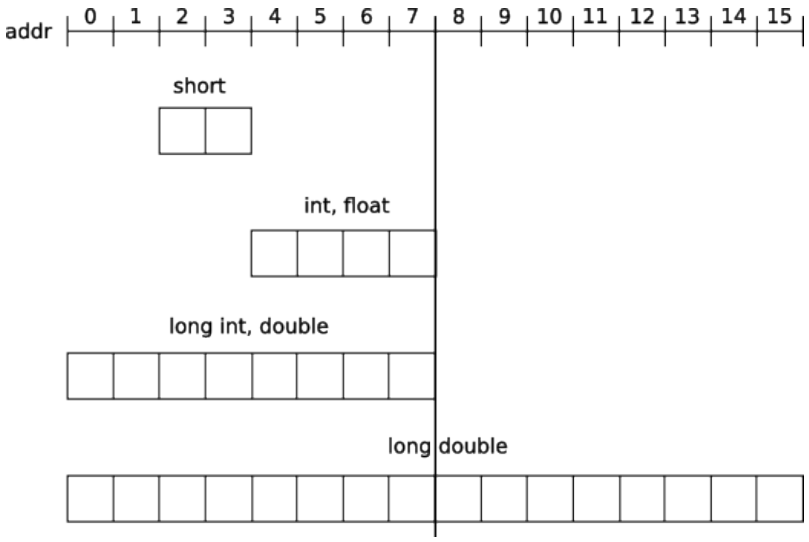
обе ове меморијске речи (у два трансфера преко 64-битне магистрале података) и да након тога интерно сложи тражени 4-бајтни податак.



Слика 7.12: Приступ 4-бајтном целобројном податку који захтева два трансфера (*int 1*) и један трансфер (*int 2*) преко магистрале

Ова два сценарија приказана су на слици 7.12. Јасно је да је овај други сценарио неповољнији и захтева више времена за учитавање истог податка. Он је последица организације меморије и чињенице да се подаци из меморије увек физички читају у виду 64-битних речи чије су адресе дељиве са осам. Овај проблем се обично решава применом адекватног *меморијског поравнања* (енгл. *memory alignment*). Ово значи да се уводе правила која захтевају да подаци одређеног типа у меморији увек морају бити поравнати на адреси која је дељива одговарајућим степеном двојке. На пример, за податак величине два бајта (попут `short` типа у *C*-оликим језицима) довољно је захтевати да увек буде на адреси која је дељива са два. Заиста, на тај начин ћемо избећи сценарио у коме је један бајт у једној речи, а други у другој. За податке величине четири бајта (попут типова `int` или `float`), поравнање мора бити четири, тј. подаци ових типова морају увек бити на адресама које су дељиве са четири. Тиме се гарантује да ће сваки податак увек у целини бити у оквиру једне 8-бајтне речи. Најзад, за осмобајтне (64-битне) податке (попут типа `double` или `long`), захтев је да имају поравнање осам, тј. да увек буду на адреси која је дељива са осам. Овим ће сваки овакав податак увек заузимати једну целу 8-бајтну реч у меморији и моћи ће да буде очитан у једном трансферу на магистрали. Са друге стране, подаци величине 16 бајтова (попут типа *long double* у *C*-у) не морају имати поравнање 16, већ је довољно поравнање 8. Заиста, на тај начин ћемо обезбедити да овакав податак увек заузима неке две суседне речи у меморији, те да се пренос врши у два циклуса путем магистрале, што је оптимално за овако велике податке. Наведена поравнања типова приказана су на слици 7.13.

Описана политика поравнања може понекад бити наметнута самим хардвером. То значи да сам процесор захтева да подаци одређеног типа (тачније, операнди одређених инструкција) буду поравнати на одговарајући начин. Ако то није случај, процесор избацује *изузетак*, тј. прекида рад програма и предаје контролу оперативном систему (о овоме ће бити више речи касније). Ово значи да програмер мора да обезбеди одговарајуће поравнање података у својим програмима, у супротном програми просто неће радити. Овакав приступ се користи код, нпр. архитектуре *ARM32*. Са друге стране, постоје архитектуре које допуштају да вишебајтни подаци не буду поравнати адекватно – у том случају ће процесор сам захтевати више



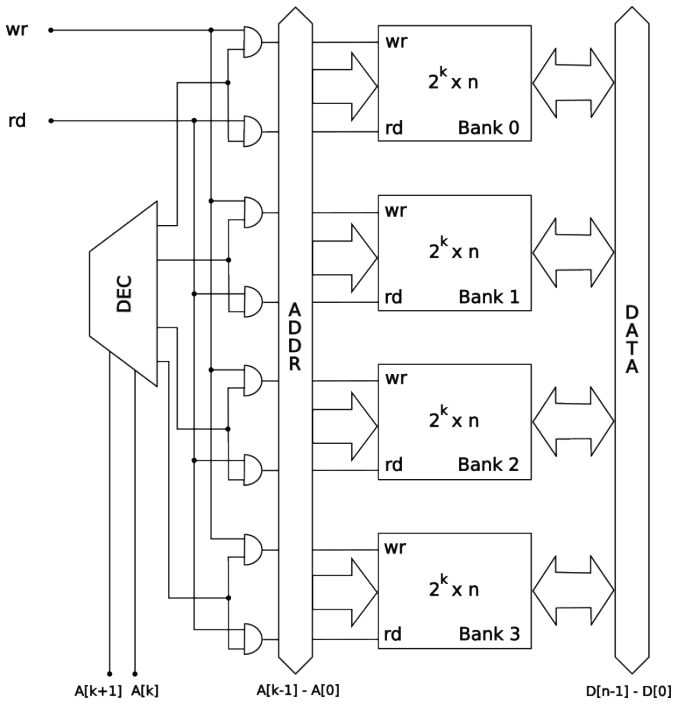
Слика 7.13: Поравнања стандардних типова података у језику *C*

узастопних трансфера преко магистрале и обавити „склапање" траженог податка након што сви делови буду учитани (цео овај поступак ће бити потпуно невидљив за програмера). Овакав је случај са архитектурама *x86* и *x86-64*. Ово пружа програмеру додатну удобност, јер његови програми неће бити прекинути ако није адекватно поравнао своје податке у меморији, већ се се само извршавати спорије. Ипак, како спорије извршавање није оно што желимо у пракси, веома је важно да водимо рачуна о поравнању приликом писања програма у асемблеру. Са друге стране, с обзиром да се у модерно време програми обично не пишу у асемблеру већ у неком вишем програмском језику, програмер најчешће не мора да води рачуна о поравнању, већ о томе брине преводилац.[3]

7.5 Испреплетени приступ

Као што је описано у одељку 4.5.2, веће меморије се могу једноставно конструисати од мањих. Ово можемо постићи тако што више меморијских блокова исте ширине сложимо у колону, као што је приказано на слици 7.14. У овом примеру, четири меморијска блока ширине n са k-битним адресама (тј. величине $2^k \times n$) се слажу у једну меморију која има 2^{k+2} локација ширине n. Адреса је сада ширине $k+2$, при чему се *више* два бита користе за избор меморијског блока (помоћу посебног декодера), док се *нижих* k битова испоручују меморијским блоковима за одабир одговарајуће локације унутар блока.

[3]Поједини програмски језици вишег нивоа имају могућност експлицитног задавања жељеног поравнања за неки податак или тип података, чиме се мења подразумевано понашање преводиоца. На пример, у језику *C++*, постоји кључна реч `alignas` којом се може задати жељено поравнање декларисаног податка. Задато поравнање мора бити степен двојке и не сме бити мање од подразумеваног.



Слика 7.14: Слагање меморијских блокова без преплитања

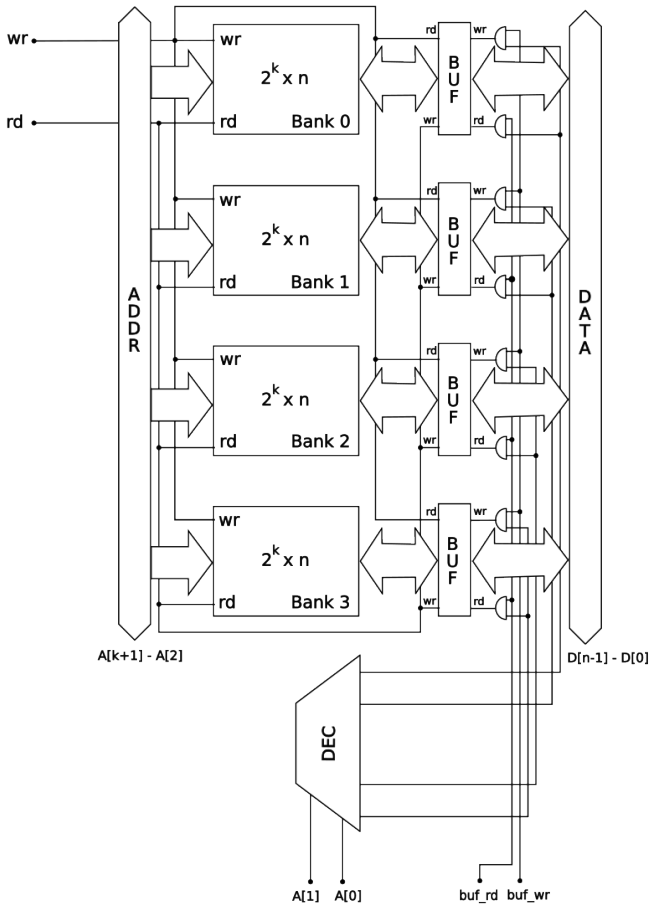
Последица оваквог дизајна је да се у резултујућој меморији првих 2^k локација налази у нултом меморијском блоку, следећих 2^k у следећем и тд. С обзиром да се подаци и инструкције из меморије обично читају секвенцијално, то значи да ће приликом извршавања једног програма процесор углавном приступати једном те истом меморијском блоку. Узастопни приступи истом меморијском блоку могу подразумевати извесно кашњење при сваком приступу док се одговарајућа операција припреми у меморијском колу. Једноставна идеја која пада на памет је да би можда било ефикасније ако бисмо подацима приступали наизменично из различитих блокова. Тада бисмо, док читамо податке из једног блока, могли истовремено да иницирамо припрему читања следећег податка из другог блока. Када завршимо са читањем из првог блока, прелазимо на следећи блок у коме је припрема већ завршена и податак се може прочитати одмах. На тај начин, кашњења различитих меморијских блокова се у доброј мери преклапају, што свеукупно чини приступ меморији ефикаснијим.

Међутим, да би овај приступ могао да се примени, потребно је обезбедити да суседне меморијске локације буду у различитим блоковима, с обзиром да подацима обично приступамо секвенцијално. На пример, ако приступамо елементима низа који се налазе на адресама $8, 9, 10$ и 11, природно је да ћемо им приступати редом, у некој петљи, и обрађивати их једног по једног. Да бисмо могли да применимо претходни приступ, неопходно је да адресе $8, 9, 10$ и 11 буду у различитим блоковима. Да бисмо ово постигли, неопходно је да променимо начин на који су меморијски

блокови повезани на адресне линије.

Пример таквог другачијег начина повезивања дат је на слици 7.15. Кључна разлика у односу на претходну слику је у томе што се сада два *најнижа* бита адресе користе за избор модула, док се виших k битова користе за избор локације у оквиру изабраног модула. Ефекат је да ће све адресе које се у бинарном запису завршавају са 00 (тј. дељиве са 4) бити у нултом блоку, адресе које се завршавају са 01 (тј. оне које дају остатак 1 при дељењу са 4) ће бити у блоку број 1, и тд. У нашем примеру са елементима низа на адресама $8, 9, 10$ и 11, елемент на адреси 8 ће бити у нултом блоку, елемент на адреси 9 у првом, елемент на адреси 10 у другом, а елемент на адреси 11 у трећем блоку. Приступањем овим елементима редом (што је уобичајено у нашим програмима) ми ћемо заправо наизменично приступати различитим блоковима. Ово ће блоковима дати времена да се припреме за следећу операцију. Овакав начин организације меморије је познат и као *испреплетени приступ* (енгл. *interleaved memory access*). У контексту испреплетеног приступа, меморијски блокови који се преплићу се обично називају *банке* (енгл. *memory bank*).

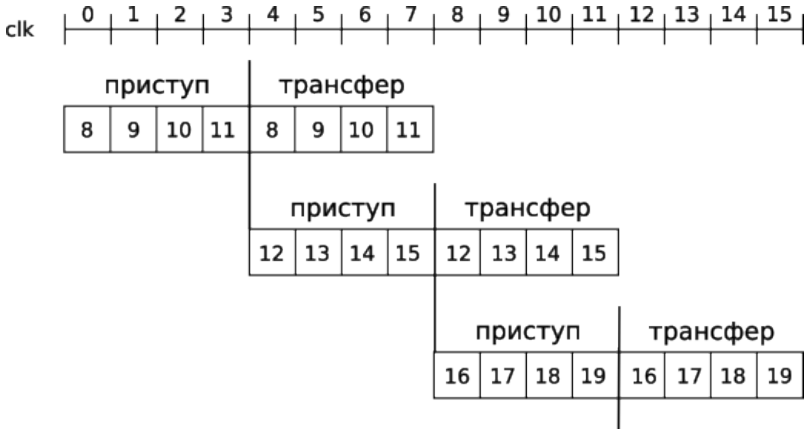
Разјаснимо детаљније начин рада меморије дате на слици 7.15. Претпоставимо да желимо да прочитамо податке на адресама $8, 9, 10, 11, 12, 13, 14$ и 15, тим редом. Једноставности ради, претпоставимо да су адресе 8-битне, тј. да је укупни капацитет меморије 256 n-битних локација, при чему свака банка садржи по 64 локације. У бинарном запису, 8-битне адресе $8, 9, 10$ и 11 се записују као 00001000, 00001001, 00001010 і 00001011. На основу два најнижа бита, закључујемо да се ове адресе редом налазе у банкама $0, 1, 2$ и 3. Приликом приступа, најпре се виши део адресе 000010 (без два најнижа бита) који је заједнички за све ове четири адресе предаје свим банкама путем адресне магистрале и иницира се очитавање података (путем *act*, *ras* и *cas* сигнала који нису уцртани на слици, због једноставности). Ово очитавање подразумева неко иницијално кашњење. На примеру временског дијаграма приказаном на слици 7.16, ово кашњење износи четири циклуса. Након што прође иницијално кашњење (које је исто за све банке и у потпуности је временски преклопљено), може се задати *rd* сигнал којим ће се ова четири податка наћи на излазима банки. Исти сигнал активира упис у излазне бафере (регистри означени са *BUF*) чиме се прочитани подаци памте у баферима. Након тога се у наредна четири циклуса могу задавати два нижа бита адреса ових података редом (00, 01, 10, 11) на улаз декодера који врши избор једног од четири бафера. Истовремено се активира *buf_rd* сигнал који омогућава изабраном баферу да испусти своју вредност на магистралу података. На тај начин ће се у четири наредна циклуса (циклуси од 4 до 7 на слици 7.16) подаци са адреса $8, 9, 10$ и 11 редом послати путем магистрале података. Даље, потребно је прочитати следећа четири податка, са адреса $12, 13, 14$ и 15. Овде следи кључна оптимизација коју испреплетени приступ омогућава – чим смо податке са адреса $8, 9, 10$ и 11 сместили у бафере (на почетку циклуса број 4), можемо одмах на адресну магистралу поставити виши део који је заједнички за адресе $12, 13, 14$ и 15. Ове адресе се у бинарном облику записују као 000001100, 00001101, 00001110 и 00001111. Дакле, заједничих виших 6 битова су 000011. Постављањем ове вредности на адресну магистралу долази до припреме и читања ових података из свих банки истовремено. Овај процес се обавља паралелно са пребацивањем



Слика 7.15: Слагање меморијских блокова са преплитањем

претходно очитаних података са адреса $8, 9, 10$ и 11 из бафера путем магистрале података (слика 7.16, циклуси од 4 до 7). Када се пребацивање ових података заврши, очитани подаци из меморијских банки са адреса $12, 13, 14$ и 15 ће бити спремни и могу се пребацити у бафере (на почетку циклуса број 8) и затим пребацивати путем магистрале података у наредна четири циклуса (циклуси од 8 до 11), без додатног кашњења.

Кључна претпоставка овде је да кашњење приликом очитавања података из банки није веће од четири циклуса. То значи да ће, када подаци $8, 9, 10$ и 11 буду пребачени путем магистрале података, одмах бити могуће да у бафере учитамо следећих четири података (са адреса $12, 13, 14$ и 15) и започнемо њихово пребацивање. Уколико би било потребно да даље читамо још четири следећа податка из меморије (са адреса $16, 17, 18$ и 19), процес њиховог очитавања из банки би могао да се врши паралелно са пребацивањем података са адреса $12, 13, 14$ и 15 из бафера преко магистрале података (циклуси од 8 до 11), како би када се тај трансфер заврши (на почетку циклуса број 12) подаци са адреса $16, 17, 18$ и 19 могли одмах



Слика 7.16: Временски дијаграм читања у испреплетеном приступу

да се прихвате у бафере и започне њихово пребацивање, без додатног кашњења. Као што се види на временском дијаграму на слици 7.16, једино кашњење које би под том претпоставком постојало је иницијално кашњење при очитавању података са адреса $8, 9, 10$ и 11 (четири циклуса). Након тога би се даље читање података обављало у суседним циклусима, без икаквог застоја. У пракси се број банки које се преплићу прилагођава кашњењу меморије и циљаној фреквенцији рада меморије, како би се обезбедило да у описаном процесу читања не постоји застој. Број банки је увек неки степен двојке (нпр. 4, 8 или 16).

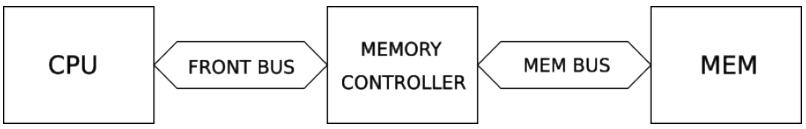
Испреплетени приступ се може користити и приликом операције писања. На пример, ако желимо да упишемо редом податке на адресе $8, 9, 10, 11, 12, 13, 14$ и 15, најпре бисмо виши део адресе заједнички за прва четири податка (000010) поставили на адресну магистралу и иницирали бисмо приступ тим подацима. Док тај процес траје (а он може подразумевати извесно кашњење за које претпостављамо да је мање од 4 циклуса), ми бисмо жељене податке за упис путем магистрале података испоручивали баферима (користећи ниже делове адреса 00, 01, 10 и 11, редом, у четири циклуса) који би их памтили (активирајући у ту сврху buf_wr сигнал). Након што се приступ подацима у банкама заврши, подаци се преузимају из бафера и иницира се операција уписа (задавањем wr сигнала). Затим се на адресну магистралу поставља заједничи део адресе следећа четири податка (000011) и иницира се приступ адресама $12, 13, 14$ и 15. Истовремено се, док овај процес траје, путем магистрале података нови подаци за упис испоручују у бафере путем магистрале података, настављајући тако описани процес на исти начин, без додатног кашњења.

Испреплетени приступ је, уз раније описани FPM режим који постоји унутар појединачних меморијских блокова, један од основних начина за повећање ефикасности при секвенцијалном приступу меморији. Преплитање се користи унутар појединачних меморијских чипова, тј. сваки меморијски чип по правилу у себи има већи број банки које су међусобно испреплетене. Такође, преплитање се може користити и између различитих рангова у оквиру истог или различитих меморијских модула.

Напоменимо да преплитање побољшава ефикасност само у случају секвенцијалног приступа. Уколико су адресе којима приступамо „разбацане" унутар меморије, испреплетени приступ неће бити од велике користи, јер се он заснива на идеји да унапред читамо податке са суседних меморијских локација. Ипак, како је секвенцијални приступ подацима и инструкцијама у нашим програмима правило, а не изузетак, у пракси је корист од испреплетеног приступа веома велика.

7.6 Контролер меморије

У претходним поглављима приказали смо детаљно различите принципе организације меморије. Видели смо на који начин су организоване меморијске матрице, као и како се брзина приступа може побољшати помоћу преплитања. Све ово је знатно закомпликовало саму организацију меморије, као и управљање меморијом и комуникацију са њом. Како терет тог компликованог управљања не би пао на централни процесор, модерни рачунари обично укључују посебну компоненту која преузима ту улогу. Ова компонента се назива *меморијски контролер* (енгл. *memory controller*). У питању је секвенцијално коло чија је улога да прихвата захтеве за меморијским операцијама од процесора и да те захтеве прослеђује меморијским модулима на одговарајући начин, водећи рачуна о ефикасности као и о ограничењима које намећу специфичности меморијских кола.



Слика 7.17: Меморијски контролер

Поједностављена шема рачунарског система који укључује меморијски контролер дата је на слици 7.17. Видимо да се меморијски контролер сада налази између процесора и оперативне меморије. Процесор своје захтеве за читањем и писањем испоручује меморијском контролеру путем магистрале која функционише по протоколу попут оног који смо описали у поглављу 7.1. У овом разматрању, ту магистралу ћемо називати *предња магистрала* (енгл. *front bus*). Меморијски контролер прихвата те захтеве и обезбеђује њихово извршавање. Контролер је са меморијом повезан другом магистралом коју ћемо у овом контексту називати *меморијском магистралом* (на слици означена са *mem bus*). Путем ове магистрале контролер испоручује захтеве оперативној меморији, користећи адекватну сигнализацију коју смо описали у претходним поглављима. Протокол комуникације путем ове магистрале је по правилу знатно компликованији, јер он укључује управљање функционалностима меморије ниског нивоа, попут отварања врсте, слања вишег и нижег дела адресе, испреплетеног приступа, освежавање динамичких ћелија и сл. Све ове функционалности су скривене од процесора, како он њима не би морао да се бави. За њихову адекватну реализацију одговоран је меморијски контролер. Он познаје

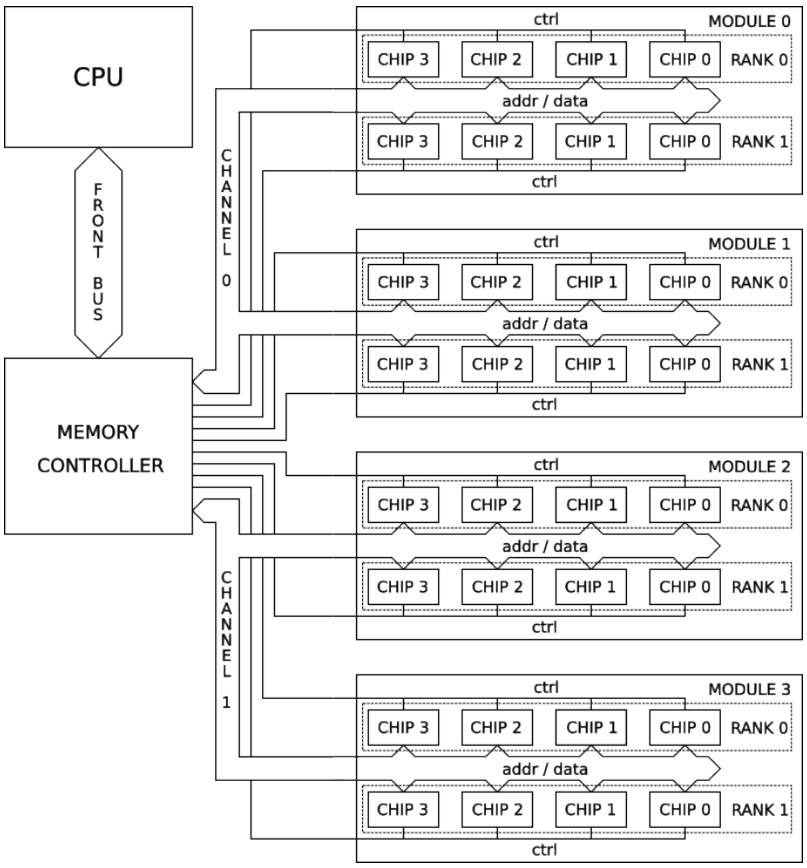
спецификације меморија и њихово понашање (укључујући и временска ограничења, кашњења и сл.) и у складу са тим распоређује одговарајуће контролне сигнале у времену. Протокол којим се врши комуникација између меморијског контролера и меморијских модула путем меморијске магистрале дефинисан је *JEDEC* стандардом (енгл. *Joint Electron Device Engineering Council*). Овај стандард дефинише све операције ниског нивоа које меморијски модули типично подржавају.

Посао меморијског контролера је додатно закомпликован чињеницом да оперативна меморија није монолитна, већ обично има прилично сложену архитектуру. На слици 7.18 приказан је пример архитектуре оперативне меморије на модерним рачунарима. Основне јединице оперативне меморије су меморијски чипови који су обично ширине 8 или 16 битова, а који у себи садрже већи број испреплетених меморијских блокова (банки). Ови чипови се слажу у рангове ради постизања жељене ширине меморијске речи (типично 64 бита). Физички, чипови који чине ранг се монтирају на интегрисано коло које се назива *меморијски модул* (енгл. *memory module*). Сваки модул може садржати један или више рангова (на слици је дат пример модула са два ранга). Модули се монтирају на матичну плочу рачунара помоћу прикључака које називамо *меморијски слотови* (енгл. *memory slot*). Типично, матичне плоче садрже два или четири меморијска слота који не морају сви бити искоришћени. Меморијски слотови су обично са контролером повезани једном истом меморијском магистралом. Ово значи да ће контролер у сваком тренутку моћи да комуницира само са једним модулом (тачније, једним рангом) путем ове магистрале. Обично се над ранговима такође врши испреплетени приступ, ради повећања ефикасности. У новије време, постоје меморијски контролери који имају више инстанци меморијских магистрала које називамо *канали* (енгл. *channels*). Уколико контролер подржава два канала, тада такав систем називамо *двоканални* (енгл. *dual channel*). На слици 7.18 је приказан пример двоканалног система. У овом примеру, два меморијска слота су повезана са контролером путем једног канала, а друга два меморијска слота путем другог канала. Основни бенефит оваквог повезивања је то што је могуће и физички размењивати податке са два меморијска модула истовремено, што може значајно убрзати трансфер података.[4] Постоје и четвороканални системи (енгл. *quad channel*). Канали у себи садрже линије за податке и адресне линије и оне су дељене између свих модула и рангова који се повезују путем тог канала. Са друге стране, сваки ранг има засебне контролне сигнале и може се са њим независно оперисати (сигнали означени са *ctrl* на слици 7.18).[5]

С обзиром на описану сложену архитектуру меморије, меморијски контролер мора да обезбеди адекватно управљање свим компонентама меморије. У наставку детаљније описујемо главне функционалности меморијског контролера, имајући у виду све што смо до сада навели.

[4] Приликом монтирања меморијских модула, треба водити рачуна да буду прикључени на слотове који су повезани на различите канале, како би се ова могућност искористила. Обично су слотови који су повезани на различите канале обојени различитим бојама.

[5] Заправо, у пракси обично сви рангови и модули имају заједничке контролне сигнале, али сваки ранг има свој засебан *enable* сигнал којим се омогућава дејство осталих контролних сигнала на тај ранг. На овај начин ће само селектовани ранг реаговати на контролне сигнале, док ће их остали игнорисати.



Слика 7.18: Архитектура оперативне меморије

Иницијализација и препознавање меморијских модула. С обзиром да не морају сви меморијски слотови бити попуњени, меморијски контролер мора по укључењу рачунара да утврди који су слотови заузети, као и да одреди капацитете меморијских модула који су инсталирани. На основу ових информација се меморијски контролер интерно конфигурише и припрема за даљи рад. Интерна конфигурација контролера врши се постављањем вредности посебних конфигурационих регистара унутар меморијског контролера.

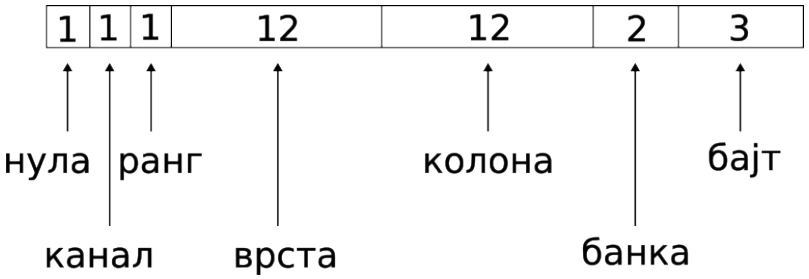
Мапирање меморијских адреса. Меморијски контролер расподељује адресни простор процесора на појединачне меморијске модуле и обезбеђује транслацију адреса које процесор шаље путем предње магистрале у адресе које се испоручују меморијским модулима. Наиме, процесор свој адресни простор посматра као целину, тј. као континуални низ бајтова. На пример, ако је адреса 32-битна, то значи да процесор „види" адресни простор од $4GB$ (2^{32} бајтова). Процесор контролеру шаље адресу 64-битне меморијске речи коју жели да прочита или упише (у питању је 29-битна адреса која настаје уклањањем три најнижа бита из адресе бајта). Меморијски контролер ову

адресу мора да мапира у следеће компоненте:

- број канала (у случају вишеканалних система)
- број модула и ранга
- број банке у чипу
- адреса врсте
- адреса колоне

На пример, претпоставимо да имамо инсталирана два меморијска модула, при чему сваки садржи по два ранга, где сваки ранг има капацитет од $512MB$ (2^{29} бајтова). Претпоставимо да је систем двоканални, као и да су на сваки од канала повезана два меморијска слота, од којих је један попуњен, а други је празан. Укупан капацитет инсталиране меморије је $2GB$ (2^{31} бајтова). Ово значи да је само половина теоријског адресног простора процесора искоришћено, те да адресе бајтова које процесор може да адресира иду од 0 до $2^{31} - 1$.

Претпоставка је да се сваки ранг састоји из четири 16-битна чипа, са укупно 2^{26} локација. Унутар сваког чипа постоје по четири банке у испреплетеном режиму приступа. То значи да се нижа два бита 26-битне адресе ових локација користе за избор банке, док су виша 24 бита подељена на виши 12-битни део који представља адресу врсте и нижи 12-битни део који представља адресу колоне. У таквој конфигурацији, меморијски контролер може примењивати мапирање адреса приказано на слици 7.19.



Слика 7.19: Пример мапирања адресе у меморијском контролеру

Адреса приказана на слици 7.19 је 32-битна. Као што је раније речено, виших 29 бита представљају адресу меморијске речи, док нижих 3 бита представљају адресу бајта у оквиру 8-бајтне речи. Само виших 29 бита се шаљу меморијском контролеру путем адресних линија у оквиру предње магистрале. У нашем примеру, највиши бит ове адресе ће увек бити нула, с обзиром да имамо само $2GB$ инсталиране меморије. Наредна два бита одређују канал и ранг који је потребно активирати помоћу контролних сигнала. Најпре се селектованом рангу шаље адреса врсте (наредних 12 битова у адреси) и иницира се отварање врсте. Након тога се шаље адреса колоне (наредних 12 битова) којом се бира колона у оквиру чипова у изабраном рангу. Приступа се одговарајућим локацијама у свим банкама

истовремено, што омогућава испреплетени приступ. Најнижа 2 бита у адреси меморијске речи користе се као редни број банке.

Нагласимо да се конкретно меморијско мапирање одређује динамички, на основу инсталиране меморије. Овај процес се врши по укључивању рачунара.

Оптимизација приступа меморији. Меморија омогућава различите видове оптимизације приступа као што су FPM режим и испреплетени приступ. Међутим, ови механизми пре свега имају ефекта када се меморијским локацијама приступа редом у адресном простору процесора. Меморијски контролер мора да препознаје ове обрасце приступа и да у таквим ситуацијама ефикасно користи одговарајуће механизме. На пример, меморијски контролер треба да води евиденцију о томе које су врсте тренутно отворене у ком меморијском рангу. На овај начин ће контролер моћи да препозна да ли се следећи захтев процесора односи на податак унутар отворене врсте и да, ако је одговор потврдан, прескочи скупи процес отварања врсте и само избором колоне приступи већ очитаном податку. Слична је ситуација са испреплетеним приступом. Меморијски контролер мора бити свестан који су подаци већ очитани у баферима банки како би могао да их ефикасно проследи процесору уколико се они траже у следећем захтеву. Такође, меморијски контролер ће типично унапред иницирати очитавање следећих података из банки, антиципирајући њихово коришћење у наредним захтевима процесора.

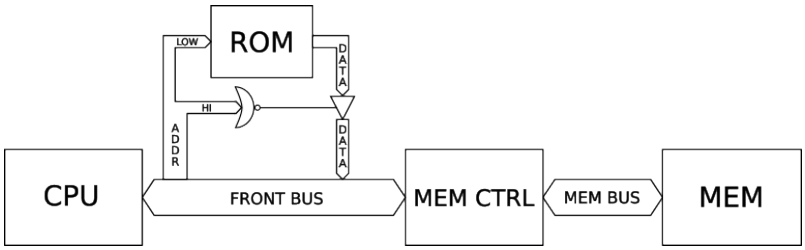
Напоменимо да мапирање описано у претходном параграфу подразумева испреплетени приступ само на нивоу банки унутар меморијских чипова. Ако желимо додатно преплитање између рангова, тада је потребно у горњем мапирању битове за избор канала и ранга поставити да буду најнижи битови у адреси. Наравно, ово додатно компликује посао контролеру, јер је потребно истовремено активирати више рангова, због ефикасног приступа, али је у пракси веома заступљено.

Освежавање меморије. Као што је раније речено, меморијске ћелије у динамичким меморијама се састоје из кондензатора који чувањем наелектрисања чувају вредност бита. Како кондензатори нису идеално изоловани, они постепено „цуре" и током времена могу спонтано изгубити сачувану вредност бита. Да се то не би десило, меморијске ћелије се периодично „освежавају" помоћу механизма деструктивног читања, тј. отварањем врсте. Иницирање овог поступка, као и управљање целим процесом освежавања је такође у надлежности меморијског контролера. Освежавање меморије привремено спречава рад са меморијом, те захтеви процесора који пристижу у међувремену бивају запамћени од стране контролера и обрађени касније, када се освежавање заврши. Период освежавања је обично неколико десетина милисекунди (на модерним системима је типично $64ms$). Притом, не освежавају се све ћелије одједном, јер би то предуго трајало, већ се освежава врста по врста на сваких, нпр. $10\mu s$. Такође, обично се не освежавају сви рангови и све банке истовремено, већ једна по једна, јер је на тај начин само део меморије недоступан у сваком тренутку, док се остатак меморије може нормално користити.

7.7 Перзистентни садржај у меморији

Технологија израде савремених оперативних меморија је таква да оне могу чувати свој садржај само док постоји електрично напајање. Приликом искључења рачунара, комплетан садржај меморије се губи[6], те је при поновном укључењу рачунара оперативна меморија потпуно празна.

Са друге стране, процесору је одмах по укључењу рачунара потребан програм који би извршавао. Због тога је потребно обезбедити да неки иницијални програм буде присутан у адресном простору процесора одмах по укључењу. Уобичајени начин да се то постигне је употреба неизмењиве (ROM) меморије, са којом смо се упознали у одељку 3.3.1. Као што знамо, у питању је меморија која има фабрички фиксиран садржај који се не може мењати, али се не може ни заборавити. На овај начин се обезбеђује да неки унапред предвиђени програм буде константно доступан процесору, без могућности да он буде изгубљен, оштећен или модификован на било који начин. Сврха овог програма је да омогући процесору да по укључењу рачунара иницијализује и контролише хардверске компоненте рачунара, као и да омогући учитавање и покретање других програма.



Слика 7.20: Повезивање ROM меморије

На слици 7.20 је приказан један једноставан начин[7] како се ROM меморија може уградити у адресни простор процесора. Идеја је да неки унапред фиксирани део адресног простора буде резервисан за неизмењиви садржај из ROM-а. У нашем примеру, претпоставићемо да је ROM меморија величине 1MB (тј. адреса коју ROM очекује је 20-битна), те да је првих 1MB 32-битног адресног простора резервисан за ROM. То значи да би адресе у опсегу од 0 до $2^{20} - 1$ (у хексадекадној нотацији од `00000000` до `000fffff`) требало да реферишу на адресе из ROM-а, док би у остатак адресног простора могли да се мапирају RAM модули. Карактеристика горе наведеног опсега адреса је да су виших 12 битова једнаки нули, док преосталих 20 битова представљају адресу у ROM меморији. Зато се виших 12 битова доводе на улазе НИЛИ кола које контролише бафер са три стања који пропушта излаз ROM меморије на магистралу података. Када

[6] Приметимо да у неким ранијим периодима развоја рачунара овај проблем није постојао, јер су се тада меморије израђивале на потпуно другачији начин, тако да нису заборављале свој садржај при искључењу напајања. На пример, ово је био случај са меморијама заснованим на магнетним језгрима које су биле у употреби од педесетих до седамдесетих година 20. века.

[7] Овај начин повезивања ROM меморије се у новије време више не користи, али га овде илуструјемо због једноставности.

су ови битови сви нуле, НИЛИ коло даје на излазу јединицу која омогућава да се изабрана вредност из ROM меморије пропусти на магистралу. За вредности адресе изван датог опсега, вредност из ROM меморије неће бити пуштена на магистралу, па ће магистрала бити слободна за комуникацију са контролером RAM меморије.

Процесор се фабрички конфигурише тако да по укључењу напајања започиње извршавање почев од неке унапред фиксиране адресе. Меморијски опсег у који ћемо уградити ROM меморију мора обухватати ту адресу, а ROM меморија мора почев од те адресе обезбедити почетак програма за иницијализацију рачунара. У нашем примеру, можемо претпоставити да се програмски бројач процесора по укључењу рачунара иницијализује на вредност 0, као и да ROM меморија почев од нулте адресе садржи неки смислени програм који ће се извршавати по укључењу рачунара.

Глава 8

Кеш меморија

У моделу рачунара који смо разматрали у претходним главама (приказан на слици 6.1), рачунар се састојао из процесора и оперативне (главне) меморије који су били повезани меморијском магистралом. Један од недостатака овог модела је то што је приступ меморији релативно спор, имајући у виду њену величину и технологију њене израде, као и чињеницу да је физички удаљена од процесора. Због тога је процесор по правилу принуђен да чека при сваком меморијском трансферу, било да је у питању дохватање инструкције или дохватање или чување меморијског операнда. Ово чекање за последицу има да процесор пропушта велики број циклуса часовника, притом не радећи ништа. Број циклуса чекања може варирати и зависи од много фактора (нпр. да ли је одговарајућа врста већ отворена, да ли је у току освежавање меморије, да ли се подацима приступа секвенцијално и сл). У пракси, ово чекање може трајати и *више стотина* циклуса процесора.

Уобичајено решење за овај проблем које се користи у модерним рачунарима је *кеш меморија* (енгл. *cache memory*). У питању је сасвим мала (од пар десетина килобајта до неколико мегабајта), али веома брза, статичка меморија. Идеја је да се у овој меморији налазе *копије* најчешће коришћених података и инструкција из главне меморије. Када је процесору потребан неки податак или инструкција, он ће прво проверити да ли се копија тог податка или инструкције налази у кешу. Ако је одговор потврдан, тада ће податак бити преузет из кеша, а главној меморији се неће ни приступати. Овакав сценарио је познат и као *погодак кеша* (енгл. *cache hit*). С обзиром на брзину кеш меморије, број циклуса чекања ће у случају поготка кеша бити значајно мањи, обично пар десетина циклуса, често и мање од тога[1]. Са друге стране, ако се испостави да тражени податак није у кешу, тада се мора контактирати главна меморија, што узрокује значајно дуже чекање. Овакав сценарио је познат и као *промашај кеша* (енгл. *cache miss*). У случају промашаја кеша, податак се најпре копира из меморије у кеш, а затим се прослеђује процесору. Овим се спречава промашај кеша при наредном приступу истом податку (што се обично очекује у блиској будућности).

Структура једноставног рачунарског система који користи кеш меморију дата је на слици 8.1. Као што видимо, кеш меморија се умеће између

[1] Модерне кеш меморије на првом нивоу (*L1*) могу имати кашњење од само једног циклуса часовника.

Слика 8.1: Рачунарски систем са кеш меморијом

процесора и главне меморије. Процесор више не комуницира директно са меморијским контролером као раније, већ комуницира са кеш меморијом путем магистрале коју ћемо у овом контексту називати *магистрала кеша* (на слици означена са *cache bus*).[2] Путем ове магистрале процесор од кеш меморије тражи податак или инструкцију са жељене адресе. Кеш меморија проверава да ли је податак са датом адресом садржан у њој или не. Ако се податак пронађе у кешу, он се просто испоручује процесору. У супротном, кеш меморија контактира меморијски контролер (путем предње магистрале, као што је то раније радио сâм процесор), са захтевом да јој се испоручи податак са тражене адресе. Након што меморија пошаље податак, он се смешта у кеш, а затим га кеш меморија испоручује процесору. Цео овај процес је невидљив за процесор – он све време комуницира са кеш меморијом и од ње добија тражени податак, са мање или више кашњења.

Ефикасност описаног решења зависи пре свега од тога колики је *удео погодака кеша* (енгл. *hit ratio*) у укупном броју захтева процесора за приступом подацима и инструкцијама. Овај удео зависи од тога колико често ће податак који тражимо бити у кешу, а колико често ћемо морати да одлазимо у меморију по њега. Јасно је да ако би кеш садржао случајно одабрани скуп података из главне меморије, удео погодака кеша не би био нарочито велики, имајући у виду однос величина кеш меморије и главне меморије. Ипак, пракса показује да је у модерним рачунарима удео погодака кеша обично веома висок (често и преко 95%). То значи да ћемо у великој већини случаја заправо избећи комуникацију са главном меморијом и комуницирати искључиво са кешом. Тиме се постиже драматично убрзање рада рачунара. Разлог за овако висок удео погодака кеша је у томе што се подацима и инструкцијама ипак не приступа потпуно случајно, што омогућава кеш меморији да предвиди који ће подаци и инструкције највероватније бити потребни процесору у блиској будућности. О овоме детаљно говоримо у наставку ове главе.

8.1 Принципи локалности

Као што је речено, удео погодака кеша је у модерним рачунарима обично веома велики, што обезбеђује знатно ефикаснији приступ подацима и инструкцијама из меморије. Разлог за овако висок удео погодака кеша (често и преко 95%) је у начину извршавања програма који подразумева велику предвидивост приликом приступа меморији. Ова предвидивост се огледа у такозваним *принципима локалности*. Постоји *временска* и *просторна локалност*.

[2] У модерним рачунарима, магистрала кеша обично не постоји као класична магистрала, с обзиром да се кеш меморија налази интегрисана на истом чипу са процесором, па је са њим повезана брзим интерним везама.

Принцип временске локалности гласи: *ако је програму у неком тренутку потребан неки податак или инструкција, врло је вероватно да ће му исти податак или инструкција бити поново потребан у блиској будућности.*

Илуструјмо овај феномен следећим примером кôда у језику *C++*:

```
int a[100] = { ... }; // niz celih brojeva
int S = 0;
for(int i = 0; i < 100; i++)
  S = S + a[i];
```

Дакле, у једној петљи сумирамо елементе неког низа a дужине 100. Пре петље, врши се иницијализација промењиве S на вредност 0. Ово је први приступ промењивој S, али свакако не и последњи, јер ће се промењивој S приступати још 100 пута у петљи која следи. Дакле, сасвим је уобичајено да једну те исту промењиву користимо више пута у неком делу програма (чак велики број пута, у случају петљи). Са друге стране, веома су ретке ситуације да некој промењивој приступимо једном, а да јој затим дуго не приступамо поново. Отуда можемо претпоставити (са великом вероватноћом) да ћемо, када једном приступимо промењивој S, истој промењивој приступати поново у блиској будућности, те би било добро да нам она буде „при руци".

Слична ситуација је и са инструкцијама. На пример, превод горњег кôда на машински језик би подразумевао пар десетина инструкција које би се извршавале 100 пута, с обзиром да имамо петљу. Дакле, у случају петљи (које су веома честе у програмирању), ми изнова и изнова дохватамо исте инструкције. Због тога је згодно да инструкције петље која се тренутно извршава буду негде „близу", да се могу брзо дохватити.

Кеш меморија користи принцип временске локалности управо тако што податке и инструкције које дохвата из меморије увек памти у кешу пре него што их испоручи процесору. На тај начин, она антиципира да ће ти подаци и инструкције бити потребни у блиској будућности, вођена принципом временске локалности. Подаци ће у кешу бити присутни док кеш не буде био принуђен да их избрише, због тога што је неопходно ослободити простор за неке друге податке. То се обично дешава када подаци дуго не буду коришћени, па кеш може да претпостави да више нису потребни процесору. За податак из кеша коме дуго није приступано кажемо да је *хладан у кешу* (енгл. *cold in cache*). Са друге стране, за податак коме је скоро приступано кажемо да је *врућ у кешу* (енгл. *hot in cache*). Кеш меморије обично бришу податке који су хладни, а задржавају оне који су врући, што је у сагласности са принципом временске локалности.

Принцип просторне локалности гласи: *ако је програму у неком тренутку потребан неки податак (или инструкција), врло је вероватно да ће му у блиској будућности бити потребни подаци (инструкције) из његове околине у меморији.*

Овај феномен такође можемо илустровати на примеру претходног кôда, где се у петљи редом приступа елементима низа a. Као што је познато, елементи низа се у меморији налазе један за другим, на узастопним меморијским адресама. Врло је честа појава као у датом примеру, да се елементи низа обилазе и обрађују редом. То значи да приликом приступа елементу $a[0]$ можемо претпоставити да ће нам у блиској будућности бити

потребни елементи $a[1]$, $a[2]$, и тд. Ови елементи се налазе у околини елемента $a[0]$, тј. на суседним адресама. Отуда је добро да нам и ове околне адресе буду брзо и лако доступне.

Слична је ситуација и са инструкцијама које се по правилу извршавају једна за другом, осим у случају инструкција скока. Како су скокови релативно ретки, можемо са великом вероватноћом претпоставити да ће се након неке инструкције извршавати управо инструкција која за њом следи у меморији. Отуда је добра идеја и ову инструкцију (као и инструкције које следе за њом) имати при руци ради брзог дохватања.

Кеш меморија користи принцип просторне локалности тако што никада из меморије не учитава само један податак са дате адресе, већ читаву *околину* дате адресе из меморије. Ове околине су обично величине неколико десетина бајтова (нпр. 32 или 64 бајта). На тај начин ће са траженим податком или инструкцијом у кеш бити учитани и суседни подаци или инструкције за које претпостављамо да ће нам ускоро бити потребни, на основу принципа просторне локалности. Учитавање ових околина у кеш се врши релативно ефикасно, захваљујући операцији читања блока података коју подржавају модерне меморије, а коју смо детаљно описали у претходној глави.

Напоменимо да, иако су феномени засновани на принципима локалности природно присутни у нашим програмима, још бољем искоришћењу принципа локалности може допринети и добра програмерска пракса. Илуструјмо то следећим примером:

```
int a[100][100] = { ... }; // matrica celih brojeva
int S = 0;
for(int i = 0; i < 100; i++)
  for(int j = 0; j < 100; j++)
    S = S + a[i][j];
```

У овом примеру сумирамо елементе матрице целих бројева димензије 100×100. Спољашња петља иде по врстама, док унутрашња иде по колонама. То значи да се на суму најпре редом додају елементи нулте врсте, затим елементи прве врсте и тд. Алтернативно, могли смо имати следећи кôд:

```
int a[100][100] = { ... }; // matrica celih brojeva
int S = 0;
for(int j = 0; j < 100; j++)
  for(int i = 0; i < 100; i++)
    S = S + a[i][j];
```

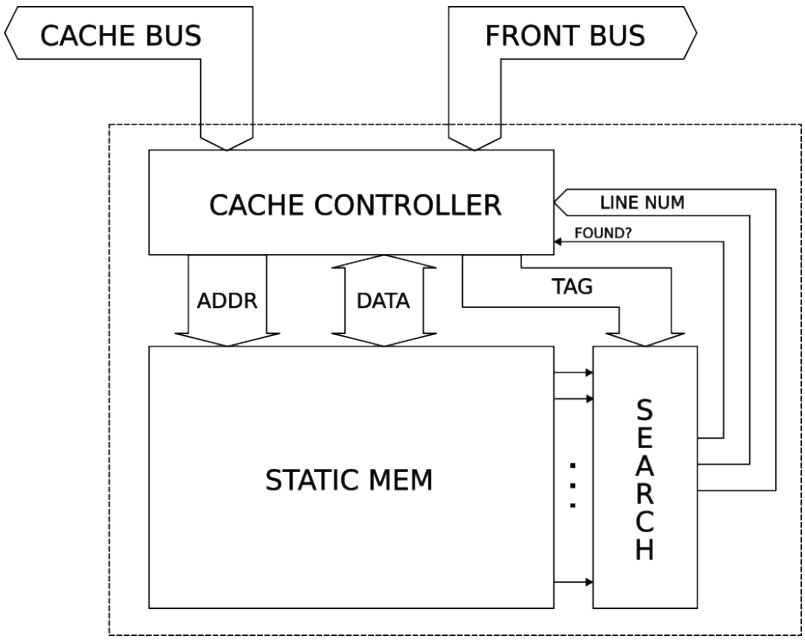
У овом случају, редослед сумирања елемената је по колонама, тј. најпре се редом на суму додају елементи нулте колоне, затим прве колоне и тд. Наравно, крајњи резултат је исти. Ипак, у језику *C++* је прва варијанта бољи избор, због тога што је матрица имплементирана као низ низова, тачније, као низ врсти. То значи да ће у меморији најпре бити смештени редом елементи нулте врсте, а одмах за њима елементи прве врсте и тд. Отуда сумирање по врстама иде у прилог искоришћењу принципа просторне локалности, с обзиром да ће приликом приступа елементу $a[0][0]$ заједно са њим у кеш бити учитани и елементи $a[0][1]$, $a[0][2]$, и тд. Са друге стране, друга варијанта кôда која подразумева сумирање по колонама би након

елемента $a[0][0]$ приступала елементу $a[1][0]$, што је податак који је читавих 400 бајтова удаљен, те вероватно неће бити у кешу у том тренутку.

Са друге стране, постоје програмски језици у којима се матрице чувају у меморији редом по колонама (попут језика *Fortran*). У случају таквих програмских језика погодније је сумирати редом по колонама, а не по врстама, ради бољег искоришћења кеша. Отуда је понекад важно знати неке карактеристике ниског нивоа у вези са програмским језицима које користимо, ако желимо да пишемо ефикасне програме.

8.2 Организација кеша

Структура једноставне кеш меморије дата је на слици 8.2. Главни део кеша је мала статичка меморија која чува податке. Функцијама кеша управља секвенцијално коло које се назива *контролер кеша* (енгл. *cache controller*). Ово коло је задужено за комуникацију како са процесором, тако и са контролером меморије. Такође, ово коло контролише читање и упис у кеш, претрагу кеша и остале функције. Посебна логика за претрагу (коло означено са *search* на слици) биће објашњена касније.



Слика 8.2: Структура кеша

Свака адресибилна локација у кеш меморији представља једну *линију кеша* (енгл. *cache line*). Пример структуре једне линије кеша приказан је на слици 8.3. Главни део кеш линије чине битови који садрже податке копиране из главне меморије (означен са *data* на слици). Типично, кеш линија садржи копије 32 или 64 суседних бајтова из оперативне меморије. Поред тога, кеш линија садржи битове који се користе за претрагу, а које ћемо називати *ознака* или *таг* (енгл. *tag*). Такође, обично постоји један бит

који означава да ли кеш линија садржи податке или је празна. Овај бит ћемо називати *валидни бит* (енгл. *valid bit*) и означаваћемо га са V. Поред тога, може постојати и додатни бит који означава да ли је садржај кеш линије модификован након што је учитан у кеш. Овај бит ћемо називати *прљави бит* (енгл. *dirty bit*) и означаваћемо га са D. Улогу ових битова објаснићемо детаљније касније.

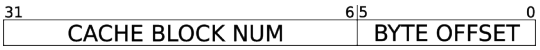
Слика 8.3: Структура кеш линије

У наставку овог излагања, одређености ради, претпостављаћемо да је величина кеш линије 64 бајта. Такође, претпостављаћемо да је кеш меморија величине 1MB (2^{20} бајтова), што значи да имамо 2^{14} кеш линија са по $2^6 = 64$ бајта. Најзад, претпоставићемо да је адресни простор процесора 32-битни, тј. да су адресе које процесор испоручује кеш меморији у својим захтевима 32-битне.

У кеш линију се из меморије увек копира неки блок од 64 бајтова који почиње на адреси која је дељива са 64. Овакав блок података називаћемо *кеш блок*. На пример, бајтови на адресама од 0 до 63 чине нулти кеш блок у оперативној меморији, бајтови на адресама од 64 до 127 чине следећи кеш блок и тд. Кеш блокови се увек у целини копирају у неку од слободних кеш линија. Типично, ово се постиже операцијом читања блока података на магистрали. На пример, ако је магистрала података 64-битна (8 бајтова), тада можемо имати операцију читања блока података која се састоји из читања осам суседних меморијских 8-бајтних речи које заједно чине један кеш блок од 64 бајта. Ово је типичан сценарио на *x86-64* архитектури.

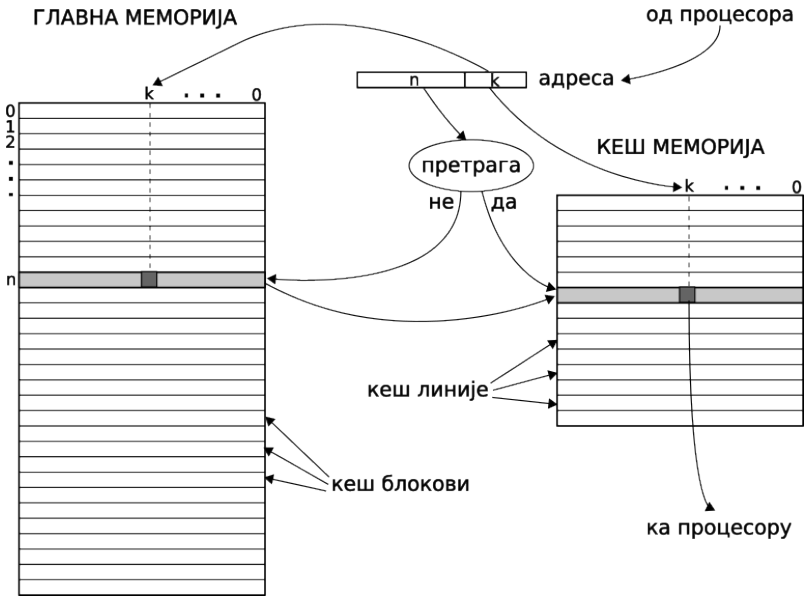
Кеш блокови играју улогу „околина" које се заједно са траженим податком учитавају у кеш, као што је дискутовано у претходном поглављу када смо разматрали феномен просторне локалности. Када процесор захтева учитавање податка са неке адресе, тада се одређује јединствени кеш блок који садржи дату адресу и проверава се да ли се тај кеш блок налази у некој од кеш линија (претрагом кеша која ће бити објашњена касније). Ако је одговор потврдан, тада се из кеш линије издвајају они бајтови које је процесор тражио. У супротном, цео кеш блок се учитава у неку од слободних кеш линија, а затим се, као и малопре, тражени бајтови из кеш линије испоручују процесору.

Чињеница да су кеш блокови у меморији поравнати на адреси која је дељива са 64 омогућава једноставно одређивање адресе кеш блока на основу адресе податка који процесор захтева. На слици 8.4 приказана је логичка структура адресе бајта у меморији, имајући у виду кеш блокове. Виших 26 битова 32-битне адресе одређују редни број кеш блока, док нижих 6 битова одређују померај бајта у оквиру кеш блока.



Слика 8.4: Декомпозиција адресе на број кеш блока и померај у оквиру блока

Приступ бајту на датој адреси у меморији је илустрован на слици 8.5. Цео n-ти кеш блок се копира у неку од слободних кеш линија, а затим се очитава k-ти бајт из изабране кеш линије. У следећем приступу истом податку, кеш меморија ће утврдити да се n-ти кеш блок већ налази у кешу и одредиће редни број кеш линије у којој се налази. Из ове кеш линије биће очитан k-ти бајт и прослеђен процесору.



Слика 8.5: Илустрација приступа податку у кешу и главној меморији на основу дате адресе

8.3 Претрага кеша

Кључни услов за ефикасни приступ подацима у кешу је ефикасна имплементација претраге. Притом, треба имати у виду да се ова претрага у потпуности имплементира у хардверу. То значи да се мора водити рачуна о цени ове имплементације, у смислу сложености логичког кола које ту претрагу врши. У том смислу, приликом дизајна кеша, често је неопходно правити компромисе између прихватљиве цене и добијене флексибилности у имплементацији, која директно утиче на ефикасност употребе кеша, као што ћемо касније видети.

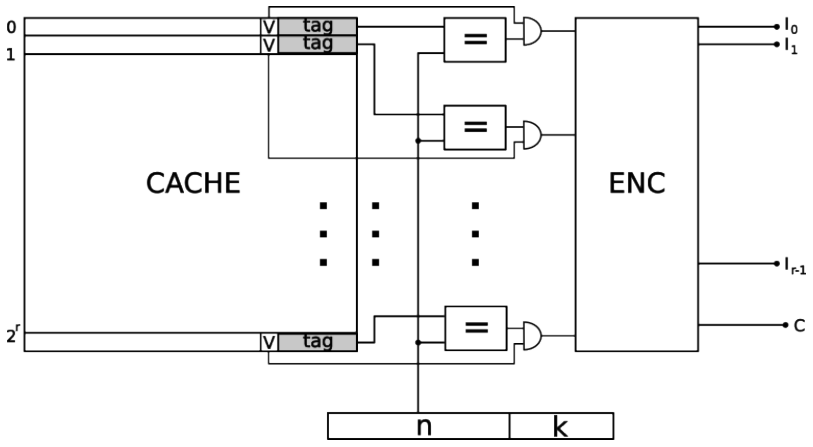
8.3.1 Потпуно асоцијативни кеш

Претпоставимо најпре случај у коме имамо максималну могућу флексибилност, у смислу да се сваки кеш блок може налазити у било којој од кеш линија. У том случају је приликом претраге потребно тражити истовремено у свим кеш линијама, како би се утврдило да ли нека од њих садржи тражени кеш блок. Свака кеш линија садржи таг поље које чува

редни број кеш блока који се у тој кеш линији налази. Када проверавамо да ли се податак налази у кешу, таг поља свих кеш линија ћемо упоређивати са вишим делом адресе који представља редни број кеш блока који садржи тражени податак (сетимо се слике 8.4 из претходног одељка). Уколико пронађемо кеш линију чије се таг поље поклапа са вишим делом адресе, тада је одговор потврдан, тј. кеш блок је пронађен у тој кеш линији. Коло за претрагу ће вратити редни број те линије, а затим ће контролер кеша из кеш линије са тим редним бројем издвојити тражени податак, на основу нижег дела адресе. Уколико не пронађемо ни једну кеш линију чије се таг поље поклапа са вишим делом адресе, одговор ће бити негативан, тј. податак се не налази у кешу. У том случају ће контролер кеша морати да приступи меморији да би из ње учитао одговарајући кеш блок.

Напоменимо да се приликом претраге разматрају само оне кеш линије чији је валидни бит V једнак јединици. Остале кеш линије су празне, па вредности њихових таг поља нису од важности и не учествују у претрази.

Уобичајен начин имплементације претраге кеша дат је на слици 8.6. За сваку кеш линију постоји један компаратор на једнакост (попут оног који је описан у одељку 3.2.4). Ови компаратори упоређују таг поља кеш линија са вишим делом адресе који представља редни број кеш блока (означен са n на слици). У нашем примеру, у питању су 26-битни компаратори, с обзиром да су редни бројеви кеш блокова, као и таг поља, 26-битни (слика 8.4). Излази ових компаратора се најпре пропуштају кроз конјункције са валидним битом одговарајућих кеш линија, а затим се прослеђују 2^r-на-r кодеру који на излазу даје редни број компаратора који на излазу има јединицу (претпоставка је да постоји највише један такав, с обзиром да се тражени кеш блок може налази у највише једној кеш линији). У случају да сви компаратори имају нуле на излазу, то значи да је претрага неуспешна, што ће се манифестовати нулом на контролном излазу C.



Слика 8.6: Имплементација претраге кеша

Ако је претрага неуспешна, тада се бира једна од празних кеш линија и у њу се учитава тражени кеш блок. Том приликом, у таг поље изабране кеш линије се уписује виших 26 битова адресе, тј. редни број кеш блока

који је учитан. На овај начин, уз сваку кеш линију чувамо редни број кеш блока који она садржи. Такође, валидни бит се приликом учитавања кеш блока поставља на један, како бисмо знали да кеш линија није празна и да је треба узимати у обзир приликом наредних претрага.

У случају да је кеш попуњен, тј. да нема празних кеш линија, тада је неопходни извршити *замену линија кеша* (енгл. *cache line replacement*). У ту сврху се најпре бира једна од кеш линија и кеш блок који се у њој налази се уклања из кеша како би се линија ослободила за кеш блок који је потребно учитати. Избор кеш линије која ће бити „жртва" приликом замене се врши на основу унапред одређене политике која је имплементирана у контролеру кеша. Најзначајније политике замене разматраћемо касније.

Описана организација кеша је позната и као *потпуно асоцијативни кеш* (енгл. *fully associative cache*). Највећи изазов у овом приступу је превелика цена описане имплементације претраге. Наиме, број кеш линија може бити прилично велики. У нашем примеру, имаћемо $2^{14} = 16384$ кеш линија и, самим тим исто толико 26-битних компаратора. Такође, потребно је конструисати кодер са 16384 улаза, што је веома скупо. Због свега овога, потпуно асоцијативни кеш се у пракси користи искључиво за веома мале кеш меморије код којих број линија не прелази неколико стотина.[3]

8.3.2 Кеш са директним мапирањем

Алтернатива потпуно асоцијативном кешу је *кеш са директним мапирањем* (енгл. *direct mapped cache*). Код ове врсте организације кеша иде се у другу крајност – сваки кеш блок може бити учитан у само једну, унапред фиксирану кеш линију чији се редни број може детерминистички одредити на основу редног броја кеш блока. Уколико је та кеш линија тренутно заузета, тада се она мора ослободити (тј. подаци из ње се морају избацити) да би се у њу учитао дати кеш блок. Дакле, не постоји могућност да се кеш блок учита у неку другу кеш линију, без обзира колико празних кеш линија у том тренутку у кешу постоји. Ово решење је, очигледно, врло нефлексибилно и може довести до врло лошег искоришћења кеша. Ипак, предност овакве организације кеша је веома јефтина имплементација претраге. Наиме, како унапред знамо у којој кеш линији се мора налазити дати кеш блок, довољно је да постоји само један компаратор на једнакост који ће проверити да ли се заиста тражени кеш блок налази у тој кеш линији или не. Остале кеш линије се не морају проверавати, јер се у њима тражени кеш блок сигурно не налази. Самим тим, ни кодер нам више није потребан, јер број кеш линије знамо унапред.

Логичка структура адресе података у меморији се у случају директног мапирања може разумети на начин приказан на слици 8.7. Виши део адресе који представља редни број кеш блока (виших 26 битова у нашем примеру) се логички дели на два дела – нижих 14 битова који одређују јединствени број кеш линије у коју тај кеш блок мора да иде, и виших 12 битова који ће играти улогу тага за упоређивање. Оно што одмах примећујемо је да ће сви кеш блокови чији се редни бројеви поклапају на нижих 14 битова (тј. који су конгруентни по модулу 2^{14}) имати исту придружену кеш линију. Отуда,

[3] Типичан пример оваквог кеша је *TLB* (енгл. *translation lookaside buffer*) кеш који се користи за транслацију виртуелних адреса у физичке, о чему ће бити више речи у некој од наредних глава.

овакви кеш блокови никако неће истовремено моћи да буду у кешу. Овакве кеш блокове називаћемо *конфликтни кеш блокови*.

Приликом учитавања кеш блока у одговарајућу кеш линију, у таг поље кеш линије се уписује виших 12 битова редног броја учитаног кеш блока. На тај начин, кеш меморија зна који се од међусобно конфликтних кеш блокова налази у датој кеш линији. Дакле, таг поља у кеш линијама ће сада бити 12-битна уместо 26-битна, што такође представља једну од уштеда у односу на потпуно асоцијативни кеш.

Слика 8.7: Структура адресе код директног мапирања

Претрага се сада одвија на следећи начин. Када кеш контролер добије адресу податка од процесора, он најпре очита 14 битова адресе који одређују редни број кеш линије у којој се једино може налазити одговарајући кеш блок (битови од 6 до 19 на слици 8.7). Затим се проверава валидни бит V ове кеш линије. Ако је он једнак нули, то значи да је кеш линија празна, па одмах знамо да имамо промашај кеша. У супротном, таг поље ове кеш линије се компаратором упоређује са највиших 12 битова адресе (битови 20 до 31 на слици 8.7). Ако се вредности поклапају, тада имамо погодак кеша, те се из дате кеш линије издваја податак на основу најнижег дела адресе. У супротном, поново имамо промашај кеша.

У случају промашаја кеша, кеш блок се мора учитати у кеш. Притом, мора се учитати баш у ту једнозначно одређену кеш линију. Овде постоје две могућности. Прва могућност је да је кеш линија празна, у ком случају се кеш блок једноставно учитава у ту кеш линију. Друга могућност је да се у кеш линији у том тренутку налази неки други конфликтни кеш блок. У том случају је неопходно извршити замену линија кеша, тј. кеш блок који се налази у кеш линији се мора избацити из кеша да би се кеш линија ослободила за кеш блок који желимо да учитамо.

Главна предност кеша са директним мапирањем је врло једноставна и јефтина имплементација претраге која не зависи од величине кеша. Са друге стране, недостатак флексибилности при избору линије у коју ћемо учитати кеш блок може довести до тога да је неопходно избацивати податке из кеша чак и када је велики број кеш линија празан. Овако нешто се у случају потпуно асоцијативног кеша никада не дешава – докле год имамо бар једну празну кеш линију, ми можемо тражени кеш блок учитати у ту линију, без потребе да из кеша избацујемо постојеће податке. Ово омогућава да подаци дуже остају у кешу, као и да кеш буде боље попуњен, што повећава удео погодака кеша.

Посматрајмо један једноставан пример. Претпоставимо да наш програм користи две глобалне променљиве x и y које се у меморији налазе редом на адресама:

```
x: 000000000000 00000000000000 011000
y: 000000000001 00000000000000 100100
```

Притом, горње 32-битне бинарне записе смо намерно поделили на три дела: виших 12 битова који представљају таг поље, наредних 14 битова

који представљају редни број кеш линије (у случају директног мапирања) и нижих 6 битова који представљају померај у оквиру кеш блока. Као што видимо, ове две променљиве се налазе у два различита, међусобно конфликтна кеш блока (оба кеш блока се пресликавају у нулту кеш линију). Уколико програм наизменично приступа овим променљивама, тада ћемо стално морати да један кеш блок избацујемо из кеш линије да бисмо други кеш блок учитали. Притом, имаћемо промашај кеша у сваком приступу. Приметимо да у тој ситуацији остатак кеша може бити полупразан, али ми немамо могућност да то искористимо, с обзиром да оба кеш блока морају ићи у исту кеш линију. Овај проблем никада не бисмо имали у случају потпуно асоцијативног кеша, јер бисмо ова два кеш блока могли да учитамо у две различите кеш линије и да их, самим тим, имамо истовремено у кешу.

Описани недостатак флексибилности је разлог због ког се кеш меморије са директним мапирањем данас ретко користе у модерним процесорима опште намене.[4] Са друге стране, за потпуно асоцијативни кеш смо већ констатовали да је у пракси обично прескуп за имплементацију, нарочито за кеш меморије веће величине. Отуда је неопходно пронаћи компромис између две описане организације који ће понудити више флексибилности, уз разумну цену имплементације. Такав компромисни дизајн разматрамо у следећем одељку.

8.3.3 Скуп-асоцијативни кеш

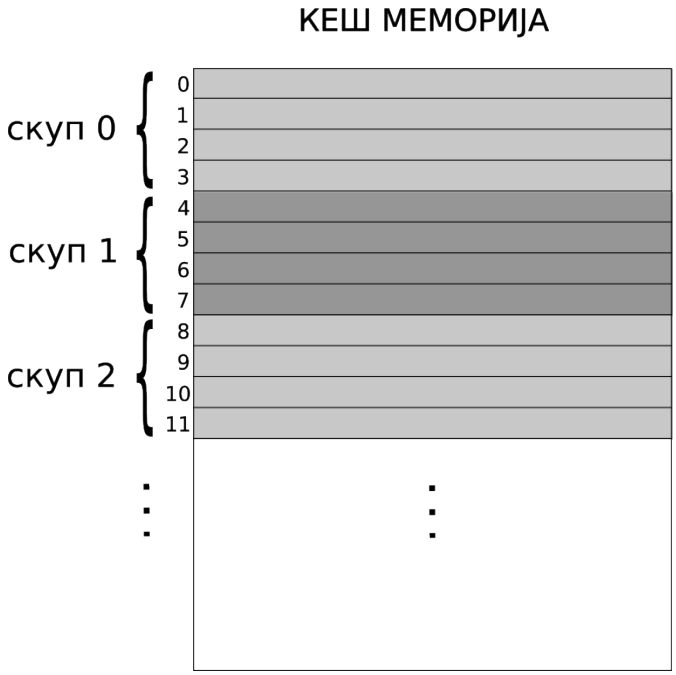
Компромис између претходна два приступа је употреба *скуп-асоцијативног кеша* (енгл. *set-associative cache*). Код овог приступа, скуп свих кеш линија се партиционише на подскупове једнаких величина. Типично, величине ових подскупова су мали степени двојке (нпр. 2,4,8 или 16). У том смислу, разликујемо 2-скуп-асоцијативни кеш, 4-скуп-асоцијативни кеш, и тд. Сваки скуп окупира одговарајући број суседних кеш линија (слика 8.8).

За сваки кеш блок из меморије сада постоји јединствен *скуп* кеш линија који му је придружен – кеш блок може бити учитан искључиво у неку од линија тог скупа, било коју која је слободна у том тренутку. На тај начин се повећава флексибилност приликом учитавања кеш блока, јер сада ипак имамо некакав избор, тј. нисмо везани за једну једину кеш линију, већ за скуп линија. Са друге стране, имплементација претраге се фокусира за један фиксирани, релативно мали скуп линија, уместо да се претражује цео кеш, као у случају потпуно асоцијативног кеша.

Ради илустрације, у даљем разматрању ћемо, уз претпоставке које су већ наведене у претходним одељцима, додатно претпостављати да су скупови линија у нашем примеру величине четири, тј. да имамо 4-скуп-асоцијативни кеш. То значи да ћемо наш скуп од 2^{14} кеш линија поделити у 2^{12} скупова од по четири кеш линије. Линије $0, 1, 2, 3$ ће чинити скуп са редним бројем 0, линије $4, 5, 6, 7$ ће чинити скуп са редним бројем 1 и тд. (слика 8.8).

Уз наведене претпоставке, логичка структура адресе у случају скуп-асоцијативног кеша је приказана на слици 8.9. Сада је виши део адресе

[4] У данашње време, типична примена кеш меморија са директним мапирањем је у једноставним процесорима специјализоване намене који се користе у микроконтролерима или уграђеним (*embedded*) системима. Код оваквих уређаја, цена и једноставност су често на првом месту.



Слика 8.8: Подела кеш линија на скупове код 4-скуп-асоцијативног кеша

који представља редни број кеш блока у меморији (виших 26 бита) подељен на нижих 12 битова који одређују редни број *скупа* у кеш меморији у који се дати кеш блок пресликава, док виших 14 битова чине таг поље. Притом, као и раније, кеш блокове који се пресликавају у исти скуп (тј. чији су редни бројеви конгруентни по модулу 2^{12}) називамо *конфликтни* кеш блокови. За разлику од директног мапирања, овде постоји могућност да конфликтни блокови коегзистирају у кешу, јер имамо четири линије у сваком скупу. Приликом учитавања кеш блока у неку од кеш линија одговарајућег скупа, у таг поље те кеш линије уписује се 14 највиших битова адресе, за потребе касније претраге. Дакле, у овом случају, таг поља ће бити 14-битна.

Слика 8.9: Структура адресе код скуп-асоцијативног кеша

Приликом претраге, контролер кеша ће најпре очитати 12 битова који представљају редни број скупа (битови од 6 до 17 на слици 8.9). На основу тога ће се претрага фокусирати на тај скуп, јер знамо да се тражени кеш блок може налазити само у некој од линија тог скупа. За саму претрагу ће нам сада бити потребно четири 14-битна компаратора на једнакост, као и један 4-на-2 кодер који је релативно једноставно направити. Највиших 14 битова адресе се истовремено пореде са 14-битним таг пољима кеш линија у изабраном скупу и на тај начин се утврђује да ли се кеш блок налази

у некој од те четири кеш линије (тј. да ли имамо погодак или промашај кеша).

У случају поготка кеша, тражени податак ће, уз помоћ нижих битова адресе, бити издвојен из пронађене кеш линије. У супротном, кеш блок мора бити учитан у кеш и то баш у неку од кеш линија тог скупа. Уколико је нека од кеш линија датог скупа слободна, учитаћемо кеш блок у ту кеш линију и поставићемо њено таг поље на највиших 14 битова адресе. Уколико су све четири линије скупа заузете, тада је потребно једну од њих испразнити, како би се иста ослободила за учитавање траженог кеш блока. Као и код потпуно асоцијативног кеша, и овде је потребно на неки начин одлучити коју од четири линије изабрати за „жртву" приликом замене линија кеша. Конкретне политике које одређују овај избор разматраћемо касније.

Из претходног разматрања закључујемо да је имплементација претраге код скуп-асоцијативног кеша релативно јефтина. Са друге стране, јасно је да ће негативан сценарио приказан у примеру из претходног одељка у случају скуп-асоцијативног кеша бити избегнут, јер ћемо моћи истовремено у кешу да имамо и променљиву x и променљиву y. Да би се код 4-асоцијативног кеша догодио сценарио попут оног описаног у горњем примеру, било би потребно да имамо програм који истовремено ради са пет различитих променљивих које се налазе на удаљеним локацијама у меморији, а које су, сасвим случајно, у међусобно конфликтним кеш блоковима (тј. блоковима који се пресликавају у исти скуп линија у кешу). На пример, претпоставимо да имамо променљиве x, y, z, u, v на следећим адресама у меморији:

```
x: 00000000000000 000000000000 011000
y: 00000000000001 000000000000 100100
z: 00000000000010 000000000000 001000
u: 00000000000110 000000000000 111000
v: 00000000010100 000000000000 110000
```

Овде смо у горњем запису адресе поделили на делове према шеми датој на слици 8.9. Дакле, видимо да све адресе одговарају истом, нултом скупу линија у кешу, али припадају различитим кеш блоковима. Отуда ће само четири од њих моћи да истовремено коегзистирају у кешу. Претпоставимо да су то, у датом тренутку, променљиве x, y, z, u, док променљива v није у кешу. Ако програм приступи променљивој v, тада ћемо имати промашај кеша, а контролер кеша ће изабрати једну од кеш линија коју ће испразнити, да би кеш блок који садржи променљиву v био учитан. Претпоставимо да је то кеш линија која садржи променљиву y. Након замене линија кеша, имаћемо променљиве x, z, u, v у кешу, а променљива y више неће бити у кешу. Када променљива y поново буде била коришћена, поново ћемо имати промашај кеша, где ће y бити учитана у кеш, а нека друга од променљивих (нпр. променљива z) бити избачена из кеша. Уколико, на пример, алгоритам у некој петљи изнова и изнова користи свих пет променљивих, гарантовано ћемо имати бар један промашај кеша у свакој итерацији петље. Притом, кеш и даље може бити полупразан, али нама то не помаже, јер смо ми ограничени на тај један четворочлани скуп кеш линија.

Ипак, јасно је да је вероватноћа појаве оваквог сценарија знатно мања код скуп-асоцијативног кеша него код кеша са директним мапирањем.

Додатно, та вероватноћа се значајно смањује са повећањем степена асоцијативности кеша (нпр. много је мања код 8-асоцијативног кеша него код 2-асоцијативног кеша). Отуда скуп-асоцијативни приступ обично пружа сасвим задовољавајуће перформансе, уз релативно једноставну имплементацију. Зато се у модерним рачунарима готово искључиво користе различите варијанте скуп-асоцијативних кеш меморија.

8.4 Модификација података у кешу

У досадашњем разматрању, ми смо се углавном фокусирали на читање података (или инструкција) из меморије и видели смо на који начин се тај процес одвија у присуству кеша. Укратко, подаци се најпре довлаче у кеш (заједно са читавим кеш блоком који их садржи), а затим се надаље читају из кеша по потреби. У овом поглављу разматрамо на који се начин операција уписа податка у меморију обавља у присуству кеш меморије.

Пре разматрања саме операције уписа, подсетимо се да за податке који се налазе у кешу ми заправо имамо две копије – једну у кешу, а другу у главној меморији. Уколико се подаци само читају и никада се не модификују, тада ће копија податка у кешу увек бити идентична копији у главној меморији. Међутим, уколико процесор иницира операцију уписа, тада се поставља питање где ће се упис извршити – у кешу, у меморији или на оба места?

Одговор на ово питање зависи од изабране *политике уписа* (енгл. *cache write policy*). У наставку разматрамо најчешће коришћене политике уписа.

8.4.1 Политика директног уписа

Политика *директног уписа* (енгл. *write-through*) подразумева да се при сваком упису вредност податка ажурира и у кешу и у меморији. На овај начин, обезбеђује се да копије података у кешу и у меморији увек буду идентичне, што обезбеђује конзистентност података у сваком тренутку. Предност овог приступа је једноставност имплементације, јер не захтева никакву додатну логику за одржавање конзистентности података. Очигледан недостатак је то што је потребно приступати меморији при свакој операцији уписа. Другим речима, предности кеша се ефективно не користе приликом уписа, већ само приликом читања.

Поставља се и питање шта се дешава уколико вршимо упис у променљиву која се тренутно не налази у кешу. Оваква ситуација је позната и као *промашај при упису* (енгл. *write miss*). Овде постоје две варијанте. Прва је *упис без алокације* (енгл. *write-no-allocate*), где се просто ажурира податак у меморији, без учитавања података у кеш. Ова политика је најједноставнија, али не обезбеђује никакве бенефите при наредним операцијама читања (које ће врло вероватно да уследе, јер смо вероватно модификовали променљиву да бисмо затим користили њену нову вредност). Алтернативно, имамо *упис са алокацијом* (енгл. *write-allocate*) код кога се врши ажурирање података у меморији, након чега се одговарајући кеш блок учитава у кеш, како би био доступан за наредне операције читања.

8.4.2 Политика одложеног уписа

Политика *одложеног уписа* (енгл. *write-back*) подразумева да се приликом уписа ажурира само копија у кешу, док копија у меморији остаје неажурна. Упис у меморију се врши тек када се садржај избацује из кеш линије, услед замене кеш линија. Том приликом се цео садржај кеш линије уписује у одговарајући кеш блок у меморији, након чега се кеш линија празни, а у њу уписује нови садржај. Дакле, садржаји кеш линије и одговарајућег кеш блока се синхронизују тек када то постане неопходно, непосредно пре него што ћемо садржај кеш линије обрисати. Приметимо да овакав приступ не нарушава конзистентност при читању података, зато што приликом читања ми увек најпре проверавамо да ли постоји копија у кешу и, ако постоји, користимо ту копију. То значи да ће након модификације податка у кешу све наредне операције читања тог податка очитавати ту модификовану вредност из кеша – дакле, нову вредност, а не ону стару из меморије.

Додатна оптимизација која се обично користи је вођење евиденције о томе да ли је садржај кеш линије модификован или не од како је учитан у кеш. У ову сврху се користи раније поменути прљави бит D. Приликом учитавања кеш блока у кеш линију, прљави бит те линије се поставља на нулу, што означава да је садржај кеш линије идентичан садржају одговарајућег кеш блока у главној меморији. Када дође до операције уписа којом се модификује неки податак у тој кеш линији, тада се бит D поставља на један, што означава да је копија садржаја у кеш линији новија од копије која стоји у одговарајућем кеш блоку у главној меморији. Приликом замене линија, уколико је потребно испразнити ту кеш линију, најпре се проверава вредност бита D. Ако је он једнак нули, тада знамо да је садржај кеш линије идентичан оном у меморији, па не морамо вршити упис, већ можемо слободно у кеш линију учитати нови садржај. Уколико је бит D једнак јединици, тада знамо да копија у меморији више није ажурна, те је неопходно садржај кеш линије уписати у одговарајући кеш блок, пре него што се тај садржај замени новим. На овај начин се упис у меморију врши само онда када је заиста неопходно, тј. када је копија података из кеша ажурнија од копије у главној меморији.

Као и код политике директног уписа, и овде постоје две могућности у случају промашаја при упису: упис без алокације и упис са алокацијом. Притом, упис са алокацијом се много чешће користи у контексту политике одложеног уписа. Том приликом се кеш блок који садржи податак учитава у неку од кеш линија, а затим се ажурира копија податка у кешу (али не и у меморији). Упис без алокације функционише на исти начин као и код политике директног уписа, али се у контексту одложеног уписа знатно ређе користи.

Главна предност политике одложеног уписа је у томе што се приликом уписа не приступа главној меморији, већ само кешу, као и приликом читања. Приступ главној меморији се врши само приликом замене линије кеша, што се дешава тек када се кеш (или одговарајући скуп линија) попуни. На овај начин, ефикасност кеша долази до изражаја и приликом уписа, а не само приликом читања података.

Недостатак политике одложеног уписа је у нешто сложенијој имплементацији операције замене линија кеша, јер у општем случају није

довољно само уписати нови садржај у изабрану кеш линију, већ може бити неопходно да се тренутни садржај кеш линије претходно сачува. Одговорност за тај поступак је на контролеру кеша чија ће логика, отуд, бити нешто сложенија у случају политике одложеног уписа. Ипак, корист од употребе политике одложеног уписа далеко превазилази описане недостатке, па се из тог разлога политика одложеног уписа доминантно користи у модерним рачунарима.

8.5 Замена линија кеша

Као што је већ речено, постоје ситуације у којима је неопходно један садржај неке кеш линије заменити другим, тј. обрисати постојеће податке из кеш линије да би у њу био учитан неки нови кеш блок. Овај поступак се назива *замена кеш линије* (енгл. *cache line replacement*) и спроводи се када дође до промашаја кеша, и то у следећим случајевима:

- код потпуно асоцијативног кеша, када је кеш попуњен, тј. не постоји ни једна празна кеш линија у коју би се уписао тражени кеш блок
- код кеша са директним мапирањем, када је попуњена јединствена кеш линија у коју се мора учитати тражени кеш блок
- код скуп-асоцијативног кеша, када је попуњен одговарајући скуп који је придружен траженом кеш блоку, тј. ни једна линија у том скупу није празна

У свим овим случајевима, не постоји могућност да се тражени кеш блок учита у кеш меморију, а да се претходно нека кеш линија не испразни. У случају кеша са директним мапирањем, не постоји дилема коју кеш линију морамо испразнити, те је у том случају ова одлука тривијална. Са друге стране, код потпуно асоцијативног кеша, кандидати за избацивање су све кеш линије, док су код скуп-асоцијативног кеша кандидати за избацивање линије кеша из одговарајућег скупа. Када имамо више потенцијалних кандидата за избацивање, потребно је на неки начин одабрати ону линију коју је најбоље избацити из кеша. У ту сврху користе се различите *политике замене кеш линија* (енгл. *cache replacement policy*).

Пре него што пређемо на разматрање конкретних политика замене кеш линија, најпре треба да разјаснимо шта у овом контексту значи *најбољи* кандидат за избацивање. Подсетимо се да је наш превасходни циљ да максимизујемо ефикасност употребе кеша, која се, као што знамо, огледа у уделу погодака кеша. Другим речима, ми желимо да одаберемо кандидата за избацивање тако да минимизујемо број промашаја кеша у будућности. У том смислу, идеална политика би се састојала у томе да избацујемо ону кеш линију која садржи податке који најдуже неће бити коришћени у будућности, јер ћемо тиме максимално одложити евентуалне промашаје кеша. Међутим, проблем је у томе што ми не можемо са сигурношћу знати које ће податке процесор користити у будућности. Отуда ову идеалну политику није могуће имплементирати у пракси. Једино што нам преостаје је да размотримо политике које ће најбоље апроксимирати ову идеалну политику. Већина ових апроксимација се заснивају на идеји да

се разматрају претходни приступи кешу и да се на основу тога антиципирају будући захтеви процесора.

8.5.1 LRU политика

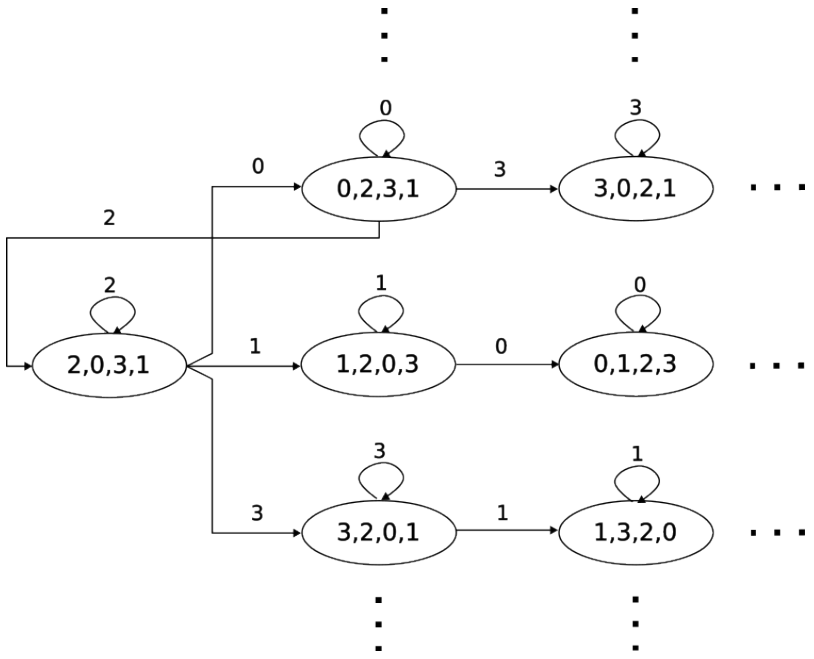
Једна од најчешће коришћених политика замене кеш линија је *замена најдуже некоришћене линије* (енгл. *least recently used* (LRU)). Код ове политике се води евиденција о томе којој линији најдуже није приступано у претходном периоду. Када дође до замене линија, за избацивање се бира управо линија која најдуже није коришћена. Оправдање за овакву политику се огледа у претпоставци да линија која дуго није коришћена вероватно више није потребна програму (јер је програм вероватно изашао из те петље, функције или блока) те се може безбедно уклонити из кеша. Са друге стране, линије које су недавно коришћене вероватно садрже податке који се активно користе у програму, те није паметно избацивати их из кеша. Другим речима, LRU политика се заснива на идеји да је боље избацивати податке који су *хладни у кешу*, док *врућe* податке треба задржати. Ово је у складу са раније описаним принципом временске локалности. Од свих политика замене, LRU политика најбоље подупире тај принцип, због чега се обично сматра најбољом политиком замене у пракси.

Да бисмо имплементирали LRU политику, потребно је да омогућимо да се у сваком тренутку одржава информација о хронологији приступа линијама које су кандидати за избацивање. На пример, претпоставимо да имамо 4-скуп-асоцијативни кеш. У сваком четворочланом скупу линија потребно је знати којој линији је најскорије приступано, којој линији није приступано нешто дуже и тд. На пример, поредак $2, 0, 3, 1$ би означавао да је најскорије приступано линији 2, нешто дуже није приступано линији 0, па линији 3, а најдуже није приступано линији 1. Овај поредак је потребно ажурирати при сваком приступу некој од линија кеша у том скупу. На пример, ако приступимо неком податку у линији 3, тада се поредак мења у $3, 2, 0, 1$. Дакле, линија 3 долази на почетак реда, док поредак осталих линија остаје неизмењен. Ако се након тога приступи линији 1, поредак ће се променити у $1, 3, 2, 0$. Када дође до потребе за заменом линија кеша, биће изабрана она линија која је последња у поретку (у нашем примеру, линија 0).

Имплементација описаног алгоритма се у пракси своди на *коначни аутомат* чија ће стања бити сви могући поретци, а прелази између њих ће бити означени линијама којима се приступа (слика 8.10). На пример, из стања $2, 0, 3, 1$ ћемо преласком по линији 3 прећи у стање $3, 2, 0, 1$, док ћемо преласком по линији 1 прећи у стање $1, 2, 0, 3$. Овај аутомат ће имати $4! = 24$ стања. Прелазак из стања у стање се врши при сваком приступу некој линији кеша из тог скупа. На тај начин, у сваком тренутку знамо којој линији најдуже није приступано, те можемо донети одлуку о избацивању, ако дође до замене линија.

За чување стања потребно је $\lceil \log_2(24) \rceil = 5$ битова. За сваки скуп чува се по једно стање, док је имплементација комбинаторне логике аутомата заједничка за све скупове.

Главни проблем са имплементацијом LRU политике је у томе што број стања горњег аутомата расте веома брзо са величином скупа. Већ у случају 8-асоцијативног кеша, имамо $8! = 40320$ стања, што је веома компликовано



Слика 8.10: Фрагмент LRA аутомата код 4-скуп-асоцијативног кеша

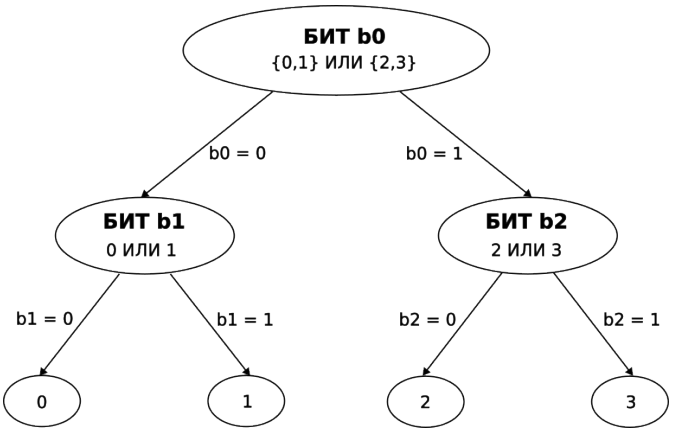
имплементирати у хардверу. Отуда се ова политика у пракси користи само за скуп-асоцијативне кеш меморије малог степена асоцијативности (обично до степена 4).

8.5.2 Псеудо-LRA политика

У случају скуп-асоцијативног кеша са већим степеном асоцијативности (8 или више линија по скупу), уместо LRA политике обично се разматрају неке њене апроксимације. Једна таква политика позната је као *псеудо-LRA политика*. Код ове политике не инсистирамо на томе да пронађемо линију која најдуже није коришћена, довољно нам је да пронађемо неку линију за коју знамо да није коришћена довољно дуго. Другим речима, не захтевамо да одабрана линија буде последња у редоследу приступа (као у претходном одељку), већ нам је довољно да буде близу зачеља, или да макар не буде прва. Ово нас ослобађа потребе да чувамо потпуни редослед приступа, већ можемо чувати само неке информације о том поретку.

Псеудо-LRA политика се имплементира тако што се скуп линија кандидата дели на два једнака дела, нижи и виши, при чему се користи један бит за чување информације о томе да ли је линија којој је последњој приступано у нижем или вишем делу. На пример, у случају 4-скуп-асоцијативног кеша, имамо четири линије $0, 1, 2, 3$ у скупу, при чему скуп $\{0, 1\}$ чини нижи део, а скуп $\{2, 3\}$ чини виши део. Означимо са b_0 горе поменути бит. Ако је бит $b_0 = 0$, то значи да је последњи приступ био у линији 0 или линији 1, док $b_0 = 1$ значи да је последњи приступ био у линији 2 или линији 3. Даље се ова два подскупа поново деле на два једнака

дела по истом принципу. На пример, скуп $\{0,1\}$ се дели на нижи део који чини линија 0 и на виши део који чини линија 1. Поново имамо један бит b_1 који одређује да ли је скорији приступ био у линији 0 ($b_1 = 0$) или у линији 1 ($b_1 = 1$). Слично, скуп $\{2,3\}$ се дели на нижи део (линија 2) и виши део (линија 3) при чему бит b_2 одређује да ли се скорије приступало линији 2 или линији 3.



Слика 8.11: Стабло одлучивања код псеудо-LRA политике

Описани приступ је илустрован на слици 8.11 у облику стабла одлучивања. Када год се приступи некој од линија, у приказаном стаблу се иде од корена ка листу који представља ту линију и битови на тој путањи се ажурирају. На пример, ако приступи линији 2, тада ћемо најпре бит b_0 поставити на 1 (јер идемо у десно подстабло), а затим ћемо бит b_2 поставити на 0 (јер идемо у лево подстабло). На овај начин ћемо увек моћи да пронађемо линију којој је најскорије приступано, просто пратећи вредности битова од корена ка листовима. Међутим, нећемо моћи да утврдимо којој линији најдуже није приступано. Ово је зато што битови b_0, b_1 и b_2 чувају само парцијални поредак међу линијама, у односу на време приступа. На пример, бит b_1 нам само говори да ли је скорије приступано линији 0 или линији 1, ништа више од тога. Слично је и са битом b_2 који одређује однос линија 2 и 3. Бит b_0 нам само говори у којој се половини скупа налази најскорије коришћена линија. Из ових парцијалних информација ми не можемо потпуно реконструисати поредак, па самим тим не можемо знати која је линија најдуже некоришћена.

Поступак избора кандидата за избацивање се своди на пролазак кроз стабло од корена ка листу, притом избегавајући онај пут који нас води ка скорије коришћеним линијама. На пример, ако је $b_0 = 1$, ми знамо да линија којој је најскорије приступано није у левом подстаблу. Зато ћемо ми одабрати лево подстабло, јер ћемо тако избећи избацивање најскорије коришћене линије. Ако је, даље, бит $b_1 = 0$, то значи да је линија 0 скорије коришћена од линије 1. Зато ћемо ми одабрати линију 1, јер желимо да избегнемо избацивање скорије коришћене линије 0. Дакле, за избацивање биће одабрана линија 1. Као што смо претходно констатовали, не постоји гаранција да је линија 1 заиста најдуже некоришћена. Ипак, знамо да

постоје бар две линије које су скорије коришћене од ње (једна је у десном подстаблу, а друга је линија 0). То значи да ће изабрана линија бити или најдуже некоришћена, или друга најдуже некоришћена у скупу.

У општем случају, ако је величина скупа n (неки степен двојке), потребан нам је $n-1$ бит за чување информација о парцијалном поретку (број унутрашњих чворова потпуног бинарног стабла). По један овакав скуп битова постоји за сваки скуп линија. Сам поступак ажурирања битова и проналажења кандидата за избацивање који смо малочас описали се једноставно може имплементирати комбинаторним колом које је заједничко за све скупове у кеш меморији. За линију коју изаберемо знаћемо да, у најгорем случају, постоји бар $\log_2(n)$ линија које су скорије коришћене од ње. У просеку, обично је и знатно боље од тога.

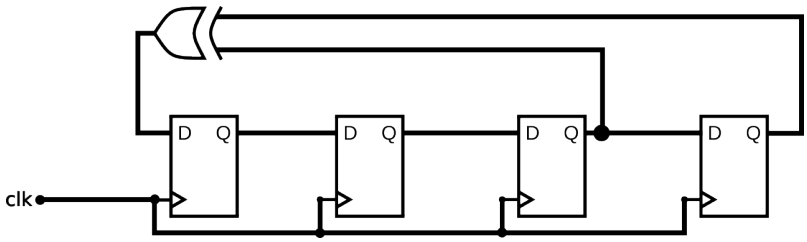
8.5.3 Политика случајног избора

У случају скуп-асоцијативног кеша са већим степеном асоцијативности, као и у случају потпуно асоцијативног кеша, често се, због једноставности, користи и *политика случајног избора*. Код ове политике се на случајан начин бира једна од линија која се избацује из кеша. Иако на први поглед ово не изгледа као нарочито добра политика, испоставља се да је она упоредива са LRU по перформансама. Наиме, при случајном избору, углавном ћемо бирати неку од „средњих" линија у поретку приступа. За веће скупове, ово је обично довољно „хладно" у кешу, тако да избацивање таквих линија неће значајно пореметити перформансе.

Са друге стране, имплементација политике случајног избора је веома једноставна и јефтина, чак и у случају потпуно асоцијативног кеша. Све што је потребно је имплементирати хардверски генератор псеудо-случајних бројева. Ово се може урадити тако што формирамо бројач са произвољним редоследом стања (поглавље 4.7), при чему стања представљају редне бројеве линија, а редослед ових стања се пажљиво одабере тако да симулира случајност. Стања се чувају засебно за сваки скуп, док је комбинатнорна логика за израчунавање наредног стања заједничка за све скупове. При свакој замени линија, бројач прелази у наредно стање и то стање се користи као редни број линије коју треба избацити. Други начин је употреба тзв. *померачких регистара са линеарном повратном спрегом* (енгл. *linear feedback shift register*(LFSR)). Ови регистри су веома једноставни, а имају веома добре карактеристике када је у питању симулација случајности приликом преласка из стања у стање. Пример имплементације једног четворобитног LFSR-а је дат на слици 8.12.

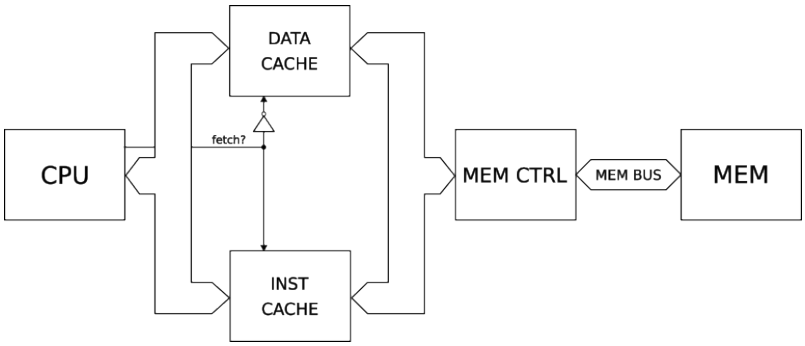
8.6 Раздвојени и унификовани кеш

Током извршавања програма, процесор приступа како инструкцијама, тако и подацима у меморији. С обзиром да се при сваком приступу меморији кеш блок који садржи тражену адресу пребацује у кеш меморију, јасно је да се у кеш меморији могу наћи и инструкције и подаци. У претходном разматрању смо претпостављали да постоји јединствен меморијски простор у кешу у који се смештају и инструкције и подаци на потпуно исти начин. Другим речима, кеш меморија није разликовала захтев за дохватање



Слика 8.12: Четворобитни LFSR регистар

инструкције од захтева за дохватање податка – у оба случаја смо просто добијали од процесора адресу и дохватали садржај са те адресе (тачније, кеш блок који садржи ту адресу). Овакав кеш се назива *унификовани кеш* (енгл. *unified cache*).



Слика 8.13: Раздвојени кеш

Алтернативни приступ је да постоји посебан кеш за инструкције, а посебан за податке. Оваква архитектура кеш меморије се назива *раздвојени кеш* (енгл. *split cache*), а њена поједностављена имплементација приказана је на слици 8.13. Доњи кеш на овој слици садржи само инструкције, док горњи кеш садржи само податке. Процесор је повезан са оба кеша, а посебним контролним сигналом се одређује да ли ће се приступати инструкцијском кешу (у случају дохватања инструкције) или кешу за податке (у свим осталим случајевима). Ова два кеша функционишу независно један од другог, а оба могу контактирати меморијски контролер по потреби, у случају промашаја кеша.

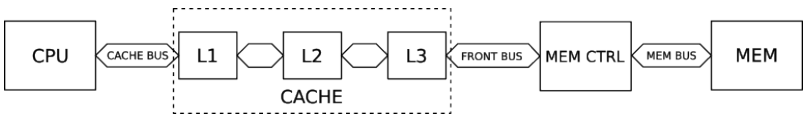
Предност раздвојеног кеша је у томе што се кеш може боље прилагодити обрасцима приступа који могу бити различити у случају података и у случају инструкција. Кеш за инструкције и кеш за податке не морају бити исте величине (мада често јесу), нити морају имати исти степен асоцијативности, што нам даје флексибилност приликом дизајна. Такође, инструкцијски кеш се може имплементирати једноставније, зато што се инструкције само читају, никада се не модификују. Отуда није потребно имплементирати политике уписа, чувати информације о променама (прљави бит) и сл.

Још једна предност раздвојеног кеша је у томе што може значајно смањити учесталост колизија приликом наизменичног приступа инструкцијама и подацима. Наиме, инструкције и подаци се обично налазе у различитим деловима адресног простора, често на веома удаљеним адресама. Ипак, како удаљени кеш блокови могу бити конфликтни, у случају унификованог кеша може се догодити да дохватање неке инструкције доведе до избацивање кеш линије са подацима и обрнуто. У случају раздвојеног кеша до овога неће доћи, с обзиром да се подаци и инструкције држе у различитим кеш меморијама.

Са друге стране, главна предност унификованог кеша је једноставност, јер не морамо да имплементирамо две кеш меморије и два контролера кеша. Такође, унификовани кеш је флексибилнији по питању искоришћења простора у кешу. Наиме, код раздвојеног кеша се може догодити да нпр. инструкцијски кеш буде потпуно попуњен, а да истовремено кеш за податке буде полупразан. Приликом дохватања нових инструкција, ми нећемо моћи да искористимо празан простор у кешу за податке, јер инструкције не могу ићи у тај кеш. Због тога ћемо морати да вршимо замену линија у инструкцијском кешу. Код унификованог кеша немамо тај проблем, јер се простори искоришћени за инструкције и за податке у кешу могу динамички мењати, према потреби. Отуда се раздвојени кеш типично користи тамо где је неопходно максимизовати перформансе, док се унификовани кеш користи тамо где је битно оптимално користити простор, минимизовати цену и поједноставити имплементацију.

8.7 Нивои кеша

У досадашњем разматрању, претпостављали смо да између процесора и главне меморије постоји само једна кеш меморија, било у унификованој или у раздвојеној варијанти. Међутим, у пракси се испоставља да једна кеш меморија не може у довољној мери амортизовати разлику у брзини између процесора и главне меморије. Наиме, брзина кеш меморије опада са њеном величином, као што је то и уобичајено код свих меморија. Уколико желимо да брзина приступа кешу буде блиска брзини самог процесора, кеш мора бити веома мали. Међутим, мали кеш ће се брзо напунити, те ћемо брзо имати проблем са честим промашајима кеша и заменама линија. Са друге стране, ако имамо већи кеш, он ће садржати више података, па ће нам обезбедити већи удео погодака кеша, али ће његова брзина бити знатно мања од брзине процесора, те ће корист од употребе кеша бити релативно мала.



Слика 8.14: Нивои кеша

Решење коме се обично приступа је да се користи више кеш меморија различитих величина (самим тим, и брзина) које су надовезане једна на

другу између процесора и главне меморије. Ове кеш меморије називамо *нивои кеша* (енгл. *cache levels*). Пример архитектуре кеша са три нивоа приказан је на слици 8.14. Овај број нивоа је уобичајен на модерним системима. Нивои кеша су обележени са L_1, L_2 и L_3, респективно, почев од кеша који је најближи процесору. За ниво кеша кажемо да је *виши* ако је ближи процесору. На пример, L_1 је највиши ниво кеша, док је L_3 најнижи ниво кеша.[5]

Процесор директно комуницира само са L_1 кешом, који је најмањи и најбржи. Кеш L_1 је повезан са L_2 кешом који је нешто већи, али нешто спорији. Најзад, L_2 кеш је повезан са L_3 кешом који је највећи и најспорији, и који директно комуницира са контролером оперативне меморије. Процесор увек тражи податак или инструкцију на жељеној адреси од L_1 кеша. Контролер L_1 кеша проверава да ли садржи тражени податак или инструкцију. Ако је одговор потврдан, податак се одмах прослеђује процесору, без потребе за контактирањем следећег нивоа кеша. У супротном, контролер L_1 кеша се обраћа L_2 кешу од кога захтева да му проследи тражени податак или инструкцију. Контролер L_2 кеша проверава да ли садржи податак на траженој адреси. Ако је одговор потврдан, податак се (заједно са одговарајућом кеш линијом) прослеђује L_1 кешу који читаву кеш линију памти код себе, а затим тражени податак прослеђује процесору. У супротном, контролер L_2 кеша контактира L_3 кеш. Уколико L_3 кеш садржи тражени податак, он ће га, заједно са припадајућом кеш линијом, проследити L_2 кешу. L_2 кеш ће добијену кеш линију проследити L_1 кешу, а овај ће тражени податак проследити процесору. Најзад, ако ни L_3 кеш не садржи податак или инструкцију са тражене адресе, његов контролер ће контактирати главну меморију, одакле ће дохватити одговарајући кеш блок и даље га проследити L_2 кешу.

Овакав приступ обезбеђује знатно бољу амортизацију разлике у брзинама између веома брзог процесора и веома споре оперативне меморије. Наиме, како је L_1 кеш веома брз, подаци и инструкције се из њега могу дохватати са веома малим кашњењем које често може износити и само један циклус часовника процесора. Ово се, поред мале величине, постиже и веома ефикасном (али и веома скупом) имплементацијом контролне логике. Такође, L_1 кеш се по правилу израђује као раздвојени кеш (обично 32KB за инструкције и 32KB за податке), што омогућава максимизацију брзине приступа како подацима, тако и инструкцијама. Најзад, L_1 кеш се физички смешта уз само језгро процесора, близу његовим регистрима и ALU јединици. Са друге стране, мала величина L_1 кеша може довести до релативно честих промашаја кеша. Међутим, цена ових промашаја сада неће бити тако скупа, зато што нећемо морати да одлазимо у главну меморију по податке, већ ћемо се обратити L_2 кешу. Овај кеш јесте нешто спорији од L_1 кеша (обично захтева десетак циклуса за приступ податку), али је и даље знатно бржи од оперативне меморије (која може захтевати и више стотина циклуса за приступ податку). Слично, ако дође до промашаја у L_2 кешу (што се дешава знатно ређе, због веће величине која обично износи од 256KB до 512KB), цена овог промашаја ће и даље бити релативно мала, јер ћемо уместо главној меморији приступати L_3 кешу. Величина L_3

[5] Оваква терминологија потиче одатле што се на дијаграмима процесор обично налази горе, меморија доле, а кешеви између, при чему је кеш који је најближи процесору нацртан изнад осталих, тј. на највишем је месту на дијаграму.

кеша је обично од 2MB до 32MB, док је време приступа обично између 30 и 40 циклуса процесора. Тек ако и у овом кешу дође до промашаја (што се још ређе дешава), мораћемо да платимо скупу цену приступа оперативној меморији. L_2 и L_3 кеш меморије се по правилу израђују као унификовани кеш. И ове кеш меморије се у данашње време смештају на исти чип са процесором, али су физички нешто удаљеније од њега у односу на L_1 кеш.[6]

8.7.1 Инклузивни и ексклузивни кеш

Једно од основних питања које се поставља приликом дизајна кеш меморије на више нивоа је да ли ће бити дуплирања података између кеш меморија на различитим нивоима. На пример, ако се неки податак налази у L_1 кешу, питање је да ли ће се копије тог податка налазити и у кеш меморијама на нижим нивоима. У том смислу, разликујемо *инклузивни* и *екскузивни кеш*. Код инклузивног кеша, ако се неки податак налази у кешу на неком нивоу, налази се и у кешевима на свим нижим нивоима. Другим речима, скуп података које садржи L_2 кеш је надскуп скупа података које садржи L_1 кеш. Слично, скуп података које садржи L_3 кеш је надскуп скупа података које садржи L_2 кеш. Када дође до промашаја у L_1 кешу, одговарајућа кеш линија се копира из L_2 кеша, при чему копија кеш линије *остаје* и у L_2 кешу. Слично, приликом промашаја у L_2 кешу, одговарајућа кеш линија се копира из L_3 кеша, при чему њена копија остаје и у L_3 кешу. Приликом замене линија у L_1 кешу, избачена линија остаје у L_2 кешу, одакле се, по потреби, може поново учитати у L_1 кеш. Слично, приликом замене линија у L_2 кешу, избачена кеш линија остаје у L_3 кешу.

Са друге стране, код ексклузивног кеша, садржаји сва три нивоа кеша су у сваком тренутку дисјунктни. Приликом учитавања податка из главне меморије, одговарајући кеш блок се смешта само у L_1 кеш. Када дође до замене линија у L_1 кешу, избачена линија се пребацује у L_2 кеш. Слично, када дође до замене линија у L_2 кешу, избачена линија се пребацује у L_3 кеш. Обрнуто, када дође до промашаја у L_1 кешу, одговарајућа кеш линија се брише из L_2 кеша и пребацује у L_1 кеш. Слично, када дође до промашаја у L_2 кешу, одговарајућа кеш линија се брише из L_3 кеша и пребацује у L_2 кеш. На овај начин, најскорије коришћени подаци се налазе у L_1 кешу, они мало дуже некоришћени се налазе у L_2 кешу, док се најдуже некоришћени подаци налазе у L_3 кешу.

Предност ексклузивног кеша је у ефикаснијој употреби меморијског простора, с обзиром да нема дуплирања података. Како кеш меморије на нижим нивоима не садрже копије података из L_1 кеша, могу садржати више других података и тиме убрзати приступ тим подацима. Са друге стране, имплементација ексклузивне кеш меморије је обично нешто компликованија, јер је потребна сложенија логика за интеракцију између различитих нивоа кеша. Наиме, ексклузивни кеш захтева *премештање* података са нивоа на ниво, и то у оба смера (од нижих ка вишим при промашају кеша, а од виших ка нижим при замени кеш линија). Ово премештање може бити каскадно, На пример, ако се податак налази у L_3 кешу, он ће најпре бити премештен у L_2 кеш, а затим из њега у L_1 кеш.

[6] У случају процесора са више језгара, обично свако језго има сопствени L_1 и L_2 кеш, док је L_3 кеш заједнички за сва језгра. Све ово се, наравно, налази интегрисано на истом чипу.

Слично, при избацивању линије из L_1 кеша, може се догодити да у L_2 кешу нема места да избачена линија буде прихваћена. Због тога је неопходно неку од линија L_2 кеша пребацити у L_3 кеш. Са друге стране, инклузивни кеш подразумева само *копирање* података са нижег нивоа на виши ниво приликом промашаја кеша, што је уобичајено понашање које не захтева никакву додатну логику (тј. кешеви на различитим нивоима у највећој мери функционишу независно један од другог). Једина додатна функционалност коју инклузивни кеш мора да обезбеди је да се приликом брисања неке линије кеша на нижем нивоу одговарајући подаци морају обрисати и из кешева на вишем нивоу, како би се задржала инклузивност. На пример, приликом брисања линије L_2 кеша, неопходно је исте те податке обрисати и из L_1 кеша, ако су у њему присутни. Ово се обично имплементира релативно једноставно, тако што L_2 кеш уз сваку своју линију чува информацију о томе да ли се иста линија налази и у L_1 кешу. Ако то јесте случај, приликом избацивања те кеш линије, L_2 кеш ће јавити контролеру L_1 кеша да обрише исту кеш линију код себе.

Додатна флексибилност инклузивног кеша је и то што величине кеш линија на различитим нивоима не морају бити исте. Наиме, кеш линије нижих нивоа могу садржати неколико кеш линија виших нивоа. На пример, кеш линија L_1 кеша може бити величине 16 бајтова, док кеш линија L_2 кеша може бити величине 64 бајтова. То значи да ће један L_2-кеш блок у меморији обухватати четири суседна L_1-кеш блока. Приликом приступа неком податку, у L_2 кеш ће бити учитан одговарајући 64-бајтни L_2-кеш блок, док ће у L_1 кеш бити учитан 16-бајтни L_1-кеш блок који је подскуп L_2-кеш блока учитаног у L_2 кеш. На тај начин, нижи нивои кеша ће садржати ширу околину траженог податка, док ће виши нивои кеша садржати ужу околину. Ово доприноси бољој експлоатацији просторне локалности, јер ће подаци ближи траженом податку бити потребни са већом вероватноћом, па ће бити ближи процесору, док ће подаци из нешто шире околине бити присутни у удаљенијим нивоима кеша. Овако нешто није могуће реализовати код ексклузивног кеша, јер он подразумева премештање линија кеша, па зато оне морају бити исте величине на свим нивоима. Инклузивни кеш подразумева копирање са нижег на виши ниво, тако да нас ништа не спречава да копирамо само део линије са нижег нивоа у мању кеш линију на вишем нивоу.

Једноставност имплементације, као и већа флексибилност су главни разлог због чега се инклузивни кеш знатно чешће користи на модерним системима. На пример, модерни процесори компаније *Intel* (попут *Core i3, i5, i7, i9* процесора) користе инклузивну организацију кеша. Са друге стране, неки старији процесори компаније *AMD* који такође имплементирају архитектуру *x86-64* (попут *K8*) су користили ексклузивни кеш, ради бољег искоришћења простора у кешу. Такође, неки новији процесори (процесори компаније *AMD*, попут *Ryzen* процесора, али и неки процесори који имплементирају *ARM* архитектуру) имплементирају хибридни приступ код кога подаци могу али не морају бити дуплирани између различитих нивоа кеша. Опис таквих приступа превазилази оквире овог текста.

8.7.2 Упис података у случају кеша на више нивоа

На крају, разјаснимо и питање уписа података у случају кеша на више нивоа. У случају инклузивног кеша, измена се врши само на највишем нивоу, у L_1 кешу. Копије на нижим нивоима, као и у оперативној меморији, остају неажурне. Њихово ажурирање се врши приликом брисања копије са вишег нивоа, по принципу одложеног уписа. Притом, врши се ажурирање само најближе копије. На пример, ако бришемо кеш линију из L_1 кеша, тада се подаци из ње копирају у L_2 кеш, тј. ажурира се копија из L_2 кеша. Остале копије (L_3 и оперативна меморија) и даље остају неажурне. L_3 копија биће ажурирана тек када се врши избацивање из L_2 кеша, док ће копија у оперативној меморији бити ажурирана тек када се буде вршило избацивање из L_3 кеша. Такође је могуће реализовати и политику директног уписа, у ком случају би увек биле ажуриране све копије (како у свим нивоима кеша, тако и у главној меморији). Ипак, ова политика је прилично неефикасна, тако да се веома ретко користи.[7]

У случају ексклузивног кеша, одговарајућа кеш линија се најпре премести у L_1 кеш (ако већ није у њему), а затим се ажурира та копија. Копија у меморији остаје неажурна у случају политике одложеног уписа, док се и она ажурира у случају политике директног уписа.

[7] Постоји и могућност да различити нивои кеша имплементирају различите политике уписа. На пример, могуће је да нивои L_1 и L_2 имплементирају политику директног уписа, док L_3 имплементира политику одложеног уписа. То значи да ће приликом уписа бити ажуриране све три копије у кешу, али не и у главној меморији.