

Архитектура и организација рачунара

Милан Банковић

18. април 2024.

Садржај

I	Основи дигиталне логике	5
1	Логичке функције и логички изрази	7
1.1	Булова алгебра	7
1.1.1	Аксиоме и основни закони Булове алгебре	7
1.2	Логички изрази и њихове нормалне форме	9
1.2.1	Конјунктивна и дисјунктивна нормална форма	9
1.2.2	Савршена конјунктивна и дисјунктивна форма	11
1.3	Логичке функције	12
1.3.1	Савршена дисјунктивна (конјунктивна) нормална форма функције	12
1.3.2	Потпуни скупови везника	14
1.3.3	n -арни везници	15
1.4	Минимизација логичких израза	16
1.4.1	Метод алгебарских трансформација	16
1.4.2	Метод Карноових мапа	18
1.4.3	Метод Квин-Мекласког	25
1.4.4	Минимизација у присуству небитних вредности	33
1.4.5	Минимална КНФ форма	36
2	Логичка кола	39
2.1	О логичким колима	39
2.2	Вредност високе импедансе	40
2.3	Логичке капије	41
2.4	Кашњење логичког кола	42
2.5	Имплементација логичких капија у савременим рачунарима	44
2.5.1	НЕ коло	46
2.5.2	НИ и И коло	48
2.5.3	НИЛИ и ИЛИ коло	48
2.5.4	ЕИЛИ коло	48
2.5.5	Бафер	49
2.5.6	Бафер са три стања	50
2.5.7	Пропусни транзистори и преносне капије	51
2.5.8	Бафер са три стања и преносне капије	52
2.5.9	ЕИЛИ коло и преносне капије	52
2.5.10	Вишеулазне логичке капије	53

3	Комбинаторна кола	57
3.1	Основна комбинаторна кола	58
3.1.1	Мултиплексер	58
3.1.2	Демултиплексер	63
3.1.3	Декодер	65
3.1.4	Кодер	66
3.2	Аритметичко-логичка кола	68
3.2.1	Битовске операције	68
3.2.2	Померачи	69
3.2.3	Сабирачи и одузимаачи	70
3.2.4	Компаратори	81
3.2.5	Аритметичко-логичка јединица	83
3.3	Општа комбинаторна кола	85
3.3.1	Неизмењиве меморије	85
3.3.2	PLA кола и PAL кола	87
4	Секвенцијална кола	89
4.1	Резе	92
4.2	Синхрона и асинхрона секвенцијална кола	95
4.3	Флип-флопови	97
4.3.1	SR флип-флоп	98
4.3.2	D флип-флоп	100
4.3.3	JK флип-флоп	101
4.3.4	T флип-флоп	102
4.3.5	Проблем „хватања јединице”	103
4.3.6	Време поставке и време задржавања	105
4.4	Регистри	106
4.5	Меморије	108
4.5.1	Синхроне меморије	109
4.5.2	Конструкција већих меморија помоћу мањих	110
4.5.3	Асинхроне меморије	111
4.5.4	Оптимизација синхроних меморија	113
4.5.5	О произвољном приступу	115
4.5.6	Динамичке меморије	116
4.6	Бројачи	117
4.7	Бројачи са произвољним редоследом стања	120
4.8	Коначни аутомати	123
5	Принцип рада рачунара	127
5.1	Рачунари са фиксираним програмом	128
5.1.1	Програмирање рачунара са фиксираним програмом	134
5.2	Рачунари са ускладиштеним програмом	142
5.2.1	Алгоритам контролне јединице	147
5.2.2	Аутомат контролне јединице	153
5.2.3	Програмирање рачунара са ускладиштеним програмом	153

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*:

¹Формулација 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 пионером 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)
- $x + \bar{x} = 1$, $x \cdot \bar{x} = 0$ (комплементарност)

Изрази над Буловом алгебром називају се *буловски изрази*. Приликом записивања буловских израза подразумевамо да оператор $\bar{}$ 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)
- $\bar{\bar{x}} = x$ (закон двојне негације)
- $\overline{x + y} = \bar{x} \cdot \bar{y}$, $\overline{x \cdot y} = \bar{x} + \bar{y}$ (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 $\bar{}$, \cdot и $+$ дефинисане на начин дат у табели 1.1.

x	\bar{x}	x	y	$x \cdot y$	x	y	$x + y$
0	1	0	0	0	0	0	0
0	1	0	1	0	0	1	1
1	0	1	0	0	1	0	1
1	0	1	1	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 $\bar{}$ 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*

²Приметимо 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 $\bar{}$ на произвољан начин. Изрази могу садржати и заграде којима се може променити уобичајени приоритет оператора (највиши приоритет има негација, затим којункција, 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 $v : P \rightarrow \{0, 1\}$. 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, \bar{y}, z). *Елементарна конјункција* је израз који се састоји из конјункције литерала (нпр. $x\bar{y}z\bar{u}\bar{v}$). За израз кажемо да је

³Напоменимо 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.

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

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

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:

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

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:

$$\bar{\bar{e}} = e \quad \overline{e_1 + e_2} = \bar{e}_1 \cdot \bar{e}_2 \quad \overline{e_1 \cdot e_2} = \bar{e}_1 + \bar{e}_2$$

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 x и \bar{x} , тада је та конјункција еквивалентна са 0 (јер је $x \cdot \bar{x} \cdot e = 0 \cdot e = 0$) 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 $\overline{(x + \overline{y} + z + \overline{wy})} \cdot (\overline{y} + w)$, односно $(x + \overline{y} + \overline{z \cdot \overline{wy}}) \cdot (\overline{y} + w)$, и најзад $(x + \overline{y} + \overline{zwy}) \cdot (\overline{y} + w)$. У трећем кораку примењујемо дистрибутивни закон и добијамо $x\overline{y} + xw + \overline{y}y + \overline{y}w + \overline{zwy}\overline{y} + \overline{zwy}w$. Даљим упрошћавањем добијамо $x\overline{y} + xw + \overline{y} + \overline{y}w + \overline{zwy}$. Најзад, конјункције $x\overline{y}$ и $\overline{y}w$ се могу обрисати, јер дисјункција садржи \overline{y} . Коначни ДНФ је $xw + \overline{y} + \overline{zwy}$.

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 $(x + \overline{y} + \overline{zwy}) \cdot (\overline{y} + w)$. 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, $\overline{x}y\overline{z}\overline{w}$ је савршена елементарна конјункција (у односу на P), а $x\overline{z}$ то није.

За конјунктивну (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 $f : \{0, 1\}^n \rightarrow \{0, 1\}$ које свакој n -торци логичких вредности $(x_1, \dots, x_n) \in \{0, 1\}^n$ придружује логичку вредност $y \in \{0, 1\}$. Записиваћемо и $y = f(x_1, \dots, x_n)$. Логичке променљиве x_1, \dots, 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	$f(x) = \bar{x}$

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	$f(x, y) = \bar{x}$
Негација друге пројекције	$f(x, y) = \bar{y}$
Конјункција	$f(x, y) = x \cdot y$
Дисјункција	$f(x, y) = x + y$
Шеферова функција (НИ) $x \uparrow y$	$f(x, y) = \bar{x}\bar{y} = \overline{x + y}$
Пирсова (Лукашиевичева) функција (НИЛИ) $x \downarrow y$	$f(x, y) = \overline{x + y} = \bar{x} \cdot \bar{y}$
Импликација $x \Rightarrow y$	$f(x, y) = \bar{x} + y$
Импликација $y \Rightarrow x$	$f(x, y) = x + \bar{y}$
Негација импликације $\bar{x} \Rightarrow \bar{y}$	$x\bar{y}$
Негација импликације $\bar{y} \Rightarrow \bar{x}$	$\bar{x}y$
Ексклузивна дисјункција $x \oplus y$	$f(x, y) = x\bar{y} + \bar{x}y$
Еквиваленција	$f(x, y) = xy + \bar{x}\bar{y}$

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 $\bar{x}\bar{y}\bar{z}$). Дакле, ако је вредност променљиве x у тој комбинацији 1, узимамо литерал x , а ако је вредност 0, узимамо литерал \bar{x} (слично и за друге две променљиве).

⁵Будући 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:

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

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 $x + y + \bar{z}$). Дакле, ако је вредност променљиве x у тој комбинацији 0 узимамо литерал x , а ако је вредност 1, узимамо литерал \bar{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:

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

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 $C = \{., +, \bar{\cdot}\}$ (тј. скуп основних логичких везника) није минималан потпун скуп везника, јер је и његов прави подскуп $C' = \{., \bar{\cdot}\}$ 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 $C^+ = \{+, \bar{\cdot}\}$ потпун скуп везника. Са друге стране, скупови C' 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 потпун 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 $\bar{x} = \overline{x \cdot x} = x \uparrow x$ и $x \cdot y = \overline{\overline{x \cdot y}} = \overline{\overline{x} \cdot \overline{y}} = (x \uparrow y) \uparrow (x \uparrow y)$ 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: $x_1 \cdot x_2 \cdot x_3 \cdot \dots \cdot x_n \equiv (\dots((x_1 \cdot x_2) \cdot x_3) \cdot \dots) \cdot x_n$. Може се показати да ће n -арна конјункција дати вредност 1 ако су сви x_i једнаки 1.
- n -арна дисјункција: $x_1 + x_2 + x_3 + \dots + x_n \equiv (\dots((x_1 + x_2) + x_3) + \dots) + x_n$. Може се показати да ће n -арна дисјункција дати вредност 1 ако је бар једно x_i једнако 1.
- n -арна ексклузивна дисјункција: $x_1 \oplus x_2 \oplus x_3 \oplus \dots \oplus x_n \equiv (\dots((x_1 \oplus x_2) \oplus x_3) \oplus \dots) \oplus x_n$. Може се показати да ће n -арна ексклузивна дисјункција дати вредност 1 ако је непаран број вредности x_i једнако 1.
- n -арни Шеферов (НИ) везник: $x_1 \uparrow \dots \uparrow x_n \equiv \overline{x_1 \cdot \dots \cdot x_n}$. Јасно је да ће n -арни НИ везник дати вредност 1 ако бар једно x_i има вредност 0.
- n -арни Пирсов (НИЛИ) везник: $x_1 \downarrow \dots \downarrow x_n \equiv \overline{x_1 + \dots + x_n}$. Лако се види да ће n -арни НИЛИ везник дати вредност 1 ако су сви x_i једнаки 0.

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

⁶И 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' могу представити помоћу везника \uparrow , с обзиром да је скуп C' 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 $xyzi$ 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 $x \uparrow y \uparrow z \neq (x \uparrow y) \uparrow z$, као и да $x \uparrow y \uparrow z \neq x \uparrow (y \uparrow z)$. 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.

⁷Други 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 $\bar{x}K$, 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 $xK + \bar{x}K = (x + \bar{x}) \cdot K = 1 \cdot K = K$. Овај корак зовемо *груписање* елементарних конјункција. На пример, ако имамо конјункције $xy\bar{z}$ и $x\bar{y}\bar{z}$, тада груписањем од ове две конјункције добијамо једну конјункцију $x\bar{z}$ (дакле, уклањамо променљиву по којој се разликују, а задржавамо оно што им је заједничко).
- уколико у ДНФ-у једну те исту конјункцију 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:

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

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

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

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

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

одакле следи:

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

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:

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

Приметимо да је у питању израз сличан изразу у претходном примеру – једина разлика је у још једној додатној конјункцији $\bar{x}y\bar{z}$. 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:

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

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

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

одакле следи:

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

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

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

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.

	$\bar{x}\bar{y}$	$\bar{x}y$	xy	$x\bar{y}$
\bar{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

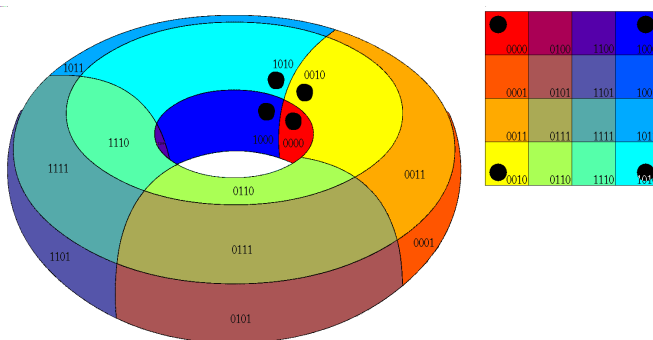
вредност променљиве z фиксирана. По вертикали се мења само вредност променљиве z . Другим речима, свака два суседна поља мапе (по вертикали или хоризонтално) се разликују само по вредности једне променљиве. Приметимо, притом, да се прво и последње поље произвољне врсте такође разликују само по једној 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.

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

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 хоризонтално или вертикално) 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 торуца, док спој леве и десне ивице мапе чини попречни пресек торуца (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: $\bar{x}\bar{y}\bar{z}\bar{u}$, $\bar{x}\bar{y}zu$, $\bar{x}y\bar{z}\bar{u}$, $\bar{x}y\bar{z}u$. Груписањем прве две (лева два поља тог квадрата) и друге две (десна два поља тог квадрата) добијемо конјункције $\bar{x}\bar{y}\bar{z}$ и $\bar{x}y\bar{z}$. Ове две конјункције су такође „суседне“, јер се разликују само по променљивој y , па њиховим груписањем добијемо $\bar{x}\bar{z}$. Литерали \bar{x} и \bar{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:

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

(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

	$\bar{x}\bar{y}$	$\bar{x}y$	xy	$x\bar{y}$
\bar{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.

	$\bar{x}\bar{y}$	$\bar{x}y$	xy	$x\bar{y}$
\bar{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 $\bar{x}\bar{y}$, левом хоризонталном заокруживању одговара конјункција $\bar{x}z$, док заокруживању које групише крајње јединице друге врсте одговара конјункција $\bar{y}z$. Отуда је минимални ДНФ израз:

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

Пример 6. Посматрајмо функцију дату у табели 1.6.

Овој функцији одговара савршена ДНФ:

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

(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

	$\bar{x}y$	$\bar{x}\bar{y}$	xy	$x\bar{y}$
\bar{z}	1	1	0	0
z	1	1	0	1

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:

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

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:

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

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

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

Слика 1.7: Карноова мапа за пример 7

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

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 табели одговара Карноова мапа на слици 1.9.

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

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

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

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

Слика 1.10: Прво решење примера 8

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

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

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

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

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 два метода, али је погоднија за аутоматизацију, тј. имплементацију у рачунару. Такође, примењива је на функције произвољног реда.⁸

Идеја алгорита 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 смањило 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 конјункција. 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 алгорита 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 $i = 0, 1, \dots, n - 1$). 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 алгорита 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 импликанти* који се преносе у другу фазу алгорита.

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

⁸Ипак, 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:

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

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

0	$\bar{x}\bar{y}\bar{z}$
1	$\bar{x}\bar{y}z$
2	$\bar{x}yz$
	$x\bar{y}z$

Груписањем у првој итерацији добијамо:

0	$\bar{x}\bar{y}\bar{z}\checkmark$	$\bar{x}\bar{y}$
1	$\bar{x}\bar{y}z\checkmark$	$\bar{x}z$
		$\bar{y}z$
2	$\bar{x}yz\checkmark$	
	$x\bar{y}z\checkmark$	

Симболом \checkmark 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:

	$\bar{x}\bar{y}\bar{z}$	$\bar{x}\bar{y}z$	$\bar{x}yz$	$x\bar{y}z$
$\bar{x}\bar{y}$	+	+		
$\bar{x}z$		+	+	
$\bar{y}z$		+		+

Дакле, за сваку врсту, симболом \oplus 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).

	$\bar{x}\bar{y}\bar{z}$	$\bar{x}\bar{y}z$	$\bar{x}yz$	$x\bar{y}z$
$\bar{x}\bar{y}$	\oplus	$+$		
$\bar{x}z$		$+$	\oplus	
$\bar{y}z$		$+$		\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):

	$\bar{x}\bar{y}\bar{z}$	$\bar{x}\bar{y}z$	$\bar{x}yz$	$x\bar{y}z$
$\bar{x}\bar{y}$	\oplus	\boxplus		
$\bar{x}z$		\boxplus	\oplus	
$\bar{y}z$		\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:

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

Пример 10. Размотримо сада израз из примера 4:

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

У првој фази имамо:

0	$\bar{x}\bar{y}\bar{z}\checkmark$	$\bar{x}\bar{y}\checkmark$ $\bar{x}\bar{z}\checkmark$	\bar{x}
1	$\bar{x}\bar{y}z\checkmark$ $\bar{x}y\bar{z}\checkmark$	$\bar{x}z\checkmark$ $\bar{y}z$ $\bar{x}y\checkmark$	
2	$\bar{x}yz\checkmark$ $x\bar{y}z\checkmark$		

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

	$\bar{x}\bar{y}\bar{z}$	$\bar{x}\bar{y}z$	$\bar{x}yz$	$x\bar{y}z$	$\bar{x}y\bar{z}$
\bar{x}	\oplus	\boxplus	\oplus		\oplus
$\bar{y}z$		\boxplus		\oplus	

Како нема непокривених колона, минимални ДНФ је:

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

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

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

у првој фази имамо:

0	$\bar{x}\bar{y}\bar{z}\bar{u}\checkmark$	$\bar{x}\bar{y}\bar{u}\checkmark$ $\bar{x}\bar{z}\bar{u}\checkmark$ $\bar{y}\bar{z}\bar{u}\checkmark$	$\bar{x}\bar{u}$ $\bar{y}\bar{u}$ $\bar{z}\bar{u}$
1	$\bar{x}\bar{y}z\bar{u}\checkmark$ $\bar{x}y\bar{z}\bar{u}\checkmark$ $x\bar{y}\bar{z}\bar{u}\checkmark$	$\bar{x}z\bar{u}\checkmark$ $\bar{y}z\bar{u}\checkmark$ $\bar{x}y\bar{z}\checkmark$ $\bar{x}y\bar{u}\checkmark$ $y\bar{z}\bar{u}\checkmark$ $x\bar{y}\bar{u}\checkmark$ $x\bar{z}\bar{u}\checkmark$	$y\bar{z}$
2	$\bar{x}y\bar{z}u\checkmark$ $\bar{x}yz\bar{u}\checkmark$ $x\bar{y}z\bar{u}\checkmark$ $xy\bar{z}\bar{u}\checkmark$	$y\bar{z}u\checkmark$ $xyz\checkmark$	
3	$xyz\bar{u}\checkmark$		

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

	$\bar{x}\bar{y}\bar{z}\bar{u}$	$\bar{x}\bar{y}z\bar{u}$	$\bar{x}y\bar{z}\bar{u}$	$\bar{x}yz\bar{u}$	$\bar{x}y\bar{z}u$	$\bar{x}yzu$	$x\bar{y}\bar{z}\bar{u}$	$x\bar{y}z\bar{u}$	$xy\bar{z}\bar{u}$	$xyz\bar{u}$
$\bar{x}\bar{u}$	⊕	⊕	⊕			⊕				
$\bar{y}\bar{u}$	⊕	⊕					⊕	⊕		
$\bar{z}\bar{u}$	+		+				+		+	
$y\bar{z}$			⊕	⊕					⊕	⊕

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

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

Приметимо да је конјункција $\bar{z}\bar{u}$ 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:

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

У првој фази имамо:

0	$\bar{x}\bar{y}\bar{z}\bar{u}\checkmark$	$\bar{x}\bar{y}\bar{u}$ $\bar{y}\bar{z}\bar{u}$
1	$\bar{x}\bar{y}z\bar{u}\checkmark$ $x\bar{y}\bar{z}\bar{u}\checkmark$	$\bar{x}\bar{y}z$ $\bar{x}z\bar{u}$ $x\bar{y}\bar{z}$
2	$\bar{x}\bar{y}z\bar{u}\checkmark$ $\bar{x}y\bar{z}u$ $\bar{x}yz\bar{u}\checkmark$ $x\bar{y}\bar{z}u\checkmark$	

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:

	$\bar{x}\bar{y}\bar{z}\bar{u}$	$\bar{x}\bar{y}z\bar{u}$	$\bar{x}\bar{y}z\bar{u}$	$\bar{x}y\bar{z}u$	$\bar{x}yz\bar{u}$	$x\bar{y}\bar{z}\bar{u}$	$x\bar{y}\bar{z}u$
$\bar{x}y\bar{z}u$				\oplus			
$\bar{x}\bar{y}\bar{u}$	+	+					
$\bar{y}\bar{z}\bar{u}$	+					+	
$\bar{x}\bar{y}z$		\boxplus	\oplus				
$\bar{x}z\bar{u}$		\boxplus			\oplus		
$x\bar{y}\bar{z}$						\boxplus	\oplus

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

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

и

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

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 скуп⁹ 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

⁹Кад кажемо најмањи могући, мислимо скуп простих импликаната 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 $D_j = p_{i_1} + p_{i_2} + \dots + p_{i_{k_j}}$, при чему су $p_{i_1}, p_{i_2}, \dots, p_{i_{k_j}}$ логичке променљиве које одговарају врстама чији прости импликанти покривају j -ту савршену конјункцију. Сада формирамо КНФ формулу $K = D_1 \cdot D_2 \cdot \dots \cdot D_m$, 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 ако 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	$\bar{x}\bar{y}\bar{z}\bar{u}\checkmark$	$\bar{x}\bar{y}\bar{z}$ $\bar{x}\bar{z}\bar{u}$
1	$\bar{x}\bar{y}zu\checkmark$ $\bar{x}y\bar{z}\bar{u}\checkmark$	$\bar{y}zu$ $\bar{x}y\bar{u}$ $y\bar{z}\bar{u}$
2	$\bar{x}yz\bar{u}\checkmark$ $x\bar{y}zu\checkmark$ $xy\bar{z}\bar{u}\checkmark$	$\bar{x}yz$ xzu $xy\bar{z}$
3	$\bar{x}yzu\checkmark$ $xyzu\checkmark$	

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:

	$\bar{x}\bar{y}\bar{z}\bar{u}$	$\bar{x}\bar{y}\bar{z}u$	$\bar{x}y\bar{z}\bar{u}$	$\bar{x}yz\bar{u}$	$\bar{x}yzi$	$x\bar{y}\bar{z}u$	$xyz\bar{u}$	$xyzi$
$\bar{x}\bar{y}\bar{z}$	+	+						
$\bar{x}\bar{z}\bar{u}$	+		+					
$\bar{y}\bar{z}u$		+				+		
$\bar{x}y\bar{u}$			+	+				
$y\bar{z}\bar{u}$			+				+	
$\bar{x}yz$				⊕	⊕			
$x\bar{z}u$						+		+
$xy\bar{z}$							+	+

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:

		$\bar{x}\bar{y}\bar{z}\bar{u}$	$\bar{x}\bar{y}\bar{z}u$	$\bar{x}y\bar{z}\bar{u}$	$\bar{x}yz\bar{u}$	$x\bar{y}\bar{z}u$	$xyz\bar{u}$	$xyzi$
p_1	$\bar{x}\bar{y}\bar{z}$	+	+					
p_2	$\bar{x}\bar{z}\bar{u}$	+		+				
p_3	$\bar{y}\bar{z}u$		+			+		
p_4	$\bar{x}y\bar{u}$			+				
p_5	$y\bar{z}\bar{u}$			+			+	
p_6	$x\bar{z}u$					+		+
p_7	$xy\bar{z}$						+	+

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 подскупе 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 $\{\bar{x}\bar{y}\bar{z}, y\bar{z}\bar{u}, x\bar{z}u\}$, а другој $\{\bar{x}\bar{z}\bar{u}, \bar{y}\bar{z}u, xy\bar{z}\}$. Оба скупа имају укупно по 9 литерала, па је отуда свеједно који ћемо изабрати. Отуда је једна минимална ДНФ форма наше функције:

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

а друга:

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

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

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:

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

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

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:

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

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

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	$\bar{x}\bar{y}\bar{z}u\checkmark$ $\bar{x}y\bar{z}\bar{u}\checkmark$ $x\bar{y}\bar{z}\bar{u}\checkmark$	$\bar{x}\bar{y}u\checkmark$ $\bar{x}\bar{z}u\checkmark$ $\bar{y}\bar{z}u\checkmark$ $\bar{x}y\bar{z}\checkmark$ $\bar{x}y\bar{u}\checkmark$ $y\bar{z}\bar{u}\checkmark$ $x\bar{y}\bar{z}\checkmark$ $x\bar{z}\bar{u}\checkmark$	$\bar{x}u$ $\bar{z}u$ $\bar{x}y\checkmark$ $y\bar{z}\checkmark$ $y\bar{u}\checkmark$ $x\bar{z}$	y
2	$\bar{x}\bar{y}zu\checkmark$ $\bar{x}y\bar{z}u\checkmark$ $\bar{x}yz\bar{u}\checkmark$ $x\bar{y}\bar{z}u\checkmark$ $xy\bar{z}\bar{u}\checkmark$	$\bar{x}zu\checkmark$ $\bar{x}yu\checkmark$ $y\bar{z}u\checkmark$ $\bar{x}yz\checkmark$ $yz\bar{u}\checkmark$ $x\bar{z}u\checkmark$ $xy\bar{z}\checkmark$ $xy\bar{u}\checkmark$	$yu\checkmark$ $yz\checkmark$ $xy\checkmark$	
3	$\bar{x}yzu\checkmark$ $xy\bar{z}u\checkmark$ $xyz\bar{u}\checkmark$	$yzu\checkmark$ $xyu\checkmark$ $xyz\checkmark$		
4	$xyzu\checkmark$			

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

	$\bar{x}\bar{y}zu$	$\bar{x}y\bar{z}u$	$xyz\bar{u}$	$xyz\bar{u}$
$\bar{x}u$	\oplus	\boxplus		
$\bar{z}u$		$+$		
$x\bar{z}$			$+$	
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 $\bar{x}u$. Како они покривају све колоне табеле, минимални ДНФ је:

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

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 \bar{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 \bar{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, \dots, x_n) и m излаза (y_1, y_2, \dots, 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 времена¹, тј. $x_i = x_i(t)$ ($y_j = y_j(t)$). Ово најчешће нећемо посебно наглашавати, када говоримо о вредностима улаза и излаза у

¹Притом, 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, \dots . 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.²³ 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.

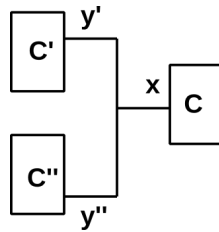
Да бисмо формално дефинисали понашање „искључених” излаза, уводимо тзв. *вредност високе импедансе*⁴. Ову вредност означаваћемо са **Z** и сматраћемо да је имају сви излази који су искључени, тј. који не производе никакву вредност. Уколико на неком излазу имамо вредност **Z**,

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.

²³За 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.

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

⁴Назив 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


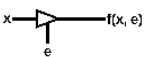

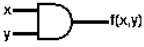
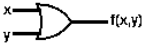
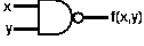
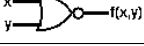
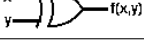

2.3 Логичке капије

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,

⁵Неформално, 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 улазу 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. <i>buffer</i>)	$f(x) = x$	
Бафер са три стања (енгл. <i>three-state buffer</i>)	$f(x, e) = \begin{cases} x, & \text{за } e = 1 \\ \mathbf{Z}, & \text{за } e = 0 \end{cases}$	
НЕ коло (енгл. <i>NOT</i>)	$f(x) = \bar{x}$	
И коло (енгл. <i>AND</i>)	$f(x, y) = x \cdot y$	
ИЛИ коло (енгл. <i>OR</i>)	$f(x, y) = x + y$	
НИ коло (енгл. <i>NAND</i>)	$f(x, y) = x \uparrow y$	
НИЛИ коло (енгл. <i>NOR</i>)	$f(x, y) = x \downarrow y$	
ЕИЛИ коло (енгл. <i>XOR</i>)	$f(x, y) = x \oplus y$	
НЕИЛИ коло (енгл. <i>XNOR</i>)	$f(x, y) = x \sim y$	

Табела 2.1: Логичке капије

тада ће и на излазу бити јединица, чак и да је други улаз једнак \mathbf{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 \mathbf{Z} .

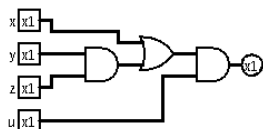
Подсетимо се да се бинарни логички везници могу уопштити и посматрати као n -арни везници (одељак 1.3.3). Отуда уместо двоулазних можемо разматрати и вишеулазне логичке капије које имплементирају семантику одговарајућих n -арних везника.

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 дубини⁶ 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 интеграције⁸ у последњих неколико деценија довело до драматичног убрзавања рада савремених рачунара.

⁶Под дубином израза подразумевамо висину синтаксног стабла тог израза.

⁷Приметимо 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.

⁸Степен 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

⁹У 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.

2.5. ИМПЛЕМЕНТАЦИЈА ЛОГИЧКИХ КАПИЈА У САВРЕМЕНИМ РАЧУНАРИМА 45

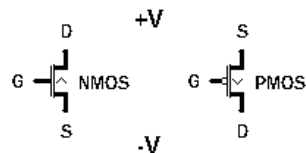
транзистор може функционисати као прекидач. Постоје две основне врсте транзистора: *биполарни* и *униполарни*. Савремени рачунари су углавном засновани на једној посебној врсти униполарних транзистора – на *MOS транзисторима*.¹⁰

MOS транзистори представљају полупроводничку компоненту која има три прикључка: *сорс*, *дрејн* и *гејт*. По свом понашању, MOS транзистор функционише као прекидач – струја може протичати 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

¹⁰Енглеска 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 транзисторе, јер њихов сорс мора бити повезан на масу.

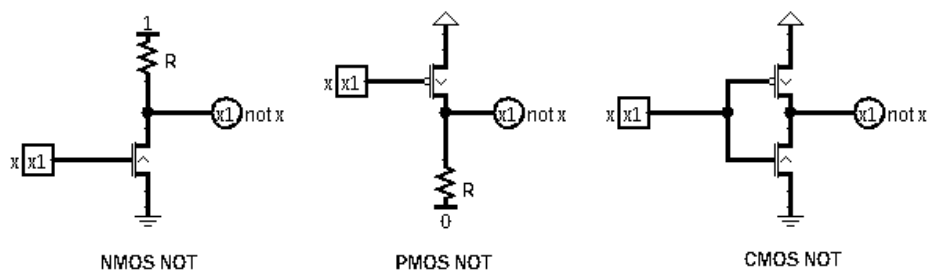


Слика 2.5: Општа структура логичке капије

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.

2.5. ИМПЛЕМЕНТАЦИЈА ЛОГИЧКИХ КАПИЈА У САВРЕМЕНИМ РАЧУНАРИМА 47

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 скупу CMOS технологију која ове проблеме нема.

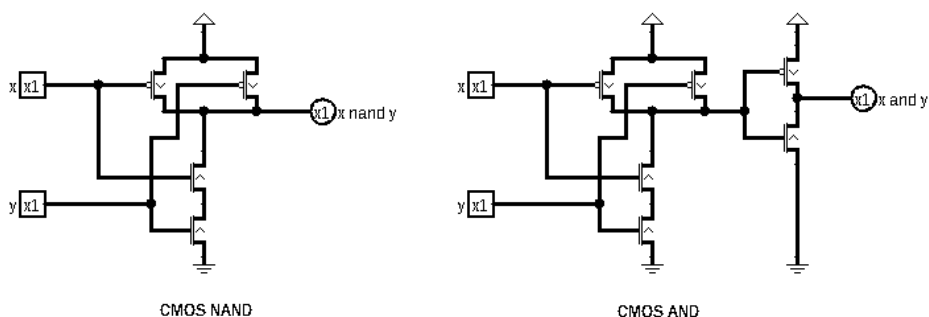
CMOS имплементација НЕ кола је дата на трећој слици. Како су нам сада доступни и NMOS и PMOS транзистори, отпорници више нису потребни. У случају логичке нуле на улазу доњи транзистор је затворен, а горњи отворен, па имамо везу излаза са напајањем (логичка јединица). У случају логичке јединице на улазу 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

¹¹Овај 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.

¹²Заправо, 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

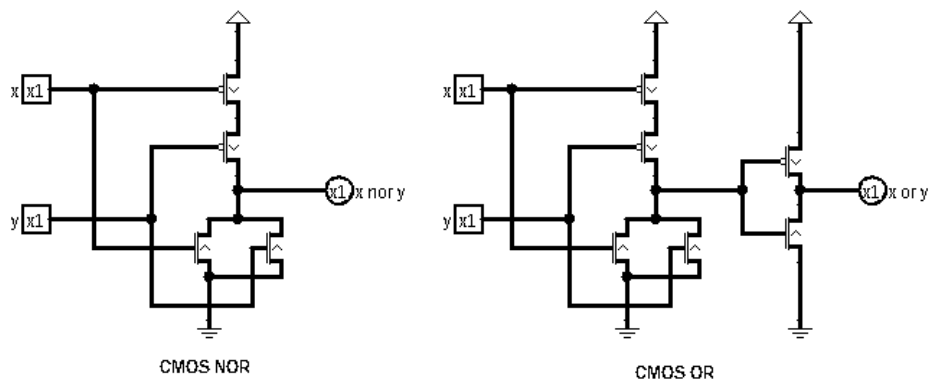
Десна слика приказује И коло у CMOS технологији. Оно се добија надовезивањем једне негације (НЕ кола) на излаз НИ кола. Дакле, иако ми интуитивно НИ доживљавамо као „негирано И”, у CMOS технологији је заправо једноставније направити НИ коло, док се И коло представља као „негирано НИ”.

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 кола. 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 \bar{x} , а друга \bar{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

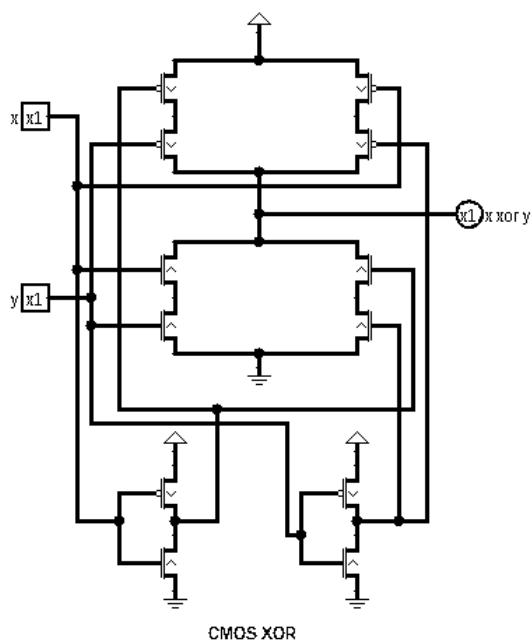
горње мреже повезани су на y и \bar{x} , док су транзистори у десној грани горње мреже повезани на x и \bar{y} . Отуда горња мрежа проводи струју ако је или $x = 1, y = 0$ (лева грана) или $x = 0, y = 1$ (десна грана). У доњој мрежи ситуација је комплементарна: транзистори леве гране су повезани на x и y , док су транзистори десне гране повезани на \bar{x} и \bar{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

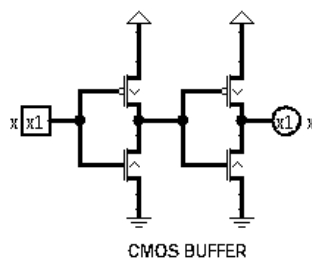
Приметимо да постојање бафера, иако нема логичког ефекта, уводи додатно кашњење. Ово кашњење је резултат тога што је потребно извесно време да се вредност са улаза бафера пропагира на излаз (тј. да прође кроз две негације). Иако се кашњење углавном посматра као негативна појава, има ситуација када је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

¹³Уобичајено 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.



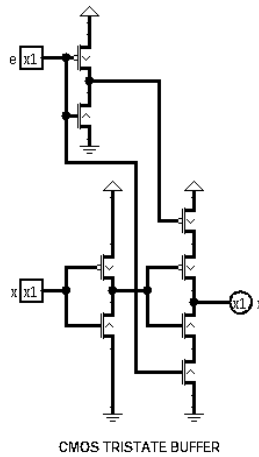
Слика 2.10: Бафер у CMOS-у

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

2.5. ИМПЛЕМЕНТАЦИЈА ЛОГИЧКИХ КАПИЈА У САВРЕМЕНИМ РАЧУНАРИМА 51

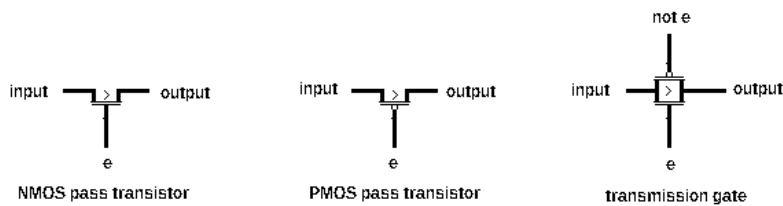
су оба спољашња транзистора потпуно отворена, те се унутрашњи пар транзистора понаша као инвертер и даје на излазу x .



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 \mathbf{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 пропуштена 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 \bar{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 \bar{e} – у супротном, неопходно је увести још једно НЕ коло које ће да инвертује контролни улаз.

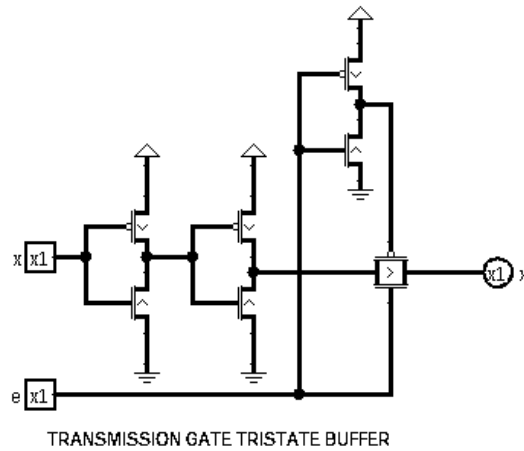
2.5.8 Бафер са три стања и преносне капије

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.

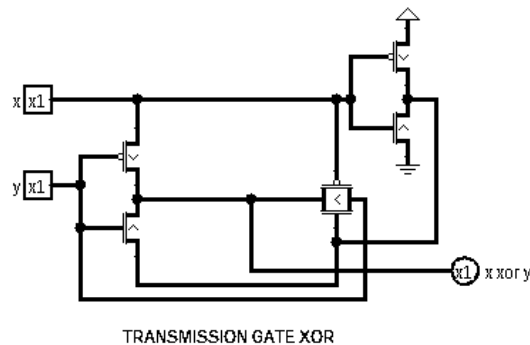
2.5.9 ЕИЛИ коло и преносне капије

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 $x \oplus y = \bar{y}$, па на излаз треба пропустити инвертовано y . У горњем десном углу шеме налази се једно НЕ коло који производи \bar{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 \bar{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 транзистора доводи \bar{x} .

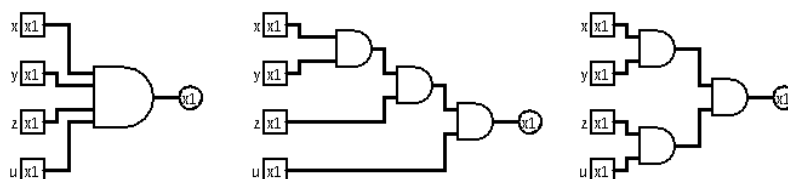


Слика 2.14: ЕИЛИ коло реализовано помоћу преносне капије

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 свде на бинарне везнике истог типа, груписањем подизраза (одељак 1.3.3). Груписање се може

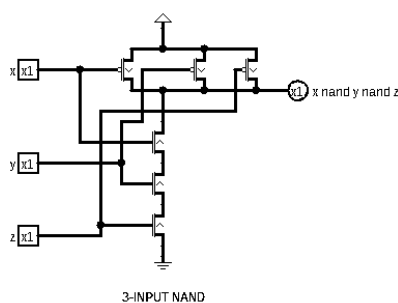
извршити на произвољан начин, имајући у виду закон асоцијативности који важи за бинарне везнике конјункције, дисјункције и ексклузивне дисјункције. На пример, израз $x \cdot y \cdot z$ је еквивалентан изразу $(x \cdot y) \cdot z$, па се ова конјункција три вредности увек може реализовати надовезивањем 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 $x y z u$ 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 изрази 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.



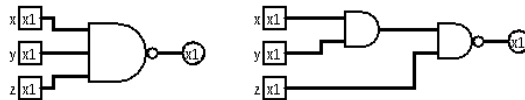
Слика 2.16: Директна реализација четвороулазне конјункције у CMOS-у

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 $x \uparrow y \uparrow z = \overline{xyz} \neq \overline{xy}z = (x \uparrow y) \uparrow z$). 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.

¹⁴Осим 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. 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 хијерархијски¹ – 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 излазима кола поистовећивати са бинарним цифрама, па ћемо их често

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

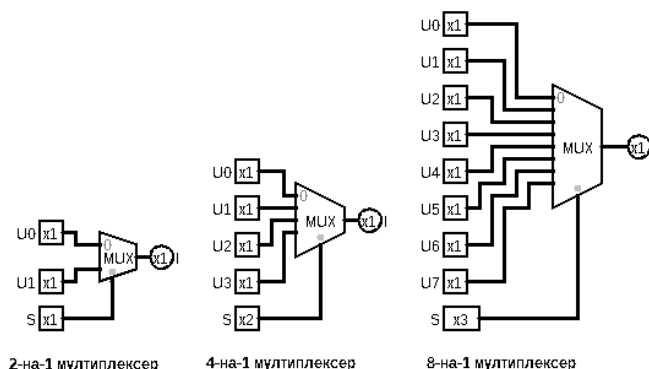
називати и *битовима*. Такође, поједине групе улаза (или излаза) ћемо често посматрати као целине, тј. као вишебитне бинарне бројеве. Такве улазе и излазе ћемо називати *вишебитним*.

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

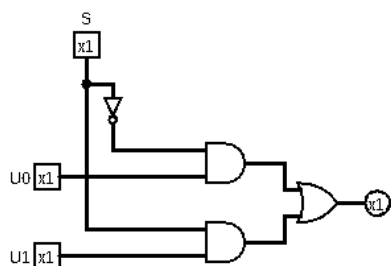
Селекционе улазе 2^k -на-1 мултиплексера можемо посматрати као један k -битни цео број који представља индекс улаза који желимо да проследимо на излаз. На пример, ако у случају 8-на-1 мултиплексера на селекционим улазима имамо комбинацију 010, тада ће се вредност улаза U_2 проследити на излаз.

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

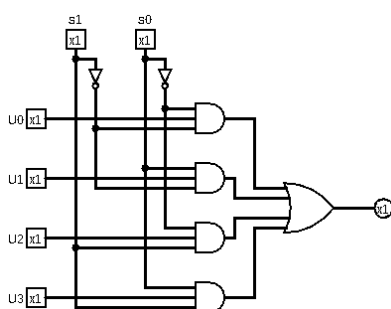
Уколико је $S = 0$, тада ће горња конјункција на свом излазу имати вредност U_0 , док ће доња конјункција на излазу имати 0. Отуда ће вредност на излазу дисјункције бити једнака U_0 . Ако је $S = 1$, тада ће горња конјункција на свом излазу имати вредност 0, док ће доња конјункција имати вредност U_1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

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

Овога пута имамо троулазне конјункције уместо двоулазних. Сваку од конјункција активира одговарајућа комбинација селекционих улаза. На



Слика 3.2: Имплементација 2-на-1 мултиплексера



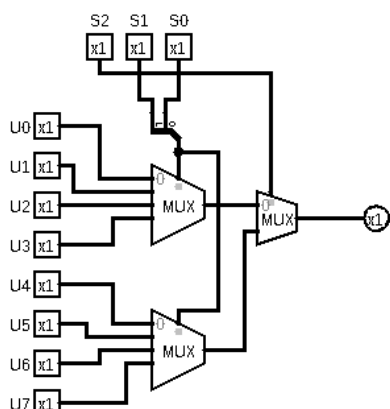
Слика 3.3: Имплементација 4-на-1 мултиплексера

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 сведе 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 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



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

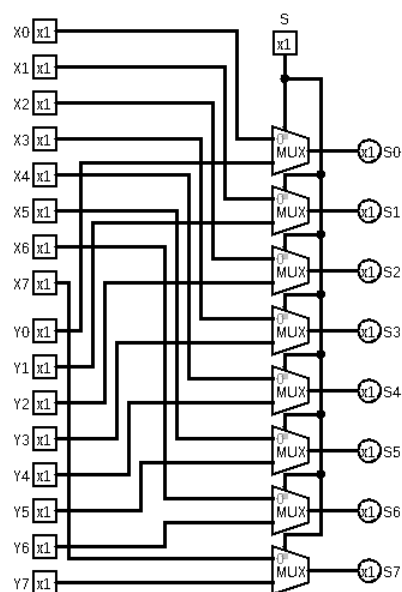
се врши у два корака: најпре се бира група од четири улаза, а затим се из те групе бира један од улаза. Овај приступ се може примењивати на више нивоа, чиме се могу добити још сложенији мултиплексери.

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.

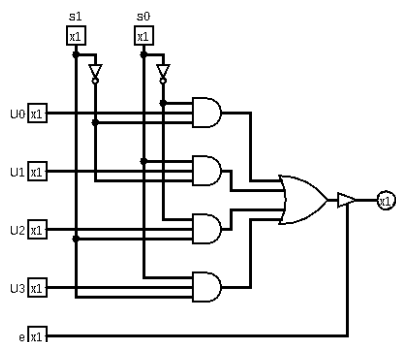
Улази x_0 до x_7 чине један осмобитни улаз, док улази y_0 до y_7 чине други осмобитни улаз. Улази x_i и y_i се прослеђују на улазе i -тог мултиплексера. Сви мултиплексери су контролисани од стране истог селекционог улаза S .



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

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 \mathbf{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.



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

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 ако није, биће вредност израза E_1 . Дакле, овакво коло заправо израчунава C -израз $(C \ ? \ E_2 : E_1)$. 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

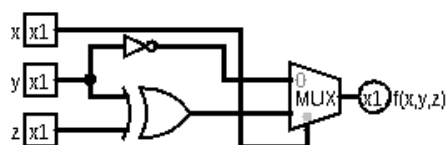
Табела 3.1: Функција реда 3

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 $f_0(y, z) = \bar{y}$, 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).

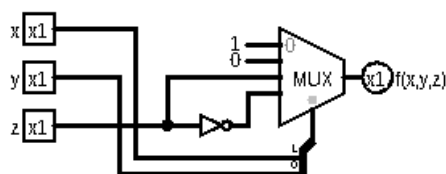
Други начин је да функцију f посматрамо овако:



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 $f_{11}(z) = \bar{z}$. 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} .



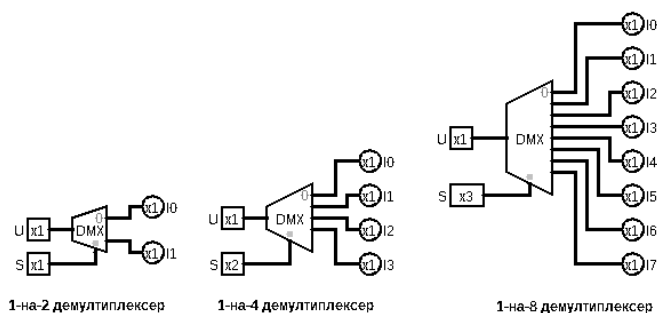
Слика 3.8: Декомпозиција функције реда 3 на четири функције реда 1

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.

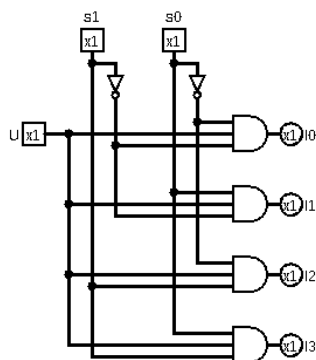
3.1.2 Демултиплексер

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.



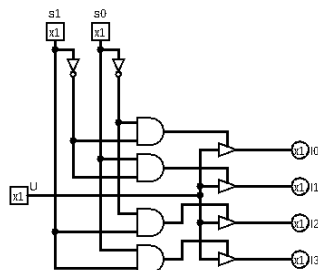
Слика 3.9: Шематске ознаке демултиплексера



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

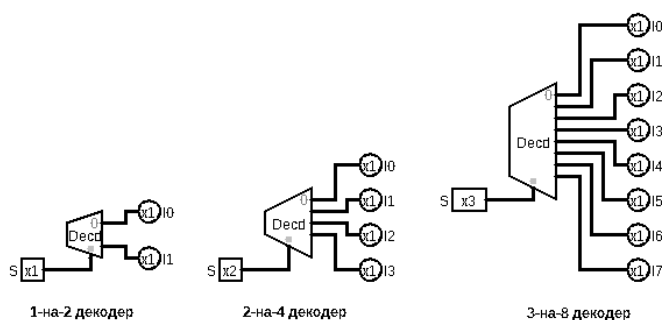
У овој имплементацији се користе бафери са три стања. Свака комбинација на селекционим улазима даје јединицу на одговарајућој конјункцији која активира одговарајући бафер, чиме се пропушта улаз на жељени излаз. Остали бафери су искључени и на њиховим излазима се налази **Z**.

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 сигналимa 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.

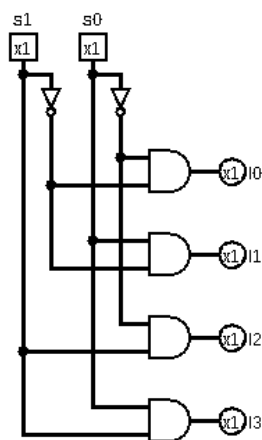


Слика 3.12: Шематске ознаке декодера

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



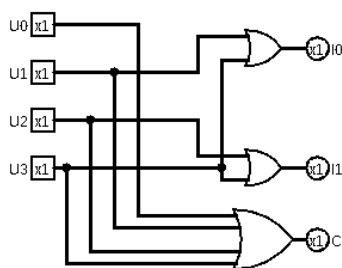
Слика 3.13: Пример имплементације 2-на-4 декодера

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 $(R_0, R_1, \dots, R_{15})$ 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

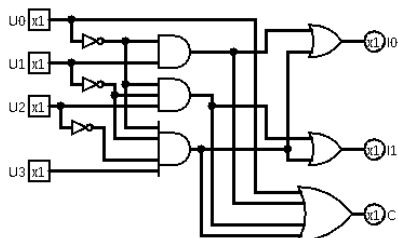
Кодер (енгл. *encoder*) је коло које врши обрнуту функцију од декодера. Кодер има 2^k улаза и k излаза (такав кодер називамо 2^k -на- k кодер). Претпоставка је да је у сваком тренутку највише један од улаза једнак 1, док су остали 0. Излази кодера се могу тумачити као k -битни бинарни број чија је вредност 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 $I_1 I_0$ 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 одреди индекс уређаја који је захтевао прекид, а који има највиши приоритет.

3.2 Аритметичко-логичка кола

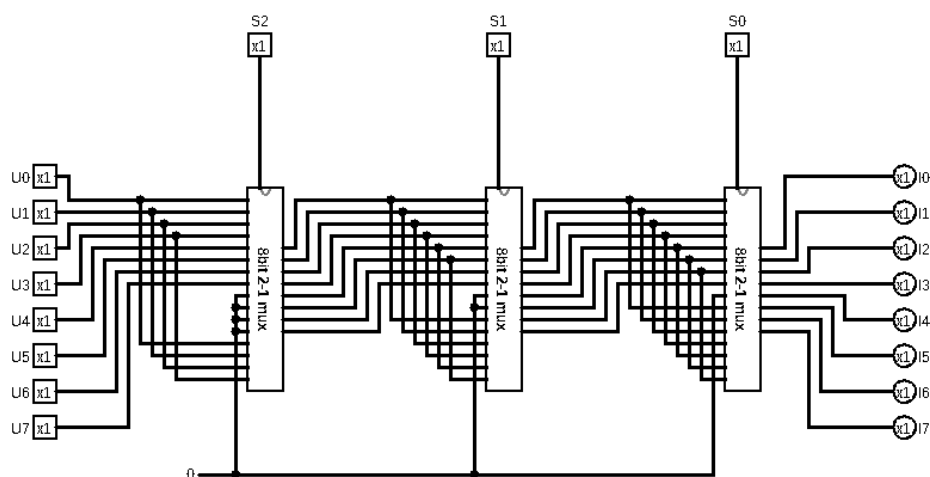
3.2.1 Битовске операције

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 мултиплексера 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 податка \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 сабере 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.

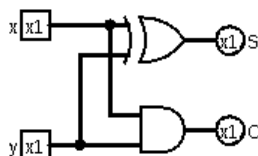
Једнобитни сабирач

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

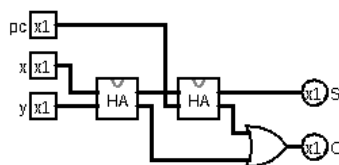
Да бисмо имплементирали полусабирач, посматрајмо најпре његову функцију задату таблично (табела 3.2). Дакле, полусабирач сабира два бита x и y и као резултат даје њихов збир S и пренос C (који се може разумети и као виши бит збира). Из таблице се види да је излаз $S = x \oplus y$, док је $C = x \cdot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

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

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

Слика 3.19: Карноова мапа функције S

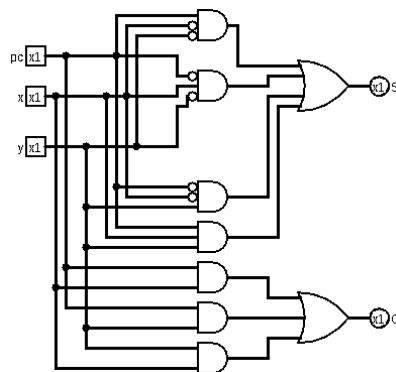
Карноова мапа функције C дата је на слици 3.20. Из ове мапе добијамо израз:

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

Имплементација² потпуног сабирача према добијеним логичким изразима дата је на слици 3.21.

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

	$\bar{x}\bar{y}$	$\bar{x}y$	xy	$x\bar{y}$
$\bar{p}c$	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 закључак³ да је исплативије користити имплементацију засновану на надовезивању два полусабирача.

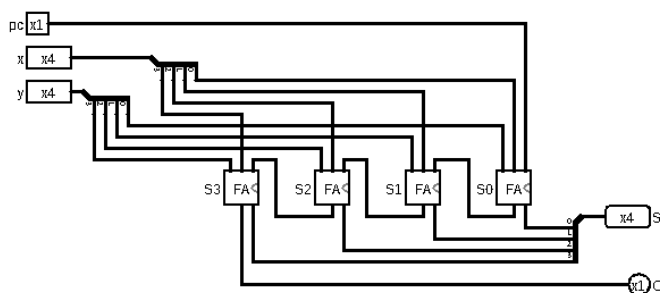
Вишебитни сабирач

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

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

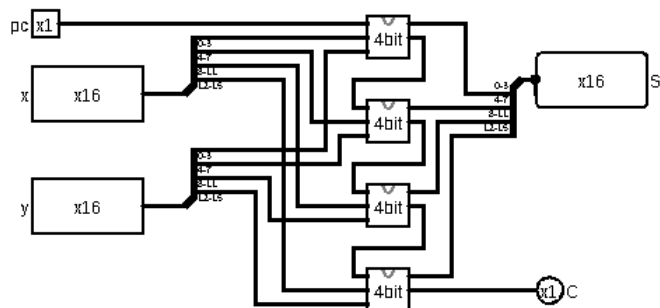
³Овај закључак важи под наведеним претпоставкама о кашњењима гејтова, тј. да сви двоулазни гејтови имају једнако кашњење, као и да се вишеулазни гејтови реализују балансираним уланчавањем двоулазних. Уколико ово није случај, закључак може бити другачији. На пример, ако технологија не подржава директну 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 пропацијом 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 пропације преноса у облику таласа често назива и *таласаста 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 израде као непосредне функције од улаза кола. Нека су C_0, C_1, \dots, C_{n-1} редом преноси на излазима сваког од n потпуних сабирача. Тада 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, пропација 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

⁴Приметимо 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 :

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

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 ($P_1 G_0$), или пренос постоји на pc улазу, а пропагира се кроз позиције 0 и 1 ($P_1 P_0 pc$). Слично, имамо:

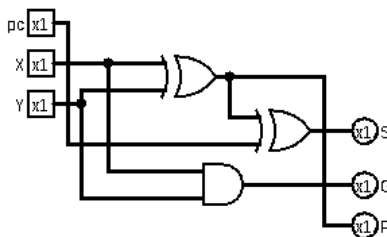
$$C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 pc$$

као и:

$$C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 pc$$

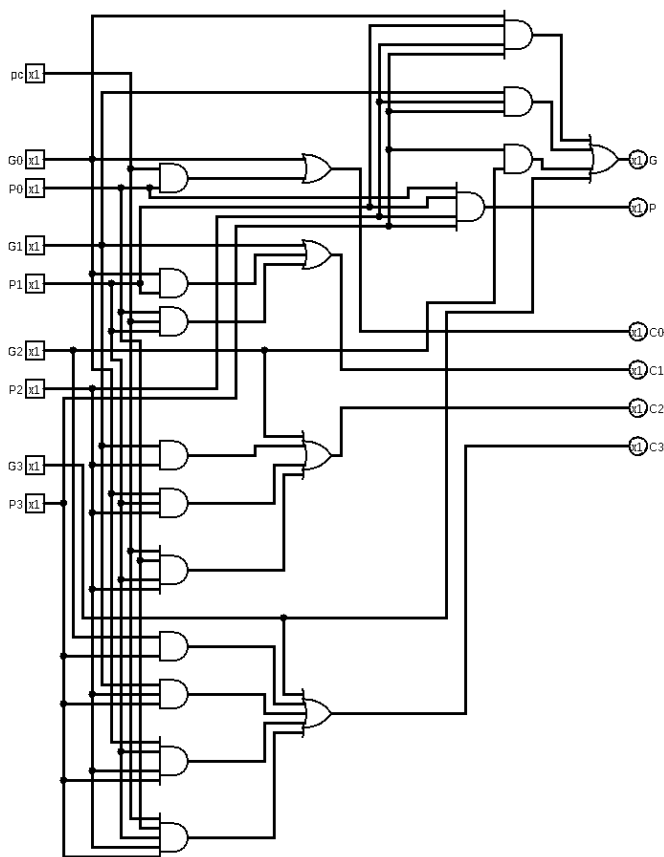
Приметимо да вредности G_i и P_i зависе само од x_i и y_i , па се (за свако $i = 0, 1, 2, 3$) могу израчунати са кашњењем 1Δ , где је Δ кашњење једне двоулазне капије. Отуда се вредност C_i може израчунати са укупним кашњењем $(1 + 2 \cdot \lceil \log_2(i + 2) \rceil) \cdot \Delta$ (имамо И кола са највише $i + 2$ 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.

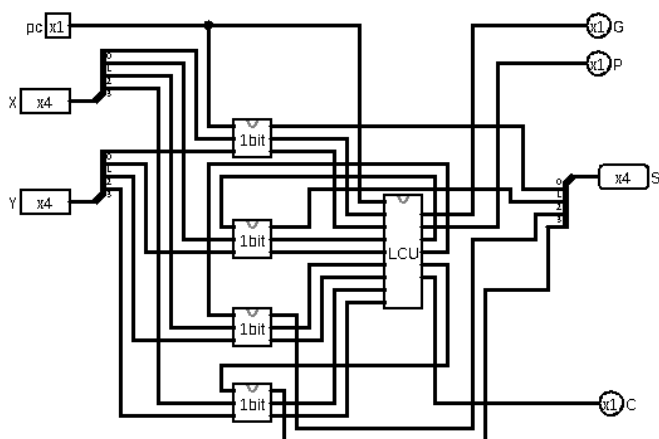


Слика 3.25: Имплементација LCU јединице

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 $(1+2+\dots+i+1)+i+1 = (i+1)\cdot(i+2)/2+i+1 = (i+1)\cdot(i+4)/2$. 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 смањило, 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

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

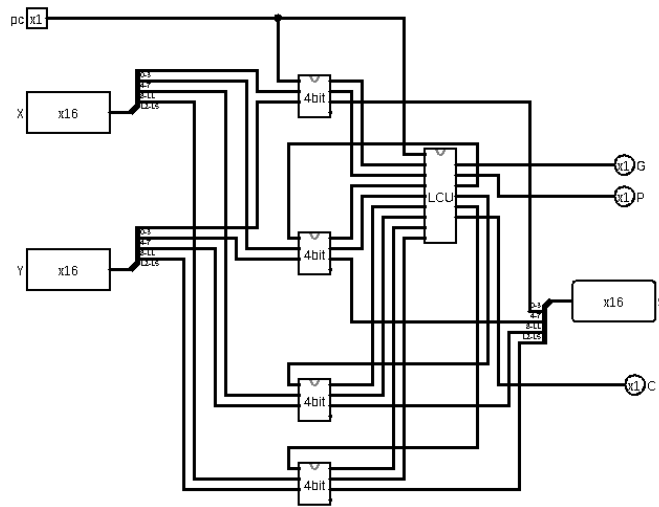
и

$$P_G = P_0P_1P_2P_3$$

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 пропација 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, $(4 \cdot (k-1) + 1 + 4k) \cdot \Delta = (8k - 2) \cdot \Delta$. 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

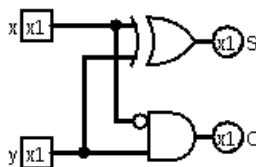
n -битни бинарни одузимач има два n -битна улаза – умањеник x и умањилац y . Вредност разлике добија се на n -битном излазу S . Поред тога, постоји још један једнобитни излаз C који, као и код сабирача, представља индикатор прекорачења. Овај излаз у ствари представља позајмицу на последњој позицији и биће једнак 1 уколико је умањеник мањи од умањеоца. 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

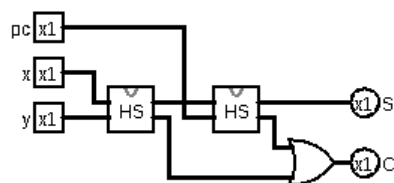
Табела 3.4: Функција полуодузимача

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



Слика 3.28: Полуодузимач

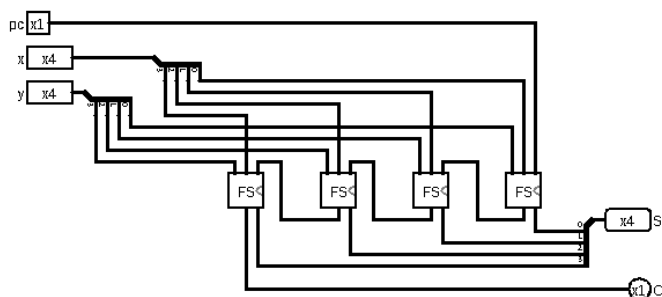
У поређењу са полусабирачем, видимо да је ово коло веома слично, осим што је улаз конјункције на који се доводи x негиран. *Потпуни одузимач* који узима у обзир претходну позајмицу се може добити од два полуодузимача на идентичан начин као у случају потпуних сабирача (слика 3.29).



Слика 3.29: Потпуни одузимач

На горњој слици, са HS означили смо полуодузимаче. Најпре од x одуземо y једним полуодузимачем, а затим од те разлике одуземо 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).



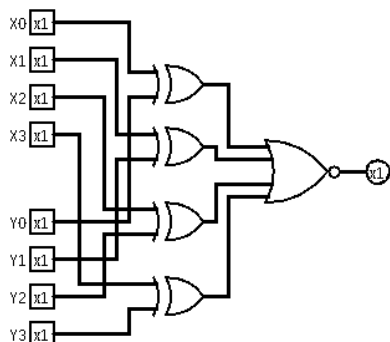
Слика 3.30: 4-битни одузимач

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

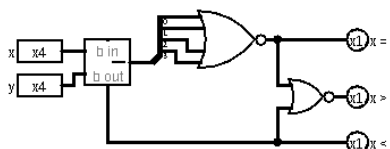
информацију да ли су та два податка једнака или не. Пример имплементације 4-битног компаратора дат је на слици 3.31.



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.

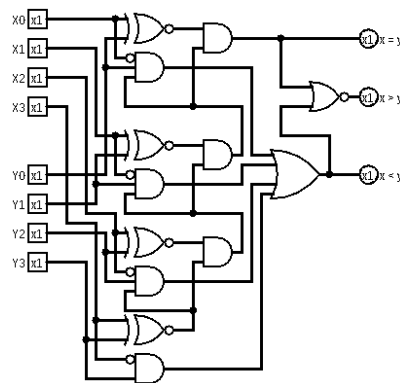


Слика 3.32: 4-битни потпуни компаратор

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

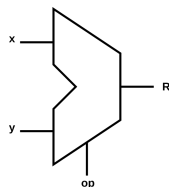
вишим позицијама. Такође, на свакој битској позицији имамо и по једну конјункцију са негираним улазом за x бит (лево на слици, у истој колони са НЕИЛИ колима), која испитује да ли је x бит једнак 0, а y бит једнак 1, под условом да је на свим вишим битским позицијама важила једнакост. Јасно је да уколико 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

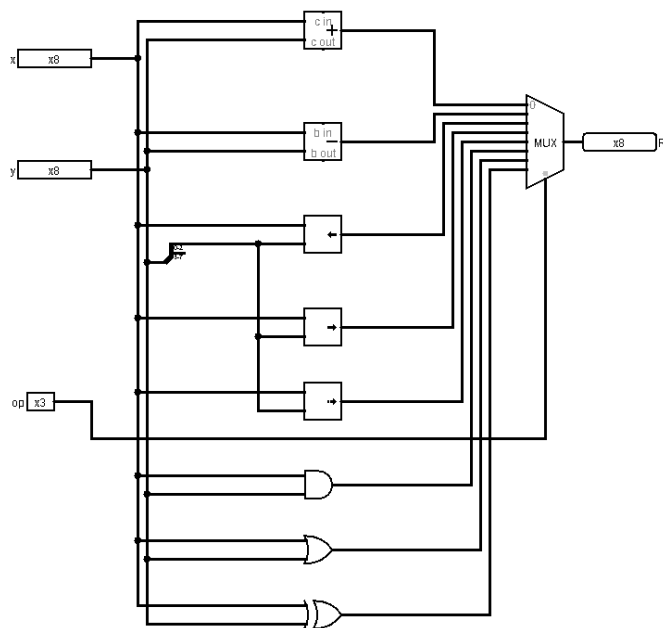
Аритметичко логичка јединица (енгл. *arithmetic logic unit (ALU)*) је један од најважнијих делова савремених рачунарских система и саставни је део сваког модерног процесора. У питању је комбинаторно коло које може да извршава различите аритметичке и логичке операције, у складу са захтевом корисника. 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 код операције коју желимо да ALU коло изврши. На излазу R се добија тражени

результат. 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, одузимаачи, помераачи, компаратори и сл.) као и једног мултиплексера који на основу задатог кода 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).

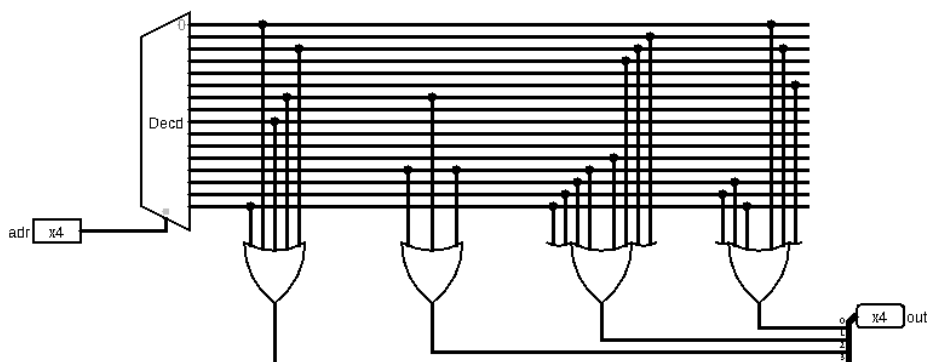
3.3 Општа комбинаторна кола

Поред раније описаних комбинаторних кола која имају специфичну намену, често постоји потреба и за општим комбинаторним 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 zgodније 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.

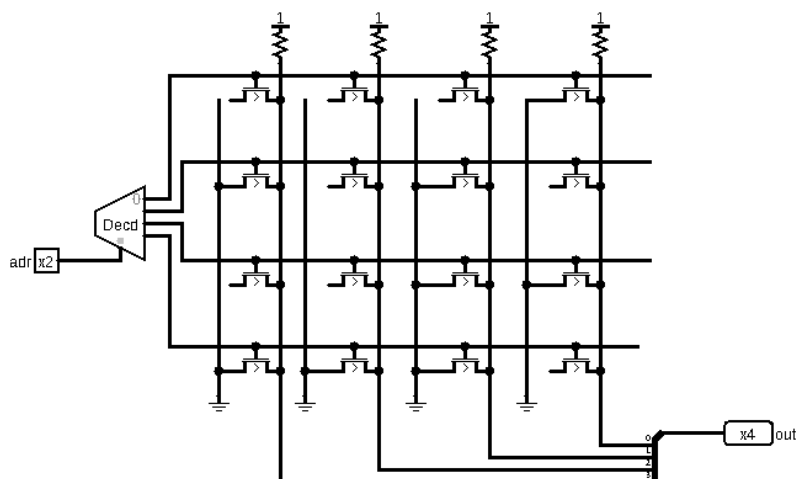


Слика 3.36: Концептуална шема ROM меморије

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 фиксираниог 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 табlici 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*).

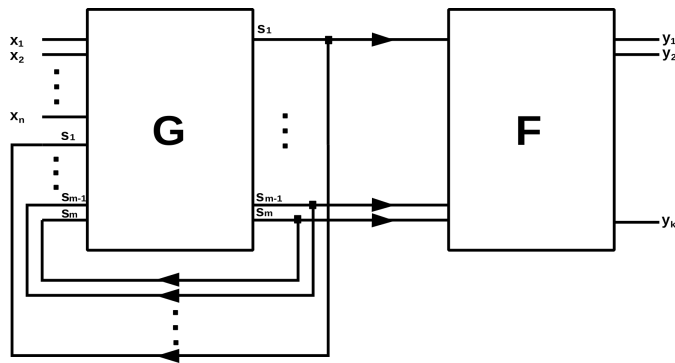
Глава 4

Секвенцијална кола

Код комбинаторних кола, вредност на излазу $Y = (y_1, \dots, y_k)$ у неком тренутку t зависи искључиво од вредности улаза $X = (x_1, \dots, x_n)$ 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 векторска¹ 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 бита $S = (s_1, s_2, \dots, s_m)$ 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: Концептуална шема секвенцијалног кола

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

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

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 стабилизovati 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. $G(s) = \bar{\bar{s}} = s$, 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. $G(s) = \bar{\bar{\bar{s}}} = \bar{s}$. 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 стабилизovati 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:

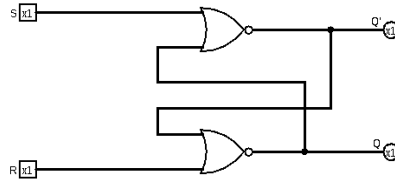
$$\mathcal{S}(G) = \{S \mid \exists X. S = G(X, S)\}$$

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 — *реза*.

4.1 Резе

Реза је меморијско коло које има могућност чувања једнобитног стања. Улази *реза* омогућавају постављање и одржавање вредности стања, док излази омогућавају читавање 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 стабилизovati 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 стабилизovati 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 $Q' = \bar{Q}$.

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 зовео постављање или сетовање бита). Улаз $(S, R) = (0, 1)$ ће довести до тога да се бит Q постави на 0 (овај поступак зовео 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.

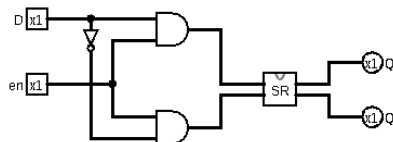
Функција преласка SR резе дата је у табели 4.2.

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 тренутно 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 асинхронно 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 *улазни руб* или *улазна ивица*, док се прелазак са 1 на 0 назива *силазни руб* или *силазна ивица* часовника. Временски период између две улазне 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 улазним (или силазним) рубовима часовника. Следеће стање се одређује на основу текућег стања, као и вредности улаза у тренутку наилаaska 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 читавају 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 наилаaska одговарајућег руба часовника. На пример, ако бисмо имали синхрону варијанту D резе (коју ћемо називати и *D флип-флор*), тада би једино било битно да у тренутку наилаaska 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 наилаaska 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 наилаaska одговарајућег руба часовника имају стабилизване вредности на улазима. Те вредности ће бити прочитане у тренутку наилаaska 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 наилаaska 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-флор* (енгл. *flip-flop*).

flip). Флип-флоп, као и реза, чува један бит податка. Међутим, у случају флип-флопа постоји и додатни *clk* улаз који синхронизује промену његовог стања. Промене стања могу се догодити само у тренутку наилазак 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*).

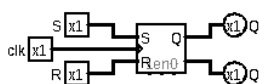
4.3.1 SR флип-флоп

SR флип-флоп има исте улазе и излазе као и SR реза, као и исту таблицу преласка³ (табела 4.4).

<i>S</i>	<i>R</i>	<i>Q</i>	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).



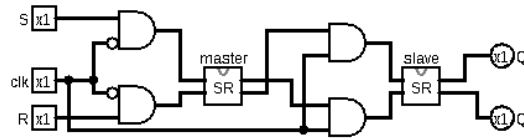
Слика 4.5: Шематска ознака SR флип-флопа

Када се на часовнику појави одговарајући руб, коло у том тренутку читава 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

²Напоменимо 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.

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

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



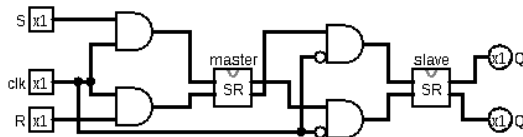
Слика 4.6: Имплементација SR флип-флопа

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 \bar{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 $(Q, \bar{Q}) = (1, 0)$ 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 контролисани 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 наилаaska 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

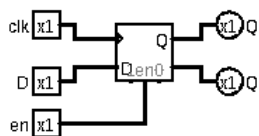
4.3.2 D флип-флоп

D флип-флоп има исте улазе и исту семантику као и D резе. 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

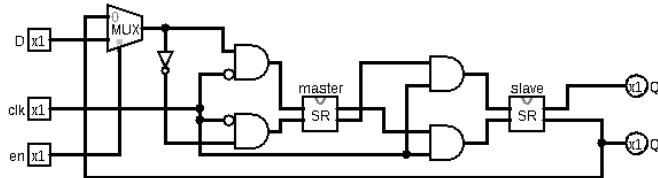
Додатни улаз clk омогућава синхронизацију помоћу часовника. Шематска ознака D флип-флопа је дата на слици 4.8.



Слика 4.8: Шематска ознака D флип-флопа

Имплементација „господар-слуга” дата је на слици 4.9. Већи део ове шеме одговара претходно приказаној имплементацији SR флип-флопа, па самим тим читав „господар-слуга” механизам функционише на потпуно исти начин. Постоје две битне разлике. Прва, као и код D резе, је у томе што су S и R улази главне резе (испред конјункција) повезани негацијом. На овај начин се обезбеђује да у сваком тренутку имамо или комбинацију $(S, R) = (1, 0)$ или комбинацију $(S, R) = (0, 1)$, што значи да ћемо на улазном рубу часовника увек запамтити једну од две вредности на основу тренутне комбинације на улазу. Друга разлика је у додатном 2-1 мултиплексеру који обезбеђује да комбинација на улазу буде права. Када је на en улазу 0, тада је по табlici 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 en 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 4.9: Имплементација D флип-флопа

4.3.3 JK флип-флоп

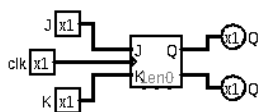
JK флип-флоп претставља другачији приступ решавању проблема непотпуне семантике SR флип-флопа. Сетимо се да је код SR флип-флопа главни проблем био то што је за две јединице на улазу понашање флип-флопа било недефинисано. Овде је идеја да покушамо да и за ову комбинацију на улазу дефинишемо неко смислено 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 новедефинисану улогу – да инвертује вредност сачувану у флип-флопу. Таблица преласка JK флип-флопа је дата у табели 4.6.

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

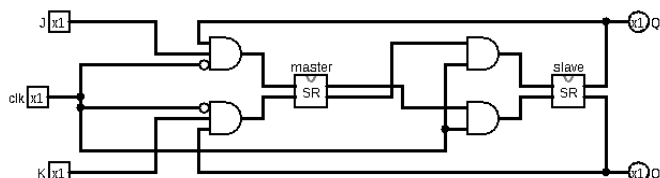
Табела 4.6: Таблица преласка JK флип-флопа

Шематска ознака JK флип-флопа дата је на слици 4.10, а „господар-слуга” имплементација JK флип-флопа који реагује на улазне руб часовника дата је на слици 4.11.

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



Слика 4.10: Шематска ознака JK флип-флопа



Слика 4.11: Имплементација JK флип-флопа

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.

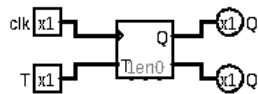
4.3.4 Т флип-флоп

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

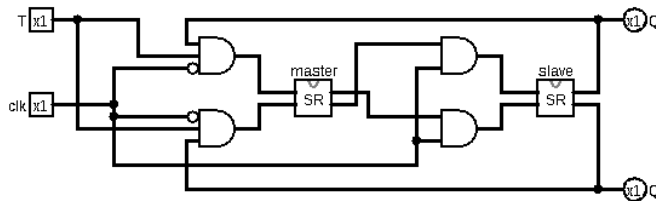
Табела 4.7: Таблица преласка Т флип-флопа

Шематска ознака Т флип-флопа дата је на слици 4.12.



Слика 4.12: Шематска ознака Т флип-флопа

Имплементација „господар-слуга” дата је на слици 4.13.



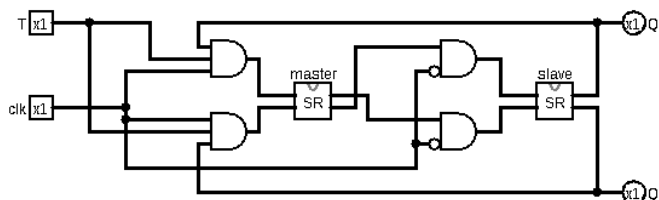
Слика 4.13: Имплементација Т флип-флопа

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”

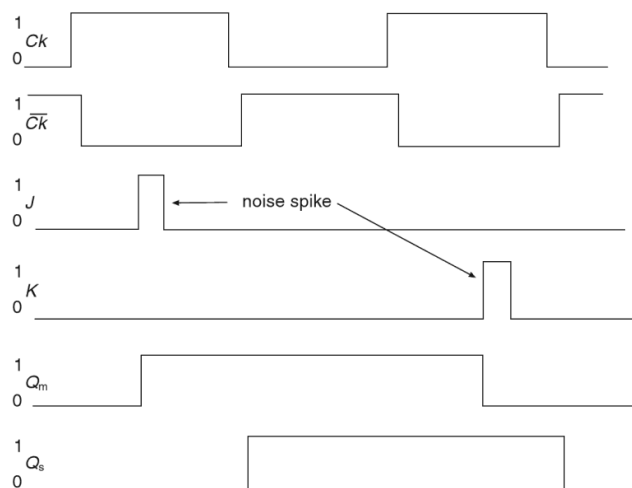
Проблем „хватања јединице” (енгл. *is catching problem*) је проблем који се у одређеним околностима може манифестовати код JK флип-флопа. Наиме, сетимо се да би синхрона кола требало да читавају своје улазе само у тренутку наилазак узлазног (или силазног) руба часовника и на основу вредности улаза у 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 4.14. Приметимо

да ова имплементација реагује на силазну ивицу часовника, али то није суштински битно.



Слика 4.14: JK флип-флоп који мења стање на силазној ивици

Посматрајмо временски дијаграм сигнала дат на слици 4.15.



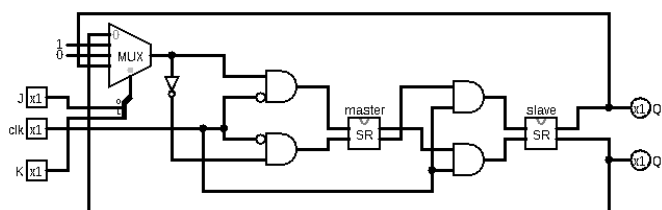
Слика 4.15: Хватање јединице приказано на временском дијаграму

Током позитивног 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 силазној 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 силазне 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 се дешава ако се на улазу K појави краткотрајна јединица током позитивног дела часовника, када је тренутно стање флип-флопа један (у десном делу дијаграма).

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 стабилизovati 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 краткотрајни импулс прође. Као и код D флип-флопа, рачунаће се само оно што се на улазима затекло у тренутку наиласка одговарајућег руба часовника. На сличан начин се може решити проблем хватања јединице и код SR и код T флип-флопа.

4.3.6 Време поставке и време задржавања

Код комбинаторних кола, најзначајнији параметар је било кашњење, тј. време пропације сигнала од улаза до излаза. Ову карактеристику имају и секвенцијална кола. Конкретно, код синхроних секвенцијалних кола, *време пропације* 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 наилаaska 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 наилаaska 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.

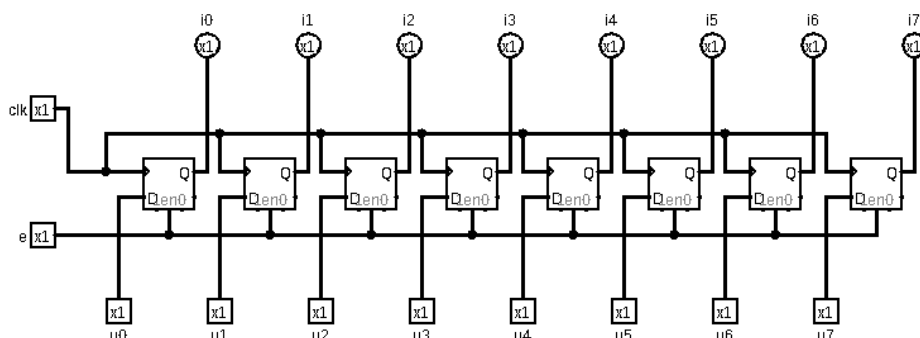
Сетимо се да смо раније рекли да би синхрона секвенцијална кола требало да узимају у обзир вредности улаза само у тачно одређеном тренутку, тј. у тренутку наилаaska руба часовника t_0 . У пракси видимо да ова кола никада нису идеална, тако да уместо једног тренутка t_0 постоји изванредан интервал $(t_0 - t_s, t_0 + t_h)$ 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 4.17.

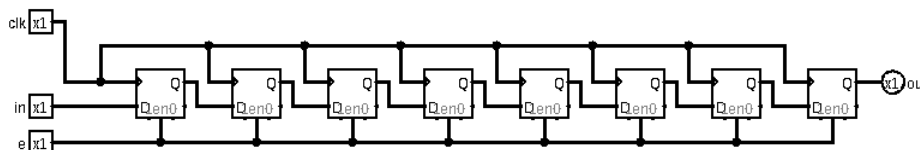
Приметимо да су сви D флип-флопови повезани на исте clk и e сигнале. Такође, сви флип-флопови су конструисани тако да реагују на исти руб часовника. Када је e сигнал нула, регистар чува вредност коју је имао и у претходном циклусу, док се за $e = 1$ у регистру памте вредности које се у тренутку одговарајућег руба часовника нађу на улазима $u_0 - u_7$. Читање вредности сачуване у регистру се у сваком тренутку може обавити читавањем вредности на излазима $I_0 - I_7$.

Поред читања и уписа, регистри могу подржавати и друге операције.



Слика 4.17: 8-битни регистар

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 4.18 дат је једноставан померачки регистар.

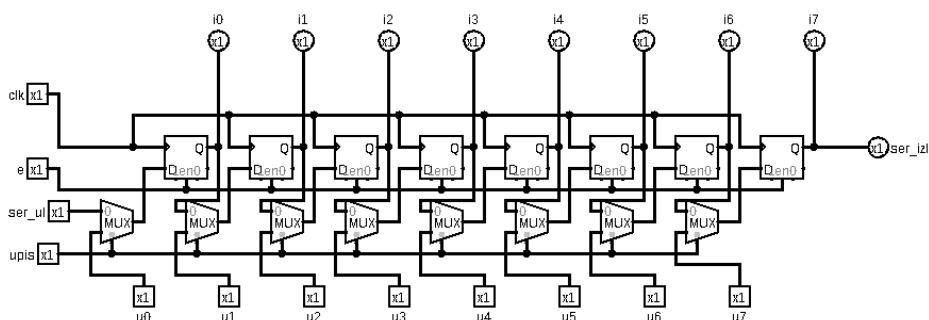


Слика 4.18: Померачки регистар

Код овог једноставног померачког регистра излаз сваког флип-флопа се повезује на улаз следећег. На овај начин се обезбеђује да се у сваком 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 највећи 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 4.19.

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



Слика 4.19: Померачки регистар са паралелним улазом и излазом

упишу одједном, а да се затим та вредност помера у регистру. Типична примена оваквих регистара је у имплементацији алгоритама множења (попут Бутовог алгорита). 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 податка се преносе одједном, путем одговарајућег броја паралелних линија. Уколико је потребно да се податак који је до неке тачке пренешен 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

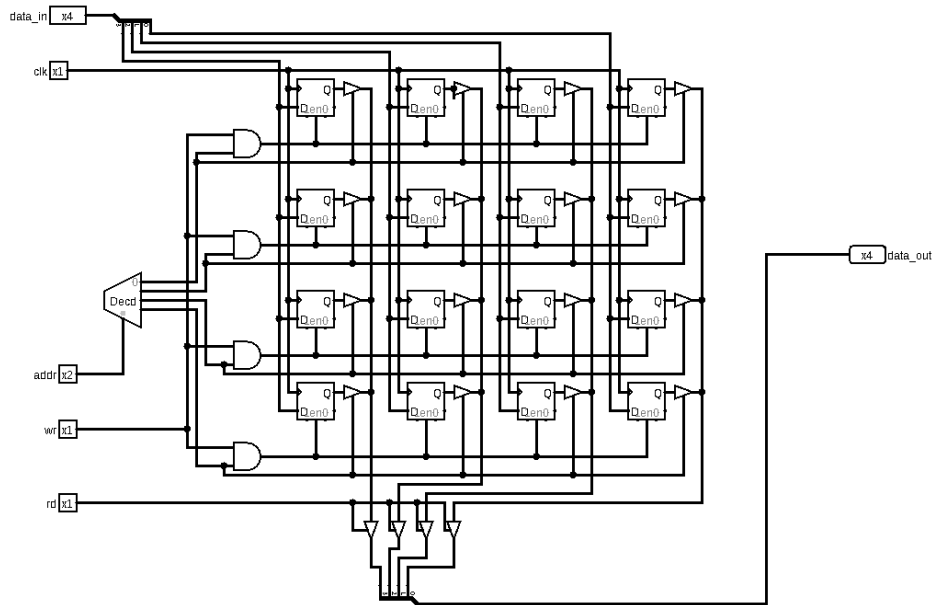
Меморија представља низ *меморијских локација*, при чему свака меморијска локација може да чува низ битова фиксне дужине. Уколико имамо m меморијских локација од којих се свака састоји од n битова, тада кажемо да је меморија димензије $m \times n$. Свака од m меморијских локација има своју *адресу*. 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.

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. Такође, многи механизми за

побољшање ефикасности приступа раде искључиво у синхронном режиму (попут испреплетених меморија). Због тога се данас чешће користе синхроне меморије.

4.5.1 Синхроне меморије

Пример једноставне синхроне меморије 4×4 дат је на слици 4.20.



Слика 4.20: 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 прочитамо. Постоје и два контролна једнобитна улаза: улаз *rd* активира излаз за податке и омогућава читавање 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 улазном рубу часовника.

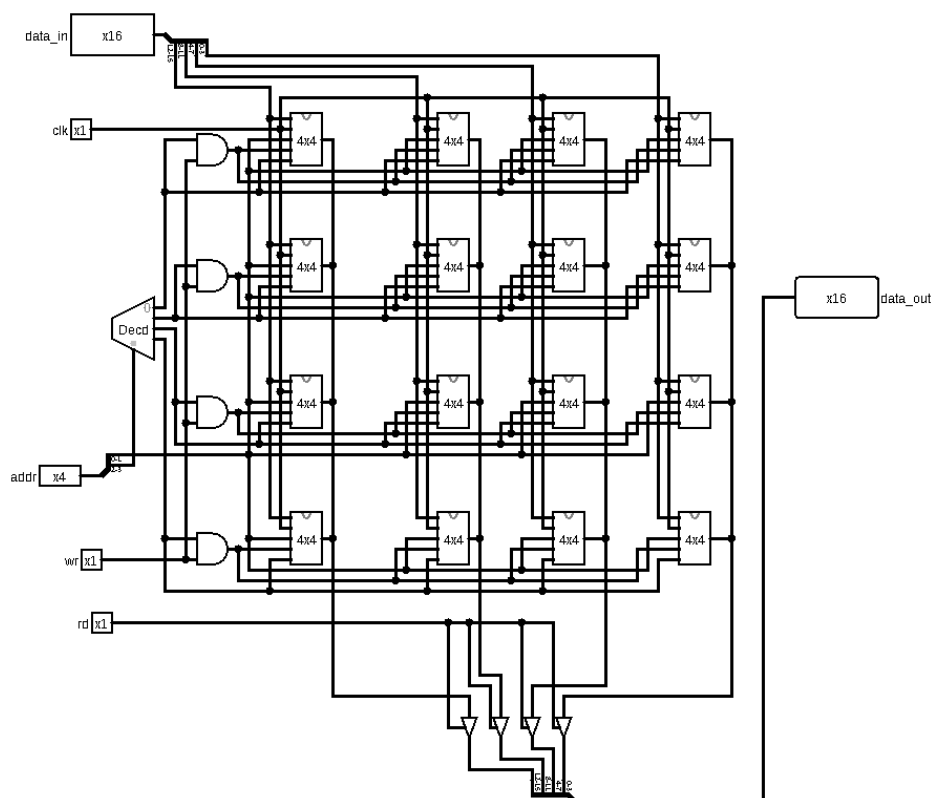
Меморија је структурно организована као матрица $m \times n$, где свака врста представља једну меморијску локацију. Адреса се декодира уз помоћ декодера, при чему свака излазна линија декодера активира одговарајућу врсту матрице. У случају операције уписа, излазна линија декодера се користи да активира 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 $m = 2^k$ регистра 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. Отуда су веће меморије 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 4.21.



Слика 4.21: Сихрна меморија 16×16 реализована помоћу меморија 4×4

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 као 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 селекују 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.

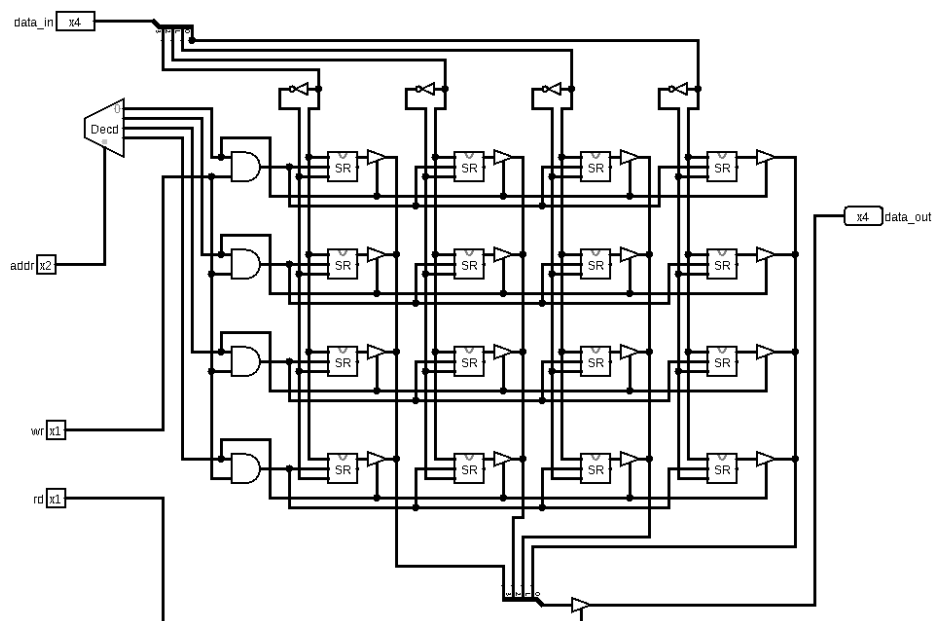
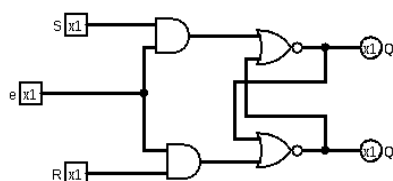
Меморија се може слагати на више начина. На пример, меморија 16×16 се може добити и тако што се меморије 8×8 слажу у матрицу 2×2 , али и тако што се меморије 4×1 слажу у матрицу 4×16 . У пракси се веома често користе меморијски блокови димензије $m \times 1$, тј. меморијске локације у њима су једнобитне. Овакви блокови се могу једноставно слагати тако да се добију меморијске локације произвољне дужине.

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 4.22. На овој слици, свака од SR реза има додатни e улаз, тј. изгледа као D реза из које је избачено NE коло (слика 4.23).

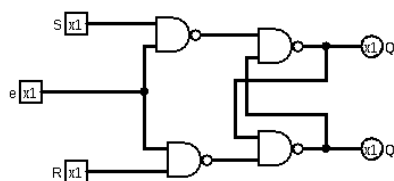
Уместо да свака реза има своје NE коло, имамо по једно NE коло у свакој колони. Вредност одговарајућег бита са улаза се једном негира, а онда се та вредност и њена негација спроводе респективно на S и R улазе SR резе

Слика 4.22: Асинхрона меморија 4×4 Слика 4.23: SR реза са додатним *enable* улазом

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 4.24. Читаоцу остављамо да провери да су ове две имплементације међусобно еквивалентне. Ова алтернативна реализација захтева 16 транзистора, јер се састоји из 4 NAND кола.

Још ефикаснија реализација реза у асинхроној меморији приказана је на слици 4.25. На овој слици приказан је део матрице меморије који садржи 4 резе, како бисмо имали осећај на који се начин овакве резе могу слагати тако да чине меморију. Свака врста представља једну меморијску локацију и активира се одговарајућом *линијом речи* (енгл. *word line*) која



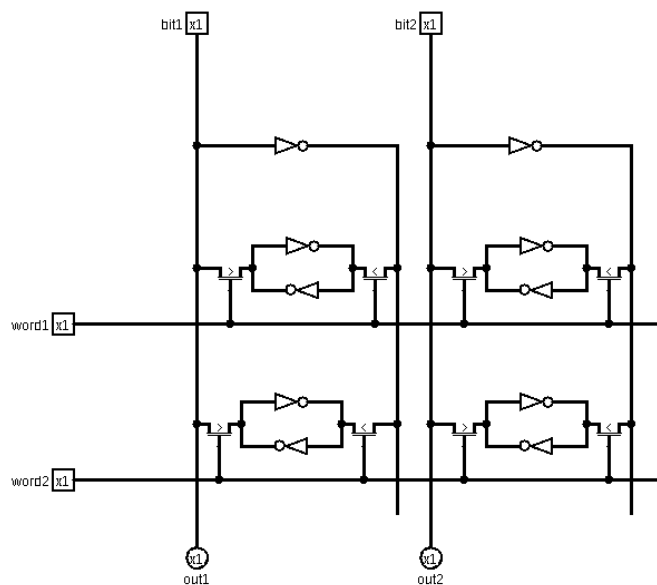
Слика 4.24: SR реза са додатним *enable* улазом реализована помоћу НИ кола

представља одговарајућу излазну линију меморијског декодера адресе. Са друге стране, свака колона се повезује на једну *линију бита* (енгл. *bit line*) којом се одговарајући бит са улаза доводи до меморијских ћелија. Истовремено, линије битова се користе и за повезивање на излаз. 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 дође негације (тј. вредност на „левој страни” резе) се сматра вредношћу која се чува у рези. Када желимо да читамо вредност, тада декодер активира одговарајућу линију речи чиме се отварају одговарајући пропусни транзистори и вредности које се чувају у ћелијама те врсте се пропуштају на линије битова. Када желимо да извршимо упис, тада се вредности које хоћемо да упишемо доведу на линије битова, а онда се активира одговарајућа линија речи. Притом, пропусни транзистори се праве тако да буду снажнији од транзистора који се налазе у НЕ колима. На тај начин, вредности које они пропусте са битских линија надјачаће вредности које се тренутно чувају у резама и натераће их да промене своја стања која ће након искључивања линије речи остати сачувана у резама.

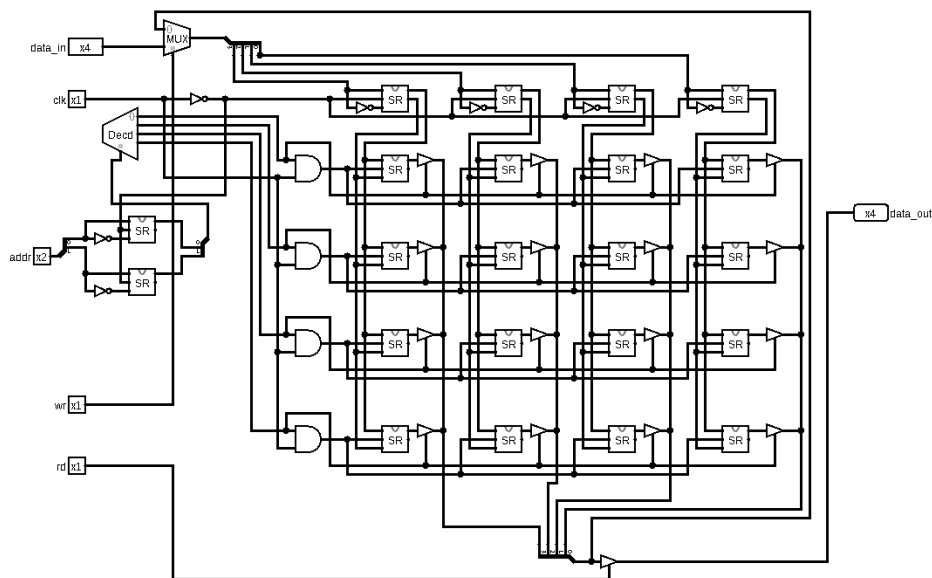
При овој имплементацији за сваки бит потребно је свега 6 транзистора, што је много ефикасније од раније приказаних реализација. Додатно, ова имплементација је веома брза, тј. има веома мало кашњење на нивоу меморијских ћелија (не рачунамо овде кашњење декодера које зависи од величине меморије). Колико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

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 4.26.



Слика 4.25: Фрагмент ефикасне имплементације асинхроне меморије



Слика 4.26: Оптимизована синхрона меморија

На слици 4.26 дата је меморија 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 мастер резе. Притом, то се дешава само ако је *wr* сигнал укључен. У супротном, мултиплексер у горњем левом углу ће на мастер резе проследити тренутну вредност изабране меморијске локације која ће на тај начин бити поново уписана, тј. нећемо имати промену вредности локације (слично као што смо имали код имплементације *D* флип-флопа). 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 наилаaska 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.

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 , 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, адресних реза и једног мултиплексера. У сложенијим меморијама (нарочито динамичким меморијама, о којима ће касније бити речи) ова синхронизациона логика може бити знатно сложенија. Међутим, принцип је увек исти – најпре направимо матрицу асинхроне меморије која се састоји из асинхроних меморијских ћелија, а затим је синхронизујемо додавањем синхронизационе логике.

4.5.5 О произвољном приступу

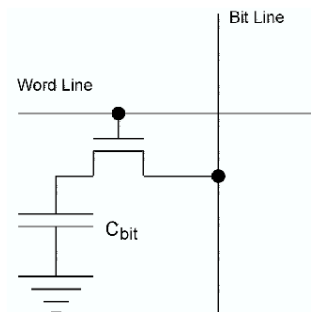
Меморије које смо приказали у претходним одељцима имају својство да се свакој меморијској локацији може приступити у приближно једнаком времену. Под *временом приступа* (енгл. *access time*) подразумевамо време од тренутка када се изда захтев за читање или упис појединачне локације до тренутка када се 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)). 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 састоји се од само једног транзистора и једног кондензатора (слика 4.27).



Слика 4.27: 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 испразњен, тада је потенцијал његове горње електроде низак, па је сачувана вредност 0. Транзистор контролише везу кондензатора са спољним светом. Када желимо да упишемо вредност у ћелију, на битску линију доводимо одговарајућу вредност и активирамо линију речи. Тиме се отвара транзистор, а кондензатор се пуни ако је линија бита на високом потенцијалу (вредност 1), а празни ако је линија бита на ниском потенцијалу (вредност 0). Приликом читања, вредност битске линије се најпре наелектрише на неки међупотенцијал (нпр. око $2.5V$), а онда се активира линија речи. Ако је кондензатор напуњен (вредност 1), тада ће се наелектрисање из њега пренети на битску линију чиме ће њен потенцијал постати за нијансу већи (нпр. око $2.6V$), док ће се кондензатор испразнити. Посебно електронско коло за појачавање ће регистровати ту малу промену потенцијала битске линије и појачаће је до „пуне” логичке јединице (тј. до $5V$), па ћемо моћи да је прочитамо на излазу. Такође, када се то деси, кондензатор ће се преко отвореног транзистора поново напунити, тако да ће и даље чувати вредност 1. Слично, ако је кондензатор био празан (тј. ћелија је чувала вредност 0), тада ће се део наелектрисања са битске линије пренети преко отвореног транзистора у кондензатор, па ће потенцијал битске линије постати за нијансу мањи (нпр. око $2.4V$). Појачавачко коло ће регистровати ову малу

промену и појачаће је, тј. потенцијал битске линије ће после извесног времена постати $0V$, па ће вредност 0 бити прочитана на излазу. Такође, низак потенцијал на битској линији ће поново испразнити кондензатор, чиме ће се у њега поново уписати вредност 0.

Оно што примећујемо је да се приликом сваког читавања наелектрисање у кондензатору промени (испразни или напуни), да би се након појачавања вредности битске линије поново вратио на старо стање. Ова појава се зове *деструктивно читање*. Другим речима, при сваком читању, уписане вредности се најпре униште, па се затим поново упишу. Цео овај поступак (који се обично назива *отварање врсте*, јер се истовремено одвија над свим ћелијама у истој врсти) захева пуњење и пражњење кондензатора и веома је спор. Отуда је време приступа ћелијама динамичке меморије знатно веће него код статичких меморија. Додатно, кондензатори се 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. Ово захтева веома сложену логику за освежавање и додатно успорава рад целе меморије.

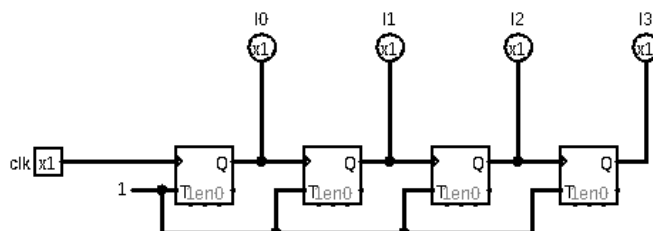
Описани дизајн динамичких меморија је у основи асинхрон. Додавањем синхронизационе логике добијају се *синхроне динамичке меморије са произвољним приступом* (енгл. *synchronous dynamic random access memory* (SDRAM)). Синхронизациона логика је овде знатно сложенија у односу на раније приказану синхронизациону логику код статичких меморија, због тога што је рад са динамичким меморијама сложенији: уместо да имамо само операције читања и писања, код динамичких меморија имамо операције отварања и затварања врсте, операције освежавања, и тд. Додатно, синхронизациона логика омогућава имплементацију сложенијих техника приступа (попут технике испреплетених меморија) које значајно смањују време приступа приликом приступа суседним меморијским локацијама.

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).

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).



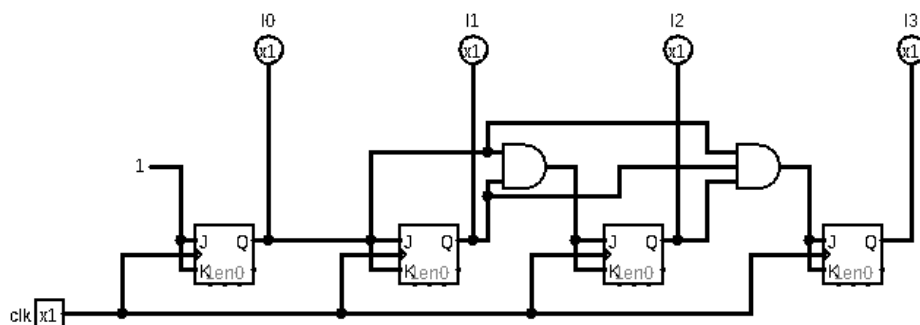
Слика 4.28: Асинхрони бројач

Пример имплементације бројача дат је на слици 4.28. 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 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 $000 \rightarrow 001 \rightarrow 010 \rightarrow 011 \rightarrow 100 \rightarrow 101 \rightarrow 110 \rightarrow 111 \rightarrow 000$. 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 $111 \rightarrow 000$ 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 стабилизovati 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 синхронног бројача дат је на слици 4.29.



Слика 4.29: Синхрони бројач

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 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 $00 \rightarrow 01$, $01 \rightarrow 10$, $10 \rightarrow 11$, $11 \rightarrow 00$, 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. $011 \rightarrow 100$ и $111 \rightarrow 000$, 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 k -тог 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.

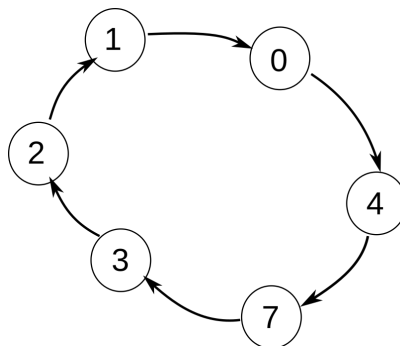
Сви флип-флопови који треба да промене своје стање у неком циклусу мењају стање у истом тренутку, тј. у тренутку узлазног руба часовника. Кашњење бројача сада зависи само од кашњења конјункција које рачунају вредности на J и K улазима, као и од кашњења самих флип-флопова, при чему нема акумулације кашњења. Зато су овакви бројачи по правилу

⁴у 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.

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 $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 0$, он броји у редоследу $0 \rightarrow 4 \rightarrow 7 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0$. 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 4.30.



Слика 4.30: 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 доведемо 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 табели 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: $0 \rightarrow 4 \rightarrow 7 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0$, 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 $000 \rightarrow 100 \rightarrow 111 \rightarrow 011 \rightarrow 010 \rightarrow 001 \rightarrow 000$. 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 табели ексцитације за JK флип-фlop, одређујемо на које вредности је потребно поставити J и K улазе

одговарајућег флип-флопа, да би се такав прелаз десио.

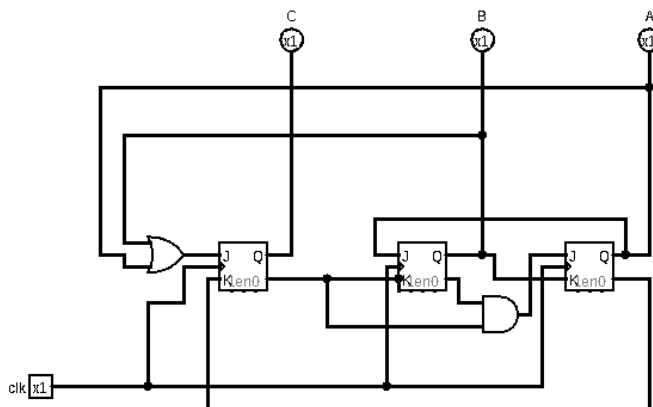
A	B	C	A^{sled}	B^{sled}	C^{sled}	J_A	K_A	J_B	K_B	J_C	K_C
0	0	0	1	0	0	1	—	0	—	0	—
0	0	1	0	0	0	0	—	0	—	—	1
0	1	0	0	0	1	0	—	—	1	1	—
0	1	1	0	1	0	0	—	—	0	—	1
1	0	0	1	1	1	—	0	1	—	1	—
1	0	1	—	—	—	—	—	—	—	—	—
1	1	0	—	—	—	—	—	—	—	—	—
1	1	1	0	1	1	—	1	—	0	—	0

Табела 4.9: Таблица ексцитације бројача са слике 4.30

Приметимо да бројач не мора пролазити кроз сва могућа тробитна стања. На пример, наш бројач не пролази кроз стања 5 и 6. У табlici 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 (нпр. помоћу Карноових мапа, што остављамо читаоцу за вежбу):

$$\begin{aligned} J_A &= \overline{B} \overline{C} & J_B &= A & J_C &= B + A \\ K_A &= B & K_B &= \overline{C} & K_C &= \overline{A} \end{aligned}$$



Слика 4.31: Имплементација бројача са слике 4.30

Имплементација бројача дата је на слици 4.31. 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 ште 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 НЕ 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.

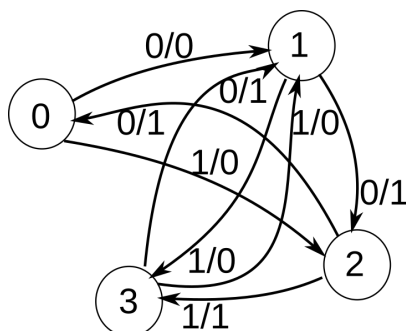
Q	X	Q^{sled}	Y
0	0	1	0
0	1	2	0
1	0	2	1
1	1	3	0
2	0	0	1
2	1	3	1
3	0	1	1
3	1	1	0

Табела 4.10: Пример таблице преласка аутомата

Овај аутомат можемо представити и графом, као на слици 4.32. 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

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

прелазу. На пример, када је аутомат у стању 0, а на улазу је вредност 1, тада аутомат на следећем улазном рубу часовника прелази у стање 2, а на излазу се добија вредност 0.



Слика 4.32: 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

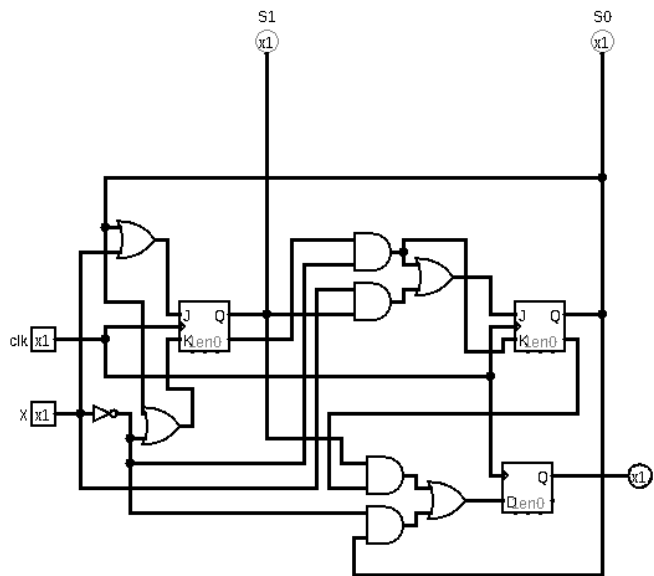
Дизајн коначних аутомата се обавља на врло сличан начин као код дизајна бројача са произвољним редоследом стања. На основу таблице преласка генеришемо таблицу ексцитације (табела 4.11).

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 ексцитације аутомата из табеле 4.10

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 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 ексцитације 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:

$$\begin{aligned} J_1 &= S_0 + X & K_1 &= S_0 + \bar{X} \\ J_0 &= \bar{S}_1 \bar{X} + S_1 X & K_0 &= \bar{S}_1 \bar{X} \\ Y &= S_1 \bar{S}_0 + S_0 \bar{X} \end{aligned}$$



Слика 4.33: Имплементација аутомата из табеле 4.10

Имплементација аутомата дата је на слици 4.33. Трансдуктор 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 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 улазном 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 стања⁶ називају се

⁶Ово није у контрадикцији са претходно изнетом чињеницом да је излаз функција од

аутомати Муровог 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 синхронизације промена на излазу може понашање аутомата учинити мање предвидивим. 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.

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.

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 извориште 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.

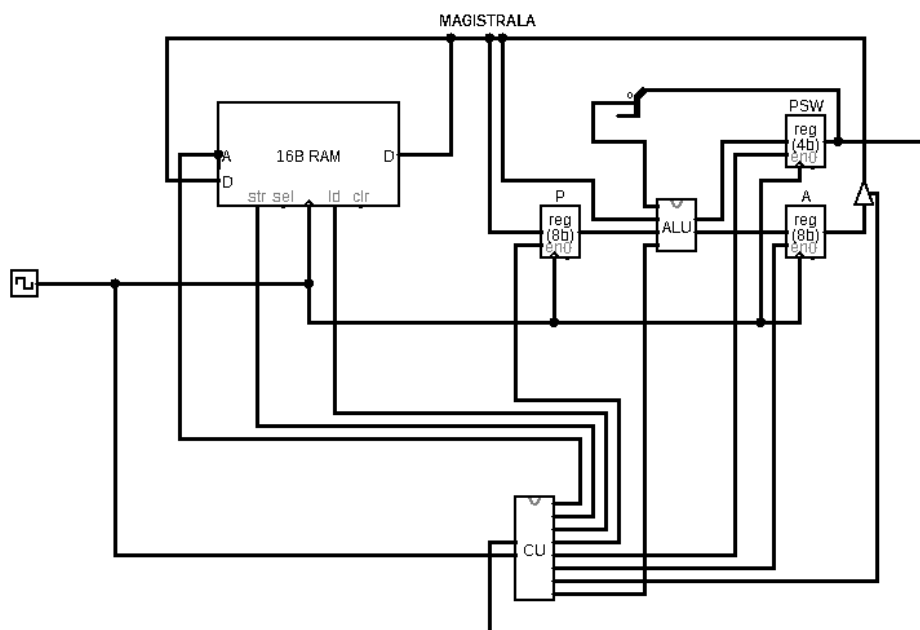
Јединица рачунара која је одговорна за управљање током програма назива се *контролна јединица*. У својој основној варијанти, контролна јединица представља коначни аутомат чије стање означава корак програма који се тренутно извршава. На улаз контролне јединице се доводе информације које утичу на ток програма (попут статуса претходно извршене операције), док се на излазу налазе контролни сигнали који управљају током података у том кораку (тј. одређују извориште, 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 меморијских локација – регистра, које ћемо означавати са R_0, R_1, \dots, 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 наилазак улазног руба часовника (тј. улаз str одговара улазу wr који смо имали раније).

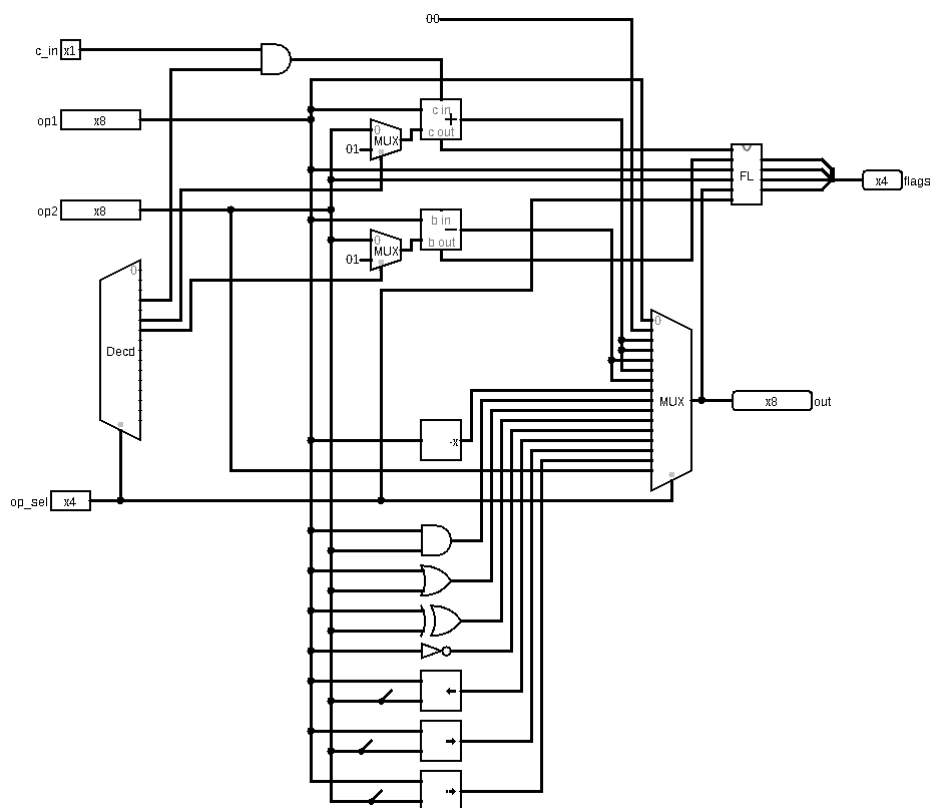
ALU јединица. Аритметичко логичка јединица (ALU) је осмобитна и подржава 16 различитих операција (отуда је улаз за избор операције четворобитни). Подржане операције дате су у табели 5.1.

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]	$out = x \ll y$
1101	[shr]	$out = x \gg y$
1110	[sar]	$out = x \ggg y$
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 $x \gg y$ означава логичко, а $x \ggg y$ 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 ако 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)

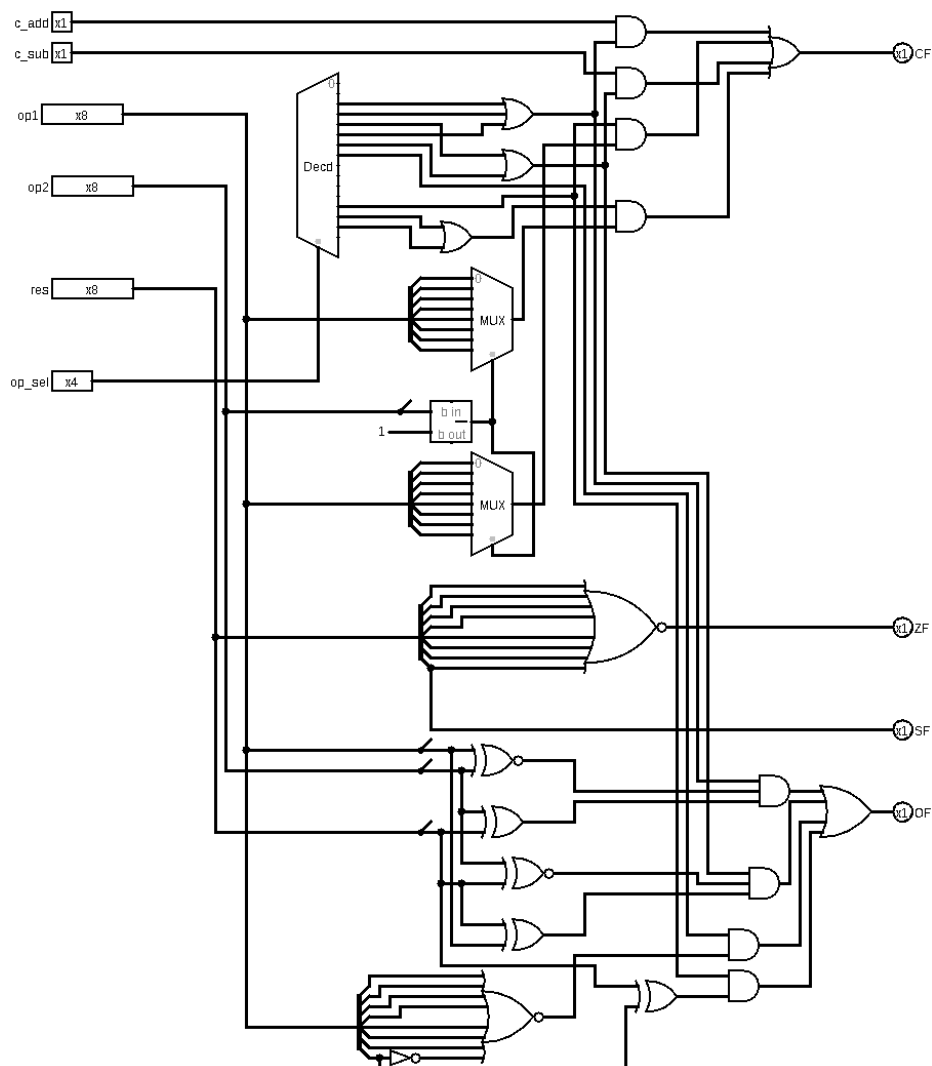
Иначе, вредност овог бита је 0.

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 улазном рубу часовника. Особитни 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).

Особитни 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

¹Напоменимо 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 пребацити у одредиште.



Слика 5.3: Имплементација израчунавања флегова

Четворобитни регистар *PSW* (енгл. *process 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 селекује 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 селекује 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 постаче *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 формиран 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:

- $R_i \rightarrow P$: трансфер вредности из меморијског регистра R_i у регистар P
- $R_i \text{ op } P \rightarrow A, PSW$: израчунавање операције $R_i \text{ op } P$ у ALU јединици и смештање резултата и флегова у регистре A и PSW , респективно
- $A \rightarrow R_i$: трансфер вредности из регистра A у меморијски регистар R_i
- $A \rightarrow P$: трансфер вредности из регистра A у регистар P
- $A \text{ op } P \rightarrow A, PSW$: израчунавање операције $A \text{ op } P$ 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 магистралама 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

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 користимо следећу нотацију:

текуће_стање) услов ? операција (*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.

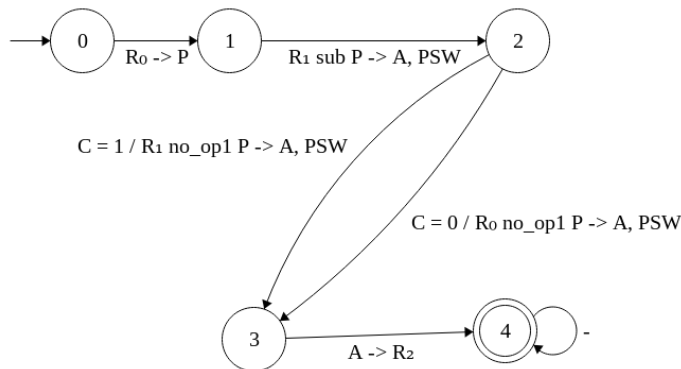
Као илустрацију, посматрајмо једноставан програм који израчунава минимум два неозначена броја који се налазе у регистрима R_0 и R_1 , а резултат се смешта у регистар R_2 . На језику високог нивоа, овај програм би се могао описати на следећи начин:

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


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

- 0) $R_0 \rightarrow P$ (1)
- 1) $R_1 [sub] P \rightarrow A, PSW$ (2)
- 2) $C = 1 ? R_1 [no_op1] P \rightarrow A, PSW$ (3)
- 2) $C = 0 ? R_0 [no_op1] P \rightarrow A, PSW$ (3)
- 3) $A \rightarrow R_2$ (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 флага врши пребацивање одговарајућег меморијског регистра у A (користећи $[no_op1]$ операцију). Уколико је $C = 1$, то значи да је умањеник R_1 мањи од умањеоца 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 екситације аутомата контролне јединице (табела 5.3). С обзиром да имамо пет стања, за чување стања довољна су три ЖК 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 $R_0 \rightarrow P$ 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 табели 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:

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

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)
{
  if(R0 > R1)
    R0 = 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	-	-	-	-	-	-

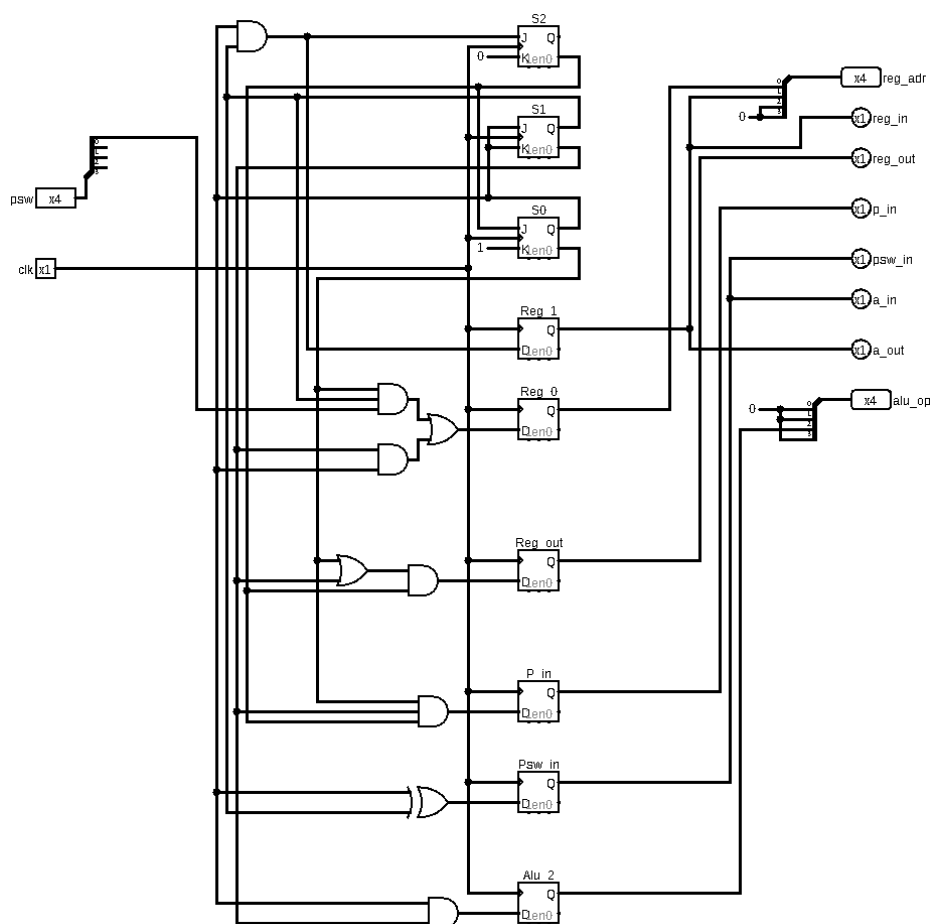
S_2	S_1	S_0	C	p_{in}	psw_{in}	a_{in}	a_{out}	$alu[3]$	$alu[2]$	$alu[1]$	$alu[0]$
0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0	0	0	0
0	0	1	0	0	1	1	0	0	1	0	0
0	0	1	1	0	1	1	0	0	1	0	0
0	1	0	0	0	1	1	0	0	0	0	0
0	1	0	1	0	1	1	0	0	0	0	0
0	1	1	0	0	0	0	1	0	0	0	0
0	1	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0
1	0	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 екситације за аутомат контролне јединице за рачунање минимума два броја (табела је, због прегледности, подељена на три дела)

```

else
  R1 = R1 - R0;
}
// Rezultat ostaje u R0

```



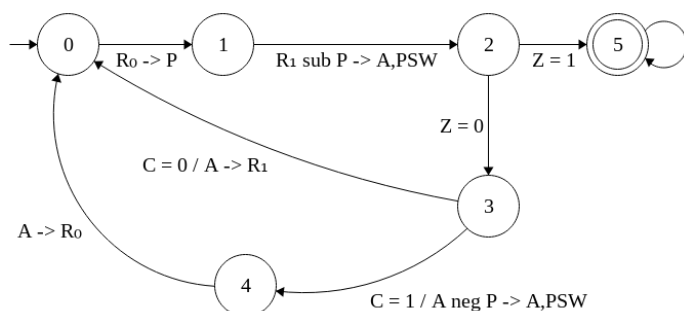
Слика 5.5: Имплементација контролне јединице за пример рачунања минимума два броја

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) $R_0 \rightarrow P$ (1)
- 1) $R_1 [sub] P \rightarrow A, PSW$ (2)
- 2) $Z = 1 ?$ – (5)
- 2) $Z = 0 ?$ – (3)

- 3) $C = 0 ? A \rightarrow R_1$ (0)
- 3) $C = 1 ? A [neg] P \rightarrow A, PSW$ (4)
- 4) $A \rightarrow R_0$ (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 ЖК 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 дуго 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

²Приметимо да у случају софтверске интерпретације нпр. *Python* програма, ми заправо имамо интерпретацију на два нивоа. Наиме, сам *Python* интерпретатор је један

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 PC³. 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 РС 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.

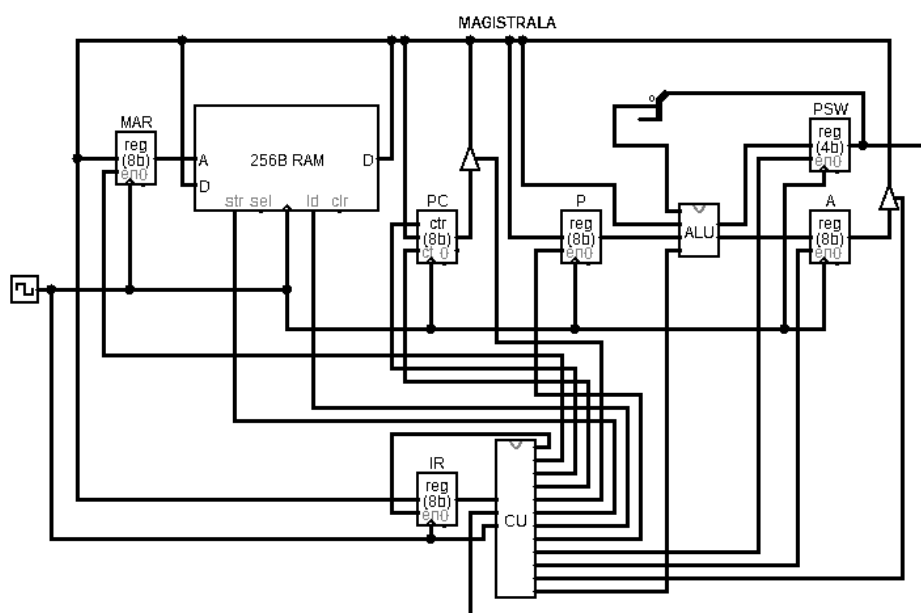
Да би инструкција на коју указује програмски бројач била извршена, потребно је проследити је контролној јединици која ће на основу садржаја инструкције одлучити о даљим корацима. Дакле, рад коначног аутомата контролне јединице више не зависи само од стања флегова, већ и од тога која се инструкција тренутно 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 запами, како би у даљим корацима могла да је користи. У ту сврху постоји још један специјални 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.

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.

³У 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.

⁴Екстремни пример тога је архитектура ARM, код које готово све инструкције могу бити извршене условно.



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, множењик, дељеник) имплицитно увек налазити у А регистру. Резултат сабирања ће се поново смештати у регистар А. Унарне аритметичке операције (попут промене знака или битовске негације) немају операнд, већ се операција извршава над регистром А, и резултат се опет уписује у регистар А. 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 А, или податак у који се уписује вредност регистра А. Инструкције трансфера дате су у табели 5.5. Инструкција LOAD ажурира флегове на основу вредности која је уписана у регистар А (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 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

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

Табела 5.4: Скуп аритметичко логичких инструкција рачунара са слике 5.7

Инструкција	Операциони код	Операција
LOAD const	00000000	$A = \text{const}$
LOAD addr	00000001	$A = \text{MEM}[\text{addr}]$
LOAD [addr]	00000010	$A = \text{MEM}[\text{MEM}[\text{addr}]]$
STORE addr	00010001	$\text{MEM}[\text{addr}] = A$
STORE [addr]	00010010	$\text{MEM}[\text{MEM}[\text{addr}]] = A$

Табела 5.5: Инструкције трансфера рачунара са слике 5.7

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.

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

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	-	$\rightarrow halt_state$

Табела 5.6: Инструкције контроле тока рачунара са слике 5.7

наставку.

Дохватање инструкције. Прва фаза у извршењу инструкције је увек *дохватање инструкције* (енгл. *fetch*). Циљ дохватања је да се операциони код 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 читава операциони код текуће инструкције и пребацује магистралом у IR

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

- 0) $PC \rightarrow MAR, pc_inc(1)$

- 1) $MEM \rightarrow IR$ (2)

Притом, ознака $MEM \rightarrow IR$ 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 операнда 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 читан 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) $PC \rightarrow MAR, pc_inc$ (5)

Приметимо да се након читавања другог бајта инструкције регистар РС 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 операнда, 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) $PC \rightarrow MAR, pc_inc$ (3)
- 3) $MEM \rightarrow MAR$ (5)

Као и малопре, најпре пребацујемо вредност РС регистра у MAR, како бисмо прочитали 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 РС 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 фиксированој 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) $PC \rightarrow MAR, pc_inc$ (3)
- 3) $MEM \rightarrow MAR$ (4)
- 4) $MEM \rightarrow MAR$ (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 прочитати 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 код, 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):

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

- 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) $A \rightarrow MEM$ (0)

- ако је у питању аритметичко-логичка инструкција која нема операнд (унарна операција), тада се она може извршити на следећи начин:

– 5) $A \text{ op } P \rightarrow A, PSW$ (0)

где је *op* одговарајућа унарна операција ALU јединице. Дакле, операција се изводи над вредношћу регистра А и резултат се поново смешта у регистар А, уз ажурирање флегова.

- у случају аритметичко-логичке инструкције која има операнд (бинарна операција), можемо извршити следеће кораке:

– 5) $MEM \rightarrow P$ (6)

– 6) $A \text{ op } P \rightarrow A, PSW$ (0)

где је *op* код одговарајуће бинарне операције ALU јединице. Дакле, најпре је потребно меморијски операнд (који је увек други операнд операције) пребацити у Р регистар, а затим извршити операцију користећи вредност регистра А као први операнд операције. Резултат се поново смешта у А, 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 РС):

– 5) – (0)

У супротном, ако услов за скок јесте испуњен, примењујемо следећи корак:

– 5) $MEM \rightarrow PC$ (0)

Овим кораком операнд инструкције (који представља адресу на коју треба скочити) копирамо у РС 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, извршаваће настави од неке произвољне адресе коју уписујемо у програмски бројач.

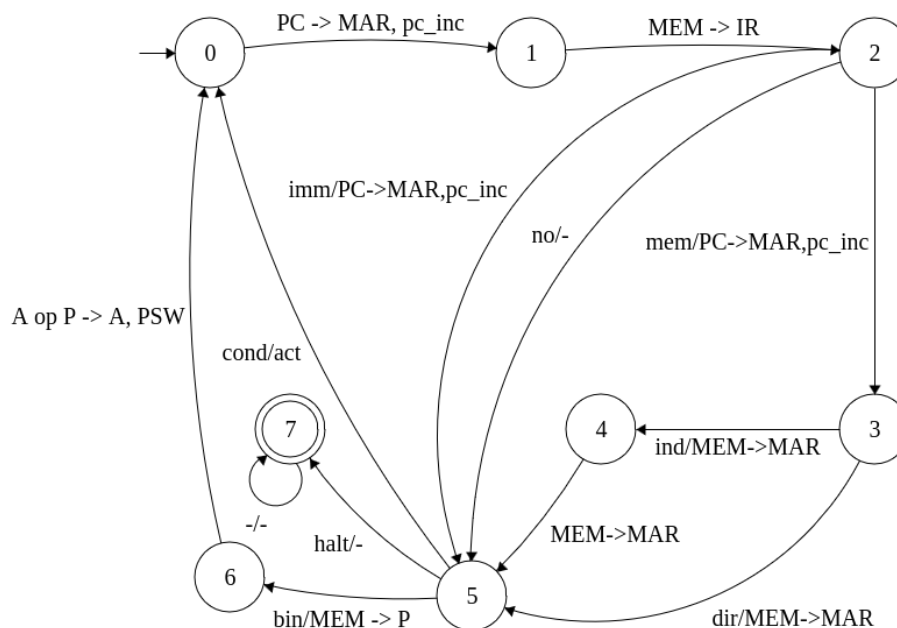
5.2.2 Аутомат контролне јединице

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 регистра. Фаза дохватања обухвата стања 0 и 1, а одговарајући преласци се обављају безусловно. У стању 2 започиње декодирање инструкције и простира се до стања 5. У случају да инструкција нема операнда (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 ЖК 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 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



Слика 5.8: Аутомат контролне јединице рачунара са слике 5.7

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 операнда 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:
    STORE x
    JMP start

end:
    HALT
```

Алгоритам у потпуности одговара алгоритму описаном у поглављу 5.1. Превођењем на машински језик, добили бисмо:

⁵У 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.

```
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 читава 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.