Analiza in načrtovanje bioloških vezij z ogrodjem GReNMlin ter ostale teme s področja biološkega procesiranja

Uredila prof. dr. Miha Moškon in prof. dr. Miha Mraz

Januar, 2025

Predgovor

Navodila za urejanje:

- Vaše datoteke se nahajajo v direktorijih `Skupina*`, kjer * predstavlja številko vaše skupine - glavna datoteka je textttmain.text.
- Če mape za vašo skupino še, jo ustvarite.
- Slike shranjujte v svoj direktorij.
- Vse labele začnite z znaki `g*:`, kjer * predstavlja številko vaše skupine.
- Pri referenciranju virov uporabite datoteko `references.bib`, ki se nahaja v korenskem direktoriju projekta.
- Pred dodajanjem novih virov v datoteko `references.bib` dobro preverite, če je vir mogoče že vsebovan v datoteki - v tem primeru se sklicujte na obstoječ vnos.

Seminarska dela so osredotočena na načrtovanje in optimizacijo bioloških logičnih gradnikov implementiranimi z gensko regulatornimi omrežji. Pri tem boste izhajali iz knjižnice GReNMlin[1], ki je namenjena delu s tovrstnimi gradniki. V svojih delih se lahko osredotočite na načrtovanje, analizo in optimizacijo novih bioloških logičnih vezij ali pa na razširitve predlagane knjižnice. Izberete si lahko tudi katerakoli druga orodja ali teme s področja biološkega procesiranja.

Ljubljana,
januar, 2025

prof. dr. Miha Moškon
prof. dr. Miha Mraz

[1] https://github.com/mmoskon/GRenMlin/

Zahvala

TODO

prof. dr. Miha Moškon in prof. dr. Miha Mraz, Ljubljana, januar 2025

Kazalo

Poglavje 1
Realizacija MUX in DEMUX z orodjem GReNMlin

Anže Arhar, Kristjan Kostanjšek in Nejc Ločičnik

Povzetek V seminarski nalogi obravnavamo implementacijo in analizo multiplekserjev (MUX) in demultiplekserjev (DEMUX) z uporabo orodja GReNMlin, Pythonovega paketa za modeliranje genskih regulacijskih omrežij. Predstavimo uspešne implementacije osnovnih komponent, kot sta 2:1 MUX in 1:2 DEMUX, ter njihovo širitev na bolj kompleksne različice, kot sta 4:1 MUX in 1:4 DEMUX. Poleg tega razvijemo funkcije, ki omogočajo generiranje poljubnega MUX-a ali DEMUX-a glede na število vhodnih ali izhodnih linij, ki jih uporabnik določi kot parameter funkcije. Te implementacije predstavljajo nadgradnjo orodja GReNMlin, ki omogoča večjo prilagodljivost pri generiranju različnih MUX in DEMUX struktur. Razvito funkcionalnost tudi ocenjujemo in izpostavljamo morebitne pomanjkljivosti, predvsem glede na časovno zahtevnost pri večjih strukturah. V zadnjem delu naloge preučujemo optimizacijo izhodov modelov genskih regulacijskih omrežij z uporabo genetskih algoritmov. Zaradi dolgotrajnega izvajanja in omejenega vpliva na izboljšanje rezultatov pa genetskih algoritmov nismo vključili v samo implementacijo funkcij za generiranje posplošenih MUX in DEMUX struktur.

1.1 Uvod

V uvodnem delu seminarske naloge bomo najprej opredelili temeljne pojme, ki so ključni za razumevanje obravnavane vsebine. To vključuje osnovne koncepte genskih regulacijskih omrežij (GRN) ter njihovo uporabo pri modeliranju digitalnih logičnih komponent. Poleg tega bomo predstavili orodje GReNMlin, ki omogoča

Anže Arhar
UL FRI, Večna pot 113, 1000 Ljubljana, e-mail: aa3178@student.uni-lj.si

Kristjan Kostanjšek
UL FRI, Večna pot 113, 1000 Ljubljana, e-mail: kk7609@student.uni-lj.si

Nejc Ločičnik
UL FRI, Večna pot 113, 1000 Ljubljana, e-mail: nl4952@student.uni-lj.si

simulacijo in analizo GRN, ter podali pregled osnovnih funkcionalnosti multiplekserjev in demultiplekserjev, ki predstavljajo osnovo praktičnega dela naloge.

1.1.1 Gensko regulatorno omrežje

V biološkem procesiranju je modeliranje genskih regulacijskih omrežij (GRN) ključen računalniški pristop za raziskovanje kompleksnih interakcij med geni, transkripcijskimi faktorji in vplivi iz okolja. Ta omrežja uravnavajo izražanje genov, kar omogoča celicam prilagoditev na okoljske razmere, diferenciacijo v specifične celične tipe in vzdrževanje homeostaze.

Z uporabo računalniških tehnik za modeliranje GRN lahko raziskovalci simulirajo in napovedujejo vedenje bioloških sistemov, analizirajo njihovo odpornost na motnje ter načrtujejo sintetične regulacijske sisteme za biotehnološke in medicinske aplikacije.

V zadnjih desetletjih je modeliranje genskega izražanja postalo nepogrešljivo orodje v biološkem raziskovanju. Simulacije omogočajo preverjanje hipotez o izražanju genov in bistveno zmanjšujejo potrebo po dolgotrajnih ter dragih eksperimentih. Računalniško podprto modeliranje omogoča raziskovalcem iterativno preverjanje in izboljševanje hipotez s simulacijami, s čimer se zmanjša tveganje za neuspešne eksperimente. Takšen pristop pospešuje napredek pri razumevanju bioloških procesov ter izboljšuje učinkovitost eksperimentalnega dela.

1.1.2 Orodje GReNMlin

GReNMlin (Gene Regulatory Network Modeling) je Pythonov paket, zasnovan za konstruiranje, simulacijo in analizo osnovnih logičnih komponent, kot so multiplekserji, števci in druga digitalna vezja, z uporabo genskih regulacijskih omrežij. Temelji na sistemih diferencialnih enačb, ki opisujejo dinamiko interakcij med geni in njihovimi regulatorji. Te enačbe omogočajo natančno modeliranje procesov, kot so aktivacija, represija in sinteza genov, kar uporabnikom omogoča simulacijo vedenja GRN skozi čas.

Modeliranje logičnih komponent s pomočjo GRN v GReNMlinu se začne z jasno definicijo omrežja. Pri tem je treba določiti species, torej vhodne in izhodne vrste, regulatorje, ki predstavljajo povezave med vhodnimi vrstami, in produkt, ki ustreza izhodni vrsti. Takšna struktura omogoča fleksibilno oblikovanje omrežja in povezovanje vhodov z izhodi na način, ki je skladen z biološkimi interakcijami.

Na podlagi preučenih primerov smo ugotovili, da je najpreprostejši način za realizacijo osnovnih logičnih komponent uporaba oblike Sum of Products (SOP). Pri tem vsak regulator ustreza posameznemu produktu (operacija AND), medtem ko združevanje več regulatorjev v en produkt (izhodno vrsto) ustreza njihovi vsoti (operacija OR). Inverzijo tipa regulatorja je mogoče uporabiti za implementacijo

funkcije NOR, kar omogoča dodatno prilagodljivost pri oblikovanju logičnih komponent.

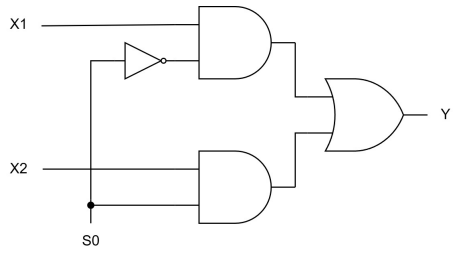
Ta pristop omogoča preprost in pregleden način za konstruiranje osnovnih logičnih komponent s pomočjo GRN. Poleg tega zagotavlja čiste signale že privzeto, brez potrebe po optimizaciji parametrov, kot sta k_d ali n. To pomeni, da lahko uporabniki z uporabo GReNMlin dosežejo stabilne in zanesljive simulacije brez dolgotrajnih prilagoditev parametrov.

1.1.3 Multiplekser

Multiplekser je osnovno logično vezje, ki ima ključno vlogo v računalniškem svetu. Njegova zasnova vključuje 2^n vhodnih signalov, n adresnih signalov (imenovanih tudi kontrolni signali ali izbirne linije) in en izhodni signal. Njegovo osnovno delovanje temelji na izbiri enega izmed vhodnih signalov na podlagi vrednosti adresnih signalov ter preslikavi izbranega signala na izhod.

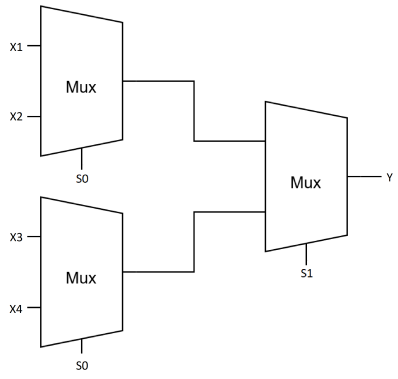
Uporaba multiplekserjev je široka in vključuje številna področja. V digitalnih sistemih se uporabljajo za usmerjanje signalov, optimizacijo vodil in zmanjševanje števila potrebnih povezav. V procesorskih enotah so ključni za izbiro med različnimi viri podatkov ali ukazov, medtem ko v komunikacijskih sistemih omogočajo združevanje več signalov na eno prenosno pot, kar povečuje učinkovitost prenosa podatkov.

Na sliki (1.1) je prikazana osnovna logična zasnova multiplekserja, izvedena z uporabo logičnih vrat NOT, AND in OR. Prav ta zasnova je služila kot osnova za našo realizacijo multiplekserja z uporabo orodja GReNMlin.



Slika 1.1 Osnovna realizacija multipleksorja (AND, OR, NOT).

Poleg raznih aplikacij, multiplekserji predstavljajo poln funkcijski nabor, kar pomeni, da z njihovo uporabo lahko implementiramo katerokoli logično funkcijo. Z združevanjem več osnovnih multiplekserjev, kot so 2:1, lahko ustvarimo večje in zmogljivejše konfiguracije, na primer 4:1, 8:1 in še večje. Na sliki (1.2) je prikazan primer, kako lahko kombinacija osnovnih multiplekserjev vodi do izgradnje kompleksnejšega multiplekserja.



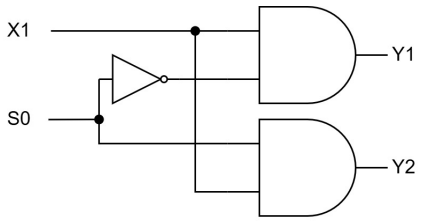
Slika 1.2 Kaskadno združevanje multiplekserjev.

1.1.4 Demultiplekser

Demultiplekser (DEMUX) je logično vezje, ki deluje obratno od multiplekserja, vendar temelji na podobnem principu uporabe adresnih signalov. Medtem ko multiplekser omogoča izbiro enega izmed več vhodnih signalov in njegovo usmeritev na izhod, demultiplekser prejme en vhodni signal in ga preusmeri na enega izmed več izhodnih signalov. Njegova zasnova vključuje en vhodni signal, n adresnih signalov in 2^n izhodnih signalov.

Demultiplekserji imajo ključno vlogo v digitalnih sistemih, kjer je potrebno razporediti podatke ali signale iz enega vira na več ciljev. Pogosto se uporabljajo v komunikacijskih sistemih za distribucijo signalov v več kanalov, pri dekodiranju naslovov v pomnilniških enotah in v procesorjih za upravljanje z ukazi.

Na sliki (1.3) je prikazana osnovna logična zasnova demultiplekserja, izvedena z uporabo logičnih vrat NOT in AND. Prav ta zasnova je bila osnova za implementacijo demultiplekserja z uporabo orodja GReNMlin.



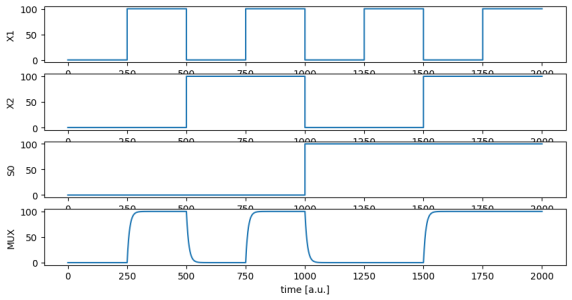
Slika 1.3 Osnovna realizacija demultipleksorja (AND, OR, NOT).

1.2 Metode

1.2.1 Realizacija multiplekserja

2:1 multiplekser smo z orodjem GReNMlin implementirali na podlagi zasnove, prikazane na sliki (1.1). Implementacija osnovnega 2:1 multiplekserja je že od začetka delovala brez težav, kar potrjujejo tudi rezultati na sliki (1.4). Ta uspešna zasnova je služila kot temelj za razvoj bolj kompleksnih multiplekserjev, kot sta 4:1 in 8:1 MUX, pri katerih prav tako nismo naleteli na nobene težave.

Na podlagi teh uspešnih rezultatov se izkazuje velik potencial za razvoj funkcije posplošenega multiplekserja. Takšna funkcija bi omogočala generiranje multiplekserjev poljubne kompleksnosti glede na število vhodnih linij, ki jih določi uporabnik. Posplošeni multiplekser bi tako lahko predstavljal pomembno razširitev in izboljšavo obstoječega orodja GReNMlin.



Slika 1.4 Rezultati simulacije 2:1 multipleksorja.

1.2.1.1 Posplošen MUX

Posplošen MUX smo razvili po izrazu:

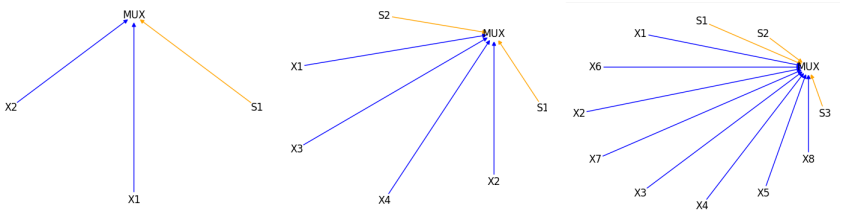
$$Y = (X_1 \cdot \overline{S}_1 \cdot \overline{S}_2 \cdot \ldots \cdot \overline{S}_{\log_2 N}) + (X_2 \cdot S_1 \cdot \overline{S}_2 \cdot \ldots \cdot \overline{S}_{\log_2 N}) + \ldots + (X_N \cdot S_1 \cdot S_2 \cdot \ldots \cdot S_{\log_2 N})$$

Spodnji izsek programske kode prikazuje logiko delovanja funkcije, ki generira poljuben MUX, definiran s parametrom N, ki predstavlja število vhodnih linij. Na začetku preverimo, ali je uporabnik za parameter N vnesel potenco števila 2, ki je večja od 1 (torej 2, 4, 8, 16, ...). Nato definiramo ustrezno število vhodnih in izhodnih vrst (angl. species), na koncu pa generiramo željeni MUX, kot je bilo definirano zgoraj.

```
# Check if N is power of 2
if (N < 2) or not (N and (N & (N - 1)) == 0):
```

```
        raise ValueError("N must be a power of 2
            and larger than 1 (e.g., 2, 4, 8, 16, ...)")

    num_select_lines = int(np.log2(N))

    # Initialize GRN
    my_grn = grn.grn()

    # Add input species for data lines X1, X2...
    for i in range(1, N + 1):
        my_grn.add_input_species(f"X{i}")

    # Add input species for select lines S1, S2...
    for i in range(1, num_select_lines + 1):
        my_grn.add_input_species(f"S{i}")

    # Add output species for MUX
    my_grn.add_species("MUX", 0.1)

    # MUX logic
    for i in range(1, N + 1):
        # Determine binary representation
        binary_select =
            f"{i - 1:0{num_select_lines}b}"[::-1]

        # Define regulators for the current gene
        regulators = [{"name": f"X{i}", "type": 1,
            "Kd": 5, "n": 2}]
        for j, bit in enumerate(binary_select):
            regulators.append({
                "name": f"S{j + 1}",
                "type": 1 if bit == "1" else -1,
                "Kd":   5,
                "n":    3
            })

        # Define products for the current gene
        products = [{"name": "MUX"}]

        # Add the gene to the GRN
        my_grn.add_gene(10, regulators, products)
```

Testiranje funkcije za posplošen multiplekser je pokazalo, da funkcija deluje pravilno in v skladu s pričakovanji. Omrežja, pridobljena z izvajanjem funkcije za različne vrednosti parametra N, so prikazana na sliki 1.5.



Slika 1.5 Omrežja 2:1, 4:1 in 8:1 multipleksorja.

Edina težava, na katero smo naleteli, je strmo naraščanje časa simulacije s povečevanjem parametra N (glej tabelo 1.1). Medtem ko je simulacija 2:1 in 4:1 multiplekserja trajala manj kot sekundo, je simulacija 8:1 multiplekserja že zahtevala 30 sekund. Izvedba 16:1 multiplekserja pa je trajala tako dolgo, da nismo uspeli izmeriti časa. Po sedmih urah delovanja funkcije še vedno ni bilo videti, da bi se simulacija kmalu zaključila, zato smo jo ročno prekinili.

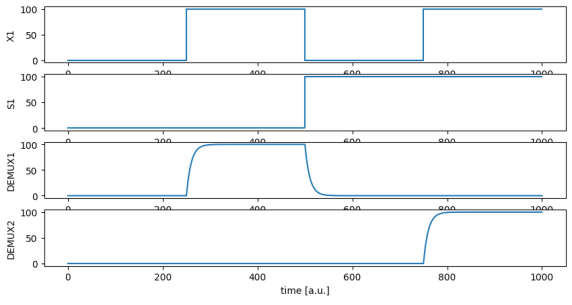
Zaradi tega bi bilo smiselno funkcijo za posplošen multiplekser uporabljati predvsem za osnovnejše multiplekserje (2:1, 4:1 in 8:1 MUX).

N	Čas za izvedbo [s]
2	0.12
4	0.43
8	33.86

Tabela 1.1 Čas izvedbe glede na velikost N multipleksorja

1.2.2 Realizacija demultipleksorja

2:1 DEMUX smo implementirali po sliki (1.3). Rezultati testiranja so prikazani na sliki (1.6).



Slika 1.6 Rezultati 1:2 demultipleksorja.

Tudi pri implementaciji demultiplekserjev nismo naleteli na nobene težave. Brez problemov smo razvili tudi kompleksnejše demultiplekserje (4:1, 8:1, ...). Tokrat smo ponovno prepoznali potencial za razvoj funkcije, ki bi generirala posplošene demultiplekserje. Takšna funkcija bi se, poleg posplošenega multiplekserja, lahko vključila kot izboljšava orodja GReNMlin.

1.2.2.1 Posplošen DEMUX

Posplošen DEMUX smo razvili po sledečih izrazih:

$$Y_1 = X_1 \cdot \overline{S}_1 \cdot \overline{S}_2 \cdot \ldots \cdot \overline{S}_{\log_2 N}$$

$$Y_2 = X_1 \cdot S_1 \cdot \overline{S}_2 \cdot \ldots \cdot \overline{S}_{\log_2 N}$$

$$\vdots$$

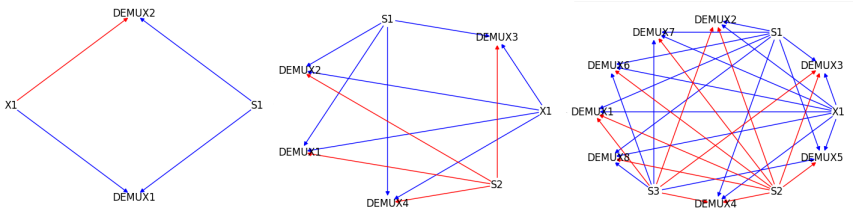
$$Y_N = X_1 \cdot S_1 \cdot S_2 \cdot \ldots \cdot S_{\log_2 N}$$

Spodnji izsek programske kode prikazuje našo implementacijo funkcije. Funkcija deluje na zelo podoben način kot funkcija za posplošen MUX. Najprej preverimo, ali je uporabnik vnesel ustrezen parameter N, nato definiramo vhodne in izhodne vrste. Sledi DEMUX logika, ki na podlagi zgoraj definiranega logičnega izraza ustrezno nastavi regulatorje.

```
# Check if N is power of 2
if (N < 2) or not (N and (N & (N - 1)) == 0):
    raise ValueError("N must be a power of 2
        and larger than 1 (e.g., 2, 4, 8, 16, ...)")

num_select_lines = int(np.log2(N))

# Initialize the GRN
my_grn = grn.grn()

# Add input species X1 and S1, S2, ..., Slog2(N)
my_grn.add_input_species("X1")
for i in range(1, num_select_lines + 1):
    my_grn.add_input_species(f"S{i}")

# Add output species for each Y_i
for i in range(1, N + 1):
    my_grn.add_species(f"DEMUX{i}", 0.1)

# DEMUX logic
for i in range(1, N + 1):
    # Determine binary representation
```

```
binary_select =
    f"{i - 1:0{num_select_lines}b}"[::-1]

# Define regulators
regulators = [{'name': 'X1', 'type': 1,
    'Kd': 5, 'n': 2}]
for j, bit in enumerate(binary_select):
    regulators.append({
        "name": f"S{j + 1}",
        "type": 1 if bit == "1" else -1,
        "Kd": 5,
        "n": 3
    })

# Output gene for DEMUX_i
products = [{'name': f"DEMUX{i}"}]

# Add the gene to the GRN
my_grn.add_gene(10, regulators, products)
```

Tudi tokrat je funkcija delovala skladno s pričakovanji. Omrežja, ki jih pridobimo z izvajanjem funkcije za različne vrednosti parametra N, so prikazana na sliki 1.7.



Slika 1.7 Omrežja 1:2, 1:4 in 1:8 demultipleksorja.

Poleg tega je funkcija delovala hitreje kot funkcija za posplošen MUX. Čas izvedbe je sicer še vedno naraščal s parametrom N, vendar tokrat manj strmo. Čas za izvedbo funkcije glede na velikost parametra N je podan v spodnji tabeli (1.2).

N	Čas za izvedbo [s]
2	0.11
4	0.14
8	0.36
16	2.12
32	16.22

Tabela 1.2 Čas izvedbe glede na velikost N demultipleksorja

1.2.3 Optimizacija parametrov

Za izboljšanje naših implementacij smo uporabili genetske algoritme. Ti so nam omogočili optimizacijo parametrov K_d in n regulatornih elementov. Simulacije agentov v algoritmu smo ocenili s cenilno funkcijo.

$$f(s) = \frac{1}{\sum_{i=1}^{N} |s_i - o_i| + \varepsilon}$$

Kjer N predstavlja število diskretnih korakov simulacije in $\varepsilon = 10^{-6}$, cenilna funkcija izračuna absolutno razliko med optimalno (o) in simulirano (s) vrednostjo na diskretnih korakih.

Z implementacijo optimizacije z genetskimi algoritmi smo dosegli le nekoliko boljše rezultate v primerjavi z ročno optimizacijo parametrov. Predvidevamo, da bi za boljše rezultate morali bistveno povečati število generacij in velikost populacije. Po začetnih poskusih povečanja kateregakoli parametra na primeru 2:1 MUX, pa smo ugotovili, da se čas optimizacije povečuje in lahko traja več ur. Na koncu smo se odločili, da naši rezultati ustrezajo zastavljenim ciljem, in da dodatna optimizacija parametrov ne prinese dovolj pomembnih izboljšav v primerjavi z vloženim časom.

1.3 Zaključek

V seminarski nalogi smo v programskem jeziku Python s pomočjo orodja GReNMlin implementirali osnovne strukture multiplekserjev in demultiplekserjev. Zaradi uspešnega delovanja teh osnovnih struktur smo razvili funkcije, ki omogočajo samodejno generiranje želenih MUX in DEMUX struktur glede na parametre, ki jih določi uporabnik. Medtem ko funkcija za generiranje DEMUX struktur uspešno zgenerira tudi kompleksne demultiplekserje (npr. 1:32 DEMUX) v sprejemljivem času, smo pri multiplekserjih naleteli na težave. Simulacija bolj kompleksnih multiplekserjev, kot je 16:1 MUX, je postala preveč časovno zahtevna, kar omejuje njihovo praktično uporabnost.

Poleg tega smo raziskali uporabo genetskih algoritmov za optimizacijo izhodov multiplekserjev in demultiplekserjev. Čeprav ti algoritmi ponujajo možnosti za izboljšave, jih zaradi dolgotrajnega izvajanja in relativno majhnega vpliva na kakovost rezultatov nismo vključili v implementacijo.

V prihodnjem razvoju bi bilo smiselno raziskati metode za hitrejše generiranje kompleksnejših multiplekserjev in učinkovitejšo optimizacijo parametrov, kar bi omogočilo dosego bolj optimalnih rezultatov v realnem času.

1.4 Doprinosi avtorjev

Anže Arhar	Optimizacija parametrov
Kristjan Kostanjšek	Posplošitev MUX in DEMUX
Nejc Ločičnik	Osnovni 2-8:1 MUX in 1:2-4 DEMUX

Literatura