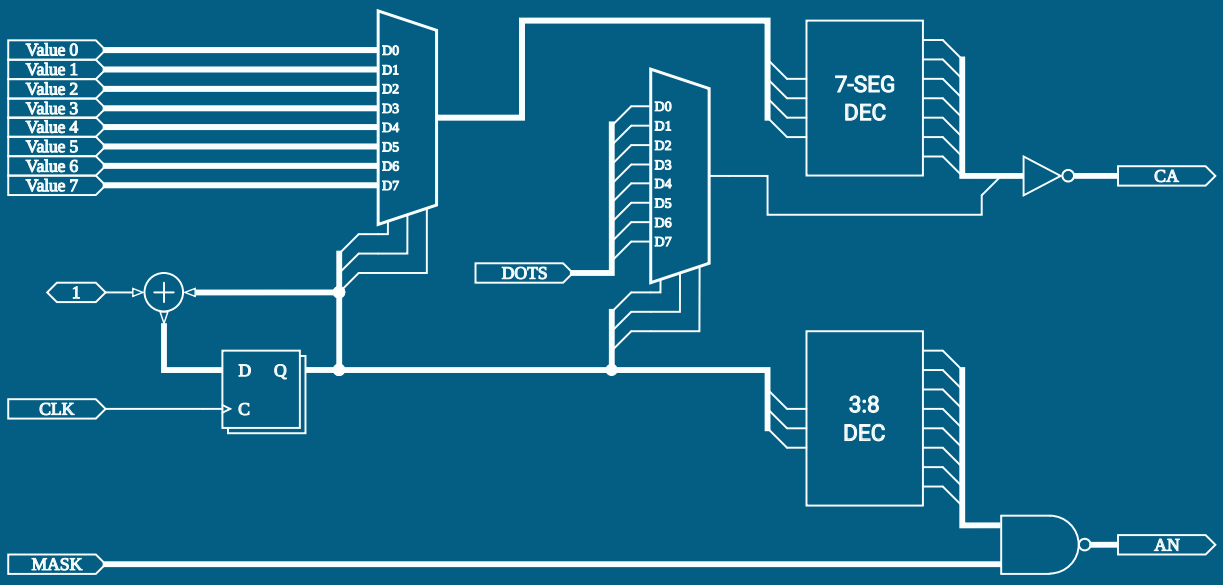
# DIGITALNE STRUKTURE IN SISTEMI

## Laboratorijske vaje




Gregor Donaj

Univerzitetna založba Univerze v Mariboru

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko

# DIGITALNE STRUKTURE IN SISTEMI

Laboratorijske vaje

Avtor
**Gregor Donaj**

Maribor, 2023

**Naslov** **Digitalne strukture in sistemi**
*Title* *Digital Structures and Systems*

**Podnaslov** **Laboratorijske vaje**
*Subtitle* *Laboratory Exercises*

**Avtor** Gregor Donaj
*Author* (Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko)

**Recenzija** Iztok Kramberger
*Review* (Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko)

**Jezikovni pregled** Valerija Vegič
*Language editing*

**Tehnična urednika** Gregor Donaj
*Technical editors* (Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko)

Jan Perša
(Univerza v Mariboru, Univerzitetna založba)

**Oblikovanje ovitka** Gregor Donaj
*Cover designer* (Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko)

**Grafika na ovitku** Avtor, 2023
*Cover graphics*

**Grafične priloge** Vse grafike v publikaciji so
*Graphic material* avtorske. Donaj, 2023

**Založnik** **Univerza v Mariboru**
*Published by* **Univerzitetna založba**
Slomškov trg 15
2000 Maribor, Slovenija
https://press.um.si,
zalozba@um.si

**Izdajatelj** **Univerza v Mariboru**
*Issued by* **Fakulteta za elektrotehniko, računalništvo in informatiko**
Koroška cesta 46
2000 Maribor, Slovenija
https://feri.um.si, feri@um.si

**Izdaja** Prva izdaja
*Edition*

**Izdano** Maribor, marec 2023
*Published at*

**Vrsta publikacije** E-knjiga
*Publication type*

**Dostopno na** https://press.um.si/index.php/ump/catalog/book/767
*Availabe at*

CIP - Kataložni zapis o publikaciji
Univerzitetna knjižnica Maribor

004.3'144:621.3.049.75(076)(0.034.2)

DONAJ, Gregor DONAJ, Gregor
Digitalne strukture in sistemi [Elektronski vir] : laboratorijske vaje / avtor Gregor Donaj. - 1. izd. - E-knjiga. - Maribor : Univerza v Mariboru, Univerzitetna založba, 2023

Način dostopa (URL): https://press.um.si/index.php/ump/catalog/book/767.
ISBN 978-961-286-719-5
doi: 10.18690/um.feri.3.2023
COBISS.SI-ID 144854019



**ISBN** 978-961-286-719-5 (pdf)

**DOI** https://doi.org/10.18690/um.feri.3.2023

**Cena** Brezplačni izvod
*Price*

**Odgovorna oseba založnika** prof. dr. Zdravko Kačič,
*For publisher* rektor Univerze v Mariboru

**Citiranje** Donaj, G. (2023). Digitalne strukture in sistemi: Laboratorijske vaje. Univerza v Mariboru, Univerzitetna založba
*Attribution* doi: https://doi.org/10.18690/um.feri.3.2023

# Kazalo

# 1 Uvod

Navodila za laboratorijske vaje pri predmetu *Digitalne strukture in sistemi* so namenjena praktičnemu spoznavanju delovanja nekaterih digitalnih vezij, izvedbe teh vezij v programirljivih vezjih FPGA in simulacije digitalnih vezij. Pri tem je opisan tudi jezik za opisovanje strojne opreme Verilog, s katerim opisujemo vezja in poteke simulacij. Navodila se lahko uporabijo tudi pri vajah drugih predmetov s podobno vsebino.

V naslednjih treh poglavjih navodil je na začetku poglavja kratki teoretični uvod v obravnavano tematiko. Nato sledijo rešeni primeri implementacij vezij ali simulacij v jeziku Verilog. Teoretični opis in rešeni primeri so namenjeni samostojnemu delu študentov – pripravam na vaje. Med rešenimi primeri so opisani tudi posamezni ukazi in konstrukti jezika Verilog.

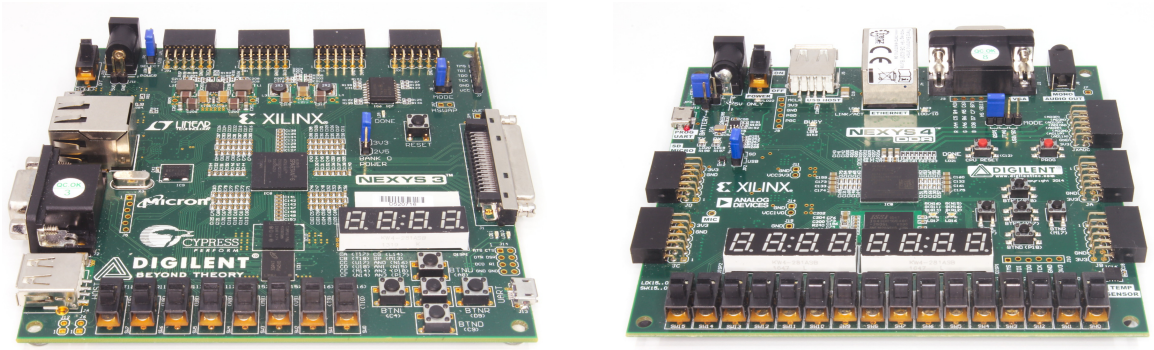
Kontaktne ure laboratorijskih vaj so namenjene kratki ponovitvi teoretičnega uvoda in primerov ter reševanju vaj, ki sledijo ob koncu vsakega poglavja. Vaje so podobne rešenim primerom, kjer je dodana implementacija rešenih vaj na razvojni plošči oz. izvedba simulacije v uporabljenem razvojnem okolju.

V petem poglavju sledijo vaje iz primerov nekaterih praktičnih vezij, v okvirju katerih so obravnavane nekatere dodatne tematike.

Navodila ne nudijo splošnega pregleda digitalne elektronike ali pa jezika Verilog, temveč le osnove za opravljanje laboratorijskih vaj. Dodatno znanje iz digitalne tehnike in temeljitejši pregled jezika Verilog najdemo v navedeni literaturi ob koncu navodil.

## 1.1 Strojna in programska oprema

Za izvedbo vaj potrebujemo ustrezno opremo. Strojna oprema je razvojna plošča za vezja FPGA, na kateri je prisotna tudi periferija: stikala, tipke, svetleče diode, večmestni 7-segmentni prikazovalnik in oscilator za uro. Primeri takšnih plošč, ki jih uporabljamo na vajah, so Digilent Nexys 3, na kateri je vezje FPGA iz družine Xilinx Spartan 6, ali pa Digilent Nexys 4DDR, na kateri je vezje iz družine Xilinx Artix 7. Razvojni plošči sta prikazani na sliki 1.




Slika 1: Razvojni plošči Digilent Nexys 3 (levo) in Digilent Nexys 4DDR (desno)

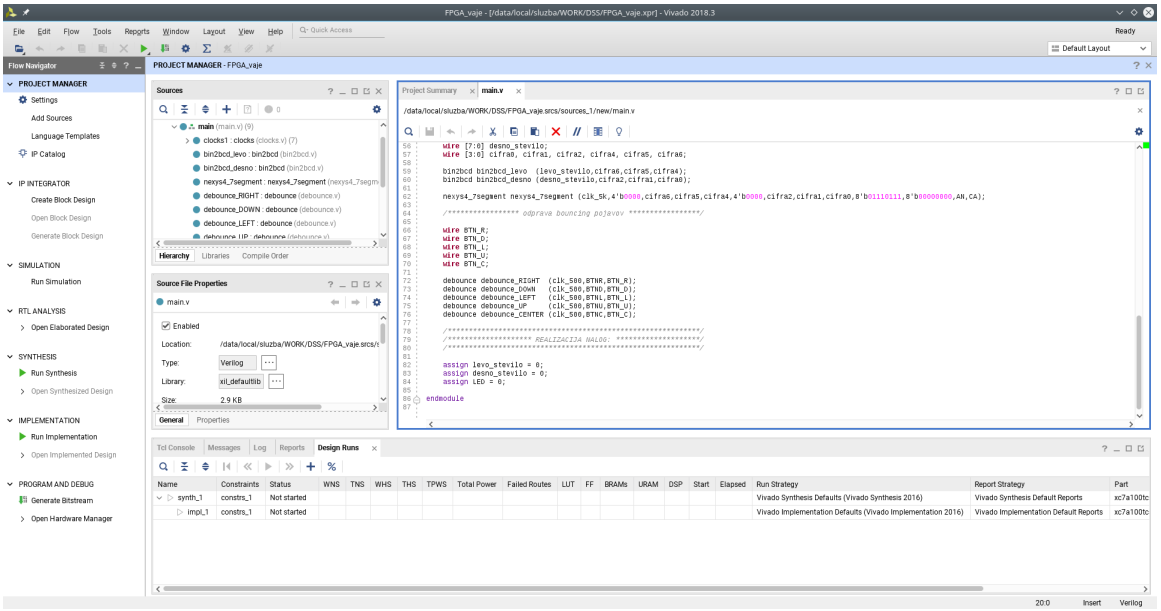
Na spletni strani proizvajalca lahko najdemo tehnično dokumentacijo, ki med drugim vključuje shemo plošče, podporne datoteke za načrtovanje vezij in primere projektov[1,2].

Programska oprema je razvojno okolje, ki omogoča opisovanje in simulacijo digitalnih vezij, ter programiranje vezij FPGA. Uporabljamo orodje Xilinx ISE Design Suite za starejše generacije

[1] `https://reference.digilentinc.com/reference/programmable-logic/nexys-3/start`
[2] `https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/start`

vezij (npr. Nexys 3 z vezjem Spartan 6) ali pa Xilinx Vivado Design Suite za novejše generacije (npr. Nexys 4DDR z vezjem Artix 7). Obe okolji sta prosto dostopni z omejeno funkcionalnostjo, kar je dovolj za izvedbo vaj.

Za izvedbo laboratorijskih vaj se lahko v razvojnem okolju uporablja en projekt, kjer se vsaka vaja napiše kot samostojni modul. Te module pa nato posamezno vključimo v glavni modul, da lahko preverimo njihovo delovanje. Na sliki 2 je prikazana zaslonska slika razvojnega okolja, v katerem je odprt projekt za laboratorijske vaje.



Slika 2: Zaslonska slika orodja Xilinx Vivado Design Suite

## 1.2 Verilog

Verilog je jezik za opisovanje strojne opreme (angl. Hardware Description Language – HDL). Uporabljamo ga za opisovanje delovanja digitalnih vezij na različnih ravneh abstrakcije, od nižjih ravneh opisovanja posameznih tranzistorjev do algoritmičnega opisovanja delovanja vezij. Na vajah bomo uporabljali višje ravni abstrakcije opisovanja.

V Verilogu opisujemo delovanje vezij tako, da opis delimo na module, znotraj katerih opišemo delovanje notranjih signalov in izhodov glede na vhode. Pri tem uporabimo razne ukaze in konstrukte Veriloga, ki bodo opisani v rešenih primerih.

Verilog se uporablja tudi za opisovanje testnih primerov, s katerimi v obliki simulacije preverimo pravilno delovanje vezja, ki smo ga opisali (načrtovali). Za ta namen vsebuje Verilog tudi konstrukte, ki se upoštevajo le pri simulacijah.

Opisovanje signalov pomeni, da jim priredimo vrednost na podlagi vrednosti drugih signalov. Pri tem lahko uporabimo različne operatorje: logične, bitne, aritmetične, primerjalne itd. Za razumevanje delovanja operatorjev je priporočljivo splošno predznanje programiranja, saj je uporaba operatorjev podobna uporabi v programskih jezikih. Sama sintaksa jezika Verilog pa je nekoliko podobna sintaksi programskega jezika C.

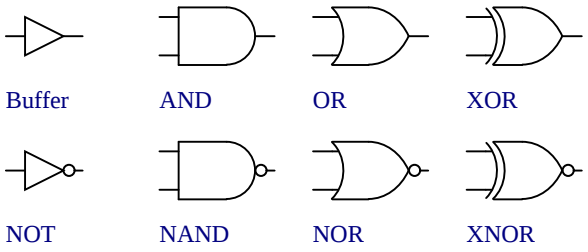
# 2 Preklopna vezja

Logična vezja lahko v splošnem delimo na dve skupini: preklopna vezja in sekvenčna vezja. Značilnost preklopnih vezij je ta, da so vrednosti izhodov odvisne le od trenutnih vrednosti vhodov. Pravimo tudi, da takšna vezja nimajo spomina.

Vezja imajo lahko enega ali več izhodov. Da opišemo delovanje vezja, moramo nekako določiti delovanje vsakega izhoda ob vseh možnih kombinacijah vrednosti signalov na vhodu. To lahko storimo v obliki pravilnostne tabele, opisa delovanja izhoda z logično funkcijo ali pa tudi podamo shemo vezja, iz katere se lahko razbere njegovo delovanje.

Ker je delovanje vezja odvisno le od trenutnih vrednosti na vhodih, se bodo izhodi lahko spreminjali le v trenutkih, ko se vsaj en vhod spremeni. Odvisno od delovanja pa lahko tudi izhodi pri določenih spremembah vhodov ostanejo nespremenjeni. V realnih vezjih ta sočasnost ne velja popolnoma, saj prihaja do kratkih zakasnitev zaradi preklopnih časov tranzistorjev, iz katerih so sestavljena vrata, in omejenih hitrosti signalov. Lahko pa v vezjih nastajajo statični ali dinamični hazardi (neželene spremembe signalov), ki pa jih sicer v tem poglavju ne bomo obravnavali.

Preklopna vezja so sestavljena iz množice logičnih vrat, kjer vsak tip vrat izvaja neko logično operacijo. Da bodo takšna vezja preklopna, se v njih ne smejo pojavljati povratne vezave. To pomeni, da signal na izhodu nekih vrat ne sme (neposredno ali čez druga vrata) biti speljan spet na enega od vhodov teh vrat.



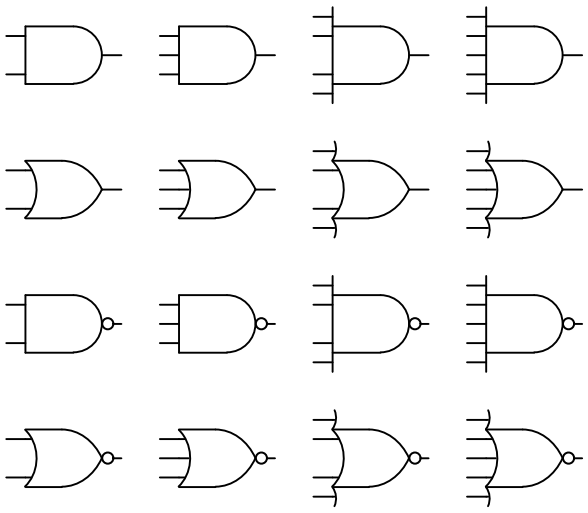

Slika 3: Različni tipi logičnih vrat

Za logična vrata poznamo različne standarde simbolov. Na sliki 3 so prikazani simboli po standardu ANSI/IEEE. Logična operacija, ki jo izvajajo vrata, je ponazorjena z obliko simbola. Na levi strani simbolov so vedno vhodi, na desni pa je izhod. Čeprav lahko simbole pri risanju shem obračamo in pri tem še vedno prepoznamo vhode in izhode, poteka risanje shem tako, da se simboli večinoma rišejo v zgoraj prikazani orientaciji ter da signali na shemi potujejo od leve proti desni.

V zgornji vrstici slike vidimo simbole za 4 vrata. Prva vrata so neinvertirajoči vmesnik (angl. buffer), kjer je izhod vedno enak vhodu. Naslednja so vrata IN (angl. AND), kjer je izhod enak 1, kadar so vsi vhodi enaki 1, sicer pa je izhod enak 0. Rečemo *vsi* vhodi, saj imajo takšna vrata lahko tudi več kot dva vhoda. Podobno velja za preostala vrata. Naslednja so vrata ALI (angl. OR), kjer je izhod enak 1, kadar je vsaj en vhod enak 1, sicer pa je izhod enak 0. Zadnja prikazana vrata so vrata ekskluzivni ALI (angl. exclusive OR = XOR), kjer je izhod enak ena, če je en ali drugi vhod enak 1, ne pa, če sta oba vhoda enaka 1. Delovanje vrat XOR lahko posplošimo na več vhodov. Tako lahko povemo, da je izhod enak 1, kadar je liho število vhodov enako 1, sicer pa je izhod enak 0.

V drugi vrstici slike vidimo še štiri vrata, kjer pa je na simbolu dodan krogec pri izhodu. Krogec nakazuje negacijo signala in se pri nekaterih simbolih lahko pojavi tudi na enem ali več vhodih. Negacija pomeni, da se vrednost spremeni, torej vrednost 1 se spremeni v 0, vrednost 0 pa se spremeni v 1.

Vrata v drugi vrstici so negator (angl. NOT), negiran IN (angl. NAND), negiran ALI (angl. NOR) in ekskluzivni negiran ALI (angl. XNOR). Delovanje teh vrat dobimo tako, da vzamemo delovanje ustreznih vrat brez negacije in spremenimo vrednost izhodnega signala.



Slika 4: Logična vrata z različnimi števili vhodov

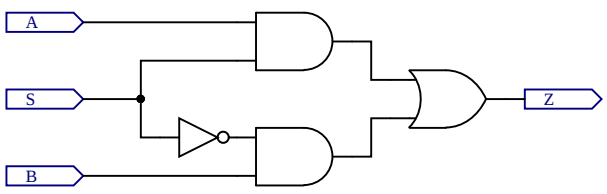
Na sliki 4 vidimo primere simbolov vrat z različnim številom vhodov. Tukaj nimamo neinvertirajočega vmesnika in negatorja, saj imata ta lahko le en vhod.

S pomočjo logičnih vrat lahko sestavimo preklopno vezje glede na poljubno želeno delovanje. Poznamo pa tudi posebna preklopna vezja, ki izvajajo specifične funkcije. Primeri takšnih vezij so: multiplekser, dekodirnik, prioritetni kodirnik, drugi tipi podatkovnih pretvornikov, aritmetična vezja, bralni pomnilnik itd.

## 2.1 Posebna vezja

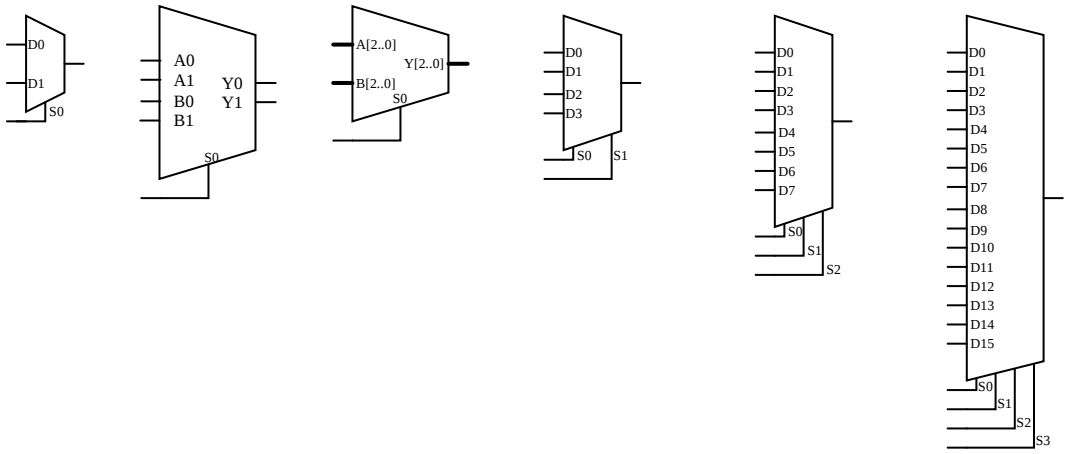
Multiplekser je vezje, ki ima enobitni ali večbitni naslovni vhod, več podatkovnih vhodov in en podatkovni izhod. Multiplekser deluje kot podatkovno preklopno stikalo, kjer izberemo enega od podatkovnih vhodov, njegovo vrednost pa preslikamo na izhod. Vrednost signala na naslovnem vhodu določa izbran vhod.

Kadar uporabljamo diskretne elemente, je število podatkovnih bitov običajno potenca števila 2 (2, 4, 8 ...), število bitov v naslovnem vhodu pa je enako stopnji potence, s katero dobimo število podatkovnih vhodov.




Slika 5: Preprosti multiplekser 2:1

Na sliki 5 je prikazan preprosti multiplekser, kjer s signalom `S` izbiramo, ali se bo vrednost signala `A` ali `B` pojavila na izhodu `Z`. Multiplekserje z več vhodi dobimo z vezji, ki vsebujejo več medsebojno povezanih kopij vezja na sliki.




Slika 6: Simboli multiplekserjev

Na sliki 6 so prikazani razni možni simboli različnih multiplekserjev. Prvi simbol prikazuje multiplekser 2:1, kjer izbiramo med dvema vhodoma. V tem primeru potrebujemo le en bit kot naslovni signal. V drugem simbolu je prav tako prikazan multiplekser 2:1, le da v tem primeru imamo podatkovna vhoda, ki sta sestavljena iz dveh signalov oz. dveh bitov. Posledično ima seveda tudi podatkovni izhod dva signala oz. dva bita. V tretjem simbolu imamo podobni multiplekser, kjer so podatkovni signali 3-bitni, ampak tokrat imamo podatkovne signale označene z eno debelejšo črto, iz zapisa na simbolu pa je razvidno število bitov. Oznaka prikazuje obseg številk bitov v signalu. Oznaka je `[2..0]` (štejemo od 0 naprej z desne proti levi) in nakazuje, da imamo 3 bite. Preostali trije simboli prikazujejo multiplekserje 4:1, 8:1 in 16:1, ki imajo 2, 3 in 4 bite v naslovnem vhodu.

Dekodirnik je preklopno vezje s podatkovnim vhodom in podatkovnim izhodom, kjer je število bitov v izhodu tipično večje od števila bitov na vhodu. Naloga dekodirnika je pretvorba podatka med različnimi kodiranji.

Enostavni dekodirnik ima $n$-bitni vhod in $2^n$-bitni izhod. Vhod lahko zavzame vseh $2^n$ možnih vrednosti (kombinacij vrednosti posameznih bitov). Izhod je kodiran tako, da imajo vsi biti enako vrednost, razen enega. Kateri od teh bitov ima drugačno vrednost, je določeno z vrednostjo na vhodu. Dodatno lahko ima dekodirnik tudi vhod za omogočanje delovanja in druge krmilne vhode. Delovanje dekodirnikov lahko preprosto ponazorimo s pravilnostno tabelo.

Tabela 1: Pravilnostna tabela dekodirnika 2 na 4

| Vhod | | Izhod | | | |
|---|---|---|---|---|---|
| X[1] | X[0] | Y[3] | Y[2] | Y[1] | Y[0] |
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

Primer pravilnostne tabele za dekodirnik 2 na 4 je prikazan v tabeli 1, kjer je `X` vhodni podatek, `Y` pa je izhodni podatek. Številke v oglatih oklepajih označujejo številko bita v obeh podatkih. Iz tabele lahko prepoznamo delovanje dekodirnika. Vhodni podatek predstavlja binarno število. Glede na njegovo vrednost je v izhodnem podatku ustrezni bit postavljen na vrednost 1, medtem ko so preostali biti postavljeni na 0. Na tak način nosita tako vhodni kot izhodni signal isti podatek (v tem primeru število med 0 in 3), kjer pa je razlika le v obliki kodiranja tega podatka.

Kodirniki delujejo v nasprotni smeri kot dekodirniki. Imajo več bitov na vhodu kot pa na izhodu. Pogledamo, na katerem bitu na vhodu se pojavi vrednost 1, medtem ko se na drugih bitih pojavi vrednost 0. Glede na položaj te enice nato nastavimo izhod kot binarno število, ki predstavlja ta položaj. Ponovno je to vezje, ki ima na vhodu in na izhodu enak podatek v dveh različnih kodiranjih.

Pri kodirniku imamo več kombinacij vhodov kot pa izhodov, še vedno pa moramo za vsako vhodno kombinacijo določiti izhod. Posledično bo več vhodnih kombinacij moralo imeti isti izhod. Vprašamo se, kako naj kodirnik deluje, kadar se na več kot enem bitu na vhodu pojavi vrednost 1. V tem primeru damo prednost najbolj levi ali najbolj desni enici. To je t. i. prioritetni kodirnik.

Tabela 2: Pravilnostna tabela prioritetnega kodirnika 8 na 3 s prioriteto na levi in izhodom za veljaven vhod

| Vhod | | | | | | | | Izhod | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| X[7] | X[6] | X[5] | X[4] | X[3] | X[2] | X[1] | X[0] | Y[2] | Y[1] | Y[0] | Veljavnost |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | x | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 1 | x | x | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 | 1 | x | x | x | 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 | x | x | x | x | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | x | x | x | x | x | 1 | 0 | 1 | 1 |
| 0 | 1 | x | x | x | x | x | x | 1 | 1 | 0 | 1 |
| 1 | x | x | x | x | x | x | x | 1 | 1 | 1 | 1 |

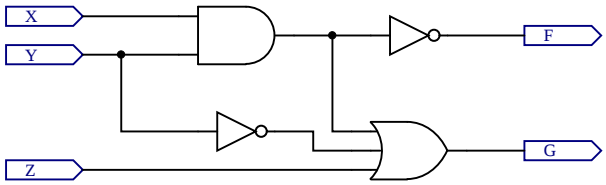
V tabeli 2 je primer pravilnostne tabele prioritetnega kodirnika, ki ima dodatni izhod. Ta pove, ali je izhodna vrednost veljavna. Vrednost ni veljavna le v primeru, da se na nobenem vhodu ne pojavi enica. Kadar se pojavi na vhodu vsaj ena enica, je prioriteta na levi (bolj uteženi) strani. Vsi vhodni biti, ki so desno od kakšne enice, več ne vplivajo na izhodno vrednost. Vrednost `x` v tabeli predstavlja *don't care*, kar pomeni, da je lahko vrednost tako izraženega bita 0 ali 1.

Poznamo tudi druge tipe vezij, ki na neki način spreminjajo kodiranje podatkov, kot so 7-segmentni dekodirnik ali pa kodirnik med binarno in Grayevo kodo. Ta in še nekatera druga vezja bomo v nadaljevanju spoznali pri primerih in vajah.

Pri tem se ne bomo ukvarjali s shemami teh vezij in z vprašanjem, kako takšna vezja sestavimo iz posameznih vrat, temveč bomo obravnavali le njihov opis v jeziku Verilog.

## 2.2 Primeri

**Primer 1.** Implementirati želimo preprosto preklopno vezje na sliki 7 kot modul v jeziku Verilog. Rešitev je prikazana v kodi 1.




Slika 7: Primer preprostega preklopnega vezja (1)

```
module primer1(
  input X,
  input Y,
  input Z,
  output F,
  output G
  );

  assign F = ~(X & Y);
  assign G = (X & Y) | (~Y) | Z;

endmodule
```

Izvorna koda 1: Rešitev primera s slike 7

V rešitvi vidimo, kako je sestavljen modul. Začnemo ga z ukazom **`module`**, ki mu sledi ime modula. Nato znotraj običajnih oklepajev naštejemo signale na vmesniku modula – to so njegovi vhodi in izhodi. Za zaklepajem sledi podpičje. Modul zaključimo z ukazom **`endmodule`**.

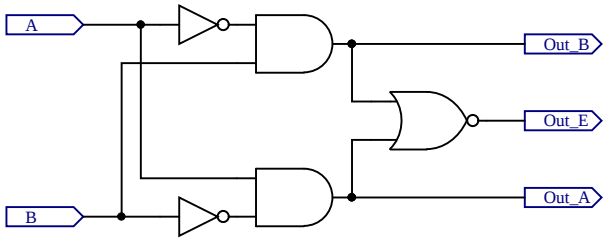
Vhode opišemo z ukazom **`input`**, izhode pa z ukazom **`output`**. Vsi signali na vmesniku so pri navajanju ločeni z vejicami. Bodimo pozorni, da za zadnjim signalom ni dodatne vejice.

V rešitvi je uporabljan ukaz za stalno prirejanje **`assign`**. Z njim prirejamo vrednosti signalom tako, da jih izrazimo z uporabo drugih signalov in operacij, ki so na voljo v Verilogu. Izrazi za prirejanje izhodnih signalov vsebujejo logične operacije in ustrezajo logičnim funkcijam, ki jih lahko razberemo iz sheme vezja na sliki.

Ukaze `assign` uporabljamo za opisovanje preklopnih vezij oz. delov vezij. Z njimi prirejamo vrednosti signalom, ki so opisani kot privzeti izhodi (`output`) ali notranje signale v modulih, ki so deklarirani kot tip **`wire`**.

Za imena modulov in signalov lahko uporabimo velike črke, male črke, številke, podčrtaj in simbol $, vendar pa mora prvi znak v imenu biti črka ali podčrtaj. V primerih bomo module vedno imenovali primer1, primer2 itd. V praksi pa je seveda smiselno dati modulom in signalom opisna imena, iz katerih lahko prepoznamo njihov namen oz. funkcionalnost.

**Primer 2.** Implementirati želimo preprosto preklopno vezje na sliki 8 kot modul v jeziku Verilog. Rešitev je prikazana v kodi 2.




Slika 8: Primer preprostega preklopnega vezja (2)

```
module primer2(
  input A,
  input B,
  output reg Out_A,
  output reg Out_B,
  output reg Out_E
  );

  always @(A or B)
  begin
    Out_A = A & ~B;
    Out_B = ~A & B;
    Out_E = ~(Out_A | Out_B);
  end

endmodule
```

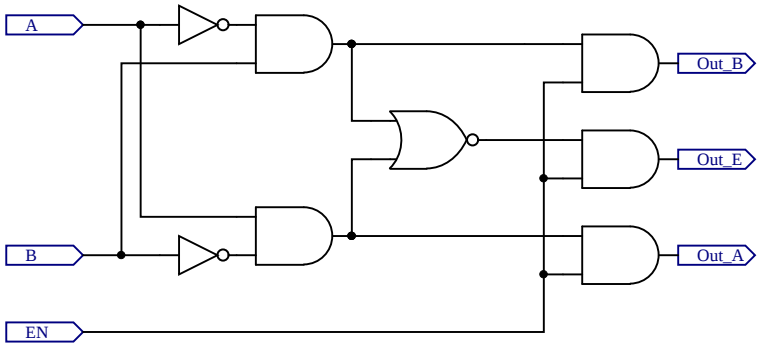
Izvorna koda 2: Rešitev primera na sliki 8

V rešitvi je uporabljen proceduralni blok **`always`**. V njem prirejamo vrednosti izhodom, ki so deklarirani z **`output reg`**, ter notranjim signalom tipa **`reg`** (register). Lahko tudi rečemo v nasprotni smeri, da signali, ki jim prirejamo vrednosti v proceduralnih blokih, morajo biti tipa `reg`.

V proceduralnih blokih za prirejanje vrednosti uporabljamo druge signale in operacije podobno kot pri ukazu `assign`. Lahko pa uporabljamo tudi ukaze za proceduralno opisovanje delovanja vezij (več o tem pri poznejših primerih). Vsebino bloka začnemo z ukazom **`begin`** in končamo z ukazom **`end`**. Če bi znotraj bloka bil samo en ukaz ali eno prirejanje vrednosti, ukaza `begin` in `end` nista nujna.

Za blok `always` moramo podati pogoj, ob katerem se blok izvede in signali dobijo novo vrednost. Kot pogoj smo morali navesti vse signale, ki se znotraj bloka pojavijo na desnih straneh ukazov za prirejanje, in jih ločiti z ukazom **`or`**.

**Primer 3.** Implementirati želimo preprosto preklopno vezje na sliki 9 kot modul v jeziku Verilog. Rešitev je prikazana v kodi 3. Vezje je skoraj enako vezju pri prejšnjem primeru. Razlika je v dodanem vhodnem signalu, ki se imenuje `EN`, in treh dodanih vratih IN pred izhodi.




Slika 9: Primer preprostega preklopnega vezja (3)

```
module primer3(
  input A,
  input B,
  input EN,
  output reg Out_A,
  output reg Out_B,
  output reg Out_E
  );

  always @(A or B or EN)
  begin
    Out_A = EN & A & ~B;
    Out_B = EN & ~A & B;
    Out_E = EN & ~( (A & ~B) | (~A & B) );
  end

endmodule
```

Izvorna koda 3: Rešitev primera na sliki 9

Če si pogledamo delovanje vezja, lahko vidimo, kakšno funkcijo opravlja signal `EN` v vezju. Kadar je signal enak 0, bodo vsi izhodi enaki 0. Kadar pa bo signal 1, bo vezje delovalo enako kot vezje pri prejšnjem primeru.. Na tak način signal `EN` omogoča delovanje vezja.

Dobra praksa je, da signale smiselno poimenujemo. Signal, ki bo omogočal delovanje vezja oz. delov vezja, imenujemo `ENABLE` (angl. za omogoči) ali na kratko `EN`. Ime signala naj kaže njegov pomen, kadar je vrednost signala enaka 1.

Signal, ki bi z vrednostjo 0 omogočil delovanje vezja, z vrednostjo 1 pa 'blokiral' delovanje vezja, bi smiselno imenovali `DISABLE` ali `INHIBIT`. Lahko pa bi še vedno uporabili besedo *enable*, kjer bi na neki način dodali simbol, ki nakazuje negacijo. To je v shemah pogosto črtica nad imenom signala, v kodi pa je lahko mala črka $n$ pred ali za imenom, npr. `nEN`, `ENn` ali `EN_n`.

**Primer 4.** Implementirati želimo multiplekser 4:1. Pri tem želimo, da bodo podatkovni signali (vhodi in izhod multiplekserja) vektorji 4 bitov. Tudi naslovni signal naj bo opisan kot vektor. Rešitev je prikazana v kodi 4.

```
module primer4(
  input [3:0] data0,
  input [3:0] data1,
  input [3:0] data2,
  input [3:0] data3,
  input [1:0] select,
  output reg [3:0] data_output
  );

  always @(*)
  begin
    if (select == 0)
      data_output = data0;
    else if (select == 1)
      data_output = data1;
    else if (select == 2)
      data_output = data2;
    else
      data_output = data3;
  end

endmodule
```

Izvorna koda 4: Multiplekser 4:1 za 4-bitne signale

V rešitvi vidimo, da so vektorji signalov deklarirani z oglatimi oklepaji pred imenom signala. V oklepajih je naveden obseg bitov v vektorju s številkami, kjer začnemo šteti od 0 z desne proti levi. Če obravnavamo vrednost signala kot nepredznačeno celo število, številke bitov tako predstavljajo uteži binarnih števk.

Štiribitni vektorji so opisani z obsegom **`[3:0]`**, dvobitni vektor pa z **`[1:0]`**. Ker v nadaljevanju opisa modula gledamo in prirejamo vektorje po celotnem obsegu bitov, lahko uporabimo samo imena teh vektorjev, pri tem pa ni treba navajati obsega bitov, ki ga želimo upoštevati.

V opisu delovanja smo uporabili blok `always`, vendar smo poenostavili opisovanje bloka tako, da smo kot pogoj napisali *wildcard* simbol **`*`** (zvezdica), katerega pomen je: katerikoli signal. Na ta način zagotovimo, da smo opisali preklopno vezje. Sicer bi lahko pomotoma opisovali vezje, ki vsebuje zadrževalnike.

Delovanje multiplekserja je opisano z **`if...else if...else`** stavkom. Signal `select` se šteje kot dvobitno nepredznačeno število, kar pomeni, da ima lahko vrednosti od 0 do 3. V pogojih preverjamo vrednosti signala `select` z možnostmi 0, 1 in 2. V zadnjem delu uporabimo še možnost `else`, da zajamemo preostale možnosti. V našem primeru je edina preostala možnost vrednost 3.

**Primer 5.** Implementirati želimo preprosti dekodirnik 2 na 4, ki deluje v aktivno visoki logiki. Rešitev je prikazana v kodi 5.

```
module primer5(
  input [1:0] data_in,
  output reg [3:0] data_out
  );

  always @(*)
  begin
    case(data_in)
      2'b00 : data_out = 4'b0001;
      2'b01 : data_out = 4'b0010;
      2'b10 : data_out = 4'b0100;
      2'b11 : data_out = 4'b1000;
    endcase
  end

endmodule
```

Izvorna koda 5: Dekodirnik 2 na 4

Vhod je vektor z dvema bitoma, izhod pa vektor s štirimi biti. Rešitev je implementirana v bloku `always`, kjer je uporabljen stavek **`case...endcase`**. S tem stavkom preverjamo vrednost signala ali izraza, ki je podan za ukazom `case` v okroglih oklepajih. V našem primeru je to signal `data_in`. Primerjamo ga s štirimi možnimi vrednostmi od 0 do 3. Zapis **`2'b`** ´pomeni, da gledamo dvobitno vrednost, izraženo v binarni obliki. Podobno zapis **`4'b`** pomeni 4-bitno število.

Za vsako možno vrednostjo, s katero primerjamo pogojni signal, sledita dvopičje in ukaz, za katerega želimo, da se izvede, kadar bo signal enak tej vrednosti. Ko želimo izvesti več ukazov, moramo uporabiti blok, ki ga začnemo z `begin` in končamo z `end`.

Tudi v ukazih, kjer prirejamo vrednost izhodnemu signalu, smo uporabili binarni zapis za 4-bitni izhodni signal.

Znotraj stavka `case` vidimo funkcionalnost dekodirnika. Za vsako od možnih vrednosti vhoda (od 0 do 3) postavimo ustrezni bit (od 0 do 3) v izhodu na vrednost 1, medtem ko preostale bite postavimo na 0. Na primer: če se na vhodu pojavi vrednost 2, se na izhodu pojavi enica na bitu številka 2.

Prav tako je takšen opis vezja sistematičen in spominja na obliko pravilnostne tabele. Na podoben način lahko opišemo tudi razne druge podatkovne kodirnike, dekodirnike in prekodirnike.

**Primer 6.** Implementirati želimo prioritetni kodirnik 8 na 3 z dodatnim izhodnim signalom. Če nobeden od vhodnih signalov ni aktiven, naj ima kodiran izhod vrednost 0, dodatni izhod pa naj bo tudi enak 0. Ko je vsaj en vhod aktiven, naj bo dodatni izhod enak 1. Prioriteta naj bo na levi (bolj uteženi) strani. Rešitev je prikazana v kodi 6.

```
module primer6(
  input [7:0] data_in,
  output reg [2:0] data_out,
  output reg data_valid
  );

  always @(*)
  begin
    casex(data_in)
      8'b00000001 : data_out = 3'b000;
      8'b0000001x : data_out = 3'b001;
      8'b000001xx : data_out = 3'b010;
      8'b00001xxx : data_out = 3'b011;
      8'b0001xxxx : data_out = 3'b100;
      8'b001xxxxx : data_out = 3'b101;
      8'b01xxxxxx : data_out = 3'b110;
      8'b1xxxxxxx : data_out = 3'b111;
      default     : data_out = 3'b000;
    endcase
  end

  always @(*)
    data_valid = (data_in != 3'b000);

endmodule
```

Izvorna koda 6: Prioritetni kodirnik 8 na 3

V rešitvi je prikazana uporaba stavka **`casex`**. Medtem ko smo pri prejšnjem primeru uporabili stavek `case`, ki zahteva točno navedbo vseh možnih vrednosti za preverjanje, s stavkom `casex` lahko za posamezne bite uporabimo vrednost **`x`** (*don't care*). To pomeni, da se bo primer upošteval, če se v izrazu stavka `casex` pojavi na tem bitu vrednost 0 ali 1. Podobno deluje tudi stavek **`casez`**.

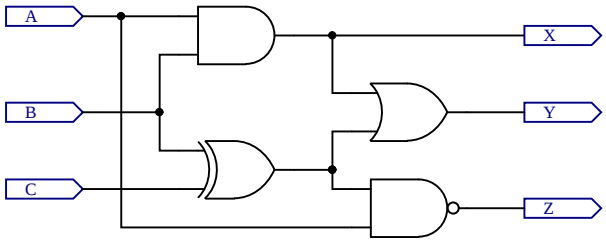
V zgornji rešitvi je drugi naveden primer vrednost `8'b0000001x`. Ker je zadnji bit lahko 0 ali 1, bo ta primer v stavku zajemal dve možnosti: `8'b00000010` in `8'b00000011`. Podobno bodo nadaljnji primeri, kjer se pojavi več vrednosti `x`, zajemali še več možnih vrednosti. Če smo navedli več primerov vrednosti, ki ustrezajo vrednosti signala, ki ga preverjamo, se bo upošteval samo prvi.

Nato smo izkoristili še možnost **`default`**, ki zajema vse možnosti, ki pri prejšnjih vrednostih niso bile zajete. V našem primeru je to vrednost, kjer imamo v vhodnem vektorju same ničle. Če ne bi uporabili te možnosti in bi se na vhodu pojavil vektor iz samih ničel, bi vezje moralo ohranjevati prejšnjo vrednost na izhodnem signalu. Tako bi nastalo vezje s spominom (sekvenčno vezje). Če želimo implementirati preklopna vezja, je priporočljivo, da vedno uporabimo možnost `default` v stavkih `case` in `casex`, podobno kot možnost `else` v stavkih `if`.

Delovanje dodatnega izhodnega signala smo lahko implementirali v eni vrstici. S primerjalnim operatorjem gledamo, ali je vhodna vrednost različna od 0. V tem primeru operacija primerjanja vrne vrednost 1, sicer pa vrednost 0. To vrednost priredimo dodatnemu izhodu `data_valid`.

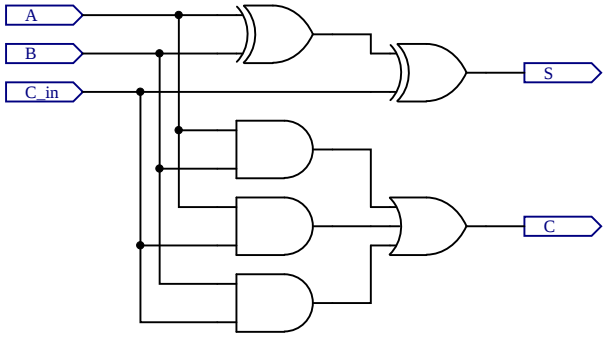
## 2.3 Vaje

**Vaja 1.** Implementirajte vezje na sliki 10 kot modul v jeziku Verilog. Pri implementaciji uporabite ukaze `assign`. Preverite pravilno delovanje svoje implementacije na razvojni plošči.




Slika 10: Primer preprostega preklopnega vezja (4)

**Vaja 2.** Implementirajte vezje na sliki 11 kot modul v jeziku Verilog. Pri implementaciji uporabite proceduralni blok `always`. Preverite pravilno delovanje svoje implementacije na razvojni plošči. Opišite funkcijo vezja. Poskusite poenostaviti (skrajšati) zapis za prirejanje vrednosti.




Slika 11: Primer preprostega preklopnega vezja (5)

**Vaja 3.** Implementirajte multiplekser 4:1 in mu dodajte vhod za omogočanje delovanja. Kadar vhod za omogočanje ni aktiven, naj bo izhod multiplekserja enak 0. Pri implementaciji uporabite ukaz `case`. Preverite pravilno delovanje svoje implementacije na razvojni plošči.

**Vaja 4.** Implementirajte dekodirnik 3 na 8 in mu dodajte aktivno nizki vhod za omogočanje delovanja. Pri implementaciji uporabite ukaz `assign` in poljubne operacije, ki so na voljo v jeziku Verilog. Preverite pravilno delovanje svoje implementacije na razvojni plošči.

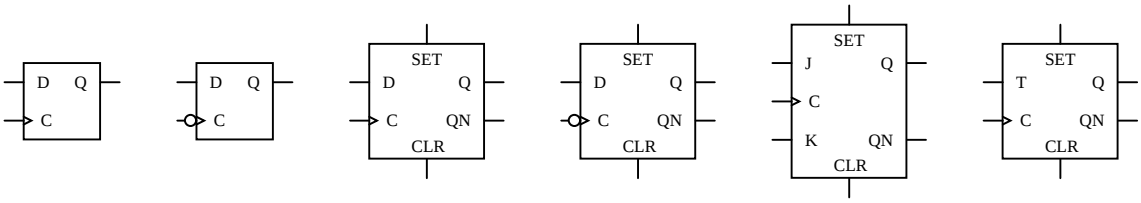
**Vaja 5.** Implementirajte podatkovni prekodirnik, ki bo 8-bitno vhodno vrednost, izraženo v (običajni) binarni obliki, pretvoril v Grayevo kodo. Preverite pravilno delovanje svoje implementacije na razvojni plošči. Poskušajte implementacijo posplošiti na signale s poljubno širino.

# 3 Sekvenčna vezja

Glavna značilnost sekvenčnih vezij je, da vsebujejo pomnilne elemente – spomin. Tako imamo vezje, katerega izhodi so odvisni tako od trenutnih vrednosti na vhodu kot tudi od zaporedja preteklih vrednosti.

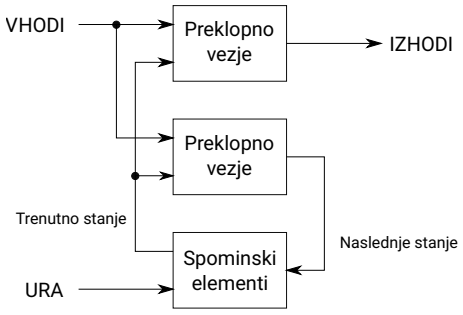
Osnovni pomnilni elementi, ki jih bomo obravnavali, so flip flopi. Poznamo več vrst flip flopov. Za namen dela z vezji FPGA pa bomo tukaj omenili le flip flop tipa D. Ta ima podatkovni vhod `D`, vhod za uro `clk` ter podatkovni izhod `Q` in njegovo negacijo $\bar{Q}$. Podatkovni izhod `Q` nam vedno pove stanje flip flopa – vrednost, ki je v njem shranjena. Imamo tudi vhod za reset, ki stanje `Q` postavi na 0. Ta vhod je pogosto označen s `clr` (clear).

Flip flop tipa D deluje tako, da spremeni stanje le ob ustrezni spremembi ure. To je lahko sprememba z 0 na 1, ki ji pravimo pozitivna ali naraščajoča fronta, ali pa sprememba z 1 na 0, ki ji pravimo negativna ali padajoča fronta. Ob ustrezni fronti se v flip flop shrani tista vrednost, ki je v tem trenutku bila na vhodu `D`. Ta vrednost se takrat pojavi na izhodu `Q`.




Slika 12: Razni tipi flip flopov

Na sliki 12 vidimo primere simbolov za različne tipe flip flopov. Prvi simbol je preprosti simbol za flip flop tipa D, ki je prožen na naraščajočo fronto, drugi simbol pa tak, ki je prožen na padajočo fronto. Naslednja simbola prikazujeta primera, kjer so dodani tudi signal za postavljanje flip flopa na vrednost 0 neodvisno od signala D (`clr`), signal za postavitev na vrednost 1 neodvisno od signala D (`set`) in signal za izhod z negirano vrednostjo flip flopa (`QN`). Nato sledi še enak flip flop, ki pa je prožen na padajočo fronto ure. Zadnja simbola prikazujeta dva druga tipa flip flopov: tip `JK` in tip `T`.




Slika 13: Splošna predstavitev sinhronega sekvenčnega vezja

Sekvenčna vezja lahko delimo na asinhrona in sinhrona vezja. Asinhronih vezij v teh vajah ne bomo obravnavali. Sinhrona sekvenčna vezja so vezja, kjer so spominski elementi proženi na isti signal. Ta signal običajno imenujemo ura ali takt. Na sliki 13 je prikazana splošna predstavitev sinhronega sekvenčnega vezja. Sestavljeno je iz treh delov. Spominski elementi so flip flopi. Skupni nabor vrednosti flip flopov pa predstavljajo stanje vezja. Vhodi v spominske elemente so naslednje stanje vezja, v katero bo vezje prešlo ob naslednji ustrezni fronti ure.

Preostali del je preklopno vezje, ki ga lahko razdelimo na dva dela. Na podlagi vhodov in trenutnega stanja prvi del preklopnega vezja določi naslednje stanje, drugi del pa določi izhode vezja. Takšno splošno vezje imenujemo vezje tipa Mealy. Če so izhodi odvisni le od trenutnega stanja (in ne od vhodov vezja), pa govorimo o vezju tipa Moore.

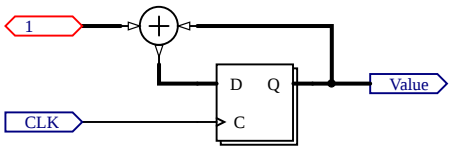
Pri opisovanju delovanja sekvenčnih vezij z jeziki HDL opišemo, ob kateri fronti naj se določi nova vrednost za spominske elemente. Ti so lahko posamezni flip flopi, večbitni registri in podobno. V Verilogu za opisovanje delovanja sekvenčnih vezij uporabljamo proceduralne bloke `always`, kjer opišemo, ali so proženi na naraščajočo fronto (`posedge`) ali padajočo fronto (`negedge`) nekega signala (ure).

Kadar v bloku `always` ne določimo nove vrednosti (npr. vse prireditve so znotraj stavkov `if`, kjer pa niso izpolnjeni pogoji stavkov `if`), ostanejo vrednosti nespremenjene.

V sekvenčnih vezjih lahko flip flope poljubno kombiniramo in povezujemo. Obstajajo pa nekatera standardna vezja, ki uporabljajo več flip flopov. Primeri takšnih vezij so števci in pomikalni registri.

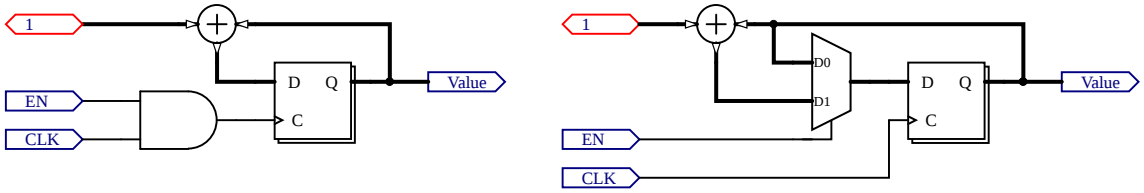
## 3.1 Posebna vezja

Števci so sekvenčna vezja, ki vsebujejo več flip flopov, kjer skupno stanje vezja štejemo kot (običajno celo) število. Stanje v tem primeru lahko imenujemo trenutna vrednost števca. Najpreprostejše delovanje števcev je, da se vrednost, ki je shranjena v števcu, ob vsaki periodi ure poveča za 1.




Slika 14: Preprosti sinhroni števec

Preprosta predstavitev števca je na sliki 14, kjer pa simbol flip flopa predstavlja več flip flopov, večbitni signali pa so ponazorjeni z debelejšimi povezavami. V vezju imamo vhod za uro, ki določa, kdaj se vrednost poveča za ena. Lahko bi imeli tudi dodatni vhod, ki bi določal, za koliko naj se vrednost poveča ali zmanjša. Števec na sliki je sinhroni števec.

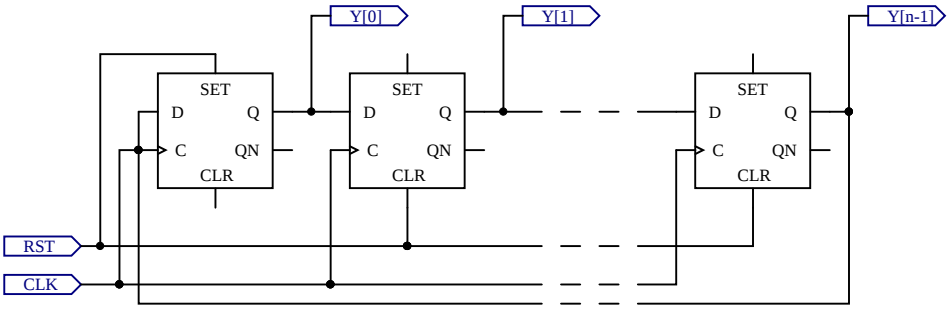



Slika 15: Števec z vhodom za omogočanje

Na sliki 15 sta prikazani dve možnosti števca z vhodom za omogočanje delovanja. V levem vezju uporabimo omogočanje za nadzor signala ure. V tem primeru za proženje flip flopov ne uporabljamo prvotne ure, ampak signal na izhodu vrat (angl. gated clock). Takšna implementacija pa vodi do asinhronega delovanja in pogosto ni priporočena.

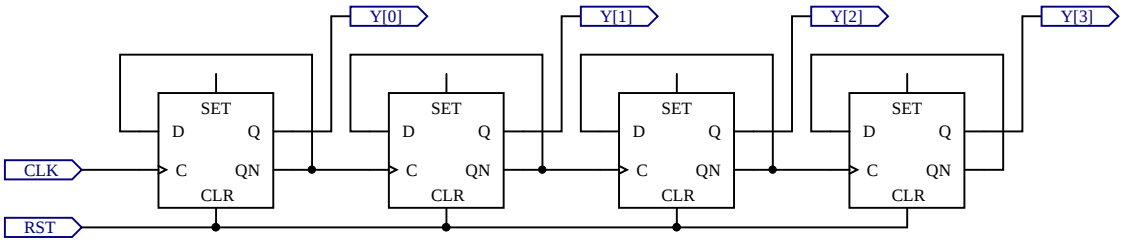
V desnem vezju imamo prikazan sinhroni števec, kjer ura neposredno proži vse flip flope. S signalom za omogočanje in z multiplekserjem izbiramo, ali bo nova vrednost v števcu enaka prejšnji vrednosti ali pa se bo povečala za želeno vrednost.

Števci imajo omejen obseg možnih vrednosti. V binarnih števcih je število zapisano v običajni binarni obliki – vsak flip flop predstavlja eno števko v binarnem zapisu števila. Takšni števci lahko imajo vrednosti od 0 do $2^n - 1$, kjer je $n$ število flip flopov v števcu. Lahko pa zapisane bite v števcu štejemo tudi kot predznačena števila. Če uporabimo dvojiški komplement, dobimo vrednosti od $-2^{n-1}$ do $+2^{n-1} - 1$.




Slika 16: Krožni števec z $n$ biti

Na sliki 16 je prikazana shema krožnega števca. Signal resetiranja deluje tako, da se prvi (na sliki skrajno levi) flip flop postavi na vrednost 1, drugi pa se postavijo na vrednost 0. Po vsaki periodi ure pa se vrednost iz vsakega flip flopa prenese v naslednjega, vrednost iz zadnjega pa se prenese v prvega. Na ta način enica potuje krožno čez vse flip flope v vezju, njen položaj pa nakazuje trenutno vrednost števca. Tak števec lahko deluje le tako, da se njegova vrednost povečuje za 1, obseg možnih vrednosti pa je od 0 do $n - 1$. Ko števec doseže najvišjo vrednost, se nato spet vrne na 0. Signal `Y` na sliki predstavlja vrednost na števcu. Krožni števec deluje sinhrono, vendar zanj porabimo več flip flopov kot pa za binarni števec, če želimo doseči enak obseg možnih vrednosti.




Slika 17: Asinhroni 4-bitni števec

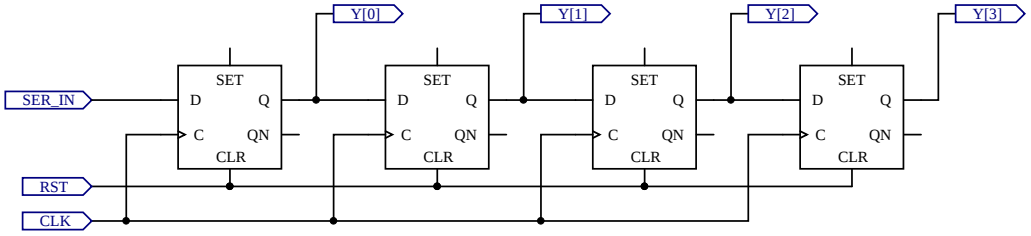
Na sliki 17 je prikazana shema asinhronega števca. Na shemi vidimo, da je negirana vrednost vsakega flip flopa uporabljena kot novi podatek zanj, kar pomeni, da se ob vsakem proženju vrednost negira. Ta signal je uporabljen tudi kot signal za proženje naslednjega flip flopa. Če so ti proženi na naraščajočo fronto signala, se bo na njih vrednost spremenila vedno takrat, ko se bo na prejšnjem flip flopu spremenila vrednost z 1 na 0. Če si pogledamo delovanje binarnih števil, ko jih povečamo za 1, se lahko prepričamo o pravilnosti delovanja takšnega števca. Zaradi zakasnitev, ki nastanejo med posameznimi flip flopi, se biti v izhodu števca ne spreminjajo sočasno. Zato govorimo o asinhronem števcu.

Krožni in asinhroni števec delujeta le tako, da se vrednost na njem poveča za ena. S slik pa je vidno, da asinhroni in krožni števec ne potrebujeta aritmetičnega vezja za določanje nove vrednosti števca. Na splošno pa prevladuje načrtovanje sinhronih sekvenčnih vezij. To velja predvsem v visokofrekvenčnih vezjih, kjer nepredvidene zakasnitve lahko povzročijo napačno delovanje vezja.

Poznamo tudi druge tipe števcev. Števci BCD (angl. binary coded decimal) imajo skupine po 4 bite, kjer binarno štejemo števila od 0 do 9. Nato nadaljujejo z 0 in istočasno povečamo vrednost v naslednji skupini 4 bitov. Tako dobimo v vsaki skupini vrednost enega mesta v decimalnem zapisu števila. Takšni števci so uporabni v vezjih, kjer želimo število pozneje predstaviti v berljivi obliki na kakšnem prikazovalniku.

Spet drugi tip so števci RTC (angl. real time counter), v katerih hranimo čas in datum tako, da imamo skupine bitov, ki predstavljajo sekunde, minute, ure, dneve, mesece in leta. Takšne števce najdemo pogosto v mikrokrmilnikih ali pa obstajajo kot samostojna integrirana vezja.

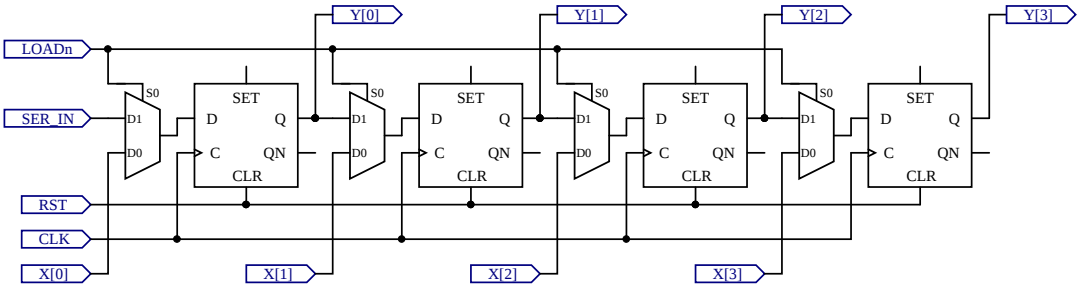
Druga vezja so pomikalni registri (angl. shift register), ki se pogosto uporabljajo za paralelno-serijsko in serijsko-paralelno pretvorbo. Vsebujejo zaporedje medsebojno povezanih flip flopov. Ob normalnem delovanju (pomikanju) se podatki v registru premikajo v levo ali desno smer. Tem registrom lahko dodajamo reset, nalaganje vrednosti, izbiro pomika v levo ali v desno itn.




Slika 18: Preprosti pomikalni register

Na sliki 18 je primer pomikalnega registra, kjer imamo en vhod, na izhodu pa lahko spremljamo vrednosti, ki so bile na vhodu ob zadnjih štirih periodah ure. Tak register lahko deluje kot serijsko-paralelni pretvornik. Imamo serijski vhod, katerega vrednost se ob vsakem ciklu ure shrani v prvi flip flop. Vrednost tega je izhod `Y[0]`. Vrednosti v prvih treh flip flopih se vedno pomikajo v naslednji flip flop, vrednost iz zadnjega pa se ob vsakem ciklu ure izgubi.

Čeprav se vrednosti na prikazani shemi gibljejo v desno, pa prikazanemu vezju pravimo, da je to pomikalni register s pomikom v levo. Shranjene vrednosti v registru se vedno pomikajo proti bolj uteženemu bitu, pri prikazovanju ali zapisovanju vrednosti pa pišemo bolj utežene bite na levi. Za implementacijo pomika v desno moramo izhode registra interpretirati v drugi smeri.




Slika 19: Pomikalni register s paralelnim in serijskim vhodom ter aktivno nizkim vhodom za nalaganje paralelne vrednosti

Na sliki 19 je prikazan pomikalni register s serijskim in paralelnim vhodom. Z nizko vrednostjo na signalu `LOADn` dosežemo, da se v register vpiše vrednost iz vhoda `X`, z visoko vrednostjo pa dosežemo enako delovanje kot pri pomikalnem registru na sliki 18. Takšen pomikalni register lahko uporabimo kot paralelno-serijski pretvornik, če vzamemo izhodni bit `Y[3]` kot serijski izhod.

## 3.2 Primeri

**Primer 7.** Implementirati želimo modul s tremi flip flopi tipa D. Vhodi modula naj bodo podatkovni signal za vse flip flope, signal ure in signal za reset. Izhodi naj bodo vrednosti, shranjene v vseh treh flip flopih. Prvi flip flop naj ne upošteva reseta, drugi naj uporabi sinhroni reset, tretji pa naj uporabi asinhroni reset. Rešitev je prikazana v kodi 7.

```
module primer7(
  input D,
  input clk,
  input rst,
  output reg Q,
  output reg Qsr,
  output reg Qar
  );

  always @(posedge clk)
  begin
    Q = D;
  end

  always @(posedge clk)
  begin
    if (rst)
      Qsr = 0;
    else
      Qsr = D;
  end

  always @(posedge clk or posedge rst)
  begin
    if (rst)
      Qar = 0;
    else
      Qar = D;
  end

endmodule
```

Izvorna koda 7: Flip flopi z različnimi tipi reseta

V rešitvi vidimo, da so spominski elementi implementirani s proceduralnimi bloki `always`, ki so sproženi na določene spremembe signalov. V tem primeru jih prožimo z naraščajočo fronto (**posedge**) signala `clk`. Druga možnost je, da jih prožimo na padajočo fronto (**negedge**).

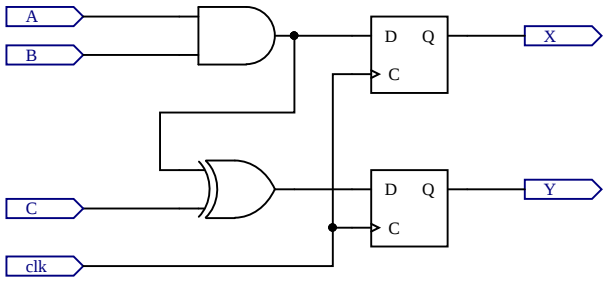
Delovanje samega flip flopa je nato preprosto. V trenutku, ko je sprožen, bo njegova vrednost oz. njegovo stanje postalo enako vrednosti vhodnega signala.

Sinhroni reset pomeni, da se flip flop resetira (njegova vrednost se postavi na 0) le v trenutku ustrezne fronte ure. Reset je torej sočasen (sinhron) z delovanjem ure. V primeru asinhronega reseta, pa bo fronta signala za resetiranje sprožila ustrezno delovanje neodvisno od ure. Zato v tem primeru pišemo kot pogoj bloka `always` tako fronto signala ure kot fronto signala za reset.

Znotraj samega bloka potem v obeh primerih na enak način upoštevamo vrednost signala za resetiranje. Če je reset signal enak 1, se vrednost flip flopa postavi na 0, sicer pa nadaljujemo z opisovanjem običajnega delovanja flip flopa.

Način opisovanja blokov glede na sinhroni ali asinhroni reset lahko posplošimo iz posameznega flip flopa na sekvenčna vezja oz. dele vezij s sinhronim ali asinhronim resetom.

**Primer 8.** Implementirati želimo preprosto sekvenčno vezje na sliki 20 kot modul v jeziku Verilog. Rešitev je prikazana v kodi 8.




Slika 20: Primer preprostega sekvenčnega vezja (1)

```
module primer8(
  input A,
  input B,
  input C,
  input clk,
  output reg X,
  output reg Y
  );

  always @(posedge clk)
  begin
    X = A & B;
    Y = (A & B) ^ C;
  end

endmodule
```

Izvorna koda 8: Rešitev primera na sliki 20

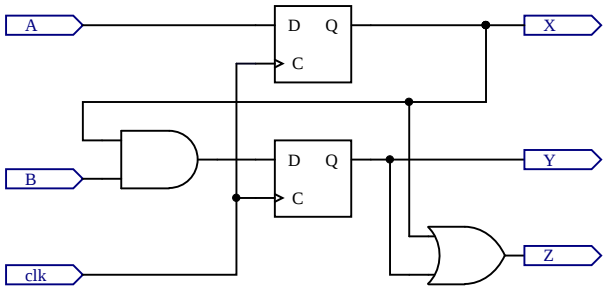
V rešitvi primera vidimo, da smo lahko v enem bloku `always` priredili vrednosti več signalom. Priredili smo vrednosti signaloma, ki ustrezata vrednostma v obeh flip flopih in sta v tem primeru hkrati tudi izhoda celotnega vezja.

Način prirejanja, ki smo ga uporabili, je **proceduralno ali blokirno prirejanje**, kjer uporabimo operator `=`. Koda znotraj proceduralnega bloka se interpretira vrstica za vrstico. Pri tem se določi vrednost izraza na desni strani, nato se vrednost priredi signalu na levi strani prirejanja, šele nato nadaljujemo na naslednjo vrstico in ponavljamo postopek.

Ko nadaljujemo v naslednjo vrstico, že upoštevamo nove vrednosti signala na levi strani prejšnje vrstice. V zgornjem primeru je to signal `X`. Ker pa nikjer pozneje v bloku ne uporabimo vrednosti signala `X`, je delovanje tega vezja neodvisno od načina prirejanja in od vrstnega reda vrstic v proceduralnem bloku.

Čeprav se koda interpretira vrstica po vrstico, se končno vezje implementira tako, da se vse spremembe zgodijo v enem trenutku.

**Primer 9.** Implementirati želimo preprosto preklopno vezje na sliki 21 kot modul v jeziku Verilog. Rešitev je prikazana v kodi 9.




Slika 21: Primer preprostega sekvenčnega vezja (2)

```
module primer9(
  input A,
  input B,
  input clk,
  output reg X,
  output reg Y,
  output reg Z
  );

  reg FF1;
  reg FF2;

  always @(posedge clk)
  begin
    FF1 <= A;
    FF2 <= FF1 & B;
  end

  always @(*)
  begin
    X = FF1;
    Y = FF2;
    Z = FF1 | FF2;
  end

endmodule
```

Izvorna koda 9: Rešitev primera na sliki 21

Način prirejanja, ki smo ga uporabili v bloku `always` pri tem primeru, je **stalno proceduralno ali neblokirno prirejanje**, kjer uporabimo operator **<=**. Pri tem prirejanju se najprej ovrednotijo vsi izrazi na desnih straneh prirejanj, šele nato se te vrednosti priredijo signalom na levi strani. Pri tem se vedno upoštevajo prejšnje (stare) vrednosti signalov.

V tem primeru način prirejanja vpliva na delovanje vezja. V prvi vrstici znotraj prvega bloka `always` smo določili vrednost signalu FF1. Ker pa je ta signal uporabljen v naslednji vrstici, je pomembno, ali se upošteva stara ali nova vrednost signala. Glede na vezje na shemi moramo upoštevati staro vrednost.

Prav tako vidimo, da imamo v vezju dva notranja signala, `FF1` in `FF2`, ki predstavljata vrednosti na obeh flip flopih – stanje vezja. Ker ta signala prirejamo v bloku, morata biti deklarirana kot tip `reg`.

Drugi blok `always` predstavlja preklopno vezje, ki določa vrednosti izhodov vezja.

**Primer 10.** Implementirati želimo preprosti binarni števec, ki povečuje vrednost za 1 ob padajoči fronti ure ter ima signala za sinhrono resetiranje in omogočanje. Število bitov števca naj bo določeno s parametrom in naj ima privzeto vrednost 8. Rešitev je prikazana v kodi 10.

```
module primer10(
  clk,
  rst,
  en,
  value
  );

  parameter WIDTH=8;

  input clk;
  input rst;
  input en;
  output reg [WIDTH-1:0] value;

  always @(negedge clk)
  begin
    if (rst)
      value <= 0;
    else if (en)
      value <= value + 1'b1;
  end

endmodule
```

Izvorna koda 10: Binarni števec s parametrom

V rešitvi vidimo **drugi način opisovanja vhodov in izhodov**. V vmesniku modula so le našteta imena vhodov in izhodov, vendar ni navedeno, kateri signali so vhodi, kateri so izhodi ter kateri so vektorji. Znotraj modula nato sledi opis vrednosti `WIDTH` z ukazom **`parameter`**. Ta vrednost za nas predstavlja število bitov v števcu (njegovo širino).

Če želimo širino števca pozneje spremeniti, lahko to storimo tako, da spremimo vrednost parametra. V tem primeru sicer s tem nimamo manj dela, kot pa če bi širino signala neposredno vpisali v deklaracijo izhoda. Če bi parameter uporabljali pri več signalih (vhodnih, izhodnih ali notranjih) oz. na več mestih znotraj modula, pa si lahko z uporabo parametra olajšamo spreminjanje delovanja vezja. Imamo tudi možnost spremembe parametra pri sami vključitvi modula z ukazom **`defparam`**.

Šele po opisu parametra opišemo vhode in izhode ter vektorje, pri tem pa lahko uporabimo prej opisan parameter. V primerih, kjer imamo parameter, ki opisuje število bitov, pri navedbi obsega uporabimo širino zmanjšano za ena, saj bite začnemo šteti z 0.

Delovanje samega števca je nato opisano zelo preprosto. Števec je prožen na padajočo fronto signala `clk`. V primeru aktivnega signala `rst` se števec postavi na vrednost 0. Če signal `rst` ni aktiven, preverimo, ali je aktiven signal `en`. Če je aktiven, vrednost števca povečamo za 1. Oba krmilna signala (`rst` in `en`) sta aktivno visoka.

Vidimo, da stavka `if` nismo zaključili s končno možnostjo `else`, kar v sekvenčnih vezjih ni nujno. Če v našem primeru nobeden od krmilnih signalov ne bo aktiven (noben od pogojev v stavku `if` ne bo izpolnjen), se bo blok `always` izvedel brez prirejanja signalu `value`. To pa pomeni, da bo ta števec ohranil prejšnjo vrednost, kar je ravno delovanje vezja, ki ga želimo.

**Primer 11.** Implementirati želimo preprosti 8-bitni pomikalni register, ki bo deloval kot paralelno-serijski pretvornik. Rešitev je prikazana v kodi 11.

```
module primer11(
  input clk,
  input rst,
  input load,
  input [7:0] parallel_in,
  output serial_out
  );

  reg [7:0] value;

  always @(posedge clk)
  begin
    if (rst)
      value <= 0;
    else if (load)
      value <= parallel_in;
    else
      value <= value >> 1;
  end

  assign serial_out = value[0];

endmodule
```

Izvorna koda 11: Pomikalni register kot serijsko-paralelni pretvornik

Vezje ima običajna vhoda za sekvenčna vezja: uro in reset. Imamo pa tudi krmilni signal `load`, vhodni podatkovni signal `parallel_in` in izhodni podatkovni signal `serial_out`.

Naloga vezja je, da sprejema podatke v vzporedni obliki (več bitov istočasno) na njegovem vhodu ter jih nato prikazuje na njegovem izhodu v zaporedni obliki (en bit za drugim). Poznamo dva vrstna reda prikazovanja bitov na izhodu. Sistem MSB (angl. most significant bit) First deluje tako, da se najprej prikaže najbolj utežen (skrajno levi) bit, nato pa nadaljujemo proti najmanj uteženemu bitu. Sistem LSB (angl. least signicifact bit) First pa prikazuje bite v obratnem vrstnem redu. Slednji sistem je uporabljen v zgornji rešitvi.

Vezje vsebuje notranji 8-bitni register, imenovan `value`. Če je aktiven signal `load`, se vhodni podatek naloži v ta register, sicer pa se vrednost v registru ob vsakem ciklu ure logično premika v desno smer za 1 bit.

Logični pomik v desno za en bit pomeni, da se vrednost skrajno desnega bita izgubi, vrednost iz vsakega naslednjega bita pa se preslika za eno mesto v desno (proti najmanj uteženemu bitu). Na skrajno levem bitu se vedno pojavi vrednost 0, razen če eksplicitno opišemo drugačno delovanje. Pomik v levo ustrezno deluje v drugi smeri.

Skrajno desni bit iz registra uporabimo kot izhodni signal vezja. Tako bo skrajno desni bit vhodnega podatka najprej prikazan na izhodu. Po vsakem logičnem premiku pa bo prikazan bit, ki je bil v vhodnem podatku eno mesto v levo. Tukaj smo uporabili **izbiro bita v vektorju**. Skrajno desni bit (bit številka 0) smo izbrali z zapisom **oglatih oklepajev za imenom signala**. Podobno lahko izberemo katerikoli drugi bit ali pa obseg bitov.

**Primer 12.** Implementirati želimo 10-bitni krožni števec. Števec naj ima tudi vhod za omogočanje delovanja in dodatni izhod, ki naj bo imenovan CarryOut in naj bo enak 0, kadar je vrednost na števcu 5 ali več. Števec naj deluje na naraščajočo fronto ure. Rešitev je prikazana v kodi 12.

```
module primer12(
  input clk,
  input rst,
  input en,
  output reg [9:0] value,
  output CarryOut
  );

  always @(posedge clk)
    if (rst)
      value <= 10'b000000001;
    else if (en)
      value <= {value[8:0],value[9]};

  assign CarryOut = (value[4:0] != 0);

endmodule
```

Izvorna koda 12: Krožni števec

Krožni števec deluje tako, da je vedno en bit v izhodu enak ena, drugi pa so enaki nič. Položaj enice nakazuje vrednost števca. Možne vrednosti v števcu so tako od 0 do 9.

Delovanje je implementirano v obliki krožnega registra. Ob resetu se vanj nastavi vrednost 0 (enica na bitu številka 0). Če je omogočeno delovanje števca, vrednosti na bitih v registru ob vsakem ciklu ure krožijo v levo. Delovanje je podobno pomikalnemu registru, le da se vrednost iz skrajno levega bita ponovno shrani na skrajno desni bit.

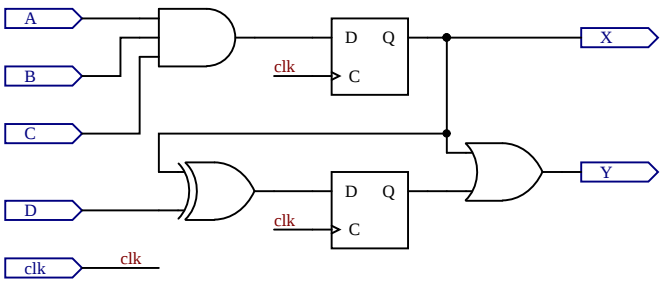
Za opis delovanja je uporabljen operator **sestavljanja signalov**, za katerega uporabimo zavite oklepaje `{...}`. Signal sestavimo iz drugih signalov, ki so znotraj zavitih oklepajev zaporedno našteti in ločeni z vejicami. Signal se sestavi tako, da je signal, ki je prvi zapisan znotraj oklepajev, postavi na najbolj levi del končnega signala. Podobno nadaljujemo z drugimi naštetimi signali.

Delne signale smo pridobili iz prejšnje vrednosti števca, kjer smo izbirali obseg bitov s pomočjo oglatih oklepajev `[...]` za imenom signala. Prvič smo izbrali obseg bitov od 8 do 0, nato pa posamezni bit številka 9.

Za delovanje signala `CarryOut` enostavno preverimo, ali se na bitih od 0 do 4 pojavi kakšna enica.

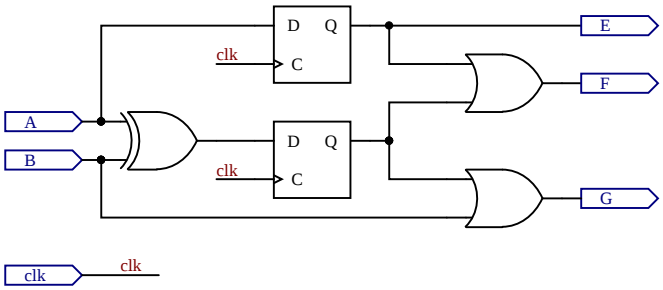
## 3.3 Vaje

**Vaja 6.** Implementirajte vezje na sliki 22 kot modul v jeziku Verilog. Pri implementaciji uporabite neblokirno prirejanje. Preizkusite in preverite pravilno delovanje svoje implementacije na razvojni plošči.




Slika 22: Primer preprostega sekvenčnega vezja (3)

**Vaja 7.** Implementirajte vezje na sliki 23 kot modul v jeziku Verilog. Pri implementaciji sekvenčnega dela uporabite več blokov `always`. Preizkusite in preverite pravilno delovanje svoje implementacije na razvojni plošči.




Slika 23: Primer preprostega sekvenčnega vezja (4)

**Vaja 8.** Implementirajte delilnik frekvence. Vezje naj tvori dva signala, katerih frekvenca je šestkrat manjša od frekvence vhodnega signala. Oba signala tvorite z uporabo bloka `always`, kjer v enem uporabite neblokirno prirejanje, v drugem pa blokirno. Kot vhodni signal uporabite uro z dovolj nizko frekvenco, da boste lahko opazovali delovanje svoje implementacije. Preverite pravilno delovanje svoje implementacije na razvojni plošči.

**Vaja 9.** Implementirajte 8-bitni pulzno-širinski modulator (angl. pulse-width modulator – PWM). Dodajte mu signal za sinhroni reset števca v modulatorju, za vhodno uro pa uporabite obstoječi signal ure. Uporabite parameter za nastavitev števila bitov. Preverite pravilno delovanje svoje implementacije na razvojni plošči tako, da vključite dva modulatorja z različnima vrednostma parametra.

**Vaja 10.** Implementirajte 3-mestni števec BCD z asinhronim resetom, aktivno nizkim signalom za omogočanje štetja in vsemi nujnimi signali za nalaganje vrednosti. Dodajte funkcionalnost, ki preverja, ali imajo vhodni signali za nalaganje vrednosti veljavno vrednost, in v nasprotnem primeru javi napako na dodatnem izhodu. Preverite pravilno delovanje svoje implementacije na razvojni plošči.

# 4 Simulacije logičnega delovanja vezij

Namen izvajanja simulacij digitalnega vezja je preverjanje pravilne implementacije njegove funkcionalnosti oz. iskanje napak v implementaciji. Pri tem v posebnem modulu opišemo, kako naj delujejo vhodi v vezje, pri sami izvedbi simulacije pa opazujemo, kako delujejo izhodi vezja in morebiti tudi notranji signali v vezju.

V jeziku Verilog imamo digitalno vezje oz. del digitalnega vezja opisano v modulu, ki lahko vsebuje nadaljnje podmodule. Ta modul, pri katerem želimo preveriti delovanje, bomo imenovali testiran modul.

Za pripravo simulacije pa moramo napisati še dodatni modul, v katerem vključimo testiran modul. Imenujemo ga simulacijski modul ali testni primer, ki ga prav tako napišemo v jeziku Verilog. Simulacijski modul tipično nima signalov na njegovem vmesniku, temveč v njem opisujemo delovanje njegovih notranjih signalov, ki so vhodi v testiran modul. Pri tem lahko uporabljamo že znane ukaze in konstrukte jezika Verilog. Spoznali bomo dve novi komponenti, ki ju uporabljamo samo v simulacijah, saj zanju ni možna sinteza (implementacija v realizirano vezje). To sta zakasnitev in proceduralni blok `initial`.

V simulacijah uporabljamo zakasnitve tako, da podamo le število kot merilo za zakasnitev. Za pravilno izvedbo simulacije je treba ločeno podati tudi časovno enoto za vse zakasnitve. To običajno naredimo na vrhu datoteke, v kateri pišemo simulacijski modul. Pri tem uporabimo ukaz za prevajalnik `` `timescale `` (ukaz se začne s simbolom ostrivec), za katerim zapišemo dva časa, ločena s poševnico. Prvi čas je enota, ki se uporabi pri zakasnitvah, drugi čas pa je natančnost upoštevanja zakasnitev (mera, kjer zaokrožujemo podane zakasnitve).

Oba časa sta podana s številko, ki je lahko 1, 10 ali 100, ter enoto, ki je lahko katerakoli časovna enota, od sekunde do femtosekunde (`s`, `ms`, `us`, `ns`, `ps`, `fs`). Primer, kjer je enota 1 ns in natančnost 100 ps, je prikazan v kodi 13.

```
`timescale 1ns/100ps
```

Izvorna koda 13: Primer definicije časovne skale

V proceduralnih blokih `initial` opisujemo delovanje signalov od začetka simulacije. Pri tem lahko uporabimo večkratne prireditve vrednosti posameznim signalom, med njimi pa zakasnitve. S tem lahko opišemo delovanje signalov čez celoten potek simulacije. V blokih `initial` lahko uporabimo vse ukaze, ki jih uporabljamo v blokih `always`. V simulacijskem modulu imamo lahko več blokov `initial`, zato ni treba opisovati delovanja vseh vhodnih signalov v istem bloku. Signali, ki jim prirejamo vrednosti v blokih `initial`, morajo biti tipa `reg`, tako kot pri blokih `always`.

Same zakasnitve vključimo s simbolom `#`, za katerim navedemo dolžino zakasnitve. Primer uporabe je prikazan v kodi 14.

```
initial
begin
  rst = 0;
  #100 rst = 1;
  #20  rst = 0;
end
```

Izvorna koda 14: Primer uporabe zakasnitve v bloku `initial`

V prikazanem primeru imamo signal `rst`, ki ob začetku simulacije dobi vrednost 0. 100 časovnih enot pozneje dobi vrednost 1, še nadaljnjih 20 časovnih enot pozneje pa spet vrednost 0. Prikazana koda je tipična oblika uporabe impulza za resetiranje v simulacijah sekvenčnih vezij.

Drugi pogosti signal, ki ga opisujemo v simulacijskih modulih, je ura za sekvenčna vezja. V kodi 15 je prikazan primer opisa ure. Začetna vrednost signala je 0, nato pa se vsakih 5 časovnih enot vrednost signala spremeni. S tem je perioda ure 10 časovnih enot (dve spremembi v eni periodi). Če poznamo tudi časovno skalo, lahko določimo periodo v sekundah oz. frekvenco v Hertzih.

```
initial
begin
  clk = 0;
  while(1)
  begin
    #5 clk = 1;
    #5 clk = 0;
  end
end
```

Izvorna koda 15: Opis ure za simulacijo

V zgornjem primeru bo ura delovala celoten čas simulacije. Če želimo delovanje ure le v omejenem času simulacije, moramo ustrezno uporabiti zakasnitve in končne zanke ali dodatne konstrukte Veriloga, ki so na voljo za simulacije in jih v teh navodilih ne bomo obravnavali.

V kodi 16 pa je prikazana druga možnost opisa ure, v katerem je perioda ponovno 10 časovnih enot. Prav tako vidimo, da smo isti signal opisovali v dveh proceduralnih blokih, kar lahko opravimo v simulacijah.

V tem primeru je tudi uporabljen blok `always` brez pogoja izvedbe. Njegovo delovanje ustreza neskončni zanki, ki se začne ob začetku simulacije. Pri takšni uporabi pa moramo dodati zakasnitev. V nasprotnem primeru simulacija ne bi delovala.

```
initial
  clk = 0;

always
  #5 clk = ~clk;
```

Izvorna koda 16: Alternativna možnost opisa ure za simulacijo

V simulacijskem modulu najprej deklariramo signale. To so lahko signali, ki se bodo povezali na vmesnik testiranega modula, ali pa signali, ki bodo le v simulacijskem modulu. Tisti signali, ki bodo vhodi v testiran modul, bodo večinoma tipa `reg`, saj jih večinoma določamo v blokih `initial` in `always`. Če pa bo kateri signal določen z ukazi `assign`, pa morajo biti tipa `wire`. Signali, ki se bodo povezali na izhode testiranega modula, morajo prav tako biti tipa `wire`.

Nato običajno vključimo instanco testiranega modula. V simulacijah nismo omejeni na testiranje le enega modula hkrati, zato lahko tukaj vključimo več istih ali različnih testiranih modulov. V simulaciji pa moramo dodati tudi ime instance modula.

Nato sledijo proceduralni bloki `intial` in `always` ter ukazi `assign` za opisovanje delovanja vhodnih signalov v testirane module in drugih signalov. Uporabljamo lahko tudi ukaze za izpisovanje rezultatov simulacije v tekstovni vmesnik.

Za ime simulacijskega modula bomo običajno izbrali ime testiranega modula in mu dodali končnico `_sim` (krajše za simulacija). Kadar bomo v njem vključili instanco testiranega modula, bomo morali to instanco tudi poimenovati. Za ime instance pa bomo imenu testiranega modula dodali končnico `_uut` (krajše za *unit under test*). Čeprav lahko simulacijske module in instance modulov poimenujemo poljubno, je priporočljivo imeti dogovor poimenovanja. Prav tako se lahko dogovorimo, da bodo imena signalov v simulacijskem modulu enaka imenom signalov na vmesniku testiranega modula. Izjema je običajno takrat, ko simuliramo delovanje več modulov hkrati.

V kodi 17 je prikazan primer tipične strukture simulacijskega modula.

```
`timescale 1ns/100ps

module vezje_sim();

  reg vhod1, vhod2, vhod3;
  reg clk, rst;
  wire izhod1, izhod2;

  vezje vezje_uut(clk, rst, vhod1, vhod2, vhod3, izhod1, izhod2);

  initial
  begin
    clk = 0;
    rst = 0;
    #100 rst = 1;
    #10  rst = 0;
  end

  always
    #5 clk = ~clk;

  initial
  begin
    vhod1 = 0;
    vhod2 = 0;
    vhod3 = 0;

    #150 vhod1 = 1;

    ...

  end
endmodule
```

Izvorna koda 17: Struktura modula za simulacijo

Pred izvajanjem same simulacije določimo svoj simulacijski modul kot najvišji modul v projektu. Ta korak je nujen, ker moramo določiti, iz katerega modula v projektu naj simulacija izhaja. Določimo tudi privzeto dolžino trajanja simulacije. Ta koraka sta odvisna od programske opreme, ki jo uporabljamo.

Način prikaza signalov je prav tako odvisen od programske opreme, ki jo uporabljamo. Tako lahko na primer signale prikazujemo v različnih barvah, za večbitne signale pa lahko uporabimo prikaz kot predznačeno ali nepredznačeno število v različnih številskih sistemih (decimalno, binarno, heksadecimalno). V nekaterih orodjih lahko prikazujemo potek večbitnih signalov tudi obliki 'analognega' grafa.

## 4.1 Primeri

**Primer 13.** V simulaciji želimo prikazati delovanje modula s preprostim preklopnim vezjem. Preveriti želimo vse možne kombinacije vhodov. Testiran modul je opisan s kodo 18, simulacijski modul pa s kodo 19.

```
module primer13(
  input A, B, C,
  output X
  );

  assign X = A & B | C;

endmodule
```

Izvorna koda 18: Koda testiranega modula za primer 13

```
`timescale 1us/100ns

module primer13_sim();

  reg A, B, C;
  wire X;

  primer13 primer13_uut(A,B,C,X);

  initial
  begin
          A = 0; B = 0; C = 0;
    #100  A = 0; B = 0; C = 1;
    #100  A = 0; B = 1; C = 0;
    #100  A = 0; B = 1; C = 1;
    #100  A = 1; B = 0; C = 0;
    #100  A = 1; B = 0; C = 1;
    #100  A = 1; B = 1; C = 0;
    #100  A = 1; B = 1; C = 1;
  end

endmodule
```

Izvorna koda 19: Koda simulacijskega modula za primer 13

V prvi vrstici je z ukazom za prevajalnik **`timescale`** opisana časovna skala za izvedbo simulacije. Časovna enota za zakasnitve je 1 $\mu$s, natančnost upoštevanja zakasnitev pa je 100 ns.

Simulacijski modul smo poimenovali tako, da smo vzeli ime testiranega modula ter mu dodali končnico `_sim`. V vmesniku simulacijskega modula vidimo, da ni signalov.

V samem modulu imamo opisane 4 signale, ki jih povežemo na testiran modul. Vhodi v testiran modul so v simulacijskem modulu tipa `reg`, saj jim priredimo vrednost znotraj proceduralnih blokov (`initial` ali `always`). Če bi se signalom priredila vrednost z ukazom `assign` ali pa s pomočjo tretjega modula, bi ti signali bili tipa `wire`. Signali, ki jih povežemo na izhode podmodulov, so vedno tipa `wire`.

V bloku **`initial`** smo opisali delovanje vhodnih signalov. Najprej smo zapisali začetne vrednosti, nato pa smo z zakasnitvami 100 $\mu$s spreminjali vrednosti vhodnih signalov in tako preverili vseh 8 možnih kombinacij. Za **zakasnitve** uporabljamo operator **`#`**.

Rezultat simulacije si lahko nato ogledamo v grafičnem prikazu poteka signalov.

**Primer 14.** Simulaciji vezja iz prejšnjega primera želimo dodati izpisovanje besedila in vrednosti signalov v tekstovni vmesnik. Ponovno želimo preveriti vse kombinacije vhodnih vrednosti. Izpis naj bo v obliki, ki spominja na pravilnostno tabelo za izhodni signal. Izpisu hočemo dodati še čas v simulaciji, ob katerem je bil narejen izpis.

Testiran modul in del simulacijskega modula ostaneta nespremenjena. Spremenimo le vsebino bloka `initial` v simulacijskem modulu, kot je prikazano v kodi 20.

```
initial
begin

  {A,B,C} = 3'b000;
  $display("\t\t\t\ttime,\tA,\tB,\tS,\tX");
  #50 $display("%d,\t%b,\t%b,\t%b,\t%d",$time, A,B,C,X);

  while(1)
  begin
    #50  {A,B,C} = {A,B,C} + 1'b1;
    #50  $display("%d,\t%b,\t%b,\t%b,\t%d",$time, A,B,C,X);
  end

end
```

Izvorna koda 20: Koda simulacijskega modula za primer 14

Najprej nastavimo začetne vrednosti signalov. Tukaj vidimo, da lahko **operator sestavljanja** signalov uporabimo **tudi pri prirejanju vrednosti**. Signal, sestavljen iz enobitnih signalov A, B in C, nastavimo na vrednost 0 (`3'b000`), kar pomeni, da vsak posamezni signal dobi vrednost 0. Nato zapišemo imena spremenljivk, ki jih bomo pozneje izpisovali. Za izpisovanje v tekstovni vmesnik uporabljamo t. i. sistemsko funkcijo **`$display`**. Čas od začetka simulacije pa pozneje pridobimo s sistemsko funkcijo **`$time`**.

Nato sledi opis delovanja signalov, kjer smo uporabili neskončno zanko, znotraj katere spreminjamo vrednosti signalov A, B in C v smislu bitov v števcu. Po vsaki spremembi ponovno izvedemo izpis na tekstovni vmesnik. Čas smo vedno izpisali v decimalni obliki, signale pa v binarni. Med samimi izpisanimi signali pa sta vejica in tabulator. Vse zakasnitve so postavljene tako, da so izpisi vrednosti vedno na sredini med spremembami signalov.

Spodaj je prikazano besedilo, ki se izpiše v tekstovnem vmesniku.

```
time,     A,     B,     C,     X
  50,     0,     0,     0,     0
 150,     0,     0,     1,     1
 250,     0,     1,     0,     0
 350,     0,     1,     1,     1
 450,     1,     0,     0,     0
 550,     1,     0,     1,     1
 650,     1,     1,     0,     1
 750,     1,     1,     1,     1
 850,     0,     0,     0,     0
```

**Primer 15.** Simulirati in primerjati želimo delovanje dveh vezij istočasno. Vezji sta opisani z moduloma v kodi 21. Uporabiti želimo signal, ki bo nakazoval, ali vezji različno delujeta. Simulacijski modul, s katerim testiramo oba modula istočasno, je opisan v kodi 22.

```
module primer15_A(
  input A, B, Cin,
  output S, Cout
  );

  assign S = A ^ B ^ Cin;
  assign Cout = (A & B) | (A & Cin) | (B & Cin);

endmodule

module primer15_B(
  input A, B, Cin,
  output S, Cout
  );

  assign {Cout,S} = A + B + Cin;

endmodule
```

Izvorna koda 21: Kodi testiranih modulov za primer 15

```
`timescale 1us/100ns

module primer15_sim();

  reg A, B, Cin;
  wire S_A, S_B, Cout_A, Cout_B;

  primer15_A primer15_A_uut(A,B,Cin,S_A,Cout_A);
  primer15_B primer15_B_uut(A,B,Cin,S_B,Cout_B);

  initial
  begin
    {A,B,Cin} = 3'b000;
    while(1)
      #100 {A,B,Cin} = {A,B,Cin} + 1'b1;
  end

  wire diff;
  assign diff = {S_A,Cout_A} != {S_B,Cout_B};

endmodule
```

Izvorna koda 22: Koda simulacijskega modula za primer 15

Pri deklaraciji signalov vidimo, da lahko uporabimo iste signale kot vhode v oba modula, izhode iz modulov pa moramo povezati na različne signale. Delovanje vhodov smo opisali podobno kot pri prejšnjem primeru.

Dodali smo še signal `diff`, ki dobi vrednost 1, če se izhoda iz modulov razlikujeta. Ta signal ni določen z blokom `initial`, kjer sami določamo vrednosti signalov v simulaciji, niti ni neposredno izhod iz testiranih modulov. Signal opišemo kot rezultat drugih signalov, in sicer podobno, kot smo to storili pri prejšnjih poglavjih.

Po zagonu simulacije lahko ugotovimo, da signal `diff` nikoli nima vrednosti 1, kar pomeni, da modula delujeta enako.

**Primer 16.** V simulaciji želimo prikazati delovanje modula s preprostim sekvenčnim vezjem. Preveriti želimo več možnih prehodov med stanji vezja. Testiran modul je opisan s kodo 23, simulacijski pa s kodo 24.

```
module primer16(
  input A, B, C, clk, rst,
  output X
  );

  reg [1:0] state;
  always @(posedge clk)
    if (rst)
      state <= 2'b00;
    else
      state <= { (A & B), (state[0] | C) };

  assign X = state[0] ^ state[1];

endmodule
```

Izvorna koda 23: Koda testiranega modula za primer 16

```
`timescale 1us/100ns

module primer16_sim();
  reg A, B, C, clk, rst;
  wire X;
  primer16 primer16_uut(A, B, C, clk, rst, X);

  initial
  begin
    clk = 0;
    rst = 0;
    #50 rst = 1;
    #50 rst = 0;
  end

  always
    #5 clk = ~clk;

  initial
  begin
        {A,B,C} = 3'b000;
    #70 {A,B,C} = 3'b110;
    #30 {A,B,C} = 3'b011;
  end

endmodule
```

Izvorna koda 24: Koda simulacijskega modula za primer 16

V simulacijskem modulu tvorimo uro s periodo 10 $\mu$s oz. s frekvenco 100 kHz. Uporabimo tudi impulz za resetiranje vezja. Ta ima ob začetku simulacije vrednost 0, 50 $\mu$s pozneje se njegova vrednost spremeni na 1, dodatnih 50 $\mu$s pozneje se vrne na 0.

Zadnji blok `initial` v simulacijskem modulu določa delovanje vhodnih signalov A, B in C. Navedene so samo tri različne kombinacije vrednosti signalov in zakasnitve pred spremembo vrednosti. Za popolni preizkus vezja bi dodatno uporabili več različnih kombinacij in morda tudi več vmesnih resetiranj vezja.

**Primer 17.** Simulirati želimo delovanje preprostega paralelno-serijskega pretvornika, ki je opisan s kodo 25. Preveriti želimo nalaganja vhodnega signala in zaporedje izhodnih bitov. Simulacijski modul je opisan s kodo 26. Razmislite, kako deluje.

```
module primer17(
  input clk, rst, load,
  input [7:0] data_in,
  output data_out
  );

  reg [7:0] ShiftRegister;
  always @(posedge clk)
    if (rst)
      ShiftRegister <= 8'hFF;
    else if (load)
      ShiftRegister <= data_in;
    else
      ShiftRegister <= (ShiftRegister << 1) | 8'h01;

  assign data_out = ShiftRegister[7];

endmodule
```

Izvorna koda 25: Koda testiranega modula za primer 17

```
`timescale 1us/100ns

module primer17_sim();
  reg clk, rst, load;
  reg [7:0] data_in;
  wire data_out;

  primer17 primer17_uut(clk, rst, load, data_in, data_out);

  initial begin
    clk = 0;
    rst = 0;
    #50 rst = 1;
    #50 rst = 0;
  end

  always
    #5 clk = ~clk;

  initial begin
    data_in = 8'b11001010;
    #100 load = 1;
    #10  load = 0;
  end

  integer i;
  initial begin
    #105 $display("Izhodno zaporedje: ");
    for (i=0;i<8;i=i+1)
      #10 $display("%d", data_out);
  end

endmodule
```

Izvorna koda 26: Koda simulacijskega modula za primer 17

**Primer 18.** Simulirati želimo delovanje preprostega serijsko-paralelnega pretvornika s sinhronim prenosom, ki je opisan s kodo 27. Sinhroni prenos pomeni, da je na voljo ura, ki se prenaša s podatki in skrbi za pravilno časovno usklajenost. Simulacijski modul je opisan s kodo 28. Razmislite, kako deluje.

```
module primer18(
  input clk, rst, data_in,
  output reg [7:0] data_out
  );

  always @(posedge clk)
    if (rst)
      data_out <= 8'hFF;
    else
      data_out <= (data_out << 1) | {7'b0000000,data_in} ;

endmodule
```

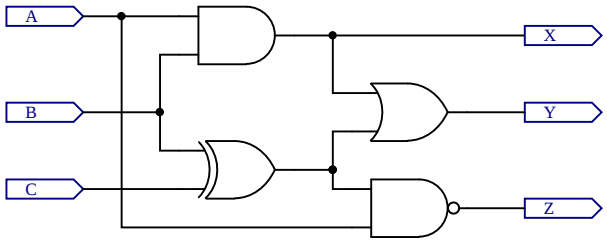
Izvorna koda 27: Koda testiranega modula za primer 18

```
`timescale 1us/100ns

module primer18_sim();
  reg clk_tx, clk_rx, rst_tx, rst_rx, load;
  reg [7:0] data_in;
  wire [7:0] data_out;
  wire data_connect;

  primer17 primer17_uut(clk_tx, rst_tx, load, data_in, data_connect);
  primer18 primer18_uut(clk_rx, rst_rx, data_connect, data_out);

  initial
  begin
    clk_tx = 0; clk_rx = 0;
    rst_tx = 0; rst_rx = 0;
    #50 rst_tx = 1; rst_rx = 1;
    #50 rst_tx = 0; rst_rx = 0;
  end

  always
    #5 clk_rx = ~clk_rx;

  always
    #5 clk_tx = ~clk_tx;

  initial
  begin
    data_in = 8'b11001010;
    #100 load = 1;
    #10  load = 0;
  end

  initial
    #115 $display("Izhodni␣podatek:␣%d", data_out);

endmodule
```

Izvorna koda 28: Koda simulacijskega modula za primer 18

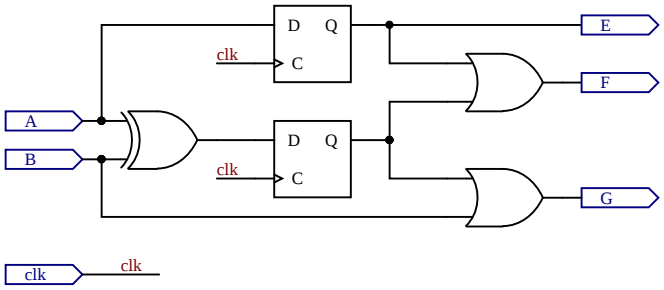
## 4.2 Vaje

**Vaja 11.** Simulirajte delovanje preklopnega vezja na sliki 24. Preverite grafični prikaz rezultatov in izpis v tekstovnem vmesniku.




Slika 24: Primer preprostega preklopnega vezja za simulacijo

**Vaja 12.** Simulirajte delovanje sekvenčnega vezja na sliki 25. Preverite grafični prikaz rezultatov in izpis v tekstovnem vmesniku.




Slika 25: Primer preprostega sekvenčnega vezja za simulacijo

**Vaja 13.** Simulirajte delovanje vezja, opisanega v izvorni kodi 29. Preverite grafični prikaz rezultatov, kjer na graf dodate poteke notranjega signala iz testiranega modula. Uporabite še spremembo parametra in ponovite simulacijo.

```
module vaja_code(
  input clk,
  input IN,
  output OUT
  );

  parameter CODE = 8'b01101001;
  reg [7:0] internal;

  always @(posedge clk)
    internal <= {internal[6:0],IN};

  assign OUT = (internal == CODE);

endmodule
```

Izvorna koda 29: Koda modula za simulacijo – zaznavanje kode

**Vaja 14.** Simulirajte delovanje modulatorja PWM, kjer uporabite različne vhodne podatke. Preverite grafični prikaz rezultatov in dodajte slike notranjih signalov. Uporabite še spremembo parametra.

**Vaja 15.** Simulirajte delovanje vezja, opisanega v izvorni kodi 30. Pri tem preverite funkcionalnost vezja, ki jo lahko razpoznate iz kode. Preverite grafični prikaz rezultatov, kjer na graf dodate poteke notranjih signalov iz testiranega modula.

```
module vaja_edge_detect(
  input clk,
  input rst,
  input IN,
  output OUT1,
  output OUT2,
  output OUT3
  );

  reg OLD;
  reg NEW;

  always @(posedge clk)
  begin
    NEW <= IN;
    OLD <= NEW;
  end

  assign OUT1 = NEW ^ OLD;
  assign OUT2 = ~NEW & OLD;
  assign OUT3 = NEW & ~OLD;

endmodule
```

Izvorna koda 30: Koda za simulacijo – zaznavanje prehodov

**Vaja 16.** Simulirajte delovanje vezja, opisanega v izvorni kodi 31. Pri tem preverite funkcionalnost vezja, ki jo lahko razpoznate iz kode. Preverite grafični prikaz rezultatov, kjer na grafu uporabite 'analogni' prikaz.

```
module vaja_phase_acc(
  input clk,
  input rst,
  input [3:0] freq,
  input [7:0] phase,
  output [7:0] signal
  );

  reg [7:0] acc;
  always @(posedge clk)
    if (rst)
      acc = 0;
    else
      acc = acc + freq;

  assign signal = acc + phase;

endmodule
```

Izvorna koda 31: Koda za simulacijo – fazni akumulator

# 5 Primeri uporabnih vezij

V tem poglavju si bomo ogledali razne implementacije uporabnih logičnih vezij. Prvi vaji – gonilnik za 7-segmentni prikazovalnik – se navezujeta na prikazovalnik na razvojnih ploščah. Ti vaji predstavljata primer uporabe raznih logičnih sklopov, ki smo jih spoznali pri prejšnjih poglavjih, ter njihovega povezovanja. Prikazovalnik nato uporabljamo pri naslednjih vajah.

Sledijo vaje z implementacijo merilnika frekvence, v katerih hočemo izmeriti frekvenco nekega signala in rezultat izpisati na prikazovalniku. Zato pri tej vaji tudi spoznamo medsebojno povezovanje več modulov za doseganje končnega delovanja.

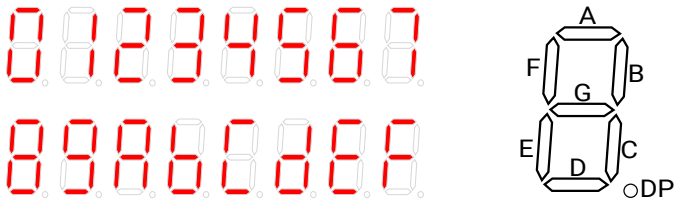
Naslednji vaji se nanašata na uporabo dodatnih zunanjih vezij: digitalno-analognega in analogno-digitalnega pretvornika. Pri tej vaji bo treba komunicirati z zunanjimi digitalnimi vezji s pomočjo serijskih vodil, ob tem pa spoznamo tudi uporabo logičnega analizatorja.

Sledijo še razne vaje z numerično krmiljenim oscilatorjem, generatorjem psevdonaključnih števil in moduli za asinhrono komunikacijo.

## 5.1 Gonilnik za 7-segmentni prikazovalnik

Implementirati želimo gonilnik za večmestni 7-segmentni prikazovalnik s svetlečimi diodami, ki je na razvojni plošči. Za vsako mesto imajo takšni prikazovalniki v sebi 7 segmentov za prikazovanje simbolov in pogosto tudi osmi segment, ki predstavlja decimalno piko za simbolom. Z osnovnimi 7 segmenti lahko prikazujemo števke od 0 do 9 in črke od ‘A’ do ‘F’.

Da lahko posamezni simbol prikažemo, moramo vklopiti ustrezne segmente (svetleče diode), ki so poimenovani s črkami od ‘A’ do ‘F’, ‘DP’ pa označuje decimalni simbol (angl. decimal point). Razporeditev segmentov in oblike možnih simbolov za prikaz lahko vidimo na sliki 26.




Slika 26: Simboli na 7-segmentnem prikazovalniku in posamezni segmenti prikazovalnika

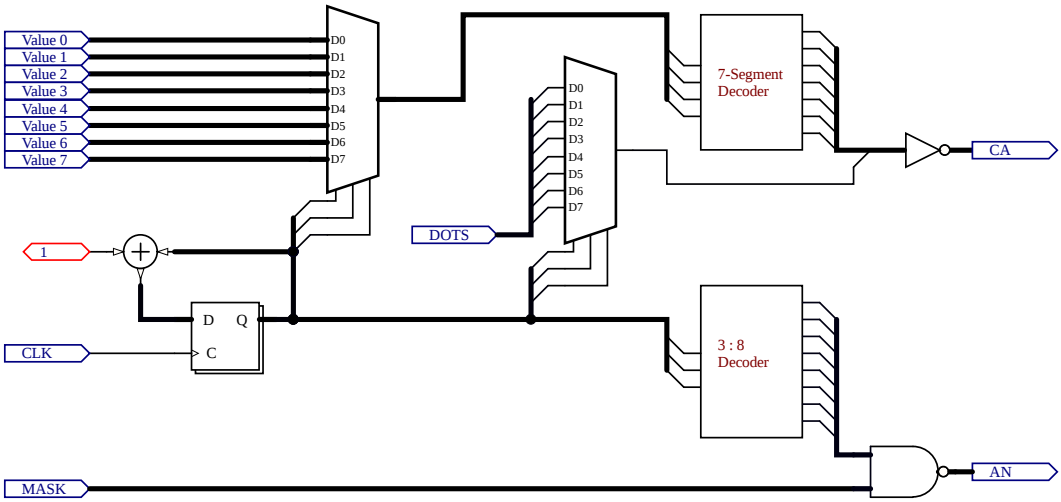
Predpostavimo, da imamo na voljo število v 4-bitni binarni obliki. To število ima lahko vrednosti od 0 do 15. Prav te vrednosti pa ponazorimo s heksadecimalnimi števkami od 0 do F. Preklopno vezje, ki ga potrebujemo za pretvorbo iz 4-bitnega števila v 7-bitni signal za segmente (brez decimalne pike), se imenuje 7-segmentni dekodirnik. Pri tem moramo imeti dogovorjeno, v kateri smeri razporedimo bite, ki ustrezajo segmentom.

Odvisno od izvedbe prikazovalnika – razlikujemo prikazovalnike s skupno katodo in prikazovalnike s skupno anodo – in njegovih povezav na digitalno vezje moramo preveriti, ali segmente vklopimo z logično enico ali logično ničlo.

Pri uporabi večmestnega prikazovalnika izmenjujemo trenutno prikazan simbol. Če to delamo dovolj hitro, ne bomo videli utripanja posameznih simbolov, ampak se nam bo zdelo, da so na vseh mestih prikazovalnika ves čas prikazani vsi simboli. Ta način delovanja pomeni, da bomo za delovanje prikazovalnika potrebovali dva izhodna vektorja signalov. Prvi bo določal, kateri segmenti se vklopijo in bo 8-bitni. Drugi pa bo določal, na katerih mestih prikazovalnika se vklopijo. Število bitov v drugem vektorju mora biti enako številu mest na prikazovalniku.

**Vaja 17.** Implementirajte gonilnik za 7-segmentni prikazovalnik za uporabljeno razvojno ploščo in ga preizkusite na njej. Implementiran naj bo tako, da lahko z vhodi nastavite vsak simbol za vsako mesto na prikazovalniku, da lahko vklopite posamezne segmente za decimalne pike in da lahko posamezna mesta tudi maskirate (jih popolnoma izklopite).

Pri implementaciji si pomagajte s shemo na sliki 27. Za implementacijo 7-segmentnega dekodirnika pa si lahko pomagate s kodo 32, vendar bodite pozorni na to, da se v tej kodi segmenti vklopijo z logično enico.




Slika 27: Shema vezja za gonilnik za 7-segmentni prikazovalnik

```
always @(*)
  case(IN)
    4'h0    : OUT = 7'b0111111;
    4'h1    : OUT = 7'b0000110;
    4'h2    : OUT = 7'b1011011;
    4'h3    : OUT = 7'b1001111;
    4'h4    : OUT = 7'b1100110;
    4'h5    : OUT = 7'b1101101;
    4'h6    : OUT = 7'b1111101;
    4'h7    : OUT = 7'b0000111;
    4'h8    : OUT = 7'b1111111;
    4'h9    : OUT = 7'b1100111;
    4'hA    : OUT = 7'b1110111;
    4'hB    : OUT = 7'b1111100;
    4'hC    : OUT = 7'b0111001;
    4'hD    : OUT = 7'b1011110;
    4'hE    : OUT = 7'b1111001;
    4'hF    : OUT = 7'b1110001;
    default: OUT = 7'b1001001;
  endcase
```

Izvorna koda 32: 7-segmentni dekodirnik

**Vaja 18.** Prejšnjo nalogo nadgradite tako, da boste lahko s parametri določali, ali bo modul uporabljen s prikazovalnikom s skupnimi katodami ali prikazovalnikom s skupnimi anodami. Dodajte še parameter, s katerim lahko določate število mest na prikazovalniku (omejite se na števila mest, ki so potence števila 2).

Vezje preizkusite v simulaciji.

## 5.2 Merilnik frekvence

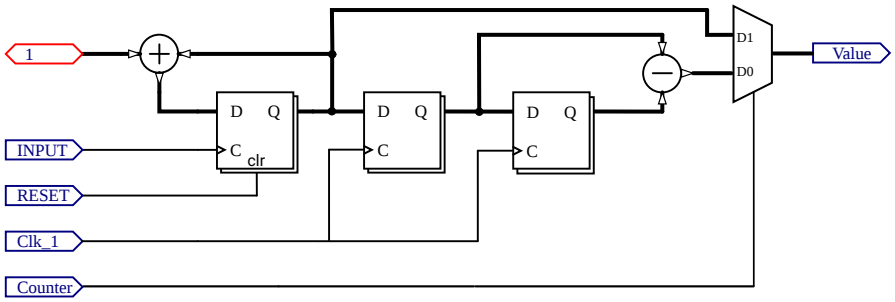
Implementirati želimo merilnik frekvence. Predpostavimo, da imamo signal, ki je že v primerni digitalni obliki, kar pomeni, da ne potrebujemo dodatnega zunanjega vezja za preoblikovanje signala.

Princip merjenja frekvence je dokaj enostaven. V vezju je na voljo referenčna ura z natančno znano frekvenco. Z njeno pomočjo lahko določimo časovni interval znane dolžine, npr. 1 sekundo. V vezju imamo števec, ki šteje periode vhodnega signala neznane frekvence. Na začetku merilnega intervala števec resetiramo na vrednost 0, na koncu intervala pa preberemo vrednost na števcu.

Delovanje lahko poenostavimo tako, da števca ni treba resetirati ob začetku vsakega intervala, moramo pa namesto tega na koncu merilnega intervala gledati spremembo vrednosti na števcu glede na konec prejšnjega intervala.

Če je dolžina merilnega intervala 1 sekunda, potem bo izmerjena vrednost na števcu predstavljala frekvenco v Hertzih. Za merilne intervale drugih dolžin moramo vrednost v števcu ustrezno preračunati v želeno enoto.

**Vaja 19.** Implementirajte merilnik frekvence z območjem delovanja do 10 MHz in resolucijo 1 Hz. Dodajte funkcijo, kjer bo vezje delovalo kot števec, ki ga s tipko ročno resetirate na vrednost 0. Izhod merilnika naj bo podatek o frekvenci v binarni obliki. Pri implementaciji si pomagajte s shemo na sliki 28.




Slika 28: Shema vezja za merilnik frekvence

Izhod povežite na vhod modula za pretvorbo iz binarne oblike števila v obliko BCD. Nato pa rezultat pretvorbe povežite na vhode gonilnika za 7-segmentni prikazovalnik. Pred preizkusom si poglejte prikazano vezje v programski opremi in bodite pozorni na medsebojne povezave modulov.

Pri preizkusu za vhodni signal uporabite zunanji priključek razvojne plošče, na katerega povežete funkcijski generator. Pri uporabi funkcijskega generatorja bodite pozorni, da tvorite pravokotni signal s primernimi napetostnimi nivoji.

**Vaja 20.** Merilnik frekvence iz prejšnje naloge nadgradite tako, da lahko izbirate 3 različne resolucije: 10 Hz, 1 Hz in 0,1 Hz. Pri tem poskrbite, da bo rezultat na prikazovalniku vedno prikazan kot rezultat v kHz in da bo ustrezno prikazana decimalna vejica na zaslonu.

## 5.3 Modula za pretvornika DAC in ADC

Implementirati želimo dva modula, s katerima krmilimo digitalno-analogni (DA) in analogno-digitalni (AD) pretvornik, ki se uporabljata s preprostim serijskim vmesnikom. Za pravilno uporabo obeh pretvornikov moramo poznati pravilne poteke vseh signalov pri komunikaciji med digitalnim vezjem in pretvornikom. Pravilne poteke signalov najdemo v tehnični dokumentaciji obeh pretvornikov.

Za pretvornik DAC8551 potrebujemo 3 digitalne povezave: $D_{IN}$ (vhodni podatek), SCLK (ura serijske povezave) in $\overline{\text{SYNC}}$ (negiran sinhronizacijski signal). Vsi trije signali so vhodi v pretvornik DA, povežemo pa jih na digitalno vezje.

Pri uporabi pretvornika postavimo signal $\overline{\text{SYNC}}$ na vrednost 0. Nato na podatkovni povezavi $D_{IN}$ prenesemo 24-bitni podatek v načinu MSB First. Podatki se prenašajo sinhrono z uro, pri tem pa se vrednost bitov spreminja z naraščajočo fronto ure, bere pa se s padajočo fronto. Po prenašanju 24 bitov podatka lahko postavimo vrednost signala $\overline{\text{SYNC}}$ spet na 1. Počakati moramo vsaj 100 ns od 24. padajoče fronte ure do naslednje negativne fronte signala $\overline{\text{SYNC}}$. 24-bitni podatek je sestavljen tako, da so biti 23 do 18 neuporabljeni, bita 17 in 16 predstavljata vrednost za krmilni register, biti 15 do 0 pa predstavljajo 16-bitno digitalno vrednost, ki naj se pretvori v analogno napetost. V krmilni register vpišemo vrednost 00 (normalno delovanje).

Za pretvornik ADS8320 potrebujemo prav tako 3 digitalne povezave. Signal $D_{OUT}$ je podatkovna povezava, ki je izhod iz pretvornika. Preostala signala sta njegova vhoda. To sta signala DCLOCK (ura serijske povezave) in $\overline{\text{CS}}$/SHDN (negiran signal za izbiro čipa oz. signal za mirovanje). Vse tri signale povežemo na digitalno vezje.

Signal $\overline{\text{CS}}$/SHDN postavimo na vrednost 0 ob padajoči fronti ure. Nato moramo čakati vsaj 20 ns do prve naraščajoče fronte ure. Pri delovanju je nato treba tvoriti 5 ciklov ure, preden začnemo brati vrednosti na podatkovni povezavi. Prva vrednost, ki jo preberemo, je vrednost 0, nato pa sledi 16 bitov digitalnega podatka o napetosti. Biti se spreminjajo na padajočo fronto, beremo pa jih na naraščajočo fronto. Po zaključku branja postavimo signal $\overline{\text{CS}}$/SHDN spet na vrednost 1.

Za pravilno povezovanje vmesnika modula s priključkom na razvojni plošči si pomagamo s shemama obeh razširitvenih modulov in shemo oz. dokumentacijo razvojne plošče.

**Vaja 21.** Implementirajte modul, s katerim krmilite digitalno-analogni pretvornik DAC8551. Vhod v modul naj bo 8-bitni podatek ter ura s frekvenco 1 MHz. Implementirajte modul tako, da bo pretvorba delovala s hitrostjo 20 kSPS. Modul povežite na pine razširitvenega priključka na razvojni plošči.

Preverite pravilno delovanje svoje implementacije tako, da priključite zunanji modul s pretvornikom DA in voltmeter, istočasno pa signale med pretvornikom in razvojno ploščo opazujete z logičnim analizatorjem.

**Vaja 22.** Implementirajte modul, s katerim krmilite analogno-digitalni pretvornik ADS8320. Izhod iz modula naj bo 8-bitni podatek, ki ga povežete na prikazovalnik. Uporabite uro s frekvenco 1 MHz. Implementirajte modul tako, da bo pretvorba delovala s hitrostjo 10 kSPS. Modul povežite na pine razširitvenega priključka na razvojni plošči.

Preverite pravilno delovanje svoje implementacije tako, da priključite zunanji modul s pretvornikom AD in napetostnim virom, istočasno pa signale med pretvornikom in razvojno ploščo opazujete z logičnim analizatorjem.

## 5.4 Numerično krmiljen oscilator

V sistemih za neposredno digitalno sintezo signalov uporabljamo numerično krmiljen oscilator za tvorjenje periodičnega digitalnega signala. Za delovanje oscilatorja potrebujemo referenčno uro ter vhodni podatek za nastavitev frekvence. V samem oscilatorju imamo fazni akumulator, ki ga sestavljata seštevalnik in register. Vrednost v faznem akumulatorju (registru) predstavlja trenutni fazni kot signala, ki ga tvorimo. Če imamo v registru $n$ bitov, bomo imeli $2^n$ možnih faznih kotov, razlika med zaporednimi faznimi koti pa bo $360°/(2^n)$.

Vrednost v faznem akumulatorju se ob vsakem ciklu referenčne ure postavi na novo vrednost, ki je vsota stare vrednosti in nastavljene frekvence. Višja je nastavitev frekvence, hitreje se povečuje vrednost v faznem akumulatorju. Kadar pride do prekoračitve vrednosti, se ta vrne na ustrezno nižjo vrednost. Frekvenco signala nato dobimo z izrazom

$$f = \frac{F}{2^n} \cdot f_{clk},$$

kjer je $F$ numerično nastavljena frekvenca, $n$ število bitov v faznem akumulatorju ter $f_{clk}$ frekvenca referenčnega signala ure.

Če vrednosti v faznem akumulatorju prištejemo še neko konstantno vrednost, dobimo vrednost novega trenutnega faznega kota, ki pa bo predstavljal signal, ki je v primerjavi s prvim fazno zamaknjen za kot

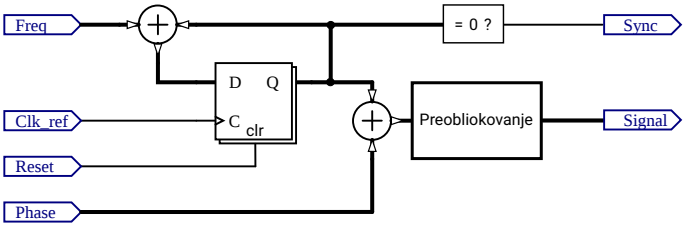
$$\varphi = \frac{P}{2^n} \cdot 360°,$$

kjer je $P$ numerično nastavljen fazni zamik, $n$ pa je ponovno število bitov v registru.

Če gledamo vrednost faznega akumulatorja kot na signal, bo to signal naraščajoče žagaste oblike. Uporabimo ga kot vhod v vezje, ki signal preoblikuje. V tem vezju imamo tabelirane vrednosti za signala želene oblike za vseh $2^n$ faznih kotov signala. Žagasti signal uporabimo kot naslovni vhod, ki pove, katero 'vrstico' tabele beremo, prebrana vrednost pa nato predstavlja končni signal.

Ker so oblike signala tabelirane, lahko na ta način tvorimo signale poljubne oblike, smo pa omejeni s številom faznih zamikov in natančnostjo predstavitve vrednosti signala.

**Vaja 23.** Implementirajte numerično krmiljen oscilator z 8-bitnim faznim akumulatorjem, kjer tvorite sinusni signal z 8-bitno natančnostjo vrednosti. Implementirajte možnost nastavljanja frekvence in faznega zamika. Dodajte še izhod, kjer tvorite sinhronizacijski signal ob začetku vsake periode signala. Dodajte še vhod za resetiranje. Vhodna ura, ki poganja fazni akumulator, naj ima frekvenco 20 kHz. Pri implementaciji si pomagajte s shemo na sliki 29.




Slika 29: Shema vezja za numerično krmiljen oscilator

Delovanje najprej preizkusite v simulaciji, nato pa povežite signal še na vhod digitalno-analognega pretvornika in preverite delovanje z osciloskopom.

## 5.5 Generator psevdonaključnih števil

Poznamo več vrst generatorjev psevdonaključnih števil. Med preprostejšimi so linearni kongruenčni generatorji, ki tvorijo zaporedje števil po enačbi

$$X_{i+1} = (a \cdot X_i + b)(\text{mod } m),$$

kjer je $X_i$ trenutna vrednost (stanje generatorja), $X_{i+1}$ je naslednja vrednost, $a$ je množitelj, $b$ je inkrement in $m$ modul. Začetno stanje generatorja imenujemo seme. Med delovanjem bo generator tvoril zaporedje števil, ki je vedno enolično določeno z vrednostjo semena, vendar pa daje vtis naključnih števil.

**Vaja 24.** Implementirajte linearni kongruenčni generator psevdonaključnih števil. Tako naj ima vhod za uro, vhod za vrednost semena in vhod za reset, ki stanje generatorja postavi na vrednost semena. Generator naj bo 32-biten ($m = 2^{32}$) in v njem naj se uporabljajo parametri

$$a = 1664525 \quad \text{in} \quad b = 1013904223.$$

Delovanje preizkusite najprej v simulaciji, nato pa zgornjih 8 bitov števila povežite na vhod pretvornika DA in opazujte končni signal na osciloskopu. Dodatno povežite te bite še na 7-segmentni prikazovalnik.

## 5.6 Asinhrona komunikacija

Samostojno preučite delovanje asinhrone serijske komunikacije UART – *Universal asynchronous receiver-transmitter*. Oglejte si predvsem obliko signalov pri tej komunikaciji.

**Vaja 25.** Implementirajte oddajnik UART za hitrost 9600 bitov na sekundo, brez paritete, z 8 podatkovnimi biti in enim stop bitom. Delovanje preverite najprej v simulaciji, nato pa opravite implementacijo in preverite delovanje z logičnim analizatorjem.

**Vaja 26.** Oddajnik iz prejšnje vaje posplošite tako, da mu lahko nastavljate bitno hitrost, pariteto in število bitov. Delovanje preverite na enak način kot prej.

**Vaja 27.** Implementirajte sprejemnik UART za fiksno hitrost 9600 bitov na sekundo, brez paritete, z 8 podatkovnimi biti in enim stop bitom. Vezje naj ima podatkovni izhod, kjer se prikaže vrednost, in dodatni enobitni izhod, ki nakaže, v katerem trenutku se je pojavila nova vrednost.

Delovanje preverite tako, da v simulaciji med seboj povežete oddajnik in sprejemnik ter preverjate, ali se podatki pravilno prenašajo. Nato delovanje preverite tako, da uporabite dve razvojni plošči, kjer je na eni oddajnik, na drugi pa sprejemnik.

**Vaja 28.** Sprejemnik iz prejšnje vaje posplošite tako, da mu lahko nastavljate bitno hitrost, pariteto in število bitov. Delovanje preverite na enak način kot prej.

# 6 Literatura

[1] Pong P. Chu, *FPGA Prototyping by Verilog Examples: Xilinx Spartan-3 Version.* Wiley-Interscience, 2008.

[2] Steve Kilts, *Advanced FPGA Design: Architecture, Implementation, and Optimization.* Wiley-IEEE Press, 2007.

[3] John F. Wakerly, *Digital Design: Principles and Practices.* 5th edition, Pearson, 2008.

[4] John P. Hayes, *Introduction to Digital Logic Design.* Addison-Wesley, 1993.

[5] Stuart Sutherland, *Verilog HDL Quick Reference Guide: based on the Verilog-2001 standard (IEEE Std 1364-2001).* Sutherland HDL, Incorporated, 2001.

# Digitalne strukture in sistemi: Laboratorijske vaje

Gregor Donaj

Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor, Slovenija. gregor.donaj@um.si

**Povzetek** Gradivo vsebuje navodila za laboratorijske vaje pri predmetu Digitalne strukture in sistemi za študente prve stopnje univerzitetnega študijskega programa Elektrotehnika, smer Elektronika. Laboratorijske vaje obravnavajo začetno spoznavanje uporabe jezikov za opisovanje strojne opreme in uporabo programirljivih logičnih vezij. Vaje so tako primerne tudi za druge predmete s podobno vsebino. Vsebina vaj obsega implementacijo preprostih preklopnih vezij, implementacijo preprostih sekvenčnih vezij, simulacijo digitalnih vezij in primere nekoliko kompleksnejših uporabnih vezij. Pri teh primerih so obravnavane tudi druge teme, kot so povezovanja digitalnih sklopov oz. modulov, uporaba logičnega analizatorja in podrobnejša analiza delovanja načrtovanih vezij.

**Ključne besede:** elektronika, programirljiva logična vezja, Verilog, simulacije digitalnih vezij

Univerzitetna založba Univerze v Mariboru

DOI https://doi.org/10.18690/um.feri.3.2023 ISBN 978-961-286-719-5

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko