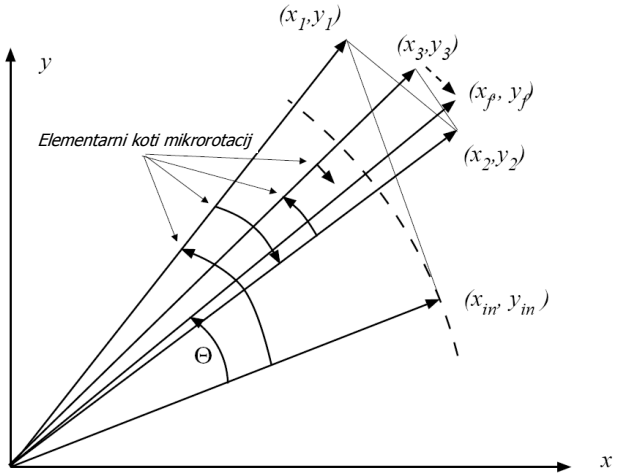
CORDIC algoritem za izračun sin(θ) in cos(θ)

V VHDL programirajte arhitekturo splošne strukture CORDIC (ang. COordinate Rotation DIgital Computer) vezje za izračun kotnih funkcij sin in cos. Delovanje CORDIC algoritma kratko povzema spodnja psevdokoda, ki se nahaja v JavaScript dokumentu (cordic_rotation_circular.htm) v predlogi vaje. V tem dokumentu se izpišeta izračuna sin in cos po CORDIC algoritmu in z uporabo matematičnih funkcij knjižnice v jeziku JavaScript. Program za dani kot θ izračuna razliko med CORDIC algoritmom in izračunom z uporabo matematičnih funkcij ter izpiše vse iteracije CORDIC algoritma.

```
for (k = 0; k < width; k++) {
      var x_temp = x;
      if ( z >= 0) {
            x -= (y >> k);
            y += (x_temp >> k);
            z -= elementary_angles[k];
      }
      else {
            x += (y >> k);
            y -= (x_temp >> k);
            z += elementary_angles[k];
      }
}
```



Slika 1: CORDIC mikrorotacije vektorja.

CORDIC algoritem v načinu *vrtenja* (ang. rotation mode) temelji na splošnem vrtenju krajevnega vektorja v ravnini (x, y) za kot θ, kot prikazuje Slika 1. Krajevni vektor točke (x_{in}, y_{in}) zavrtimo za kot θ tako, da tvorimo zaporedje mikrorotacij (x_1, y_1), (x_2, y_2), (x_3, y_3) … do končnega krajevnega vektorja točke (x_f, y_f). Krajevni vektor zavrtimo vsakič za vnaprej definirani elementarni kot (α_k), pri čemer želimo zmanjšati razliko med trenutnim kotom vrtenja in θ proti 0.

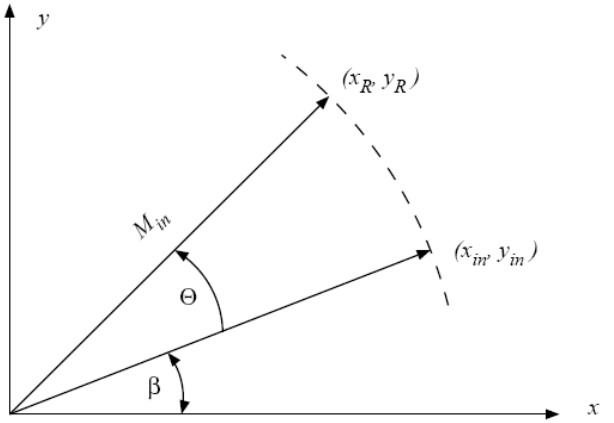
Matematično operacijo vrtenja krajevnega vektorja zapišemo kot:

$$M_{in} \cdot e^{j(\beta+\theta)} = \begin{cases} x_R = M_{in} \cdot \cos(\beta+\theta) \\ y_R = M_{in} \cdot \sin(\beta+\theta) \end{cases}$$

$$x_R = M_{in} \cdot \cos(\beta+\theta) = M_{in} \cdot (\cos(\beta) \cdot \cos(\theta) - \sin(\beta) \cdot \sin(\theta))$$
$$y_R = M_{in} \cdot \sin(\beta+\theta) = M_{in} \cdot (\sin(\beta) \cdot \cos(\theta) + \cos(\beta) \cdot \sin(\theta))$$

$$x_{in} = M_{in} \cos(\beta)$$
$$y_{in} = M_{in} \sin(\beta)$$

$$x_R = x_{in} \cos(\theta) - y_{in} \sin(\theta)$$
$$y_R = x_{in} \sin(\theta) + y_{in} \cos(\theta)$$



Slika 2: Vrtenje krajevnega vektorja v ravnini.

Kot vrtenja θ lahko izrazimo kot vsoto ustrezno predznačenih elementarnih kotov α_k. Ta vsota se v Eulerjevem operatorju vrtenja ($e^{j\theta}$) prevede v produkt posameznih mikrorotacij.

$$\theta = \sum_{k=0}^{\infty} \alpha_k$$
$$M_{in} \cdot e^{j\theta} = M_{in} \cdot \prod_{k=0}^{\infty} e^{j \cdot (\alpha_k)} \qquad (1.1)$$
$$e^{\pm j \cdot (\alpha_k)} = \cos(\alpha_k) \pm j \cdot \sin(\alpha_k)$$

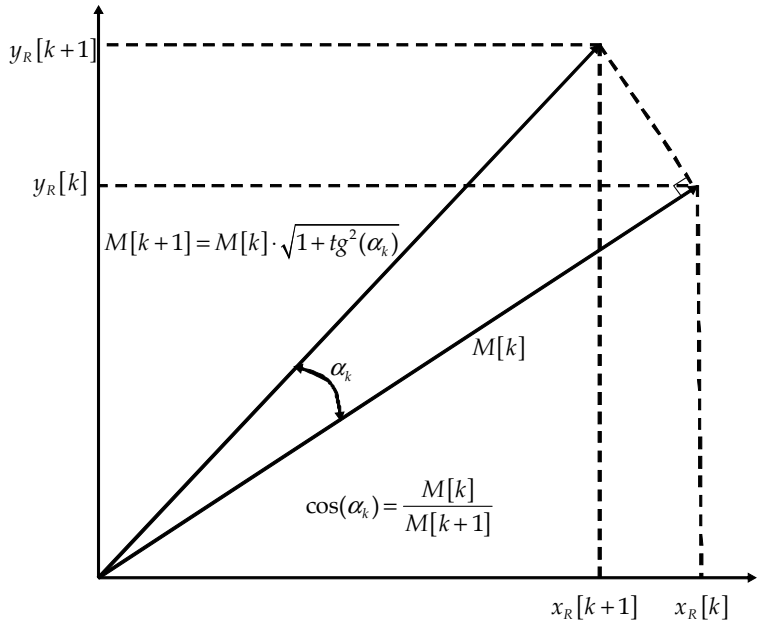
Če operator vrtenja ($e^{j\theta}$) izrazimo za k-to mikrorotacijo dobimo:

$$x_R[k+1] = x_R[k] \cdot \cos(\alpha_k) - y_R[k] \cdot \sin(\alpha_k)$$
$$y_R[k+1] = x_R[k] \cdot \sin(\alpha_k) + y_R[k] \cdot \cos(\alpha_k) \qquad (1.2)$$

Izpostavimo cos(α_k) in dobimo:

$$\begin{aligned} x_R[k+1] &= \cos(\alpha_k)\cdot\left(x_R[k]-y_R[k]\cdot tg(\alpha_k)\right) \\ y_R[k+1] &= \cos(\alpha_k)\cdot\left(y_R[k]+x_R[k]\cdot tg(\alpha_k)\right) \end{aligned} \tag{1.3}$$

Ilustracijo enačbe (1.3) prikazuje slika 3.



Slika 3:Mikrorotacija krajevnega vektorja za kot α_k.

Iz slike 3 in enačbe (1.3) sledita pomembni lastnosti:

1.) Rezultat mikrorotacije (vektor $x_R[k+1]$, $y_R[k+1]$) je za člen $\cos(\alpha_j)$ daljši od prejšnje iteracije $x_R[k]$ $y_R[k]$, kot je prikazano na sliki 3. Pri mikrorotaciji vektorja se njegova dolžina ne sme spreminjati, zato moramo podaljšanje kompenzirati z členom K[k]. Ker se prispevki podaljšanj K[k] medsebojno množijo glede na enačbo (1.1), jih lahko izpostavimo in obravnavamo posebej, tako da izrazimo celotni popravek K kot produkt členov K[k].

$$M[k+1] = K[k] \cdot M[k] = \frac{1}{\cos(\alpha_k)} \cdot M[k]$$
$$K = \prod_{k=0}^{\infty} \left(\frac{1}{\cos(\alpha_k)} \right) = \prod_{k=0}^{\infty} \left(\sqrt{1 + tg^2(\alpha_k)} \right) \tag{1.4}$$

2.) Operacija mikrorotacije zahteva množenje s $tg(\alpha_k)$, kar je aritmetično potratna operacija, če elementarnega kota (α_k) ne izberemo pravilno.
Elementarni kot (α_k) zato izberemo kot *arctg(2^{-k})*, da se izognemo operaciji množenja:

$$tg(\alpha_k) = \pm 2^{-k}$$
$$tg(\alpha_k) = \sigma_k \cdot 2^{-k} \quad \sigma_k = \{1, -1\} \tag{1.5}$$

Če vstavimo izraz za elementarni kot (α_k), dobljeni par enačb ne vsebuje več množenja, temveč smo ga prevedli na predznačeno (σ_k) pomikanje za *k* mest desno - operacija SRA (ang. shift right arithmetic) operandov x in y.

$$x_R[j+1] = \cos(\alpha_j) \cdot \left(x_R[j] - y_R[j] \cdot \sigma_j \cdot 2^{-j} \right)$$
$$y_R[j+1] = \cos(\alpha_j) \cdot \left(y_R[j] + x_R[j] \cdot \sigma_j \cdot 2^{-j} \right) \tag{1.6}$$

V izraz za korekcijski faktor K vstavimo (α_k) in dobimo <u>konvergentno vrsto</u>:

$$K = \prod_{k=0}^{\infty} \left(\sqrt{1 + tg^2(\alpha_k)} \right) = \prod_{k=0}^{\infty} \left(\sqrt{1 + 2^{-2k}} \right) \approx 1.64676025812107 \tag{1.7}$$

CORDIC algoritem v načinu vtenja (ang. rotation mode) uporablja spremenljivki **x** in **y** za hranjenje vrednosti projekcij krajevnega vektorja vrtenja na abscisno in ordinatno os, medtem ko spremenljivka **z** hrani trenutni kot vrtenja. V CORDIC algoritmu za izračun **sin(θ)** in **cos(θ)** spremenljivko **z** spreminjamo z ustrezno predznačenimi (α_k) tako, da se končna vrednost približa **z→0**. Če se ozremo nazaj na sliko 1, CORDIC za splošno vrednost vrtenja krajevnega vektorja (x_{in}, y_{in}) za kot θ izračuna:

$$\begin{aligned} x_f &= K \cdot \left(x_{in} \cos(\theta) - y_{in} \sin(\theta) \right) \\ y_f &= K \cdot \left(x_{in} \sin(\theta) + y_{in} \cos(\theta) \right) \\ z_f &= 0 \end{aligned} \tag{1.8}$$

V primeru izračuna funkcij sin(θ) in cos(θ) postavimo za začetno iteracijo enotni vektor na abscisno os ($y_{in}=y[0]=0$). Glede na enačbo (1.8) dolžino enotnega vektorja korigiramo tako, da ga postavimo na obratno vrednost korekcijskega faktorja ($x_{in}=x[0]=1/K$), s čimer se izognemo končnemu popravljanju rezultata za konstanto K. Po zadnji iteraciji zaporedja mikrorotacij (x_f, y_f) se v spremenljivki ***x*** nahaja vrednost funkcije **cos(θ)**, v spremenljivki **y** pa vrednost **sin(θ)**. Če enačbe CORDIC iteracije strnemo, dobimo:

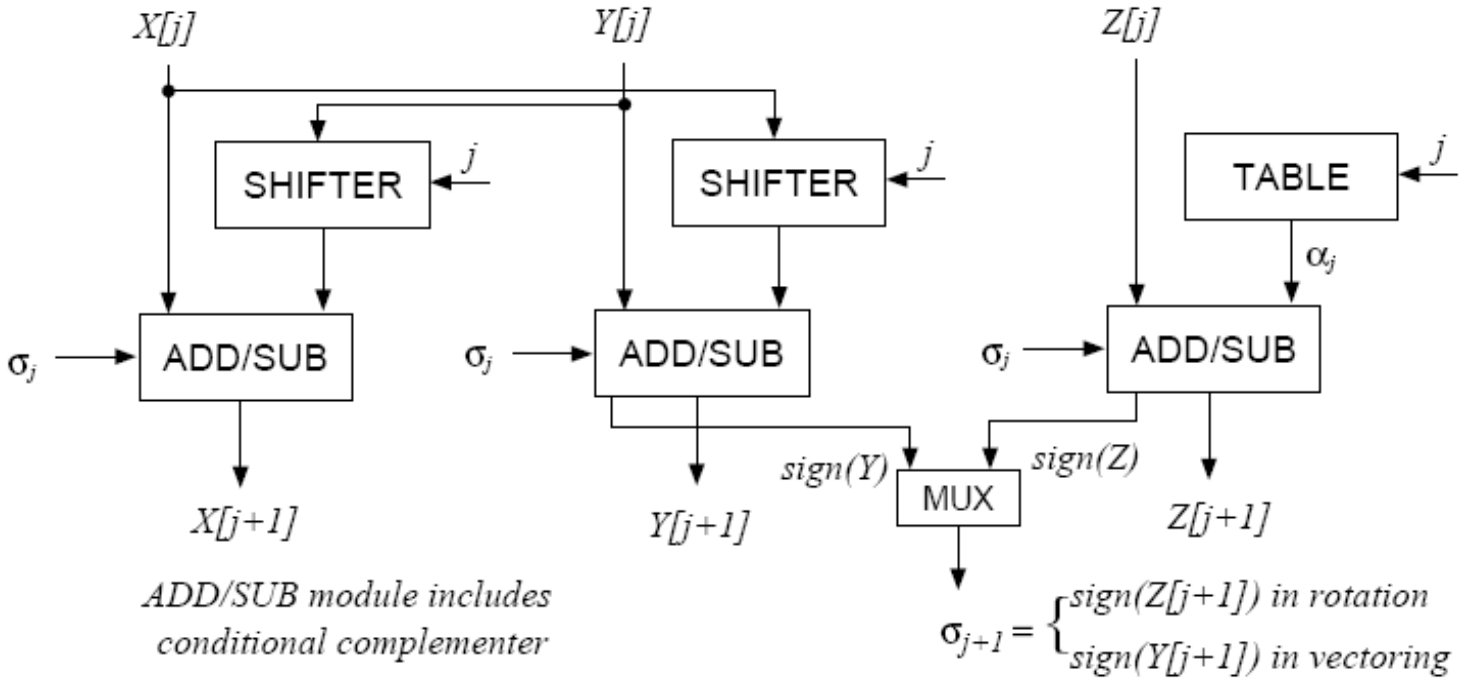
$$\begin{aligned} x[k+1] &= x[k] - y[k] \cdot \sigma_k \cdot 2^{-k} \\ y[k+1] &= y[k] + x[k] \cdot \sigma_k \cdot 2^{-k} \\ z[k+1] &= z[k] - \sigma_k \cdot arctg\left(2^{-k}\right) \end{aligned} \tag{1.9}$$

$$x[0] \approx \frac{1}{1.6468} \quad y[0] = 0 \quad z[0] = \theta \quad \sigma_k = \operatorname{sgn}\left(z[k]\right)$$

Opis podatkovne poti CORDIC algoritma

Slika 4 prikazuje podatkovno pot opisanega CORDIC algoritma. Podatkovna pot vsebuje:

- ROM pomnilnik, ki hrani tabelo elementarnih kotov (TABLE),
- primerjalnik predznaka spremenljivke **z** (σ_k) (SIGN(Z)),
- splošni vzporedni pomikalnik predznačenih števil (SHIFTER),
- vezje seštevalnika/odštevalnika (ADD/SUB) ter
- števca navzgor, ki realizira (`for`) zanko v psevdokodi na začetku predloge (ni narisan).



Slika 4: Poenostavljena podatkovna pot CORDIC algoritma za izračun funkcij sin in cos.

Entiteta izdelane strukture **cordic** naj ima priključke:

```
ENTITY cordic IS
   GENERIC(
      WIDTH : integer := 32
   );
   PORT(
      clk, nRST, start  : IN  std_logic;
      angle             : IN  std_logic_vector ( WIDTH - 1 DOWNTO 0);  -- angle(radians)!!!
      sin, cos          : OUT std_logic_vector ( WIDTH - 1 DOWNTO 0);
      done              : OUT std_logic
   );
END cordic;
```

Vezje vsebuje vhod za kot θ (**angle**), katerega kotni funkciji računamo, in izhoda (**sin, cos**), v katerih se pojavita rezultata operacij sin in cos. Širina kota (**angle**) in rezultata izračuna (**sin, cos**) je (**width**). Izračun kotnih funkcij se izvaja sekvenčno, ko postavimo vhod (**start**) v aktivno stanje '1'. Ko je izračun končan, se postavi izhod (**done**) na '1'. Vezje postavimo v začetno stanje z aktiviranjem vhoda (**nRST**).

Elementi podatkovne poti CORDIC algoritma

ROM pomnilnik s tabelo elementarnih kotov

Za zapis posamezne vrednosti elementarnega kota v tabeli uporabimo obliko zapisa s fiksno vejico (ang. fixpoint notation). Zapis števila v obliki s fiksno vejico podaja spodnja tabela. Število zapišemo na 32 mestih, pri čemer MSB mesto predstavlja predznak števila ('1' → negativno), sledi celi del števila, ki je sestavljen iz dveh bitov uteži 2^1 in 2^0. Na mestih 28…0 se nahaja neceli del števila. Tak zapis ima sicer zelo omejen obseg (±3.999999998), vendar zadošča za računanje kotov v območju ±90°.

Tabela 1: Zapis števila po utežeh v obliki s fiksno vejico

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
±	1	0	−1	−2	−3	−4	−5	−6	−7	−8	−9	−10	−11	−12	−13	−14	−15	−16	−17	−18	−19	−20	−21	−22	−23	−24	−25	−26	−27	−28	−29

Največji kot, ki ga lahko vstavimo v opisani CORDIC algoritem, znaša:

$$\theta_{\max} = \sum_{k=0}^{\infty} arctg\left(2^{-k}\right) \approx 1.74328 rad \approx 99° \tag{1.10}$$

Preostale kote lahko izračunamo z določanjem kvadranta, kot je prikazano v funkcijah (`CORDIC_SIN, CORDIC_COS`) v JavaScript predlogi vaje.
Elementarne kote v podatkovni poti vpišemo v tabelo, ki se nahaja v predlogi vaje **`rom.vhd`**. Pretvorbo iz zapisa s plavajočo vejico in opisanim zapisom s fiksno vejico opravlja funkcija (**`Conv2fixedPt`**) v datoteki (**`Cordic_pkg.vhd`**). Funkcija pretvori realno število (**`x`**) v zapis s fiksno vejico z (**`n`**) dvojiškimi decimalkami.

```
function Conv2fixedPt (x : real; n : integer) return std_logic_vector
```

Obratno pretvorbo opravlja funkcija (**`Conv2real`**), ki pretvori število (**`s`**), zapisano s fiksno vejico z (**`n`**) dvojiškimi decimalkami v realno število.

```
function Conv2real (s, n : in integer) return real
```

Funkcijo (**`Conv2fixedPt`**) bomo uporabljali za pretvorbo začetnih CORDIC vrednosti za register (**`x0`**), medtem ko se funkcija (**`Conv2real`**) uporablja v priloženi datoteki testnih vrednosti.

Vsebino ROM pomnilnika določimo po enačbi (1.5). ROM element bo vseboval (`sizeof(SIZE - 1)`) naslovov, na katerih se nahajajo zapisani elementarni koti širine (`width`).

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use WORK.cordic_pkg.all;

entity CORDIC_ROM is
   GENERIC(
         WIDTH :     integer := 32;
         SIZE  :     integer := 24
   );
   port (      addr : in std_logic_vector(sizeof(SIZE - 1) - 1 downto 0);
               dout : out std_logic_vector(WIDTH - 1 DOWNTO 0)
         );
end CORDIC_ROM;
```

Tabela 2: Elementarni koti za CORDIC izračun funkcij sin in cos

j	$arctg(2^{-j})$	$arctg(2^{-j})2^{29}$	ROM_{16}	j	$arctg(2^{-j})$	$arctg(2^{-j})2^{29}$	ROM_{16}	j	$arctg(2^{-j})$	$arctg(2^{-j})2^{29}$	ROM_{16}	j	$arctg(2^{-j})$	$arctg(2^{-j})2^{29}$	ROM_{16}
0	0.785398163	421657428	`1921FB54`	*8*	0.00390623	2097141	`1FFFF5`	*16*	1.52588E-05	8192	`2000`	*24*	5.96046E-08	32	`20`
1	0.463647609	248918915	`ED63383`	*9*	0.001953123	1048575	`FFFFF`	*17*	7.62939E-06	4096	`1000`	*25*	2.98023E-08	16	`10`
2	0.244978663	131521918	`7D6DD7E`	*10*	0.000976562	524288	`80000`	*18*	3.8147E-06	2048	`800`	*26*	1.49012E-08	8	`8`
3	0.124354995	66762579	`3FAB753`	*11*	0.000488281	262144	`40000`	*19*	1.90735E-06	1024	`400`	*27*	7.45058E-09	4	`4`
4	0.06241881	33510843	`1FF55BB`	*12*	0.000244141	131072	`20000`	*20*	9.53674E-07	512	`200`	*28*	3.72529E-09	2	`2`
5	0.031239833	16771758	`FFEAAE`	*13*	0.00012207	65536	`10000`	*21*	4.76837E-07	256	`100`	*29*	1.86265E-09	1	`1`
6	0.015623729	8387925	`7FFD55`	*14*	6.10352E-05	32768	`8000`	*22*	2.38419E-07	128	`80`	*30*	9.31323E-10	0	`0`
7	0.007812341	4194219	`3FFFAB`	*15*	3.05176E-05	16384	`4000`	*23*	1.19209E-07	64	`40`	*31*	4.65661E-10	0	`0`

Števec iteracij CORDIC zanke

Števec iteracij je izveden kot dvojiški števec navzgor po nastavljivem modulu (`MODULO`). Vsebina štetja (`count`) se ob aktiviranju signala (`nRST`) postavi na nič. Če je štetje omogočeno (`count_enable`), se ob vsakem prednjem robu signala ure (`clk`) vsebina štetja (`count`) poveča za 1. Ko štetje doseže najvišjo vrednost (`MODULO - 1`) se kombinacijsko aktivira izhod (`rco`). Funkcija (`sizeof`) vrne število mest, s katerimi lahko zapišemo najvišjo vrednost štetja (`MODULO - 1`). Definirana je v paketni datoteki (`cordic_pkg.vhd`). Arhitektura števca je podana v predlogi vaje.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use WORK.cordic_pkg.all;

ENTITY up_counter IS
   GENERIC(
         MODULO : integer := 24
   );
   PORT( clk,
         nRST,
         count_enable      : IN std_logic;
         count             : OUT std_logic_vector( sizeof(MODULO - 1) - 1 DOWNTO 0);
         rco               : OUT std_logic
   );
END up_counter;
```

Podatkovni register

Podatkovni register (`data_reg`), ki ga boste uporabljali za izvedbo podatkovne poti CORDIC algoritma, je splošni podatkovni register velikosti (`WIDTH`), ki vsebuje vhod za sinhrono nalaganje (`load`). Vsebina na vhodu za vzporedno nalaganje (`X`) se ob prednjem robu signala ure (`clk`) naloži v register (`Q`), če je aktiviran signal (`load`), sicer se vsebina ohranja. Arhitektura registra je podana v predlogi vaje.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY data_reg IS
   GENERIC(
         WIDTH : integer := 32
   );
   PORT( clk, load   : IN std_logic;
         X           : IN std_logic_vector (WIDTH-1 DOWNTO 0);       -- load value (loads X when load = '1')
         Q           : OUT std_logic_vector (WIDTH-1 DOWNTO 0)       -- output value
   );
END data_reg;
```

Enota za seštevanje/odštevanje

Enota za seštevanje/odštevanje (`addsub`), ki jo boste uporabljali za izvedbo podatkovne poti CORDIC algoritma, je poenostavljena aritmetično-logična enota (ALU), ki nad vhodnima operandoma (`A, B`) širine (`WIDTH`) opravlja operacijo seštevanja, ko je krmilni signal (`add_nsub = '1'`), sicer operanda odšteva. Kratko bi delovanje enote povzeli z enim VHDL stavkom:

```
S <= (A + B) when ( add_nsub = '1') else (A - B);     -- ta realizacija uporablja RC seštevalnik
```

vendar bi tovrstna realizacija vodila do uporabe RC seštevalnika. Vezje seštevalnika/odštevalnika zaradi hitrosti delovanja realiziramo kot CLA seštevalnik, pri katerem nad vhodnim operandom (`B`) tvorimo pogojni dvojiški komplement (`-B`), kot je opisano na predavanjih (prosojnica 156). Dvojiški komplement tvorimo takrat, ko je zahtevana operacij odštevanja (`add_nsub = '0'`). Takrat vsebino operanda (`B`) negiramo, da dobimo eniški komplement:

```
B_sig <= B when ( add_nsub = '1') else not(B); -- eniški komplement
```

in na vhodni prenos seštevalnika postavimo na '1', kar storimo z ukazom (`Cin => nadd_sub`) v povezovalnem stavku. Arhitektura enote za seštevanje/odštevanje je podana v predlogi vaje.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY addsub IS
   GENERIC(
         WIDTH : integer := 32
   );
   PORT(
         A , B       : IN     std_logic_vector (WIDTH-1 DOWNTO 0);
         S           : OUT    std_logic_vector (WIDTH-1 DOWNTO 0);
         add_nsub    : IN     std_logic      -- add_nsub = '0'->subtraction, '1'->addition
   );
END addsub;
```

Naloge

1. Ustvarite *nov projekt* z imenom **BARREL_SHIFTER_SRA** v katerega dodajte VHDL datoteko **barrel_shifter_sra.vhd** iz predloge projekta v imeniku **BARREL_SHIFTER_SRA**. V datoteki (**barrel_shifter_sra.vhd**) programirajte arhitekturo splošne strukture vzporednega pomikalnika predznačenih podatkov (**barrel_shifter_sra**). Podana je entiteta strukture:

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;
use WORK.cordic_pkg.all;

ENTITY barrel_shifter_sra IS
	GENERIC(
		width : integer := 32;
		size  : integer := 32
	);
	PORT(
		input  : IN     std_logic_vector (WIDTH-1 DOWNTO 0);
		output : OUT    std_logic_vector (WIDTH-1 DOWNTO 0);
		n      : IN     std_logic_vector (sizeof(size - 1) - 1 DOWNTO 0)
	);
END barrel_shifter_sra;

architecture ndv of barrel_shifter_sra is

type barrel_shifter_type is array (0 to size - 1) of std_logic_vector (width - 1 downto 0);
signal barrel_shifter : barrel_shifter_type := ( others => (others => '0') );
begin
end ndv;
```

Parameter (`width`) določa število bitov podatka vzporednega pomikalnika predznačenih podatkov, parameter (`size`) določa koliko biten je lahko največji pomik pomikalnika. Vzporedni pomikalnik podatkov kombinacijsko izvaja operacijo SRA (ang. shift right arithmetic) nad vhodnim podatkom (`input`). Izhod vezja (`output`) je za (`n`) mest pomaknjen vhodni podatek, pri čemer se predznak vhodnega podatka ohranja (ang. sign extension). Vezje programirajte tako, da znotraj procesnega stavka v (`for…loop`) zanki tvorite vse možne pomike vhodnega podatka (`input`) in jih zapišete v enodimenzionalno polje (`barrel_shifter`). Vse možne pomike vhodnega podatka izvedete z ukazom (`resize`). Ta ukaz dano vhodno število spremeni na podano velikost, s tem da pri tem upošteva predznak števila. Če je število predznačeno (`signed`), bo ta ukaz preostala mesta do podane velikosti dopolnil z mestom predznaka (MSB bit števila). Če je vhodni podatek

del vektorja od MSB do tekočega indeksa i (`input(WIDTH - 1 downto i)`), bo zanka (`for…loop`) pri iteracijah (`size-1 downto 0`) tvorila vse možne pomike. Če povedano strnemo, dobimo:

```
barrel_shifter(i) <= std_logic_vector(resize(signed(input(WIDTH - 1 downto i)), output'length));
```

Ko je enodimenzionalno polje (`barrel_shifter`) ustvarjeno, je potrebno glede na vhodni parameter (`n`) izbrati element polja, ki se pojavi na izhodu. To storimo z izbiralnikom vodil:

```
output <= barrel_shifter(to_integer(unsigned(n))); -- bus multiplexer
```

Pri preverjanju pravilnosti delovanja uporabite priloženo datoteko testnih vrednosti (**`barrel_shifter_sra_tb.vhd`**) in s simulacijo preverite pravilnost delovanja splošne strukture vzporednega pomikalnika predznačenih podatkov.

2. Ustvarite *nov projekt* z imenom **CORDIC_FSM** (imenik v predlogi) v katerega dodajte VHDL datoteko **cordic_fsm.vhd** iz predloge projekta v imeniku **CORDIC_FSM**. V datoteki (**cordic_fsm.vhd**) programirajte entiteto in arhitekturo Moore-ovega avtomata končnih stanj, ki krmili delovanje CORDIC algoritma. Podana je entiteta avtomata (CORDIC_FSM) in diagram prehajanja stanj:

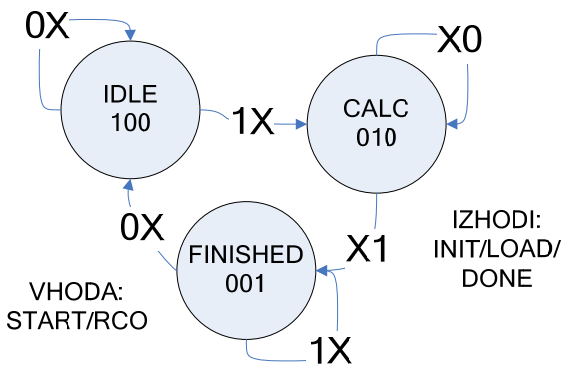
```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

ENTITY CORDIC_FSM IS
      PORT(
            clk, nRST, start, rco   : IN std_logic;
            init, load, done        : OUT std_logic
      );
END CORDIC_FSM;

ARCHITECTURE rtl OF CORDIC_FSM IS

TYPE STATE_TYPE IS ( IDLE, CALC, FINISHED );
SIGNAL state, next_state : STATE_TYPE := IDLE;

BEGIN
END rtl;
```



Slika 5: Avtomat končnih stanj za krmiljenje CORDIC algoritma.

Programirajte delovanje avtomata v VHDL po priloženem diagramu prehajanja stanj na sliki 5. Avtomat ima tri stanja (**IDLE, CALC, FINISHED**). Ob aktivnem signalu (nRST) se postavi v stanje (IDLE). Ob aktiviranju (start) signala preide v stanje (CALC), kjer ostaja dokler se ne aktivira vhod iz števca iteracij CORDIC algoritma (rco). V stanju (CALC) postavi izhod (load). Takrat preide v stanje (FINISHED), kjer čaka na deaktiviranje (start) signala. V stanju (FINISHED) postavi izhod (done). Ko se signal (start) deaktivira se vrne v stanje (IDLE).

Pri preverjanju pravilnosti delovanja uporabite priloženo datoteko testnih vrednosti (**CORDIC_FSM_tb.vhd**) in s simulacijo preverite pravilnost delovanja Moore-ovega avtomata končnih stanj, ki krmili delovanje CORDIC algoritma.

3. Ustvarite *nov projekt* z imenom **CORDIC_MAIN** (imenik v predlogi) v katerega dodajte VHDL datoteko **cordic.vhd** iz predloge projekta v imeniku **CORDIC_MAIN** in datoteke prej izdelanega vzporednega pomikalnika podatkov (**barrel_shifter_sra.vhd**) ter avtomata končnih stanj (**cordic_fsm.vhd**). V datoteki (**cordic.vhd**) programirajte arhitekturo CORDIC algoritma. Podana je entiteta:

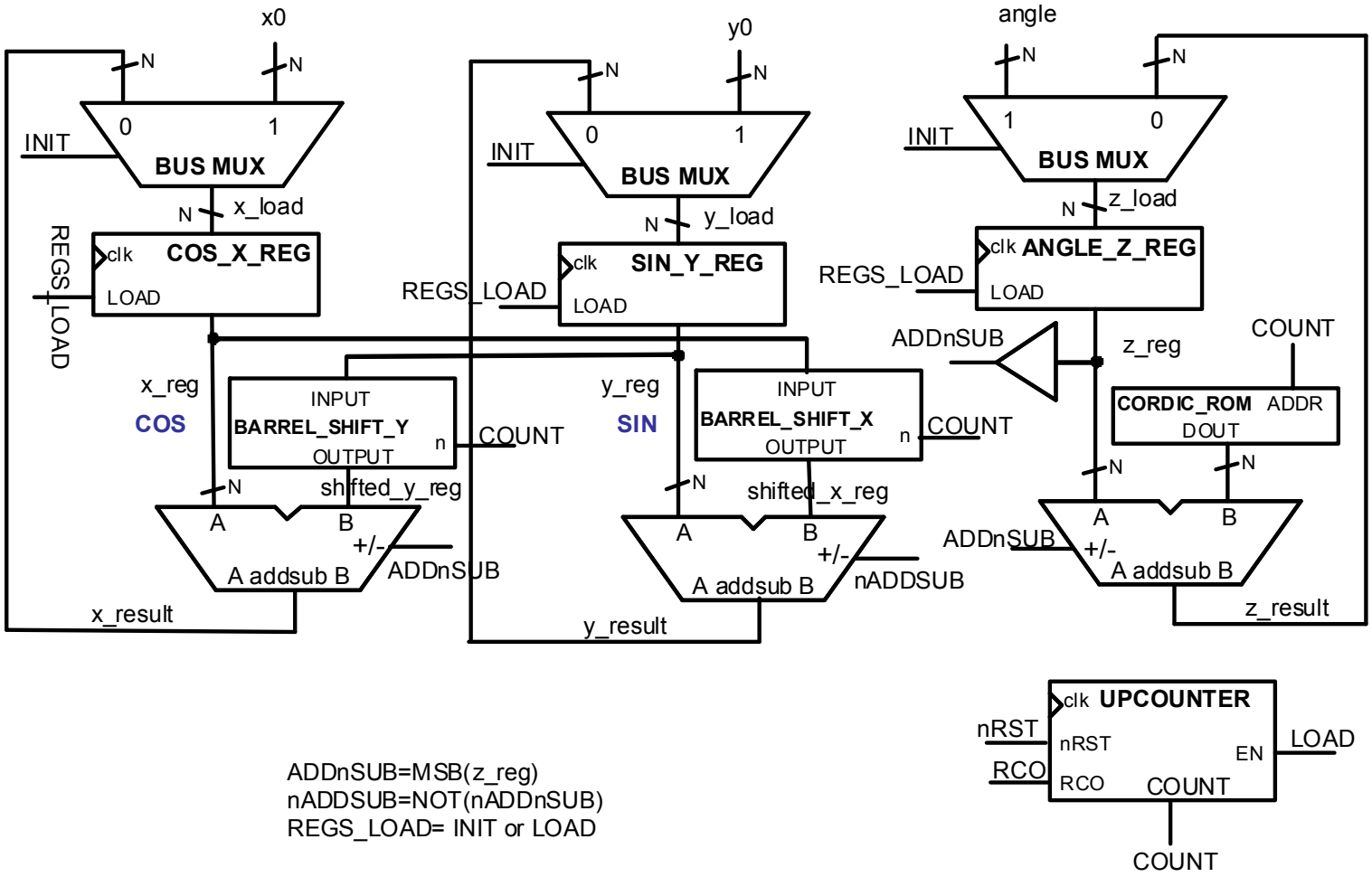
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use WORK.cordic_pkg.all;

ENTITY cordic IS
   GENERIC(
      WIDTH : integer := 32
   );
   PORT(
      clk, nRST, start  : IN  std_logic;
      angle             : IN  std_logic_vector ( WIDTH - 1 DOWNTO 0);  -- angle(radians)!!!
      sin, cos          : OUT std_logic_vector ( WIDTH - 1 DOWNTO 0);
      done              : OUT std_logic
   );
END cordic;

ARCHITECTURE rtl OF cordic IS

SIGNAL      count : std_logic_vector(sizeof(WIDTH - 1) - 1 DOWNTO 0);  -- if width=32, count goes from 0 to 31
SIGNAL      x_load, y_load, z_load,
            x_reg, y_reg, z_reg,
            x_result, y_result, z_result,
            cordic_coefficient,
            shifted_y_reg, shifted_x_reg  : std_logic_vector(WIDTH-1 DOWNTO 0) := (others => '0');
SIGNAL init, load, add_nsub, nadd_sub, rco, regs_load : std_logic;
constant x0 : std_logic_vector(WIDTH - 1 downto 0) := Conv2fixedPt ( inverseK, WIDTH);  -- 1/1.64676025812107
constant y0 : std_logic_vector(WIDTH - 1 downto 0) := (others => '0');
BEGIN
END rtl;
```

Povezovanje komponent podatkovne poti izvedite po priloženi shemi na sliki 6. Potrebne komponente podatkovne poti se nahajajo v imeniku predloge, manjkata datoteki krmilnega avtomata in vzporednega pomikalnika podatkov. V datoteko (**cordic.vhd**) povežite krmilni avtomat (**CORDIC_FSM**), ki ste ga programirali v prejšnji točki. Izbiralnike vodil na sliki 6 (ang. bus multiplexer) programirajte z uporabo (`when-else`) stavka. Pri preverjanju pravilnosti delovanja uporabite priloženo datoteko testnih vrednosti (**CORDIC_tb.vhd**) in s simulacijo preverite pravilnost delovanja CORDIC algoritma.



Slika 6: Podrobna shema podatkovne poti CORDIC algoritma.

Na strežnik naložite *samo* datoteke:

`barrel_shifter_sra.vhd`,
`cordic_fsm.vhd`,
`cordic.vhd`.

Dodatno

CORDIC algoritem poleg načina vrtenja (ang. rotation mode) obstaja tudi v vektorskem načinu (ang. vectoring mode). Poleg krožnega koordinatnega sistema (ang. circular) ga lahko uporabljamo še v linearnem (ang. linear) ali hiperboličnem (ang. hyperbolic) koordinatnem sistemu. Če želimo izračunati vrednosti funkcij *sinh* in *cosh*, ohranimo elemente podatkovne poti te vaje. Spremenimo le tabelo elementarnih kotov: Izračunamo jih kot $tanh(\alpha_j)=2^{-j}$, pri čemer indeks j teče od 1. Nekatere vrednosti j se ponovijo, s čimer dosežemo konvergenco vrste korekcijskih faktorjev K. Vsota te konvergentne vrste znaša $K_{-1}=0.82816$. Začetno vrednost registra x prednaložimo z vrednostjo $x_0=1/K_{-1}$. Po mikrorotacijah se v registru x nahaja vrednost *cosh*, v registru y pa *sinh*. Operacije, ki jih lahko računamo z uporabo CORDIC algoritma, so strnjene v tabelah 3 in 4 [1].

Tabela 3: Poenoten opis CORDIC algoritma.

Coordinates	Rotation mode $\sigma_j = sign(z[j])^+$	Vectoring mode $\sigma_j = -sign(y[j])^+$
Circular $(m = 1)$ $\alpha_j = \tan^{-1}(2^{-j})$ initial $j = 0$ $j = 0, 1, 2, .., n$ $K_1 \approx 1.64676$ $\theta_{max} \approx 1.74329$	$x_f = K_1(x_{in}\cos(z_{in}) - y_{in}\sin(z_{in}))$ $y_f = K_1(x_{in}\sin(z_{in}) + y_{in}\cos(z_{in}))$ $z_f = 0$	$x_f = K_1(x_{in}^2 + y_{in}^2)^{1/2}$ $y_f = 0$ $z_f = z_{in} + \tan^{-1}(\frac{y_{in}}{x_{in}})$
Linear $(m = 0)$ $\alpha_j = 2^{-j}$ initial $j = 0$ $j = 0, 1, , 2, ..., n$ $K_0 = 1$ $\theta_{max} = 2 - 2^{-n}$	$x_f = x_{in}$ $y_f = y_{in} + x_{in}z_{in}$ $z_f = 0$	$x_f = x_{in}$ $y_f = 0$ $z_f = z_{in} + \frac{y_{in}}{x_{in}}$
Hyperbolic $(m = -1)$ $\alpha_j = \tanh^{-1}(2^{-j})$ initial $j = 1$ $j = 1, 2, 3, 4, 4, 5...13, 13, ...$ $K_{-1} \approx 0.82816$ $\theta_{max} \approx 1.11817$	$x_f = K_{-1}(x_{in}\cosh(z_{in}) + y_{in}\sinh(z_{in}))$ $y_f = K_{-1}(x_{in}\sinh(z_{in}) + y_{in}\cosh(z_{in}))$ $z_f = 0$	$x_f = K_{-1}(x_{in}^2 - y_{in}^2)^{1/2}$ $y_f = 0$ $z_f = z_{in} + \tanh^{-1}(\frac{y_{in}}{x_{in}})$

$^+$ $sign(a) = 1 \; if \; a \geq 0, \quad sign(a) = -1 \; if \; a < 0.$

Tabela 4: Nekatere operacije, ki jih računa CORDIC algoritem.

m	Mode	Initial values			Functions	
		x_{in}	y_{in}	z_{in}	x_R	y_R or z_R
1	rotation	1	0	θ	$\cos\theta$	$y_R = \sin\theta$
-1	rotation	1	0	θ	$\cosh\theta$	$y_R = \sinh\theta$
-1	rotation	a	a	θ	ae^{θ}	$y_R = ae^{\theta}$
1	vectoring	1	a	$\pi/2$	$\sqrt{a^2+1}$	$z_R = \cot^{-1}(a)$
-1	vectoring	a	1	0	$\sqrt{a^2-1}$	$z_R = \coth^{-1}(a)$
-1	vectoring	$a+1$	$a-1$	0	$2\sqrt{a}$	$z_R = 0.5\ln(a)$
-1	vectoring	$a+\frac{1}{4}$	$a-\frac{1}{4}$	0	$\sqrt{a}$	$z_R = \ln(\frac{1}{4}a)$
-1	vectoring	$a+b$	$a-b$	0	$2\sqrt{ab}$	$z_R = 0.5\ln(\frac{a}{b})$

Seznam literature

[1] Ercegovac, Lang "*Digital arithmetic*", spletno gradivo k knjigi,
http://www.cs.ucla.edu/digital_arithmetic/files/ch11.pdf (obiskano dne 22.6.2013)

[2] Ercegovac, Lang, "*Digital arithmetic*", Elsevier Science, 2003, ISBN 1-55860-798-6

[3] Pongyupinpanich Surapong, Faizal Arya Samman and Manfred Glesner "*Design and Analysis of Extension-Rotation CORDIC Algorithms based on Non-Redundant Method*",
http://www.sersc.org/journals/IJSIP/vol5_no1/6.pdf (obiskano dne 18.6.2013)

[4] J. S. Walther, "*A unified algorithm for elementary functions*", Spring Joint Computer Conference proceedings, 1971
http://www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf (obiskano dne 8.6.2013)

[5] Dirk, "*CORDIC Algorithm COordinate Rotation DIgital Computer*", Course INF5430 - Spring 2012
http://www.uio.no/studier/emner/matnat/ifi/INF5430/v12/undervisningsmateriale/dirk/Lecture_cordic.pdf (obiskano dne 8.6.2013)

[6] Rick Parris, " Elementary Functions and Calculators", Phillips Exeter Academy
http://math.exeter.edu/rparris/peanut/cordic.pdf (obiskano dne 12.6.2013)

[7] M. J. Flynn,*"The Higher Level Functions (HLFs)"*, Stanford University,
http://www.stanford.edu/class/ee486/doc/lecture18.pdf (obiskano dne 6.6.2013)

[8] Young W. Lim "*CORDIC in VHDL (1A)*", 2011,
http://upload.wikimedia.org/wikiversity/en/0/0d/CORDIC.VHDL.1.A.20111110.pdf (obiskano dne 17.6.2013)

[9] J.C. Bajard, S. Kla, J. M. Muller: "*BKM: A New Hardware Algorithm for Complex Elementary Functions*",
IEEE TRANSACTIONS ON COMPUTERS, VOL. 43, NO. 8, 1994 str. 955−963
http://hal.archives-ouvertes.fr/docs/00/08/68/94/PDF/BKM94.pdf (obiskano dne 17.6.2013)