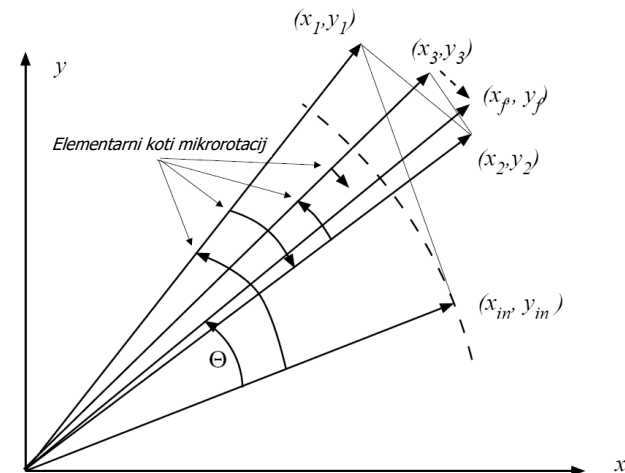


CORDIC algoritem za izračun $\sin(\theta)$ in $\cos(\theta)$

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_unified.htm](#)) v predlogi vaje. V tem dokumentu se izpišejo izračuni vseh operacij po poenotenem 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:

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

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

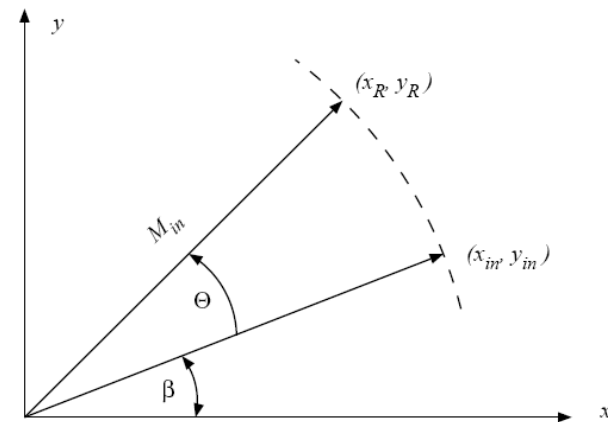
$$y_R = M_{in} \cdot \sin(\beta + \theta) = M_{in} \cdot (\sin(\beta) \cdot \cos(\theta) + \cos(\beta) \cdot \sin(\theta))$$

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

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

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

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



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.

$$\begin{aligned} \theta &= \sum_{k=0}^{\infty} \alpha_k \\ M_{in} \cdot e^{j\theta} &= M_{in} \cdot \prod_{k=0}^{\infty} e^{j(\alpha_k)} \\ e^{\pm j(\alpha_k)} &= \cos(\alpha_k) \pm j \cdot \sin(\alpha_k) \end{aligned} \tag{0.1}$$

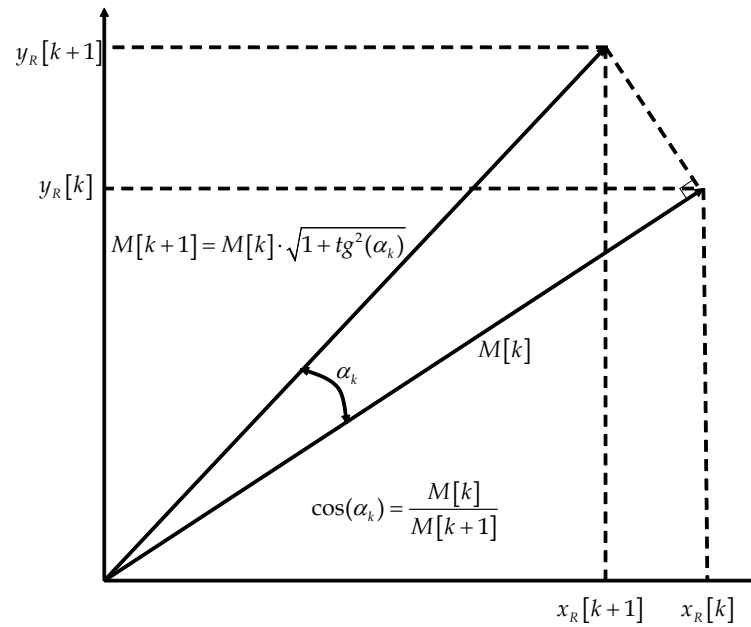
Če operator vrtenja ($e^{j\theta}$) izrazimo za k-to mikrorotacijo dobimo:

$$\begin{aligned}x_R[k+1] &= x_R[k] \cdot \cos(\alpha_k) - y_R[k] \cdot \sin(\alpha_k) \\ y_R[k+1] &= x_R[k] \cdot \sin(\alpha_k) + y_R[k] \cdot \cos(\alpha_k)\end{aligned}\quad (0.2)$$

Izpostavimo $\cos(\alpha_k)$ in dobimo:

$$\begin{aligned}x_R[k+1] &= \cos(\alpha_k) \cdot (x_R[k] - y_R[k] \cdot \tan(\alpha_k)) \\ y_R[k+1] &= \cos(\alpha_k) \cdot (y_R[k] + x_R[k] \cdot \tan(\alpha_k))\end{aligned}\quad (0.3)$$

Ilustracijo enačbe (0.3) prikazuje slika 3.



Slika 3: Mikrorotacija krajevnega vektorja za kot α_k .

Iz slike 3 in enačbe (0.3) 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 (0.1), jih lahko izpostavimo in obravnavamo posebej, tako da izrazimo celotni popravek K kot produkt členov $K[k]$.

$$\begin{aligned} M[k+1] &= K[k] \cdot M[k] = \frac{1}{\cos(\alpha_k)} \cdot M[k] \\ K &= \prod_{k=0}^{\infty} \left(\frac{1}{\cos(\alpha_k)} \right) = \prod_{k=0}^{\infty} \left(\sqrt{1 + tg^2(\alpha_k)} \right) \end{aligned} \quad (0.4)$$

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:

$$\begin{aligned} tg(\alpha_k) &= \pm 2^{-k} \\ tg(\alpha_k) &= \sigma_k \cdot 2^{-k} \quad \sigma_k = \{1, -1\} \end{aligned} \quad (0.5)$$

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 .

$$\begin{aligned} x_R[j+1] &= \cos(\alpha_j) \cdot (x_R[j] - y_R[j] \cdot \sigma_j \cdot 2^{-j}) \\ y_R[j+1] &= \cos(\alpha_j) \cdot (y_R[j] + x_R[j] \cdot \sigma_j \cdot 2^{-j}) \end{aligned} \quad (0.6)$$

V izraz za korekcijski faktor K vstavimo (α_k) in dobimo konvergentno vrsto:

$$K = \prod_{k=0}^{\infty} \left(\sqrt{1 + tg^2(\alpha_k)} \right) = \prod_{k=0}^{\infty} \left(\sqrt{1 + 2^{-2k}} \right) \approx 1.64676025812107 \quad (0.7)$$

CORDIC algoritem v načinu vrtenja (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 $\sin(\theta)$ in $\cos(\theta)$ spremenljivko z spreminjamo z ustrezno predznačenimi (α_k) tako, da se končna vrednost približa $z \rightarrow 0$. Če se ozremo nazaj na sliko 1, CORDIC za splošno vrednost vrtenja krajevnega vektorja (x_{in}, y_{in}) za kot θ izračuna:

$$\begin{aligned} x_f &= K \cdot (x_{in} \cos(\theta) - y_{in} \sin(\theta)) \\ y_f &= K \cdot (x_{in} \sin(\theta) + y_{in} \cos(\theta)) \\ z_f &= 0 \end{aligned} \quad (0.8)$$

V primeru izračuna funkcij $\sin(\theta)$ in $\cos(\theta)$ postavimo za začetno iteracijo enotni vektor na abscisno os ($y_{in}=y[0]=0$). Glede na enačbo (0.8) 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 $\cos(\theta)$, v spremenljivki y pa vrednost $\sin(\theta)$. Če enačbe CORDIC iteracije strnemo, dobimo:

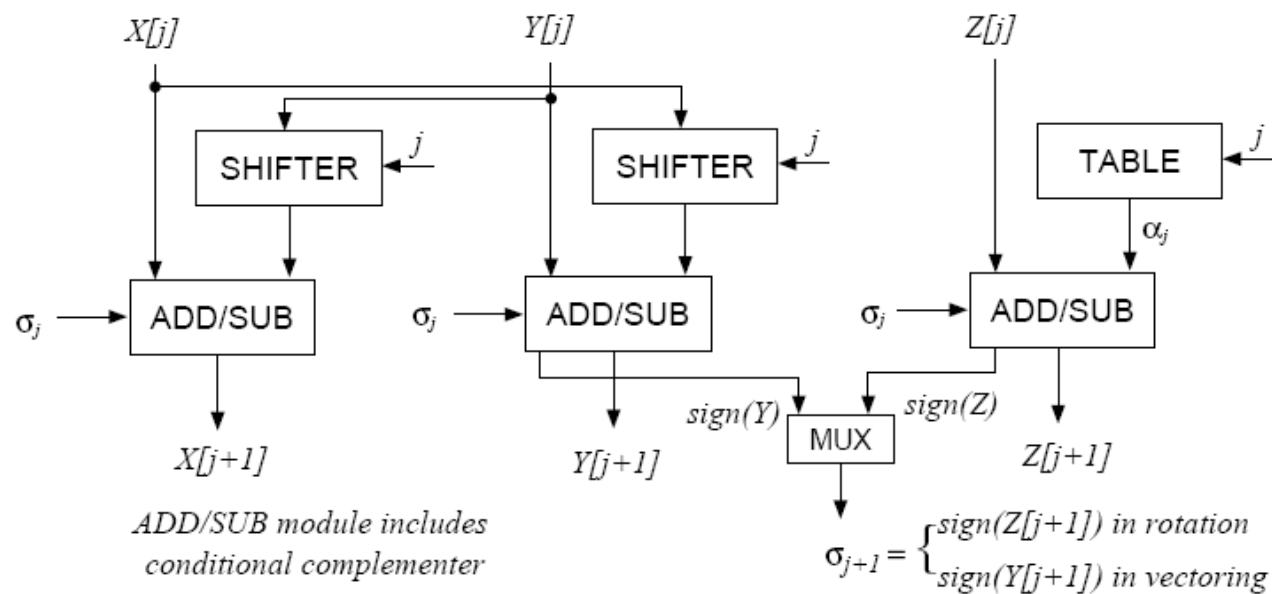
$$\begin{aligned} x[k+1] &= x[k] - y[k] \cdot \sigma_k \cdot 2^{-k} \\ y[k+1] &= y[k] + x[k] \cdot \sigma_k \cdot 2^{-k} \\ z[k+1] &= z[k] - \sigma_k \cdot \arctg(2^{-k}) \end{aligned} \quad (0.9)$$

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

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),
- števec iteracij, ki realizira (**for**) zanko v psevdokodi na začetku predloge (ni narisano),
- Krmilni avtomat (FSM) kot uporabniški vmesnik (ni narisano).



Slika 4: Podatkovna pot CORDIC algoritma v načinu vrtenja in vektorskem načinu.

Entiteta izdelane strukture `cordic_unified` naj ima priključke:

```
entity cordic_unified is
generic(
    WIDTH : integer := 32;
    SIZE : integer := 24
);
port(
    clk, nRST, start : in std_logic;
    system           : in COORD_SYSTEM;
    mode             : in CORDIC_MODE;
    angle_in         : in std_logic_vector (WIDTH-1 downto 0);
    x_in, y_in       : in std_logic_vector (WIDTH-1 downto 0);
    res_x, res_y     : out std_logic_vector (WIDTH-1 downto 0);
    res_z            : out std_logic_vector (WIDTH-1 downto 0);
    done             : out std_logic
);
end cordic_unified;
```

Vezje vsebuje vhod za kot θ (`angle_in`), katerega kotni funkciji računamo, in izhode registrov (`res_x`, `res_y`, `res_z`), v katerih se pojavijo rezultati operacij CORDIC glede na izbrani način delovanja (`mode`) in koordinatni sistem (`system`). Širina zapisa vhodne točke krajevnega vektorja (`x_in`, `y_in`) in začetnega kota (`angle_in`) ter rezultatov izračuna (`res_x`, `res_y`, `res_z`) je (`WIDTH`). Število elementov tabele elementarnih kotov v ROM pomnilniku je (`SIZE`). 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 $\pm 90^\circ$. Za ostale operacije moramo vedno preveriti, če slučajno korak katerega izračuna ne vodi v nasičenje – takrat CORDIC ne konvergira in rezultat bo napačen.

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:

$$\theta_{\max} = \sum_{k=0}^{\infty} \arctg(2^{-k}) \approx 1.74328 \text{ rad} \approx 99^\circ \quad (0.10)$$

Pretvorbo iz zapisa s plavajočo vejico in opisanim zapisom s fiksno vejico opravlja funkcija (`Conv2fixedPt`) v datoteki (`cordic_unified_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 za izpis v berljivi obliki z decimalno vejico.

Elementarne kote v podatkovni poti vpišemo v tabelo, ki bo vsebovana v ROM komponenti, deklarirani kot (`rom_type`):

```
type rom_type is array (0 to SIZE-1) of signed(WIDTH-1 downto 0);
```

ROM element bo vseboval (`SIZE`) naslovov, na katerih se nahajajo zapisani elementarni koti s fiksno vejico širine (`WIDTH`). Vsebino ROM pomnilnika bomo določili s funkcijo (`init_circ_rom`), ki bo vrnila polje izračunanih elementarnih kotov v zapisu s fiksno vejico. Definirajte funkcijo (`function init_circ_rom`), ki vrne tip (`rom_type`). Znotraj funkcije definirajte spremenljivki (`temp_rom`) tipa (`rom_type`), ki služi sprotnemu hranjenju izračunanih kotov. Definirajte tudi pomožno spremenljivko (`val_real`) tipa (`real`). V zanki, ki teče po vseh elementih polja (`temp_rom`) izračunajte dani kot po enačbi (0.5), s tem da tekoči indeks zanke (`i`) pretvorite v realno število (`real(i)`). Z dobljenim številom nato izračunate potenco 2^{-i} ter jo shranite v (`val_real`). Tekoči element ROM (`temp_rom(i)`) določimo tako, da nad izračunano vrednostjo (`val_real`) izvedemo matematično funkcijo (`ARCTAN`), ki se nahaja v matematični knjižnici (`ieee.math_real.all`). Dobljeno vrednost kota pretvorimo v zapis s fiksno vejico širine (`WIDTH`) z uporabo funkcije (`Conv2fixedPt`). Ob izhodu funkcije vrnemo izračunane kote v polju (`rom_type`). Za potrebe CORDIC v krožnem koordinatnem sistemu (ang. circular) bomo deklarirali konstanto (`constant LUT_CIRC`), katere začetno vrednost bomo vzpostavili s klicem funkcije (`init_circ_rom`).

V nadaljevanju vaje ob razširitvi algoritma bomo uporabljali tudi funkcijo (`init_hyp_rom`) v hiperboličnem koordinatnem sistemu – izvedba je podobna, le da se elementarni koti računajo z uporabo funkcije (`ARCTANH`), ki se nahaja v matematični knjižnici

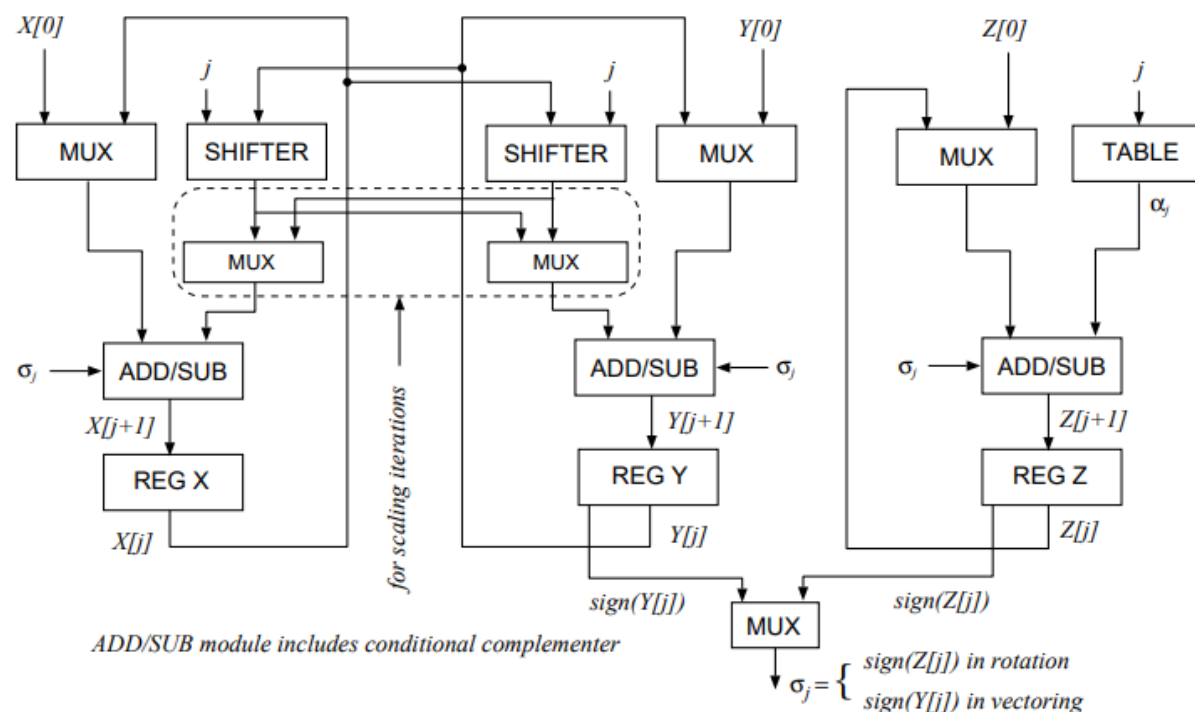
(`ieee.math_real.all;`). Za potrebe CORDIC v hiperboličnem koordinatnem sistemu (ang. hyperbolic) bomo deklarirali konstanto (`constant` LUT_HYP), katere začetno vrednost bomo vzpostavili s klicem funkcije (`init_hyp_rom`).

Tabela 2: Elementarni koti za CORDIC v krožnem koordinatnem sistemu.

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

Naloge

1. Ustvarite nov projekt z imenom **CORDIC_UNIFIED** (imenik v predlogi) v katerega dodajte VHDL paketno datoteko **cordic_unified_pkg.vhd**, ki vsebuje deklaracije komponent, naštevnik tipov in funkcij. Dodajte tudi datoteko **cordic_unified.vhd**, kamor vpisujete kodo entitete ter datoteko (**cordic_unified_tb.vhd**), kamor boste programirali različne arhitekture testnih scenarijev. Za začetek v datoteko (**cordic_unified.vhd**) najprej programirajte arhitekturo CORDIC algoritma samo v načinu vrtenja za krožni koordinatni sistem po spodnji sliki.



Slika 5: Podatkovna pot CORDIC algoritma v načinu vrtenja in vektorskem načinu.

Za izvedbo algoritma potrebujemo enostaven krmilni avtomat, ki bo definirал uporabniški vmesnik algoritma. S stališča uporabnika vezje najprej ponastavimo (`nRST = '0'`) – vezje se nahaja v začetnem stanju (`IDLE`), v katerem ima uporabnik možnost nastavljanja začetnih parametrov entitete: koordinatni sistem (`system`), način delovanja (`mode`), začetni kot (`angle_in`) ter splošno začetno točko krajevnega vektorja (`x_in`, `y_in`). Ko uporabnik uspešno vnese parametre, postavi vhod (`start`), nakar krmilni algoritem preide v stanje izračuna (`CALC`). Ko algoritem konča, postavi zastavico (`done`) in preide v končno stanje (`FINISHED`), v katerem ima uporabnik možnost nadaljnje obdelave rezultatov. Za programiranje krmilnega avtomata imamo našeta stanja (`type STATE_TYPE is (IDLE, CALC, FINISHED);`) in spremenljivko (`state`), s katero prehajamo med stanji. Definiramo procesni stavek, v katerem ga bomo programirali kot Moore-ov avtomat končnih stanj.

V procesnem stavku definiramo pomožni spremenljivki (`shifted_x`, `shifted_y`) kot predznačen `standard_logic_vector` tip (`signed(WIDTH-1 downto 0)`), ki bosta predstavljali izhod vzporednega pomikalnika podatkov (ang. barrel shifter) in pa celoštevilsko spremenljivko (`d_i`), ki bo označevala smer približevanja končni vrednosti (+1 – zmanjševanje, -1 – povečevanje).

Znotraj procesnega stavka definiramo osnovni (`if nRST = '0' then ... elsif rising_edge(clk) then ...`) stavek, s katerim bomo krmilili signale v ustreznem stanju. V stanju ponastavitve (`nRST = '0'`) postavimo stanje (`state`) v (`IDLE`) in zberemo zastavico (`done`). V delu avtomata, ki se spreminja na prednji rob signal ure (`clk`) postavimo (`case`) stavek, v katerem za vsako stanje opišemo zahtevano delovanje. V stanju (`IDLE`) zberemo zastavico (`done`) in preverimo, če je uporabnik postavil krmilni signal (`start`). Takrat postavimo začetne vrednosti (`x_in`, `y_in`, `angle_in`) v ustrezne soležne registre (`x_reg`, `y_reg`, `z_reg`) in preidemo v stanje (`CALC`). V stanju (`CALC`) moramo preiti vseh (`SIZE`) iteracij po lokacijah ROM elementa, za kar uporabljamo celoštevilsko (`integer`) spremenljivko (`step`). Poleg nje definiramo še eno spremenljivko iteracij (`i_cnt`), ki bo imela zaenkrat enako vlogo kot (`step`), šele v hiperboličnem koordinatnem sistemu se bo poznala razlika. Bistvo algoritma kodirajte v stanju (`CALC`) po sliki 6. Vzporedni pomikalnik podatkov (blok SHIFTER, Slika 5) izvedete z operatorjem (`shift_right`), ki ustrezeni vektor (`x_reg`, `y_reg`) pomakne desno za trenutno vrednost iteracij (`i_cnt`). V pogojnem stavku (`if/else`) določite smer vrtenja (`d_i`) glede na predznak vektorja (`z_reg`) – smer vrtenja postane -1, ko je predznak vektorja (`z_reg`) negativen, sicer je

1. V novem pogojnem stavku (**if/else**) glede na določeno smer vrtenja izvedete pogojna odštevanja/seštevanja (bloki ADD/SUB, , Slika 5).

Operacije blokov ADD/SUB izvajamo kar z vgrajenimi operatorji (+,-), ki veljajo za predznačena števila (**signed**):

Če je (**d_i = 1**) izvedete odštevanje pomaknjene vrednosti (**shifted_y**) nad registrom (**x_reg**) – rezultat shranite nazaj v (**x_reg**). Podobno nad registrom (**z_reg**) odštejete tekočo vrednost tabele elementarnih kotov (**LUT_CIRC(step)**) in rezultat shranite v register (**z_reg**). Nad registrom (**y_reg**) izvedete prištevanje pomaknjene vrednosti (**shifted_x**) – rezultat shranite nazaj v (**y_reg**).

Če je (**d_i = -1**), izvedete prištevanje pomaknjene vrednosti (**shifted_y**) nad registrom (**x_reg**) – rezultat shranite nazaj v (**x_reg**). Podobno nad registrom (**z_reg**) prištejete tekočo vrednost tabele elementarnih kotov (**LUT_CIRC(step)**) in rezultat shranite v register (**z_reg**). Nad registrom (**y_reg**) izvedete odštevanje pomaknjene vrednosti (**shifted_x**) – rezultat shranite nazaj v (**y_reg**).

V stanju (**CALC**) samo še povečate vrednosti spremenljivk (**step**, **i_cnt**) za 1. Na koncu preverite (**if**), če je (**step**) že prišel do konca tabele elementarnih kotov (**step = SIZE - 1**) – takrat de premaknete v stanje (**FINISHED**).

V stanju (**FINISHED**) pretvorite vrednosti registrov (**x_reg**, **y_reg**, **z_reg**) v ustrezni rezultat (**res_x**, **res_y**, **res_z**) z operatorjem (**std_logic_vector**) in postavite zastavico (**done**) na '1', ter preidete v stanje (**IDLE**).

Nadgradnja v vektorski način

CORDIC algoritem poleg načina vrtenja (ang. rotation mode) obstaja tudi v vektorskem načinu (ang. vectoring mode). Osnovne enačbe so vezane na strojno shemo in so v obeh načinih enake (za krožni koordinatni sistem):

$$\begin{aligned}x[j+1] &= x[j] - \sigma_j 2^{-j} y[j] \\y[j+1] &= y[j] + \sigma_j 2^{-j} x[j] \\z[j+1] &= z[j] - \sigma_j \tan^{-1}(2^{-j})\end{aligned}$$

V načinu vrtenja zmanjšujemo Z register (z_reg) proti 0, v vektorskem načinu pa zmanjšujemo Y register (y_reg) proti 0. Pozor, poleg tega se predznak vrtenja (σ_j) obrne:

$$\sigma_j = \begin{cases} 1 & \text{if } z[j] \geq 0 \\ -1 & \text{if } z[j] < 0 \end{cases} \quad \sigma_j = \begin{cases} 1 & \text{if } y[j] < 0 \\ -1 & \text{if } y[j] \geq 0 \end{cases}$$

Slika 6: Smer vrtenja (σ_j) v načinu vrtenja.

Slika 7: Smer vrtenja (σ_j) v vektorskem načinu.

Dopolnimo izdelan CORDIC algoritem s parametrom načina (`mode`), ki ima v paketni datoteki definiran naštevni tip (`CORDIC_MODE`) katerega vrednosti sta (`ROTATION`, `VECTORING`). Za dopolnitev v vektorski način je treba dopisati pogoj (`if`) stavka, ki je realiziral smer vrtenja (`d_i`) glede na način delovanja (Slika 6, Slika 7): V krožnem načinu smo imeli smer vrtenja (`d_i`) definirano samo glede na predznak registra (`z_reg`). To določitev obdamo s pogojnim stavkom (`if mode = ROTATION then ... end if`). Vektorski način spravimo v drugi del tega pogojnega stavka (`else`); ko je (`y_reg`) negativen, postavimo (`d_i := 1`), ko pa je pozitiven pa (`d_i := -1`).

Če smo pri načinu vrtenja (Slika 8) postavili začetni vektor na $x_{in}=1/K$, $y_{in}=0$ v register z_{in} naložili začetni kot θ , bomo po vseh rotacijah dosegli končni cilj. Če vstavimo ti začetni vrednosti (x_{in}, y_{in}) v enačbi na sliki 9, dobimo v $x_f=\cos(\theta)$, $y_f=\sin(\theta)$, register z_f pa se je zmanjšal proti 0.

Če bi v vektorskem načinu (Slika 9) postavili neko začetno vrednost $x_{in}=x_0$, $y_{in}=y_0$, $z_{in}=z_0$, dobimo po končanih rotacijah »podobno« izražavo kot je zapis krajevnega vektorja $(x_{in} \ y_{in})$ v polarnih koordinatah, zavrnega (seštevanje) za začetni kot z_{in} . Za poseben primer, ko je $z_{in}=0$, algoritem izračuna vrednost obratne tangens funkcije $\text{atan}(y/x)$. V registru x_f se takrat nahaja »podaljšana« (zato beseda podobno) dolžina krajevnega vektorja (množena s K) - izraz za x_f se uporablja v AM demodulaciji pri iskanju ovojnice signala in v spektralni analizi kot pretvornik iz kompleksnega FFT izhoda v spekter moči.

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

Slika 8: Končni cilj v načinu vrtenja
(krožni koordinatni sistem).

$$\begin{aligned}x_f &= K(x_{in}^2 + y_{in}^2)^{1/2} \\y_f &= 0 \\z_f &= z_{in} + \tan^{-1}\left(\frac{y_{in}}{x_{in}}\right)\end{aligned}$$

Slika 9: Končni cilj v vektorskem načinu
(krožni koordinatni sistem).

Način vrtenja v krožnem koordinatnem sistemu omogoča izračun funkcij \sin in \cos , medtem ko v vektorskem načinu omogoča izračun obratne tangens funkcije razmerja dveh števil – delovanje teh funkcij bomo preverili v datoteki testnih vrednosti (`cordic_unified_tb.vhd`). V tej datoteki je že definirana entiteta (`CORDIC_tb`) z generičnima parametroma (`WIDTH`), ki določa širino zapisa fiksne vejice in (`SIZE`), ki določa dolžino tabele elementarnih kotov:

```

entity CORDIC_tb is
    generic (
        WIDTH : integer := 32;
        SIZE : integer := 32
    );
end CORDIC_tb;

```

V datoteki (cordic_unified_tb.vhd) dodajte novo arhitekturo (ndv_rotation) entitete (CORDIC_tb) v kateri definirate časovno (time) konstanto periode signala ure (PERIOD) in jo postavite na (20 ns). Definirajte vse potrebne signale za povezovanje programirane komponente (clk, nRST, start, angle_in, res_x, res_y, res_z, done) in signal tipa (boolean) za ustavitev simulacije (stop_sim), ki ga postavite na (false). Programirano entiteto (entity work.cordic_unified) povežite s povezovalnim stavkom (generic map, port map), pri čemer vhod koordinatnega sistema (system) zaenkrat postavite na (CIRCULAR), način (mode) postavite na (ROTATION), začetno vrednost (x_in) postavite na vrednost obratne konstante 1/K (definirana v paketni datoteki, K_CIRC_INV) in jo z uporabo funkcije (Conv2fixedPt) pretvorite v vektor dolžine (WIDTH). Začetno vrednost (y_in) z agregatom (others) postavite na 0. Ostale signale povežite na soimenske signale v komponenti. Generator ure lahko definirate v ločenem procesu, ki teče znotraj (while ... loop) in se konča, ko postane (stop_sim = '1'). V simulacijskem procesu definirajte realne (real) spremenljivke (current_angle, s_val, c_val) in realno konstanto koraka kota (STEP), ki jo postavite na vrednost 22.5° v radianih (0.392699).

Najprej izvedite ponastavitev sistema (nRST <= '0'), nato počakajte (100 ns), nakar aktivirate sistem ter počakate še eno periodo (PERIOD). Spremenljivko (current_angle) postavite na vrednost -90° (-1.570796) in v zanki (while ... loop) preletite vse kote do 90° (1.570797) s korakom (STEP). Ob vsaki iteraciji zanke nastavite nov kot (angle_in) tako, da spremenljivko (current_angle) pretvorite z uporabo funkcije (Conv2fixedPt) v vektor širine (WIDTH). Aktivirate izračun (start <= '1';), počakate naslednji rob ure (rising_edge(clk) in deaktivirate izračun (start <= '0';). Ko se izračun konča (wait until done = '1'), počakate še periodo (PERIOD) in pretvorite dobljena rezultata v registrih (res_x, res_y) v spremenljivki (c_val, s_val) z uporabo funkcije (Conv2real) in (to_integer(signed(...))). V poročilu (report) izpišite izračunano vrednost (real'image(c_val)), idealno matematično vrednost (real'image(cos(current_angle))) in napako

izračuna kot razliko med njima. Podobno za sinusno funkcijo. Postavite tudi trditev (**assert**), ki določa največjo dovoljeno napako izračuna, sicer javi napako (**error**).

Po končani zanki postavite signal za ustavitev procesa ure (`stop_sim <= true`) in končate.

Za testiranje izračuna obratne tangens funkcije razmerja v datoteki (`cordic_unified_tb.vhd`) dodajte novo arhitekturo (`ndv_arctan`), katere struktura je zelo podobna prejšnji: podobno kot prej definirate časovno (**time**) konstanto periode signala ure (`PERIOD`) in jo postavite na (**20 ns**). Definirate vse potrebne signale za povezovanje programirane komponente (`clk`, `nRST`, `start`, `angle_in`, `res_x`, `res_y`, `res_z`, `done`) in signal tipa (**boolean**) za ustavitev simulacije (`stop_sim`), ki ga postavite na (`false`). Programirano entiteto (**entity** `work.cordic_unified`) povežite s povezovalnim stavkom (**generic map, port map**), pri čemer vhod koordinatnega sistema (`system`) zaenkrat postavite na (`CIRCULAR`), način (`mode`) postavite na (`VECTURING`), ob vzpostavitvi komponente vse tri začetne vrednosti (`x_in`, `y_in`, `z_in`) z agregatom (**others**) postavite na 0. V štirih točkah (`x_in`, `y_in`) preverite rezultat obratne tangens funkcije:

$(x_{in} = 1, y_{in} = 1) \rightarrow 45^\circ$, $(x_{in} = 0.5, y_{in} = 0.866) \rightarrow 60^\circ$, $(x_{in} = 1, y_{in} = 0) \rightarrow 0^\circ$, $(x_{in} = 0.5, y_{in} = -0.5) \rightarrow -45^\circ$. Za preverjanje pravilnosti izračuna uporabite funkcijo v knjižnici (`ieee.math_real.arctan`).

Uvedba hiperboličnega koordinatnega sistema

Poleg krožnega koordinatnega sistema (ang. circular) CORDIC v t.im. poenoteni (ang. unified) obliki lahko uporabljamo še v linearnem (ang. linear) ali hiperboličnem (ang. hyperbolic) koordinatnem sistemu. Če želimo izračunati vrednosti funkcij *sinh* in *cosh*, ohranimo vse do sedaj programirane elemente. Uvedemo še zadnji parameter (*system*), ki ga bomo v primeru krožnega sistema postavili na (*CIRCULAR*), v primeru hiperboličnega sistema pa na (*HYPERBOLIC*). Tabela elementarnih kotov se v hiperboličnem sistemu računa z uporabo (*ARCTANH*) funkcije, ki smo jo že opisali – zdaj definiramo dodaten ROM element (*LUT_HYP*) in ga s klicem funkcije (*init_hyp_rom*) postavimo na začetne vrednosti podobno kot prej v krožnem sistemu. Vrsta v hiperboličnem sistemu *ni* konvergentna, zato zahteva začetek pri indeksu (*i_cnt*) 1, ne 0 (kot v krožnem sistemu)! Konvergenco vrste dosegamo s ponovitvijo (ang. repetition) korakov – točneje vsak korak v zaporedju $i_{n+1} = 3 \times i_n + 1$, kar pomeni 4, 13, 40, 121, 364 korak... moramo ponoviti dvakrat. Kot indikator ponavljanja uvedemo spremenljivko (*i_cnt_repeated*) tipa (*std_logic*), katere vrednost bo '1', ko bo ponovitev treba izvesti, sicer '0'.

V glavnem avtomatu v stanju (*IDLE*) dodajte samo začetno vrednost indikatorja ponovitev (*i_cnt_repeated* \leq '0'). V stanju (*CALC*) morate vzporedni pomikalnik podatkov (*SHIFTER*, Slika 5) nadgraditi s pogojnimi stavkom (*if*) – prejšnji primer velja za pogoj (*system* = *CIRCULAR*) – dodamo (*else*) pogoj, kjer indeks povečamo za 1 (*shifted_x* := *shift_right*(*x_reg*, *i_cnt* + 1)) – in tako enostavno dosežemo, da tečejo iteracije pomikov (*i_cnt*) v hiperboličnem načinu od 1 dalje, v krožnem pa od 0.

Pogojni stavek za interpretacijo smeri vrtenja (*d_i*) moramo dopolniti z dodatnima pogojnima stavkoma – znotraj (*if*) in znotraj (*else*) definiramo dodaten pogojni stavek, ki loči koordinatna sistema: Za pogoj (*system* = *CIRCULAR*) ostanejo stvari enake kot prej, sicer (*system* = *HYPERBOLIC*) pa definiramo preračun registrov v hiperboličnem načinu (Slika 5):

Če je (*d_i* = 1) izvedete prištevanje pomaknjene vrednosti (*shifted_y*) nad registrom (*x_reg*) – rezultat shranite nazaj v (*x_reg*). Podobno nad registrom (*z_reg*) odštejete tekočo vrednost tabele elementarnih kotov (*LUT_HYP*(*i_cnt*)) in rezultat shranite v register (*z_reg*). Nad registrom (*y_reg*) izvedete prištevanje pomaknjene vrednosti (*shifted_x*) – rezultat shranite nazaj v (*y_reg*) - enako kot v krožnem sistemu.

Če je ($d_i = -1$), izvedete odštevanje pomaknjene vrednosti ($shifted_y$) nad registrom (x_{reg}) – rezultat shranite nazaj v (x_{reg}). Podobno nad registrom (z_{reg}) prištejete tekočo vrednost tabele elementarnih kotov ($LUT_HYP(i_cnt)$) in rezultat shranite v register (z_{reg}). Nad registrom (y_{reg}) izvedete odštevanje pomaknjene vrednosti ($shifted_x$) – rezultat shranite nazaj v (y_{reg}) - enako kot v krožnem sistemu. Dodati moramo še logiko za pogojno podvojevanje iteracij. Definiramo pogojni stavek, ki se izvede, če je ($system = HYPERBOLIC$) in če je izid funkcije v paketni datoteki ($is_hyp_repeat(i_cnt + 1)$) in če je indikator ponovitve '0' ($i_cnt_repeated = '0'$). Če so vsi trije pogoji izpolnjeni, postavimo indikator ($i_cnt_repeated \leq '1'$). Če zgornji pogoj ni res – recimo, da smo v krožnem sistemu, ali da se je ponovitev koraka že zgodila, potem ponastavimo indikator ($i_cnt_repeated \leq '0'$) in enako kot prej povečamo števec iteracij ($cnt \leq i_cnt + 1$). Končni del stanja (CALC) ostaja enak - podobno kot v krožnem povečamo korak ($step \leq step + 1$) in preverimo število iteracij ($step = SIZE - 1$).

Za testiranje hiperboličnega koordinatnega sistema v datoteki (`cordic_unified_tb.vhd`) dodajte novo arhitekturo (`ln_exp_sqrt_test`), katere struktura je zelo podobna prejšnji: podobno kot prej definirate časovno (`time`) konstanto periode signala ure (`PERIOD`) in jo postavite na (`20 ns`). Definirate vse potrebne signale za povezovanje programirane komponente (`clk`, `nRST`, `start`, `angle_in`, `res_x`, `res_y`, `res_z`, `done`) in signal tipa (`boolean`) za ustavitev simulacije (`stop_sim`), ki ga postavite na (`false`). Programirano entiteto (`entity work.cordic_unified`) povežite s povezovalnim stavkom (`generic map, port map`), pri čemer vhod koordinatnega sistema (`system`) nastavite hiperbolični koordinatni sistem (`HYPERBOLIC`), način (`mode`) bomo spreminjali glede na vrsto testa, ob vzpostavitvi komponente vse tri začetne vrednosti (`x_in`, `y_in`, `z_in`) z agregatom (`others`) postavite na 0.

V prvem testu bomo izračunali vrednost $e^{0.75}$: Ustvarite simulacijski proces in v njem definirajte spremenljivke tipa (`real`), ki jih bomo rabili za hranjenje rezultatov testa (`actual_val`, `ideal_val`, `error`). Podobno kot prej najprej izvedite ponastavitev sistema (`nRST \leq '0'`), nato počakate (`100 ns`), nakar aktivirate sistem ter počakate še eno periodo (`PERIOD`). Za ta namen moramo najprej nastaviti način (`mode`) na (`ROTATION`), (`x_in`, `y_in`) postaviti na obratno vrednost konstante K za hiperbolični sistem (`K_HYP_INV`) in ju s funkcijo (`Conv2fixedPt`)

pretvoriti v (*signed*) vektor širine (*WIDTH*). V spremenljivko (*angle_in*) naložimo vrednost (*0.75*), ki jo pretvorimo v zapis s fiksno vejico z uporabo funkcije (*Conv2fixedPt*).

Aktivirate izračun (*start <= '1'*), počakate naslednji rob ure (*rising_edge*(*clk*)) in deaktivirate izračun (*start <= '0'*). Ko se izračun konča (*wait until* *done = '1'*), počakate še periodo (*PERIOD*) in pretvorite dobljeni rezultat v registru (*res_x*) v spremenljivko (*actual_val*) z uporabo funkcije (*Conv2real*) in (*to_integer(signed(...))*). V poročilu (*report*) izpišite izračunano vrednost (*real'image*(*actual_val*)), idealno (*ideal_val*) matematično vrednost (*exp(0.75)*) in napako izračuna kot razliko med njima (*error*), ki je absolutna vrednost razlike (*actual_val - ideal_val*). Postavite tudi trditev (*assert*), ki določa največjo dovoljeno napako izračuna, sicer javi napako (*error*). Izdelani test kopirajte in ponovite za robno vrednost negativnega eksponenta ($e^{-0.5}$).

V drugem testu bomo izračunali vrednost naravnega logaritma $\ln(2)$. V vektorskem načinu, hiperboličnem koordinatnem sistemu pravzaprav izvajamo izračun po sliki

$$z_n = z_0 + atanh\left(\frac{y_{in}}{x_{in}}\right)$$

Slika 10: Iteracije z v vektorskem načinu, hiperbolični koordinatni sistem.

$$\ln(x) = 2 \cdot atanh\left(\frac{x-1}{x+1}\right)$$

Slika 11: Identiteta za izračun naravnega logaritma.

Če želimo izračunati $\ln(2)$, moramo na vhode nastaviti $x_{in} = 3$ in $y_{in} = 1$, začetni kot postavimo na ($z_{in} = 0$). Slika 11 kaže, da se bo rezultat nahajal v registru z (*res_z*) – za pravilni rezultat morate dobljeno vrednost množiti z 2, kar lahko storite s pomikom levo (*shift_left(signed(res_z), 1)*).

Za izračun naravnega logaritma potrebujemo še vmesno spremenljivko (*val_w*) tipa (*real*), ki jo definirate pri ostalih spremenljivkah v procesu. Z njo bomo tvorili prištevanje/odštevanje 1 (Slika 11) za tvorbo argumentov (x_{in} , y_{in}). Najprej nastavite način (*mode*) na (*VECTURING*), (x_{in}) postavite na (*val_w + 1.0*), (y_{in}) postavite na (*val_w - 1.0*) in ju s funkcijo (*Conv2fixedPt*) pretvorite v (*signed*) vektor širine

(`WIDTH`). Podobno kot prej naredimo (`start`) pulz in počakamo na zastavico (`done`). V poročilu (`report`) izpišite izračunano vrednost (`real' image(actual_val)`), idealno (`ideal_val`) matematično vrednost (`log(val_w)`; **to ni napaka - dejansko je log v knjižnici kot naravni logaritem!**) in napako izračuna kot razliko med njima (`error`), ki je absolutna vrednost razlike (`actual_val - ideal_val`). Postavite tudi trditev (`assert`), ki določa največjo dovoljeno napako izračuna, sicer javi napako (`error`). Izdelani test kopirajte in ponovite za robno vrednost, ki je zelo blizu 1 (`ln(1.05)`).

V tretjem testu bomo izračunali vrednost kvadratnega korena $\sqrt[3]{2}$. V vektorskem načinu, hiperboličnem koordinatnem sistemu pravzaprav izvajamo izračun po sliki

$$x_n = K_{hyp} \cdot \sqrt{x_{in}^2 - y_{in}^2}$$


$$\begin{aligned} x_{in} &= w + 0.25 \\ y_{in} &= w - 0.25 \\ (w + 0.25)^2 - (w - 0.25)^2 &= \\ (w^2 + 0.5w + 0.0625) - (w^2 - 0.5w + 0.0625) &= w \end{aligned}$$

Slika 12: Iteracije x v vektorskem načinu, hiperbolični koordinatni sistem.

Slika 13: Identiteta za izračun kvadratnega korena.

Za izračun $\sqrt[3]{w}$ moramo vhoda x_{in} in y_{in} pripraviti tako, da velja $x_{in}^2 - y_{in}^2 = w$, kar dosežemo z izbiro premaknjenih x_{in} in y_{in} (Slika 13). Zakaj je odmik x_{in} in y_{in} ravno 0.25? Gre za preprosto algebro razlike kvadratov (Slika 13), s katero dobimo koren števila $\sqrt[3]{w}$, ne da bi za to potrebovali kakršnokoli drugo pred-operacijo razen dveh seštevanj.

Če želimo izračunati $\sqrt[3]{2}$, moramo na vhode nastaviti $x_{in} = 2.25$ in $y_{in} = 1.75$ - s tem da vrednosti skaliramo z obratno vrednostjo konstante v hiperboličnem koordinatnem sistemu (`K_HYP_INV`, Slika 12), začetni kot postavimo na ($z_{in} = 0$). Slika 12 kaže, da se bo rezultat nahajal v registru `z(res_x)`. Za izračun kvadratnega korena uporabimo vmesno spremenljivko (`val_w`). Z njo bomo tvorili prištevanje/odštevanje 0.25

(Slika 11) za tvorbo argumentov (x_{in} , y_{in}). Najprej nastavite način (mode) na (VECTURING), (x_{in}) postavite na $((val_w + 0.25) * K_HYP_INV)$, vrednost (y_{in}) postavite na $((val_w - 0.25) * K_HYP_INV)$ in ju s funkcijo (Conv2fixedPt) pretvorite v (signed) vektor širine (WIDTH). Podobno kot prej naredimo (start) pulz in počakamo na zastavico (done). V poročilu (report) izpišite izračunano vrednost (real'  (actual_val)), idealno (ideal_val) matematično vrednost (sqrt (val_w) ;) in napako izračuna kot razliko med njima (error), ki je absolutna vrednost razlike (actual_val - ideal_val). Postavite tudi trditev (assert), ki določa največjo dovoljeno napako izračuna, sicer javi napako (error). Izdelani test kopirajte in ponovite za robno vrednost, ki je zelo blizu 0 $\sqrt{(0.1)}$.

Zadnji test bo prikazal večkratni prehod in ponovno uporabo (ang. re-entrant use) CORDIC algoritma v različnih načinih in koordinatnih sistemih: Izračunali bomo splošno potenco števila x^y - CORDIC ne zna neposredno potencirati, zato moramo operacijo moramo pretvoriti v obliko, ki uporablja naravni logaritem in eksponentno funkcijo.

$$x^y = e^{y \cdot \ln(x)}$$

Slika 14: Identiteta za izračun potence števila

Potenciranje razdelimo v dva prehoda skozi isto strojno opremo:

1. Prvi prehod: Vektorski hiperbolični način \rightarrow izračunamo $\ln(x)$.
2. Množenje: Rezultat logaritma množimo z eksponentom y .
3. Drugi prehod: Rotacijski hiperbolični način \rightarrow izračunamo e^{produkt} .

Množenje sicer lahko realiziramo v CORDIC (zahteva tudi linearni koordinatni sistem), vendar je počasno, saj je sekvenčno (traja `SIZE` ciklov). V našem primeru za množenje zapisa v fiksni vejici (Q3.29 – najvišji celi del je 3, sledi 29 decimalnih mest) raje uporabimo inferenco (operator `*`), ki se sintetizira v kombinacijski strojni množilnik, realiziran s sysDSP bloki.

Za testiranje potenciranja v datoteki (`cordic_unified_tb.vhd`) dodajte novo arhitekturo (`pwr_test`), katere struktura je zelo podobna prejšnjim: podobno kot prej definirate časovno (`time`) konstanto periode signala ure (`PERIOD`) in jo postavite na (`20 ns`). Definirate vse potrebne signale za povezovanje programirane komponente (`clk`, `nRST`, `start`, `angle_in`, `res_x`, `res_y`, `res_z`, `done`, `stop_sim`) in signale za fiksni množilnik (`mul_a`, `mul_b`, `mul_res`) tipa (`std_logic_vector`) širine (`WIDTH`). Dodajte še vmesni signal (`ln_full_fixed`) tipa (`std_logic_vector`) širine (`WIDTH`). Programirano entiteto (`entity work.cordic_unified`) povežite s povezovalnim stavkom (`generic map, port map`), pri čemer bomo vhoda koordinatnega sistema (`system`) in način (`mode`) spreminjali glede na trenutno iteracijo. Programirano entiteto množilnika fiksne vejice (`entity work.Q3_29_multiplier_comb`) povežite s povezovalnim stavkom (`generic map, port map`), pri čemer sta operanda (`a_in => ln_full_fixed`, `b_in => mul_b`), produkt pa vezite na (`res_out => mul_res`). Podobno kot prej definirajte proces za tvorbo signala ure in simulacijski proces. V njem definirajte spremenljivke tipa (`real`), ki jih bomo rabili za hranjenje rezultatov testa (`v_base`, `v_exp`, `v_actual`, `v_ideal`, `v_error`) ter spremenljivko (`l`) tipa (`line`). Podobno kot prej izvedete ponastavitev. V prvi ob vzpostavitvi komponente vse tri začetne vrednosti (`x_in`, `y_in`, `z_in`) z agregatom (`others`) postavite na 0.

Za izračun prve faze potenciranja - naravni logaritem - najprej nastavite način (`mode`) na (`VECTURING`), sistem (`system`) na (`HYPERBOLIC`), (`x_in`) postavite na (`(v_base + 1.0) * K_HYP_INV`), (`y_in`) postavite na (`(v_base - 1.0) * K_HYP_INV`) in ju s funkcijo (`Conv2fixedPt`) pretvorite v (`signed`) vektor širine (`WIDTH`). Parameter začetnega kota (`angle_in`) z agregatom (`others`) postavite na 0. Podobno kot prej naredimo (`start`) pulz in počakamo na zastavico (`done`). Rezultat se pojavi v registru (`res_z`). Za preverjanje morate rezultat (`res_z`) pretvoriti v realno število (`Conv2real(to_integer(signed ...)`) in množiti z 2. V poročilu (`report`) izpišite izračunano vrednost (`real'image(actual_val)`), idealno (`ideal_val`) matematično vrednost (`log(v_base)`) in napako izračuna kot razliko med njima (`v_error`), ki je absolutna vrednost razlike (`actual_val - ideal_val`). Postavite tudi trditev (`assert`), ki določa največjo dovoljeno napako izračuna,

sicer javi napako (**error**).

V vmesni fazi (množenje) moramo register (`res_z`) popraviti, tako da ga strojno množimo z 2^{-z} ukazom (`std_logic_vector(shift_left(signed(res_z), 1));`) izvedemo pomik levo eno mesto in rezultat shranimo v vektor (`ln_full_fixed`). Počakamo periodo signala ure (`PERIOD`) in preverimo rezultat množenja, tako da izhod množilnika (`mul_res`) pretvorimo v fiksno vejico z uporabo funkcije (`Conv2real`) in rezultat shranimo v (`v_actual`). Izračunamo vmesni rezultat za preverjanje (`v_ideal`) z uporabo funkcij matematične knjižnice (`v_exp * log(v_base)`) in napako (`v_error`) kot absolutno vrednost razlike (`abs(v_actual - v_ideal)`). Izpišemo poročilo (**report**), v katerem izpišemo vse tri spremenljivke (`v_actual`, `v_ideal`, `v_error`). Postavite tudi trditev (**assert**), ki določa največjo dovoljeno napako izračuna, sicer javi napako (**error**).

V drugi fazi – eksponent vmesnega rezultata - najprej nastavite način (`mode`) na (`ROTATION`), sistem (`system`) na (`HYPERBOLIC`), (`x_in`, `y_in`) postavite na (`K_HYP_INV`) in ju s funkcijo (`Conv2fixedPt`) pretvorite v (`signed`) vektor širine (`WIDTH`). Parameter začetnega kota (`angle_in`) postavite na izhod množilnika (`mul_res`). Podobno kot prej naredite (`start`) pulz in počakate na zastavico (`done`). Rezultat se pojavi v registru (`res_x`). Za preverjanje morate rezultat (`res_x`) pretvoriti v realno število (`Conv2real(to_integer(signed ...))`). V poročilu (**report**) izpišite izračunano vrednost (`real'image(actual_val)`), idealno (`ideal_val`) matematično vrednost (`v_base ** v_exp`) in napako izračuna kot razliko med njima (`v_error`), ki je absolutna vrednost razlike (`actual_val - ideal_val`). Postavite tudi trditev (**assert**), ki določa največjo dovoljeno napako izračuna, sicer javi napako (**error**).

Na tem mestu velja omeniti razliko med izvedbami: V našem primeru smo postavili oba registra (`x_in`, `y_in`) na vrednost (`K_HYP_INV`), zato se po končanih rotacijah v registrih (`x_in`, `y_in`) nahaja izračunani eksponent (e^a) – zato upoštevamo ali (`x_in`) ali (`y_in`), ne oba, sicer dobimo dvojno vrednost eksponenta. Druga možnost bi bila, da bi samo enega (recimo `x_in`) postavili na vrednost (`K_HYP_INV`), (`y_in`) pa na 0 – takrat pride v registru (`x_in = cosh(a)`), v registru (`y_in = sinh(a)`), Po Eulerjevi identiteti ($\cosh(a) + \sinh(a) = e^a$) moramo sešteti oba, da dobimo pravilni eksponent (e^a).

Na koncu simulacijskega procesa postavite zastavico (`stop_sim <= true`), da javite procesu za tvorbo ure, naj se ustavi.

Operacije poenotenega CORDIC algoritma

Zgodovina poenotenega **CORDIC** algoritma je zgodba o tem, kako so inženirji v dobi pred zmogljivimi procesorji iskali načine za izvajanje kompleksne matematike s pomočjo le najosnovnejših digitalnih operacij: premikanja bitov (shift) in seštevanja (add).

Tukaj je pregled ključnih mejnikov, ki so privedli do algoritma, ki smo ga predstavili:

1. 1959: Jack Volder in rojstvo algoritma

Algoritem CORDIC (**CO**ordinate **R**otation **DI**gital Computer) je leta 1959 izumil **Jack E. Volder**. Volder je delal pri podjetju Convair, kjer je potreboval rešitev za zamenjavo analognega navigacijskega računalnika v letalu B-58 Hustler z digitalnim.

Njegov prvotni algoritem je bil omejen le na **krožni koordinatni sistem** (trigonometrija). Omogočal je izračun sinusa, kosinusa in obratnega tangensa, kar je bilo ključno za rotacijo vektorjev v navigaciji. Njegova genialna ideja je bila, da poljuben kot razstavi na vsoto vnaprej določenih "osnovnih" kotov, za katere je tangens potenca števila 2.

2. 1971: J.S. Walther in unifikacija

Čeprav je bil Volderjev algoritem uporaben, je bil omejen. Preboj se je zgodil leta 1971, ko je **John Stephen Walther** pri podjetju Hewlett-Packard (HP) objavil članek, v katerem je pokazal, da lahko CORDIC razširimo na **hiperbolične** in **linearne** funkcije. Walther je uvedel parameter ($m = mode$), ki določa koordinatni sistem oz. geometrijo:

m=1: Krožni (\sin , \cos)	m=0: Linearni (množenje/deljenje)	m=-1: Hiperbolični ($\ln(x)$, e^x , \sqrt{x})
---------------------------------	-----------------------------------	--

Tako je CORDIC postal poenoten – eno samo strojno vezje je nenadoma računalo skoraj vse elementarne funkcije. Večino teh smo spoznali, oz. prikazali princip delovanja. V spodnjih tabelah (Tabela 3, Tabela 4) najdete še nekatere operacije, ki bi jih lahko realizirali s poenotanim CORDIC algoritmom.

Tabela 3: Enokoračne operacije poenotenega CORDIC algoritma.

Operacija	Sistem	Način	x _{in}	y _{in}	z _{in}	Končni x	Končni y	Končni z	Opomba
Sin, cos	CIRCULAR	ROTATION	$(K_{\text{CIRC}})^{-1}$	0	α	$\cos(\alpha)$	$\sin(\alpha)$	0	
arctg	CIRCULAR	VECTURING	1	y	0	$\sqrt{1+y^2}$	0	$\arctg(\frac{y}{x})$	
arcctg	CIRCULAR	VECTURING	1	a	$\pi/2$				$\text{arcctg}(x) = \frac{\pi}{2} - \arctg(\frac{x}{1})$
množenje	LINEAR	ROTATION	x	0	z	x	$x \cdot z$	0	
deljenje	LINEAR	VECTURING	x	y	0	x	0	y/x	
sinh, cosh	HYPERBOLIC	ROTATION	$(K_{\text{HYP}})^{-1}$	0	α	$\cosh(\alpha)$	$\sinh(\alpha)$	0	
e^a	HYPERBOLIC	ROTATION	K_{HYP}^{-1}	K_{HYP}^{-1}	a	e^a	e^a	0	
ln	HYPERBOLIC	VECTURING	$v +$	$v -$	0	\sqrt{v}	0	$\frac{1}{2} \ln(v)$	
sqrt	HYPERBOLIC	VECTURING	$v +$	$v -$	0	\sqrt{v}	0	...	

Komentar glede linearnega geometrijskega sistema:

V sodobnih vezjih FPGA (npr. Lattice ECP5) in procesorjih linearni način CORDIC (za množenje in deljenje) vse bolj izpodrivajo namenske strojne enote. Razlogov za to je več, vsi pa so povezani z razvojem tehnologije izdelave čipov:

CORDIC je po naravi **iterativen algoritem**. Za 32-bitno natančnost potrebuje CORDIC do 32 urinih ciklov (odvisno od konvergence operacije), da pride do rezultata. Sodobni procesorji in FPGA imajo vgrajene **DSP bloke** (npr. matrični množilnik), ki izvedejo množenje v **enem samem urinem ciklu**, kot smo spoznali pri tem predmetu – časovna zahtevnost O (ang. order) je torej $O(n)$, medtem ko je pri strojnem množilniku reda $O(1)$ ali $O(\log n)$ z uporabo drevesnih struktur (npr. Wallace/Dadda drevesnih seštevalnikov in prekodiranja v višje baze - Booth). Danes ima Lattice ECP5-45F na stotine **DSP rezin** (DSP slices), ki so fizično optimizirane na čipu (hard-IP), kar pomeni, da zasedejo

manj prostora kot CORDIC, implementiran z uporabo splošne logike (LUT), ob tem porabijo bistveno manj energije in dosežejo veliko višje hitrosti delovanja.

Čeprav je deljenje v CORDIC (linearni vektorski način) elegantno, sodobni sistemi večinoma zahtevajo standard **IEEE-754 za plavajočo vejico**. Namenske enote za deljenje (npr. Newton-Raphsonova metoda ali Goldschmidtov algoritem) so hitrejše in lažje podpirajo posebne primere, kot so NaN (ang. Not a Number), neskončnost in denormalizirana števila. Namenski množilniki omogočajo zelo globok cevovod (ang. multistage pipelining), kar pomeni, da lahko v vsakem urinem ciklu začnemo novo množenje. Pri CORDIC bi za takšno prepustnost (ang. throughput) potreboval ogromno kopij strojne opreme (ang. unrolled implementation), kar bi drastično povečalo porabo logičnih vrat. Kljub premoči namenskih množilnikov, linearni CORDIC še ni povsem izumrl. Na zelo majhnih in poceni FPGA ali ASIC, ki nimajo DSP blokov, je CORDIC še vedno najboljši način za implementacijo deljenja. Z njim lahko dosegamo poljubno natančnost (pustimo ceno za to) - če potrebujemo deljenje na poljubno število bitov (npr. 128-bitno), kjer standardni DSP bloki ne zadoščajo.

Tabela 4: Dvokoračne operacije poenotenega CORDIC algoritma

Končni cilj	Matematična zveza (Identiteta)	1. korak			2. korak		
		op	system	mode	op	system	mode
x^y	$x^y = e^{y \cdot \ln(x)}$	ln(x)	HYPERBOLIC	VECTURING	e^x	HYPERBOLIC	ROTATION
sqrt	$\sqrt{x} = e^{0.5 \cdot \ln(x)}$	ln(x)	HYPERBOLIC	VECTURING	e^x	HYPERBOLIC	ROTATION
log _b	$\log_b(x) = \frac{\ln(x)}{\ln(b)}$	ln(x) ln(b)	HYPERBOLIC	VECTURING	deljenje	LINEAR	VECTURING
tg	$tg(x) = \frac{\sin(x)}{\cos(x)}$	sin(x) cos(x)	CIRCULAR	ROTATION	deljenje	LINEAR	VECTURING
arcctg	$arcctg(x) = \frac{\pi}{2} - arctg\left(\frac{x}{1}\right)$						
Magnituda faza	$R = \sqrt{x^2 + y^2}$ in $\theta = arctg\left(\frac{y}{x}\right)$	arctg	CIRCULAR	VECTURING	/	/	/

Dolžina 3D vektorja	$R = \sqrt{x^2 + y^2 + z^2} = \sqrt{\left(\sqrt{x^2 + y^2}\right)^2 + z^2}$	$\sqrt{x^2 + y^2}$	CIRCULAR	VECTORING	$\sqrt{(\dots)^2 + z^2}$	CIRCULAR	VECTORING
------------------------	---	--------------------	----------	-----------	--------------------------	----------	-----------

3. Zlata doba: Žepni kalkulatorji

Najbolj znana uporaba poenotenega CORDIC algoritma je bila v legendarnem kalkulatorju **HP-35** (1972), ki je bil prvi znanstveni žepni kalkulator na svetu. Pred tem so inženirji uporabljali logaritemska oz. pomična računalna (nem. Rechenschieber/Rechenstab) ali obsežne tabele (npr. Vegove tablice). HP-35 ni imel dovolj pomnilnika za shranjevanje tabel ali moči za razvoj Taylorjevih vrst. Poenoteni CORDIC je zasedel minimalno število logičnih vrat in zagotavljal visoko natančnost. Večina sodobnih kalkulatorjev (vključno s tistimi od Texas Instruments in Casio) še danes uporablja različice Waltherjevega algoritma.

4. Zakaj je CORDIC preživel?

Morda bi mislili, da je v dobi modernih procesorjev CORDIC zastarel, a ni tako:

- **FPGA in ASIC:** V specializiranih čipih (npr. Lattice ECP5) je CORDIC še vedno bolj učinkovit kot namenski množilniki za izračun funkcij, kot je \tan^{-1} , $\ln(x)$
- **Brezžične komunikacije:** Pri obdelavi signalov (SDR, 5G, radarji) se CORDIC uporablja za mešanje signalov in frekvenčno pretvorbo v realnem času.
- **Grafika:** Prvi 3D procesorji so uporabljali CORDIC za transformacije koordinat.

Uporaba CORDIC-a v družini **STM32** (predvsem v serijah **G4, H7 in novih modelih**) je odličen primer, kako se ta klasični algoritem še vedno uporablja za optimizacijo sodobnih vgrajenih sistemov.

STMicroelectronics je v te procesorje vgradil namenski strojni pospeševalnik, imenovan **CORDIC IP block**, ki razbremeni glavno jedro (CPU) pri zahtevnih matematičnih izračunih.

CORDIC algoritem mikrokrmilnikov STM32

1. Zakaj STM32 uporablja CORDIC namesto CPU?

Čeprav imajo procesorji Cortex-M4 ali M7 enoto za plavajočo vejico (FPU), je CORDIC v določenih nalogah boljši:

- **Vzporedno delovanje:** Medtem ko CORDIC računa npr. sinus kota, lahko CPU izvaja druge naloge (npr. komunikacijo preko CAN vodila ali regulacijsko zanko).
- **Varčevanje z energijo:** Strojni CORDIC blok porabi bistveno manj energije na izračun kot splošno namensko jedro, ki mora poganjati celoten nabor instrukcij.
- **Hitrost pri fiksni vrednosti:** CORDIC v STM32 deluje neposredno v zapisu s fiksno vejico v formatu Q1.31 ali Q1.15, kar je idealno za podatke, ki prihajajo neposredno iz ADC (analogno-digitalnih pretvornikov).

2. Glavne aplikacije v STM32

Najpogosteje se CORDIC v STM32 uporablja v naslednjih primerih:

- **Vodenje elektromotorjev (FOC - Field Oriented Control):**

V motorne krmilnike so vgrajene Clarkove in Parkove transformacije. Te zahtevajo nenehno računanje sinusa in kosinusa za pretvorbo med statorskimi in rotorskimi koordinatami. CORDIC to izvede izjemno hitro.

- **Digitalna obdelava signalov (DSP):**

Računanje magnitude in faze signala (pretvarja iz pravokotnih v polarno koordinato), kar je ključno pri filtriranju ali FFT analizi.

- **Zamenjava za math.h:**

Namesto uporabe standardnih knjižnic v C, ki so počasne, programerji uporabljajo `HAL_CORDIC_Calculate`, ki izkoristi strojni blok.

3. Kako deluje STM32 CORDIC blok?

Za razliko od poenotenega CORDIC v tej vaji, ki ga nastavljamo ročno, STM32 CORDIC deluje preko registrov:

- **Config register:** Izberete funkcijo (sin, cos, sinh, ln ..).
- **Input data register:** Zapišete vhodne vrednosti.
- **Output data register:** Takoj (ali po nekaj ciklih) preberete rezultat.

STM32 CORDIC podpira 6 osnovnih funkcij:

<ul style="list-style-type: none">• SIN/COS• PHASE / MAGNITUDE• ATAN2.	<ul style="list-style-type: none">• LN• SQRT• SINH/COSH
---	--

Seznam literature

- [1] Ercegovac, Lang "*Digital arithmetic*", spletno gradivo k knjigi,
http://web.cs.ucla.edu/digital_arithmetic/files/ch11.pdf (obiskano dne 31.12.2025)
- [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*",
https://www.google.com/search?q=https://www.researchgate.net/publication/267417534_Design_and_Analysis_of_Extension-Rotation_CORDIC_Algorithms_based_on_Non-Redundant_Method (obiskano dne 31.12.2025)
- [4] J. S. Walther, "*A unified algorithm for elementary functions*", Spring Joint Computer Conference proceedings, 1971
https://www.ece.ucdavis.edu/~vojin/CLASSES/EPFL/Papers/21-Welther-El_funcn.pdf (obiskano dne 31.12.2025)
- [5] Rick Parris, "Elementary Functions and Calculators", Phillips Exeter Academy
<https://www.scribd.com/document/500142042/calculatorsParris-rev> (obiskano dne 31.12.2025)
- [6] M. J. Flynn, "*The Higher Level Functions (HLFs)*", Stanford University,
<https://web.stanford.edu/class/ee486/doc/lecture18.pdf> (obiskano dne 31.12.2025)
- [7] Young W. Lim "*CORDIC in VHDL (1A)*", 2011,
<https://upload.wikimedia.org/wikiversity/en/0/0d/CORDIC.VHDL.1.A.20111110.pdf> (obiskano dne 31.12.2025)
- [8] 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
<https://ens-lyon.hal.science/ensl-00086894/document> (obiskano dne 31.12.2025)