УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
Трг Доситеја Обрадовића 6, 21000 Нови Сад, Југославија
Деканат: 021 350-413; 021 450-810; Централа: 021 350-122
Рачуноводство: 021 58-220; Студентска служба: 021 350-763
Телефакс: 021 58-133; e-mail: ftndean@uns.ns.ac.yu

PROJEKAT IZ PROJEKTOVANJA SLOŽENIH DIGITALNIH SISTEMA I FUNKCIONALNE VERFIKACIJE HARDVERA

Naziv projekta:

Projektovanje i verifikacija hardverskog akceleratora za “MP3 decoder” algoritam.

Tekst zadatka:

1. Modelovati sistem pomoću VHDL jezika.
2. Implementacija sistema odgovarajućim AXI interfejsima.
3. Pravljenje blok šeme celokupnog sistema u IP Vidado integratoru.
4. Funkcionalnom verifikacijom proveriti ispravnost sistema.

Mentor projekta:

Nikola Kovačević

Projekat izradili:

Miloš Nedeljković, EE 234/2018
Vasilije Batas, EE 190/2018
Almen Brenoli, EE 109/2018

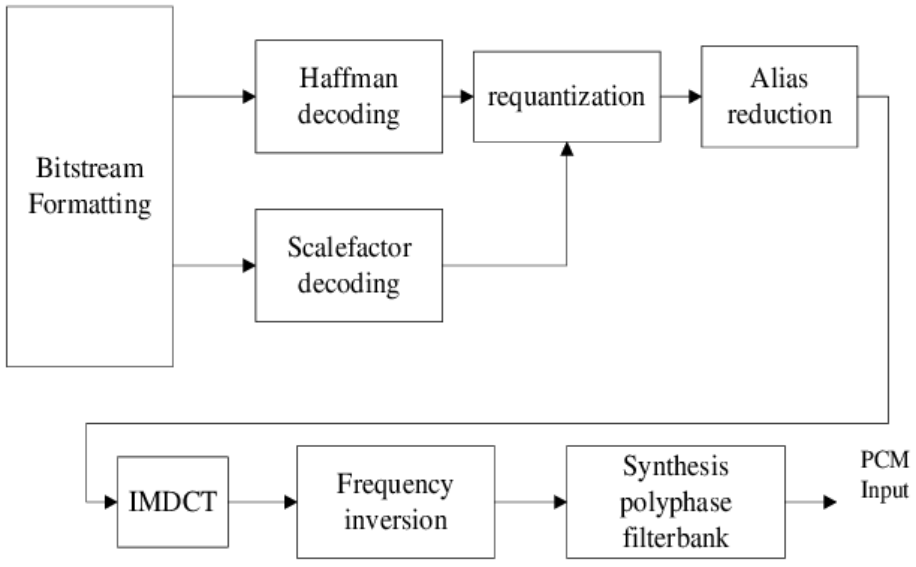
1. Uvod

U današnjem digitalnom dobu, reprodukcija i kompresija zvuka postali su ključni aspekti u mnogim aplikacijama koje se kreću od muzike i filmske industrije do telekomunikacija i multimedijalnih uređaja. Jedan od najznačajnijih formata za kompresiju zvuka je MP3 (MPEG Audio Layer III), koji omogućava efikasno skladištenje i prenos audio sadržaja uz minimalan gubitak kvaliteta. MP3 je postao standardni format za digitalnu distribuciju muzike i drugih zvučnih sadržaja.

MP3 kompresija postiže se kroz eliminaciju suvišnih informacija iz audio signala, čime se smanjuje veličina datoteke bez narušavanja percepcije ljudskog sluha. Proces dekompresije MP3 datoteka je esencijalan kako bi se originalni audio sadržaj povratio u obliku razumljivom ljudskom uhu. Ovaj postupak zahteva izuzetno visoke računske performanse, posebno kada se radi o reprodukciji u realnom vremenu.

Hardverska implementacija *MP3 decoder* algoritma pruža rešenje za efikasno i brzo dekodiranje MP3 datoteka, omogućavajući kvalitetno slušanje muzike ili reprodukciju zvuka na raznim uređajima, uključujući mobilne telefone, audio plejere, televizore i druge multimedijalne platforme.

Na slici 1.1 se može videti blok šema *MP3 decoder* algoritma



Slika 1.1 Blok dijagram MP3 decoder algoritma

Nakon profajliranja početnog *MP3 decoder* algoritma napisanog u jeziku C++ dobijeni su rezultati da je funkcija koja troši najviše resursa tokom izvršavanja algoritma: *Inverese Modified Discrete Cosine Transformation* (IMDCT). Na osnovu dobijenih rezultata odlučeno je da treba taj deo koda hardverski ubrzati kako bi dobili bolje performanse sistema.

Za hardversku implementaciju sistema korišćen je razvojni sistem Zybo Z7-10 i na njegovoj programabilnoj logici se hardverski ubrzava funkcija IMDCT, dok se ostatak koda izvršava softverski na ARM-ovom Cortex A9 procesoru.

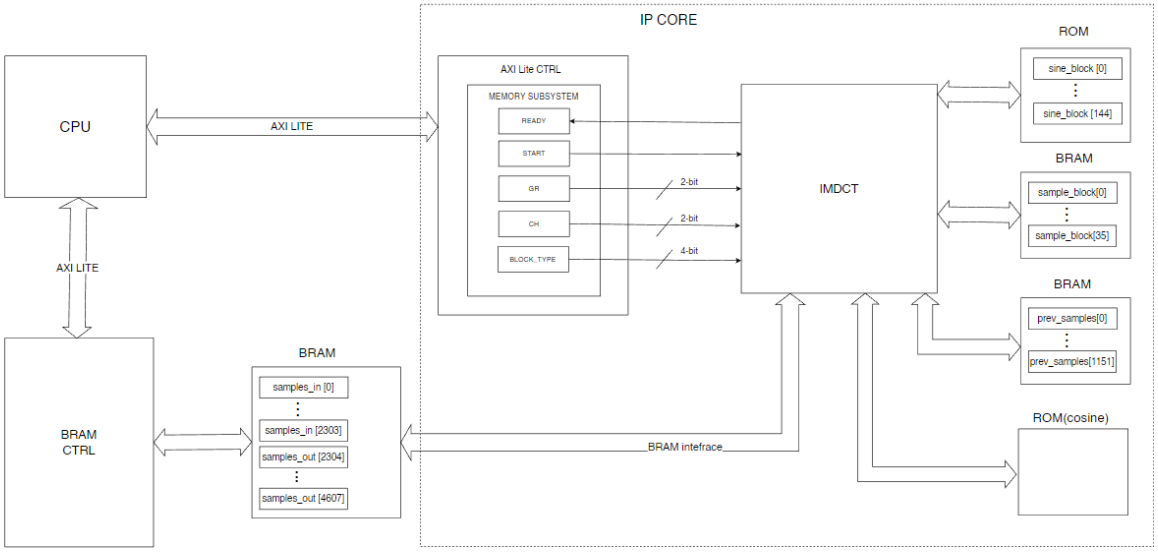
2. Specifikacija sistema

Celokupan sistem sastoji se od AXI Lite kontrolera, memorijskog podsistema, IMDCT funkcije, tri BRAM ćelije, dve ROM ćelije i jednim BRAM kontrolerom.

Preko AXI Lite interfejsa se konfigurišu registri memorijskog podsistema. U registre se upisuju vrednosti neophodne za ispravan radi IP jezgra kao i start signal kojim započinje izvršavanje IMDCT funkcije, dok se informacije o završetku izvršavanje funkcije dobija čitanjem iz statusnog registra READY.

Ulazni semplovi u sistem koji treba da budu obrađeni se šalju preko AXI Lite interfejsa, a kasnije preko BRAM kontrolera se upisuju u BRAM koji se nalazi izvan IP jezgra. Kada sistem završi obradu semplova on ih odmah zatim upisuje na nove lokacije u istom BRAM-u a zatim se oni pomoću istog AXI Lite interfejska čitaju kako bi softverski deo koda mogao nesmetano da nastavi sa izvršavanjem.

Ostale BRAM i ROM ćelije služe za skladištenje promenljivih koje koristi IMDCT funkcija, detaljniji rad IP jezgra biće kasnije razmatran.



Slika 2.1 Blok dizajn sistema

3. Opis funkcionalnosti komponenti sistema

3.1 IMDCT modul

IMDCT (Inverse Modified Discrete Cosine Transform) je ključna komponenta u *MP3 dekoder* algoritmu, odgovorna za transformaciju frekvencijskih podataka nazad u vremenski domen. Ova transformacija omogućava rekonstrukciju audio signala u njegovu originalnu formu.

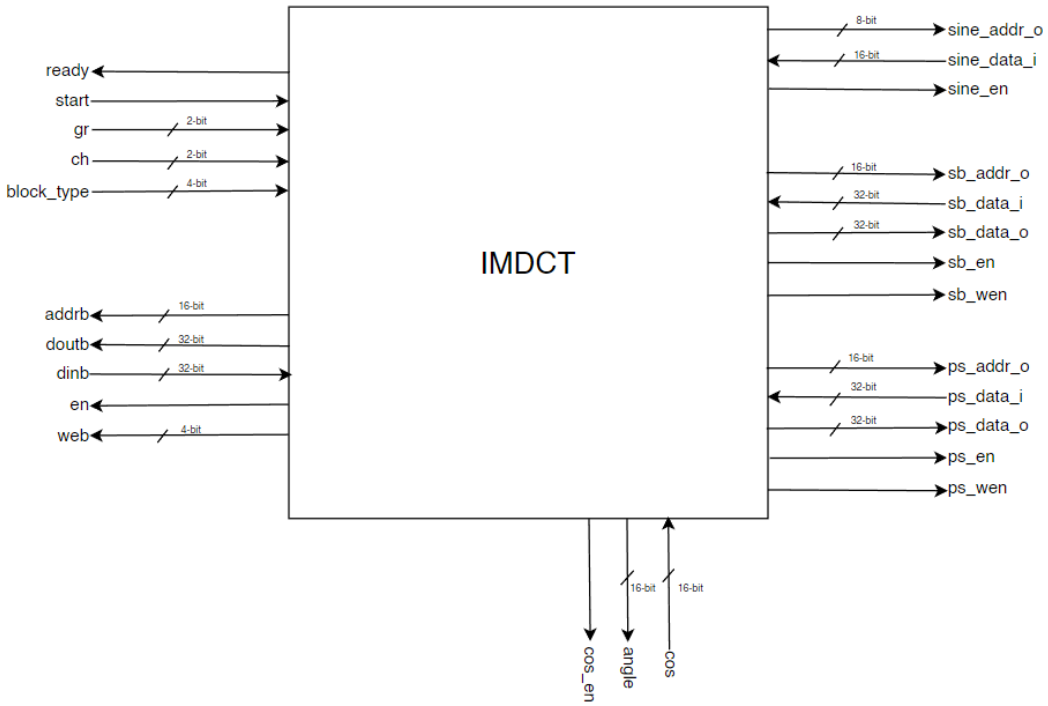
Koristi kako bi se obrnula operacija koju je izvršila MDCT (Modified Discrete Cosine Transform) na ulaznom audio signalu. MDCT je korišćena tokom kompresije u MP3 formatu kako bi se audio signal podelio u manje blokove i transformisao iz vremenskog domena u frekvencijski domen. Svaki blok se zatim kvantizuje i kodira.

Princip rada IMDCT-a je zasnovan na kosinusnoj transformaciji. Blokovi frekvencijskih podataka, dobijeni iz MDCT-a, se kombinuju sa odgovarajućim kosinusnim talasima kako bi se rekonstruisali vremenski signali. Ovaj proces se radi za svaki vremenski trenutak unutar bloka, čime se dobija vremenski signal u vremenskom domenu.

Na primer, zamislimo da imamo blok frekvencijskih podataka koji se odnose na određeni vremenski segment audio signala. IMDCT će kombinovati ove podatke sa odgovarajućim kosinusnim talasima različitih frekvencija kako bi se dobio rekonstruisani audio signal za taj segment. Ovaj rekonstruisani signal će zatim biti spojen sa ostalim segmentima da bi se dobio kompletni audio signal.

U ovom projektu IMDCT modul je projektovan kao jedna celina a promenljive koje se koriste unutar IMDCT funkcije su smestene u BRAM-ove sa kojima IMDCT modul komunicira preko BRAM interfejsa tokom izvršavanja.

Na slici 3.1.1 se mogu videti svi interfejsi koje IMDCT modul koristi kako bi komunicirao sa ostalim modulima unutar IP jezgra. Tu se nalazi interfejs za komunikaciju sa AXI Lite kontrolerom koji dalje komunicira sa softversim delom algoritma. A ostali intefejsi se koriste za komunkaciju sa BRAM i ROM memorijama, koje će biti detaljnije razmatrene u narednim poglavljima.



Slika 3.1.1 imdct modul

3.2 Blok RAM

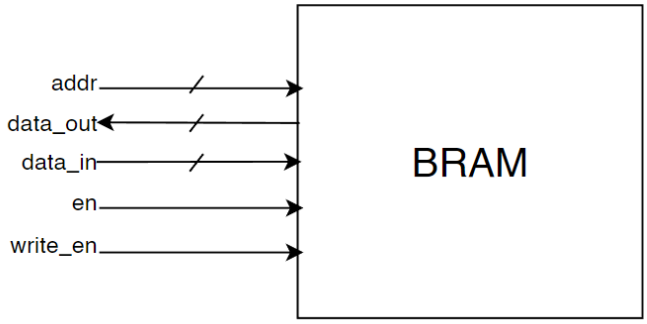
Blok RAM (BRAM) je važan resurs unutar programabilnih logičkih uređaja poput FPGA (Field-Programmable Gate Array) i SoC (System-on-Chip) čipova. To je vrsta brze i lokalne memorije koja omogućava efikasno skladištenje podataka u blizini logičkih blokova, što dovodi do ubrzanja operacija čitanja i pisanja u odnosu na udaljenije i sporije spoljne memorije.

BRAM se sastoji od niza memorijskih ćelija organizovanih u blokove. Svaki blok obično sadrži nekoliko hiljada memorijskih lokacija, a svaka lokacija može skladištiti višebitne podatke. BRAM se koristi za skladištenje različitih vrsta podataka kao što su konfiguracijske informacije, tablice za brzu obradu podataka ili privremeni rezultati izračunavanja.

Interfejsi za upis i čitanje su dva osnovna načina na koja se pristupa podacima unutar BRAM-a. Za operaciju upisa, korisnik šalje adresu memorijske lokacije koju želi da ažurira, kao i podatke koje želi da upiše. Ovi podaci se zatim beleže u odgovarajućoj memorijskoj ćeliji. Operacija čitanja zahteva slanje adrese željene lokacije, nakon čega BRAM vraća podatke sa te adrese. Bitno je napomenuti da signal *enable* treba postaviti na visok logički nivo kada se rade

ove operacije, takođe postoji i signal *write enable* kako bi sistem znao kada da čita iz memorije a kada da upisuje. Primer jednog BRAM-a sa pomenutim interfejsom je dat na slici 3.2.1

U ovom projektu su iskorišćena tri modula BRAM-a, od kojih su dva ručno projektovanja za skladištenje promenljivih *sample_block* i *prev_samples* dok je jednan BRAM modul generisan pomoću Vivado IP integratora za skladištenje ulaznih i izlaznih semplova koje IP jezgro obrađuje. Što se može videti na slici 2.1.

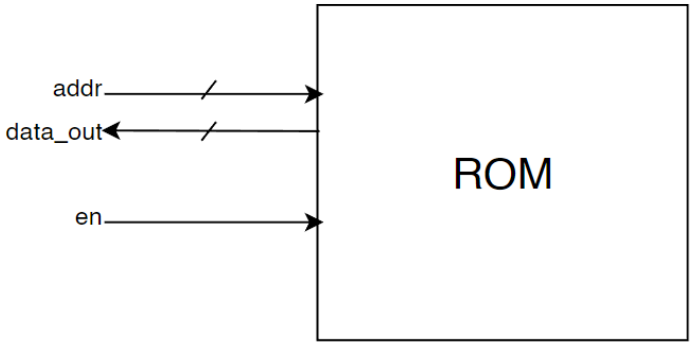


Slika 3.2.1 Blok RAM interfejs

3.3 ROM

Za potrebe ovog projekta ROM je realizovan kao prethodno opisani BRAM moduli sa razlikom u tome što ne postoji interfejs za upis u memoriju već samo za čitanje, a lokacije u memoriji su popunjene inicijalnim vrednostima tako da odgovaraju potrebama IMDCT modula.

U sistemu imamo dva ROM-a koji su iskorišćeni za sladištenje *sine_block* promenljive kao i za skladištenje vrednosti kosinus funckije za svih 360 stepeni sa rezolucijom od jednog stepena. Primer ROM-a je dat na slici 3.3.1.



Slika 3.3.1 ROM interfejs

4. Hardverska implementacija sistema

Hardverska implementacija sistema je urađen pomoću RT metodologije. Prvi korak pri implementaciji bio je da se početni algoritam (slika 4.1) napisan u C++ programskom jeziku, modifikuje tako da bude pogodan za pisanje ASM dijagrama.

```
void mp3::imdct(int gr, int ch)
{
	static bool init = true;
	static float sine_block[4][36];
	float sample_block[36];

	if (init) {
		int i;
		for (i = 0; i < 36; i++)
			sine_block[0][i] = std::sin(PI / 36.0 * (i + 0.5));
		for (i = 0; i < 18; i++)
			sine_block[1][i] = std::sin(PI / 36.0 * (i + 0.5));
		for (; i < 24; i++)
			sine_block[1][i] = 1.0;
		for (; i < 30; i++)
			sine_block[1][i] = std::sin(PI / 12.0 * (i - 18.0 + 0.5));
		for (; i < 36; i++)
			sine_block[1][i] = 0.0;
		for (i = 0; i < 12; i++)
			sine_block[2][i] = std::sin(PI / 12.0 * (i + 0.5));
		for (i = 0; i < 6; i++)
			sine_block[3][i] = 0.0;
		for (; i < 12; i++)
			sine_block[3][i] = std::sin(PI / 12.0 * (i - 6.0 + 0.5));
		for (; i < 18; i++)
			sine_block[3][i] = 1.0;
		for (; i < 36; i++)
			sine_block[3][i] = std::sin(PI / 36.0 * (i + 0.5));
		init = false;
	}

	const int n = block_type[gr][ch] == 2 ? 12 : 36;
	const int half_n = n / 2;
	int sample = 0;

	for (int block = 0; block < 32; block++) {
		for (int win = 0; win < (block_type[gr][ch] == 2 ? 3 : 1); win++) {
			for (int i = 0; i < n; i++) {
				float xi = 0.0;
				for (int k = 0; k < half_n; k++) {
					float s = samples[gr][ch][18 * block + half_n * win + k];
					xi += s * std::cos(PI / (2 * n) * (2 * i + 1 + half_n) * (2 * k + 1));
				}

				/* Windowing samples. */
				sample_block[win * n + i] = xi * sine_block[block_type[gr][ch]][i];
			}
		}

		if (block_type[gr][ch] == 2) {
			float temp_block[36];
			memcpy(temp_block, sample_block, 36 * 4);

			int i = 0;
			for (; i < 6; i++)
				sample_block[i] = 0;
			for (; i < 12; i++)
				sample_block[i] = temp_block[0 + i - 6];
			for (; i < 18; i++)
				sample_block[i] = temp_block[0 + i - 6] + temp_block[12 + i - 12];
			for (; i < 24; i++)
				sample_block[i] = temp_block[12 + i - 12] + temp_block[24 + i - 18];
			for (; i < 30; i++)
				sample_block[i] = temp_block[24 + i - 18];
			for (; i < 36; i++)
				sample_block[i] = 0;
		}

		/* Overlap. */
		for (int i = 0; i < 18; i++) {
			samples[gr][ch][sample + i] = sample_block[i] + prev_samples[ch][block][i];
			prev_samples[ch][block][i] = sample_block[18 + i];
		}
		sample += 18;
	}
}
```

Slika 4.1 IMDCT algoritam

Primećeno je da se početni deo alogoritma označen crvenom bojom na slici 4.1 izvršava uvek isto bez obzira na ulaz, odnosno da daje konstante vrednosti. Na osnovu toga odlučeno je da te konstante mogu biti smeštene u ROM memoriju radi smanjenja resursa potrebnih za njihovo računanje. Za to svrhu smo koristili memoriju opisanu pod tačkom 3.3.

Početni algoritam koristi ugrađenu C++ kosinus funkciju koja na osnovu ugla izraženog u radijanima vraća vrednost kosinusa u *float* vrednostima. Odlučeno je da će za hardversku implementaciju kosinus funkcije biti korišćena još jedna ROM memorija u kojoj će biti smešteni rezultati kosinus funkcije za svih 360 stepeni sa rezolucijom od jednog stepena, a svaki stepen će zauzimati jednu lokaciju u memoriji odnosno imaćemo memoriju od ukupno 360 lokacija. Kako je ranije spomenuto da početna kosinus funkcija radi sa radijanima a memorija sa stepenima, bilo je potrebno da konverovati radijane u stepene pri pisanju novog algoritma sa *goto* naredbama, odnosno samo pomnožiti brojem 57 što se može videti na slici 4.2 podebljanim slovima.

Kako memorija sa kosinus vrednostima sadrži samo 360 lokacija a pri izračunavanju stepena za kosinus dolazi do prekoračenja tog broja bilo je potrebno da sistem čita iz memorije u okviru dozvoljenih adresa, to je urađeno na način da je dodat još jedan registar u sistem *mapped_angle* koji predstavlja mapiran ugao kosinusa na vrednosti od 0 do 360 stepeni. Logika izračunavanja se može videti na slici 4.3 u stanjima K_INC, ANGLES, XI_CAL, na toj slici je predstavljen ASM dijagram sistema.

Dodatna modifikacija koja je odrađena je za BRAM meoriju koja skladišti vrednosti promenljive *sample_blocki.* U početnom algoritmu slika 4.1 promenljiva ima 36 lokacija a u izmenjenom je taj broj proširen na 72 iz razloga smanjenja broja potrebnih modula memorija u sistemu. Na osnovu početnog algoritma se može videti da se početne vrednosti te promenljive izračunavaju na način označen plavom bojom na slici 4.1 a kasnije ukoliko se ulazi u *if* petlju označenu zelenom bojom te vrednosti se menjanju. Odnosno zavise od vrednosti promenljive *block_type.* Zbog tog razloga odlučeno je da se početne vrednosti skladište na gornjih 36 lokacija a u slučaju da je vrednost *block_type* registra jednaka sa dva, onda se dalje proračunate vrenosti skladište na donjim 36 lokacijama u memoriji, na slici 4.2 je ta modifikacija označena zelenom bojom.

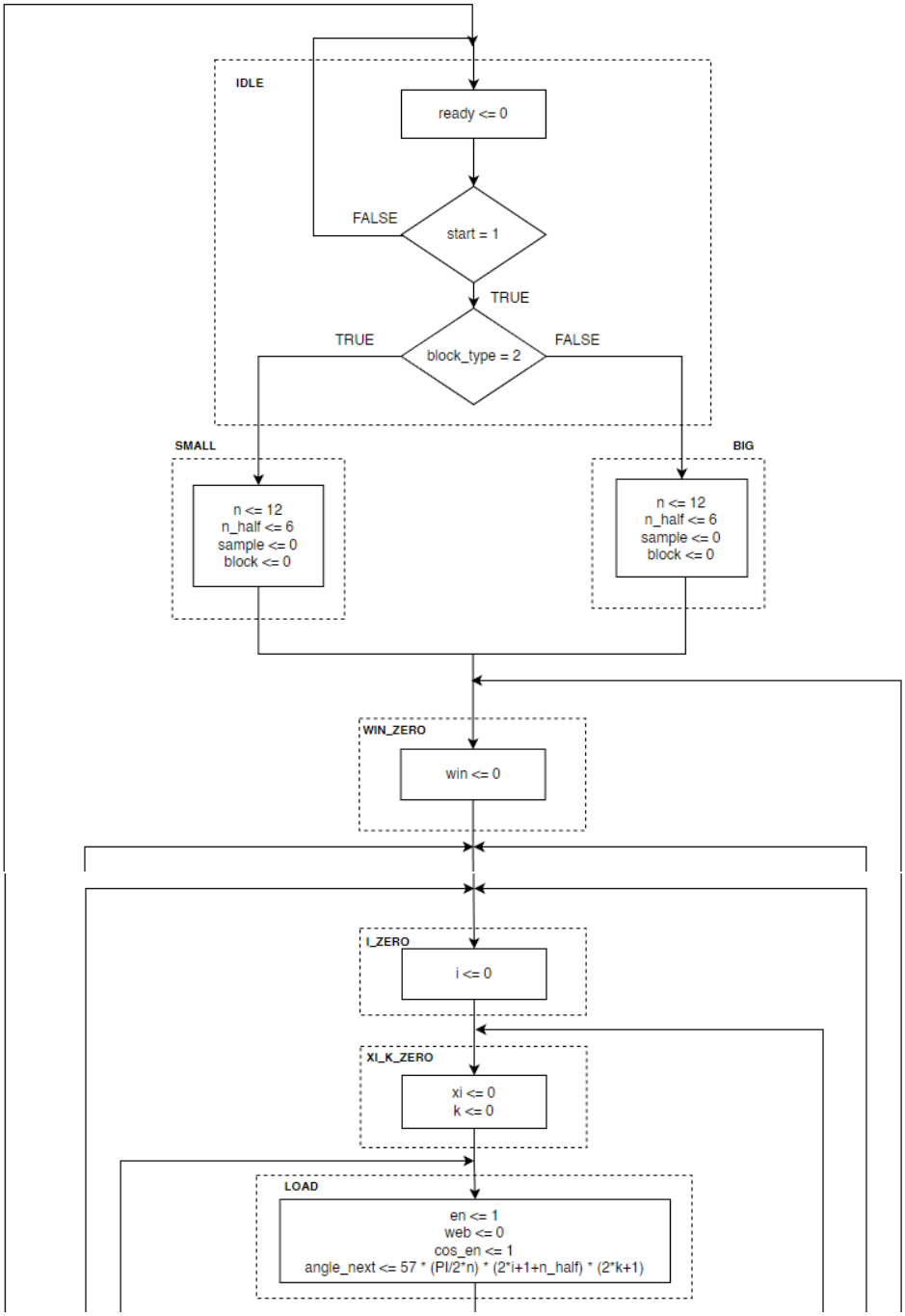
```
void imdct(int gr, int ch , int block_type)
{

    float sample_block[72];
    const int n = block_type == 2 ? 12 : 36;
    const int half_n = n / 2;
    int sample = 0;
    int k, i, win, j, block, m;
    float xi, s;

    block = 0;
    lab_5:

        win = 0;
        lab_3:
            i = 0;
            lab_2:
                xi = 0.0;
                k = 0;
                    lab_1:
                        s = samples[gr][ch][18 * block + half_n * win + k];
                        xi = xi + s * cosDegrees(57 * 0.12890625 * (2*i + 1 + half_n) * (2*k + 1));
                        k = k + 1;
                        if (k == half_n)
                            goto exit_1;
```

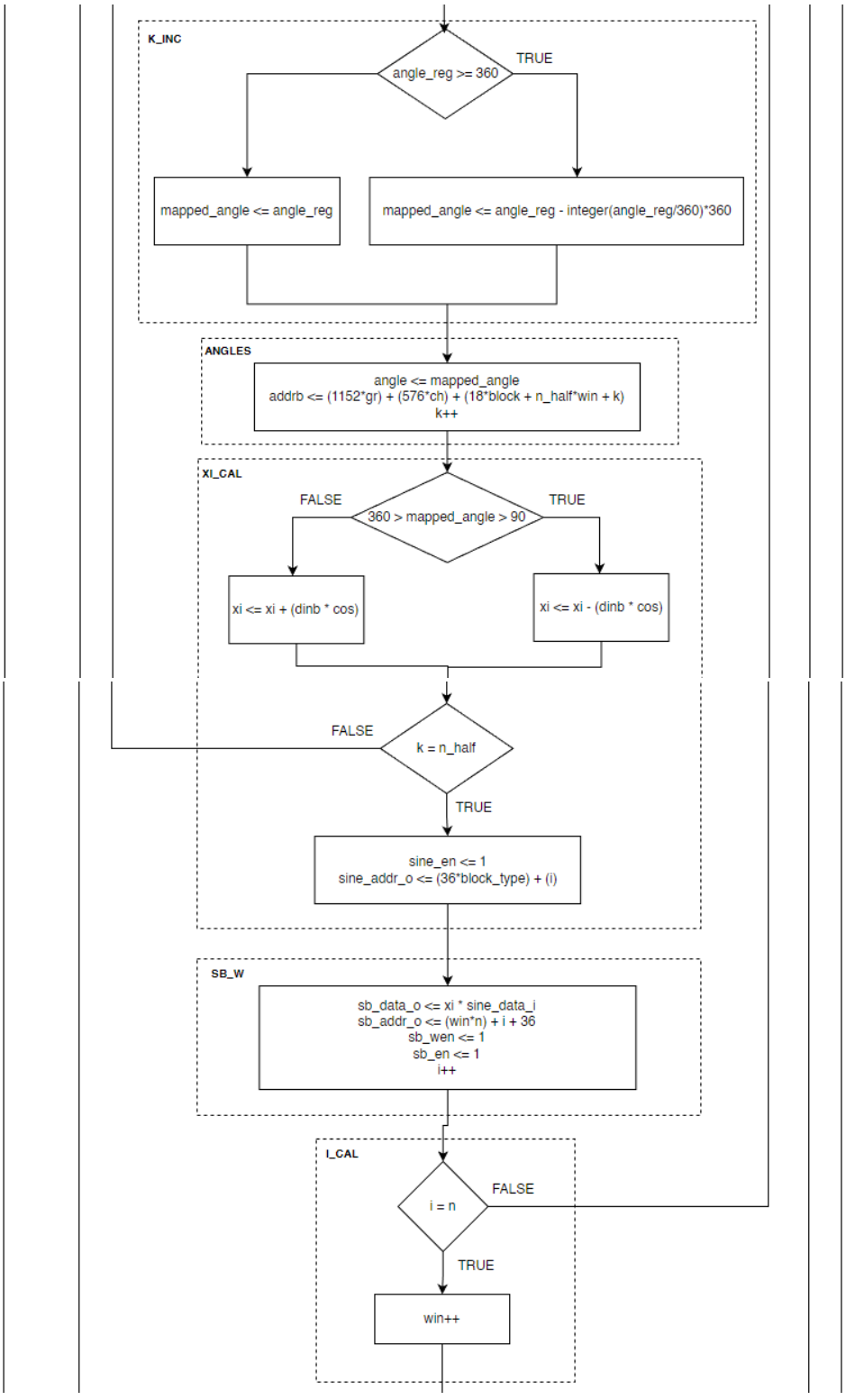
```
                else
                    goto lab_1;
            exit_1:
            sample_block[win * n + i + 36] = xi * sine_block[block_type][i];
            i = i + 1;
        if (i == n)
            goto exit_2;
        else
            goto lab_2;
    exit_2:
    win = win + 1;
    if (win == (block_type == 2 ? 3 : 1))
        goto exit_3;
    else
        goto lab_3;
exit_3:



if (block_type == 2) {

    m = 0;

    lab_6:
    sample_block[m] = 0;
    m = m + 1;

    if(m == 6)
        goto exit_6;
    else
        goto lab_6;
    exit_6:

    lab_7:
    sample_block[m] = sample_block[m - 6 + 36];
    m = m + 1;
    if(m == 12)
        goto exit_7;
    else
        goto lab_7;
    exit_7:

    lab_8:
    sample_block[m] = sample_block[m - 6 + 36] + sample_block[m + 36];
    m = m + 1;
    if(m == 18)
        goto exit_8;
    else
        goto lab_8;
    exit_8:

    lab_9:
    sample_block[m] = sample_block[m + 36] + sample_block[6 + m + 36];
    m = m + 1;
    if(m == 24)
        goto exit_9;
    else
        goto lab_9;
    exit_9:

    lab_10:
    sample_block[m] = sample_block[6 + m + 36];
    m = m + 1;
    if(m == 30)
        goto exit_10;
    else
        goto lab_10;
    exit_10:

    lab_11:
    sample_block[m] = 0;
                      file << sample_block[m] << endl;
    m = m + 1;
    if(m == 36)
        goto exit_11;
    else
        goto lab_11;
    exit_11:
    ;



}

j = 0;
lab_4:

                      if(block_type == 2)
                               prev_samples[ch][block][j] = sample_block[18 + j];
                      else
                               prev_samples[ch][block][j] = sample_block[18 + j + 36];

                      if(block_type = 2)
                               samples[gr][ch][sample + j] = sample_block[j] + prev_samples[ch][block][j];
```

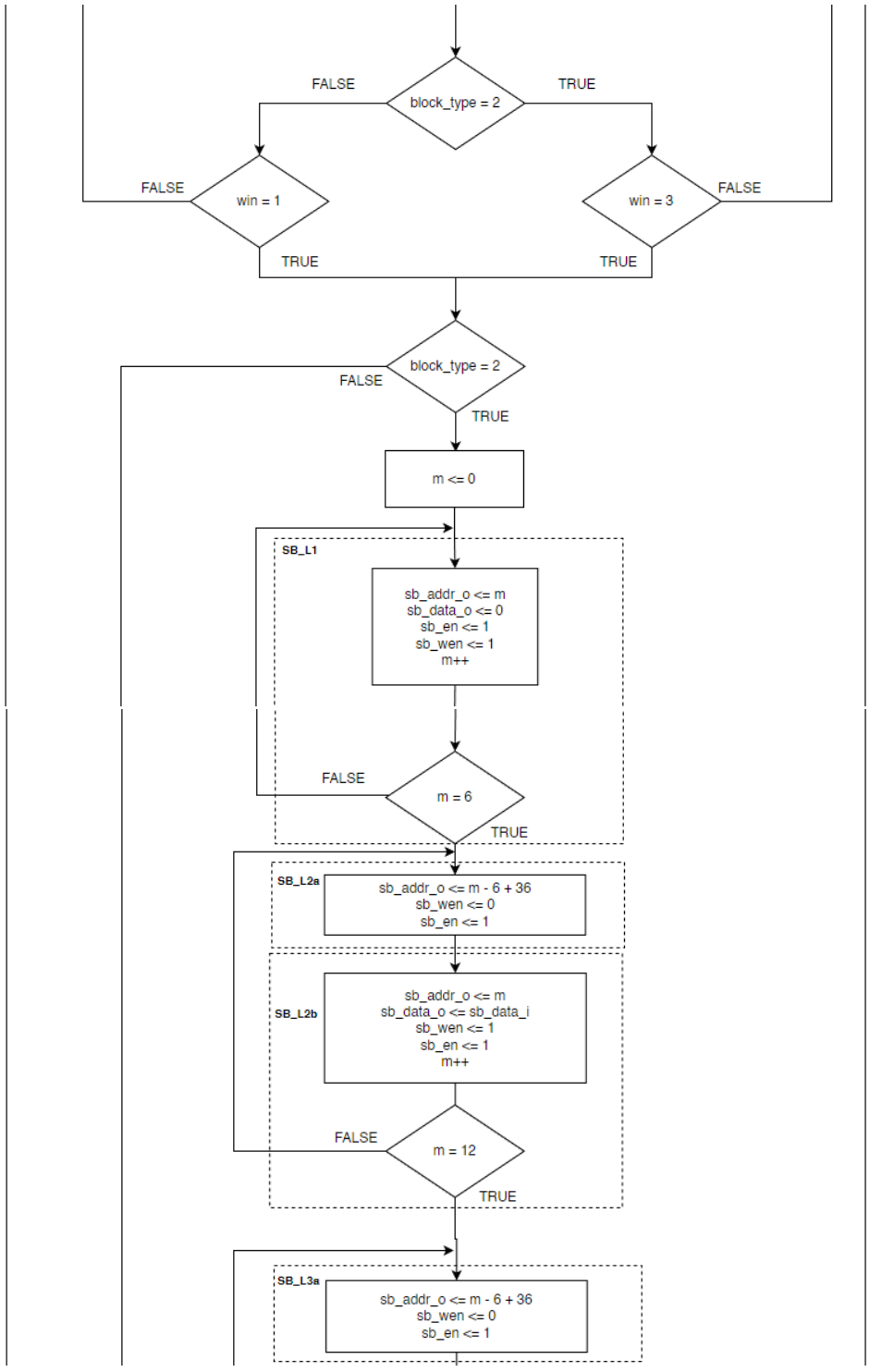
```
                            else
                                    samples[gr][ch][sample + j] = sample_block[j + 36] + prev_samples[ch][block][j];

            j = j + 1;
        if(j == 18)
            goto exit_4;
        else
            goto lab_4;

        exit_4:
        sample = sample + 18;
        block = block + 1;
    if(block == 32)
        goto exit_5;
    else
        goto lab_5;

    exit_5:
        return;
}
```

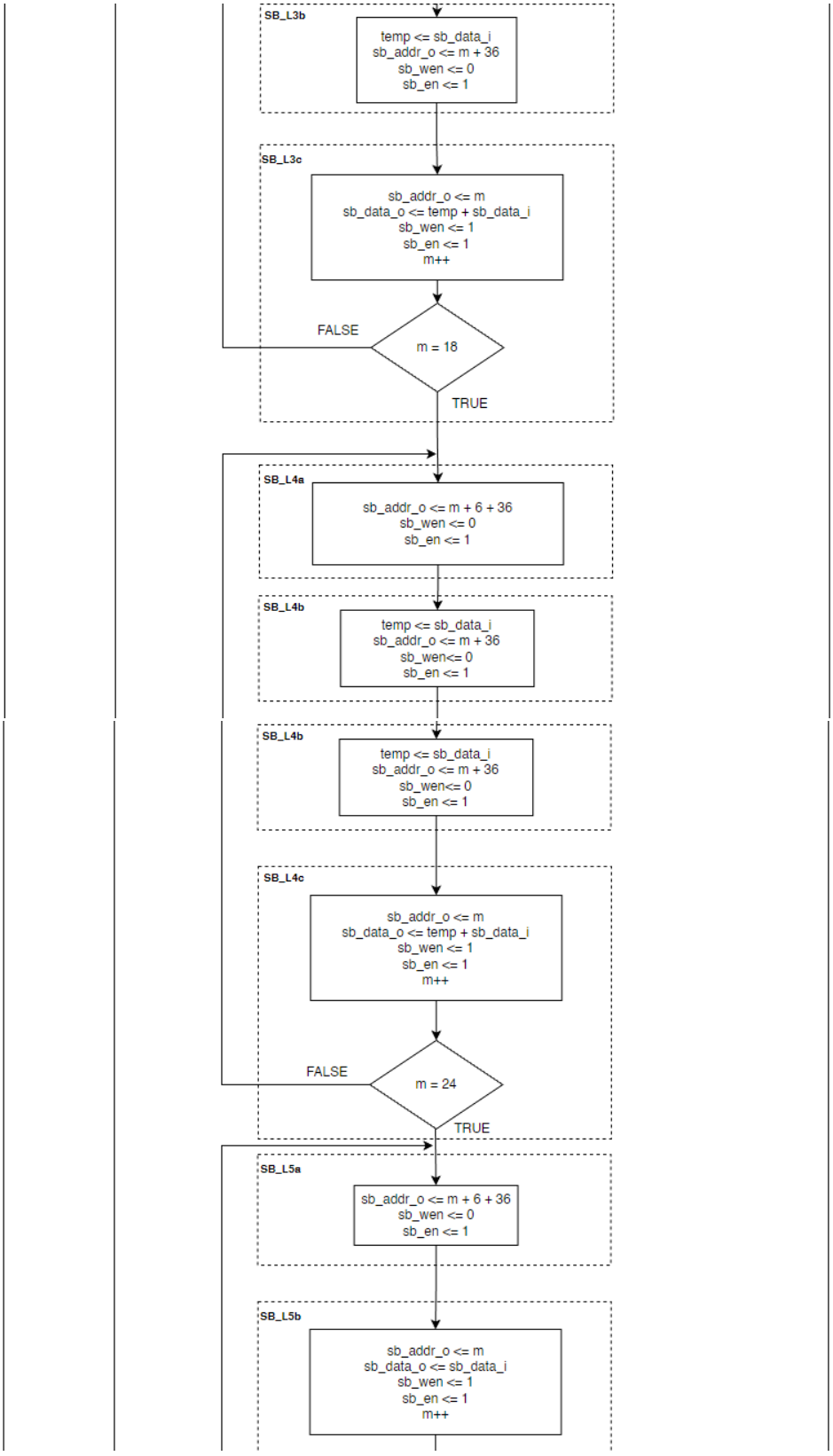
Slika 4.2 IMDCT algoritam sa goto naredbama

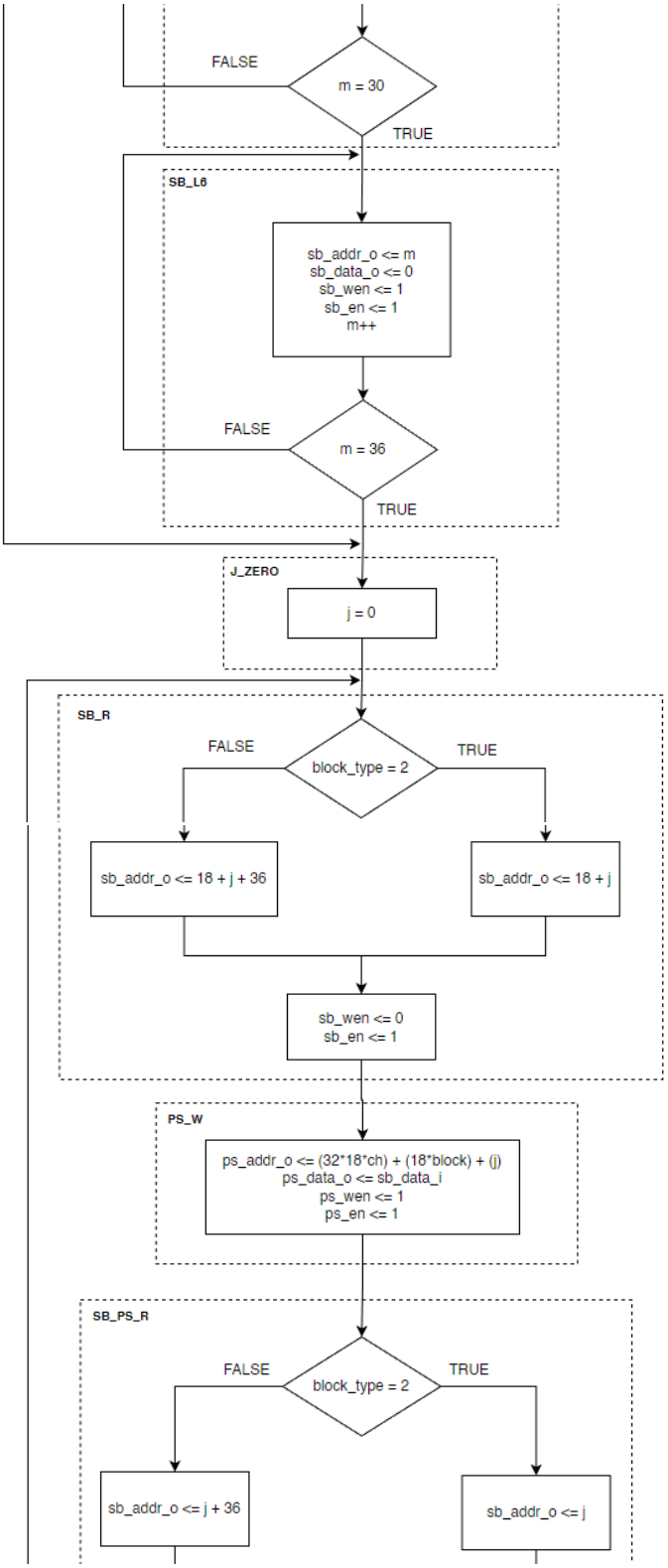
Na osnovu prethodno opisanih modifikacija algoritma i njegove implementacije sa *goto* naredbama, slika 4.2. Napravljen je ASM dijagram slika 4.3, na osnovu koga je napisan RTL model sistema u VHDL jeziku.

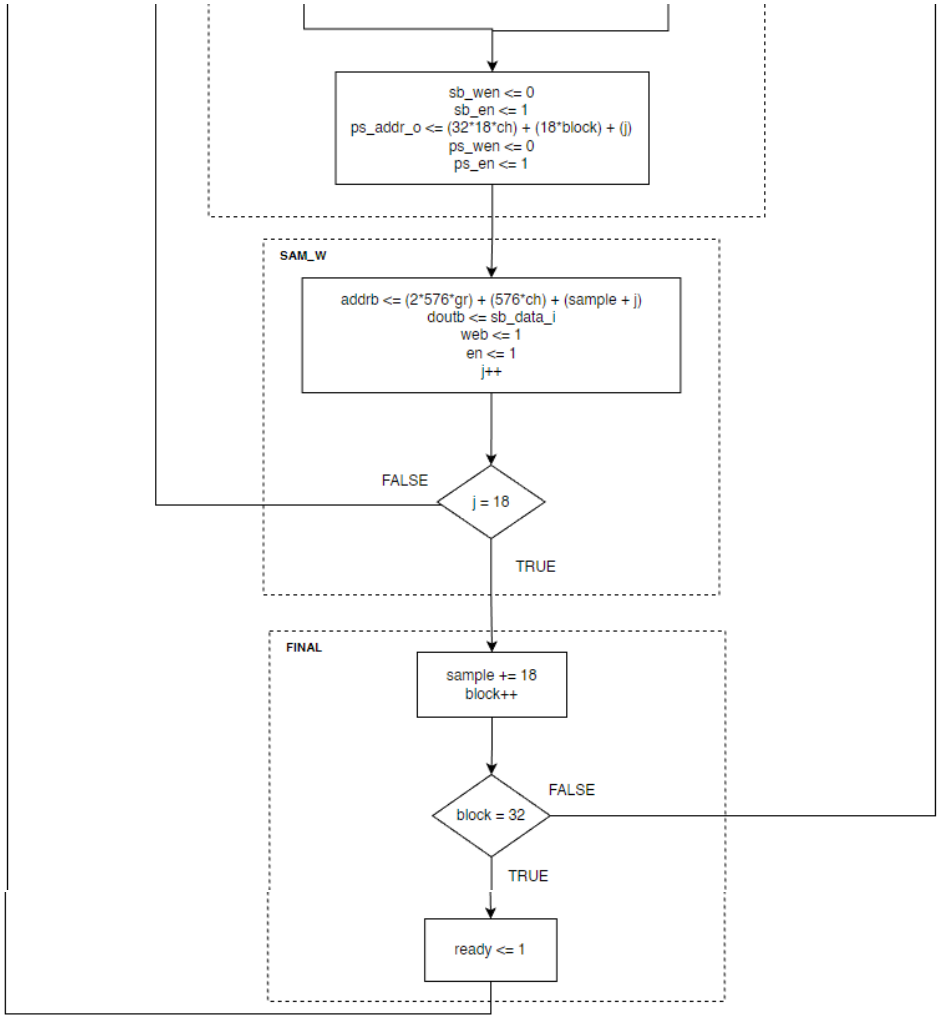

IDLE
ready <= 0
FALSE
start = 1
TRUE
TRUE
block_type = 2
FALSE
SMALL
n <= 12
n_half <= 6
sample <= 0
block <= 0
BIG
n <= 12
n_half <= 6
sample <= 0
block <= 0
WIN_ZERO
win <= 0
I_ZERO
i <= 0
XI_K_ZERO
xi <= 0
k <= 0
LOAD
en <= 1
web <= 0
cos_en <= 1
angle_next <= 57 * (PI/2*n) * (2*i+1+n_half) * (2*k+1)


K_INC
angle_reg >= 360
TRUE
mapped_angle <= angle_reg
mapped_angle <= angle_reg - integer(angle_reg/360)*360
ANGLES
angle <= mapped_angle
addrb <= (1152*gr) + (576*ch) + (18*block + n_half*win + k)
k++
XI_CAL
FALSE
360 > mapped_angle > 90
TRUE
xi <= xi + (dinb * cos)
xi <= xi - (dinb * cos)
FALSE
k = n_half
TRUE
sine_en <= 1
sine_addr_o <= (36*block_type) + (i)
SB_W
sb_data_o <= xi * sine_data_i
sb_addr_o <= (win*n) + i + 36
sb_wen <= 1
sb_en <= 1
i++
I_CAL
i = n
FALSE
TRUE
win++


block_type = 2
FALSE
TRUE
win = 1
win = 3
FALSE
FALSE
TRUE
TRUE
block_type = 2
FALSE
TRUE
m <= 0
SB_L1
sb_addr_o <= m
sb_data_o <= 0
sb_en <= 1
sb_wen <= 1
m++
m = 6
FALSE
TRUE
SB_L2a
sb_addr_o <= m - 6 + 36
sb_wen <= 0
sb_en <= 1
SB_L2b
sb_addr_o <= m
sb_data_o <= sb_data_i
sb_wen <= 1
sb_en <= 1
m++
m = 12
FALSE
TRUE
SB_L3a
sb_addr_o <= m - 6 + 36
sb_wen <= 0
sb_en <= 1


SB_L3b
temp <= sb_data_i
sb_addr_o <= m + 36
sb_wen <= 0
sb_en <= 1
SB_L3c
sb_addr_o <= m
sb_data_o <= temp + sb_data_i
sb_wen <= 1
sb_en <= 1
m++
FALSE
m = 18
TRUE
SB_L4a
sb_addr_o <= m + 6 + 36
sb_wen <= 0
sb_en <= 1
SB_L4b
temp <= sb_data_i
sb_addr_o <= m + 36
sb_wen<= 0
sb_en <= 1
SB_L4b
temp <= sb_data_i
sb_addr_o <= m + 36
sb_wen<= 0
sb_en <= 1
SB_L4c
sb_addr_o <= m
sb_data_o <= temp + sb_data_i
sb_wen <= 1
sb_en <= 1
m++
FALSE
m = 24
TRUE
SB_L5a
sb_addr_o <= m + 6 + 36
sb_wen <= 0
sb_en <= 1
SB_L5b
sb_addr_o <= m
sb_data_o <= sb_data_i
sb_wen <= 1
sb_en <= 1
m++

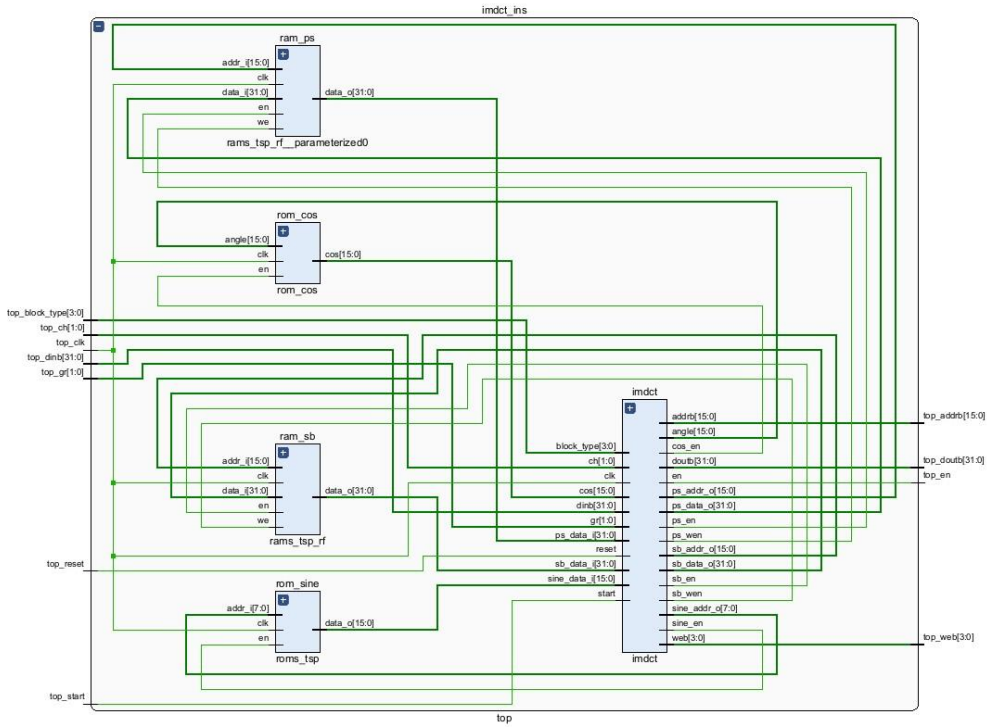

FALSE
m = 30
TRUE
SB_L6
sb_addr_o <= m
sb_data_o <= 0
sb_wen <= 1
sb_en <= 1
m++
FALSE
m = 36
TRUE
J_ZERO
j = 0
SB_R
FALSE
block_type = 2
TRUE
sb_addr_o <= 18 + j + 36
sb_addr_o <= 18 + j
sb_wen <= 0
sb_en <= 1
PS_W
ps_addr_o <= (32*18*ch) + (18*block) + (j)
ps_data_o <= sb_data_i
ps_wen <= 1
ps_en <= 1
SB_PS_R
FALSE
block_type = 2
TRUE
sb_addr_o <= j + 36
sb_addr_o <= j



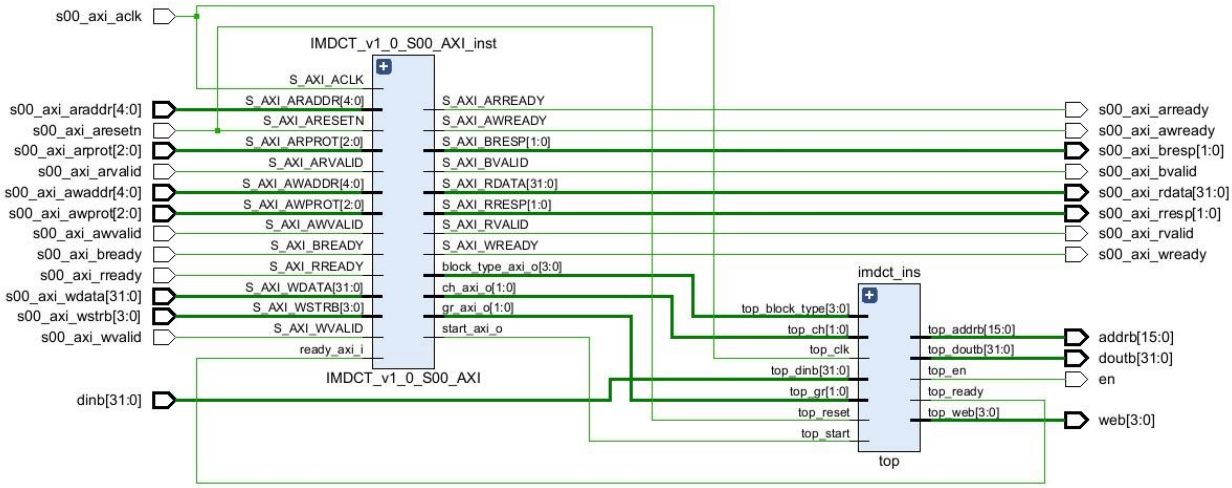
Slika 4.2 ASMD IMDCT

5. Pakovanje komponenti i povezivanje celog sistema

Interfejs komponente sa slike 5.1 je prikaz samo komponente top level nivao, bez njegovog povezivanja sa AXI Lite kontrolerom i bez oklopljivanja celog sistema. Na slici 5.2 je prikazano kako izgleda sistem kada se oklopi.

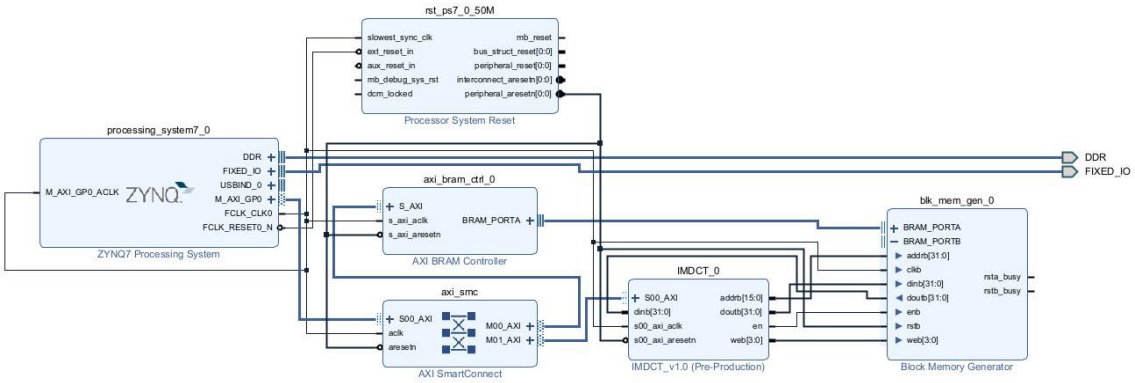


Slika 5.1 Izgled sintetizovanog IP bloka



Slika 5.2 Izgled sintetizovanog IP bloka sa AXI Lite interfejsom

Nakon pakovanja komponenta je u IP integratoru povezana sa ostatom sistema. Blok šema je prikazana na slici 5.3.



Slika 5.3 Blok šema celokupnog sistema

Na blok šemi se može videti koponente: *AXI SmartConnect* koja služi za međusobno povezivanje ostalih delova sistema, procesorski sistem *ZINQ 7 Processing System*, komponenta za procesorski reset sistema, komponenta koju smo projektovali *IMDCT_0,* jedan BRAM kontroler koji služi za upis i čitanje *sample*-ova od strane softverskog dela algoritma, tu je i *Block Memory Generator* koji instancira BRAM u kome su smešteni *sample*-ovi

U dokumentaciji System level dizajna predviđeno je da će sistem raditi na frekvenciji od 52 MHz, međutim zaključak nakon implementacije je da sistem može da radi do 50 MHz.

U svakom koraku modelovanja zabeležena je količina iskorišćenosti resursa sa kojima raspolažemo. U nastavku je prikazana količina resursa za TOP modul, IP blok i kompletan sistem respektivno.

Resurs	Ukupna količina	Potrošeno	Procentualna iskorišćenost
LUT	17600	1787	10,15%
Flip-Flop	35200	178	0,51%
Block RAM	60	3,5	5,83%

Tabela 5.1 Iskorišćenost resursa TOP modula

Resurs	Ukupna količina	Potrošeno	Procentualna iskorišćenost
LUT	17600	1826	10,38%
Flip-Flop	35200	350	0,99%
Block RAM	60	3,5	5,83%

Tabela 5.2 Iskorišćenost resursa IP bloka

Resurs	Ukupna količina	Potrošeno	Procentualna iskorišćenost
LUT	17600	5970	33,92%
Flip-Flop	35200	4749	13,49%
Block RAM	60	5,5	9,17%

Tabela 5.3 Iskorišćenost resursa IP kompletnog sistema

6. Funkcionalna verifikacija

Funkcionalna verifikacija ima zadatak da proveri da li je dizajn sistema u skladu sa specifikacijom, tj. da li obavlja onu funkciju za koju je namenjen. Za izradu verifikacionog dela projekta korišćena je UVM (eng. Universal Verification Methodology) metodologija.

Proces verifikacije se odvijao u nekoliko sledećih koraka:

- Verifikacioni plan
- Razvijanje verifikacionog okruženja
- Struktura verifikacionog okruženja
- Rezultati simulacije

6.1 Verifikacioni plan

Prvobitno je sistem verifikovan na *unit* nivou u sklopu projektovanja digitalnog sistema tako što je napravljen jednostavan *testbench* u kome je testirana osnovna funckionalnost IMDCT-a. Poslata je jedna vrednost na *dinb* ulaz ne uzimajući u obzir adresu sa koje je pročitana.

Potom je napravljeno UVM okruženje koje verifikuje potpuno zapakovan dizajn i proverava ne samo funkcionalnost već i poštovanje AXI-Lite protokola, komunikaciju itd.

Nakon toga je proverena funckionalnost *reseta* koji se aktivira pre početka simulacije i resetuje sistem. Potom je proverena osnovna funkcionalnost slanja podataka na odgovarajuće pinove AXI-litea nakon čega se čekaju odgovarajući rezultati na *gr*, *ch*, *block type*, *ready* i *start* registrima. Nakon pokretanja sistema hardverskog akceleratora (DUT-a), izvršena je simulacija BRAM memorije tako što se slao fiksan podatak sa svake adrese BRAM-a našem DUT-u i čekala se obrada tih podataka kao i odgovarajući rezultati na *doutb* i *ready* pinovima.

Za test scenarije kao i same podatke, randomizovan je unos. Nakon završetka rada DUT-a, odgovarajućim komponentama (*monitor* i *scoreboard*) se prate izlazni podaci i njihova tačnost. Kao što je već prethodno rečeno, u projektu je korišćen softverski alat Vivado a za opis sistema programski jezik System Verilog.

6.2 Razvijanje verifikacionog okruženja

Verifikaciono okruženje sadrži sledeće komponente:

- Top modul
- Environment i test
- Config
- Agent
- Driver
- Monitor
- Scoreboard
- Sequence item i sequence

Ove komponente su nasleđene iz klasa opisanih u UVM biblioteci i prilagođene su ovom dizajnu.

6.2.1 Top modul

Na najvišem nivou instancioniran je DUT, interfejsi AXI Lite i BRAM i izvršena factory registracija. Formirani su globalni takt i reset signali i pokrenut je test.

6.2.2 Environment i test

Komponenta *environment* instancira i objedinjuje prethodno opisane komponente. *Test* objekat instancira i enkapsulira environment komponentu i pokreće sekvence na odgovarajućem sekvenceru.

6.2.3 Config

Konfiguracioni objekat sadrži sve opcije vezane za konfiguraciju agenta kao i odabir režima rada. Osnovna funkcionalnost, a ujedno i ona koja se najčešće sreće je odabir kreiranja agenta, dali će biti aktivan ili pasivan. Naš slučaj zahteva samo tu funkcionalnost konfiguracionog objekta.

6.2.4 Agent

Agent je komponenta koja obuhvata driver, sekvencer i monitor. Ove tri komponente mogu da rade i bez agenta, međutim on nam omogućava ponovno korišćenje ovih komponenti. Naš slučaj zahteva dva agenta koji sadrže različite komponente. Razlog za to su dva interfejsa AXI Lite i BRAM koji imaju različitu ulogu i funkcionalnost. Svaki od njih zahteva poseban agent sa monitorom, drajverom i sekvencerom. Oba su aktivna jer su drajver i sekvencer neophodni za pravilno funkcionisanje okruženja.

6.2.5 Driver, Monitor, Scoreboard

Drajver komponenta prima sekvence preko sekvencera i poštujući protokol koji implementira postavlja signale na ulaze DUT-a. Naš dizajn zahteva dva drajvera koji implementiraju AXI Lite i BRAM interfejse. Drajver koji implementira AXI Lite interfejs šalje informacije o *gr*, *ch*, *block type*, *start* registrima kao i *ready* registar. Određene potrebne promenjive koje se čuvaju u registrima. Pomoću *start* registra, IMDCT se pokreće, a pomoću *ready* registra znamo kada je IMDCT proces završen. Drajver koji implementira BRAM interfejs šalje podatak na ulaz DUT-a nad kojim treba da seizvrši obrada. Sekvencer šalje podatke drajveru i prima odgovor od drajvera ukoliko je to potrebno. Njegova ulog je kontrola transakcija iz jedne ili više sekvenci. Za naš primer su potrebna dva sekvencera koja su paramterizovana tipovima AXI Lite i BRAM transakcija (axi_lite_item, bram_item). Monitor je komponenta za nadgledanje ulaznih i/ili izlaznih signala. Monitor za AXI Lite interfejs nadgleda signale preko kojih drajver šalje podatke

DUT-u. Od interesa su mu podaci za *gr*, *ch*, *block type* i *start* registre kao I ready registar koji označava završetak *IMDCT* procesa.

```
task main_phase(uvm_phase phase);
  //axi_lite_seq_item = axi_lite_seq_item::type_id::create("axi_lite_seq_item",this);


  forever begin

  if (vif.reset_n == 'b0) @(posedge vif.reset_n);
  if (vif.s_axi_awvalid == 'b1) begin

    axi_lite_seq_item = axi_lite_item::type_id::create("axi_lite_seq_item",this);

    axi_lite_seq_item.write = 1; //write
    axi_lite_seq_item.address  = vif.s_axi_awaddr[4:0];
    $display("Collected ...: Address :=  %d \n", axi_lite_seq_item.address);
    @(negedge vif.s_axi_wvalid);
    axi_lite_seq_item.data  = vif.s_axi_wdata;
    $display("Collected ...: Data :=  %d \n", axi_lite_seq_item.data);
    @(posedge vif.s_axi_wvalid);


    $cast(item_clone1, axi_lite_seq_item.clone());
    item_collected_port.write(item_clone1);

  end
  else if (vif.s_axi_arvalid == 'b1) begin
  read_address.sample();
    axi_lite_seq_item = axi_lite_item::type_id::create("axi_lite_seq_item",this);
    axi_lite_seq_item.read = 1;      //read
    axi_lite_seq_item.address  = vif.s_axi_araddr[4:0];
    $display("Collected ...: Address :=  %d \n", axi_lite_seq_item.address);
    @(negedge vif.s_axi_rvalid);
    axi_lite_seq_item.data  = vif.s_axi_rdata;
    $display("Collected ...: Data :=  %d \n", axi_lite_seq_item.data);
    @(posedge vif.s_axi_rvalid);

    $cast(item_clone1, axi_lite_seq_item.clone());
    item_collected_port.write(item_clone1);

  end
  @(posedge vif.clock);



  end

endtask // main_phase
```

Slika 5.4 Monitor za Axi Lite

Na slici 5.4 je prikazan kod AXI Lite monitora. Ukoliko je adresa validna kreira se novi item koji će imati ista polja kao i sequence item. Adresnom polju novog item-a dodeljuje se vrednost adrese koja u tom trenutku putuje prema DUT-u. Takođe, ukoliko je validan podatak koji putuje zajedno sa tom adresom, njegova vrednost će biti dodeljena polju za podatak (data) unutar novokreiranog item-a. Sa prikupljenim informacijama (adresom i podatkom) novokreirani

item se klonira i šalje scoreboard-u. Kada naiđe nova adresa i podatak koji putuju ka DUT-u, monitor se aktivira i ponavlja prethodno opisani postupak. Sa slike se može videti da je isti postupak odrađen i prilikom čitanja

```
36
37      task run_phase(uvm_phase phase);
38
39              bram_seq_item = bram_item::type_id::create("bram_seq_item", this);
40
41          forever begin
42
43
44              @(posedge vif.clock iff vif.s_en_bram) begin
45              bram_seq_item.address = vif.s_addr_bram;
46              bram_seq_item.en = vif.s_en_bram;
47              bram_seq_item.out_data = vif.s_dout_bram;
48              bram_seq_item.we = vif.s_we_bram;
49
50              if(vif.s_we_bram == 1)begin
51                  bram_seq_item.in_data = vif.s_din_bram;
52              end
53              $cast(item_clone, bram_seq_item.clone());
54              `uvm_info(get_type_name(), $sformatf("Address of BRAM is: \t%d, data is : \t%d", item_clone.address, vif.s_dout_bram), UVM_HIGH)
55              item_collected_port.write(item_clone);
56
57          end //(posedge vif.s_en_bram)
58      end
59      endtask : run_phase
60
61  endclass : bram_monitor
62  `endif
```

Slika 5.5 Monitor za BRAM

Na slici 5.5 je prikazan kod BRAM monitora. Kao što vidimo u kodu, prvo se kreira novi item koji će imati ista polja kao i sequence item BRAM-a, i ukoliko su clock i enable BRAM-a na 1, tada se adresnom polju novog itema dodeljuje ista vrednost adrese koja u tom trenutku putuje prema DUT-u. Takođe se dodeljuje i izlazni podatak(out_data) kao i enable i write-enable. U daljem nastavku koda, ako je write-enable na 1, to označava da se može upisivati u BRAM, i podatak koji putuje zajedno sa adresom BRAM-a će biti dodeljen polju za ulazni podatak(in_data). . Sa prikupljenim informacijama (adresom i podatkom) novokreirani item se klonira i šalje scoreboard-u

6.2.6 Sequence item, sequence

U našem slučaju imamo dva sequence itema:
- **axi_lite_item** - koji se odnosi na agenta koji upravlja AXI Lite interfejs.
- **bram_item**- koji se odnosi na agent_full koji upravlja BRAM interfejs.

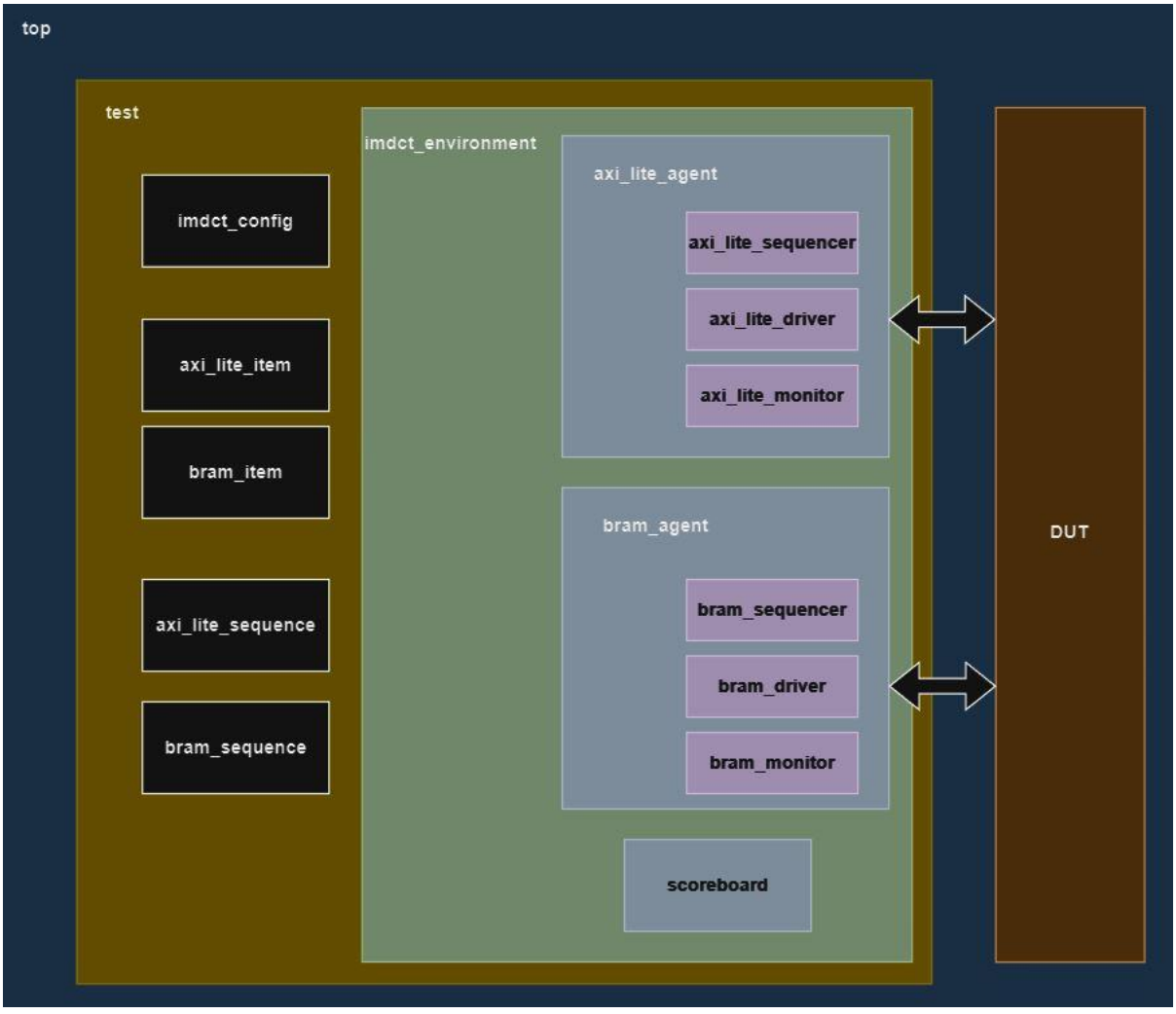
Sequence item za AXI Lite interfejs sadrži randomizovane vrednosti za adrese i podatke, kao i signali read i write kako bi znali da li se u nekom trenutku zahteva čitanje ili upis.
U sequence itemu za BRAM interfejs se nalaze randomizovane vrednosti za adrese i podatke, kao i web i en koji odredjuju upisivanje i čitanje u BRAM.

Sekvenca (*sequence*) se sastoji iz:

- **axi_lite_seq (AXI Lite)**
- **bram_seq (BRAM)**

U sekvenci za AXI Lite interfejs na početku se upisuje u registre *gr*, *ch*, *block type* i *start* koji pokreće *IMDCT*. Nakon što AXI Lite upiše u određene registre, tada IMDCT kreće da obrađuje rezultate, a u sekvenci za BRAM, šaljemo samo jednu vrednost na ulaz *dinb*.

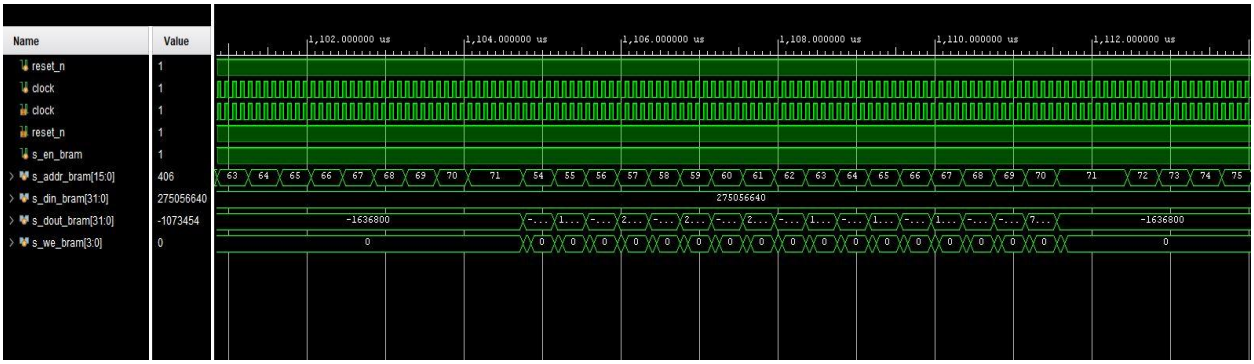
6.3 Struktura verifikacionog okruženja



Slika 5.6 Struktura

6.4 Rezultati simulacije

Na slici 5.7 prikazni su simulacioni rezultati na primeru stimulusa koji je već ranije pominjan tokom analize dizajna.



Slika 5.7 Simulacija

7. Zaključak

Na osnovu svih prikazanih analiza i testova može se zaključiti da je projektovanje sistema uspešno. Razmatrani su razni načini komunikacije u sistemu i svi oni funkcionišu na ispravan način. Naravno ovde nije kraj, jer sistem ima još puno prostora za unapređenje i to se odnosi kako na dizajn tako i na verifikaciono okruženje. Verifikaciono okruženje nije kompletno kako nije odrađena coverage analiza. Takođe nisu formirani regresioni testovi na osnovu kojih bi se zaista do kraja dokazala funkcionalnost sistema a pre toga otkrili skriveni bagovi sistema.