



УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У
НОВОМ САДУ



Дејан Грубишић

ЈЕДНО ХАРДВЕРСКО-СОФТВЕРСКО КОДИЗАЈН РЕШЕЊЕ ПРОГРАМА ЗА ИГРАЊЕ ШАХА

ДИПЛОМСКИ РАД
- Основне академске студије -

Ментор: Вук Вранковић

Нови Сад, 2018



УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
КАТЕДРА ЗА МИКРОРАЧУНАРСКУ
ЕЛЕКТРОНИКУ



ЈЕДНО ХАРДВЕРСКО-СОФТВЕРСКО КОДИЗАЈН РЕШЕЊЕ
ПРОГРАМА ЗА ИГРАЊЕ ШАХА

ДИПЛОМСКИ РАД

Основне академске студије

кандидат:

Дејан Грубишић, ЕЕ 20-2014

ментор:

др Вук Вранковић, доцент

Јул 2018



УНИВЕРЗИТЕТ У НОВОМ САДУ • ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
21000 НО ВИ СА Д, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА


Редни број, РБР:	
Идентификациони број, ИБР:	
Тип документације, ТД:	Монографска документација
Тип записа, ТЗ:	Текстуални штампани материјал
Врста рада, ВР:	Завршни рад
Аутор, АУ:	Дејан Грубишић
Ментор, МН:	др Вук Вранковић, доцент
Наслов рада, НР:	ЈЕДНО ХАРДВЕРСКО-СОФТВЕРСКО КОДИЗАЈН РЕШЕЊЕ ПРОГРАМА ЗА ИГРАЊЕ ШАХА
Језик публикације, ЈП:	Српски
Језик извода, ЈИ:	Српски
Земља публикавања, ЗП:	Република Србија
Уже географско подручје, УГП:	Војводина
Година, ГО:	2018
Издавач, ИЗ:	Ауторски репринт
Место и адреса, МА:	Нови Сад, Трг Доситеја Обрадовића 6
Физички опис рада, ФО: (поглавља/страна/ цитата/табела/слика/графика/прилога)	(5/64/0/0/32/0/0)
Научна област, НО:	Електроника
Научна дисциплина, НД:	Системско пројектовање, дизајн и верификација дигиталног система
Предметна одредница/Кључне речи, ПО:	Системско пројектовање, дизајн и верификација дигиталног система, шах, хардверска акцелерација
УДК	
Чува се, ЧУ:	Библиотека Факултета Техничких Наука, 21000 Нови Сад, Трг Доситеја Обрадовића 6
Важна напомена, ВН:	Нема
Извод, ИЗ:	У раду је извршено пројектовање хардверско-софтверског решења програма за играње шаха. У првом поглављу дат је садржај дипломског рада. У другом поглављу приказано је моделовање на системском нивоу. У трећем поглављу извршено је пројектовање хардверског модула за убрзавање евалуације позиције, док је у четвртном поглављу урађена и његова функционална верификација. Пето поглавље је закључак.
Датум прихватања теме, ДП:	
Датум одбране, ДО:	
Чланови комисије, КО:	Председник: др Растислав Струхарик, ванредни професор
	Члан: маг. Андреа Ердељан, асистент
	Члан, ментор: др Вук Вранковић, доцент
	Потпис ментора



УНИВЕРЗИТЕТ У НОВОМ САДУ • ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
21000 НО ВИ СА Д, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Accession number, ANO :	
Identification number, INO :	
Document type, DT :	Monographic publication
Type of record, TR :	Textual material, printed
Contents code, CC :	Final Thesis
Author, AU :	Dejan Grubišić
Mentor, MN :	dr Vuk Vranković, docent
Title, TI :	One hardware-software codesign solution of chess engine
Language of text, LT :	Serbian
Language of abstract, LA :	Serbian
Country of publication, CP :	Republic of Serbia
Locality of publication, LP :	Vojvodina
Publication year, PY :	2018
Publisher, PB :	Author's copy
Publication place, PP :	Novi Sad, Trg Dositeja Obradovića 6
Physical description, PD : (chapters/pages/ref./tables/pictures/graphs/appendixes)	(5/64/0/0/32/0/0)
Scientific field, SF :	Electronics
Scientific discipline, SD :	System design, digital design and functional verification
Subject/Key words, S/KW :	System design, design and verification of digital system, chess, hardware acceleration
UC	
Holding data, HD :	Library of the Faculty of Technical Sciences, 21000 Novi Sad, Trg Dositeja Obradovića 6
Note, N :	None.
Abstract, AB :	<p>This paper demonstrates the solution of the hardware-software co-design of the program for playing chess. The first chapter gives the content of graduation work. The second chapter presents modeling at the system level. The third chapter describes the design of the hardware module that accelerate evaluation of the chess position. In the fourth chapter it has been made the functional verification of designed module. Chapter 5 is conclusion.</p>
Accepted by the Scientific Board on, ASB :	
Defended on, DE :	
Defended Board, DB :	President: dr Rastislav Struharik, associate professor
	Member: M.Sc. Andrea Erdeljan, assistant
	Member, Mentor: dr Vuk Vranković, docent
	Mentor's sign

	УНИВЕРЗИТЕТ У НОВОМ САДУ • ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА 21000 НОВИ САД, Трг Доситеја Обрадовића 6	Број:
	ЗАДАТАК ЗА ЗАВРШНИ (BACHELOR) РАД	Датум:

(Податке уноси предметни наставник - ментор)

Врста студија:	а) Основне академске студије б) Основне струковне студије
Студијски програм:	ЕМБЕДЕД СИСТЕМИ И АЛГОРИТМИ
Руководилац студијског програма:	др Милан Сечујски, ванредни професор

Студент:	Дејан Грубишић	Број индекса:	ЕЕ20/2014
Област:	Системско пројектовање, дизајн и верификација дигиталног система		
Ментор:	др Вук Вранковић, доцент		

НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА ЗАВРШНИ (Bachelor) РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:

- проблем – тема рада;
- начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна;
- литература

НАСЛОВ ЗАВРШНОГ (Бечелор) РАДА:

Једно хардверско-софтверско кодизајн решење програма за играње шаха

ТЕКСТ ЗАДАТКА:

- 1) Извршити пројектовање и анализу на системском нивоу хардверско софтверског решења програма за играње шаха, коришћењем ESL методологије
- 2) Извршити пројектовање дигиталног система за евалуацију позиције, заснованом на RT методологији
- 3) Верификовати пројектовани дизајн коришћењем UVM методологије

Руководилац студијског програма:	Ментор рада:
др Милан Сечујски, ванредни професор	др Вук Вранковић, доцент

Примерак за: ☐ - Студента; ☐ - Студентску службу факултета

Sadržaj

Glava 1 Sadržaj diplomskog rada	1
Glava 2 Projektovanje aplikacije na sistemskom nivou	4
2.1 ESL Metodologija	4
2.2 Specifikacija i izvorni kod šahovskog programa	6
2.3 Analiza pre particionisanja	7
2.4 Particionisanje	8
2.5 Implementacija SystemC modela posle particionisanja	9
2.6 Implementacija eval modula u SystemC-u	12
Glava 3 Projektovanje hardverskog IP bloka	15
3.1 RT metodologija	15
3.2 Diskusija mogućih implementacija	20
3.3 Implementacija bloka Select Piece	22
3.3.1 Definisanje interfejsa	22
3.3.2 Projektovanje Controlpath modula	25
3.3.3 Projektovanje Datapath modula	27
3.4 Implementacija bloka Pawn Rank	28
3.4.1 Interfejs	28
3.4.2 ASMD dijagram	29
3.5 Implementacija bloka Eval Light/Dark Pawn	29
3.5.1 Definisanje interfejsa	29
3.5.2 Projektovanje Controlpath modula	30
3.5.3 Projektovanje Datapath modula	31
3.6 Implementacija bloka Material of Pieces	31
3.6.1 Definisanje Interfejsa	32
3.6.2 Projektovanje Controlpath modula	33
3.6.3 Projektovanje Datapath modula	33
3.7 Implementacija bloka Eval Light/Dark King	34
3.7.1 Definisanje interfejsa	34
3.7.2 Projektovanje Controlpath modula	35
3.7.3 Projektovanje Datapath modula	36
3.8 Implementacija bloka Adder	37
3.8.1 Definisanje Interfejsa	37
3.8.2 Projektovanje Controlpath modula	38

3.8.3	Projektovanje Datapath modula	38
3.9	Integrisanje u sistem i merenje performansi	40
Glava 4	Funkcionalna Verifikacija projektovanog IP bloka	43
4.1	UVM metodologija i SystemVerilog	45
4.2	Projektovanje verifikacionog okruženja za dizajnirani IP blok	47
4.2.1	Projektovanje sekvenci i sekvencera	47
4.2.2	Projektovanje drajvera	49
4.2.3	Projektovanje monitora	49
4.2.4	Projektovanje agenta	50
4.2.5	Projektovanje skorborda	50
4.2.6	Projektovanje modula za skupljanje pokrivenosti	51
4.2.7	Projektovanje okruženja	51
4.2.8	Projektovanje top modula i povezivanje sa IP modulom	52
4.3	Testovi i skupljanje pokrivenosti	52
Glava 5	Zaključak	54
Dodatak A	Kodovi	55
Dodatak B	Listing koda	56
Literatura	64

Slike

Slika 1 (Poređenje Tradicionalne i ESL metodologije)	4
Slika 2 (Format unosa pozicije)	6
Slika 3 (Prikazuje rezultate profajliranja korišćenjem CodeBlocks alata)	7
Slika 4 (ESL taksonomija primenjena na SystemC)	9
Slika 5 (Funkcionalna blok šema paralelne realizacije modula eval)	13
Slika 6 (Problem implementacije algoritma u hardveru)	15
Slika 7 (Struktura Control Path-a i Data Path-a)	16
Slika 8 (Grafik dijagrama stanja i ASM dijagrama)	18
Slika 9 (Statička vremenska analiza za najbolji i najlošij slučaj)	19
Slika 10 (Format liste koja opisuje poziciju)	20
Slika 11 (Ideja rada projektovanog IP bloka)	21
Slika 12 (Blok dijagram realizovanog IP bloka eval)	22
Slika 13 (ASMD dijagram Select Piece modula)	25
Slika 14 (Rad Select Piece modula na primeru pešaka)	27
Slika 15 (ASMD dijagram modula rank_pawn)	29
Slika 16 (ASMD dijagram modula eval_light_pawn)	30
Slika 17 (Datapath modula eval_white_pawn)	31
Slika 18 (ASMD dijagram modula material_of_pieces)	33
Slika 19 (Datapath modula material_of_pieces)	34
Slika 20 (ASMD dijagram modula w_eval_king)	35
Slika 21 (Datapath modula w_eval_king)	37
Slika 22 (ASMD dijagram modula adder)	38
Slika 23 (Datapath modula adder)	39
Slika 24 (Blok dijagram sistema u Vivadu)	40
Slika 25 (Statička vremenska analiza urađena u Vivadu)	41
Slika 26 (Simulacija projektovanog IP bloka)	41
Slika 27 (Tok verifikacije)	43
Slika 28 (Princip verifikacije zasnovan na zlatnim vektorima i referentnom modelu)	44
Slika 29 (Tipična struktura UVM testbenča)	46
Slika 30 (Funkcionalna pokrivenost)	52
Slika 31 (Pokrivenost dela grupe square_value_cg)	53
Slika 32 (Strukturna pokrivenost)	53

Listing koda

Listing Koda 1	11
Listing Koda 2	42
Listing Koda 3	47
Listing Koda 4	48
Listing Koda 5	49
Listing Koda 6	50

Glava 1

Sadržaj diplomskog rada

Uvod

Ovaj diplomski rad se bavi projektovanjem hardverskog bloka za ubrzavanje šahovskog programa prvobitno napisanog u C programskom jeziku. Izvorni kod za šahovski program, korišćen u ovom radu se može naći na web sajtu datom u Literaturi pod [1]. Osnovna ideja je bila da se iskoristi paralelna priroda problema nalaženja šahovskog poteza i njena implementacija u hardveru čime bi se ubrzalo softversko sekvencijalno izračunavanje.

Glava 2 – se bavi projektovanjem na sistemskom nivou. Ovo obuhvata Particionisanje na hardverski i softverski deo, kao i analizu performansi sistema. Korišćena je ESL metodologija, a sama izvršna specifikacija projektovana je u SystemC jeziku za modelovanje sistema.

Glava 3 – se bavi projektovanjem samog IP bloka i diskusiju mogućih implementacija. Realizacija je zasnovana na RT metodologiji, pri čemu je akcenat stavljen na *datapath* što je proizišlo iz potrebe za većim iskorišćenjem paralelizma i razdvajanjem funkcionalnosti na više blokova čime se dobilo na modularnosti dizajna. IP blok je realizovan korišćenjem VHDL jezika za opis hardvera i Vivado razvojnog okruženja.

Glava 4 - se bavi funkcionalnom verifikacijom razvijenog IP bloka zasnovanom na UVM metodologiji u System Verilog jeziku i simulacionom alatu QuestaSim. Samo verifikaciono okruženje zasnovano je na principu referentnog modela zasnovanog na izvornoj C implementaciji, pri čemu je izvršena randomizacija ulaza u skladu sa pravilima šaha.

Predgovor

U prošlosti je obrada podataka i izvršavanje programa bilo zasnovano na radu jednog procesora sa datim setom instrukcija. Unapređivanje performansi dobijano je razvojem novih tehnologija poput CMOS-a i podizanjem učestalosti takta. Svaki par godina se učestanost povećavala skoro dva puta i ovaj trend je trajao do početka dvehiljaditih. Nakon toga je zbog velike potrošnje i interferencije pri visokim učestanostima bilo nemoguće unapređivati performanse samo na ovaj način. Dalji razvoj hardvera omogućio je uvođenje više jezgarnih procesora i procesora projektovanih za specijalnu namenu sa prilagođenom arhitekturom čime je bilo moguće unaprediti performanse i propusnu moć. Ovo je sa druge strane donelo nove probleme kao što su problemi u pogledu komunikacije, očuvanja validnosti podataka i sinhronizacije. Sa novim izazovima pojavile su se i potpuno nove oblasti kako u teoriji tako i u industriji i nove metodologije, koje su omogućavale siguran i brz put u projektovanju elektronskih sistema.

ESL (*Electronic System Level*) metodologija je primer upravo ovakve metodologije, koja obezbeđuje paralelan razvoj hardvera i softvera, kao i analizu budućih performansi, koje u značajnoj meri mogu doprineti u donošenju inženjerskih odluka.

Pre uvođenja ove metodologije uobičajen način projektovanja elektronskih sistema bio je da se prvo projektuje hardver, pa da se potom na gotovom hardveru piše softverski deo. Ovakav pristup odlikovao se dugačkim vremenom razvoja (*eng. time to market*) pošto bi softverski inženjeri bili u stanju da krenu svoj deo posla tek kada hardver bude gotov. Druga velika mana bila je mala fleksibilnost hardvera, pri čemu nije bilo moguće naknadno rekonfigurisanje hardverskih komponenti tako da budu pogodnije za softverske operacije. ESL metodologija izašla je na kraj sa ovakvim problemima.

Za razliku od tradicionalnog pristupa ESL metodologija je zasnovana na sukcesivnom profinjavanju modela počevši od algoritamskog nivoa i postepenim spuštanjem do same Gate Level Net-liste koja predstavlja i poslednju kariku u lancu. Nakon ovoga jedino još ostaje izrada lejauta (*eng. layout*), koje neka firma za izradu čipova može uraditi za Vas.

Najvažniji korak u ESL metodologiji je Particionisanje. U ovom koraku je potrebno definisati koji delovi aplikacije će se izvršavati u hardveru, a koji u softveru. Tendencija je da se većina funkcionalnosti izvršava u softveru, zbog veće fleksibilnosti, jeftinijeg otklanjanja grešaka i bolje podrške alata. Propusti načinjeni u hardveru su izuzetno skupi, jer za sada nije moguće dinamički rekonfigurisati hardver i odkloniti načinjene greške. Umesto toga potrebno je ponovo fabrikovanje čipova, čija cena se meri milionima dolara. Jedino delove koji su kritični po performanse, količinu resursa ili potrošnju treba implementirati u hardveru. U embeded sistemima, sama činjenica da se aplikacija mora izvršiti pri velikim ograničenjima, stavlja veliki značaj na hardverske komponente, koje su nerazdvojni deo kompleksnih modernih sistema.

Za projektovanje hardvera koristi se RT (*eng. Register Transfer*) metodologija koja predstavlja formalizovan postupak projektovanja digitalnog elektronskog sistema koji implementira izabrani algoritam. Drugim rečima, pomoću RT metodologije

moгуće je proizvoljni algoritam prevesti u odgovarajuće digitalno elektronsko kolo, odnosno izvršiti hardversku implementaciju algoritma [1]. Ova metodologija se zasniva na mapiranju promenjivih u registre i mapiranje operacija na odgovarajuće hardverske funkcionalne jedinice, koje zajedno sa registrima formiraju *datapath* modul. Sekvencijalno izvršavanje operacija unutar algoritma prevodi se u sekvencu transformacija i premeštanja podataka (RT operacija) koju specificira konačni automat, koji zapravo predstavlja *controlpath* modul [1]. Motivacija na korišćenju ove metodologije leži na jasnoj implementaciji algoritma i mogućnostima optimizacije na nivou sinhronih elektronskih sistema.

Moderni elektronski sistemi su izuzetno velike složenosti i gotovo je nemoguće dobiti uređaj koji besprekorno radi iz prvog puta. Da bi se smanjio broj grešaka i ubrzao razvoj samog hardvera, neophodno je greške što pre pronaći i shvatiti njihov uzrok. Ovaj posao upravo radi verifikacija hardvera. Ovo je proces koji se odvija paralelno sa razvojem dizajna i neprestano proverava njegovu funkcionalnost. Na ovaj način se garantuje ispravnost većine ključnih funkcionalnosti dizajna. Cena pronađene greške u procesu verifikacije je nekoliko redova veličine manja nego cena grešaka pronađenih u kasnijim fazama, zbog čega je u razvoju modernih sistema veliki akcenat stavljen na baš ovu oblast.

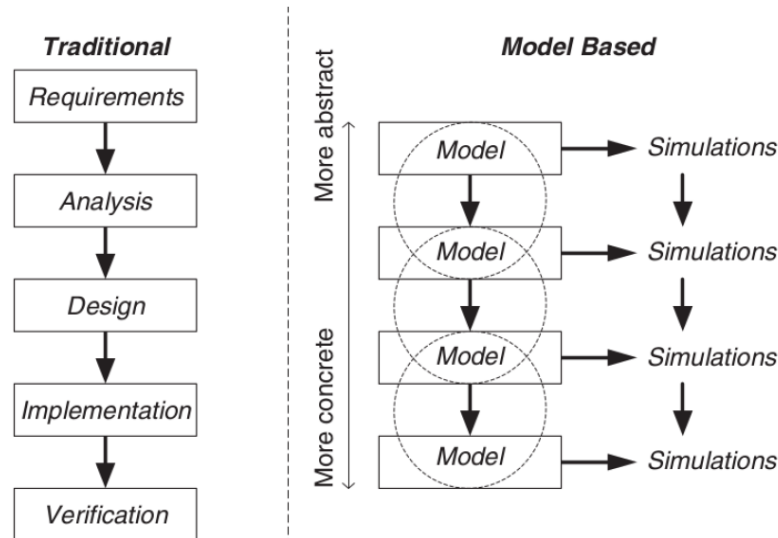
Ipak važno je napomenuti da dokazivanje kompletne ispravnosti za sada nije moguće, jer bi zahtevala pretragu svih stanja dizajna, kojih u modernim sistemima ima izuzetno mnogo. Nadu za kompletnu verifikaciju pruža nam Formalna verifikacija, koja je zasnovana na dokazivanju tvrđenja, umesto na generisanju stimulusa i proveravanje rezultata aktuelne Funkcionalne verifikacije. Nažalost, oba načina verifikovanja su zavisna od mašina na kojima se izvršavaju, i koje same po sebi uvode nova ograničenja.

Glava 2

Projektovanje aplikacije na sistemskom nivou

2.1 ESL Metodologija

ESL metodologija je usmerena na sukcesivno profinjavanje modela od najapstraktnijeg algoritamskog do najnižeg tranzistorskog. Sa druge strane tradicionalna metodologija projektovanja elektronskih sistema zasniva se na modelu vodopada gde se koraci izvršavaju samo u jednom smeru (Slika 1).



Slika 1 (Poređenje Tradicionalne i ESL metodologije)

Glavna motivacija za ESL je neuspeh tradicionalnih metodologija u rešavanju izazova složenih sistema. Ovo su samo neki od primera [3] :

- Preko 70% dizajnova nije ispunjavalo očekivanja za više od 30%,
- Preko 30% dizajnova nije ispunjavalo očekivanja za više od 50%,
- Svaki drugi projekat imao je prosečno kašnjenje od oko 4 meseca,
- Oko 13% započetih projekata je bivalo otkazano.

Prednost ESL metodologije leži u mogućnosti za raniji razvoj softvera, mogućnost procene performansi sistema u ranoj fazi razvoja, mogućnost uzajamnog prilagođavanja softvera i hardvera, kao i njihovo analitičko particionisanje. Stvaranjem modela sistema postiže se jasno definisanje zahteva i podela posla između komponenti. Ovim se postiže veći determinizam i moguće je postići viši stepen optimizacije imajući u vidu širu sliku sistema koji projektujemo. Testiranje sistema zasnovano na ko-verifikaciji, koja se izvršava nekoliko redova veličine brže nego na RTL nivou, a sa kojim možemo doći do važnih zaključaka o potencijalnim problemima u sistemu i mestima koja predstavljaju uska grla.

U ESL metodologiji sve je u vezi sa modelovanjem na određenom nivou apstrakcije. Posao sistema arhitekta je da obezbeđuje različite vrste modela i kordiniše softverskom i hardverskim timom. Pored ovoga od sistem inženjera se očekuje da demonstrira rad sistema, analizira aspekte sistema, izabere IP blokove i particioniše sistem. Ovaj posao nije nimalo jednostavan i zahteva dosta iskustva. Kordinaciju hardverskog i softverskog tima otežava još i činjenica da je cena alata drastično različita za hardver i softver, a potrebno je obezbediti kompatibilnost.

Tokom razvoja korišćenjem ESL metodologije projekat prolazi kroz sledeće korake:

1. **Specifikacija i Modelovanje** – u ovom koraku je potrebno napraviti specifikaciju sistema. Preferirani oblik specifikacije je izvršna specifikacija jer ona pruža mogućnosti za analizu, verifikaciju, procenu performansi u značajno većoj meri od tradicionalne specifikacije na papiru.
2. **Pre-Particionisanje** – ovo je korak u kome se vrše prve analize koje imaju ulogu u donošenju odluka u Particionisanju. Postoje statička, dinamička, algoritamska, analiza kompleksnosti i analiza platformi. Statička analiza koristi već razvijene postupke za utvrđivanje kompleksnosti, a najpoznatije su “-ility “ analize (*reliability, maintainability, usability, criticality*). Dinamička i algoritamska analiza se sa druge strane bave procenom tereta izračunavanja, komunikacije i potrošnje.
3. **Particionisanje** – ovaj korak se sastoji iz tri manja koraka. Prvo se identifikuju funkcije iz specifikacije, potom se odredi ciljana arhitektura i na kraju se izvrši mapiranje funkcionalnosti na arhitekturu uz definisanje interfejsa. Određuje se koji procesor ili DSP se koristi, koji operativni sistem, koje biblioteke uključuje, koje drajvere poseduje, koje aplikacije... Od ovog koraka zavisi najviše kako će sistem funkcionisati i zbog toga je važno da se dobije što više informacija iz prethodnih koraka, kako bi se donele najpovoljnije odluke u particionisanju.
4. **Analiza Posle Particionisanja** – u ovom trenutku imamo više informacija o samoj implementaciji i moguće je izvršiti više analiza. Najčešće analize u ovom koraku su funkcionalna, analiza performansi, interfejsa, potrošnje, površine, cene i mogućnosti debaga.
5. **Verifikacija Posle Particionisanja** – ovaj korak treba da pokaže da je sistem zadržao sve funkcionalnosti koje je imao i pre particionisanja.
6. **Implementacija Hardvera i Softvera** – u ovom koraku svaki tim obavlja svoj deo posla
7. **Verifikacija Implementacije** – ovo je ujedno i poslednji korak, koji treba da pokaže ispravnost realizovanog hardvera. Zasnovana je na proveru tvrdnji, kao i na generisanju stimulusa i upoređivanju odgovora. Potrebno je i omogućiti debug i posle fabrikacije, što se ostvaruje specifičnim kolima ugrađenim u sam dizajn sa ulogom da prikupljaju signale iz dizajna. Primer ovoga je JTAG (eng. *Joint Test Action Group*), ILA (eng. *integrated logic analyzer*), logika za praćenje.

Kao što je već pokazano Slikom 2.1 put ka konačnoj implementaciji nije uvek pravolinijski i ponekad je potrebno vratiti se korak u nazad, kako bi smo stvorili povoljnije mogućnosti za realizaciju sistema.

2.2 Specifikacija i izvorni kod šahovskog programa

Šahovski program zasniva se na realizaciji Toma Kerrigana (*Tom Kerrigan*), koja se može pronaći na sajtu datom u [1], u C programskom jeziku. Kako sam autor navodi, ideja ovog programa nije bila da se napravi optimalan šahovski program, već da se zainteresovani čitalac lako upozna sa konceptima na kojima je zasnovano programiranje šahovskih endžina. Sam kod sastoji se iz nekoliko delova. U osnovnoj main funkciji vrši se parsiranje i dekodovanje komande unete preko terminala u beskonačnoj *while* petlji. Korisnik na raspolaganju ima sledeće mogućnosti, dok su poslednje dve implementirane u svrhu ovog rada.

- on - računar igra sedeći potez,
- off - računar prestaje da igra,
- st n – računar traži potez n sekundi,
- sd n – dubina predraživanja,
- undo – potez unazad,
- new – nova partija,
- upis poteza u notaciji- polje sa kog se igra, polje na koje se igra, pr. e2e4,
- run n – program automatski odigrava n poteza,
- pos – korisnik definiše poziciju koju želi da postavi na tablu, gde je format popunjavanja kao na slici

```

      P0 P1 P2 P3 P4 P5 P6 P7 N0 N1 B0 B1 K  Q0 R0 R1
WHITE: a3 b2 c2 d2 X  X Qg2 h2 b1 g1 c1 f1 e1 d1 a1 h1
BLACK: a7 b7 c7 d7 e7 X X X a6 g8 c8 f8 e8 d8 a8 h8
Play side (w/b) : w

```

Slika 2 (Format unosa pozicije)

Ukoliko korisnik unese komandu da računar treba da igra aktivira se funkcija **think**. Ova funkcija proverava prvo da li je partija u stanju otvaranja i ako jeste bira sledeći potez u nizu za dato otvaranje definisano u knjizi otvaranja. Ukoliko sled poteza ne odgovara ni jednom iz knjige otvaranja, započinje se obilaženje stabla po principu alfa-beta pretrage [4], realizovanoj u funkciji **search**. Ova funkcija zajedno sa funkcijom **quiesce** predstavlja suštinu funkcionisanja programa. U njima je implementiran negamax algoritam [5], koji je varijanta minimax algoritma [6], pri čemu se podrazumeva da je

max(a, b) = - min(-a, -b). Drugim rečima vrednost koja odgovara poziciji igrača A jednaka je negaciji vrednosti koja odgovara poziciji igrača B. Ovakav pristup uprošćuje sam minimax algoritam i omogućava da se pozicija za belog i pozicija za

crnog ustvari mogu opisati istim brojem. Pored ovih funkcija implementirane su još mnoge, ali je cilj za sada samo da se shvati osnovna ideja funkcionisanja programa, bez ulaženja u detalje implementacije.

Preuzimanjem šahovskog programa sa interneta na neki način je zaobiđen prvi korak u ESL metodologiji – Specifikacija i Modelovanje. Ovaj način realizacije sistema poznat je kao *middle out* pristup, što drugim rečima znači da se projekat ne kreće od nule, već da se za početak koristi neka poznata implementacija, a sam projekat ima za cilj da tu implementaciju proširi i unapredi. U narednim koracima ovaj izvorni kod će ustvari predstavljati izvršnu specifikaciju, jer je cilj ovog projekta da funkcionalnost ostane ista, a da se izračunavanje potrebno za nalaženje poteza značajno ubrza.

2.3 Analiza pre particionisanja

U ovom koraku je potrebno izvršiti analize specifikacije, koje imaju za cilj da nam daju odgovore koje će nam pomoći da efikasnije uđemo u korak particionisanja. U ovom slučaju urađeno je profajliranje izvornog koda u želji da se odrede funkcije koje oduzimaju najviše vremena prilikom izvršavanja. Profajliranje je urađeno uz pomoć CodeBlocks profajlera koji je dao sledeće rezultate (Slika 3)

1	Flat profile:						
2							
3	Each sample counts as 0.01 seconds.						
4	%	cumulative	self	self	total		
5	time	seconds	seconds	calls	Ts/call	Ts/call	name
6	29.18	148.81	148.81				attack
7	26.38	283.32	134.51				eval
8	12.25	345.81	62.48				set_hash
9	9.27	393.10	47.28				gen_caps
10	6.06	424.00	30.91				in_check
11	3.33	440.96	16.96				sort
12	3.02	456.37	15.40				gen
13	2.23	467.74	11.37				eval_dark_pawn
14	2.11	478.48	10.74				eval_light_pawn
15	1.13	484.23	5.75				gen_push
16	0.91	488.86	4.63				makemove
17	0.72	492.54	3.68				takeback
18	0.63	495.75	3.21				eval_dark_king
19	0.61	498.85	3.10				quiesce
20	0.58	501.81	2.96				eval_dkp
21	0.53	504.49	2.68				eval_light_king
22	0.47	506.87	2.38				eval_lkp
23	0.45	509.15	2.29				search
24	0.06	509.46	0.31				reps
25	0.05	509.70	0.23				gen_promote
26	0.02	509.81	0.11				checkup
27	0.02	509.89	0.09				hash_rand
28	0.01	509.93	0.04				sort_pv

Slika 3 (Prikazuje rezultate profajliranja korišćenjem CodeBlocks alata)

Na osnovu rezultata profajliranja može se zaključiti da funkcije **attack** i **eval** zauzimanju značajno veći udeo u ukupnom vremenu izvršavanja od ostalih funkcija.

Funkcija **attack** određuje da li je određeno polje napadnuto od strane belog ili crnog igrača. Ova funkcija je implementirana jednom for petljom koja prolazi kroz sva

polja i proverava da li figura zadate boje može da dođe do zadatog polja. Kretanje figure definisano je nizom ofseta koje svaka figura ima.

Sa druge strane funkcija **eval** bavi se procenom pozicije. Koliko je pozicija povoljna po belog ili crnog igrača zavisi od više faktora. Svaka figura osim kralja ima svoju stalnu vrednost nezavisno gde se nalazi. Pored toga pešak, skakač i lovac imaju svoje dodatne vrednosti u zavisnosti od polja na kom se nalaze. Vrednost topova zavisi još i od prijateljskih i suparničkih pešaka koji se nalaze na njegovoj koloni. I na kraju konačnoj proceni doprinosi i koliko je kralj zaštićen u odnosu na to sa kolikim materijalom protivnik raspolaže.

2.4 Particionisanje

U ovom koraku potrebno je podeliti funkcionalnosti na hardverski i softverski deo, kao što je ranije rečeno. Pri ovom koraku preporuka je da se sve što može izvršava u softveru dok se samo stvari kritične po performanse izvršavaju u hardveru. U našem slučaju, logika za pretraživanje stabla i izvršavanje negamax algoritma je idealan primer za strukturu koja treba da se izvršava u softveru. Ovo je kontrolni deo koda koji ne zauzima sam po sebi mnogo procesorskog vremena, što se može videti iz rezultata profajliranja. Sa druge strane funkcije **attack** i **eval** su funkcije evaluativnog karaktera koje su statičke po prirodi i samim tim se dobro mogu mapirati u hardver. Da stvar bude još jasnija upravo ove funkcije zauzimaju najveći udeo u izračunavanju, čime se automatski svrstavaju u kandidate za hardversku akceleraciju.

Funkcija **eval** je značajno kompleksnija od funkcije **attack**, što drugim rečima znači i da raspolaže sa većom mogućnošću za optimizacijom. Iz ovog razloga je funkcija **eval** određena da bude implementirana RT metodologijom, dok je funkcija **attack** ostavljena da se uradi sintezom visokog nivoa.

Nažalost, usled nedostatka vremena, funkcija **attack** ipak nije implementirana u ovom radu, u hardveru. Autor ovog rada se ipak nada da će ovaj deo posla biti završen u budućnosti, čime bi ovaj rad bio potpuniji.

Korak particionisanja takođe definiše i ciljanu platformu na kojoj se aplikacija izvršava i u ovom radu izabrana je Zybo ploča [7], koja pripada familiji Zynq-7000.

Interfejs koji treba da bude definisan za **eval** modul se zapravo sastoji od parametara koje obrađuje **eval** funkcija. Iako se iz deklaracije same funkcije ne vidi koji su ulazni parametri, oni su definisani kao globalne promenjive koje opisuju poziciju i stranu koja je na potezu.

Preciznije rečeno to su sledeće promenjive:

- `int color[64]`
- `int piece[64]`
- `int side`

Nizovi color i piece opisuju svako od polja na tabli, pri čemu color dobija vrednost '0'- ako je figura na polju bela, '1' – ako je figura na polju crna i '6' ako nema figure na polju. Niz color dobija vrednost '0'- za pešaka, '1'- za skakača, '2' – za lovca, '3' – za topa, '4' – za kraljicu i '5' za kralja, dok ako nema figure takođe dobija vrednost '6'.

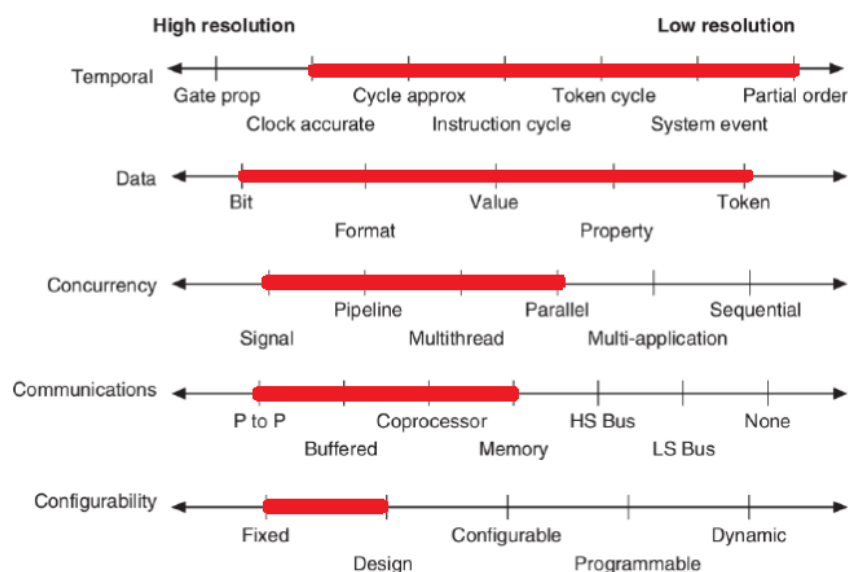
Iz prethodno prikazanog može se primetiti da je opseg mogućih vrednosti svakog polja značajno manji nego što su definisani. Sa stanovišta softvera ovo je opravdano, jer se programi obično izvršavaju na 32-bitnim ili 64-bitnim mašinama, pa je alokacija i najmanjeg polja zauzima 32 bita. U svetu hardvera jasno je da se ove promenjive moraju opisati sa što manje bita, jer se svaki bit interfejsa u hardveru prevodi u žicu, koja povećava cenu samoj implementaciji.

Imajući ovo u vidu, interfejs hardverskog modula će imati jedan bit za stranu koja je na potezu, dok će članovi nizova color i piece biti opisani sa po jedan bit za boju figure na određenom polju i tri bita za određivanje figure, dok se prazno mesto opisuje isključivo preko piece niza.

Što se tiče izlaza, on predstavlja zapravo povratnu vrednost funkcije eval i biće implementiran sa 15 bita, gde tip *signed* a opseg koji je obuhvaćen (-16384, 16384). Ovaj opseg odabran je zato što se rezultat funkcije eval kreće u opsegu od (-10000, 10000).

2.5 Implementacija SystemC modela posle particionisanja

Nakon što je razdvajanje funkcije eval od ostatka sistema i definisanje interfejsa učinjeno, potrebno je modelovati zajedno oba modula, pri čemu treba imati u vidu potrebu za konkurentnošću i komunikacijom, koja izostaje kod C programskog jezika. Za potrebe modelovanja sistema je zbog toga odabran SystemC, koji sadrži mehanizme signala, metoda i niti koje se paralelno izvršavaju, kao i finiju preciznost podataka na nivou bita. Mogućnosti Sistem C jezika su date na Slici 4.



Slika 4 (ESL taksonomija primenjena na SystemC)

Da bi se razdvojili funkcija eval i ostatak koda, potrebno je definisati dve klase koje obe javno nasleđuju baznu klasu svih komponenti – **sc_module**. Ovo je urađeno korišćenjem makroa `SC_MODULE()`, koji javno nasleđuje klasu `sc_module` i kao parametar prima ime klase. Definisanjem klase na ovaj način imamo mogućnost da koristimo konkurentno izvršavanje metoda, odnosno SystemC procesa. Da bi neka metoda bila proglašena konkurentnom, potrebno je prvo da klasa bude registrovana makroom `SC_HAS_PROCESS()`. Konkurentna metoda ne prima argumente i nemapovratnu vrednost. Da bi se proglasila konkurentnom može se koristiti sledeći makroi:

- `SC_THREAD`
- `SC_METHOD`
- `SC_CTHREAD`

Korišćenjem makroa `SC_THREAD` dobija se SystemC proces koji se izvršava paralelno i može sadržati wait naredbe koje blokiraju izvršavanje za u određenom vremenskom intervalu. `SC_METHOD` nema mogućnost korišćenja wait funkcije i mora da se izračunava u jednom simulacionom trenutku, što se koristi se za modelovanje kombinacionih mreža. `SC_CTHREAD` procesi poseduju koncepte takta i koriste se za sintezu hardvera. SystemC proces definisan na bilo koji od ova tri načina može da ima listu osetljivosti (*eng, sensitivity list*) na koji se on pokreće. Podrazumevano je da svi procesi pokreću na početku simulacije, osim onih obeleženih sa `dont_initialize()`.

U ovom radu, modelovanje korišćenjem SystemC-a imalo je za cilj da demonstrira rad budućeg hardvera na nivou transakcija i da posluži za prvu procenu performansi. Zbog ovoga korišteni su `SC_THREAD` procesi. U soft modulu izvorna main funkcija proglašena je `SC_THREAD`-om, sa novim nazivom – `soft_main`. U eval modulu su sve pomoćne funkcije koje se pozivaju u osnovnoj eval funkciji, implementirane preko procesa, a njihova komunikacija zasniva se na signalima. Naime, kada jedan proces obradi određene podatke on aktivira sledeći proces koji koristi dobijene rezultate. Ovo je prvi korak u spuštanju apstrakcije sa algoritamskog nivoa na hardver.

Povezivanje eval modula sa ostatkom koda realizuje se u **top** modulu čiji je hpp sadrži deklaracije signala preko kojih interreaguju eval modul i ostatak softvera dok je u cpp fajlu u konstruktoru dato povezivanje signala (Listing koda[1] i [2]). Iz ovog listinga može se videti da soft modul šalje signale o trenutnoj poziciji, strani koja je na potezu i start signal, a prima dva rezultata, i signal da je eval modul završio obradu. Ova dva rezultata se odnose na prvobitnu realizaciju eval funkcije, dok se drugi odnosi na rezultat paralelne implementacije, koja će biti realizovana u hardveru. Oba rezultata se vraćaju da bi se mogla izvršiti provera korektnosti modula koji projektujemo.

Da bi se signali poslali iz modula potrebno ih je mapirati na portove definisane kao `sc_in<tip> ime_ulaznog_porta`, odnosno `sc_out<tip> ime_izlaznog_porta`, koji su definisani u hpp fajlovima eval i soft modula. Signali se mapiraju na portove koristeći metode **write** i **read** koje vrednosti signala dodeljuju datim portovima.

Ovo se dešava na mestima gde se u C kodu pozivala funkcija eval(), umesto čega u SystemC modelu imamo sledeći deo koda:

Listig koda za komunikaciju sa eval modulom iz search.cpp fajla

```
for(int i = 0; i < 64; i++)
{
    soft_color[i]->write(color[i]);
    soft_piece[i]->write(piece[i]);
}
soft_side->write(side);
start_out = 1;

std::cout << "START time = " << sc_core::sc_time_stamp()
<< std::endl;
wait();
std::cout << "END time = " << sc_core::sc_time_stamp() <<
std::endl;

start_out = 0;
wait(1, SC_NS);

printf("***** Eval passed | SEQ = %d | PAR = %d
*****\n", soft_eval_in->read(), res_par->read());

if(soft_eval_in->read() != res_par->read())
    sc_stop();

return res_par->read();
```

Listing Koda 1

Nakon što se vrednosti signala proslede portovima, izvršavanje se zaustavlja na wait funkciji pri čemu se zabeležava trenutno vreme izvršavanja. Soft modul ne nastavlja da se izvršava sve dok se ponovo ne pokrene dobijanjem signala start_in od eval modula. Ovime eval potvrđuje da je završio sa radom i soft modul nastavlja izvršavanje, pri čemu se beleži novo simulaciono vreme nastavka izvršavanja. Razlikom vremena slanja signala i prijema odgovora dobijamo procenu o vremenu izvršavanja eval modula. Ipak važno je naglasiti da ovo nije nikako tačno vreme izvršavanja u hardveru, već samo gruba procena koja bi trebala samo da nam da pojam o brzini izvršavanja. U nastavku se rezultati dobijeni izvornom implementacijom eval funkcije i modelovanom paralelnom implementacijom porede i u koliko su različite simulacija se zaustavlja.

2.6 Implementacija eval modula u SystemC-u

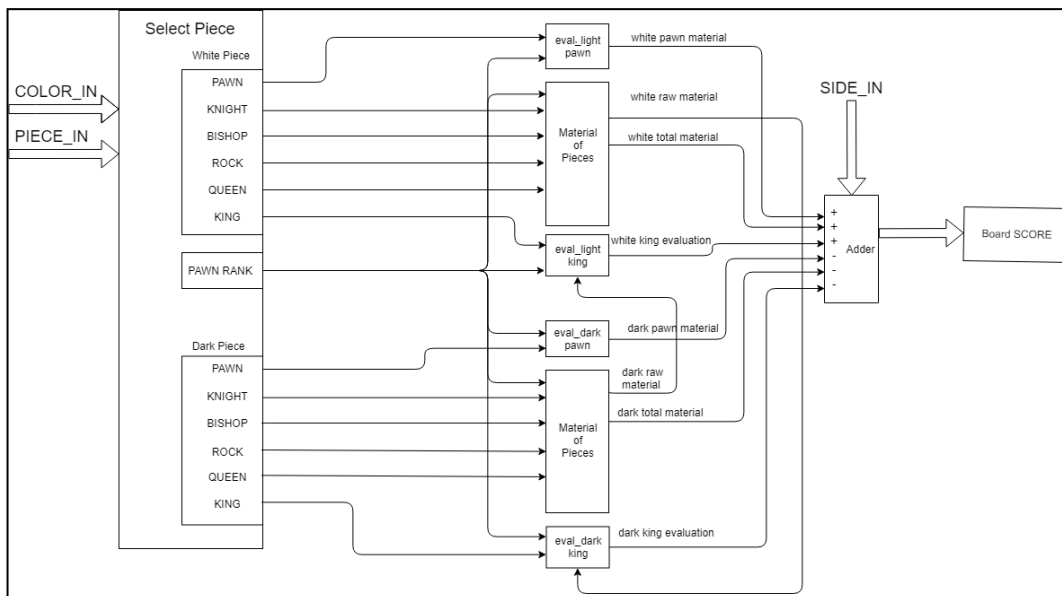
Kao što je već rečeno funkcija eval računa vrednost pozicije u odnosu na igrača koji je na potezu. Ova funkcionalnost je u izvornom kodu urađena na sledeći način

1. Korak je utvrđivanje Rank-a pešaka. Ovo je niz od 8 vrednosti za belog i crnog, koji za svaku kolonu određuje položaj najmanje napredovanog pešaka, odnosno njegovu vrstu. Dodatno se još određuje ukupna stalna vrednost svih figura. Da bi se ovaj korak izvršio potrebno je proći kroz sva polja, odnosno 64 prolaza kroz *for* petlju.
2. Korak je ponovni prolaz kroz sva polja i određivanje vrednosti figura zavisne od polja na tabli (određenim matricama pcsq definisanim za pešaka, skakača, lovca i kralja). U koliko je figura u polju:
 - Pešak – poziva se funkcija eval_light(dark)_pawn, koja uračunava vrednost iz matrice pawn_pcsq sa datog polja i dodaje negativne poene u koliko su pešaci udvojeni, izolovani, zaostali u odnosu na rank susednih kolona, odnosno pozitivne ako pešak prošao iza susednih protivnikovih pešaka. Upravo zbog ovoga je prvo izračunat Rank pešaka.
 - Skakač i Lovac – dodaje se vrednost iz odgovarajućih pcsq matrica za dato polje
 - Top - dodaju se bonus poeni u koliko kolona nema pešaka, za šta se koristi Rank i u koliko se top nalazi na sedmom redu za belog, odnosno drugom redu za crnog
 - Kraljica – ništa se ne dodaje jer vrednost kraljice ne zavisi od pozicije
 - Kralj – u koliko je stalna vrednost protivničkih figura manja od 1200, dodaje se vrednost iz king_endgame_pcsq matrice za dato polje, a ako je veća poziva se funkcija eval_light(dark)_king, koja određuje koliko je kralj zaštićen na osnovu položaja njegovih i suparničkih pešaka ispred njega, takođe definisano Rankom, i stalne vrednosti protivničkih figura.

Ovakva realizacija prolazi dva puta kroz sva polja i dodatno vreme se još troši za funkcije koje se pozivaju nailaženjem na figuru sa datog polja. Kako bi se iskoristilo paralelizam koje hardver pruža potrebno je uvideti koje međuzavisnosti postoje između instrukcija i koje se instrukcije moraju izvršiti pre drugih.

U teoriji, doprinos svih polja bi moglo da se izračunava istovremeno, ali ovakav pristup doneo bi značajne komplikacije u pogledu računanja Rank-a. To bi zapravo značilo da svako polje mora da ima informacije o poljima u susednim kolonama na kojima se nalaze pešaci. Na ovaj način bi se izračunavanje za ostale figure svakako usporilo i ne bi dobili željeno ubrzanje uprkos velikoj kompleksnosti i potrošenim resursima. Detaljnije razmatranje hardverske implementacije je dato u Glavi 3, gde se detaljno razmatra optimalna implementacija IP bloka.

Za sada je ideja samo da se prikaže osnovna ideja za paralelni dizajn eval modula. Na Slici 5 data je blok šema moguće implementacije.



Slika 5 (Funkcionalna blok šema paralelne realizacije modula eval)

Prvi blok koji započinje obradu je modul **Select Piece**, koji prima podatke COLOR_IN i PIECE_IN, koji opisuju datu poziciju (kod). Njegov zadatak je da podatke o poziciji sortira prema ostalim blokovima koji treba da izračunaju doprinos svake figure. Pošto broj figura istog tipa za svaku poziciju nije jednoznačan podaci o poljima figura se slažu u fifo registre koji su povezani na izlaz. Ovo se odnosi na sve figure osim kralja koji se prosleđuje preko običnog signala. Pored ovoga, modul bi trebao da izračuna Rank kako bi ostali delovi sistema mogli da koriste ove informacije u svojim aktivnostima U prvoj aproksimaciji ovo se dešava u jednom trenutku, nakon čega se aktivira signal *s_go* na koji čekaju ostali procesi da bi se aktivirali.

Moduli **Eval Light Pawn** i **Eval Dark Pawn** čitaju vrednosti iz fifo bafera dokle god on nije prazan i akumuliraju stalne vrednosti pešaka i vrednosti u zavisnosti od pozicije, računajući pritom i bonus poene pomenute ranije. Kada se isprazni fifo bafer, ovi moduli šalju signale modulu Adder da su završili zajedno sa akumuliranim vrednostima.

Na sličan način funkcionišu i ostali moduli. Moduli **Material Of Piece** proveravaju više fifo bafera različitih figura i u odnosu na njihov tip akumuliraju njihove stalne vrednosti vrednosti u zavisnosti od polja. Kada su svi fifo baferi prazni šalje se suma stalnih vrednosti figura, ukupna vrednost i vrednost iz matrice king_endgame_pcsq za određeno polje na kom je kralj. Suma stalnih vrednosti (raw_material) se izdvaja iz ukupne sume jer ona ima ulogu u proceni bezbednosti protivničkog kralja. Ako je manja od 1200 smatra se da nije bitno kako je kralj zaštićen, već se onda uzima u obzir njegova pozicija na tabli.

Modul Eval Light/Dark King računa zaštićenost kralja pešacima i skalira to u odnosu na jačinu protivničkih figura na tabli. Kada se završi procena, rezultat se

šalje na modul za poređene jačine protivničkih figura, kao i izlaz `king_endgame_pcsq` iz modula `Material of Pieces`, posle čega se svi rezultati sumiraju u modulu `Adder`.

`Adder` jednostavno prikuplja rezultate, kada bude poslat signal za završetak prethodnih blokova i sumiranu vrednost šalje na izlaz. `SIDE_IN` ulaz određuje suštinski samo znak rezultata, odnosno smer oduzimanja. U ovom trenutku svi prethodni blokovi se vraćaju u prvobitno stanje čekajući novi start signal.

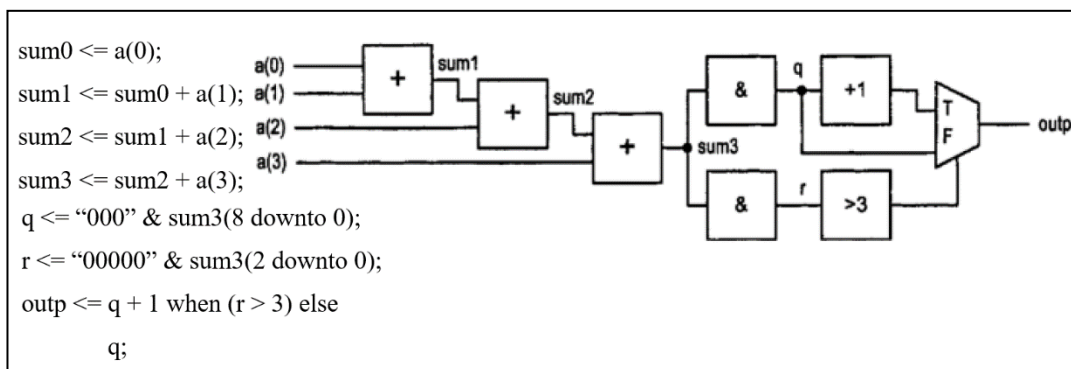
`Soft` modul takođe nastavlja gde je stao, čime se završava evaluacija pozicije. Nakon ovoga ostaje još da se uporede vrednosti paralelne implementacije i izvorne `eval` funkcije, i ako su one jednake, program nastavlja dalje sa radom.

Glava 3

Projektovanje hardverskog IP bloka

3.1 RT metodologija

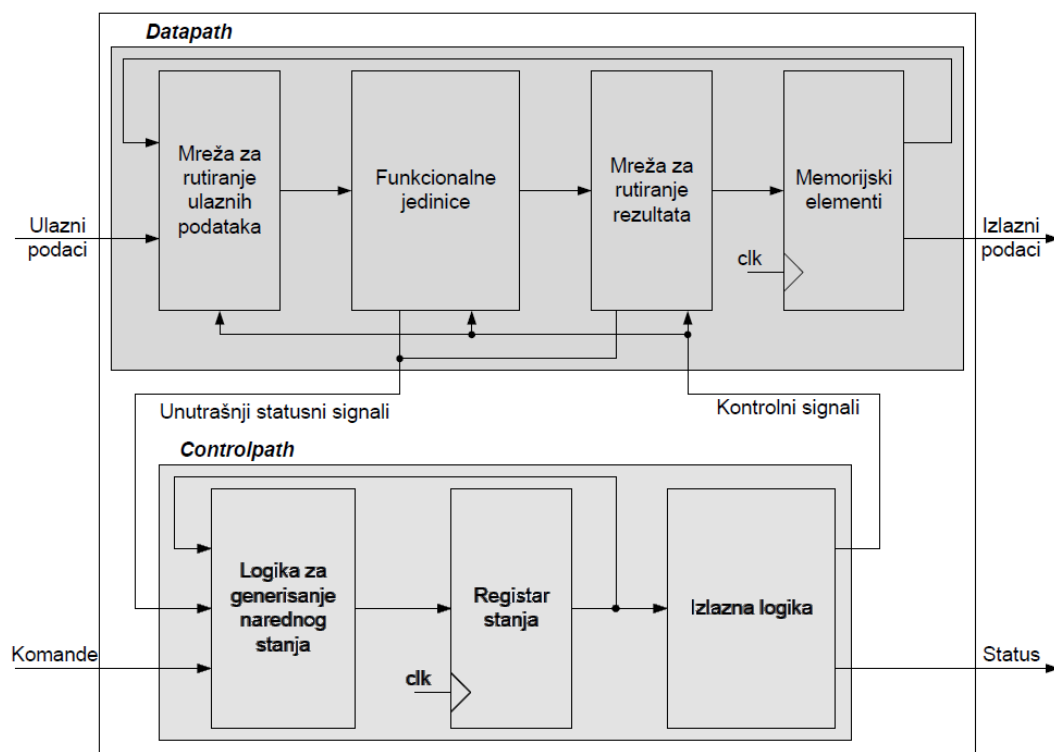
RT(*eng. Register Transfer*) metodologija je metodologija projektovanja dizajna koja opisuje operaciju sistema kao sekvencu transfera podataka i manipulaciju registara. Ova metodologija podržava varijable i sekvencijalno izvršavanje algoritma i obezbeđuje sistematičan način da se taj algoritam transformiše u hardver. Ovaj put transformacija nije nimalo jednostavan u koliko želimo da ostvarimo benefite, jer se logika sekvencijalnog izvršavanja u softveru u mnogome razlikuje od konkurentnog hardvera. Uzmimo na primer deo koda dat na Slici 6 i njegovu hardversku implementaciju.



Slika 6 (Problem implementacije algoritma u hardveru)

Na ovom jednostavnom primeru vidi se da implementacija u hardveru, za razliku od softverske implementacije, nema pojam promenjive i da se deo operacija izvršava u paraleli. Sekvencijalna logika izvršavanja je implicitno ugrađena povezivanjem komponenti redno u datapath. Lako se može uvideti da u koliko povećamo broj članova niza -a, povećava se i broj sabirača. Pored ovoga, ako bi želeli da veličinu niza menjamo dinamički naišli bi na poteškoće u implementaciji takvog hardvera, zbog njegove fiksne prirode. Jasno je da ovakav način ne može biti korišćen za kompleksnije algoritme i da je potrebna drugačija metodologija od strukturalne dataflow implementacije.

U RT metodologiji koristimo registre za smeštanje varijabli, dok se aritmetičke i logičke operacije izvršavaju posebnom kombinacionom logikom. Za razliku od predhodnog primera, RT metodologija ne sadrži samo *Data Path* već sadrži i *Control Path* koji određuje kada koja operacija treba da se izvrši. Control Path je često realizovan preko FSM (*eng. Finite State Machine*) koji koristi stanja da izvrši zadate korake, kao i uslove grananja za određivanje sledećeg stanja. Jednim imenom FSM i Data path se nazivaju FSMD i oni zajedno čine osnovni koncept RT metodologije. Na ovaj način se dobija potpuna kontrola nad dizajnom gde svako stanje FSM-a određuje ponašanje Data_path-a, dok uslovi izračunati u Data path-u određuju sledeće stanje FSM-a [2].



Slika 7 (Struktura Control Path-a i Data Path-a)

Sa Slike 7 se može videti da se digitalni sistem realizovan korišćenjem RT metodologije sastoji iz sledećih delova:

- **Datapath modula** – koji se sastoji iz sledeća tri modula:
 - **Memorijskih elemenata** – u standardnoj RT metodologiji radi se o registrima sa paralelnim ulazom i izlazom, koji služe za čuvanje svih međurezultata kao i konačnih rezultata koji se generišu tokom izvršavanja implementiranog algoritma
 - **Funkcionalnih jedinica** – koje realizuju sve operacije koje postoje u algoritmu koji se implementira. Po pravilu, funkcionalne jedinice koje se najčešće sreću su sabirači, oduzimači, množači, inkrementori, dekrementori, pomerači, komparatori, itd.
 - **Mreža za rutiranje** – služe za dovođenje izlaza odgovarajućih registara do odgovarajućih funkcionalnih jedinica, kao i za dovođenje izlaza funkcionalnih jedinica do ulaza odgovarajućih registara. Tipično su sastavljene od multipleksera, čiji selekциони ulazi su pod kontrolom *controlpath* modula.

Datapath se sadrži sledeće interfejsa:

- **Ulazni interfejs podataka** – preko koga se podaci dovode do datapath modula
- **Izlazni interfejs podataka** – preko koga se rezultati izvršavanja algoritma prosleđuju u spoljašnjem okruženju

- Kontrolni interfejs – pomoću koga controlpath modul upravlja radom datapath modula
- Statusni interfejs – preko koga se controlpath-u šalju informacije o trenutnom statusu datapath modula, na osnovu čega se donose odluke o narednom stanju
- **Controlpath modula –**
 - Logika za generisanje narednog stanja – ovaj blok određuje koje će naredno stanje biti aktuelno u FSM-u. U logici narednog stanja učestvuju trenutno stanje, ulazi u sistem i statusni signali iz datapath-a,
 - Registar stanja – u ovom registru se čuva trenutno stanje, koje često predstavlja enumerisani tip podataka, koji se sintezom prevodi u konkretne vrednosti,
 - Izlazna logika – predstavlja logiku za generisanje kontrolnih i statusnih signala. Funkcija izlazi može biti Murovog ili Milijevog tipa. U slučaju Murovih izlaza izlazi su funkcija samo trenutnog stanja, dok u Milijevom slučaju zavise i od ulaznih signala.

Interfejsi Controlpath-a

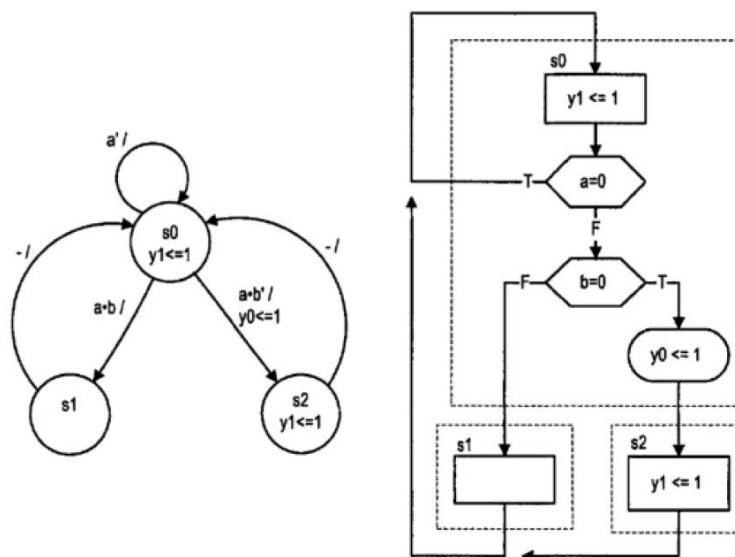
- Komandni interfejs – predstavlja ulaze preko kojih se može upravljati digitalnim sistemom (npr. start, reset),
- Izlazni statusni interfejs – preko ovog interfejsa sistem prosleđuje informacije o unutrašnjim stanjima (npr. ready, busy),
- Kontrolni interfejs – preko ovog interfejsa šalju se upravljački signali datapath-u,
- Ulazni statusni interfejs – interfejs preko kog controlpath prima informacije od datapath-a, koje učestvuju u logici narednog stanja.

Da bi se algoritam opisao preko FSM, koriste se dve grafičke reprezentacije:

1. Dijagrami stanja,
2. ASM(*eng Algorithmic State Machine*) dijagram.

Dijagrami stanja se sastoje od čvorova, obeleženih krugovima i jednosmernim strelicama koje određuju pravac tranzicije. Svaki čvor predstavlja jedinstveno stanje u FSM-u i ima jedinstveno ime, dok svaka strelica ima definisan uslov prelaza u sledeće stanje i opciono Milijeve izlaze, dok su dodele Murovih izlaza uvek zapisani u sklopu stanja.

ASM dijagrami se mogu koristiti za izražavanje kompleksnih sekvenci događaja koje uključuju ulazne komande i akcije na izlazu. Ova vrsta reprezentacije je više deskriptivna od dijagrama stanja i lako joj se može pridružiti datapath, čime se dobija ASMD dijagram. Nakon ovoga se ASMD reprezentacija može jednoznačno transformisati u VHDL. Osnovni ASM blok se sastoji od polja stanja, uslova prelaza u naredno stanje, kao i od izlaznog polja u kome se opciono definišu Milijevi signali. Primer dijagrama stanja i ASM dijagrama je dat na Slici 8.



Slika 8 (Grafik dijagrama stanja i ASM dijagrama)

Proces projektovanja digitalnog sistema korišćenjem RT metodologije treba da sadrži sledeće korake[2]:

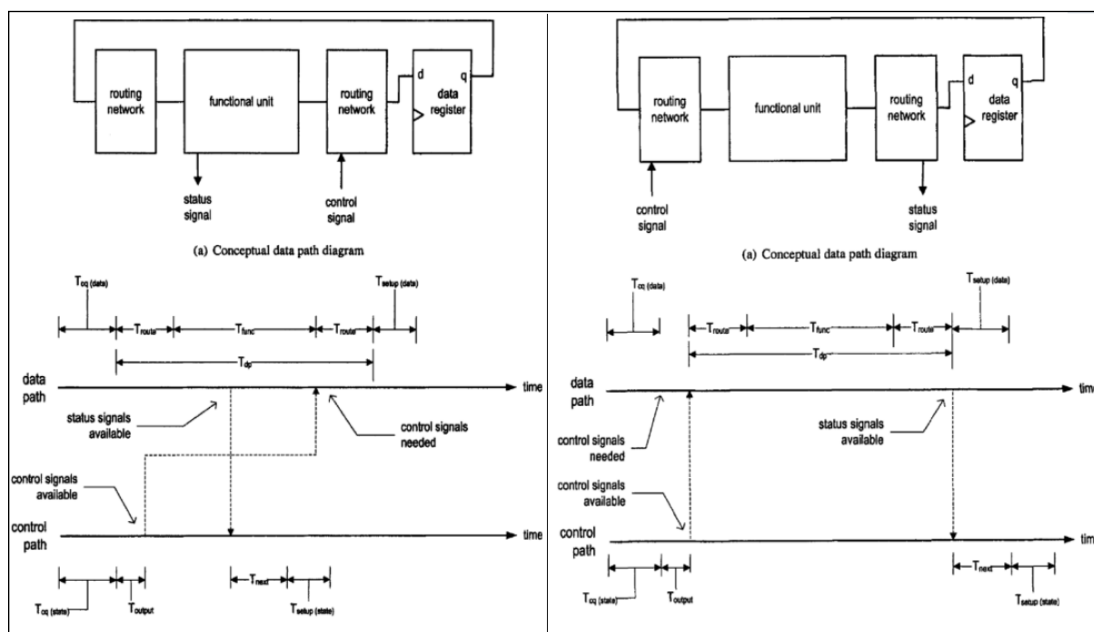
1. **Eliminacija naredbi ponavljanja** – u ovom koraku je potrebno zameniti naredbe ponavljanja u algoritmu (**for**, **repeat**, **while** petlje) sa **if-goto** naredbama. Ovaj korak potrebno je uraditi kako bi se dobila jasna implementacija ASMD dijagrama sa povratnom petljom i početnim uslovom.
2. **Definisanje interfejsa digitalnog sistema** – analizom algoritma potrebno je uočiti ulazne i izlazne promenjive u algoritmu i odrediti potrebne širine portova. Pored ovih promenjivih potrebno je ubaviti komandne i statusne interfejse sa kojim bi se moglo upravljati radom uređaja.
3. **Projektovanje controlpath modula** - na osnovu zadatog algoritma potrebno je kreirati odgovarajući ASMD dijagram algoritma. Važno je napomenuti da u ovom koraku, treba razmotriti potencijalne optimizacije u pogledu dostupnog paralelizma. Ovo je od posebnog značaja za dataflow aplikacije, kakava je upravo i tema ovog rada.
4. **Projektovanje datapath modula** – sastoji se iz četiri koraka:
 - 4.1. Identifikacija unutrašnjih promenjivih, alokacija i dimenzionisanje registara dodeljeni ovim promenjivama. Pored ovoga potrebno je pravljenje liste svih RT operacija unutar ASMD dijagrama.
 - 4.2. Grupisanje RT operacija u odnosu na njihove odredišne registre. Sve operacije sa istim odredišnim registrima se smeštaju u istu grupu.
 - 4.3. Dodavanje rutirajućih kola (multipleksera) ispred ciljanih registara.
 - 4.4. Formiranje statusnih signala datapath-a, korišćenjem kombinacione logike.
5. **Pisanje HDL modela** – u ovom delu se sastavlja hdl model, uzimajući u obzir sve prethodne korake. Za ovaj korak preporučen je dvoproceni stil

modelovanja u kom jedan proces opisuje kombinacionu logiku, a drugi sekvencijalnu. U koliko je potrebna posebna kontrola za neki blok, on se može odvojiti u poseban proces.

U idealnom svetu, prelazak između stanja i računanje izlaza se izvršavaju trenutno, dok se u hardveru ove reprezentacije implementiraju korišćenjem kombinacionih i sekvencijalnih elemenata koji unosi kašnjenje. Ovo kašnjenje može uzrokovati nepravilan rad uređaja ukoliko se ne ispoštuju vremenska ograničenja. Na primer, funkcionalna jedinica ne može izvoditi operacije, dokle god kontrolni signali ne postave selekzione signale ulaznog multipleksa, dok logika sledećeg stana ne može da se izvrši, dokle god nisu dostupni statusni signali. Kao što se može naslutiti iz prethodnog primera postoje dve povratne petlje zavisnosti između control path-a i data-patha, i od izuzetnog je značaja da odredimo najveću učestanost takta na kojoj naš dizajn radi pouzdano. Maksimalna učestanost takta zavisi od pozicije generisanja kontrolnih i statusnih signala i ne postoji univerzalni način određivanja. Ipak moguće je odrediti granice učestanosti takta korišćenjem statičke vremenske analize u kojoj se proračunava najbolji i najgori slučaj (Slika 9).

U najboljem slučaju kontrolni signali se zahtevaju u kasnijoj fazi obrade, dok se statusni signali generišu na početku datapath-a. Ovakav raspored omogućava da se izvršavanje datapath-a i određivanje narednog stanja controlpath-a može preklapati, čime se perioda takta svodi samo na kašnjenje datapath-a (Slika 9 - levo).

U najgorem slučaju datapath ne može da počne sa izvršavanjem dok ne dobije kontrolne signale, a statusni signali se generišu u kasnim fazama datapatha, tako da logika za izvršavanje narednog stanja mora da sačeka da se propagacija kroz datapath završi (Slika 9 - desno).

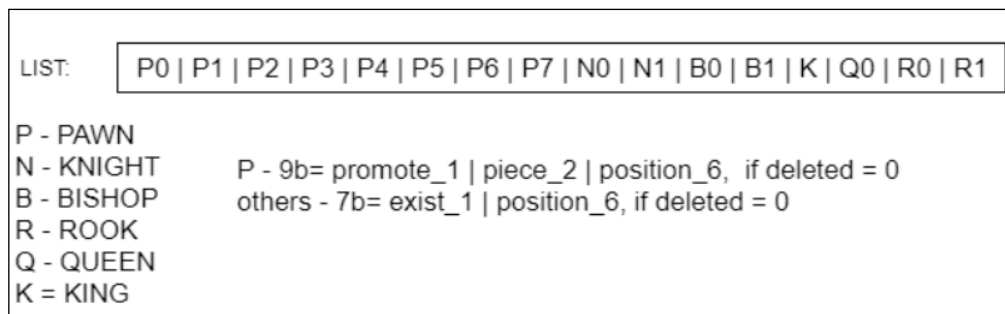


Slika 9 (Statička vremenska analiza za najbolji i najlošiji slučaj)

3.2 Diskusija mogućih implementacija

Kao što je već napomenuto u poglavlju 2, osnovna ideja realizacije eval modula zasniva se na razmotavanju for petlji i sortiranju figura prema ostalim modulima, koji treba da izračunaju njihovu vrednost. U razmatranje su uzete tri situacije.

1. Prva ideja bila je da se, još pre poziva eval funkcije, napravi lista sa pozicijama svih figura na tabli. Ovakva tabela imala bi šesnaest polja za bele i crne figure, kao na Slici 10. Polje za pešake bi imalo devet bitova u kom bi prvi određivao da li je pešak promovisan, druga dva koja je figura promovisanja, a ostali bi određivali poziciju. Za ostale figure bi postojalo samo polje za egzistenciju i polja koja određuju poziciju.



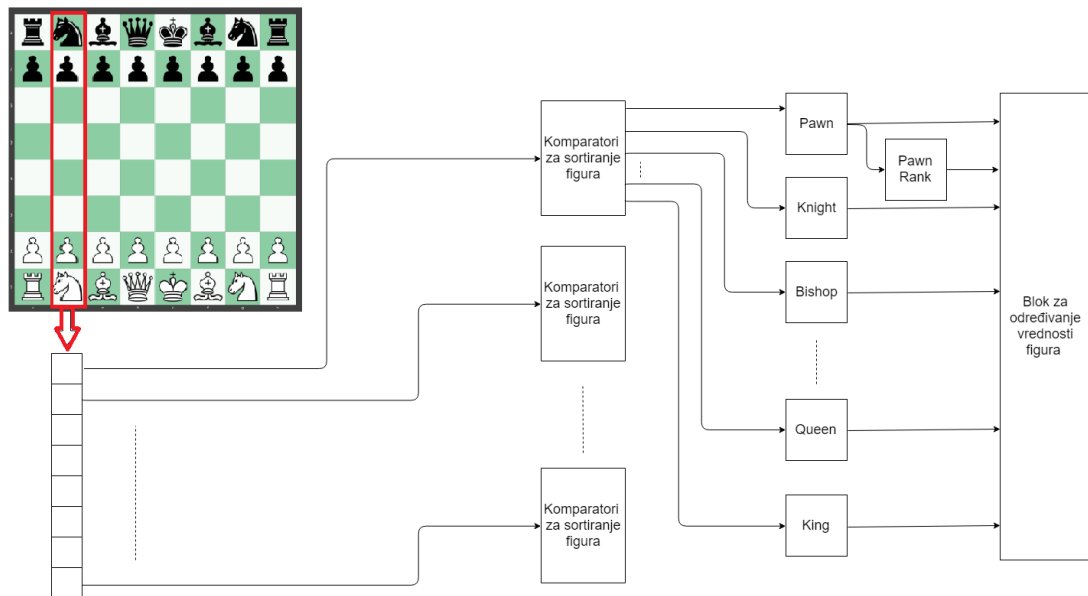
Slika 10 (Format liste koja opisuje poziciju)

Na ovaj način ne bi morali da prolazimo kroz sva polja na tabli, već samo kroz polja liste. Sama lista bi se generisala na početku partije i inicijalizovala na početna polja. U funkciji *makemove* u fajlu *board.cpp* dolazi do odabira novog poteza i na tom mestu bi se lista ažurirala, kao i na mestu gde se vraća potez u funkciji *takeback* u istom fajlu. U koliko bi došlo do promocije na mestu pešaka bi se setovali određeni biti što bi nadalje označavalo figuru koja je promovisana. Na prvi pogled ovakva realizacija deluje privlačno, međutim problem nastaje kod utvrđivanja Rank-a, jer nemožemo biti sigurni da li raspored pešaka u listi odovara koloni na tabli. Na primer, pešak inicijalno postavljen na poziciju B bi mogao da pojede dve figure i završi na poziciji D, dok bi pešak sa pozicije D mogao isto tako da završi na nekom drugom mestu. Iz ovog primera se vidi da ne bi mogli da krenemo sa izračunavanjem ostalih figura dok ne prođemo kroz sve pešake i ne utvrdimo Rank svih kolona.

2. Druga ideja je bila da se doprinos svih polja izračuna u paraleli, međutim kao što je već razmatrano u poglavlju 2, bilo bi potrebno da svako polje bude povezano sa poljima iz njegove i susednih kolona, što bi značajno iskomplikovalo implementaciju i ovaj pristup je odbačen u startu.
3. Treća ideja je bila da se iskoristi činjenica da se izračunavanje Ranka pešaka mora završiti za susedne kolone, pre računanja vrednosti pešaka, kralja i topa u odnosu na njihovo polje, a ne Rank svih polja. Ovo je prirodno vodilo ka ideji da se iskoristi protočna obrada pri odabiru figura u odnosu na kolone table, pri čemu bi izračunavanje Ranka krenulo dva takta pre izračunavanja ostalih figura. U svakom taktu bi se dakle uzimalo po osam polja koji bi se dalje vodili na

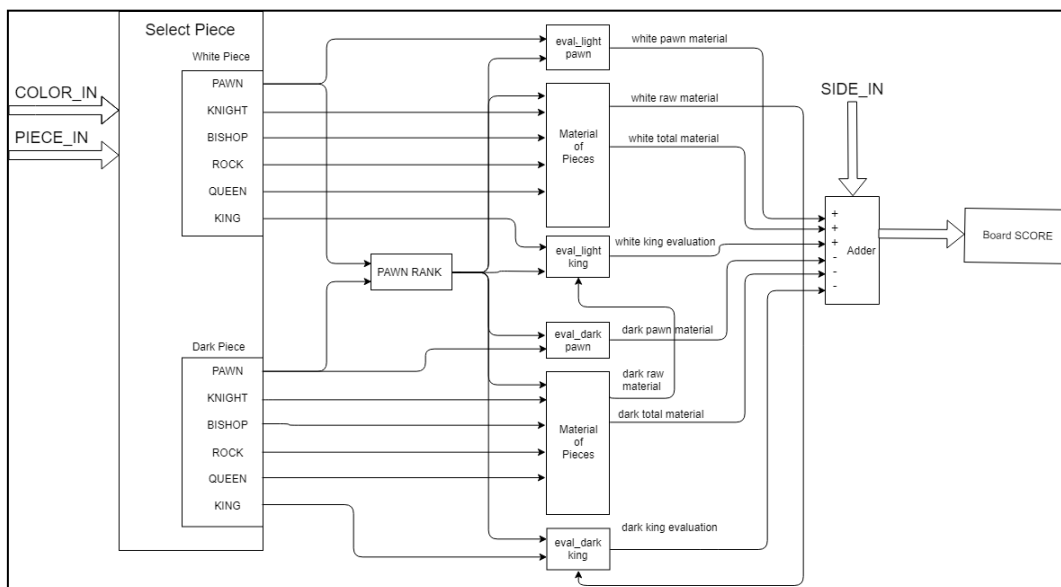
komparatore, svrstavali u fifo bafere i prosleđivali sledećim elementima za obradu (Slika 11). U najgorem slučaju bi za izračunavanje bilo potrebno 2 takta prologa, 8 taktova prolaska kroz tablu i kašnjenje logike za izračunavanje doprinosa. Ovo je naravno gruba procena i u implementaciji je utrošeno više taktova na sinhronizaciju. Ova ideja bila je odabrana je za implementaciju.

U sledećim poglavljima biće opisano projektovanje ovih modula bez koraka za eliminaciju naredbi ponavljanja, zbog toga što je za referentni model uzet SystemC model koji ima implicitno ugrađene naredbe ponavljanja u svoje procese. Takođe važno je istaći da je SystemC model bio polazna tačka za implementaciju hardveskog bloka, pri čemu su se pravili kompromisi u implementaciji, tako da se konačna implementacija i referentni model na nekim mestima razlikuju.



Slika 11 (Ideja rada projektovanog IP bloka)

Kako je ovaj dizajn modularan u nastavku će biti dat opis ključnih modula i njihov način realizacije. Blok šema realizovanog IP modula eval data je na Slici 12.



Slika 12 (Blok dijagram realizovanog IP bloka eval)

3.3 Implementacija bloka Select Piece

Ovaj modul predstavlja ulazni blok u sistem i trebao bi da razvrsta figure sa table po tipu i prosledi ih ostalim modulima. Kao što je već rečeno ovo je urađeno uvođenjem protočne obrade po kolonama, što znači da se po startovanju modula u svakom taktu iščitavaju informacije iz blok-RAM memorije o jednoj koloni sa table. U ovom modulu se takođe izračunava Rank za datu kolonu, korišćenjem prioriternih kodera, koji izračunavaju položaj najviše zaostalog pešaka te kolone, za belog i crnog igrača. Model ovog, kao i ostalih blokova mogu se naći u Dodatku A[1].

3.3.1 Definisaneje interfejsa

Pošto je algoritam pipeline-ovan po kolonama interfejs modula eval sastoji se iz ulaza color_in i piece_in, koji opisuju pozicije figure u određenoj koloni. Na izlazu postoje portovi za svaku vrstu i boju figure, a tu su još i statusni portovi. U nastavku je prikazan ceo interfejs:

- Ulazni interfejs

color_in – tipa STD_LOGIC_VECTOR (0 to 7) – opisuje boju figure na polju u koloni. Ako je ‘0’ radi se o beloj figuri, a ako je ‘1’ radi se o crnoj. Nulta pozicija odgovara osmom redu, dok pozicija 7 označava polje na prvom redu.

piece_in – tipa Array (0 to 7) of STD_LOGIC_VECTOR (2 downto 0) – opisuje tip figure, gde je tip kodovan na sledeći način:

- “0” - Pešak
- “1” – Skakač
- “2” – Lovac

“3” – Top

“4” – Kraljica

“5” – Kralj

“6” – prazno polje

eval_finished_in -tipa STD_LOGIC – označava kada je čitav blok eval u funkciji, odnosno kada je završio izvršavanje. Ovaj signal uveden je da se blok select piece ne bi ponovo aktivirao po nailasku signala za start, ako ceo modul nije završio izvršavanje.

start_in – tipa STD_LOGIC – kontrolni signal koji označava početak rada uređaja

Pored ovih signala tu su i **clk_in** i **reset_in**, koji su prisutni u svim modulima i neće biti posebno izdvajani u interfejsima sledećih modula

- Izlazni interfejs

w_pawn_out – tipa UNSIGNED(6 downto 0) – ovaj signal je izlaz iz fifo bafera belog pešaka. Bit na poziciji 6 označava da li uopšte postoji pešak i ako postoji jednak je ‘0’, dok ostali bitovi označavaju njegovu poziciju na tabli.

w_pawn_last_element_out – tipa STD_LOGIC – ovo je **statusni port** koji označava kada je poslednji element u fifo baferu belog pešaka poslat.

Na isti ovakav način definisani su svi ostali izlazni portovi koji opisuju fifo bafere različitih tipova figura i radi kompaktnosti biće samo izlistani.

w_knight_out, w_knight_last_element_out

w_bishop_out, w_bishop_last_element_out

w_rook_out, w_rook_last_element_out

b_pawn_out, b_pawn_last_element_out

b_knight_out, b_knight_last_element_out

b_bishop_out, b_bishop_last_element_out

b_rook_out, b_rook_last_element_out

Kraljica je figura koja nema svoj bafer, jer pozicija kraljice na tabli nije bitna u kasnijoj obradi, tako da se prosleđuje samo broj kraljica u datom redu preko porta **w_queen_out**, odnosno **b_queen_out** – tipa UNSIGNED(2 downto 0).

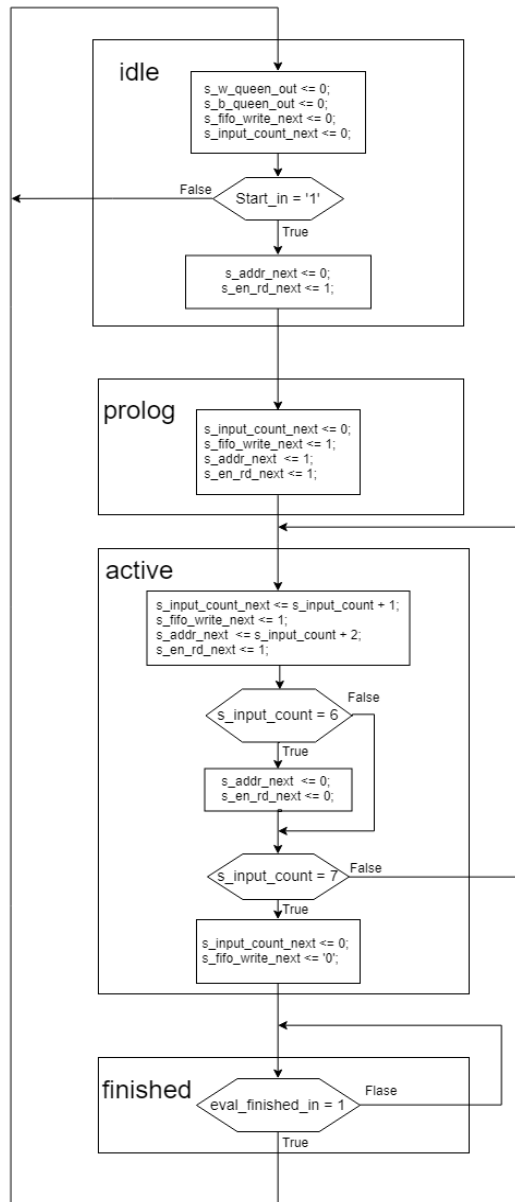
Kralj takođe nema svoj bafer jer znamo da postoji samo jedan kralj, pa se u njegovom slučaju šalje samo njegova pozicija preko porta **w_king_out**, odnosno **b_king_out** – tipa UNSIGNED(6 downto 0). Šesti bit u ovom slučaju govori kada je pozicija kralja validna pri slanju.

w_rank_out i **b_rank_out** – tipa UNSIGNED(2 downto 0) - šalju Rank trenutno obrađivane kolone.

rank_go_out – tipa STD_LOGIC – statusni port koji označava interval u kom se šalje rank preko portova **w_rank_out** i **b_rank_out**.

en_rd_out – tipa STD_LOGIC i **addr_out** – tipa UNSIGNED(2 downto 0) predstavljaju interfejse prema memoriji iz koje se iščitavaju color_in i piece_in. U ovoj implementaciji širina adrese je tri bita jer je za jednu poziciju potrebno proći kroz osam kolona.

3.3.2 Projektovanje *Controlpath* modula



Slika 13 (ASMD dijagram Select Piece modula)

prelazi se u stanje *finished*, u kome modul select piece ostaje do završetka evaluacije pozicije nakon čega se vraća u stanje idle.

U koliko nije drugačije naglašeno signali u ASMD dijagramu primaju sledeće vrednosti:

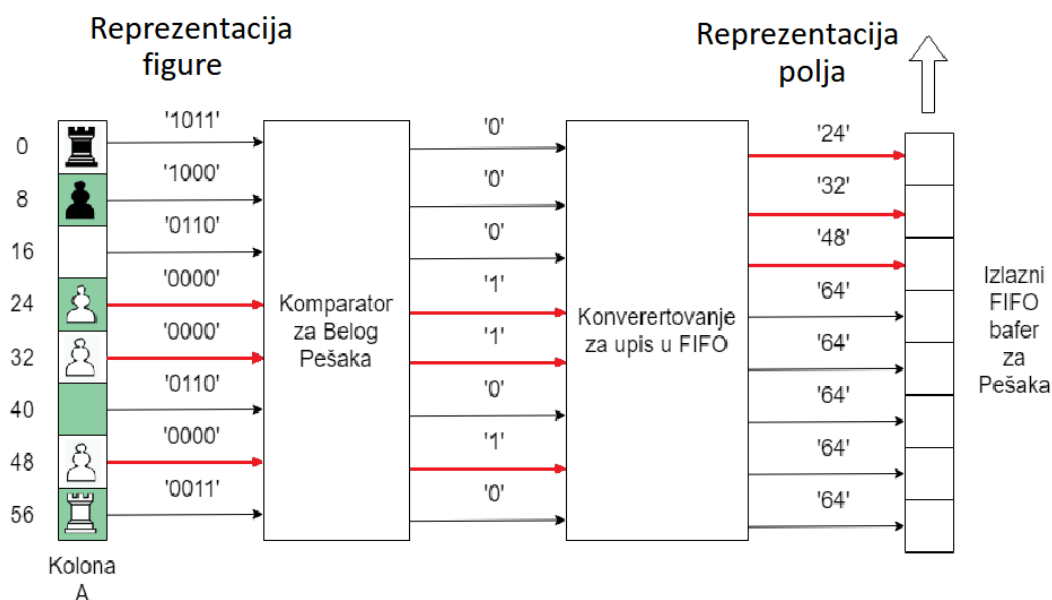
```

s_w_queen_out <= sum(s_w_queen(0:7));
s_b_queen_out <= sum(s_b_queen(0:7));
s_input_count_next <= 0;
s_fifo_write_next <= '0';
s_addr_next <= 0;
s_en_rd_next <= 0;
  
```

Modul select piece predstavlja primer aplikacije sa intenzivnim tokom podataka u kome controlpath nije preterano komplikovan i njegova uloga je u praćenju i sinhronizaciji signala. On se sastoji iz četiri stanja. U stanju *idle* se proverava start signal i ukoliko je on '1', u sledećem taktu započinje čitanje iz memorije i sistem prelazi u stanje *prolog*. U stanju idle se takođe postavlja suma kraljica za belog i crnog igrača na nulu, dok se u ostalim stanjima sumiraju izlazi iz komparatora, koji reprezentuju položaj potencijalnih kraljica u datoj koloni. Ilustracija ovoga data je u datapath delu. U stanju *prolog* se podiže signal s_fifo_write_next i u sledećem taktu započinje upis figura u fifo bafere, kada sistem prelazi u *aktivno* stanje. Tokom upisa u fifo bafere, signal s_input_count ima ulogu u definisanju aktuelne kolone. U aktivnom stanju, kada s_input_count dostigne vrednost '6', prekida se zahtev za čitanjem, zbog toga što adresa sa koje se čita broji do sedam, a startovana je jedan takt ranije. Ovo je bio i razlog za uvođenje stanja *prolog*. Kada s_input_count dostigne '7',

3.3.3 Projektovanje *Datapath* modula

Datapath predstavlja najvažniji deo modula select piece. Zbog kompleksnosti samih komponenti ovaj deo pomalo odstupa od RT metodologije, pri čemu je akcenat bio stavljen na realizaciju i povezivanje komponenti zadatim u SystemC modelu. U datapath delu su realizovani prihvati, struktuiranje podataka i slanje na izlaze. Na ulaznoj strani za svaku figuru postoji po osam komparatora koji ispituju polja jedne kolone, i vraćaju '1' u koliko je figura traženog tipa na zadanom polju (Slika 13). Ovo je realizovano *for generate* naredbom datom u Dodatku B [4]. Potom se u sledećem modulu na osnovu položaja jedinica i trenutne kolone određuju polja na kojima se nalaze figure i vrednosti njihovih polja se upisuju u fifo bafer datog tipa figure. Reprezentacija figure sastoji se iz četiri bita, gde prvi bit određuje boju, a druga tri bita određuju tip.



Slika 14 (Rad Select Piece modula na primeru pešaka)

Blok za konvertovanje podataka i upis u fifo registre realizovani su modulom `fifo_top` koji se sastoji iz modula `reduction_coder` i `fifo_reg`. Uloga `reduction_coder`-a je da ustanovi na kojoj poziciji su jedinice na izlazu iz komparatora određene figure. Rezultat ovog bloka je reprezentovan sa 8 izlaza sa po 4 bita, gde prva tri bita određuju poziciju, a četvrti bit da li figura na toj poziciji postoji. Pored ovoga validne pozicije treba da budu grupisane od nulte pozicije kako bi one kasnije bile prve na izlazu fifo bafera. Na primer, u koliko na ulazu imamo sekvencu sa Slike 14 – “00011010”, na izlazu `reduction_coder`-a treba da bude sledeći rezultat prikazan u heksadecimalnom formatu “34688888”. Pošto svaka kombinacija ulaznih pozicija ima svoju jedinstvenu reprezentaciju na izlazu `reduction_coder`-a, on je realizovan lukap tabelom (*eng. look-up table*). Ova tabela ima 2^8 , odnosno 256 mesta, da bi se pokrile sve kombinacije ulaznih vrednosti, dok je širina svakog mesta $8 \times 4 = 32$ što odgovara reprezentaciji izlaza. Ova tabela data je u fajlu `common.vhd` u Dodatku A.

Izlazi `reduction_coder`-a su povezani na modul `fifo_reg`. Ako je vrednost poslatih pozicija validna, pridružuje im se vrednost aktuelne kolone, i zajedno se upisuju u izlazni fifo. U datom primeru aktuelna kolona je A, kodovana sa nulom tako da se dodavanjem na vrednosti 3, 4 i 6 dobija “011 & 000”, “100 & 000”, “110 & 000”, što je 24, 32 i 48 respektivno. Mesto upisivanja ovih vrednosti u fifo, određeno je unutrašnjom logikom, koja vodi evidenciju o broju validnih polja u baferu i smešta nove članove na prvo slobodno mesto.

Što se tiče Rank-a, on se implementira korišćenjem prioritetnog kodera i izlaza komparatora belih, odnosno crnih pešaka. Prioritetni koder određuje najnižu poziciju (za belog), odnosno najvišu poziciju (za crnog) pešaka u koloni, što u stvari predstavlja rank te kolone. Ovi signali šalju se modulu `pawn_rank`, koji rankove kolona čuva zajedno i prosleđuje modulima `eval_light/dark_pawn`, `material_of_pieces` i `eval_light/dark_king`.

Za kralj-a i kraljicu fifo baferi nisu potrebni. U slučaju kralja se prioritetnim koderom određuje položaj jedinice nakon komparatora, nakon čega se pridružuje aktuelna kolona. U slučaju kraljice se izlazi komparatora sabiraju i ta vrednost se šalje na izlaz.

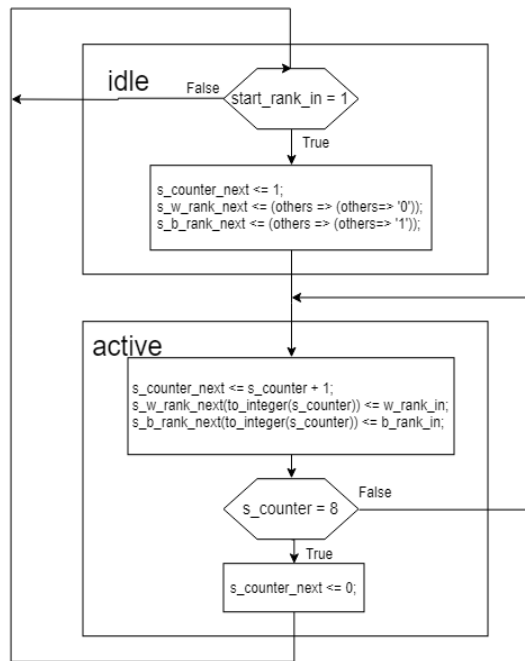
3.4 Implementacija bloka Pawn Rank

Kao što je već rečeno, ovaj blok smešta vrednosti pojedinačnih kolona zajedno u memoriju. U SystemC modelu, ovaj blok je ugrađen u modul `select_piece`, pri čemu se po detektovanju polja sa pešakom svaki put izračunava rank za njegovu kolonu. Za razliku od toga, u hardverskoj implementaciji je ovaj korak razdvojen, kako bi se dobila veća preglednost, pa se u modulu `select_piece` izračunava Rank svake kolone, dok se u modulu `pawn_rank` ove vrednosti samo skladište. Zbog jednostavnosti bloka, u nastavku se daje samo kratak opis interfejsa i ASMD dijagram.

3.4.1 Interfejs

- Ulazni interfejs
w_rank_in i **b_rank_in** – tipa UNSIGNED(2 downto 0) – koji predstavljaju vrednost Rank-a za svaku kolonu.
start_rank_in – tipa STD_LOGIC – kontrolni signal koji govori da je počinje slanje rank-a prve kolone
- Izlazni interfejs
w_rank_out i **b_rank_out** – tipa Array (0 to 9) of UNSIGNED(2 downto 0) – ovi signali omogućavaju pristup rankovima svih kolona u isto vreme. Vrednost na pozicijama nula i devet su konstantne sa vrednostima koje označavaju da u tim kolonama nema pešaka, što se koristi za lakšu implementaciju modula koji koriste rank.

3.4.2 ASMD dijagram



Kao što se može videti sa ASMD dijagrama sistem čeka u stanju **idle** do setovanja signala **start_rank_in**, koji označava da su rankovi kolona počeli da se šalju. Potom se postavljaju inicijalne vrednosti memorije za rank i prelazi se u stanje **active**.

U ovom stanju brojač se uvećava i određuje poziciju na kojoj će biti upisani rankovi kolona. Treba primetiti da on broji od jedan do osam, čime polja u memoriji na mestima nula i devet ostaju na inicijalnim vrednostima. Po završetku brojanja sistem se vraća u početno stanje.

Slika 15 (ASMD dijagram modula **rank_pawn**)

3.5 Implementacija bloka Eval Light/Dark Pawn

Moduli eval light/dark pawn, kao što i ime kaže služi za računanje vrednosti belih i crnih pešaka, pri čemu se uzima u obzir samo polje na kome je pešak i njegov položaj u odnosu na pešake iz njegove i susednih kolona. Vrednosti polja pešaka se na ulaz dovode iz fifo bafera pešaka iz modula **select_piece**, dok se Rank pešaka iščitava iz modula **pawn_rank**. U nastavku će biti dat opis samo modula koji izračunava vrednost belog pešaka, pošto modul koji evaluira crnog pešaka funkcioniše na isti način.

3.5.1 Definisane interfejsa

- Ulazni interfejs
w_pawn_in – tipa UNSIGNED(6 downto 0) – opisuje poziciju pešaka poslatu iz fifo bafera belog pešaka. Bit na poziciji šest opisuje da li se pešak stvarno nalazi na poziciji reprezentovanom preostalim bitovima.
w_last_pawn_in – tipa STD_LOGIC – ovaj signal se setuje kada se pošalje poslednji pešak iz fifo bafera.
w_rank_in i **b_rank_in** – tipa Array (0 to 9) of UNSIGNED(2 downto 0) – kako je već objašnjeno, ovi signali omogućavaju dostupnost ranka svih kolona.

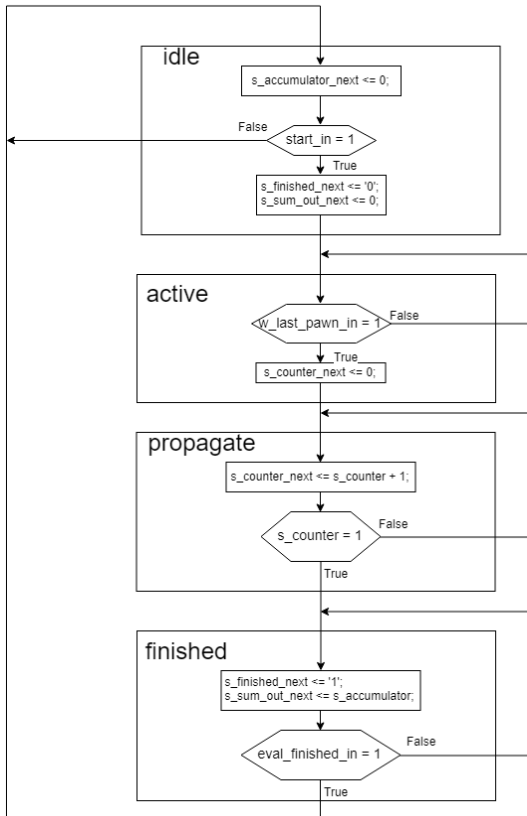
eval_finished_in – tipa STD_LOGIC – ovaj signal označava kraj ukupne evaluacije pozicije.

- Izlazni interfejs

sum_out – tipa SIGNED(11 downto 0) – suma doprinosa svih pešaka

finished_out – tipa STD_LOGIC – označava završetak računanja ovog modula.

3.5.2 Projektovanje *Controlpath* modula



Slika 16 (ASMD dijagram modula **eval_light_pawn**)

finished, u kome se izračunata suma **s_accumulator** dodeljuje signalu **s_sum_out_next**, koje se potom šalje na izlaz zajedno sa signalom **s_finished_next**, koji označava završetak računanja. Sistem ostaje u ovom stanju sve dok se ne dobije signal da je evaluacija pozicije završena.

Ukoliko nije drugačije naglašeno signali u ASMD dijagramu primaju sledeće vrednosti:

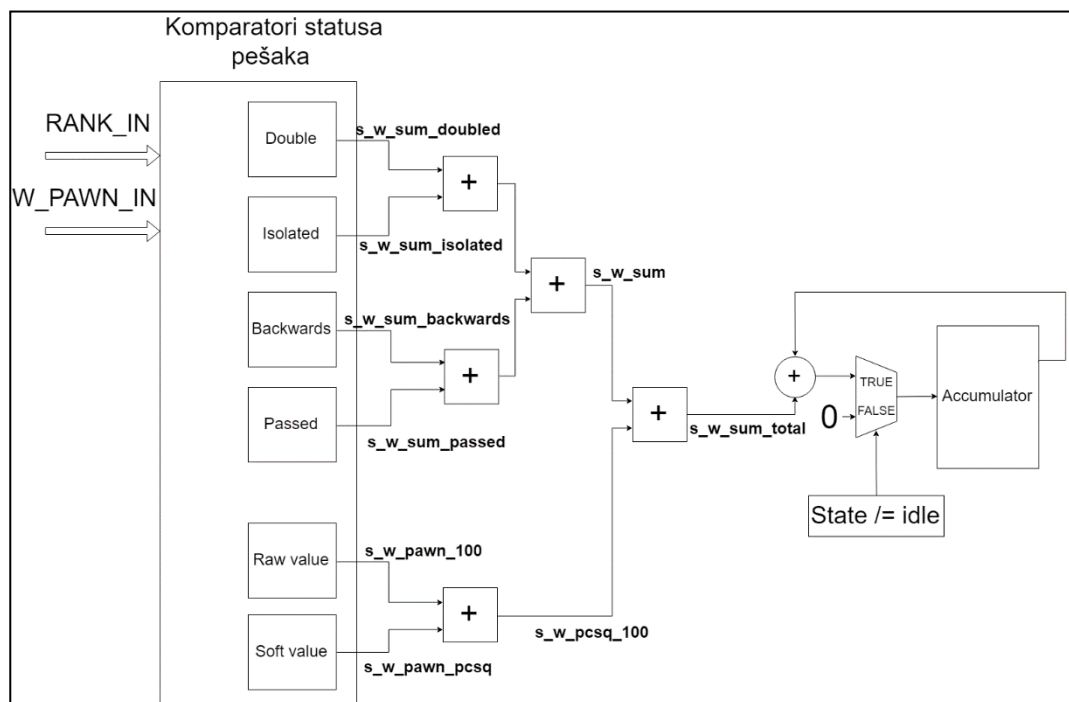
```

s_finished_next <= s_finished;
s_sum_out_next <= s_sum_out;
s_accumulator_next <= s_accumulator +
    signed(s_w_sum_total);
s_counter_next <= s_counter;
  
```

Ovaj modul takođe ima akcenat na datapath-u, pri čemu controlpath služi samo za sinhronizaciju i određivanje kada je konačan rezultat izračunat. U datapath-u ovog modula, primaju se signali poslati sa fifo bafera i vrši se procena i akumuliranje njihovih vrednosti, koji se preko signala **s_w_sum_total**, akumuliraju u registar **s_accumulator**. Po dobijanju signala za start, sistem prelazi u stanje *active* u kom se evaluiraju vrednosti i čeka nailazak signala za poslednjeg pešaka iz fifo bafera **w_last_pawn_in**. Tada FSM prelazi u sledeće stanje *propagate*, koje određuje koliko je još taktova potrebno da se doprinos poslednjeg pešaka uvrsti u ukupnu sumu. Nakon ovoga se prelazi u stanje

3.5.3 Projektovanje *Datapath* modula

Modul počinje da računa doprinos pešaka ukoliko je postavljen najviši bit signala `w_pawn_in` na ulazu na nulu, što označava da pešak postoji na polju opisanom preko ostalih bita signala `w_pawn_in`. Pored ovoga moraju biti ispunjeni uslovi za svaki od komparatora, posebno kako bi oni na svom izlazu dali vrednost bonus poena (Slika 17). Komparatori `Raw_value` i `soft_value` na izlazu uvek daju vrednost kada pešak postoji, i pri tome `Raw_value` je uvek 100, koliko iznosi stalna vrednost pešaka, dok vrednost na izlazu `Soft value` komparatora opisana položajem pešaka na tabli. Dalje se ovi izlazi iz komparatora sabiraju, pri čemu svaki sabirač na slici, predstavlja i registar u isto vreme. Na kraju sabiranja u formi stabla, signal `s_w_sum_total` se sabira sa akumulatorom, čiji izlaz se prosleđuje na izlaz modula, kako je to opisano u *Controlpath*-u.



Slika 17 (Datapath modula `eval_white_pawn`)

Treba još napomenuti da je u Referentnom modelu, signal `s_b_pawn_qcsq` računat korišćenjem matrice *flip*, koja ima ulogu da preslika “horizontalno u ogledalu” vrednosti matrice (npr. prvi red se zamenjuje sa osmim, drugi sa sedmim...). U hardveru ovo je učinjeno invertovanjem bitova kolone crnog pešaka, čime se dobija ušteda u resursima. Ovaj način korišćen je nfa svim mestima gde se koriste **pcsq** - matrice za crnog.

3.6 Implementacija bloka Material of Pieces

Uloga ovog modula je da izračuna doprinos skakača, lovca, topa i kraljice. Modul `material_of_pieces` je na svojim ulazima povezan sa fifo baferima skakača, lovca i topa, modula `select_piece`, kao i signalima koji reprezentuju broj belih i crnih kraljica u koloni. Slično predhodno opisanom modulu, na ulazima se vrši komparacija sa sedmim bitom signala koji reprezentuju figure, a potom se, u koliko

figura postoji, akumulira doprinos figure u sume Raw_material i Soft_material. Akumulator Raw_material prikuplja stalne vrednosti figura, dok se Soft material odnosi na doprinose figura u zavisnosti od pozicije.

3.6.1 Definisane Interfejsa

- Ulazni interfejs

knight_in - tipa UNSIGNED (6 downto 0) – izlaz iz fifo bafera skakača

knight_last_element_in - tipa STD_LOGIC – označava poslednji element u fifo baferu skakača

bishop_in - tipa UNSIGNED (6 downto 0) – izlaz iz fifo bafera lovca

bishop_last_element_in - tipa STD_LOGIC – označava poslednji element u fifo baferu lovca

rock_in - tipa UNSIGNED (6 downto 0) – izlaz iz fifo bafera topa

rock_last_element_in - tipa STD_LOGIC – označava poslednji element u fifo baferu topa

queen_in - tipa UNSIGNED (2 downto 0) – određuje broj kraljica na datoj koloni

w_rank_in - tipa Array (0 to 9) of UNSIGNED(2 downto 0) – rank belih pešaka

b_rank_in - tipa Array (0 to 9) of UNSIGNED(2 downto 0) – rank crnih pešaka

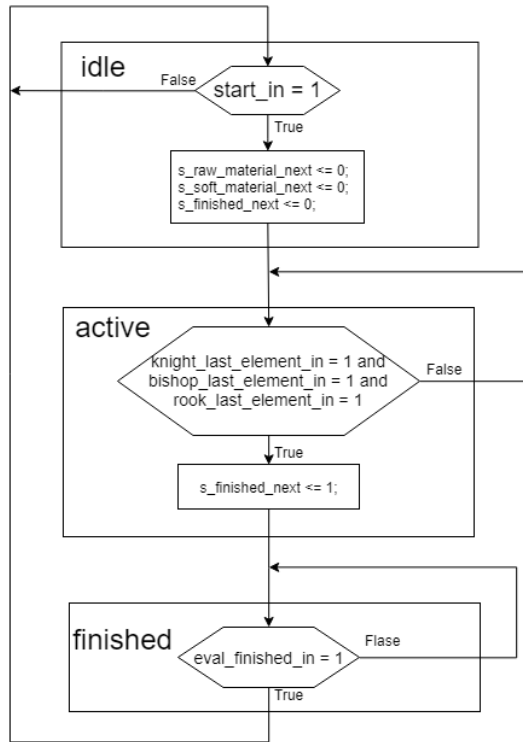
- Izlazni interfejs

raw_material_out - tipa UNSIGNED (13 downto 0) - suma stalnih vrednosti svih figura

soft_material_out - tipa SIGNED (8 downto 0) – suma vrednosti figura zavisnih od položaja na tabli

finished_out - tipa STD_LOGIC – označava kada je modul material_of_pieces završio sa radom

3.6.2 Projektovanje *Controlpath* modula



Slika 18 (ASMD dijagram modula *material_of_pieces*)

Ukoliko nije drugačije naglašeno signali u ASMD dijagramu primaju sledeće vrednosti:

```

s_raw_material_next <= s_raw_material +
    s_raw_material_temp;
s_soft_material_next <= s_soft_material +
    s_soft_material_temp;
s_finished_next <= s_finished;

```

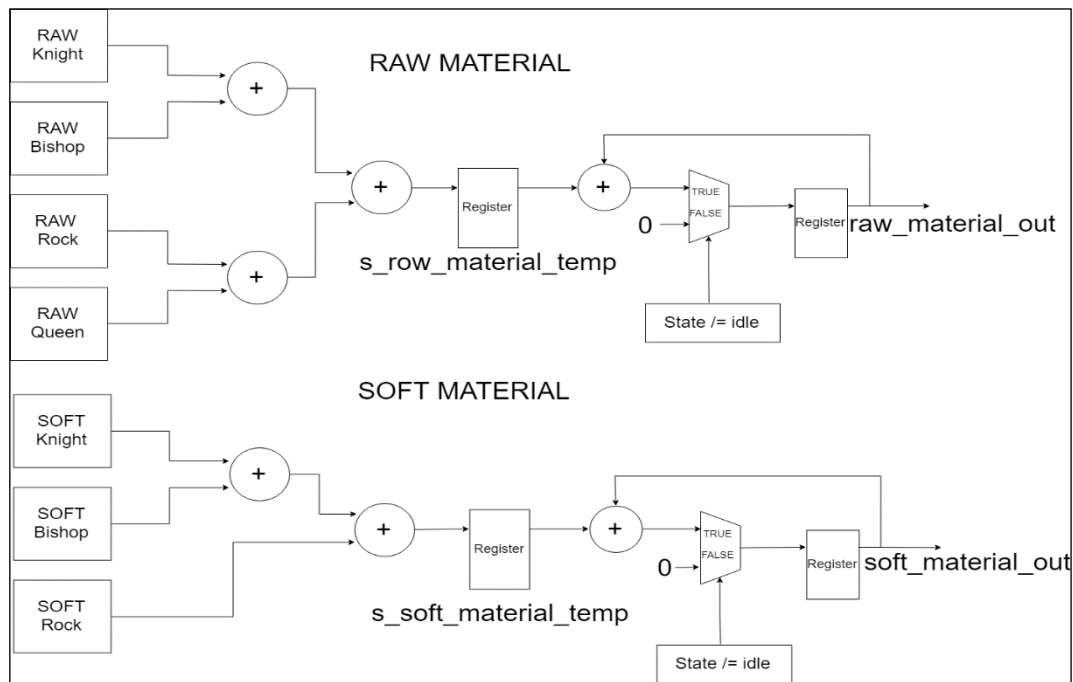
U stanju **idle**, kada *start_in* postane jedinica, sistem prelazi u stanje **active**, pri čemu se signali *s_raw_material_next* i *s_soft_material_next* postavljaju na nulu. Ovi signali su povezani na izlaz i čine akumulatore, u koje se upisuju vrednosti figura u datapath-u. U stanju **active** pristižu signali iz fifo bafera i kada se pošalju poslednji elementi iz bafera prelazi se u sledeće stanje i setuje se signal koji označava završetak. U stanju **finished** ostaje se dok se ne završi ukupna evaluacija pozicije.

3.6.3 Projektovanje *Datapath* modula

Prikaz datapath-a može se videti na Slici 19. Blokovi na ulazu proveravaju da li figure reprezentovani signalima na ulazu postoje. U slučaju da postoje njihovi izlazi dobijaju sledeće vrednosti:

- RAW Knight – 300
- RAW Bishop – 300
- RAW Rock – 500
- RAW Queen – 900

Izlazi iz SOFT Knight i SOFT Bishop modula zavise od položaja figure na tabli i dobijaju vrednosti date u *pcsq* matricama. Blok SOFT Rock dobija vrednosti u odnosu na prisustvo pešaka u njegovoj koloni i na to da li je na sedmom (za belog) ili prvom (za crnog) redu. Potom se izlazi ovih blokova sabiraju i smeštaju u registre *s_raw_material_temp* i *s_soft_material_temp*, koji se dalje akumuliraju u koliko stanje nije *idle* i rezultat se šalje na izlaz.



Slika 19 (Datapath modula `material_of_pieces`)

3.7 Implementacija bloka Eval Light/Dark King

Ovaj blok služi za računanje doprinosa kralja. Ovaj doprinos određen je položajem kralja na tabli, položajem njegovih i protivničkih pešaka ispred njega i sume stalnih vrednosti figura (*eng. raw material*) protivnika. Ovo se odnosi na slučaj kada je vrednost signala `raw material` veća od 1200, međutim kada je ova vrednost manja, smatra se da nije više bitno koliko je kralj zaštićen, već je bitno gde se nalazi na tabli, zbog čega se uzima vrednost iz matrice **king_endgame_pcsq**. U ovom delu biće prikazana realizacija za belog igrača - modul `w_eval_king`, dok je realizacija za crnog igrača urađena na sličan način.

3.7.1 Definisane interfejsa

- Ulazni Interfejs**
 - w_rank_in** i **b_rank_in** – tipa `Array(0 to 9) of UNSIGNED(2 downto 0)` – označavaju rank
 - king_in** - tipa `UNSIGNED (6 downto 0)` – ovaj signal predstavlja poziciju kralja, kao što je opisano ranije
 - raw_material_in** - tipa `UNSIGNED (13 downto 0)` – ovaj signal označava sumu stalnih figura protivničkog igrača
 - raw_material_ready_in** - tipa `STD_LOGIC` – označava kada je signal `raw_material_in` validan
 - eval_finished_in** - tipa `STD_LOGIC` – koji označava kraj ukupne evaluacije pozicije

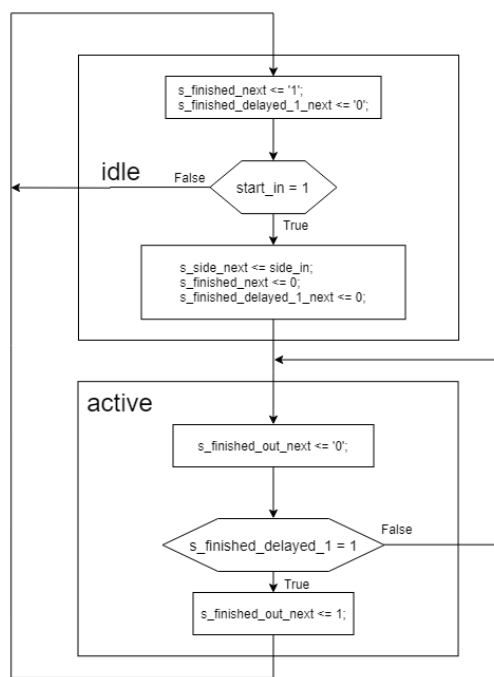
vrednosti ranka kao što je opisano na početku i prelazi se u neka od tri slučaja – **active1** , **active6** i **active_center**. Stanja koja slede nakon **active1** i **active6** su identična sa razlikom u setovanju koloni rank-a. Ove kolone postavljaju se na datapath koji putem protočne obrade ispituje položaje najzaostalijsih kraljevih pešaka u odnosu na položaje najzaostalijsih protivničkih pešaka. Potom se ulazi u stanja sa dodatkom **_1** i **_2**, u kojima datapath prosleđuje izračunate vrednosti. Treba obratiti pažnju da su ove vrednosti mogu biti negativne, što pokazuje u stvari koliko je kralj otkriven. Što se tiče stanja **active_center**, ne ispituje se uticaj ranka, već samo da li je kolona bez pešaka. Nakon ovoga se sistem prevodi u stanje **finished_1**, u kom se čeka da ulazna vrednost **raw_material_in**, protivničkog igrača postane validna, nakon čega se dobijena suma do tad množi sa dobijenim signalom **raw_material** i deli sa brojem 3100 u datapathu. Logika iza ovoga je da kralj može biti u opasnosti samo ako protivnik ima dovoljno figura da ga napadne. Ako se ispostavi da je vrednost signala **raw_material_in** manja od 1200, umesto svog računanja se uzima vrednost iz matrice **king_endgame_pcsq**, koja bolje opisuje ulogu kralja. Ovo je uvedeno zbog toga što u završnicama partije, kada nema mnogo figura, povoljnije da kralj bude aktivan i zauzima centar table. Kada sistem pređe u stanje **finished_2** setuje se signal **s_finished_next** koji označava da je procena kralja izvršena. Da bi se vratili u početno stanje čitava evaluacija table mora da se završi i da stigne novi signal za start.

3.7.3 Projektovanje *Datapath* modula

U prvom stanju kada naiđe visoka vrednost signala **king_in(6)**, na akumulator **s_sum** se upisuje doprinos kralja u zavisnosti od položaja (Slika 21). U isto vreme se na ulaze datapath-a u registre **s_w_rank** i **s_b_rank** upisuju vrednosti u odnosu na mesto gde se kralj nalazi. Od vrednosti izlaza ovih registara propuštaju se konstante, koje određuju kaznene poene, u registre **s_w_sum** i **s_b_sum**. Ovi registri se sabiraju u **s_temp** , koji se dalje oduzima od akumulatora u stanjima **active_3**, **active_3_1**, **active_3_2**, **active_8**, **active_8_1**, **active_8_2**, dok se u stanjima **active_center** u koliko ima pešaka u koloni od sume oduzima konstanta '10'. Kada se izračuna vrednost protivničkih figura **raw_materials** suma, koja je do tad dobijena se množi sa ovom vrednošću, dok se u isto vreme ispituje da li je vrednost **raw_material** manja od 1200. Ukoliko jeste na izlaz se prosleđuje vrednost iz matrice **king_endgame_pcsq** na poziciji kralja, dok se u suprotnom izlaz iz **s_sum** registra deli sa 3100 i to se šalje na izlaz.

result_out - tipa SIGNED (14 downto 0) – rezultat evaluacije pozicije
finished_out - tipa STD_LOGIC – označava završetak računanja modula adder

3.8.2 Projektovanje *Controlpath* modula



Ukoliko nije drugačije naglašeno signali u ASMD dijagramu primaju sledeće vrednosti:

```

s_finished_next <=
(w_pawn_finished_in and w_material_finished_in) and
(w_king_finished_in and b_pawn_finished_in) and
(b_material_finished_in and b_king_finished_in) ;

s_finished_delayed_1_next <= s_finished;
s_finished_out_next <= s_finished_out;
s_side_next <= s_side;

```

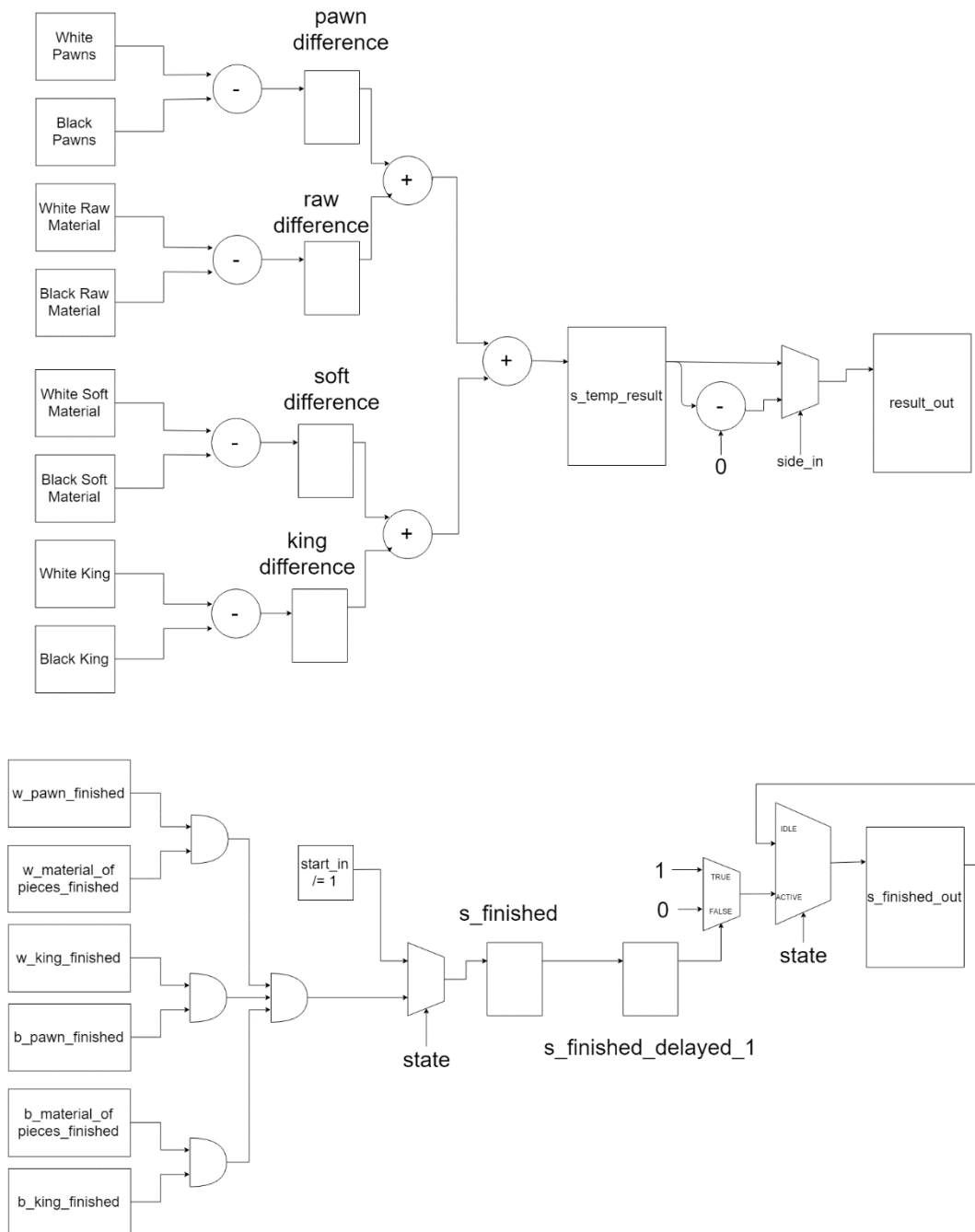
Controlpath je vrlo jednostavan i sastoji se samo iz dva stanja. U prvom stanju se čeka da se sistem pokrene signalom **start_in**. Nakon toga se prelazi u stanje u kom čeka zakašnjeni signal **s_finished**, koji je visok ako su svi moduli za evaluaciju završili sa radom. Ovo kašnjenje je ubačeno da suma dobijenih vrednosti na ulazu stigne da se sumira i prosledi na izlaz.

Slika 22 (ASMD dijagram modula adder)

3.8.3 Projektovanje *Datapath* modula

Izgled datapath-a dat je na Slici 23. Kao što je već rečeno u controlpath delu, na ulaze gornjeg dijagrama vrednosti se upisuju samo ako su adekvatni **finished** signali visoki. Potom se oduzimaju vrednosti dobijene za belog i crnog igrača i smeštaju u registre. Ovi registri se potom sabiraju u registar **s_temp_result**, čiji se rezultat prosleđuje u izlazni registar **result_out**, pri čemu **side_in** određuje znak rezultata.

Na donjem kolu, se propagiraju vrednosti signala koji označavaju završetak računanja predhodnih modula. Nakon prolaska kroz AND kapije, u zavisnosti od stanja, se rezultat upisuje u registar **s_finished**, koji se dodatno zakašnjava, kako bi se sinhronizovao sa tokom podataka i šalje na izlaz, kako je definisano u ASMD dijagramu.

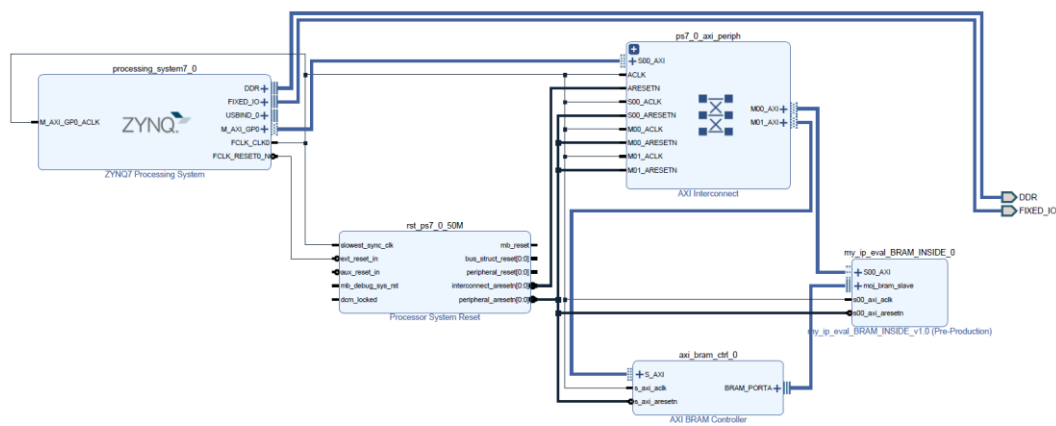


Slika 23 (Datapath modula adder)

3.9 Integrisanje u sistem i merenje performansi

Posao digitalnog dizajnera nije gotov kada završi projektovanje IP bloka. Da bi ostali korisnici mogli da koriste IP blok, potrebno je omogućiti laku komunikaciju sa ostatkom sistema. Drugim rečima ovo znači da moramo napraviti standardne interfejsa za naš modul. Postoje razni interfejsi koji se koriste za povezivanje IP bloka sa procesorom, memorijom i drugim komponentama koji omogućavaju široku lepezu mogućnosti u pogledu performansi i količine podataka. Za ovaj projekat procesor i IP blok komuniciraju preko blok-RAM (*eng. block RAM - BRAM*) memorije i AXI-Lite interfejsa. Pozicija se upisuje u memoriju na 8 memorijskih lokacija po 32 bita, pri čemu svaka četiri bita reprezentuju jedno polje. Bitovi su orijentisani u smeru od prve (najviši bitovi) do osme (najniži bitovi) kolone na tabli. Procesor preko AXI-Lite interfejsa šalje informacije o startu i strani za koju se pozicija evaluira, dok sa druge strane prima rezultat i znak da je rezultat validan. Da bi napravili AXI interfejsa na svom modulu korišćen je Vivado okruženje i alat za stvaranje i pakovanje IP jezgra. Ovim smo dobili u svom projektu jedan top-modul u kome su spojeni AXI kontroler i naš IP. Što se tiče blok Ram interfejsa oni su dobijeni mapiranjem portova BRAM interfejsa na portove IP bloka preko kojih se brši upis u memoriju.

Integriranje u sistem se vrši stvaranjem blok dizajna u Vivadu, kao što je prikazano na Slici . U ovom radu korišćen je procesor Zynk7, dok pored njega potrebno je instancirati Interconnect, koji je zadužen za povezivanje perifernih blokova sa procesorom. Da bi procesor mogao da upisuje u memoriju dizajniranog IP bloka, potrebno je instancirati i blok RAM kontroler, koji rukovodi slanjem transakcija prema IP bloku. I na kraju potrebno je iz Kataloga IP blokova izabrati projektovani IP i povezati sistem korišćenjem naredbe RUN Automation.



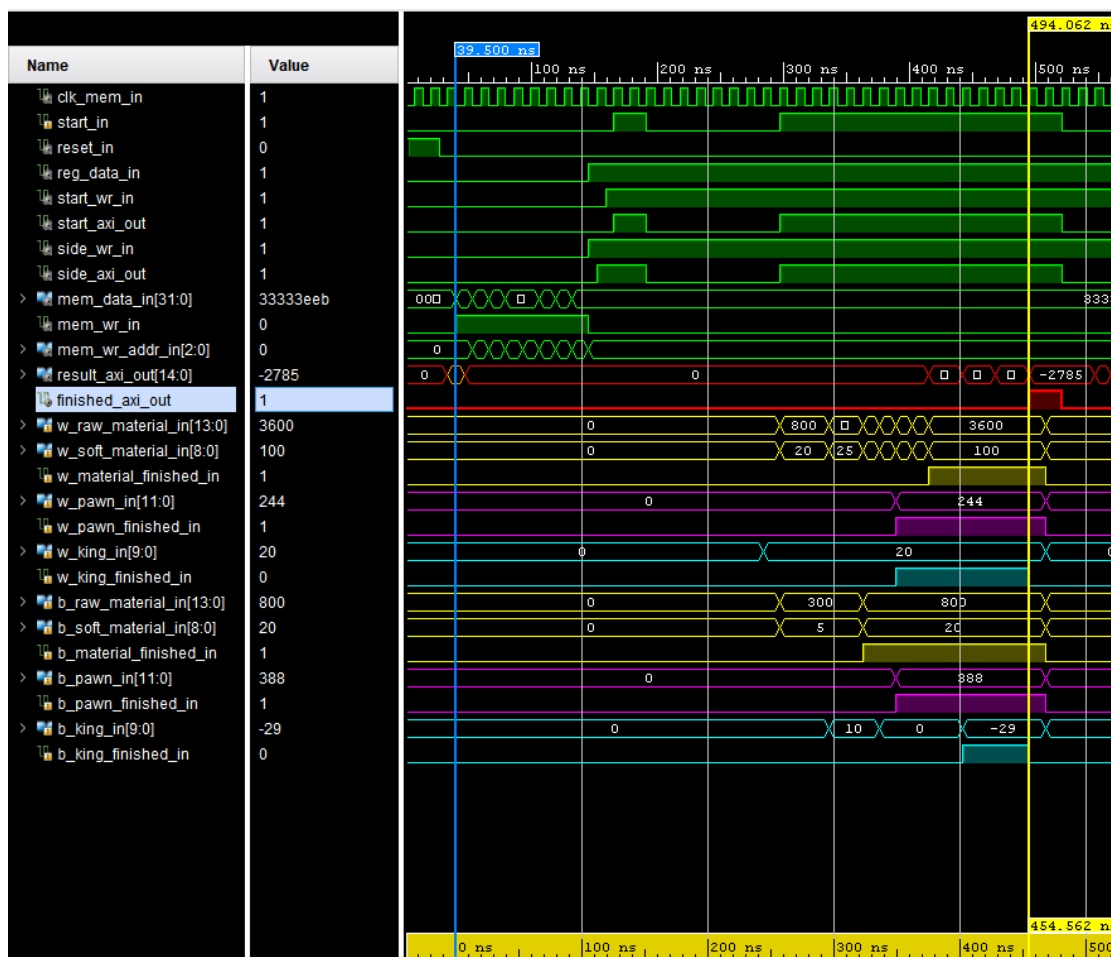
Slika 24 (Blok dijagram sistema u Vivadu)

Što se tiče procene performansi, posle pokretanja tajming analize u Vivadu dobijamo sledeće rezultate (Slika 25). Kada se sumiraju vremena **setup**, **hold** i širina periode takta dobija se da period takta mora biti najmanje 13,175 ns odnosno da je najviša moguća frekvencija 75,9MHz.

Intra-Clock Paths - s00_axi_aclk				
Clock: s00_axi_aclk				
Statistics				
Type	Worst Slack	Total Violation	Failing Endpoints	Total Endpoints
Setup	8.429 ns	0.000 ns	0	5
Hold	0.246 ns	0.000 ns	0	5
Pulse Width	4.500 ns	0.000 ns	0	6

Slika 25 (Statička vremenska analiza urađena u Vivadu)

Da bi izračunali tačno vreme izračunavanja za neku poziciju napravljena je testbenč simulacija prikazana na Slici 26.



Slika 26 (Simulacija projektovanog IP bloka)

Dužina periode takta je podešena na 13,175ns i sa slike se može videti da vreme potrebno za slanje pozicije i izračunavanje iznosi 454,562ns. Da bi se procenilo

```
int i, start, end;

start = get_ms();

for(i=0; i<1000000; i++)
    eval();

end = get_ms();

printf("Vreme = %d \n ", end - start);
```

Listing Koda 2

vreme u softveru napisan je sledeći deo koda:

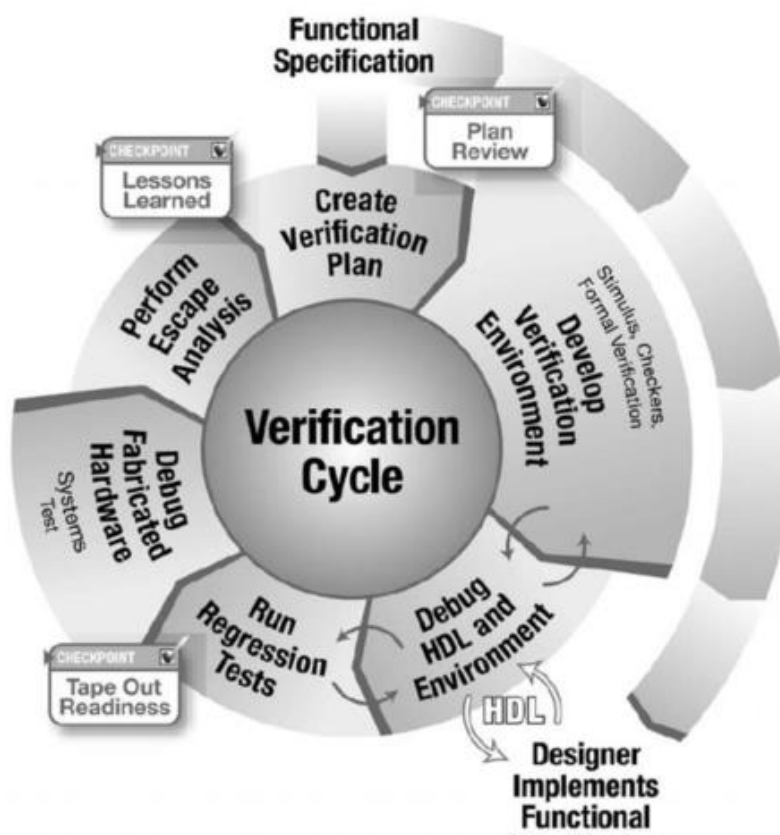
Na ovaj način se dobija prosečan rezultat od 750 ns. Odavde se može zaključiti da je ubrzanje za jednu evaluaciju u hardveru oko 1,65 puta. Ovaj broj ne deluje impresivno, međutim kada se ima u vidu koliko se često funkcija eval poziva po potezu, utisak se menja. Na osnovu jednostavne analize, koja proverava broj pozivanja funkcije eval po potezu, dobijen je prosečan rezultat na nivou partije da se funkcija eval poziva 23000 puta u svakom potezu. Ovo dalje znači da se ubrzanje u hardveru množi sa ovim brojem i da je ukupno ubrzanje na nivou poteza jednako 37950 puta. U realnom hardveru je moguće da ubrzanje usled kašnjenja kola neće biti koliko je predvićeno, ali moguće je dati granice procene. Ukoliko bi uzeli da je naša procena za 10% optimističnija, ubrzanje na nivou poteza bi bilo 34155 puta. Čitalac bi se sada mogao zapitati šta će nekome šahovski program koji radi mnogo brže. Odgovor na ovo pitanje leži u mogućnosti algoritma da iskoristi uštedeno vreme. U slučaju šaha, kvalitet poteza zavisi od dubine pretraživanja stabla pozicija, i što se dublje ide nalazi se bolji potez. Primera radi, u koliko šahovski program pretražuje moguće poteze na dubini 5, obično dete može da ga pobedi, međutim ako se pretraživanje vrši na nivou 15, šahovski program može biti u stanju da pobedi i svetskog šampiona. Isti ovaj princip važi i za druge domene programiranja i mogućnost da budete brži, daje vam sposobnost da uradite više. Upravo ovo daje motivaciju da za akceleraciju u hardveru.

Glava 4

Funkcionalna Verifikacija projektovanog IP bloka

Cena grešaka u hardveru je enormno velika. Za razliku od softvera nije moguće ažurirati aplikaciju preko interneta, već se mora početi proces proizvodnje čipova ispočetka, dok hardver koji već ima greške nije jednostavno popraviti. Uz greške samog hardvera i nezadovoljstva kupaca, tu su i tužbe koje mogu koštati kompaniju za projektovanje hardvera izuzetno mnogo. Zbog ovoga se obično koristi nekoliko strategija. Prvo i osnovno, potrebno je u samom hardveru smanjiti greške. Pored ovoga moguće je korišćenje softvera koji kontroliše hardver i njegovim prilagođavanjem mogu se ispraviti sitniji propusti načinjeni u hardveru.

Da bi ustanovili da projektovani IP dobro radi, nije dovoljno napraviti testbenč i simulirati konačan broj slučajeva. Na ovaj način nismo dovoljno sigurni da li smo pokrili sve bitne slučajeve i često se dešava da se mogu dogoditi slučajevi koje ni ne razmatramo. Da bi se to izbeglo potrebno je da se utvrdi kriterijum po kome se vrši provera, tako da se pokriju ključne funkcionalnosti, i da se obezbedi kriterijum završetka testiranja. Verifikacioni tok [8] ide paralelno sa razvojem hardvera (Slika 27). U prvom koraku opisuje se željeni proizvod koji se sastoji od specificiranja

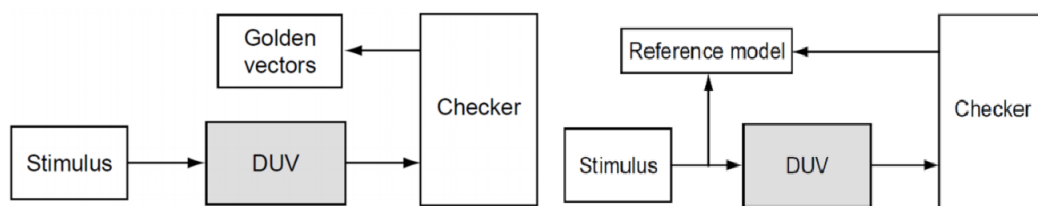


Slika 27 (Tok verifikacije)

interfejsa, funkcije koja mora da se izvršava i ograničenja koja utiču na dizajn. Potom se prelazi na kreiranje verifikacionog plana koji treba da odgovori na pitanja “Šta se verifikuje?” i “Kako se verifikuje?”. Verifikacioni plan uključuje specifične metode i testove, završni kriterijum testiranja, potrebne resurse u vidu raspoloživog hardvera, softvera i broja inženjera, funkcije koje treba da se verifikuju i funkcije koje se ne verifikuju. Razvoj okruženja podrazumeva razvoj verifikacionih komponenti koje najčešće generisanje stimulusa i proveru dobijenih rezultata. U odnosu na način testiranja postoje nekoliko različitih tipova verifikacionih okruženja i to su:

- Deterministički – testovi čiji su ulazi i izlazi poznati unapred,
- Pseudo-slučajni (*eng. Random based*) – testovi čiji se ulazi generišu na pseudo-slučajan način uz korišćenje ograničenja koja se postave,
- Zasnovani na formalnoj verifikaciji – testovi koji se zasnju na dokazivanje tvrđenja,
- Generatori scenarija – testovi koji generišu specifične scenarije.

Razvoj verifikacionog okruženja takođe podrazumeva odabir tipa testbenča i on može da se zasniva na principu zlatnih vektora i na konceptu referentnog modela (Slika 28).



Slika 28 (Princip verifikacije zasnovan na zlatnim vektorima i referentnom modelu)

Princip zlatnih vektora uključuje izračunavanje očekivanih rezultata koje dizajn koji se testira (*eng Design Under Verification - DUV*) treba ima pre puštanja simulacije. Tokom simulacije se ove vrednosti upoređuju sa rezultatima koje se dobiju na izlazu dizajna. Sa druge strane verifikaciona okruženja zasnovana na referentnom modelu u toku same simulacije izračunavaju tražene vrednosti i upoređuju ih sa dobijenim vrednostima dizajna. Za ovaj način potrebno je napisati referentni model, koji ispunjava istu funkcionalnost kao hardver, samo što je napisan na višem nivou apstrakcije, čime se lakše nalaze i otklanjaju greške. Ovakav pristup pruža nam mogućnost da u toku same simulacije zadajemo vrednosti ulaza i ispitamo sve interesantne slučajeve. Često je potrebno randomizovati ulaze kako bi pokrili slučajeve o kojima nismo razmišljali. Ovaj princip je korišćen za proveru projektovanog IP bloka.

Korak za debugovanje HDL modela i verifikacionog okruženja je po svojoj prirodi vrlo dinamičan, jer se pronalaze prve greške u dizajnu pri čemu se on menja, pa je u nekim slučajevima potrebno i napraviti promenu verifikacionog okruženja. Kada se pronade greška, nije uvek sigurno da se ona nalazi u samom dizajnu. Ponekad je moguće da se greška nađe i u komponentama verifikacionog okruženja,

referentnom modelu, pa čak i u samoj specifikaciji. Zbog ovoga je potrebno imati u vidu sve komponente koje učestvuju u testovima.

Kada broj pronađenih grešaka opadne, prelazi se na korak regresionih testova. Cilj ovog koraka je da se utvrdi da su funkcionalnosti zadate u verifikacionom planu ispoštovane. Ovo se dobija pokretanjem velikog broja testova koji ispituju te funkcionalnosti. Pri svakoj novoj promeni u dizajnu potrebno je ponovo pokrenuti zadate testove, jer se dešava da jedna promena u dizajnu uzrokuje greške na drugom mestu.

Kada prođe dovoljan broj testova i greške budu otklonjene, dizajn i verifikacioni tim se sastaju i razmatraju da li su svi kriterijumi zadati specifikacijom ispunjeni. U koliko je sve spremno, dizajn može da krene u proces fabrikacije. Ovaj trenutak naziva se i *tejpaut* (eng. *Tape-Out*), jer su u prošlosti inženjeri čuvali podatke u dizajnu na magnetnim trakama i nosili na fabrikaciju.

Ovim verifikacioni ciklus nije gotov, jer se mora utvrditi ispravnost fabrikovanog dizajna. Greške u procesu fabrikacije mogu da nastanu zbog nesavršenosti procesa, a pored ovih tu su i eventualne greške samog dizajna koje su prošle neopaženo. Da bi se sistemi lakše testirali, u njih su ubačena verifikaciona kola koja pružaju dostupnost unutrašnjim komponentama.

I pored svih testova, moguće je da se greške provuku i završe u uređajima koji se prodaju na tržištu. Nihova cena po kompaniju je ogromna, kao što je već više puta naglašeno, i da se greške ne bi ponavljale na budućim projektima, poslednji korak u verifikacionom ciklusu predstavlja *eskejp* (eng. *Escape*) analiza. Cilj ove analize je pored razmatranja propusta, da se unaprede verifikaciono okruženje i da se greške više ne ponavljaju. Ovaj korak je od velike važnosti, ali se i pored toga često zaobilazi pod pritiskom da se što pre krene u novi projekat.

4.1 UVM metodologija i SystemVerilog

UVM (engl. *Universal Verification Methodology*) je standardizovana metodologija za funkcionalnu verifikaciju sa pomoćnom bibliotekom u SystemVerilog jeziku [8]. Kreirana je sa željom da se postigne lakša ponovna upotreba testbenčeva i jednostavno kreiranje univerzalnih, visoko-kvalitetnih VIP-a (engl. *Verification Intellectual Property*). UVM je baziran na OVM-u (engl. *Open Verification Methodology*)[9] i eRM-u (engl. *e Resuse Methodology*)[10].

Jedna od glavnih karakteristika ove metodologije je UVC (engl. *Universal Verification Component*)[11], odnosno univerzalne verifikacione komponente koje imaju istu strukturu (sadrže monitore, drajvere, sekvencere) što omogućava lako korišćenje bilo kako nezavisne komponente ili kao deo većeg sistema.

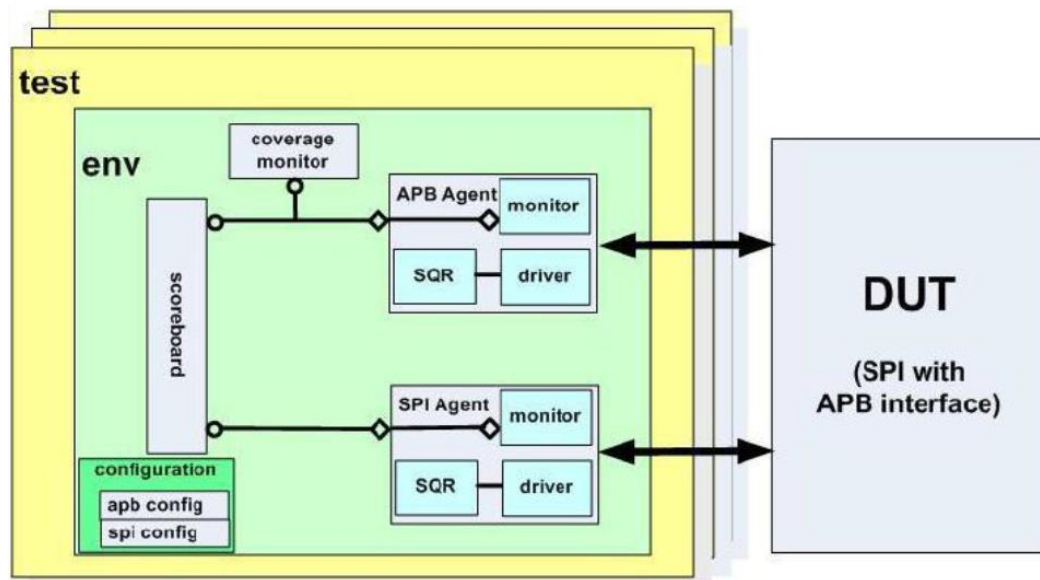
Objektno-orijentisani dizajn, kao glavna karakteristika SystemVerilog jezika, omogućava lako kreiranje verifikacionih komponenti, dok veliki broj predefinisanih funkcija i taskova znatno ubrzava proces kreiranja testova.

UVM obezbeđuje osnovu za verifikaciju zasnovanu na pokrivenosti (engl. *Coverage Driven Verification*, skraćeno CDV) koja ne zahteva kreiranje velikog broja testova, osigurava temeljnu verifikaciju na osnovu zadatih parametara i

olakšava proces pronalaženja problema. Ovo se postiže korišćenjem samoproveravajućih (*eng. self-checking*) testbenčeva, automatskog generisanja testova i korišćenjem podataka o pokrivenosti.

Struktura testbenča

Svaki UVM testbenč sadrži komponente sa jasno definisanom ulogom, koje prate metodologiju i omogućavaju laku ponovnu upotrebu. Na Slici 29 može se videti struktura tipičnog UVM testbenča.



Slika 29 (Tipična struktura UVM testbenča)

Svaki UVM test sastoji se iz hijerarhijske strukture komponenti, koje su predstavljene klasama u SystemVerilogu. Bazične komponente u jednom verifikacionom okruženju čine **sekvencer**, **drajver** i **monitor**. Sekvencer ima ulogu u generisanju transakcija stimulusa na visokom nivou apstrakcije. Drajver sa druge strane te transakcije prevodi na nivo razumljiv hardveru, dok monitor izpituje ispravnost prosleđenih transakcija, a može i skupljati pokrivenost. Ove komponente zadužene su za kontrolisanje neke funkcionalnosti i zbog lakše ponovne upotrebe grupišu se zajedno u klasu **agent**. Klasa **env** predstavlja sledeći viši nivo hijerarhije. Ova klasa može sadržati više agenata, skorbord(*eng. scoreboard*), komponentu pokrivenosti(*eng. coverage monitor*) i konfiguracioni objekat. Skorbord na osnovu podataka dobijenih iz monitora nekog agenta vrši proveru ispravnosti. U njemu su često ugrađeni referentni modeli sa čijim rezultatima se porede signali iz monitora. Komponenta pokrivenosti beleži koji su testovi urađeni, čime se dobija slika o toku projekta. Konfiguracioni fajl sadrži sve podatke potrebne za konfigurisanje ostalih komponenti i omogućava veliku fleksibilnost pri pokretanju testova. Najviši nivo hijerarhije predstavlja klasa **test** u kojoj se definišu okruženje **env**, konfigurisanje okruženja i startovanje sekvenci prema sekvenceru.

4.2 Projektovanje verifikacionog okruženja za dizajnirani IP blok

Izvorni kodovi svih blokova, kao i ceo verifikacioni deo se mogu naći u Dodatku A [3].

4.2.1 Projektovanje sekvenci i sekvencera

Kao što je već rečeno uloga sekvencera je da prenese sekvencu transakcija, napisanu na visokom nivou apstrakcije, drajveru. Sekvenca se sastoji od niza transakcija, pri čemu klasa koja opisuje definiše vrednosti jedne transakcije, ima veliki značaj, jer određuje format i ograničenja signala. U ovom radu, to je klasa **eval_base_seq**, koja nasleđuje baznu klasu **uvm_sequence_item**.

Osnovi format transakcija opisan je na sledeći način:

```
rand bit start_in;
bit reset;

rand bit side_in;
bit color_in[64];
bit [2:0] piece_in[64];

//result
int result_axi_out;
bit finished_axi_out;
```

Listing Koda 3

Polja **rand** označavaju da će promenjive do njih biti randomizovane u koliko se pozove ugrađena metoda svake klase – **randomize**. Zajedno sa ovom metodom se pozivaju i metode **pre_randomize** i **post_randomize** u koliko su definisane. Upravo ova činjenica korištena je za generisanje vrednosti promenjiva **color_in** i **piece_in**, koje su zahtevale fina ograničenja. U ovoj klasi su postavljena sledeća ograničenja:

- Na šahovskoj tabli se nikad neće naći beli pešaci na osmom redu, odnosno crni na prvom
- Broj pešaka jednog igrača ne može biti veći od osam
- Uvek postoji po jedan kralj na obe strane
- Broj ostalih figura istog tipa može biti i veći nego na početku partije, ukoliko su one promovisane. Broj promovisanih figura po igraču može biti maksimalno osam, s tim da zbir preostalih pešaka i promovisanih figura mora biti manji ili jednak sa osam.

Malo verovatno je ipak da neko u partiji promoviše većinu pešaka i u cilju ispitivanja što više realnih pozicija uvedena je težinska distribucija na randomizaciju broja figura, koji su uvedeni preko pomoćnih promenljivih. Ovo je opisano sledećim delom koda:


```

rand int w_pawn_count;      rand int b_pawn_count;
rand int w_knight_count;    rand int b_knight_count;
rand int w_bishop_count;    rand int b_bishop_count;
rand int w_rook_count;      rand int b_rook_count;
rand int w_queen_count;     rand int b_queen_count;

constraint const_w_total_promoted{
    w_pawn_count + (w_knight_count <= 2 ? 0 : (w_knight_count - 2)) +
    (w_bishop_count <= 2 ? 0 : (w_bishop_count - 2)) +
    (w_rook_count <= 2 ? 0 : (w_rook_count - 2)) +
    (w_queen_count <= 1 ? 0 : (w_queen_count - 1)) <= 8;
}

constraint const_b_total_promoted{
    b_pawn_count + (b_knight_count <= 2 ? 0 : (b_knight_count - 2)) +
    (b_bishop_count <= 2 ? 0 : (b_bishop_count - 2)) +
    (b_rook_count <= 2 ? 0 : (b_rook_count - 2)) +
    (b_queen_count <= 1 ? 0 : (b_queen_count - 1)) <= 8;
}

constraint const_w_pawn_count{w_pawn_count inside{[0:8]};}
constraint const_w_knight_count{
    w_knight_count dist{[0:2]:/900, 3:=70, 4:=20, 5:=9, [6:8]:/1 };
}
constraint const_w_bishop_count{
    w_bishop_count dist{[0:2]:/900, 3:=70, 4:=20, 5:=9, [6:8]:/1 };
}
constraint const_w_rook_count{
    w_rook_count dist{[0:2]:/900, 3:=70, 4:=20, 5:=9, [6:8]:/1 };
}
constraint const_w_queen_count{
    w_queen_count dist{[0:1]:/850, 2:=70, 3:=50, 4:=20, 5:=9, [6:8]:/1};
}

constraint const_b_pawn_count{b_pawn_count inside{[0:8]};}
constraint const_b_knight_count{
    b_knight_count dist{[0:2]:/900, 3:=70, 4:=20, 5:=9, [6:8]:/1};
}
constraint const_b_bishop_count{
    b_bishop_count dist{[0:2]:/900, 3:=70, 4:=20, 5:=9, [6:8]:/1};
}
constraint const_b_rook_count{
    b_rook_count dist{[0:2]:/900, 3:=70, 4:=20, 5:=9, [6:8]:/1};
}
constraint const_b_queen_count{
    b_queen_count dist{[0:1]:/850, 2:=70, 3:=50, 4:=20, 5:=9, [6:8]:/1};
}

```

Listing Koda 4

Oznaka [0:2]:/ 900 označava da je težina svih članova niza zajedno devetsto. Suma težina po promenljivoj je 1000, pa se težine mogu posmatrati i kao promili.

Za randomizovanje promenljivih color_in i piece_in bilo je potrebno utvrditi broj figura, da bi se zatim u zavisnosti od toga one rasporedile po tabli. Ovo je urađeno u metodi post_randomize metodi u Dodatku B[4]. Ideja post-randomizacije je da se polja upišu u listu i da se na slučajan način bira polje iz liste za određenu figuru, čiji broj je već određen u randomizaciji. Ovim se dobija uniformna raspodela po poljima i jedinstvenost figure na polju.

Klasa **eval_base_seq** koristi se za definisanje niza transakcija. Ova klasa nasleđuje klasu **uvm_sequence** i kao parametar prosleđuje format jedne transakcije opisan klasom **eval_frame**. Ona u sebi definiše sekvencer, koji generiše niz transakcija, tako što prvo kreira objekat tipa **eval_frame**, potom ga randomizuje i šalje drajveru, preko sekvencera. Ovo je urađeno korišćenjem makroa **`uvm_do(req)**, implicitno podrazumeva opisane korake.

Što se tiče projektovanja sekvencera on nasleđuje baznu klasu **uvm_sequencer** pri čemu se prosleđuje parametar sekvence – **eval_frame**. Bazna klasa implementira sve potrebne metode tako da klasa **eval_sequencer** implementira samo konstruktor.

4.2.2 Projektovanje drajvera

Drajver je preko TLM(*eng. transaction level modeling*) porta povezan na sekvencer kako bi vršio komunikaciju i sadrži virtuelni interfejs da bi slao signale IP bloku. Da bi dobio novu transakciju od sekvencera, klasa drajver poziva metodu **get_next_item**, koja u blokirajućem maniru zahteva novi objekat od sekvencera i vraća pokazivač na njega, kada je objekat poslat(Listing Koda 3).

```
task run_phase(uvm_phase phase);
    forever begin
        seq_item_port.get_next_item(req);
        `uvm_info(get_type_name(),
            $sformatf("DRIVER EVAL: Driver sending...\n%s",
                req.sprint()), UVM_HIGH)

        drive_tr();

        @(posedge vif.finished_axi_out);

        seq_item_port.item_done();
    end
endtask : run_phase
```

Listing Koda 5

Kada se dobije nova transakcija, potrebno je generisati signale koji opisuju tu transakciju i razumljive su hardveru. Ovo je urađeno taskom **driver_tr()** datom u Dodatku B[6]. Po završetku slanja transakcije, čeka se pozitivna ivica signala **finished_axi_out**, koja označava da je IP blok evaluirao poziciju. Potom se završava rukovanje sa sekvencerom i započinje se čekanje nove sekvence.

4.2.3 Projektovanje monitora

Uloga monitora je u principu suprotna od uloge drajvera. On uzima signale sa ulaznih i izlaznih portova IP bloka podiže nivo apstrakcije i prosleđuje transakcije skorbordu. Monitor koristi virtuelni interfejs, kako bi došao do signala i šalje in skorbordu preko TLM intefejsa. Da bi se signali slali prema skorbordu definisan je **uvm_analysis_port** sa parametrom **eval_frame**. Ovaj port nam omogućava neblokirajuće slanje transakcija, pri čemu se u komponenti koja prima transakcije implementira funkcija **write()**. Da bi pokupio signale pokreću se dva task-a – **collect_input** i **collect_result** (Listing Koda 4).

```

task run_phase(uvm_phase phase);
    forever begin
        current_frame = eval_frame::type_id::create("current_frame", this);

        fork
            // collect transactions
            collect_input();
            collect_result();
        join

        $display("%0t MONITOR salje rezultat DUV-a: %d
                za poziciju: ", $time, current_frame.result_axi_out);
        current_frame.print_board();

        item_collected_port.write(current_frame);

        cov.collect_coverage(current_frame.color_in, current_frame.piece_in,
                             current_frame.side_in);
    end
endtask : run_phase

```

Listing Koda 6

U Dodatku B[7] dat je listing taskova iz fork_join konstrukcije. Task collect_input u sebi ima dva paralelna procesa koji prikupljaju ulaze u IP blok, pri čemu jedan prikuplja signale memorijskih portova, dok drugi prikuplja signale side_in i start_in. Kada naiđe rastuća ivica start signala i signal start_axi_out bude '1', oba procesa se terminiraju pri čemu se trenutno stanje upisuje u lokalnu promenjivu current_frame.

Drugi task collect_result čeka na rastuću ivicu signala za kraj evaluacije i rezultat upisuje u current_frame. Potom se fork-join konstrukcija završava, ispisuje se poslata pozicija na ekran, dok se funkcijom write dobijena transakcija šalje skorbordu na validaciju.

Objekat cov predstavlja modul, koji meri pokrivenost. Pozivom metode collect_coverage šalju se podaci o poziciji i igraču na potezu, koji se dalje utiču na pokrivenost, što će biti opisano kasnije.

4.2.4 Projektovanje agenta

Klasa eval_agent nasleđuje klasu uvm_agent i služi za spajanje objekata klasa eval_driver, eval_sequencer i eval_monitor. Pri ovome se koriste podaci iz konfiguracionog fajla, koji u zavisnosti da li je agent aktivan ili pasivan povezuje njegove unutrašnje komponente. U koliko je agent aktivan inicijalizuju se sve tri komponente i povezuju se objekti klasa eval_driver i eval_sequencer korišćenjem metode connect, preko njihovih ugrađenih portova – seq_item_port i seq_item_export. U koliko agent nije aktivan instancira se samo monitor.

4.2.5 Projektovanje skorborda

Skorbord, opisan klasom eval_scoreboard, implementira funkciju write() koja se poziva pri slanju iz monitora. Kao parametar ove funkcije šalju se signali u objektu klase eval_frame. Iz ovog objekta se signali koji opisuju poziciju proslođuju u funkciju predict, koja služi kao referentni model i vraća očekivani

rezultat. Ovaj rezultat se upoređuje sa dobijenim rezultatom iz IP bloka. U koliko su rezultati isti ispisuje se poruka da je test prošao, dok ako se razlikuju simulacija se zaustavlja i ispisuje se poruka o grešci.

4.2.6 Projektovanje modula za skupljanje pokrivenosti

Pokrivenost sadrži u sebi dve metrike i to su strukturna pokrivenost (*eng. Code coverage*) i funkcionalna pokrivenost (*eng. Functional coverage*). Strukturna pokrivenost daje informacije o stepenu izvršavanja svih linija koda. Drugim rečima ona nam omogućava da vidimo da li postoji deo koda koji se nikad ne aktivira. Postoje više tipova strukturne pokrivenosti kao na primer:

- Pokrivenost naredbi(*eng. Statement coverage*) – koja proverava da li su sve naredbe u kodu izvršene
- Pokrivenost grananja(*eng. Branch*) – utvrđuje da li su uslovi grananja u kodu evaluirani i kao tačni i kao netačni
- Pokrivenost FSM stana – utvrđuje da li je dizajn posetio sva stanja

Pored ovih postoje i drugi tipovi, koji određuju pokrivenost koda na sličan način. Najbolje od svega je što je ova metrika implicitna i poziva automatski. Ipak, korišćenje samo strukturne pokrivenosti nije dovoljno, jer ne možemo biti sigurni da projektovani dizajn ispunjava zadate funkcionalnosti.

Upravo zbog ovoga koristi se funkcionalna pokrivenost, koja utvrđuje da li dizajn implementira sve zahtevane osobine. Da bi se ovo uradilo potrebno je definisati, koje vrednosti je potrebno pratiti. U SystemVerilog-u moguće je implementirati ovu metriku korišćenjem grupa pokrivenosti (*eng. Cover groups*). Ove grupe specificiraju koji signali se prate i koje vrednosti se moraju pojaviti.

U našem slučaju, definisane su četiri grupe pokrivenosti koje se odnose na pokrivenost broja pešaka, broja ostalih tipova figura, pokrivenost polja svim tipovima figura i igrača na potezu. Ovo se može videti u Dodatku B[8]. Kako je bilo potrebno napisati grupu pokrivenosti niza, ove grupe su implementirane van klase `eval_coverage`, jer bi da su u njoj, postali objekti, odnosno ugrađene grupe (*eng. embedded groups*). Pri tome je bilo potrebno predefinisati funkciju **sample**, koja odabira trenutak pamćenja signala, tako da stvara korpe (*eng. bins*) za prosleđeni parametar niza. Rezultati pokrivenosti dati su u poglavlju 4.3.

4.2.7 Projektovanje okruženja

Klasa koja opisuje verifikaciono okruženje je **eval_env**. U ovoj klasi se instanciraju objekti klase `eval_agent` i `eval_scoreboard`. Pored ovoga, u konekt fazi(*eng. connect_phase*) se povezuju monitor iz agenta sa skorboardom naredbom `connect`, na isti način na koji su povezani i drajver i sekvencer. Pri ovome se koriste portovi `item_collected_port`, na strani monitora i `frame_collected`, na strani skorboarda.

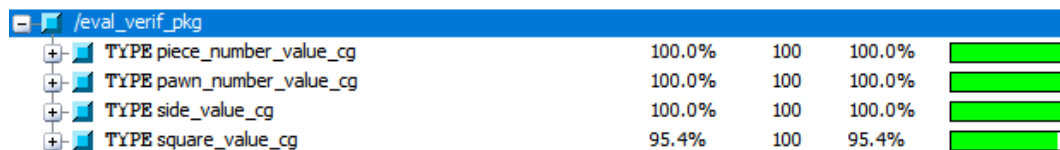
4.2.8 Projektovanje top modula i povezivanje sa IP modulom

U top modulu se instancira IP blok koji se verifikuje i povezuje se sa virtuelnim interfejsima, preko kojih komunicira sa ostatkom verifikacionog okruženja. Ovo je potrebno uraditi jer je dizajn sam po sebi statička komponenta, dok se komponente verifikacionog okruženja dinamički alociraju. U ovom modulu se još definišu takt i reset, pri čemu se još i pokreće simulacioni test.

4.3 Testovi i skupljanje pokrivenosti

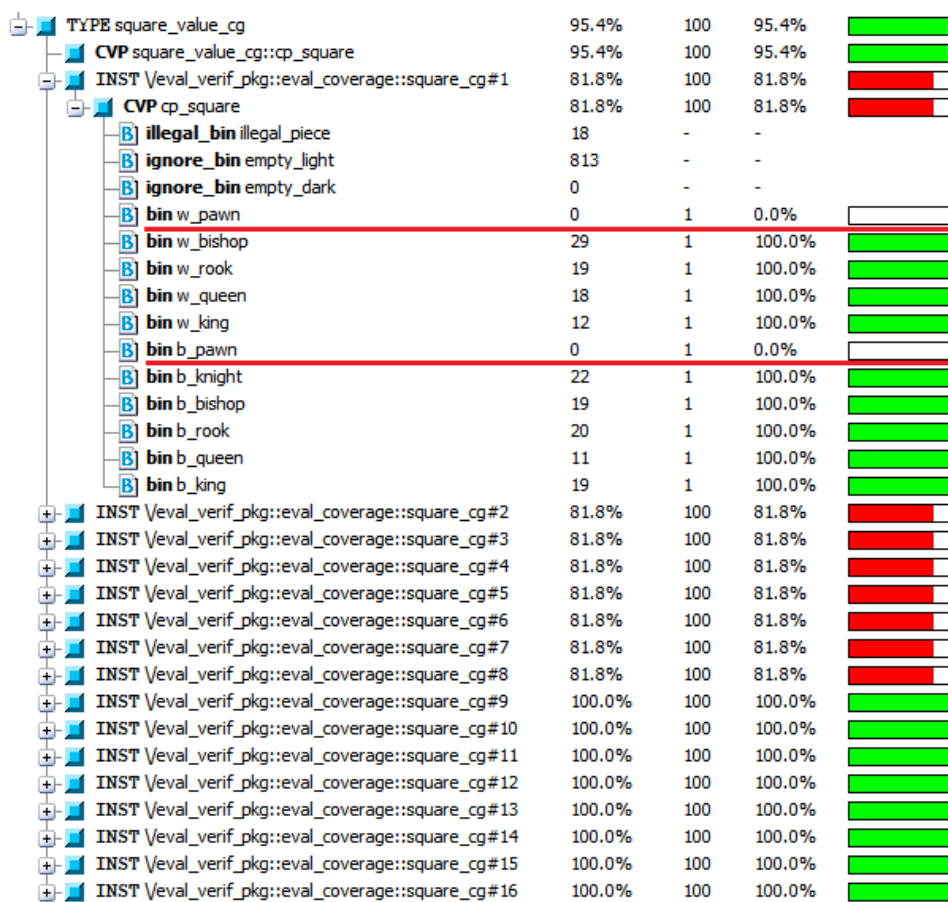
Da bi pokrenuli test potrebno je pokrenuti skriptu datu u Dodatku[9]. Prilikom pokretanja simulacije prvo se kreira biblioteka u koju se smeštaju kompajlirani fajlovi koji opisuju IP blok i verifikaciono okruženje. Zatim se optimizuje top modul i omogućava prikupljanje pokrivenosti, nakon čega se pušta test sa pseudo-slučajnim izvorom za randomizaciju. Na kraju se čuva fajl koji se dobija prikupljanjem pokrivenosti, pokreće se prikaz signala i startuje se simulacija.

Nakon pokretanja simulacije, podaci se mogu analizirati u Code Coverage Analysis, Covergroups, Instance Coverage i Coverage Details prozorima. U samoj simulaciji se pokreće 1000 testova na osnovu kojih se dobija funkcionalna pokrivenost data na Slici 30. Razlog zbog čega grupa square_value_cg nije pokrivena do kraja je zbog toga što na prvom i osmom redu nikad neće biti pešaka, što se vidi na Slici 31.



/eval_verif_pkg				
TYPE piece_number_value_cg	100.0%	100	100.0%	<div></div>
TYPE pawn_number_value_cg	100.0%	100	100.0%	<div></div>
TYPE side_value_cg	100.0%	100	100.0%	<div></div>
TYPE square_value_cg	95.4%	100	95.4%	<div></div>

Slika 30 (Funkcionalna pokrivenost)



Slika 31 (Pokrivenost dela grupe square_value_cg)

Pored funkcionalne pokrivenosti, strukturna pokrivenost se može videti na Slici 32. Razlog zbog čega modul select_pieces nema punu pokrivenost je što u fifo registrima neće biti popunjena sva mesta, zbog čega se neće ući u delove koda koji određuju logiku rada fifo bafera datu u modulu fifo_reg.vhd.

Scope ◀	TOTAL ◀	Statement ◀	Branch ◀	FSM State ◀
TOTAL	84.58%	96.61%	93.98%	100.00%
eval_ip	94.22%	100.00%	--	--
module_select_pieces	86.36%	94.56%	91.12%	100.00%
module_w_eval_pawn	92.57%	100.00%	100.00%	100.00%
module_b_eval_pawn	92.57%	100.00%	100.00%	100.00%
module_pawn_rank	100.00%	100.00%	100.00%	100.00%
module_w_material_of_pieces	93.65%	100.00%	100.00%	100.00%
module_b_material_of_pieces	93.54%	100.00%	100.00%	100.00%
module_w_eval_king	92.37%	100.00%	100.00%	100.00%
module_b_eval_king	92.70%	100.00%	100.00%	100.00%
module_adder	92.22%	100.00%	100.00%	100.00%

Slika 32 (Strukturna pokrivenost)

Glava 5

Zaključak

Projektovanje hardversko-softverskog rešenja za igranje šaha sastoji se od iz projektovanja na sistemskom nivou, projektovanje IP modula korišćenjem RT metodologije i verifikacije projektovanog bloka. Korak particionisanja i analize u sistemskom dizajnu omogućio nam se da uočimo uska grla u dizajnu i upravo te funkcionalnosti projektujemo u hardveru, kako bi dobili najbolje performanse. Prilikom spuštanja apstrakcije modela i diskusije mogućih implementacija bilo je važno ostati otvorenog uma i razmatrati različita rešenja u odnosu na ono koje je već implementirano u softveru. Odabir same ideje za implementaciju zasnivao se na proučavanju međuzavisnosti samog koda, koje je omogućilo uvođenje protočne obrade po kolonama šahovske table. Na ovaj način, umesto da smo redom ispitivali svih 64 polja, u osam koraka dobili smo pozicije svih figura. U cilju optimizacije, bilo je potrebno izdvojiti datapath blokove od bloka za upravljanje, pri čemu se vodilo računa o sinhronizaciji i brzini izračunavanja. Sam dizajn za evaluaciju pozicije radi preko pedeset posto brže od softverskog rešenja pri čemu je učestalost takta daleko manja nego kod savremenih procesora, i na ovaj način se može napraviti značajna ušteda energije. Ovakav način projektovanja mogao bi se primeniti na slične aplikacije u oblasti igara, gde je potrebno ustanoviti jačinu igrača zasnovanu na poziciji. U daljim koracima razvoja ovog projekta ostaje da se implementira funkcija attack čime bi se još više dobilo na ubrzanju, po eksponencijalnom zakonu. U postupku verifikacije proverena je funkcionalnost IP bloka u odnosu na referentni model. Da bi se što efikasnije generisale realne pozicije uvedena su ograničenja sa distribuiranom raspodelom verovatnoće. Sa druge strane, na osnovu rezultata merenja pokrivenosti ustanovljeno je kada je verifikacija uspešno završena.

Dodatak A

Kodovi

- [1] Github repozitorijum za SystemC
https://github.com/dejangrubisic/Hardware-acceleration-of-chess-engine/tree/master/SystemC_fajlovi
- [2] Github repozitorijum za VHDL
<https://github.com/dejangrubisic/Hardware-acceleration-of-chess-engine/tree/master/VER/dut>
- [3] Github repozitorijum za SystemVerilog
<https://github.com/dejangrubisic/Hardware-acceleration-of-chess-engine/tree/master/VER/verif>

Dodatak B

Listing koda

[1] top.hpp fajl

```
#ifndef _TOP_HPP_
#define _TOP_HPP_

#include <systemc>
#include "eval.hpp"
#include "soft.hpp"

using namespace sc_core;

SC_MODULE(top)
{
public:
    SC_HAS_PROCESS(top);
    top(sc_module_name);

protected:
    sc_core::sc_signal<int> s_color[64];
    sc_core::sc_signal<int> s_piece[64];
    sc_core::sc_signal<int> s_side;
    sc_core::sc_signal<bool> s_start_eval;

    sc_core::sc_signal<bool> s_return_eval;
    sc_core::sc_signal<int> s_result;
    sc_core::sc_signal<int> s_result_parallel;

    hard_acc::m_eval ev;
    soft_ip sw;
};

#endif
```

[2] top.cpp fajl

```
#include "top.hpp"
using namespace sc_core;
using namespace std;

top::top(sc_module_name n) :
    sc_module(n),
    sw("sw"), // Member initializer list
    ev("ev")
{
    cout << name() << " constructed.\n";
    //SOFT Module
    //input
```



```

sw.start_in(s_return_eval); // control signal
sw.soft_eval_in.bind(s_result); // result sequential
sw.res_par(s_result_parallel); // result parallel
//output
for(int i = 0; i < 64; i++)
{
    sw.soft_color[i].bind(s_color[i]);
    sw.soft_piece[i].bind(s_piece[i]);
}
sw.soft_side.bind(s_side);
sw.start_out(s_start_eval);

//EVAL Module
//input
for(int i = 0; i < 64; i++)
{
    ev.eval_color[i].bind(s_color[i]);
    ev.eval_piece[i].bind(s_piece[i]);
}
ev.eval_side.bind(s_side);
ev.start_in_eval(s_start_eval);
//output
ev.result_seq.bind(s_result);
ev.finished(s_return_eval); // za paralelno
ev.result_parallel(s_result_parallel);
}

```

[4] Realizacija komparatora korišćenjem for generate naredbe u select piece modulu

pieces_on_row: for i in 0 to 7 generate

```

process(color_in, piece_in)
begin

    s_w_pawn_next(i) <= '0';
    s_w_knight_next(i) <= '0';
    s_w_bishop_next(i) <= '0';
    s_w_rook_next(i) <= '0';
    s_w_queen_next(i) <= '0';
    s_w_king_next(i) <= '0';

    s_b_pawn_next(i) <= '0';
    s_b_knight_next(i) <= '0';
    s_b_bishop_next(i) <= '0';
    s_b_rook_next(i) <= '0';
    s_b_queen_next(i) <= '0';
    s_b_king_next(i) <= '0';

    case(color_in(i)&piece_in(i)) is
    ----- white -----
    when "0000" => --PAWN
        s_w_pawn_next(i) <= '1';

```

```

when "0001" => --KNIGHT
                s_w_knight_next(i) <= '1';
when "0010" => --BISHOP
                s_w_bishop_next(i) <= '1';
when "0011" => --ROOK
                s_w_rook_next(i) <= '1';
when "0100" => --QUEEN
                s_w_queen_next(i) <= '1';
when "0101" => --KING
                s_w_king_next(i) <= '1';

----- black -----
when "1000" => --PAWN
                s_b_pawn_next(i) <= '1';
when "1001" => --KNIGHT
                s_b_knight_next(i) <= '1';
when "1010" => --BISHOP
                s_b_bishop_next(i) <= '1';
when "1011" => --ROOK
                s_b_rook_next(i) <= '1';
when "1100" => --QUEEN
                s_b_queen_next(i) <= '1';
when "1101" => --KING
                s_b_king_next(i) <= '1';

when others =>

end case;

end process;

end generate;

```

[5] post_randomize metoda klase eval_frame

```

function void post_randomize();
    int square_queue[$];
    int piece_count;
    int choose_square;
    int temp; //bira iz square_queue validno polje

    for(int i = 0; i < 64; i++)
    begin
        square_queue[i] = i;
        color_in[i] = 0;
        piece_in[i] = 6; //empty
    end

    for(int color = 0; color <= 1; color++)
        for(int piece = 0; piece < 6; piece++)
            begin
                if(color == 0) begin

```

```

        case (piece)
        0: piece_count = w_pawn_count;
        1: piece_count = w_knight_count;
        2: piece_count = w_bishop_count;
        3: piece_count = w_rook_count;
        4: piece_count = w_queen_count;
        5: piece_count = 1;
        endcase
    end
    else begin
        case (piece)
        0: piece_count = b_pawn_count;
        1: piece_count = b_knight_count;
        2: piece_count = b_bishop_count;
        3: piece_count = b_rook_count;
        4: piece_count = b_queen_count;
        5: piece_count = 1;
        endcase

        end
    for(int i = 0; i < piece_count; i++)
    begin
        temp = $urandom_range(0, square_queue.size()-1);

        if(piece == 0)
            while((square_queue[temp] < 8) ||
                (square_queue[temp] > 55))
                temp = $urandom_range(0,
                    square_queue.size()-1);

        choose_square = square_queue[temp];
        square_queue.delete(temp);
        $cast(color_in[choose_square] , color);
        $cast(piece_in[choose_square] , piece);
    end
end
endfunction : post_randomize

```

[6] Implementacija taska drive_tr klase eval_driver

```

task drive_tr();
    //Drive start_in and side_in
    logic [31 : 0] temp_data_in[8];

    if(req.reset == 1'b1)
    begin
        @(posedge vif.clk);
        vif.reset = 1'b1;
    end
    else
    begin

```

```

for(int i=0; i<8; i++)
begin
    @(posedge vif.clk);
    vif.mem_wr_in = 1'b1;
    vif.mem_wr_addr_in = i;
    //Converting to format of memory 32b
    temp_data_in[vif.mem_wr_addr_in] = 0;

    for(int j=0; j<8; j++)
    begin
        temp_data_in[vif.mem_wr_addr_in] |=
        ({req.color_in[8*j+i], req.piece_in[8*j+i] } << (4*j));
    end
    vif.mem_data_in =
    temp_data_in[vif.mem_wr_addr_in];

end

@(posedge vif.clk);
vif.mem_wr_in = 1'b0;
vif.mem_data_in = 0;

vif.reg_data_in <= req.side_in;
vif.side_wr_in <= 1'b1;
@(posedge vif.clk);
vif.side_wr_in <= 1'b0;

vif.reg_data_in = req.start_in;
vif.start_wr_in = 1'b1;
@(posedge vif.clk);
vif.start_wr_in = 1'b0;

//Ispisi memorijsku mapu za tablu
$display("Memorija");
for(int i = 0; i < 8; i++)
    $display("%x", temp_data_in[i]);
end
endtask : drive_tr

```

[7] Realizacija taskova collect_input i collect_result klase eval_monitor

```

task collect_input();
    fork
        begin

            do begin
                @(posedge vif.clk );
                #1ns;
            end

```

```

        while(vif.start_axi_out != 1'b1);
            current_frame.start_in =
                vif.start_axi_out;
            current_frame.side_in =
                vif.side_axi_out;
        end
        //Upis u memoriju
        forever
        begin
            int square;
            do begin
                @(posedge vif.clk);
                #1ns;
            end
            while (vif.mem_wr_in != 1'b1);
            for(int i=0; i<8; i++)begin
                square = 8*i + vif.mem_wr_addr_in;
                current_frame.color_in[square] =
                    vif.mem_data_in[4*i+3];
                current_frame.piece_in[square][2] =
                    vif.mem_data_in[(4*i+2)];
                current_frame.piece_in[square][1] =
                    vif.mem_data_in[(4*i+1)];
                current_frame.piece_in[square][0] =
                    vif.mem_data_in[(4*i+0)];
            end
        end
        join_any
        disable fork;
    endtask : collect_input

    task collect_result();
        @(posedge vif.finished_axi_out);
        current_frame.result_axi_out =
            signed'(vif.result_axi_out);
    endtask : collect_result

```

[8] Grupe pokrivenosti implementirane u eval_coverage.sv

```

covergroup piece_number_value_cg with function sample(int
unsigned piece_number);
    option.per_instance = 1;
    cp_piece_number : coverpoint piece_number{
        bins cnt_0 = {0};
        bins cnt_1 = {1};
        bins cnt_2 = {2};
        bins cnt_3 = {3};
        bins cnt_4 = {4};
        bins cnt_5 = {5};
    }
endgroup : piece_number_value_cg

```

```

covergroup pawn_number_value_cg with function sample(int
unsigned pawn_number);
    option.per_instance = 1;
    cp_pawns_number : coverpoint pawn_number{
        bins cnt_0 = {0};
        bins cnt_1 = {1};
        bins cnt_2 = {2};
        bins cnt_3 = {3};
        bins cnt_4 = {4};
        bins cnt_5 = {5};
        bins cnt_6 = {6};
        bins cnt_7 = {7};
        bins cnt_8 = {8};
    }
endgroup : pawn_number_value_cg

covergroup side_value_cg (ref bit side);
    option.per_instance = 1;
    cp_side : coverpoint side;
endgroup : side_value_cg

covergroup square_value_cg with function sample(bit unsigned
[3:0] square);
    option.per_instance = 1;
    cp_square : coverpoint square
    {
        bins w_pawn    = { {`LIGHT, `PAWN} };
        bins w_knight  = { {`LIGHT, `KNIGHT} };
        bins w_bishop  = { {`LIGHT, `BISHOP} };
        bins w_rook    = { {`LIGHT, `ROOK} };
        bins w_queen   = { {`LIGHT, `QUEEN} };
        bins w_king    = { {`LIGHT, `KING} };
        bins b_pawn    = { {`DARK, `PAWN} };
        bins b_knight  = { {`DARK, `KNIGHT} };
        bins b_bishop  = { {`DARK, `BISHOP} };
        bins b_rook    = { {`DARK, `ROOK} };
        bins b_queen   = { {`DARK, `QUEEN} };
        bins b_king    = { {`DARK, `KING} };
        ignore_bins empty_light = { {`LIGHT, `EMPTY} };
        ignore_bins empty_dark = { {`DARK, `EMPTY} };
        illegal_bins illegal_piece = {'h7, 'hF};
    }
endgroup : square_value_cg

```

[9] Skripta za pokretanje simulacije i testiranja IP bloka

```
# Create the library.
if [ file exists work ] {
    vdel -all
}

vlib work

# compile DUT

vcom ../../dut/common.vhd
vcom ../../dut/adder.vhd
vcom ../../dut/b_eval_king.vhd
vcom ../../dut/b_eval_pawn.vhd
vcom ../../dut/w_eval_king.vhd
vcom ../../dut/w_eval_pawn.vhd
vcom ../../dut/choose_king.vhd
vcom ../../dut/reduction_coder.vhd
vcom ../../dut/fifo_reg.vhd
vcom ../../dut/fifo_top.vhd
vcom ../../dut/material_of_pieces.vhd
vcom ../../dut/memory_table.vhd
vcom ../../dut/memory_subsystem.vhd
vcom ../../dut/pawn_rank.vhd
vcom ../../dut/rank.vhd
vcom ../../dut/select_piece.vhd
vcom ../../dut/top_module.vhd
vcom ../../dut/top_with_memory.vhd

# compile testbench
vlog -sv \
    +incdir+$env(UVM_HOME) \
    +incdir+../sv \
    ../sv/eval_verif_pkg.sv \
    ../sv/eval_verif_top.sv

# run simulation
vopt eval_verif_top -o opttop +cover
vsim eval_verif_top -novopt +UVM_TESTNAME=test_eval_my_1
+UVM_VERBOSITY=UVM_LOW -sv_seed random
vsim -coverage opttop
coverage save eval_verif_top.ucdb

do wave.do
run -all
```

Literatura

- [1] <http://www.tckerrigan.com/Chess/TSCP/>
- [2] [Rastislav](#) Struharik, “Vežbe i Predavanja za predmet Projektovanje Složenih Digitalnih Sistema“, http://www.elektronika.ftn.uns.ac.rs/index.php?option=com_content&task=category§ionid=4&id=140&Itemid=139
- [3] Grant Martin, Brian Bailey, Andrew Piziali, "ESL Design and Verification: A Prescription for Electronic System Level Methodology (Systems on Silicon)"
- [4] https://sr.wikipedia.org/wiki/Alfa-beta_pretraga
- [5] https://sr.wikipedia.org/wiki/%D0%9D%D0%B5%D0%B3%D0%B0%D0%BC%D0%B0%D0%BA%D1%81_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%B0%D0%BC
- [6] [https://sr.wikipedia.org/wiki/%D0%9C%D0%B8%D0%BD%D0%B8%D0%BC%D0%B0%D0%BA%D1%81_\(%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%B0%D0%BC\)](https://sr.wikipedia.org/wiki/%D0%9C%D0%B8%D0%BD%D0%B8%D0%BC%D0%B0%D0%BA%D1%81_(%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%B0%D0%BC))
- [7] https://reference.digilentinc.com/_media/zybo:zybo_rm.pdf
- [8] Rastislav Struharik, “Vežbe i Predavanja za predmet Funkcionalna Verifikacija”, http://www.elektronika.ftn.uns.ac.rs/index.php?option=com_content&task=category§ionid=4&id=188&Itemid=209
- [9] https://en.wikipedia.org/wiki/Open_Verification_Methodology
- [10] [https://en.wikipedia.org/wiki/ERM_\(e_Reuse_Methodology\)](https://en.wikipedia.org/wiki/ERM_(e_Reuse_Methodology))
- [11] https://en.wikipedia.org/wiki/Universal_Verification_Methodology