Универзитет у Београду
Електротехнички факултет

Имплементација RISC-V процесора са подршком за екстерни увид у стање процесора

Дипломски рад

Ментор:

проф. др Захарије Радивојевић,
ванредни професор

Кандидат:

Лазар Премовић
2019/0091

Београд, Септембар 2023.

Садржај

1 Увод

Рачунари су од свог настанка па до данашњих дана нашли примену у најразличитијим областима живота. Свака од тих примена придаје различиту важност одређеним карактеристикама рачунара (нпр. перформансе, величина, цена), што је довело до настанка различитих класа рачунара. Прелазак тржишта са мејнфрејм и персоналних рачунара на данас популарне кластере великих размера и преносиве уређаје, као и експлозија јефтиних уређаја повезаних на мрежу (тзв. интернет ствари), од дизајнера хардвера захтева да преиспитају одлуке и метрике којима су се до сада водили. Ове нове класе рачунара, уместо на сирове перформансе, акценат стављају на енергетску ефикасност, тј. однос утрошене енергије и количине обрађених података. Ова промена захтева представља одличну прилику за примену алтернативних архитектура рачунара. Једну од таквих архитектура представљају *Reduced Instruction Set Computer (RISC)* процесори. Иако *RISC* архитектуре нису нова појава (први *RISC* процесори су направљени 1970-их), тренутно тржиште је веома погодно за развој оваквих процесора.

Други фактор који има значајан утицај на развој модерних процесора је смањење брзине којом се повећава број транзистора на једном чипу (тзв. *Moore*-ov закон). Последица тог смањења је да перформансе и енергетска ефикасност више не могу да се побољшавају повећањем броја транзистора и смањењем њихових димензија, већ побољшања у перформансама примарно долазе из унапређења микро-архитектуре. Што за последицу има јачање монополистичких позиција одређених компанија и тиме негативно утиче на иновативност у том пољу.

Истраживачи на Универзитету Калифорнија, Беркли су увидели ове проблеме те су дизајнирали и објавили *RISC* процесорску архитектуру отвореног кода под именом *RISC-V* [1] (изговара се *RISC five*). *RISC-V* архитектура је дизајнирана тако да буде довољно једноставна за примену у едукацији а такође и довољно моћна за примену у истраживачким [2] и комерцијалним [3] [4] пројектима. То је постигнуто модуларним приступом где постоји основна архитектура и мноштво опционих екстензија. Зато је *RISC-V* архитектура одабрана за имплементацију на процесору који је тема овог рада.

Проблем којим се овај рад директно бави се тиче отклањања грешака у софтверу који се извршава на *RISC-V* процесорима. Временом софтвер постаје све комплекснији а самим тим и подложнији грешкама. Један од најчешћих алата који се користе при проналажењу грешака је дебагер који омогућава контролу извршавања програма и увид у његово стање. Подршку за дебаговање обично пружа оперативни систем, међутим већина уграђених (енг. *embedded*) система имају веома просте (или уопште немају) оперативне системе. Код таквих система се подршка за дебаговање реализује директно у хардверу, тако што се обезбеђује посебан интерфејс преко којег се (уз помоћ посебног адаптера) циљни систем повезује на софтвер који управља дебаговањем а извршава се на десктоп рачунару.

Како је подршка за екстерно дебаговање неопходна за било коју озбиљну комерцијалну имплементацију, *RISC-V* организација је дефинисала спецификацију за екстерно дебаговање [5], коју је већина комерцијалних имплементација усвојила.

Циљ рада је имплементација *RISC-V* процесора са подршком за екстерно дебаговање која поштује званичну спецификацију [5] ради бољег разумевања, проналажења потенцијалних унапређења и процене комплексности имплементирања исте у другим истраживачким пројектима.

Имплементација је реализована на *Field Programmable Gate Array (FPGA)* чипу, те су поред симулација вршени тестови и на конфигурисаном хардверу. Имплементација се састоји од самог *RISC-V* процесорског језгра (у даљем тексту само језгро), меморије за програм и податке, модула који обезбеђује подршку за екстерно дебаговање (у даљем тексту *Debug Module (DM)*), модула који омогућава повезивање са рачунаром који управља дебаговањем (у даљем тексту *Debug Transport Module (DTM)*) и неколико периферија које доприносе живописнијој демонстрацији система. Ове компоненте су повезане коришћењем три магистрале, прва повезује језгро, меморију и периферије, друга језгро и *DM* и трећа *DM* и *DTM*.

У 2. поглављу је дат кратак опис *RISC-V* спецификације [6] и спецификације за екстерно дебаговање [5], поређење са једном комерцијалном имплементацијом, као и опис технологија и алата коришћених при изради рада. Поглавље 3 садржи детаљнији преглед целог система и његових мање битних компоненти. Поглавља 4, 5 и 6 се фокусирају на детаље имплементације језгра, *DM*-а и *DTM*-а, док се поглавље 7 односи на софтверску страну екстерног дебаговања и њену конфигурацију. У 8. поглављу се приказује методика и резултати тестирања, док се у 9. поглављу дају закључци рада.

2 Преглед спецификација и коришћених технологија

За читаоце који нису упућени у *RISC-V* екосистем, у овом поглављу је дат кратак преглед најбитнијих делова спецификације. Приказано је и поређење имплементације дате у овом раду са *Si Five Freedom E310-G002* процесором коришћеном на *Si Five HiFive RevB* [7] развојној плочи. Тај процесор је изабран због своје популарности и зато што је дизајниран за примену у уграђеним системима и као такав је по перформансама и способностима најближи имплементираном процесору. На крају је дат и кратак опис технологија и алата коришћених у изради рада.

2.1 *RISC-V* инструкцијски сет

Како би постигао примењивост у широком опсегу имплементација са различитим циљевима, *RISC-V* користи модуларан приступ, спецификација [6] прописује неколико основних инструкцијских сетова, као и велики број опционих екстензија. Основни инструкцијски сет прописује око 40 обавезних инструкција које обухватају аритметичке и логичке инструкције, инструкције за приступ меморији и инструкције контроле тока. Тренутно су ратификована два основна инструкцијска сета: *RV32I* і *RV64I*. Оба сета прописују 32 регистра опште намене а разликују се у ширини регистара (самим тиме и у величини меморијског простора) која је 32 и 64 бита респективно. Такође су предложена још два основна сета, један који смањује број регистара на 16 и један који повећава ширину регистара и меморијског простора на 128 бита.

Поред основног инструкцијског сета, *RISC-V* организација је ратификовала и 8 опционих екстензија које проширују способности процесора.

То су:

- **M** - Целобројно множење, дељење и остатак при дељењу
- **A** - Атомичне операције над меморијом
- **F** - Операције над бројевима у покретном зарезу једноструке прецизности
- **D** - Операције над бројевима у покретном зарезу двоструке прецизности
- **Q** - Операције над бројевима у покретном зарезу четвороструке прецизности
- **C** - Компримоване (16 битне) инструкције
- **Zicsr** - Читање и писање контролних и статусних регистара
- **Zifencei** - Синхронизација уписа у програмску меморију

RISC-V такође предлаже и привилеговану архитектуру [8] која садржи 3 нивоа извршавања: машински (*M*), супервизорски (*S*) и кориснички (*U*). Предложена привилегована архитектура такође подржава виртуализацију и садржи опис процедуре обраде прекида.

2.2 Подршка за екстерно дебаговање

Спецификација подршке за екстерно дебаговање [5] се састоји из 4 дела чији су мањи или већи делови опциони. То су: *DM*, *DTM*, посебан ниво извршавања у коме се процесор налази док је заустављен од стране дебагера (енг. *debug (D)* мод) и хардверски окидачи.

Debug Module (DM) прима команде које долазе од софтвера који управља дебаговањем (које долазе преко *DTM*-а і *Debug Module Interface (DMI)*-а) и на основу њих управља једним или више језгара. Већина делова *DM*-а су опциони али је неопходно да имплементирани делови омогућавају све потребне операције за успешно дебаговање, сама спецификација предлаже два подскупа спецификације који испуњавају овај услов. *DM* мора имплементирати контролу ресет сигнала језгра, механизам за покретање и заустављање језгра и приступ регистрима језгра. *DM* опционо може подржати приступ меморији из погледа језгра или коришћењем додатног газде на магистрали, извршавање произвољних инструкција и приступ контролним и статусним регистрима.

Debug Transport Module (DTM) прима команде које долазе од софтвера који управља дебаговањем одабраним протоколом и преводи их у приступе *DMI* магистрали. *DTM* може користити било који протокол али у спецификацији постоји само опис *DTM*-а који користи *Joint Test Action Group (JTAG)* протокол. Као такав, *DTM* је неопходан али није неопходно да имплементирани *DTM* користи *JTAG* протокол.

Имплементација *D* мода је обавезна и састоји се од малих промена понашања језгра у односу на *M* мод извршавања и неколико додатних контролних и статусних регистара доступних само *DM*-у.

Хардверски окидачи су опциони али могу бити имплементирани и независно од остатка спецификације јер могу бити корисни и када је подршка за дебаговање имплементирана у софтверу. Спецификација омогућава произвољан број хардверских окидача као и произвољан избор које функционалности окидача су имплементиране. У основи постоје 4 типа окидача који унутар себе имају велики број функционалности које су опционе и у које се неће улазити сада.

- Окидач на адресу или податак учитан из меморије (ово обухвата читање и упис у меморију као и дохватање и извршавање инструкције)
- Окидач на број извршених инструкција (ово је један од начина за имплементацију проласка кроз програм инструкцију по инструкцију)
- Окидач на обраду изузетка
- Окидач на обраду прекида

Када се окидач окине, у зависности од конфигурације, језгро може генерисати изузетак или прећи у *D* мод и стати са извршавањем.

2.3 Поређење са *Si Five Freedom E310-G002*

На табели 2.1 се налази поређење имплементираног процесора и подршке за дебаговање са *Si Five Freedom E310-G002* [9] по неким од параметара изнетих у претходне две секције.

Табела 2.1: Поређење *Si Five Freedom E310-G002* са имплементираним процесором

	Si Five Freedom E310-G002	Имплементирани процесор
Основни инструкцијски сет	RV32I	RV32I
Подржане екстензије	M, A, C, Zicsr	Zicsr
Подржани модови извршавања	M, U	M
Организација	Проточна обрада са 5 корака	Вишециклична
Максимални ИПЦ[1]	1	1
Фреквенција сигнала такта	до 384 MHz	35 MHz
Механизам за покретање, ресетовање и заустављање језгра	Да	Да
Приступ регистрима опште намене	Да	Да
Приступ контролним и статусним регистрима	Не	Да
Приступ регистрима без заустављања језгра	Не	Не
Извршавање произвољних инструкција	Да	Да
Величина бафера за произвољне инструкције	16 меморијских речи	16 меморијских речи
Приступ меморији из погледа језгра	Не	Да
Приступ меморији из погледа језгра без заустављања језгра	Не	Не
Број помоћних регистара[2]	1 + 1	2 + 12
Приступ меморији коришћењем додатног газде на магистрали	Не	Да
Број хардверских окидача	8	4
Окидач на адресу или податак подржан	Да	Да[3]
Окидач на број извршених инструкција подржан	Не	Не
Окидач на обраду изузетка подржан	Не	Не
Окидач на обраду прекида подржан	Не	Не
DTM протокол	*JTAG*	*JTAG*
Препоручена фреквенција *JTAG* интерфејса	4MHz	1MHz
Интегрисан *JTAG* адаптер	Да (J-Link OB)	Не

[1] Инструкције По Циклусу (ИПЦ).

[2] Мапираних као контролни и статусни регистри + меморијски мапираних.

[3] Окидач не подржава комплетан сет опционих функционалности.

Као што се може видети на табели 2.1, *E310-G002* је знатно софистицираније процесор, што је и разумљиво, јер је у питању комерцијална имплементација. Међутим подршка за дебаговање на *E310-G002* обухвата један од препоручених минималних подскупа функционалности. Имплементирани подскуп функционалности се базира на извршавању произвољних инструкција (функционалности које нису директно подржане се могу емулирати извршавањем произвољног кода који користи помоћне регистре).

Како је фокус рада примарно на подршци за дебаговање, нешто једноставнији процесор са опширнијом подршком за дебаговање представља логичан избор.

2.4 Преглед коришћених технологија и алата

У овој секцији се наводе и укратко описују технологије и алати који су коришћени при изради рада. Уколико је применљиво, наводе се тачни модели или верзије коришћеног хардвера и софтвера, заједно са референцама на њихову документацију.

2.4.1 *FPGA*

Field Programmable Gate Array (FPGA) су интегрисана кола чија се функционалност може мењати по потреби. За разлику од процесора који су такође програмабилни, *FPGA* не извршава код већ директно имплементира тражени дизајн на нивоу дигиталне логике. Дизајн за *FPGA* је репрезентован битским током (енг. *bitstream*) који посебни алати за синтезу генеришу користећи дизајн написан у неком од језика за опис хардвера.

FPGA чипови су интерно реализовани коришћењем логичких елемената (енг. *Logic Element (LE)*) који се састоје од лукап табеле (која може да репрезентује произвољну комбинациону логику) и Д флип-флопа. Велики број логичких елемената (неколико десетина хиљада) је међусобно повезано конфигурабилним везама тако да се произвољни улази и излази логичких елемената могу повезати. *FPGA* чипови поред ове конфигурабилне логике често имају и додатне компоненте које олакшавају имплементацију одређених решења, то су обично интегрисане меморије, сабирачи, множачи, фазно закључане петље (енг. *Phase Locked Loop (PLL)*) итд.

За имплементацију процесора приказаног у овом раду коришћен је *Altera Cyclone V 5CSXFC6D6F31C6N* [10] *FPGA* чип. *5CSXFC6D6F31C6N* садржи 110 хиљада логичких елемената, 5761 килобита интегрисане меморије, 6 фазно закључаних петљи, 2 интегрисана меморијска контролера и двојезгарни *ARM* микропроцесор (меморијски контролери и *ARM* микропроцесор нису коришћени у овом раду).

2.4.2 *System Verilog*

System Verilog [11] је језик за опис хардвера, настао као проширење на *Verilog*. Језици за опис хардвера омогућавају дизајнерима хардвера да коришћењем програмског кода, на једноставан начин опишу структуру и понашање жељеног хардверског дизајна. Дизајн описан у језику за опис хардвера се након креирања може симулирати, претворити у шаблон за производњу интегрисаних кола или битски ток за конфигурацију *FPGA* чипа.

Једна од предности *System Verilog*-а у односу на *Verilog* је постојање интерфејса који представљају именовану групу сигнала, што их чини идеалним за репрезентовање магистрала и представља примарни разлог његовог избора за имплементацију процесора у овом раду.

2.4.3 *Quartus* и *Questa*

За синтезу дизајна написаног у језику за опис хардвера коришћен је *Intel Quartus Prime Lite* [12] верзија *22.1std.1*, док је за симулацију и прелиминарно тестирање дизајна коришћена *Questa Intel Starter FPGA Edition-64* [13] верзија *2012.2*.

2.4.4 *Eclipse Embedded CDT*

Eclipse је лако проширив интегрисано развојно окружење (енг. *Integrated Development Environment (IDE)*) које подржава велики број програмских језика, а уз *Embedded C/C++ Development Tools (CDT)* [14] сет екстензија је специјализовано за развој уграђених система. *Embedded CDT* сет екстензија између осталог садржи подршку за *Make* систем за превођење кода, као и директну подршку за дебаговање коришћењем *OpenOCD*-а или *J-Link* софтвера. Из тих разлога се за сав развој и дебаговање софтвера који се извршава на имплементираном процесору користи *Eclipse Embedded CDT* верзија *2023-06*.

2.4.5 *GCC* и *OpenOCD*

GNU Compiler Collection (GCC) [15] је добро познати пакет програмских преводилаца и пратећих алата, који обухвата и укрштени преводилац (енг. cross-compiler) за *RISC-V* архитектуру. За превођење софтвера писаног у *C* програмском језику за имплементирани процесор коришћена је верзија *13.2.0 GCC*-а која је део пакета *xPack GNU RISC-V Embedded GCC x86_64* који је препоручен од стране *Eclipse Embedded CDT* развојног окружења.

Open On-Chip Debugger (OCD) [16] је пројекат отвореног кода који за циљ има да премости јаз између унификованог интерфејса који *GNU Debugger (GDB)* пружа програмерима и интерфејса који пружа сам циљни процесор. Ово је реализовано кроз три нивоа апстракције. На самом дну имамо ниво адаптера за дебаговање, који комуницира са адаптером за дебаговање и користећи њега извршава просте операције комуникационог протокола који циљни процесор користи. Изнад тога се налази ниво архитектуре процесора који операције, као што су читање регистара или заустављање процесора, преводи у секвенцу операција комуникационог протокола који се користи. На врху се налази ниво конкретног система где се архитектури процесора додају информације о броју језгара, меморијској мапи и повезаним периферијама.

Како *OpenOCD* већ има уграђену подршку за верзију спецификације [5] коришћену у овом раду, као и *J-Link* (коришћени адаптер за дебаговање), потребно је само дефинисати конфигурациони фајл за ниво система. У раду је коришћена верзија *0.12.0*, такође део пакета *xPack OpenOCD* препорученог од стране *Eclipse Embedded CDT*.

2.4.6 *JTAG*

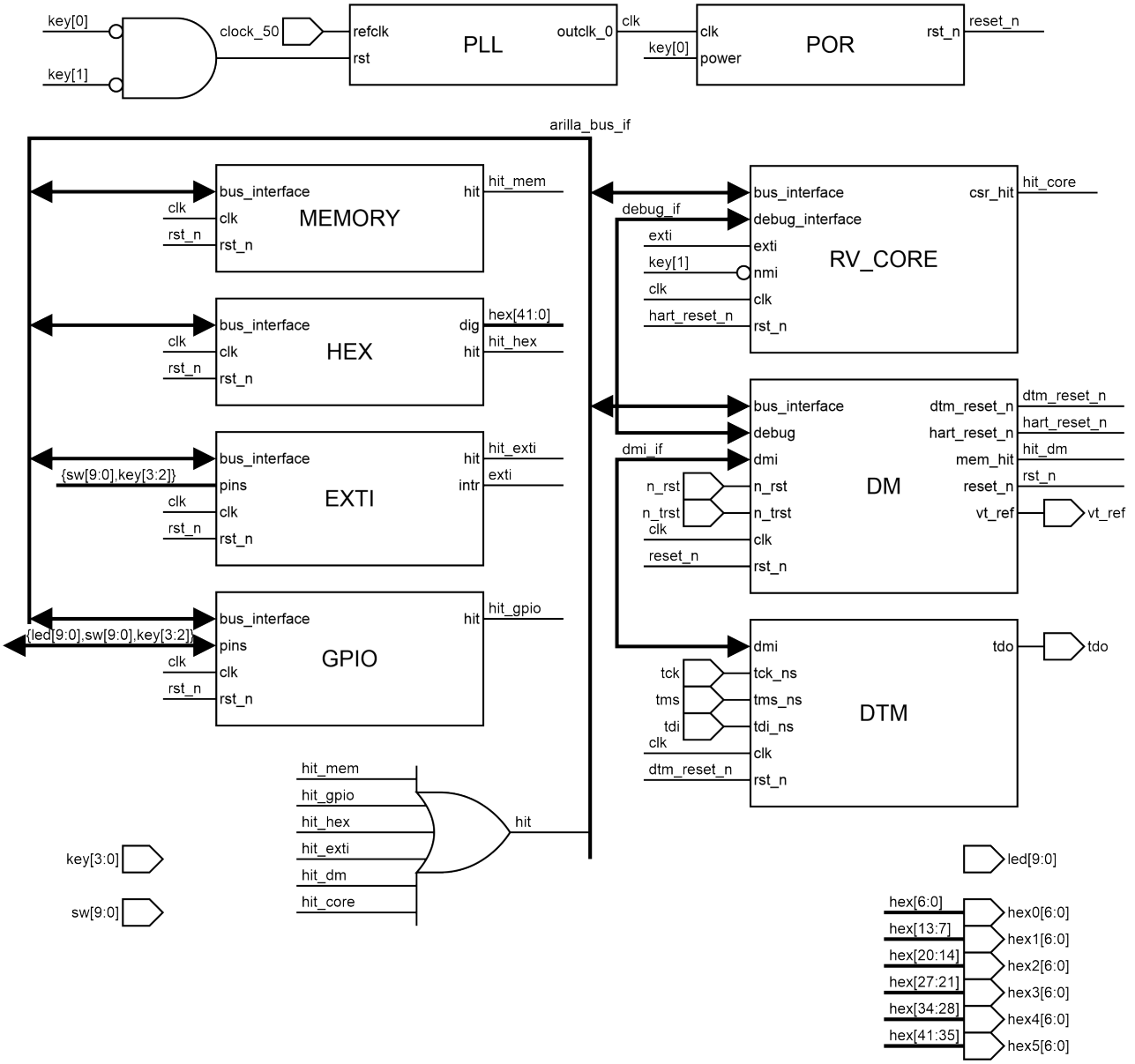
Joint Test Action Group (JTAG) [17] је комуникациони протокол, оригинално дизајниран за тестирање интегрисаних кола и штампаних плоча, који је убрзо усвојен и од стране произвођача микропроцесора као протокол за комуникацију са подршком за екстерно дебаговање. Протокол је детаљније објашњен у поглављу 6 које се бави имплементацијом *DTM*-а.

2.4.7 *J-Link*

J-Link [18] је један од најпопуларнијих адаптера за дебаговање који подржава *JTAG* протокол. *J-Link* се може у овом случају користити на два начина, директно са пратећим софтвером или коришћењем *OpenOCD*-а. Уколико се користи са пратећим софтвером, тај софтвер игра сличну улогу као *OpenOCD* али због бољег познавања хардвера може ефикасније да га искористи што резултује у бољем времену одзива при дебаговању. Уколико се користи *OpenOCD*, *J-Link* се понаша као прост адаптер и *OpenOCD* мора да управља њиме на нижем нивоу. Иако су предности првог приступа очигледне, у духу пројекта отвореног кода у поглављу 7 су приказана оба метода.

3 Преглед имплементираног система

На слици 3.1 су приказане компоненте које сачињавају имплементирани систем, *RISC-V* језгро, *DM* і *DTM* су објашњени у својим поглављима, док су остале компоненте укратко објашњене у овом поглављу.



Слика 3.1: Дијаграм комплетног система

Систем је дизајниран да буде конфигурабилан, што је постигнуто комбинацијом параметризованих компоненти и конфигурационих фајлова са дефиницијама макроа који садрже конкретне вредности параметара. Овај комбиновани приступ користи предности параметара (провера типова, инстанцирање исте компоненте са различитим вредностима параметара) а додатно сакупља све подесиве вредности на једно место.

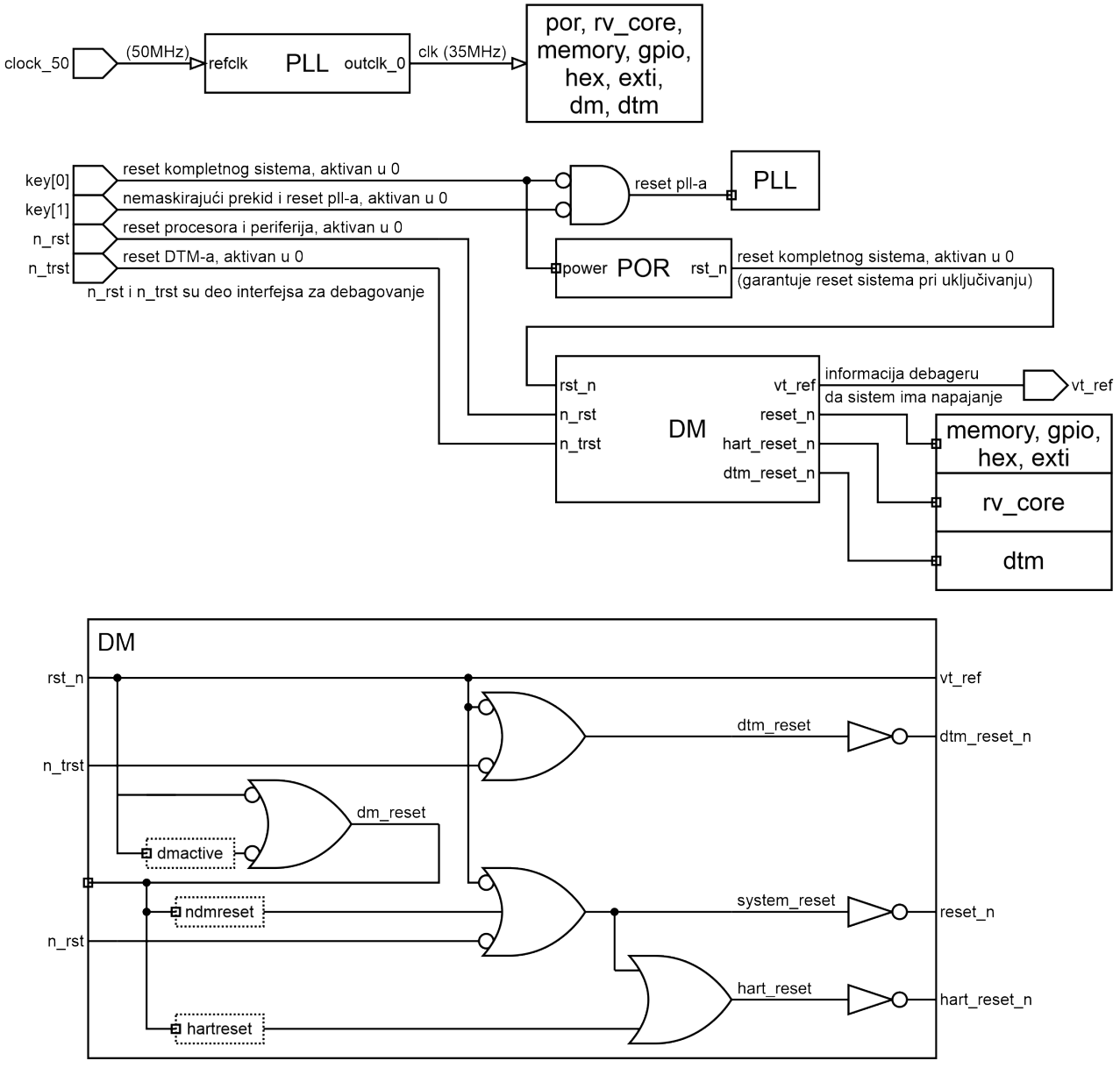
```
`ifndef SYSTEM__SVH
`define SYSTEM__SVH

`define SYSTEM__XLEN 32
`define SYSTEM__RNUM 32
`define SYSTEM__RVEC 32'd0
`define SYSTEM__NMI  32'd4
`define SYSTEM__TVEC 32'd8
`define SYSTEM__VECT 1'b1

`define SYSTEM__ALEN `SYSTEM__XLEN
`define SYSTEM__BLEN 8

`define SYSTEM__MEM_BASE 32'h0000_0000
`define SYSTEM__MEM_SIZE (64 * 1024)
//`define SYSTEM__MEM_SIZE (32 * 1024)
`define SYSTEM__MEM_INIT ''C:/Users/lazar/Desktop/RISC-V_Debug/test/program.mif''
`define SYSTEM__MEM_HINT ''ENABLE_RUNTIME_MOD=YES,INSTANCE_NAME=MAIN''

`define SYSTEM__TIME_BASE 32'hF000_0000

`define SYSTEM__GPIO_BASE 32'h1000_0000
`define SYSTEM__GPIO_NUM  22
`define SYSTEM__GPIO_MASK 22'h3FF000

`define SYSTEM__HEX_BASE 32'h2000_0000
`define SYSTEM__HEX_NUM  6

`define SYSTEM__EXTI_BASE 32'h3000_0000
`define SYSTEM__EXTI_NUM  12

//`define SYSTEM__POR_TIME 35_000_000
`define SYSTEM__POR_TIME 1

`define SYSTEM__DMI_ALEN 7

`define SYSTEM__DM_BASE 12'hF80

`endif  //SYSTEM__SVH
```

Исечак кода 3.1: Пример конфигурације система

Конфигурација система приказана у исечку 3.1 на почетку дефинише параметре самог језгра: ширину и број регистара, фиксне адресе на којима се налазе прекидне рутине за ресет и немаскирајући прекид, подразумевана вредност адресе прекидне рутине за остале прекиде и изузетке, подразумевану вредност бита који означава да ли су прекиди векторисани или не, ширину меморијске адресе и бајта у битима. Након тога је дефинисана меморијска мапа са базним адресама и величинама за меморију и периферије. На крају се налази конфигурација трајања ресет сигнала, ширина адресе *DMI* магистрале и базна адреса помоћних регистара за дебаговање.

3.1 Сигнал такта и ресет система

У овој секцији су описане компоненте и сигнали који учествују у генерисању сигнала такта за остатак система и мрежи ресет сигнала. Схема ових компоненти и сигнала се налази на слици 3.2, на којој су обележене фреквенције сигнала такта, као и описи мање очигледних сигнала. Сигнали са стрелицом која улази у компоненту или списак компоненти представља сигнал такта за те компоненте, док квадратић представља ресет сигнал те компоненте.



Слика 3.2: Дијаграм сигнала такта и ресет сигнала

3.1.1 Сигнал такта

Развојна плоча на себи има екстерни генератор сигнала такта који генерише такт фреквенције 50MHz. Екстерни сигнал такта улази на чип кроз пин **clock_50** и иде директно у фазно закључану петљу која генерише сигнал такта фреквенције 35MHz, који се користи за остатак система. Како је фазно закључана петља део не конфигурабилног хардвера *FPGA* чипа, њен опис је генерисан помоћу чаробњака унутар *Quartus* софтверског пакета. При генерисању фазно закључане петље коришћењем чаробњака, довољно је унети само фреквенцију улазног сигнала и жељену фреквенцију.

3.1.2 Ресет система

Систем у потпуности користи синхроне ресет сигнале због ефикасније имплементације на *FPGA* чиповима. Постоји 6 ресет домена и то су:

- Домен фазно закључане петље
- Домен ресета при покретању (енг. *Power-on Reset (POR)*)
- Домен *DM*-а
- Домен *DTM*-а
- Домен периферија и меморије
- Домен језгра

Фазно закључана петља се може ресетовати само уколико је комплетан остатак система такође под ресетом, што је могуће извршити активирањем захтева за немаскирајући прекид (**key[1]**) док је ресет комплетног система (**key[0]**) такође активан. Овакав неинтуитиван начин ресетовања је прихватљив јер се потреба за ресетовањем фазно закључане петље не очекује ван тестирања система. Модул за ресет при покретању се као што му само име сугерише, аутоматски ресетује при покретању система (тј. након конфигурисања *FPGA* чипа) и када ресет комплетног система (**key[0]**) има активну вредност. Када се ресетује, модул за ресет при покретању задржава остатак система ресетованим подесиви број тактова.

Преостали домени су нешто комплекснији јер корисник има већу контролу над ресетовањем индивидуалних домена те њима управља *DM*. Поред ова 4 домена, *DM* управља још једним сигналом уско везаним за ресет система, а то је сигнал **vt_ref** који даје до знања адаптеру за дебаговање да ли уређај има напајање, као и његов напон. Сам *DM* се ресетује при ресету комплетног система или уписом нуле у регистар **dmactive**[1]. *DTM* се ресетује комплетним ресетом система или активном вредношћу сигнала **n_trst** који је део конектора за екстерно дебаговање. Периферије и меморија[2] се могу ресетовати комплетним ресетом система, сигналом **n_rst** (такође део интерфејса за екстерно дебаговање) или битом **ndmreset** контролног регистра *DM*-а. Поред ресета периферија и меморије (који обухвата и само језгро), могуће је ресетовати и искључиво језгро коришћењем бита **hartreset** контролног регистра *DM*-а.

[1] Упис у **dmactive** регистар је могућ и док је остатак *DM*-а у ресету.

[2] Треба навести да садржај меморије остаје непромењен.

3.2 *Arilla bus* магистрала

Arilla bus је магистрала за приступ меморији коришћена у ауторовим ранијим имплементацијама *RISC-V* процесора [19], а која је значајно редизајнирана и унапређена за потребе овог рада. У питању је синхрона магистрала са атомичним циклусима која може да обради једну операцију сваког такта. Кашњење података при упису је нула тактова а при читању је један такт. Приступи су на нивоу речи са избором бајтова који ће бити уписани. Магистрала је конфигурабилна са произвољном ширином речи, адресе (наводи се ширина адресе са бајтовима као адресабилном јединицом јер је тако природније резоновати) и бајта. Активна вредност свих линија је 1.

```
interface arilla_bus_if #(
    parameter int DataWidth,
    parameter int ByteAddressWidth,
    parameter int ByteSize
);
    localparam int BytesPerWord     = DataWidth / ByteSize;
    localparam int WordAddressWidth = ByteAddressWidth - $clog2(BytesPerWord);

    wire [       DataWidth-1:0] data_ctp;
    wire [       DataWidth-1:0] data_ptc;
    wire [WordAddressWidth-1:0] address;
    wire [    BytesPerWord-1:0] byte_enable;
    wire                        hit;
    wire                        read;
    wire                        write;
    wire                        inhibit;
    wire                        intercept;

endinterface
```

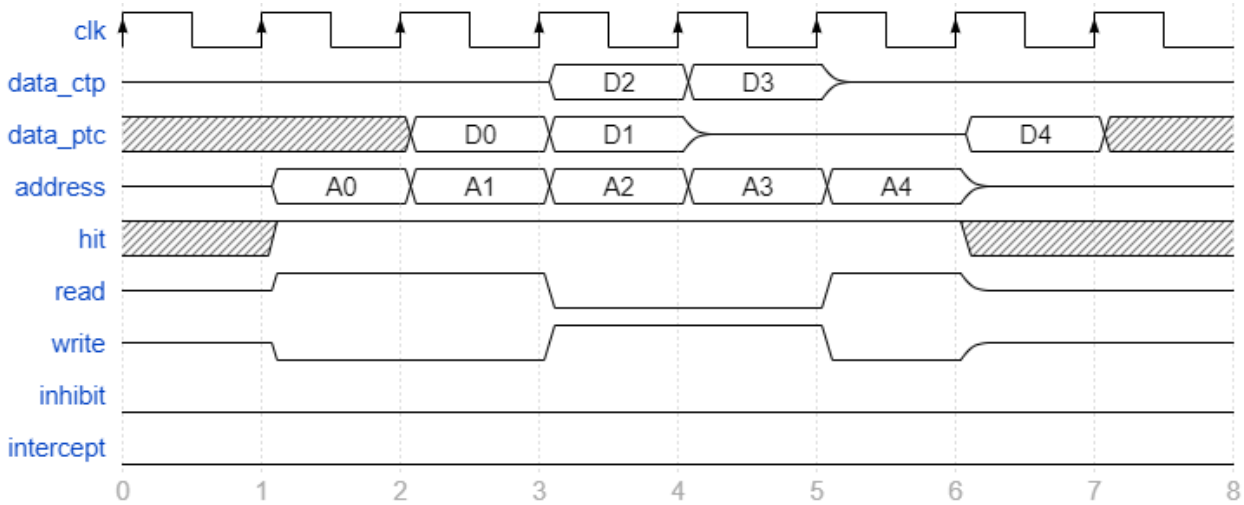
Исечак кода 3.2: Линије које сачињавају *Arilla bus* магистралу

Магистрала се састоји од линија приказаних у исечку 3.2, које су детаљније описане испод. Постоје две линије за податке: **data_ctp** (у смеру од језгра ка периферијама) и **data_ptc** (у смеру од периферија ка језгру). Предност постојања две линије за податке је могућност извршавања операције уписа одмах након операције читања, што не би било могуће са једном линијом за податке због различитог кашњења података (видети такт 3 на слици 3.3). Како постоји само једна линија за адресу није могуће извршити и операцију читања и операцију писања у истом такту, ово је прихватљива одлука јер би у том случају меморија морала да буде вишепортна, што није толико често и компликује дизајн меморије. Линија **byte_enable** представља маску бајтова који ће заправо бити уписани при операцији уписа (бајтови за које је одговарајући бит маске нула ће остати непромењени).

Магистрала има пет контролних линија. Линије **read** и **write** својом активном вредношћу означавају која операција ће бити извршена на магистрали. Понашање уколико обе линије имају активну вредност је **недефинисано**. Линија **hit** обавештава језгро да ли постоји меморија или периферија на тој адреси. У питању је комбинациони сигнал, концептуално имплементиран као ожичено или, међутим како *Quartus* не подржава ожичено или које прелази границе модула, **hit** је у овом случају имплементиран експлицитним или колом.

Линија **inhibit** омогућава просту арбитрацију. Подизањем линије **inhibit** на активну вредност приоритетнији газда комбинационо забрањује мање приоритетном газди да извршава операције на магистрали. Приоритетнији газда онда у истом такту може извршити операцију на магистрали. Иако мање приоритетни газда не може да извршава операције на магистрали, то га не спречава да заврши већ започето читање података.

Линија **intercept** омогућава имплементацију уређаја који ослушкују магистралу. Када линија **intercept** има активну вредност, слуга коме тражена адреса припада неће поставити свој податак на одговарајућу линију, омогућавајући уређају који је поставио активну вредност **intercept** линије да постави податак на линију за податке. Одзив уређаја на промену ове линије је комбинациони. Постојање више уређаја који ослушкују се може реализовати уколико се линија **intercept** реализује као ожичено или, међутим уколико више слуга покушају да пресретну исти податак понашање је **недефинисано**. Ова функционалност може да омогући додавање тачака прекида у делове меморије који су намењени само за читање (енг. Read Only) пресретањем дохватања инструкција са тражене адресе, међутим тренутно није имплементиран ниједан уређај који користи ову линију, те је она ожичена на нулу. Разлог за то је непостојање подршке за овако нешто у спецификацији [5].



Слика 3.3: Дијаграм неколико операција *Arilla bus* магистрале

Слика 3.3 приказује неколико узастопних циклуса на магистрали и на њој се може видети извршавање једне операције у једном такту, различита кашњења података за читање и упис и учешљавање операција на магистрали. Податак **D*i*** представља податак у вези са адресом истог индекса.

3.3 Меморија

Имплементиран систем поседује 64КБ меморије за програм и податке. Меморија је имплементирана користећи интегрисане меморијске блокове *FPGA* чипа. Како меморијски блокови могу бити конфигурисани да имају карактеристике које захтева *Arilla bus* магистрала, потребно је имплементирати само проверу адресе и руковање сигналима **data_ptc** и **hit**.

3.4 Периферије

Развојна плоча *DE10-Standard* [20] која је коришћена за имплементацију, поред самог *FPGA* чипа поседује и велики број периферија које су повезане на њега. Како би било могуће визуелно пратити стање извршавања програма ради поређења са стањем које приказује дебагер, одлучено је да се имплементира неколико простих периферија, приказаних у овој секцији.

3.4.1 *GPIO* периферија

General Purpose Input Output (GPIO) периферија је повезана на 10 прекидача и лед диода, као и 2 од 4 тастера. Ова периферија омогућава читање стања било којег од повезаних пинова и контролисање стања лед диода. Периферијом се управља кроз 3 регистра, приказана у исечку 3.3. *Data Direction Register (DDR)* одређује да ли се неки пин користи као улазни или излазни, уколико је пин излазни, његов ниво је дефинисан одговарајућим битом *Data Output Register (DOR)* регистра, уколико је пин улазни, он се налази у стању високе импедансе како би екстерни сигнал могао да дефинише његов ниво. *Data Input Register (DIR)* регистар садржи информацију о напонском нивоу на сваком од пинова, то значи да је у теорији могуће детектовати уколико неки други уређај покушава да дефинише супротну вредност за неки од излазних пинова.

```
#ifndef _GPIO_H_
#define _GPIO_H_

#include <stdint.h>

typedef struct
{
        uint32_t volatile DDR;
        uint32_t volatile DOR;
        uint32_t volatile DIR;
} GPIO_RegisterMapType;

#define GPIO ((GPIO_RegisterMapType *) 0x10000000)

#define GPIO_BTN_MASK(BTN) (1 << (BTN - 2))
#define GPIO_SW_MASK(SW) (1 << (SW + 2))
#define GPIO_LED_MASK(LED) (1 << (LED + 12))

#endif
```

Исечак кода 3.3: Дефиниција регистара *GPIO* периферије

Периферија је конфигурабилна са подесивом базном адресом, бројем пинова (уколико је број пинова већи од ширине речи, потребан број речи за сваки од регистара ће бити аутоматски алоциран) и маском која спецификује пинове које се могу користити искључиво као улазни (њихови бити *DDR* регистра су ожичени на нулу[3]).

3.4.2 *EXTI* периферија

External Interrupt (EXTI) периферија је повезана на 10 прекидача и 2 од 4 тастера. Ова периферија омогућава генерисање захтева за прекид на узлазну и/или силазну ивицу било ког од повезаних пинова. Периферија поседује 5 регистара, приказаних у исечку 3.4. *Interrupt Mask Register (IMR)* контролише који пинови могу да генеришу захтеве за прекиде, да би захтев за прекид био генерисан, неопходно је да одговарајући бити буду постављени и у *IMR* и у *IPR* регистру. *Interrupt Pending Register (IPR)* садржи информацију о томе са којих пинова су пристигли захтеви за прекиде, бит овог регистра ће аутоматски бити постављен уколико се на одговарајућој линији детектује ивица и *RER* и *FER* регистри су подешени да ту ивицу детектују. Одговарајући бит *IPR* регистра се брише уписивањем јединице у њега. *Interrupt Set Register (ISR)* омогућава софтверу да ручно генерише прекид на некој линији уписивањем јединице у одговарајући бит. Регистри *Rising Edge Register (RER)* и *Falling Edge Register (FER)* контролишу на које ивице (узлазну, силазну или обе) ће бити генерисан захтев за прекид.

[3]Исто важи и за неискоришћене бите у сва три регистра

```
typedef struct
{
        uint32_t volatile IMR;
        uint32_t volatile IPR;
        uint32_t volatile ISR;
        uint32_t volatile RER;
        uint32_t volatile FER;
} EXTI_RegisterMapType;

#define EXTI ((EXTI_RegisterMapType *) 0x30000000)

#define EXTI_LINE(NUM) (1 << NUM)
```

Исечак кода 3.4: Дефиниција регистара *EXTI* периферије

Периферија је конфигурабилна са подесивом базном адресом и бројем пинова.

3.4.3 *HEX* периферија

HEX периферија контролише 6 седмосегментних дисплеја који постоје на развојној плочи. Периферија поседује три мода операције који се бирају уписом одговарајуће вредности у *MODE* регистар. У *DEC* или *HEX* моду дисплеји приказују вредност одговарајућег броја најнижих бита *DATA* регистра у одговарајућем бројевном систему, док се у *MAN* моду за сваки дисплеј алоцира по 8 бита где 7 најнижих бита директно контролише индивидуалне сегменте дисплеја. Регистарска мапа периферије је приказана у исечку 3.5.

```
typedef struct
{
        uint32_t volatile MODE;
        uint32_t volatile DATA;
        uint32_t volatile DATA1;
} HEX_RegisterMapType;

#define HEX ((HEX_RegisterMapType *) 0x20000000)

#define HEX_MODE_DEC 0
#define HEX_MODE_HEX 1
#define HEX_MODE_MAN 2
```

Исечак кода 3.5: Дефиниција регистара *HEX* периферије

Периферија је конфигурабилна са подесивом базном адресом и бројем дисплеја, као што се може видети, уколико је потребно периферија алоцира више меморијских речи за *DATA* регистар.

3.4.4 Интерфејс за меморијски мапиране регистре

Све периферије које поседују меморијски мапиране регистре користе **periph_mem_interface** компоненту која пружа једноставан интерфејс за имплементацију меморијски мапираних регистара. Интерфејс је приказан у исечку 3.6. Компонента прима базну адресу и величину меморијски мапираног региона у речима (величина мора бити степен броја 2, одговорност је на периферији која користи ову компоненту да своје регистре прошири ожиченим нулама до потребног броја речи), све остале параметре меморијске магистрале компонента детектује аутоматски. Интерфејс према периферији која користи ову компоненту се састоји од три линије **data_periph_in**, **data_periph_out**, **data_periph_write**. Линија **data_periph_in** је величине *Ширина речи * Број речи* и представља податке које периферија жели да учини доступним. Линија **data_periph_out** (ширине меморијске речи) представља податке које језгро жели да упише на неку адресу меморијски мапираног региона, излаз ове линије имплементира маску бајтова за упис те није потребно да периферија имплементира ту функционалност. Линија **data_periph_write** поседује по један бит за сваку меморијску реч меморијски мапираног региона, одговарајући бит ове линије има активну вредност када језгро врши упис у ту меморијску реч.

```
module periph_mem_interface #(
    parameter int BaseAddress,
    parameter int SizeWords
) (
    clk,
    rst_n,
    bus_interface,
    hit,
    data_periph_in,
    data_periph_out,
    data_periph_write
);
    localparam int DataWidth             = $bits(bus_interface.data_ctp);
    localparam int AddressWidth          = $bits(bus_interface.address);
    localparam int BytesPerWord          = $bits(bus_interface.byte_enable);
    localparam int ByteSize              = DataWidth / BytesPerWord;
    localparam int SizeBytes             = SizeWords * BytesPerWord;
    localparam int ByteAddressWidth      = AddressWidth + $clog2(BytesPerWord);
    localparam int LocalAddressWidth     = $clog2(SizeWords);
    localparam int LocalByteAddressWidth = LocalAddressWidth + $clog2(BytesPerWord);
    localparam int DeviceAddressWidth    = AddressWidth - LocalAddressWidth;
    localparam int DeviceAddress         = BaseAddress[ByteAddressWidth-1:
        LocalByteAddressWidth];

    input clk;
    input rst_n;

    arilla_bus_if bus_interface;

    output hit;

    input [(SizeWords*DataWidth)-1:0] data_periph_in;

    output [DataWidth-1:0] data_periph_out;
    output [SizeWords-1:0] data_periph_write;
```

Исечак кода 3.6: Портови **periph_mem_interface** компоненте

4 *RISC-V* језгро

У овом поглављу су дати детаљи имплементације *RISC-V* језгра. Језгро имплементира *RV32I* основни инструкцијски сет, *Zicsr* екстензију која омогућава приступ контролним и статусним регистрима и машински мод извршавања са обрадом прекида. Организација језгра је вишетактна. Поред вишетактне, разматране су и једнотактне и организације са проточном обрадом. Једнотактна организација је одмах одбачена јер у случају инструкција за приступ меморији захтева два приступа меморији у једном такту. Вишетактна организација је изабрана уместо организације са проточном обрадом због једноставности имплементације с обзиром да је фокус на подршци за екстерно дебаговање а не на самом језгру. Иако фокус није на самом језгру, одлучено је да се покуша оптимизација перформанси језгра, колико то изабрана организација дозвољава.

Резултат је језгро које извршава инструкције приступа меморији у два такта а све остале инструкције у једном такту.

4.1 Путања података

Путања података обухвата све компоненте које врше операције над подацима које процесор обрађује, док су саме операције дефинисане углавном од стране управљачке јединице. Ова секција се фокусира на компонентама које сачињавају путању података, начину њиховог повезивања и току података кроз њих при извршавању инструкција.

Како меморија има један такт кашњења при читању, на крају претходне инструкције када се врши упис у *PC*[1] регистар, обавља се и читање из меморије на адреси нове вредности *PC* регистра, што значи да је инструкција дохваћена и спремна за извршавање на почетку следећег такта који и званично означава почетак извршавања те инструкције. Ова идеја која личи на примитивну проточну обраду је кључна у постизању извршавања већине инструкција у једном такту.

Друга техника коришћена да би се постигле жељене перформансе произилази из потребе да једина места где подаци не пролазе у истом такту буду меморија и регистарски фајл. Ово се примарно односи на *PC* и *IR*[2] регистре, који морају да памте вредност за вишетактне инструкције и рачунање нове вредности *PC* регистра, али не желимо да ти регистри додају додатно кашњење од једног такта када то није потребно. Ово је реализовано додавањем излаза **shadow_out** на те регистре. Када се врши упис у регистар, излаз **shadow_out** заобилази регистар и има вредност нове вредности која се уписује, у осталим случајевима **shadow_out** има вредност регистра.

Поред *PC* и *IR* регистара, путања података се састоји од меморијског интерфејса (**MEM_INTERFACE**), компоненте за декодовање инструкција (**INST_DECODE**), аритметичко

[1] Program Counter (PC).

[2] Instruction Register (IR).

логичке јединице (**ALU**), компоненте која рачуна следећу вредност *PC* регистра (**PC_CALC**) и регистарског фајла (**REG_FILE**).

4.1.1 MEM_INTERFACE

Компонента **MEM_INTERFACE** повезује језгро на *Arilla bus* магистралу. Компонента нема параметре већ се аутоматски адаптира магистрали на коју је повезана и подржава произвољну ширину података, адресе и бајта. Како је адресибилна јединица *Arilla bus* магистрале реч а језгро подржава и операције на нивоу полу-речи и бајта (уколико су поравнате на своју величину), главна улога ове компоненте је превођење ових операција. **MEM_INTERFACE** такође детектује невалидне ситуације (лоше поравнат приступ меморији, приступ непостојећој меморији и приступ уколико је линија **inhibit** активна) и обавештава језгро ако до њих дође.

4.1.2 INST_DECODE

RISC-V инструкција обавезно има поље са операционим кодом и може садржати неку комбинацију следећих поља: додатни спецификатор инструкције ширине 3 бита (*f3*), додатни спецификатор инструкције ширине 7 бита (*f7*), непосредну вредност, адресу првог изворног регистра, адресу другог изворног регистра и адресу одредишног регистра. Компонента **INST_DECODE** издваја ова поља, проверава да ли је инструкција валидна и одређује операцију **op** и модификатор операције **mod** за аритметичко логичку јединицу. У општем случају операција има вредност *f3* а модификатор *f7[5]*, за аритметичко логичке операције са непосредним адресирањем, модификатор је 0 и за операције приступа меморији или безусловног скока операција је фиксно сабирање а модификатор је 0.

4.1.3 ALU

Аритметичко логичка јединица у зависности од сигнала **op** и **mod** врши једну од 10 операција. Те операције су: сабирање, одузимање, мање једнако, неозначено мање једнако[3], битско ексклузивно или, битско или, битско и, логичко померање у лево, логичко померање у десно и аритметичко померање у десно. Ширина аритметичко логичке јединице је параметризована.

4.1.4 REG_FILE

Регистарски фајл има подесив број регистара подесиве ширине[4] и омогућава читање два и упис једног регистра. Регистар нула има вредност нула и упис у њега нема ефекта.

4.1.5 PC_CALC

PC_CALC израчунава адресу следеће инструкције и проверава њено поравнање (инструкције у *RV32I* су увек поравнате на 4 бајта). У зависности од операционог кода инструкције **PC_CALC**: израчунава одредиште скока (за инструкције условног или безусловног скока), проверава логички израз (за инструкције условног скока) или инкрементира *PC* за 4. За условне скокове *RISC-V* може да пореди произвољна два регистра неким од следећих оператора $==, !=, <, \geq$[5].

[3] Уколико је **a** < **b** резултат је 1, у супротном 0.

[4] Број и ширина регистара су подесиви кроз параметре компоненте.

[5] $<, \geq$ имају варијанте за означене и неозначене бројеве.

4.1.6 Ток података при извршавању инструкције

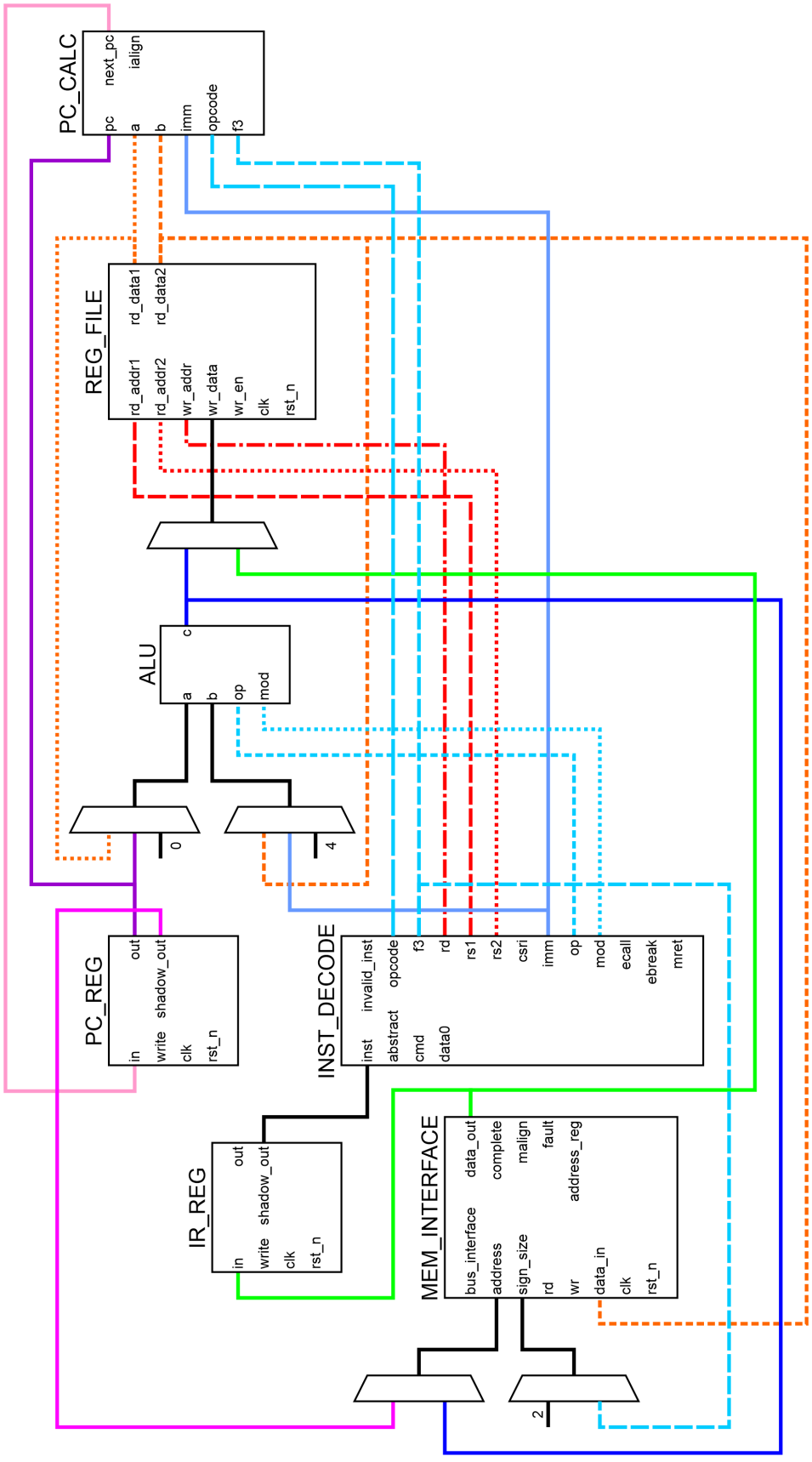
У првом такту извршавања инструкције:

1. Инструкција се налази на линијама за податке меморијске магистрале и **PC_REG** садржи адресу тренутне инструкције.
2. Инструкција улази у језгро кроз **MEM_INTERFACE** и уписује се у **IR_REG** (**INST_DECODE** користи **shadow_out** те је нова вредност одмах доступна за декодовање).
3. Инструкција се декодује у **INST_DECODE**.
4. **INST_DECODE** израчунава непосредни операнд **imm**, конфигурише **ALU** (сигнали **op** и **mod**) и **PC_CALC** (сигнали **opcode** и **f3**) и адресира изворне и одредишни регистар.
5. а) (Инструкција не приступа меморији) **ALU** израчунава резултат инструкције.
   б) (Инструкција приступа меморији) **ALU** израчунава меморијску адресу.
6. а) (Инструкција не приступа меморији а модификује регистарски фајл) Резултат инструкције се уписује у **REG_FILE**.
   б) (Инструкција приступа меморији) Приступа се меморији са израчунатом адресом, сигнал **f3** одређује величину приступа (уколико је у питању упис у меморију, податак за упис се налази у регистарском фајлу на адреси другог изворишног регистра).
7. **PC_CALC** израчунава адресу следеће инструкције користећи изворишне регистре, адресу тренутне инструкције и податке из саме инструкције.
8. (Инструкција не приступа меморији) У **PC_REG** се учитава адреса следеће инструкције (**shadow_out** има вредност адресе следеће инструкције).
9. (Инструкција не приступа меморији) Чита се из меморије са адресе следеће инструкције (**shadow_out PC_REG**-а), величина приступа је фиксна и једнака величини речи.

У другом такту извршавања инструкције (само за инструкције које приступају меморији):

1. **PC_REG** задржава вредност а **IR_REG** садржи тренутну инструкцију (како се у овом такту не уписује у **IR_REG**, **shadow_out** има вредност запамћену у регистру, што је иста вредност коју је имао у прошлом такту).
2. (Инструкција чита из меморије) У регистарски фајл се уписује вредност прочитана из меморије (**data_out MEM_INTERFACE**-а).
3. У **PC_REG** се учитава адреса следеће инструкције (**shadow_out** има вредност адресе следеће инструкције), како се вредност **PC_REG**-а није променила од прошлог такта, адреса следеће инструкције коју је **PC_CALC** израчунао у прошлом такту је и даље валидна.
4. Чита се из меморије са адресе следеће инструкције (**shadow_out PC_REG**-а), величина приступа је фиксна и једнака величини речи.

На слици 4.1 се може видети дијаграм описане путање података.



Слика 4.1: Дијаграм путање података

4.2 Управљачка јединица

Управљачка јединица одређује које операције извршава путања података, како би извршила тренутну инструкцију. Детаљи њеног функционисања су приказани у наставку. Неопходно је да управљачка јединица буде реализована Милијевим аутоматом стања јер управљачки сигнали зависе од операционог кода инструкције који се сазнаје у истом такту када и управљачки сигнали треба да буду подешени. Управљачка јединица је реализована директно у језику за опис хардвера, ово програмеру даје комфор сличан писању микропрограмске управљачке јединице док је резултујући хардвер након процеса синтезе сличнији ожиченој реализацији.

```
‘define CONTROL_SIGNALS__ADDR_ALU 1’b0
‘define CONTROL_SIGNALS__ADDR_PC  1’b1
‘define CONTROL_SIGNALS__RD_ALU   2’b00
‘define CONTROL_SIGNALS__RD_MEM   2’b01
‘define CONTROL_SIGNALS__ALU1_RS  2’b00
‘define CONTROL_SIGNALS__ALU1_PC  2’b01
‘define CONTROL_SIGNALS__ALU1_ZR  2’b10
‘define CONTROL_SIGNALS__ALU2_RS  2’b00
‘define CONTROL_SIGNALS__ALU2_IM  2’b01
‘define CONTROL_SIGNALS__ALU2_IS  2’b10

‘define CONTROL_SIGNALS__PROLOGUE   5’b10_000
‘define CONTROL_SIGNALS__DISPATCH   5’b10_001
‘define CONTROL_SIGNALS__LUI        ‘ISA__OPCODE_LUI
‘define CONTROL_SIGNALS__AUIPC      ‘ISA__OPCODE_AUIPC
‘define CONTROL_SIGNALS__JAL        ‘ISA__OPCODE_JAL
‘define CONTROL_SIGNALS__JALR       ‘ISA__OPCODE_JALR
‘define CONTROL_SIGNALS__BRANCH     ‘ISA__OPCODE_BRANCH
‘define CONTROL_SIGNALS__LOAD       ‘ISA__OPCODE_LOAD
‘define CONTROL_SIGNALS__LOAD_W     (‘ISA__OPCODE_LOAD  + 5’d1)
‘define CONTROL_SIGNALS__LOAD_1     (‘ISA__OPCODE_LOAD  + 5’d2)
‘define CONTROL_SIGNALS__STORE      ‘ISA__OPCODE_STORE
‘define CONTROL_SIGNALS__STORE_W    (‘ISA__OPCODE_STORE + 5’d1)
‘define CONTROL_SIGNALS__STORE_1    (‘ISA__OPCODE_STORE + 5’d2)
‘define CONTROL_SIGNALS__OPIMM      ‘ISA__OPCODE_OPIMM
‘define CONTROL_SIGNALS__OP         ‘ISA__OPCODE_OP
‘define CONTROL_SIGNALS__MISCMEM    ‘ISA__OPCODE_MISCMEM
‘define CONTROL_SIGNALS__SYSTEM     ‘ISA__OPCODE_SYSTEM

interface control_signals_if;
    wire                          mem_complete;
    wire [‘ISA__OPCODE_WIDTH-1:0] opcode;
    reg                           write_pc, write_ir, write_rd;
    reg                           mem_read, mem_write;
    reg                           addr_sel;
    reg  [                 1:0]   rd_sel;
    reg  [                 1:0]   alu_insel1, alu_insel2;
endinterface
```

Исечак кода 4.1: Контролни сигнали управљачке јединице

Исечак 4.1 на почетку приказује дефиниције конкретних вредности управљачких сигнала за мултиплексере (два мултиплексера која одређују меморијску адресу и величину приступа имају заједнички управљачки сигнал). Након тога долазе дефиниције стања аутомата, за стање се користи 5 бита што је једнако ширини операционог кода инструкције и омогућава пресликавање вредности операционог кода у одговарајуће стање аутомата. Поред стања за сваки имплементирани операциони код, аутомат поседује још 6 стања: по два за инструкције читања и писања меморије, стање које припрема језгро за извршење прве инструкције и псеудостање које означава први такт извршавања инструкције.

Стања **LOAD_1** и **STORE_1** означавају други такт извршавања инструкције читања и пи-

сања меморије респективно. Стања **LOAD_W** и **STORE_W** су идентична стањима **LOAD** и **STORE** осим што не врше понован упис у **IR_REG**. Аутомат улази у ова стања уколико му је линијом **inhibit** забрањено извршавање операција на магистрали, и у њима остаје док успешно не изврши тражену операцију, након чега прелази у **LOAD_1** и **STORE_1** респективно. Стање **PROLOGUE** доводи језгро у стање у којем може следећег такта да почне са извршавањем инструкције. Ово стање врши дохватање инструкције из меморије са адресе која се тренутно налази у **PC_REG**-у, и тиме гарантује испуњеност предуслова за извршавање инструкције (корак 1 извршавања првог такта инструкције) уколико приступ меморији успе. Стање **PROLOGUE** се такође користи слично стањима **LOAD_W** и **STORE_W** уколико дохватање инструкције не успе. **DISPATCH** представља псеудостање је се систем никада неће наћи у том стању, већ вредност у регистру стања означава да се контролни сигнали понашају као да је систем у стању које је једнако операционом коду инструкције.

На крају исечка 4.1 се налази дефиниција интерфејса који садржи улазне и излазне контролне сигнале. Улазни сигнали су операциони код инструкције, као и информација да ли је приступ меморији успео (ово се односи само на линију **inhibit**, остале грешке се детектују на други начин). Излазни сигнали управљају уписом у **PC_REG**, **IR_REG** и **REG_FILE**, читањем и уписом у меморију, као и избором сигнала на четири мултиплексера који су део путање података. У прилогу А се налази упрошћени код управљачке јединице.

4.3 *Zicsr* екстензија

Екстензија *Zicsr* прописује адресни простор од 4096 контролних и статусних регистара којима се приступа посебним инструкцијама које атомично читају и модификују те регистре регистарским директним или непосредним адресирањем. Ова функционалност је имплементирана у компоненти **CSR** која има следеће портове од интереса:

- **csr_interface**, интерфејс који остатак језгра користи да чита и уписује у контролне и статусне регистре.
- **rs**, adresa регистра опште намене чија вредност се уписује.
- **reg_in**, вредност регистра опште намене чија вредност се уписује.
- **imm_in**, вредност непосредног операнда који се уписује.
- **addr**, адреса контролног и статусног регистра који се уписује.
- **f3**, операција која се врши над регистром (уписивање, постављање бита или уклањање бита) и адресирање (регистарско директно или непосредно).
- **write**, контролни сигнал који врши упис у контролни и статусни регистар.
- **csr_out**, излазни сигнал који представља вредност изабраног контролног и статусног регистра.
- **invalid**, излазни сигнал који означава да је дошло до грешке (непостојећи регистар или упис у заштићен регистар).
- **conflict**, излазни сигнал који означава да је у току упис у регистар који би такође био промењен при прихватању захтева за прекид.

Како постоји велики број ових регистара у великој мери су коришћени макрои да генеришу исти хардвер за сличне регистре, ови макрои се налазе у фајлу **csr.svh** чија је скраћена верзија приказана у исечку 4.2.

```
// Primer makroa koji definise CSR registar
`define CSR__MVENDORID 12'hF11 // Adresa
`define CSR__MCYCLE_MASK 32'hFFFFFFFF // Maska bita koji mogu biti upisani
`define CSR__MCYCLE_VALUE 32'h00000000 // Podrazumevana vrednost
`define CSR__MSTATUS_MPP(csr) csr[12:11] // Makroi za pristup poljima

// Makroi koji definisu srodne grupe registara
`define CSRGEN__FOREACH_ARRAY(TARGET, CSR) \
`TARGET(CSR, 3) \
...
`TARGET(CSR, 31)

`define CSRGEN__FOREACH_MCOUNTER(TARGET) \
`TARGET(MCYCLE) \
...

`define CSRGEN__FOREACH_MHPMCOUNTER(TARGET) \
`CSRGEN__FOREACH_ARRAY(TARGET, MHPMCOUNTER) \
...

`define CSRGEN__FOREACH_MRO(TARGET) \
`TARGET(MVENDORID) \
...

`define CSRGEN__FOREACH_MRW(TARGET) \
`TARGET(MSTATUS) \
...

// Makroi koji definisu hardver
`define CSRGEN__GENERATE_INTERFACE(csr) \
reg  [`ISA__XLEN-1:0] ``csr``_reg;\
wire [`ISA__XLEN-1:0] ``csr``_in;\
wire                  ``csr``_write;

`define CSRGEN__GENERATE_READ_ASSIGN(csr) \
assign csr_out = address == `CSR__``csr`` ? csr_interface.``csr``_reg : 32'bz;\
assign hit     = address == `CSR__``csr``;

`define CSRGEN__GENERATE_READ_ASSIGN_MRO(csr) \
assign csr_out = address == `CSR__``csr`` ? `CSR__``csr``_VALUE : 32'bz;\
assign hit     = address == `CSR__``csr``;

`define CSRGEN__GENERATE_ARRAY_READ_ASSIGN_MRO(csr, i) \
assign csr_out = address == `CSR__``csr``(i) ? `CSR__``csr``_VALUE : 32'bz;\
assign hit     = address == `CSR__``csr``(i);

`define CSRGEN__GENERATE_INITIAL_VALUE(csr) \
csr_interface.``csr``_reg <= `CSR__``csr``_VALUE;

`define CSRGEN__GENERATE_WRITE(csr) \
if (address == `CSR__``csr`` && write_reg && `CSR__``csr``_MASK != `ISA__ZERO) begin\
    csr_interface.``csr``_reg <= (value & `CSR__``csr``_MASK) | (csr_interface.
        ``csr``_reg & ~`CSR__``csr``_MASK);\
end else if (csr_interface.``csr``_write) begin\
    csr_interface.``csr``_reg <= csr_interface.``csr``_in;\
end

`define CSRGEN__GENERATE_CONFLICT(csr) \
assign conflict = (address == `CSR__``csr`` && write_reg && `CSR__``csr``_MASK !=
    `ISA__ZERO);
```

Исечак кода 4.2: Пример макроа коришћених у CSR компоненти

Сама компонента **CSR** проверава да ли постоји регистар на датој адреси користећи један сигнал типа ожичено или, потом проверава да ли је покушан упис у регистар намењен само за читање (највиша два бита адресе су 11) и уколико је потребно пријављује грешку. Остатак компоненте имплементира саме регистре коришћењем макроа приказаних у исечку 4.2 (на начин приказан у исечку 4.3), као и бројаче циклуса и извршених инструкција и системски тајмер. Системски тајмер се састоји од два меморијски мапирана регистра ширине 64 бита, они су имплементирани коришћењем **periph_mem_interface** компоненте која је раније описана. Бројачи циклуса и инструкција су имплементирани користећи исти интерфејс који остатак језгра користи да интерагује са контролним и статусним регистрима. Тај интерфејс се састоји од три линије за сваки регистар које се могу видети у макроу **CSRGEN__GENERATE_INTERFACE**. Линија са суфиксом **_reg** представља вредност регистра, линија са суфиксом **_in** представља вредност коју језгро жели да упише у тај регистар и линија са суфиксом **_write** је активна у такту у коме језгро жели да упише нову вредност у тај регистар.

```
assign hit = 1'b0;
`CSRGEN__FOREACH_MCOUNTER(CSRGEN__GENERATE_READ_ASSIGN)
`CSRGEN__FOREACH_MHPMCOUNTER(CSRGEN__GENERATE_ARRAY_READ_ASSIGN_MRO)
`CSRGEN__FOREACH_MRO(CSRGEN__GENERATE_READ_ASSIGN_MRO)
`CSRGEN__FOREACH_MRW(CSRGEN__GENERATE_READ_ASSIGN)

assign conflict = 1'b0;
`CSRGEN__GENERATE_CONFLICT(MSTATUS)
`CSRGEN__GENERATE_CONFLICT(MCAUSE)
`CSRGEN__GENERATE_CONFLICT(MTVAL)
`CSRGEN__GENERATE_CONFLICT(MEPC)

always @(posedge clk) begin
    if (!rst_n) begin
        `CSRGEN__FOREACH_MCOUNTER(CSRGEN__GENERATE_INITIAL_VALUE)
        `CSRGEN__FOREACH_MRW(CSRGEN__GENERATE_INITIAL_VALUE)
    end else begin
        `CSRGEN__FOREACH_MCOUNTER(CSRGEN__GENERATE_WRITE)
        `CSRGEN__FOREACH_MRW(CSRGEN__GENERATE_WRITE)
    end
end
```

Исечак кода 4.3: Пример имплементације контролних и статусних регистара коришћењем макроа

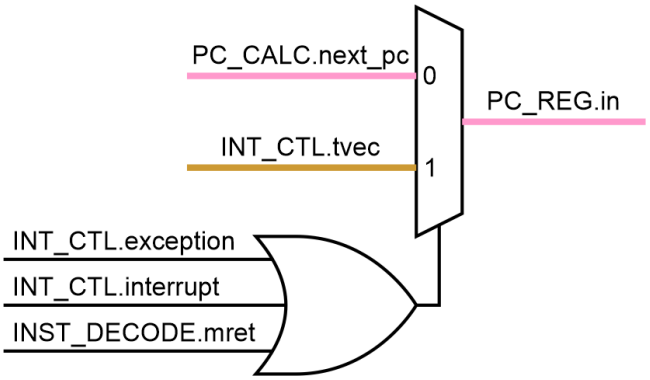
4.4 Обрада прекида

Имплементирано језгро подржава прекиде и изузетке[6] и врши њихову обраду у складу са привилегованом спецификацијом [8], метод ове обраде је описан у остатку секције.

Већина логике за обраду прекида се налази у компоненти **INT_CTL** док је поред тога потребна само минимална модификација путање података и управљачке јединице. Модификација путање података, која омогућава скок на адресу прекидне рутине при прихватању прекида или изузетка пропуштањем адресе прекидне рутине на улаз **PC_REG**-а уколико је неки од сигнала из **INT_CTL** активан, је приказана на слици 4.2. Модификација управљачке јединице се састоји од извршавања следећих акција уколико је дошло до изузетка: искључивање контролних сигнала који мењају стање (**write_rd**, **write_csr** и **mem_write**), активација сигнала за упис **PC_REG**-а

[6] *RISC-V* спецификација прекидима класификује екстерне асинхроне догађаје, док су изузеци синхрони догађаји који су најчешће последица грешке при извршавању инструкције.

када је дошло до изузетка (сигнал **write_pc_ex**), учитавање следеће инструкције макроом **CONTROL__READ_INST** и прелазак у стање **DISPATCH** или **PROLOGUE** у зависности да ли је дохватање инструкције успело. Уколико се прихвата прекид тренутна инструкција се правилно завршила те нема потребе за претходним операцијама.



Слика 4.2: Модификација путање података

INT_CTL компонента врши детекцију изузетака и прекида, одређује адресу прекидне рутине и ажурира командне и статусне регистре од интереса. Детекција изузетака је прилично једноставна, потребно је упарити линију која обавештава да је изузетак настао са тренутним стањем процесора како би се ближе одредило о ком изузетку је реч, нпр. приступ непостојећој меморији резултује једним од три изузетка дефинисаних у спецификацији [8]: *Instruction access fault*, *Load access fault* или *Store/AMO access fault*. Детекција прекида је такође једноставна, треба проверити линију која обавештава о захтеву за прекид и одговарајуће бите контролних и статусних регистара који маскирају појединачне или све прекиде. Сама одлука о скакању на прекидну рутину је нешто комплекснија и разматрају се још три услова. Да би се при обради прекида скочило на прекидну рутину потребно је да:

1. Постоји захтев за прекид који није маскиран или постоји захтев за немаскирајући прекид.
2. Уколико је у питању обичан прекид, прекиди нису глобално маскирани или уколико је у питању немаскирајући прекид, процесор тренутно не извршава прекидну рутину немаскирајућег прекида.
3. Процесор тренутно извршава последњи такт инструкције.
4. Тренутна инструкција не врши упис у неки од регистара чије вредности се ажурирају при скоку на прекидну рутину.

Провера да ли процесор тренутно обрађује прекидну рутину немаскирајућег прекида постоји зато што су све линије осетљиве на ниво што доводи до поновног скока на исту прекидну рутину након извршене једне инструкције прекидне рутине. Код обичних прекида ово је решено постављањем бита који глобално омогућава прекиде на 0 при скоку на прекидну рутину. Како сличан бит за немаскирајуће прекиде не постоји у спецификацији[7], у овој имплементацији је тај бит додат. Како је у питању само један бит чија се вредност враћа на 0 при извршавању прве инструкције повратка из прекидне рутине, потенцијално може настати проблем уколико се нека друга прекидна рутина угнезди унутар рутине немаскирајућег прекида, како је овај сценарио изузетно мало вероватан, није имплементирано боље решење, већ је неопходно да програмер буде свестан ове ситуације.

Прекиди се прихватају само на крају инструкције јер то значајно олакшава имплементацију а у општем случају одлаже прихватање прекида максимално један такт. Разматрана је опција

[7]Што је и логично, спецификација дефинише архитектуру тј. делове процесора доступне кориснику, што овај бит не би требао да буде. Такође спецификација намерно даје одређену слободу имплементацијама по питању обраде немаскирајућих прекида.

за прихватање прекида док је језгро у стању **PROLOGUE** као резултат активне линије **inhibit** међутим то би поприлично компликовало имплементацију а умањило би кашњење прихватања прекида за само један до два сигнала такта, иако језгро може провести произвољно велики број тактова у стању **PROLOGUE**. То је зато што ће језгро свакако провести исти број тактова у стању **PROLOGUE** и уколико одмах прихвати прекид јер и дохватање прве инструкције прекидне рутине захтева успешан приступ меморији. Тако да је једина разлика у кашњењу да ли ће се након уклањања активне вредности линије **inhibit** извршити једна додатна инструкција или не. Један изузетак овог правила је тај да се прекид може прихватити уколико се десио изузетак након првог такта инструкције која приступа меморији, то је зато што по спецификацији прекиди имају већи приоритет од изузетака а како инструкција која генерише изузетак не мења стање процесора безбедно је прихватити прекид уколико је адреса повратка из прекидне рутине инструкција која је изазвала прекид. Ово важи и за изузетке генерисане у последњем такту инструкције, међутим тада је правило о последњем такту инструкције испоштовано и једина разлика је адреса повратка из прекидне рутине.

Последњи случај у коме неће доћи до скока на прекидну рутину је уколико се тренутно извршава инструкција која врши упис у неки од контролних и статусних регистара који се ажурирају при скоку на прекидну рутину. Ово се примарно односи на регистар **MSTATUS** је проширено и на остале регистре (**MCAUSE**, **MTVAL** и **MEPC**). Овај сигнал генерише компонента **CSR** и постоји због једног ивичног случаја[8] у коме програм уписује 0 у бит који омогућава све прекиде (бит **MIE** регистра **MSTATUS**), како ће упис бити видљив језгру тек следећег такта, прекид се прихвата (ово само по себи није проблем, рећићемо да је прекид прихваћен пре комплетног извршења те инструкције) међутим у поље **MPIE** регистра **MSTATUS** се уписује тренутна вредност бита **MIE** а не 0 која се тренутно уписује. Што значи да ће, када се при повратку из прекидне рутине у бит **MIE** упише вредност бита **MPIE**, она бити као да се инструкција након које је прихваћен прекид није десила, што може резултовати у томе да су прекиди омогућени иако не би требали да буду. Иако за овај ивични случај сигурно постоје и алтернативна решења ово решење је поприлично једноставно и у најгорем случају одлаже прекид један такт.

Следећи корак при обради прекида и изузетака је одређивање уколико постоји више разлога за скок који ће бити испоштован. Већ је споменуто шта се дешава уколико се истовремено десе изузетак и прекид, међутим још један разлог за скок који обрађује ова компонента је повратак из прекидне рутине инструкцијом **MRET**. Табела 4.1 приказује исход сваке комбинације. Изузеци и прекиди се међусобно приоритирају по фиксним приоритетима дефинисаним у спецификацији [8].

Табела 4.1: Исход комбинација прекида, изузетака и повратка из прекидне рутине

Изузетак	Прекид	Повратак из прекидне рутине	Акција	**MEPC** =
*			Изузетак	**pc**
	*		Прекид	**next_pc**
		*	Повратак	/
*	*		Прекид	**pc**
*		*	Изузетак	**pc**
	*	*	Повратак	/
*	*	*	Прекид	**pc**

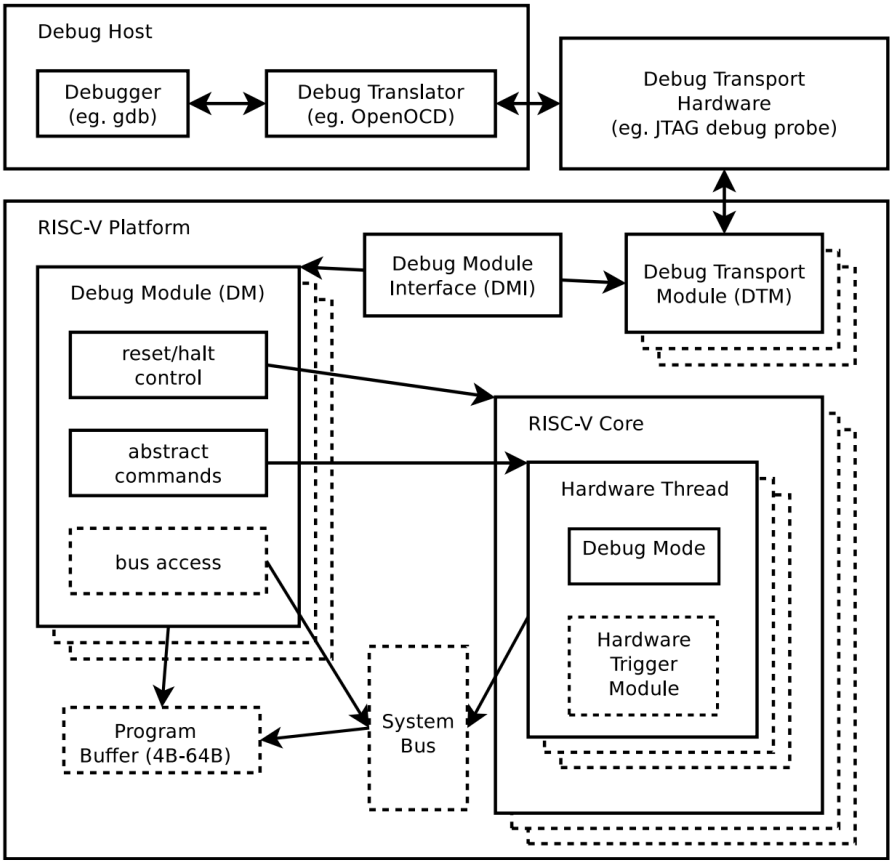
[8]Можда постоје и други ивични случајеви али један је довољан да објасни зашто је овај услов неопходан.

Одређивање адресе на коју се скаче је једноставно, уколико је у питању повратак из прекидне рутине, скаче се на адресу у **MEPC** регистру, уколико је у питању изузетак или прекид и прекиди нису векторисани скаче се на адресу у регистру **MTVEC** поравнату на 4 бајта, уколико су прекиди векторисани на адресу се додаје код узрока прекида померен за 2 бита улево.

INT_CTL такође ажурира следеће контролне и статусне регистре: Регистар **MIP** се ажурира сваког такта са вредностима линија захтева за прекид. При уласку у прекидну рутину бит **MPIE** се ажурира тренутном вредношћу бита **MIE** а **MIE** се поставља на 0. При повратку из прекидне рутине овај процес је обрнут, **MIE** добија вредност **MPIE** а **MPIE** се поставља на 1. Регистар **MEPC** се ажурира при уласку у прекидну рутину по табели 4.1. Регистри **MCAUSE** и **MTVAL** se ажурирају при уласку у прекидну рутину са кодом узрока прекида и описом изузетка (уколико је применљиво) респективно. Уколико се користи **MTVAL** најчешће има вредност меморијске адресе која је изазвала изузетак или вредности инструкције која је невалидна.

5 Подршка за екстерно дебаговање

RISC-V стандард који прописује подршку за екстерно дебаговање [5] предлаже архитектуру система приказану на слици 5.1.



Слика 5.1: Предложена архитектура подршке за екстерно дебаговање [5]

Компоненте ван самог система и *DTM* су објашњени у засебним поглављима, *System Bus* се у контексту овог рада односи на *Arilla bus* меморијску магистралу а *Program Buffer* је имплементиран као део *DM*-а. Преостале компоненте: *DM*, *DMI*, интерфејс којим комуницирају језгро и *DM*, потребне модификације језгра и хардверски окидачи су објашњене у остатку овог поглавља.

DMI је магистрала која повезује *DM* и *DTM*, и омогућава *DTM*-у приступ адресном простору *DM*-а. Спецификација подржава произвољну ширину адресе од минимално 7 бита, како један *DM* заузима мање од 128 адреса, одабрана је минимална ширина адресне линије. *DMI* се састоји од: адресне линије, линије за податке и две контролне линије које означавају операције читања и уписа. Адресирање је на нивоу речи. Како *DM* поседује релативно мали број регистара, могуће

је комбинационо поставити одговарајућу вредност на линију за податке при операцији читања. Што омогућава магистрали да изврши једну операцију сваког такта, са кашњењем података од нула тактова при читању и упису. Како спецификација налаже да читање непостојећег регистра резултује читањем вредности 0, линија за податке унутар *DM*-а (пре тростатичког бафера) је реализована као ожичено или.

DM инкорпорира већину функционалности које дебагер користи за управљање језгром и увид у његово стање. Функционалности покретања и заустављања језгра, као и извршавање апстрактних команди су имплементиране коришћењем интерфејса који повезује *DM* и језгро (дефинисаног у **debug_if.svh**). Програмски бафер и приступ меморијској магистрали су реализовани меморијски мапираним регистрима и додатним контролером газде на магистрали респективно.

Како би се подржало екстерно дебаговање неопходно је модификовати језгро. Модификације обухватају промене путање података и управљачке јединице, додавање контролних и статусних регистара и контролера језгра док је у режиму за дебаговање (компонента **D_CTL**). Додатно унутар језгра се налазе и хардверски окидачи који при приступу одређеној адреси заустављају језгро и пребацују га у режим за дебаговање.

Интерфејс који повезује *DM* и језгро се састоји од линија приказаних у исечку 5.1 (прва половина представља линије у смеру од *DM*-а ка језгру, а друга обратно).

```
interface debug_if;
    wire        halt_req;
    wire        resume_req;
    wire        exec;
    wire [31:0] command;
    wire [31:0] data0_in;
    wire [31:0] data1_in;

    wire        halted;
    wire        done;
    wire        write;
    wire        bus;
    wire        haltresume;
    wire        exception;
    wire [31:0] data0_out;
endinterface
```

Исечак кода 5.1: Линије `debug_if` интерфејса

Линијама **halt_req** и **resume_req** *DM* захтева од језгра да се заустави и настави са извршавањем респективно. Језгро повратну информацију о томе да ли је заустављено даје преко линије **halted**. Остатак линија се односи на извршавање апстрактних команди. У смеру од *DM*-а ка језгру линија **exec** је активна од тренутка када дебагер захтева извршавање апстрактне команде све док језгро не обавести *DM* да је команда извршена активном вредношћу линије **done**. Линије **command**, **data0_in** и **data1_in** садрже саму команду и њене потенцијалне аргументе, док линија **data0_out** садржи резултат уколико постоји. Резултат се уписује у одговарајући регистар *DM*-а када линија **write** има активну вредност. Линије **bus**, **haltresume** и **exception** својом активном вредношћу сигнализирају да је дошло до одговарајуће грешке.

5.1 *Debug Module (DM)*

DM имплементира следеће регистре доступне преко *DMI* магистрале: **DMSTATUS**, **DMCONTROL**, **HARTINFO**, **HALTSUM0**, **ABSTRACTCS**, **COMMAND**, **ABSTRACTAUTO**, **SBCS**, **SBADDRESS0**, **SBDATA0**, 12 **DATA** и 16 **PROGBUF** регистара. Кроз ове регистре дебагер има приступ свим функционланостима које *DM* пружа, а које су описане у наставку. **DATA** и **PROGBUF** регистри су међусобно веома слични те се за њихово генерисање користе макрои (приказани у исечку 5.2) налик онима коришћеним за контролне и статусне регистре. Регистри **DATA** и **PROGBUF** су такође доступни језгру у виду меморијски мапираних регистара, за ово је наравно коришћена **periph_mem_interface** компонента.

```
// Primer makroa koji definise DM registar
`define DEBUG__DATA0 7'h04 // Adresa
`define DEBUG__DATA0_AUTOEXEC 0 // Indeks bita u registru ABSTRACTAUTO
`define DEBUG__DATA0_OFFSET 12'd20 // Pomeraj od bazne adrese u memorijskoj mapi

`define DEBUGGEN__FOREACH_SIMPLE(TARGET) \
`TARGET(DATA0)\
`...
`TARGET(DATA11)\
`TARGET(PROGBUF0)\
...
`TARGET(PROGBUF15)\

`define DEBUGGEN__GENERATE_INTERFACE(register) \
reg  [31:0] ``register``_reg;\
wire [31:0] ``register``_in;\
wire        ``register``_write;\

`define DEBUGGEN__GENERATE_READ_ASSIGN(register) \
assign data = dmi.address == `DEBUG__``register`` ? ``register``_reg : {32{1'b0}};\

`define DEBUGGEN__GENERATE_MEMORY_ASSIGN(register) \
assign memory[(32*`DEBUG__``register``_OFFSET)+:32] = ``register``_reg;\

`define DEBUGGEN__GENERATE_MEMORY_GUARD_ASSIGN\
assign memory[(32*16)+:32] = `DEBUG__EBREAK;\
assign memory[(32*17)+:32] = `DEBUG__EBREAK;\
assign memory[(32*18)+:32] = `DEBUG__EBREAK;\
assign memory[(32*19)+:32] = `DEBUG__EBREAK;\

`define DEBUGGEN__GENERATE_INITIAL_VALUE_SIMPLE(register) \
``register``_reg <= `DEBUG__EBREAK;\

`define DEBUGGEN__GENERATE_WRITE_SIMPLE(register) \
if (dmi.address == `DEBUG__``register`` && dmi.write && busy == 1'b0) begin\
    ``register``_reg <= dmi.data;\
end else if (mem_write[`DEBUG__``register``_OFFSET]) begin\
    ``register``_reg <= mem_out;\
end else if (``register``_write) begin\
    ``register``_reg <= ``register``_in;\
end\

`define DEBUGGEN__GENERATE_AUTOEXEC(register) \
if (dmi.address == `DEBUG__``register`` && (dmi.write || dmi.read) && busy == 1'b0 &&
    cmderr == `DEBUG__AC_ERR_NO_ERR && abstractauto[`DEBUG__``register``_AUTOEXEC]) begin
    \
    busy <= 1'b1;\
end\

`define DEBUGGEN__GENERATE_BUSY_ERROR(register) \
|| dmi.address == `DEBUG__``register`` && (dmi.write || dmi.read)\
```

Исечак кода 5.2: Пример макроа коришћених у DM компоненти

5.1.1 Заустављање и поновно покретање језгра

Заустављање и поновно покретање језгра, као и увид у то да ли је тренутно језгро заустављено је могуће кроз **DMSTATUS** и **DMCONTROL** регистре. Такође је могуће аутоматски зауставити језгро када се ресетује и то пре извршавања прве инструкције. Имплементација ових функционалности са стране *DM*-а се у великој мери своди на повезивање одговарајућих бита **DMSTATUS** и **DMCONTROL** регистара на **debug_if** интерфејс.

5.1.2 Извршавање апстрактних команди

Апстрактне команде омогућавају дебагеру да чита и пише регистре опште намене, контролне и статусне регистре и меморију, као и извршавање произвољног кода који се налази у **PROGBUF** регистрима. Произвољан код је могуће извршити након или уместо приступа неком од регистара (коришћењем *Access Register Command*) или *Quick Access* командом која зауставља језгро, извршава произвољан код и поново покреће језгро. Апстрактна команда се извршава њеним уписом у **COMMAND** или уписом у **DATA** или **PROGBUF** регистар чији је бит у **ABSTRACTAUTO** регистру постављен (ово поново извршава претходну команду са новим вредностима регистара). Команде користе **DATA** регистре за своје аргументе, по правилу **DATA0**[1] садржи вредност за упис или прочитану вредност док **DATA1** садржи меморијску адресу[2]. **ABSTRACTCS** регистар примарно садржи информацију о томе да ли је се апстрактна команда тренутно извршава и о грешци уколико је до ње дошло. Само мали део функционалности апстрактних команди је обавезан а приказана имплементација имплементира све опционе функционалности[3] осим приступа регистрима и меморији без заустављања језгра.

Већина инфраструктуре за извршавање апстрактних команди се налази у језгру, док је примарна улога *DM*-а детекција одређених грешака. Грешке које *DM* детектује су: покушај извршавања апстрактне команде док је извршавање апстрактне команде у току, извршавање апстрактних команди са неподржаним опцијама и извршавање команде која захтева да језгро буде заустављено уколико оно није. Док су грешке које детектује језгро: грешка при приступу меморији[4], извршавању невалидне инструкције или заустављању језгра из погрешног разлога. Поред пријављивања грешке, уписи у регистре који се тичу апстрактних команди док се апстрактна команда извршава немају ефекта. Такође уколико дође до грешке, није могуће извршавати апстрактне команде док се статус грешке не отклони уписивањем у **ABSTRACTCS** регистар. Након провере грешака коју врши *DM*, он даје сигнал језгру да изврши апстрактну команду и прослеђује му саму команду и параметре. Језгро потом извршава команду и обавештава *DM* када је извршена или уколико је дошло до грешке. Уколико команда уписује резултат у **DATA0** регистар, вредност и сигнал да је потребно извршити упис се такође прослеђују *DM*-у. Уколико је команда успешно извршена и одговарајућа опција је подешена, *DM* инкрементира одговарајућу адресу (уколико је у питању меморијска адреса уместо за један, регистар се инкрементира за величину приступа).

[1] Ово важи за системе са ширином речи од 32 бита, системи са већом ширином речи користе неколико узастопних регистара за један аргумент.

[2] Адреса регистра је енкодована директно у команди.

[3] Које су примењиве за имплементирано језгро.

[4] Односи се само на приступ меморији помоћу апстрактних команди а не и на приступ помоћу газде на магистрали.

5.1.3 Приступ меморији коришћењем додатног газде на магистрали

Приступ меморији коришћењем додатног газде на магистрали је у потпуности имплементиран унутар *DM*-а, без потребе за било каквим модификацијама језгра[5] и користи исти контролер меморије (компоненту **MEM_INTERFACE**) само са различитим поларитетом линије **inhibit** који је подесив као параметар компоненте. Регистри **SBADDRESS0** и **SBDATA0** садрже адресу и вредност за приступ меморији а операције се започињу уписом или читањем неког од тих регистара у зависности од бита конфигурисаних у регистру **SBCS**. Поред тога **SBCS** садржи контролу величине приступа и инкрементирања адресе, као и бите који означавају да је дошло до грешке. Као и код апстрактних команди упис у **SBADDRESS0** или **SBDATA0** док је операција у току резултује у грешци и поред тога нема ефекат, слично операција се не може започети док се стање грешке не отклони уписом у одговарајуће бите **SBCS** регистра. Остале грешке до којих може доћи су: приступ непостојећој меморији, лоше поравнат приступ меморији или неподржана величина приступа. При упису у регистар који започиње операцију постављају се бити који означавају да је операција започета (**sbbusy**) и тип операције (**sb_read** или **sb_write**).

Свака операција траје два такта, у првом такту (уколико није детектована грешка у величини приступа) се поставља активна вредност линије **inhibit** и обавља се приступ меморији. У следећем такту се операција завршава уписом у **SBDATA0** уколико је била реч о операцији читања и инкрементирањем адресе за величину приступа уколико је одговарајући бит подешен.

5.2 Модификације језгра

Како би подржало екстерно дебаговање неопходно је да језгро имплементира посебан мод извршавања (*Debug* мод *D*) и још неколико контролних и статусних регистара који обухватају **DCSR**, **DPC** и два помоћна регистра које дебагер може да користи за чување регистара опште намене. Језгро се налази у овом моду док је заустављено или извршава апстрактну команду. **DCSR** регистар омогућава контролу над понашањем језгра док се налази у *D* моду. Имплементирана поља овог регистра су:

- **xdebugver** (само за читање), верзија спецификације за екстерно дебаговање коју језгро имплементира
- **ebreakm**, да ли **EBREAK** инструкција изазива изузетак или зауставља језгро
- **stepie**, да ли су прекиди омогућени у моду извршавања инструкцију по инструкцију
- **stopcount**, да ли су бројачи заустављени кад је језгро заустављено
- **stoptime**, да ли је тајмер заустављен кад је језгро заустављено
- **cause** (само за читање), разлог заустављања језгра
- **nmip** (само за читање), да ли постоји захтев за немаскирајући прекид
- **step**, да ли се језгро аутоматски зауставља након извршавања једне инструкције.

[5] Уколико је меморијски контролер у потпуности компатибилан са *Arilla bus* магистралом.

DPC регистар памти вредност програмског бројача приликом уласка у *D* мод и уписује ту вредност назад у програмски бројач при поновном покретању језгра (уписом у овај регистар је могуће променити локацију на којој ће језгро наставити извршавање). Сами регистри су имплементирани унутар **CSR** компоненте, као и логика која односи на заустављање бројача и тајмера. Остатак логике се претежно односи на прекиде и имплементиран је унутар **INT_CTL** компоненте.

5.2.1 Контролер режима за дебаговање

Контролер режима за дебаговање (компонента **D_CTL**) је повезан са *DM*-ом путем **debug_if** интерфејса и одговоран је за дистрибуцију тих информација остатку система. Како би обезбедио заустављање система, контролер води рачуна о жељеном стању система на основу различитих сигнала који захтевају заустављање језгра, као што су: хардверски окидач окинут, извршена **EBREAK** инструкција[6], захтев за заустављање од стране *DM*-а, заустављање након једне инструкције[7] и извршавање *Quick Access* апстрактне инструкције. Једини сигнал који може захтевати поновно покретање система је захтев од стране *DM*-а. Контролер о овом жељеном стању обавештава управљачку јединицу а потом прати реално стање језгра на основу стања у коме се налази управљачка јединица и о том стању обавештава *DM*, хардверске окидаче, контролер прекида и контролне и статусне регистре. При заустављању **D_CTL** такође ажурира **DCSR** кодом разлога заустављања и **DPC** адресом од које се наставља извршавање (у случају да је разлог заустављање након једне инструкције или захтев од стране *DM*-а бележи се адреса следеће инструкције[8], у супротном се бележи адреса тренутне инструкције). При извршавању апстрактних команди **D_CTL** реконфигурише путању података како би управљачка јединица могла да манипулише регистрима и меморијом и води рачуна о извршавању произвољних инструкција враћањем оригиналне конфигурације путање података, скоком на програмски бафер и започињањем извршавања кода без напуштања *D* мода. Како би успешно комуницирао са *DM*-ом при извршавању апстрактних команди, контролер врши детекцију грешака и обавештава *DM* о њима, као и о успешном завршетку извршавања апстрактне команде.

5.2.2 Модификације путање података

Како би се смањио број неопходних модификација путање података, одлучено је да се искористе исте линије које **INST_DECODE** користи, што је реализовано инкорпорирањем декодовања апстрактних команди у **INST_DECODE** тако да **D_CTL** одређује да ли се декодује инструкција или апстрактна команда. Како се операциони кодови директно мапирају на стања управљачке јединице, за потребе апстрактних команди је уведено пет нових операционих кодова о којима ће бити више речи при опису модификација управљачке јединице. Поред модификација **INST_DECODE**-а, додати су нови улази у мултиплексере који одређују меморијску адресу и величину приступа меморији, додат је нови мултиплексер који одређује одакле долазе подаци који се уписују у меморију. Такође су додата два улаза у мултиплексер који одређује следећу адресу програмског бројача а то су: вредност **DPC** регистра (користи се при поновном покретању језгра) и фиксна вредност локације почетка програмског бафера у меморијској мапи (користи се при извршавању произвољних инструкција).

[6]Уколико је **ebreakm** бит подешен.

[7]Уколико је **step** бит подешен.

[8]Уколико језгро није спремо да изврши тренутну инструкцију када добије захтев за заустављање, ипак се бележи тренутна адреса.

5.2.3 Модификације управљачке јединице

У исечку 5.3 су приказана нова стања и контролне линије додате у односу на оне приказане у примеру 4.1. Језгро се налази у стању **HALTED** када је заустављено и не извршава апстрактну команду, језгро улази у ово стање уколико је захтевано да се језгро заустави и прошли такт је био последњи такт инструкције или је језгро било у **PROLOGUE** стању. Језгро пролази кроз стање **RESUMING** при поновном покретању, тада се у програмски бројач уписује вредност **DPC** регистра и језгро следећег такта прелази у **PROLOGUE** стање. Што значи да постоји кашњење од два такта од тренутка захтева за поновно покретање до провг такта извршавања инструкције.

```
‘define CONTROL_SIGNALS__HALTED     5’b01_110
‘define CONTROL_SIGNALS__RESUMING   5’b01_111
‘define CONTROL_SIGNALS__ABS_REG    ‘DEBUG__OPCODE_ACCESS_REG
‘define CONTROL_SIGNALS__ABS_NA     ‘DEBUG__OPCODE_ACCESS_NA
‘define CONTROL_SIGNALS__ABS_EXEC   ‘DEBUG__OPCODE_EXEC
‘define CONTROL_SIGNALS__ABS_RMEM   ‘DEBUG__OPCODE_READ_MEM
‘define CONTROL_SIGNALS__ABS_RMEM_1 (‘DEBUG__OPCODE_READ_MEM + 5’d1)
‘define CONTROL_SIGNALS__ABS_WMEM   ‘DEBUG__OPCODE_WRITE_MEM
‘define CONTROL_SIGNALS__ABS_WMEM_1 (‘DEBUG__OPCODE_WRITE_MEM + 5’d1)

interface control_signals_if;
    wire [‘ISA__FUNCT3_WIDTH-1:0] f3;
    reg  [‘ISA__OPCODE_WIDTH-1:0] mcp_addr;
    reg                           write_pc_ne, write_pc_ex; // Ticu se implementacionih
        detalja podrske za prekide, nece biti diskutovane
    reg                           write_csr; // Tice se implementacionih detalja Zicsr
        ekstenzije, nece biti diskutovan
    reg                           abstract_write, abstract_done, progbuf;
endinterface
```

Исечак кода 5.3: Додати контролни сигнали управљачке јединице

Остала нова стања се користе при извршавању апстрактних команди. У одговарајуће стање се улази када је језгро заустављено и **D_CTL** реконфигурише путању података, у које тачно стање се улази зависи од операционог кода декодованог у **INST_DECODE**. **ABS_REG** се користи за *Access Register Command* уколико приступа регистрима, уколико само извршава програмски бафер или је у питању *Quick Access*, језгро прелази у **ABS_EXEC** стање, језгро такође може доћи у ово стање ако команда приступа регистрима и потом извршава програмски бафер. Уколико команда не приступа регистрима нити извршава програмски бафер (команда нема ефекта) језгро прелази у стање **ABS_NA**. **ABS_RMEM** и **ABS_WMEM** се користе при читању и писању меморије. Као и при извршавању инструкција и при извршавању апстрактних команди, приступи меморији морају да се реализују кроз два такта чему служе стања **ABS_RMEM_1** и **ABS_WMEM_1**. Уколико је магистрала блокирана линијом **inhibit** (до чега у пракси не може доћи) како је једина разлика између стања **LOAD** и **LOAD_W**[9] била упис у инструкцијски регистар а **ABS_RMEM** и **ABS_WMEM** тај упис не врше, није потребно имати одвојена стања **ABS_RMEM_W** и **ABS_WMEM_W** већ је довољно да језгро остане у истом стању уколико приступ меморији није успео. Након извршене апстрактне команде, језгро се враћа у стање **HALTED**. Контролни сигнали који су активни у овим стањима су приказани у исечку 5.4.

[9] Исто важи и за стања **STORE** и **STORE_W**.

Како би управљачка јединица разликовала апстрактне команде које уписују или читају регистре, да ли су у питању регистри опште намене или контролни и статусни регистри, као и да ли се након приступа регистрима извршава програмски бафер, **INST_DECODE** ове информације енкодује у линији **f3** којој сада и управљачка јединица има приступ. Линија **mcp_addr** садржи тренутно стање управљачке јединице, овим се избегава додавање контролних сигнала који не учествују у путањи података а користе се у само једном стању. Ту функционалност екстензивно користе **INT_CTL**, **D_CTL** и **T_CTL**. Сигналима **abstract_write**, **abstract_done** и **progbuf** управљачка јединица сигнализира **D_CTL**-у да жели да упише резултат апстрактне команде у регистар, да је апстрактна команда извршена и да жели да почне са извршавањем програмског бафера респективно.

```
            `CONTROL_SIGNALS__RESUMING: begin
                control_signals.write_pc_ex = 1'b1;
            end
            `CONTROL_SIGNALS__ABS_REG: begin
                if (control_signals.f3[2]) begin
                    if (control_signals.f3[1]) begin
                        control_signals.write_csr   = 1'b1;
                    end else begin
                        control_signals.alu_insel1 = `CONTROL_SIGNALS__ALU1_ZR;
                        control_signals.alu_insel2 = `CONTROL_SIGNALS__ALU2_IM;
                        `CONTROL__WRITE_ALU
                    end
                end else begin
                    control_signals.abstract_write = 1'b1;
                    if (control_signals.f3[1]) begin
                        control_signals.rd_sel     = `CONTROL_SIGNALS__RD_CSR;
                    end else begin
                        control_signals.alu_insel1 = `CONTROL_SIGNALS__ALU1_ZR;
                        control_signals.alu_insel2 = `CONTROL_SIGNALS__ALU2_RS;
                        control_signals.rd_sel     = `CONTROL_SIGNALS__RD_ALU;
                    end
                end
                control_signals.abstract_done = control_signals.f3[0] ? 1'b0 : 1'b1;
            end
            `CONTROL_SIGNALS__ABS_NA: begin
                control_signals.abstract_done = 1'b1;
            end
            `CONTROL_SIGNALS__ABS_EXEC: begin
                control_signals.progbuf     = 1'b1;
                control_signals.write_pc_ne = 1'b1;
            end
            `CONTROL_SIGNALS__ABS_RMEM: begin
                control_signals.mem_read = 1'b1;
            end
            `CONTROL_SIGNALS__ABS_RMEM_1: begin
                control_signals.rd_sel         = `CONTROL_SIGNALS__RD_MEM;
                control_signals.abstract_write = 1'b1;
                control_signals.abstract_done  = 1'b1;
            end
            `CONTROL_SIGNALS__ABS_WMEM: begin
                control_signals.mem_write  = 1'b1;
            end
            `CONTROL_SIGNALS__ABS_WMEM_1: begin
                control_signals.abstract_done = 1'b1;
            end
```

Исечак кода 5.4: Конфигурација контролих сигнала у додатим стањима управљачке јединице

5.3 Хардверски окидачи

Унутар језгра се налазе четири хардверска окидача чији је принцип функционисања описан у остатку секције. Хардверски окидачи подржавају окидање на адресу или вредност приступа меморији (било да је у питању дохватање инструкције, упис или читање меморије). При окидању, језгро се зауставља, прелази у режим за дебаговање и обавештава *DM* помоћу линије **halted**. Спецификација предвиђа осам контролних и статусних регистара кроз које се врши конфигурација окидача, од којих су 4 опциона и у овом случају неимплементирана. Имплементирани регистри су: **tdata1**, **tdata2**, **tinfo** и **tselect**. Сваки окидач има свој примерак прва три регистра, док регистар **tselect** одређује регистрима којег окидача се приступа. Регистар **tdata1** служи за конфигурацију окидача, подржане опције су: избор окидања на адресу или вредност приступа меморији[10], избор величине приступа меморији (8 бита, 16 бита, 32 бита или било која), избор окидања на било коју комбинацију дохватања инструкције, читања меморије и уписа у меморију. Регистар **tdata2** садржи вредност која се користи за поређење а **tinfo** је регистар намењен само за читање који садржи информацију о типовима окидача које тренутно изабран окидач подржава. Примећено је да тренутно доступни дебагери не проверавају овај регистар већ претпостављају да је једини подржан тип окидача наведен у регистру **tdata1**, ово онемогућава коришћење алтернативних типова окидача код окидача који подржавају више типова окидача, иако је то предвиђено спецификацијом.

Подршка за окидаче је имплементирана у компоненти **T_CTL** која имплементира **tselect** регистар инстанцира одговарајући број окидача (компонента **TRIGGER**) и правилно мултиплексира регистре окидача у зависности од вредности **tselect** регистра. Како су у питању контролни и статусни регистри који се не налазе унутар **CSR** компоненте, раније приказан интерфејс који језгро користи за манипулацију контролним и статусним регистрима се користи у обрнутом смеру (при извршавању инструкције која модификује неки од ових регистара **CSR** поставља жељену вредност на линију **TDATA1_in** активну вредност на линију **TDATA1_write**[11]). Овај **_reg _in _write** интерфејс користи и сама компонента **TRIGGER** како би омогућила манипулацију својим регистрима. Што омогућава једноставно мултиплексирање регистара окидача повезивањем одговарајуће **_in** линије свих окидача на **_in** линију коју пружа интерфејс **CSR** компоненте, док се **_write** линија повезује на израз формата `csr_interface.TDATA1_write && tselect == i` где је `i` индекс окидача. Компонента **T_CTL** такође формира глобални **trigger** сигнал дисјункцијом сигнала окидања свих окидача. *DM* користи тај сигнал како би зауставио језгро када се неки од окидача окине. Сам окидач (реализован у компоненти **TRIGGER**) имплементира **tdata1**, **tdata2** и **tinfo** регистре, логику која онемогућава упис неподржане конфигурације у регистре и саму логику за поређење и окидање која је реализована комбинационо.

[10] Вредност прочитана из или уписана у меморију.

[11] Исто важи и за регистре **tdata2** и **tselect**.

6 *Debug Transport Module (DTM)*

DTM прима команде од дебагера помоћу адаптера који те команде преводи у операције протокола који *DTM* подржава. Имплементирани *DTM* користи *JTAG* протокол и примљене команде преводи у приступе *DMI* магистрали. Принцип функционисања *JTAG* протокола, као и *DTM*-а је дат у овом поглављу.

6.1 *JTAG* протокол

JTAG је комуникациони протокол који *DTM* користи за комуникацију са адаптером за екстерно дебаговање. У комуникацији учествују газда (често се назива *Automated Test Equipment (ATE)*) и слуга (често се назива *Test Access Port (TAP)*), у овом случају адаптер је газда а *DTM* слуга. Размена података је серијска и синхрона на сигнал татка који задаје *ATE*.

Физички, протокол се састоји од 4 линије[1] а спецификације за екстерно дебаговање понекад додају још опционих линија.

То су:

- **Test Clock (TCK)**, сигнал такта
- **Test Mode Select (TMS)**, сигнал који управља проласком кроз аутомат стања
- **Test Data In (TDI)**, подаци од *ATE*-а ка *TAP*-у
- **Test Data Out (TDO)**, подаци од *TAP*-а ка *ATE*-у

Опционе линије које прописује спецификација:

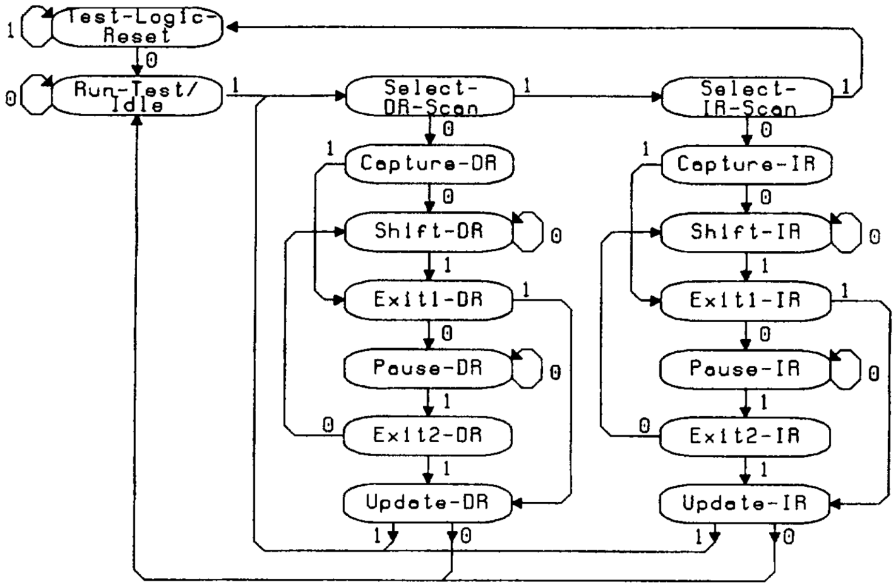
- **nRESET**, активан у нули, ресетује процесор и периферије
- **nTRST**, активан у нули, ресетује *TAP*

Концептуално *TAP* поседује шифт регистар подесиве ширине[2] и регистар инструкције, проласком кроз аутомат стања (приказан на слици 6.1) у одређеним тренуцима се врши паралелни упис или читање или серијски упис и читање шифт регистра. Аутомат стања унутар *TAP*-а мења стање на узлазну ивицу сигнала такта, стање у које прелази је спецификовано вредношћу сигнала **TMS** у том тренутку. Стања од интереса су **Capture-IR**, **Capture-DR**, **Shift-IR**, **Shift-DR**, **Update-IR** и **Update-DR**. У стању **Capture-IR** се на узлазну ивицу **TCK** у шифт регистар паралелно учитава произвољан низ битова ширине инструкцијског регистра (параметар *TAP*-а, мора бити константан) чија два најнижа бита морају бити 01 и ширина шифт регистра се поставља

[1] Постоји и верзија са 2 линије али она неће бити обрађивана овде.

[2] Спецификација користи више шифт регистара али аутор сматра да је репрезентација са једним шифт регистром лакша за разумевање.

на ширину инструкцијског регистра. У стању **Capture-DR** се на узлазну ивицу **TCK** у шифт регистар паралелно учитава вредност регистра за податке спецификована тренутном вредношћу инструкцијског регистра и ширина шифт регистра се поставља на ширину тог регистра (мора бити фиксна у зависности од инструкције, различите инструкције могу имати регистре различите дужине). У стањима **Shift-IR** и **Shift-DR** се на узлазну ивицу **TCK** у шифт регистар серијски уписује на место највишег бита вредност **TDI**, док се на силазну ивицу **TCK** на линију **TDO** поставља вредност најнижег бита шифт регистра. У стањима **Update-IR** и **Update-DR** се на силазну ивицу **TCK** паралелно уписује у инструкцијски регистар или регистар података спецификован тренутном вредношћу инструкцијског регистра. При упису регистар за податке, *TAP* може такође извршити произвољну акцију (као што је операција на магистрали).



Слика 6.1: Аутомат стања *JTAG* протокола [17]

У *JTAG* протоколу се комбинација инструкцијског регистра и регистара за податке може замислити као меморија где вредност инструкцијског регистра представља адресу а регистри за податке представљају податке, разлика је што различити регистри за податке могу бити различите ширине.

RISC-V спецификација [5] захтева да компатибилни *TAP* имплементира **BYPASS** и **IDCODE** *JTAG* инструкције, као и **DTMCS** и **DMI** које су специфичне за *RISC-V DTM*. **BYPASS** инструкција представља регистар намењен само за читање ширине једног бита чија је вредност 0. Намена ове инструкције је скраћивање ланца шифт регистара уколико је више *TAP*-ова редно повезано и *ATE* задаје команду неком другом уређају. Инструкција **IDCODE** представља регистар намењен само за читање ширине 32 бита и садржи јединствени идентификатор *TAP*-а. **DTMCS** омогућава приступ статусу *DTM*-а (бити статуса су намењени само за читање) и пружа два бита за ресетовање *DTM*-а. Један бит прекида тренутну *DMI* трансакцију а други ресетује индикатор грешке. Како су у овој имплементацији *DMI* трансакције трајања једног такта, *ATE* не може стићи да приступи неком регистру *DTM*-а док је трансакција у току, те је немогуће прекинути трансакцију у току или изазвати грешку па бити за ресет нису имплементирани. Регистар **DMI** инструкције је ширине 34 бита + ширина адресе *DMI* адресног простора. Уписом у овај регистар *ATE* може (у зависности од поља операције) извршити приступ преко *DMI* ма-

гистрале. При следећем читању **DMI** регистра, у њему се налази информација о успешности приступа и прочитана вредност уколико је била реч о операцији читања.

6.2 Имплементација *DTM*-а

Како су улазни сигнали асинхрони у односу на унутрашњи сигнал такта, они прво пролазе кроз ланац за синхронизацију где се за сигнале **Test Mode Select (TMS)**, **Test Data In (TDI)** и **Test Data Out (TDO)** користи вредност другог регистра у ланцу а за сигнал **Test Clock (TCK)** се детектују узлазне и силазне ивице коришћењем вредности другог и трећег регистра у ланцу. Имплементација саме *TAP* логике се своди на имплементацију аутомата стања, шифт регистра и учитавања правилних вредности у шифт регистар, инструкцијски регистар и регистре за податке, као што је објашњено изнад.

Само извршавање операција на *DMI* магистрали се своди на постављање одговарајућих делова **DMI** регистра на линије магистрале, такт након уписа у **DMI** регистар и учитавања информације о успешности приступа и прочитане вредности (уколико се радило о операцији читања) у **DMI** регистар такт касније.

7 Конфигурација софтвера

Како би могао да ефикасно проналази и отклања грешке у софтверу који развија, програмер мора имати приступ подршци за екстерно дебаговање на нивоу апстрактнијем од манипулисања линијама *JTAG* интерфејса. Ово поглавље приказује софтверску инфраструктуру и конфигурацију потребну за апстрактан приступ стању процесора. Како би ово било могуће, пре свега је потребно повезати *JTAG* интерфејс циљног уређаја са десктоп рачунаром. За то се користе *JTAG* адаптери који повезаном рачунару омогућавају манипулацију линијама *JTAG* интерфејса или директан упис у **IR** или **DR** *TAP*-а (у зависности од количине логике унутар самог адаптера). У овом раду је коришћен *J-Link* који у зависности од коришћеног софтвера може радити на било ком од претходно описаних нивоа апстракције.

Како би се постигао жељени ниво апстракције, потребно је додати још један слој који разуме спецификацију подршке за екстерно дебаговање повезаног чипа (у овом случају, то је [5]). У овом раду су упоређена два таква софтвера: Власнички софтвер који долази уз *J-Link* адаптер и *OpenOCD* који представља решење отвореног кода. Независно од тога који се софтвер користи, потребна је одређена конфигурација како би софтвер могао правилно да комуницира са циљним уређајем. Конфигурација *J-Link* софтвера се може обавити кроз аргументе командне линије или интерактивно.

```
Type ''connect'' to establish a target connection, '?' for help
J-Link>connect
Please specify device / core. <Default>: RISC-V
Type '?' for selection dialog
Device>RISC-V
Please specify target interface:
J) JTAG (Default)
S) SWD
T) cJTAG
TIF>JTAG
Device position in JTAG chain (IRPre,DRPre) <Default>: -1,-1 => Auto-detect
JTAGConf>-1,-1
Specify target interface speed [kHz]. <Default>: 4000 kHz
Speed>1000
Device ''RISC-V'' selected.
```

Исечак кода 7.1: Интерактивна конфигурација *J-Link* софтвера неопходна за повезивање на имплементирано језгро

Конфигурација еквивалентна интерактивној конфигурацији приказаној у исечку 7.1 се може постићи и следећим аргументима командне линије:

```
JLink.exe -device RISC-V -if JTAG -jtagconf -1,-1 -speed 1000 -autoconnect 1.
```

Конфигурација *OpenOCD*-а се врши кроз конфигурационе фајлове који се прослеђују као аргументи командне линије. Често се конфигурација дели на два фајла: конфигурација самог адаптера и конфигурација циљног уређаја. *OpenOCD* долази са великим бројем конфигурационих фајлова за већину подржаних адаптера и велики број развојних плоча. У овом случају

OpenOCD пружа готов конфигурациони фајл за *J-Link*, док је неопходно написати конфигурацију циљног уређаја (пример ове конфигурације је дат у исечку 7.2).

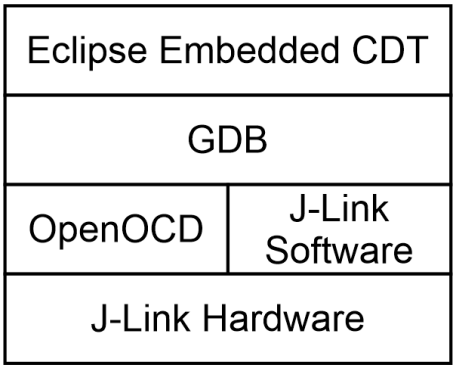
```
adapter speed 1000
transport select jtag

set _CHIPNAME riscv
jtag newtap $_CHIPNAME cpu -irlen 5 -expected-id 0x00537291

set _TARGETNAME $_CHIPNAME.cpu
target create $_TARGETNAME.0 riscv -chain-position $_TARGETNAME
$_TARGETNAME.0 configure -work-area-phys 0x00000000 -work-area-size 0x10000

init

halt
```

Исечак кода 7.2: Конфигурациони фајл циљног уређаја за *OpenOCD* (**board\custom_riscv.cfg**)

Како би се *OpenOCD* правилно покренуо, потребно је навести следеће аргументе: `openocd -f ''interface\jlink.cfg'' -f''board\custom_riscv.cfg''`.

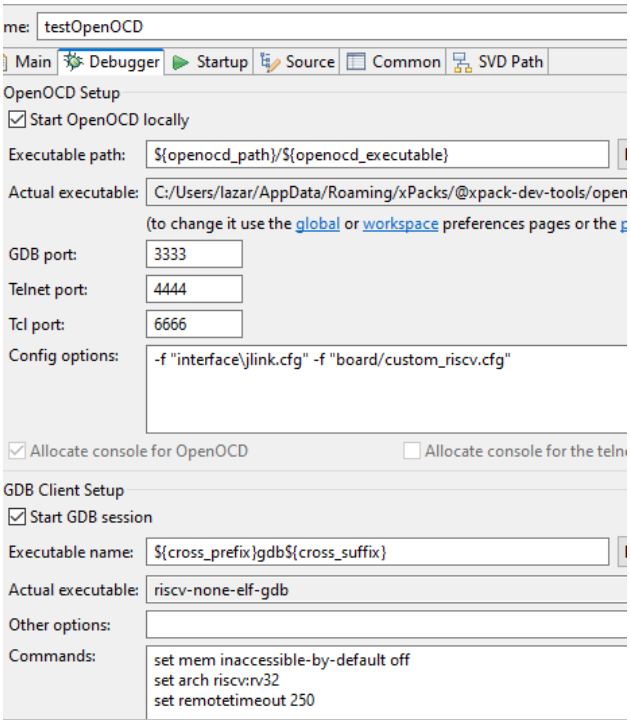
Као што се може видети, ове две конфигурације садрже већински исте информације: архитектуру процесора, комуникациони интерфејс и његову брзину. Конфигурациони фајл за *OpenOCD* такође мора да садржи ширину **IR** регистра и индекс *TAP*-а у ланцу, док *J-Link* аутоматски детектује ове податке. Очекивани идентификатор *TAP*-а, базна адреса и величина меморије су опциони.

Иако ови софтвери значајно подижу ниво апстракције, они немају концепт симбола јер нису свесни кода који се заправо извршава на циљном уређају. За то је потребан софтвер који, поред комуникације са циљним уређајем, разуме *Executable and Linkable Format (ELF)* и *DWARF*[1] информације за дебаговање унутар њега. Један од таквих алата је *GDB* који се повезује са инстанцом *gdbserver*-а коју покреће *OpenOCD* или *J-Link* софтвер. Како *GDB* пружа само текстуални кориснички интерфејс, он се може повезати са интегрисаним развојним окружењем као што је *Eclipse Embedded CDT* како би се омогућило дебаговање у комфору графичког корисничког интерфејса, што резултује у коначном софтверском стеку приказаном на слици 7.1.

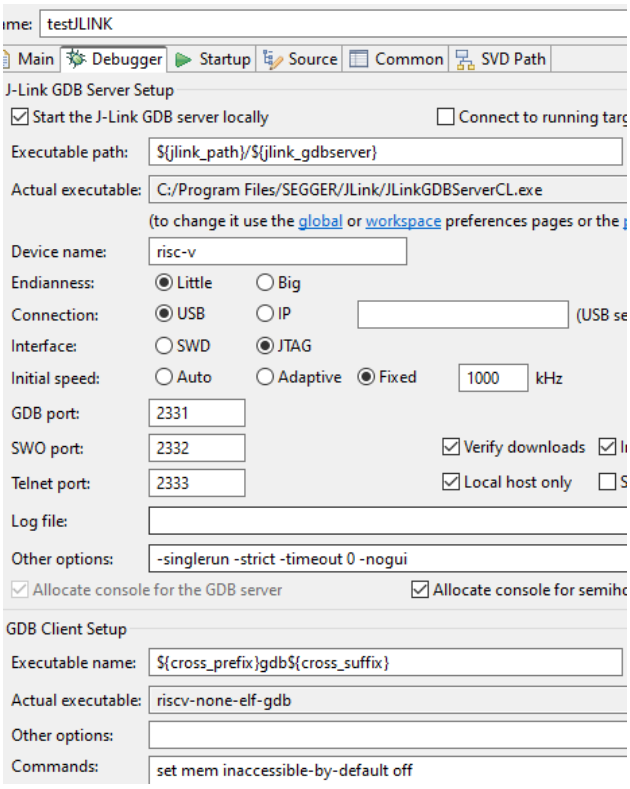


Слика 7.1: Коначни стек алата

[1] За више информација о *DWARF*-у видети [21].



Слика 7.2: Конфигурација за *OpenOCD*



Слика 7.3: Конфигурација за *J-Link* софтвер

Како *Eclipse Embedded CDT* има уграђену подршку за ове алате, потребно је само конфигурисати неколико параметара по упутству које се налази на званичном сајту [14]. Конкретне конфигурације за *OpenOCD* и *J-Link* софтвер су приказане на сликама 7.2 и 7.3 и већински садрже исте информације као конфигурације приказане раније, само у другом формату. У прилогу Г је приказан пример дебаговања коришћењем *Eclipse Embedded CDT* интегрисаног развојног окружења.

Како би програм написан за имплементирани систем могао и да се извршава на њему, неопходно је да постоји неколико помоћних фајлова који се користе при превођењу кода. То су: скрипта за алат *Make*, скрипта за повезивач (енг. *Linker*) и иницијализациони фајл написан у асемблеру (приказан у исечку 7.3). Ови фајлови су базирани на истим који су коришћени при изради [19]. Задатак скрипте за *Make* алат је превођење свих фајлова са изворним кодом и потом њихово повезивање коришћењем скрипте за повезивач. Скрипта такође генерише и *.mif* фајл који се може користити за иницијализацију садржаја меморије при конфигурисању *FPGA* чипа, чија је примена примарно у тестирању језгра. При позивању преводиоца, битно је проследити следеће аргументе: `-march=rv32i_zicsr -mabi=ilp32 -mstrict-align -nostdlib -ffreestanding` који дефинишиу архитектуру и подршане екстензије, конвенцију позивања, као и да се програм извршава у окружењу без стандардне библиотеке и оперативног система. Скрипта за повезивач дефинише базну адресу и величину меморије, поставља табелу прекидних рутина на право место и дефинише вредности показивача на врх стека, глобалног показивача, почетка и краја секције за неиницијализоване податке. Те вредности се потом користе у иницијализационом фајлу, који дефинише вредности табеле прекидних рутина, дефинише подразумевану и ресет прекидну рутину. Ресет прекидна рутина уписује вредности глобалног и стек показивача у одговарајуће регистре, иницијализује секцију са неиницијализованим подацима

нулама и скаче на **main** функцију. Иницијализација секције са неиницијализованим подацима се врши јер при ресетовању језгра садржај меморије остаје исти. Ова поновна иницијализација би требала да се врши и за податке са иницијалним вредностима, међутим то би захтевало постојање две копије тих података у меморији, те та функционалност није имплементирана.

```
.extern __global_pointer$
.extern __stack_pointer
.extern main
.global _start

.weak nmi_handler
.set nmi_handler, default_handler
.weak exception_handler
.set exception_handler, default_handler
.weak timer_handler
.set timer_handler, default_handler
.weak exti_handler
.set exti_handler, default_handler

.section .ivt, ”a”

j _start
j nmi_handler
j exception_handler
.rept 6
        j default_handler
.endr
j timer_handler
.rept 3
        j default_handler
.endr
j exti_handler

.section .startup

.type _start, %function
_start:
    .option push
    .option norelax
    la gp, __global_pointer$
    .option pop
    la sp, __stack_pointer
    la t0, __bss_start
    la t1, __BSS_END__
    li t2, 0
loop:
        beq t0,t1, next
        sw t2, 0(t0)
        addi t0, t0, 4
        j loop
next:
    call main
    j .

.type default_handler, %function
default_handler:
    j .

.end
```

Исечак кода 7.3: Иницијализациони фајл

8 Резултати

У овом поглављу ће укратко бити објашњено које су методе тестирања коришћене при изради рада и њихови резултати.

При имплементацији језгра, за тестирање су коришћени кодови и периферије оригинално написане за [19]. Кодови су покретани и унутар симулатора и на *FPGA* чипу а успешност тестова је ручно процењивана. При имплементацији екстензија језгра и подршке за дебаговање је за тестирање примарно коришћена симулација са ручном манипулацијом сигнала и проценом успешности тестова. Када је подршка за екстерно дебаговање комплетно имплементирана први корак тестирања је био провера правилне детекције свих параметара и способности језгра. То је вршено коришћењем командне линије *J-Link* софтвера.

```
Device ''RISC-V'' selected.
Connecting to target via JTAG
ConfigTargetSettings() start
ConfigTargetSettings() end - Took 12us
TotalIRLen = 5, IRPrint = 0x01
JTAG chain detection found 1 devices:
#0 Id: 0x00537291, IRLen: 05, Unknown device
Debug architecture:
RISC-V debug: 0.13
AddrBits: 7
DataBits: 32
IdleClks: 0
Memory access:
Via system bus: Yes (8/16/32-bit accesses are supported)
Via ProgBuf: Yes (16 ProgBuf entries)
Via abstract command (AAM): May be tried as last resort
DataBuf: 12 entries
autoexec[0] implemented: Yes
Detected: RV32 core
Temp. halting CPU for for feature detection...
HW instruction/data BPs: 4
Support set/clr BPs while running: No
HW data BPs trigger before execution of inst
CSR access via abs. commands: Yes
Feature detection done. Restarting core...
BG memory access support: Via SBA
Memory zones:
Zone: ''Default'' Description: Default access mode
RISC-V identified.
```

Исечак кода 8.1: Пример успешне детекције способности језгра

У исечку 8.1 је приказан испис *J-Link* софтвера након повезивања на имплементиран процесор. Може се видети да је *J-Link* софтвер успешно детектовао подршку за приступ меморијској магистрали, извршавање програмског бафера, хардверске окидаче и приступ контролним и статусним регистрима.

Након конфигурације интегрисаног развојног окружења и писања тест кода који коришћењем тајмера одбројава секунде и приказује их на седмосегментним дисплејима, мануелно тестирање функционалности подршке за екстерно дебаговање је вршено и коришћењем командне линије *J-Link* софтвера, као и коришћењем интегрисаног развојног окружења. Одређене функционалности подложне грешкама (апстрактне команде и приступ меморији) су тестиране у симулатору. За потребе овог тестирања симулатор је аутоматизован скриптама које аутоматизују побуде, али је процена успешности тестова и даље остала мануелна. Скрипте коришћене за ове тестове су приложене у додатку Д. Кроз све фазе тестирања дизајн је синтетисан и вршена је мануелна инспекција генерисане нетлисте и порука пријављених при синтези. Такође је увек вршена аутоматска провера временских ограничења.

На самом крају је комплетан хардвер тестиран једним сетом тестова који је саставила *RISC-V* организација, овај сет тестова тестира језгро и подршку за екстерно дебаговање заједно. Тест је реализован покретањем *GDB*-а који се преко *OpenOCD*-а и адаптера повезује на језгро које се извршава на *FPGA* чипу и задавањем предефинисаних команди и поређењем излаза. Конфигурација потребна за овај вид тестирања се састоји од *OpenOCD* конфигурационог фајла, скрипте за повезивач и пајтон фајла који садржи податке о језгру. Ови фајлови су такође приложени у додатку Ђ. Тестови који су ручно означени као неприменљиви су тестови у којима аутоматска детекција није правилно детектовала да нека функционалност није подржана, што се може сматрати грешком у тесту. Треба напоменути да су ови тестови застарели али да актуелни сет тестова не садржи тестове за подршку за екстерно дебаговање и да је примарно дизајниран за тестирање имплементација у симулатору. Тестирање језгра овим тестовима је разматрано, међутим повезивање имплементације са тим тестовима је значајно компликованије и ван обима овог рада.

Резултат извршавања ових тестова је **27 неприменљивих** тестова и **45 успешних** тестова, од укупно **72 извршена** теста. Детаљнији запис извршавања тестова је доступан у додатку Е.

Треба напоменути да ништа од горе наведених тестова није замена са праву и темељну верификацију система, али је тако нешто далеко ван обима овог рада и може представљати рад сам за себе.

9 Закључак

Циљ овог рада је било имплементирање *RISC-V* језгра и подршке за његово екстерно дебаговање која поштује званичну спецификацију [5], ради бољег разумевања саме спецификације и потенцијалних унапређења. Од велике важности је такође било демонстрирати функционисање овог система са стандардним алатима које програмери заправо користе за проналажење и отклањање грешака у коду.

На основу извршених тестова је могуће закључити да је имплементациони део рада успешно реализован, и да је процес имплементације резултовао у усвајању дубоког знања које се тиче имплементираних спецификација.

Уочено је да је фокус на перформансама језгра (иако је резултовао у бољим перформансама у односу на претходну имплементацију) значајно повећао комплексност решења, те се за озбиљније имплементације предлаже одабир организације са проточном обрадом, или неке још модерније организације. Имплементација комплетне спецификације подршке за екстерно дебаговање је прилично комплексна и временски захтевна, без великог бенефита у подржаној функционалности. Ово је наравно одлика спецификације и може се рећи да се и не очекује имплементација комплетне спецификације већ избор једног од могућих подскупа који пружају све потребне функционалности. Како је циљ рада био разумевање комплетне спецификације имплементација великог дела спецификације се сматра оправданом, међутим за будуће имплементације се топло препоручује избор једно од могућих подскупа. Такође када се гледају ресурси потребни за имплементацију минималног подскупа и вредност коју подршка за екстерно дебаговање доноси, сматра се да је имплементација исте оправдана или чак препоручена. Још једна опсервација је да управо тај приступ произвољног избора подскупа који чини спецификацију повољном за имплементацију, представља највећу препреку постизању добре софтверске подршке. Софтвери ће се често фокусирати на најчешће имплементиране подскупе са слабом или непостојећом подршком за остале функционалности. Ово доводи до тога да софтвери не искоришћавају неке комплексне функционалности иако су доступне у хардверу јер нису довољно честе. Што умањује подстицај дизајнерима хардвера да те функционалности имплементирају, на крају резултујући у конвергенцији на неколико минималних подскупа са мнималним функционалностима неопходним за дебаговање.

Даљи рад у овој области се може фокусирати на имплементирано језгро и проширивање његове применљивости имплементацијом додатних екстензија инструкцијског сета или побољшање његових перформанси коришћењем алтернативних организација језгра. На пољу саме подршке за дебаговање се може радити на имплементацији додатних опционих функционалности или других типова хардверских окидача, као и подршци за манипулацију регистрима без заустављања језгра. Међутим вероватно најплоднија област за даље истраживање се тиче неинвазивног праћења рада система тј. трагова извршавања. Тренутно постоје две предложене спецификације за трагове извршавања на *RISC-V* системима. Те се детаљна анализа њихових предности и мана, као и предлог начина за постизање паритета функционалности између њих се сматра преко потребним.

Литература

[1] RISC-V International. *RISC-V International. RISC-V: The Open Standard RISC Instruction Set Architecture*. Септ. 13, 2023. URL: https://riscv.org/.

[2] Krste Asanović и др. *The Rocket Chip Generator*. Техн. изв. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[3] SiFive Inc. *SiFive. Leading the RISC-V Revolution*. Септ. 13, 2023. URL: https://www.sifive.com/.

[4] Tenstorrent Inc. *Tenstorrent. Scalable and Efficient Hardware for Deep Learning*. Септ. 13, 2023. URL: https://tenstorrent.com/.

[5] *RISC-V External Debug Support*. Ур. Tim Newsome. Ур. Megan Wachs. Верзија 0.13.2. RISC--V Foundation, Мар. 22, 2019.

[6] *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Ур. Andrew Waterman. Ур. Krste Asanović. Верзија 20191213. RISC-V Foundation, Дец. 13, 2019.

[7] SiFive Inc. *HiFive1 Rev B. SiFive*. Септ. 14, 2019. URL: https://www.sifive.com/boards/hifive1-rev-b.

[8] *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Ур. Andrew Waterman. Ур. Krste Asanović. Ур. John Hauser. Верзија 20211203. RISC-V International, Дец. 4, 2021.

[9] SiFive Inc. *SiFive FE310-G002 Manual*. Верзија v1p5. Септ. 22, 2022.

[10] Altera Corporation. *Cyclone V Device Handbook, Volume 1: Device Interfaces and Integration*. Верзија CV-5V2. Јул 5, 2022.

[11] „IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language”. У: *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018), стр. 1–1315. DOI: 10.1109/IEEESTD.2018.8299595.

[12] Intel Corporation. *Intel Quartus Prime Standard Edition User Guides - Combined*. Дец. 16, 2019.

[13] Intel Corporation. *Questa Intel FPGA Edition Simulation User Guide*. Јун 7, 2023.

[14] Liviu Ionescu и др. *Eclipse Embedded CDT. Eclipse Embedded CDT (C/C++ Development Tools)*. Септ. 15, 2023. URL: https://eclipse-embed-cdt.github.io/.

[15] Free Software Foundation Inc. *GCC online documentation. GNU Project*. Септ. 15, 2023. URL: https://gcc.gnu.org/onlinedocs/.

[16] The OpenOCD Project. *OpenOCD User's Guide. Top*. Септ. 15, 2023. URL: https://openocd.org/doc/html/index.html.

[17] „IEEE Standard for Test Access Port and Boundary-Scan Architecture”. У: *IEEE Std 1149.1--2013 (Revision of IEEE Std 1149.1-2001)* (2013), стр. 1–444. DOI: 10.1109/IEEESTD.2013.6515989.

[18] Segger Microcontroller. *Debug Probes - J-Link & J-Trace. SEGGER Wiki*. Септ. 15, 2023. URL: https://wiki.segger.com/Debug_Probes_-_J-Link_%26_J-Trace.

[19] Lazar Premović, Aleksa Marković и Luka Simić. *topofkeks/arilla: Arilla. a RISC-V based microcomputer system, with a PS2 mouse controller and 12-bit RGB SVGA graphics card, running Arilla Paint*. Септ. 16, 2023. URL: https://github.com/topofkeks/arilla.

[20] Terasic Inc. *DE10-Standard User Manual*. Јан. 19, 2017.

[21] Lazar Premović. *Kako pronalazimo greške u programima? Predavanje za nedelju informatike v8.0*. Maj 15, 2023. URL: http://ni.mg.edu.rs/static/resources/v8.0/greske_u_kodu.pdf.

[22] DA Patterson и JL Hennessy. *Computer organization and design RISC-V edition: The hardware software interface*. 2nd издање. Morgan Kaufmann, 2021.

Списак скраћеница

ABSTRACTCS Abstract Control and Status.
ATE Automated Test Equipment.

CDT C/C++ Development Tools.

DCSR Debug Control and Status Register.
DDR Data Direction Register.
DIR Data Input Register.
DM Debug Module.
DMCONTROL Debug Module Control.
DMI Debug Module Interface.
DMSTATUS Debug Module Status.
DOR Data Output Register.
DPC Debug Program Counter.
DR Data Register.
DTM Debug Transport Module.
DTMCS Debug Transport Module Control and Status.
DWARF Debugging With Arbitrary Record Formats.

ELF Executable and Linkable Format.
EXTI External Interrupt.

FER Falling Edge Register.
FPGA Field Programmable Gate Array.

GCC GNU Compiler Collection.
GDB GNU Debugger.
GPIO General Purpose Input Output.

HALTSUM Halt Summary.

IDE Integrated Development Environment.
IMR Interrupt Mask Register.
IPR Interrupt Pending Register.
IR Instruction Register.
ISR Interrupt Set Register.

JTAG Joint Test Action Group.

LE Logic Element.

MCAUSE Machine Cause.
MEPC Machine Exception Program Counter.
MIE Machine Interrupt Enable.
MIP Machine Interrupt Pending.
MPIE Machine Previous Interrupt Enable.
MSTATUS Machine Status.
MTVAL Machine Trap Value.

OCD On-Chip Debugger.

PC Program Counter.
PLL Phase Locked Loop.
POR Power-on Reset.
PROGBUF Program Buffer.

RER Rising Edge Register.
RISC Reduced Instruction Set Computer.

SBADDRESS System Bus Address.
SBCS System Bus Control and Status.
SBDATA System Bus Data.

TAP Test Access Port.
TCK Test Clock.
TDI Test Data In.
TDO Test Data Out.
TMS Test Mode Select.
TRST Test Reset.

ИПЦ Инструкције По Циклусу.

Списак слика

Списак табела

Списак исечака кода

А Управљачка јединица

```
1  `define CONTROL__READ_INST   control_signals.addr_sel    = `CONTROL_SIGNALS__ADDR_PC; \
2                               control_signals.mem_read    = 1'b1;
3  `define CONTROL__NEXT_INST   control_signals.write_pc_ne = 1'b1; \
4                               `CONTROL__READ_INST
5  `define CONTROL__ALU_ADDRESS control_signals.alu_insel1  = `CONTROL_SIGNALS__ALU1_RS; \
6                               control_signals.alu_insel2  = `CONTROL_SIGNALS__ALU2_IM; \
7                               control_signals.addr_sel    = `CONTROL_SIGNALS__ADDR_ALU;
8  `define CONTROL__WRITE_ALU   control_signals.rd_sel      = `CONTROL_SIGNALS__RD_ALU; \
9                               control_signals.write_rd    = 1'b1;
10 `define CONTROL__WRITE_MEM   control_signals.rd_sel      = `CONTROL_SIGNALS__RD_MEM; \
11                              control_signals.write_rd    = 1'b1;
12 module control ( ... );
13     reg [`ISA__OPCODE_WIDTH-1:0] mcp_reg, mcp_next, mcp_addr;
14
15     always @(posedge clk) begin
16         if (!rst_n) mcp_reg <= `CONTROL_SIGNALS__PROLOGUE;
17         else mcp_reg <= mcp_next;
18     end
19
20     always_comb begin
21         mcp_addr = mcp_reg;
22         if (mcp_reg == `CONTROL_SIGNALS__DISPATCH) mcp_addr = control_signals.opcode;
23     end
24
25     assign control_signals.write_ir = !
26         (  mcp_reg == `CONTROL_SIGNALS__PROLOGUE
27         || mcp_reg == `CONTROL_SIGNALS__LOAD_W
28         || mcp_reg == `CONTROL_SIGNALS__LOAD_1
29         || mcp_reg == `CONTROL_SIGNALS__STORE_W
30         || mcp_reg == `CONTROL_SIGNALS__STORE_1);
31
32     always_comb begin
33         control_signals.write_pc    = 1'b0;
34         control_signals.write_rd    = 1'b0;
35         control_signals.write_csr   = 1'b0;
36         control_signals.mem_read    = 1'b0;
37         control_signals.mem_write   = 1'b0;
38         control_signals.addr_sel    = `CONTROL_SIGNALS__ADDR_PC;
39         control_signals.rd_sel      = `CONTROL_SIGNALS__RD_ALU;
40         control_signals.alu_insel1  = `CONTROL_SIGNALS__ALU1_RS;
41         control_signals.alu_insel2  = `CONTROL_SIGNALS__ALU2_RS;
42         case (mcp_addr)
43             `CONTROL_SIGNALS__PROLOGUE: begin
44                 `CONTROL__READ_INST
45             end
46             `CONTROL_SIGNALS__LUI: begin
47                 control_signals.alu_insel1 = `CONTROL_SIGNALS__ALU1_ZR;
48                 control_signals.alu_insel2 = `CONTROL_SIGNALS__ALU2_IM;
49                 `CONTROL__WRITE_ALU
50                 `CONTROL__NEXT_INST
51             end
52             `CONTROL_SIGNALS__AUIPC: begin
53                 control_signals.alu_insel1 = `CONTROL_SIGNALS__ALU1_PC;
54                 control_signals.alu_insel2 = `CONTROL_SIGNALS__ALU2_IM;
```

```
55                  `CONTROL__WRITE_ALU
56                  `CONTROL__NEXT_INST
57              end
58              `CONTROL_SIGNALS__JAL, `CONTROL_SIGNALS__JALR: begin
59                  control_signals.alu_insel1 = `CONTROL_SIGNALS__ALU1_PC;
60                  control_signals.alu_insel2 = `CONTROL_SIGNALS__ALU2_IS;
61                  `CONTROL__WRITE_ALU
62                  `CONTROL__NEXT_INST
63              end
64              `CONTROL_SIGNALS__BRANCH, `CONTROL_SIGNALS__STORE_1,
                    `CONTROL_SIGNALS__MISCMEM: begin
65                  `CONTROL__NEXT_INST
66              end
67              `CONTROL_SIGNALS__LOAD, `CONTROL_SIGNALS__LOAD_W: begin
68                  `CONTROL__ALU_ADDRESS
69                  control_signals.mem_read = 1'b1;
70              end
71              `CONTROL_SIGNALS__LOAD_1: begin
72                  `CONTROL__WRITE_MEM
73                  `CONTROL__NEXT_INST
74              end
75              `CONTROL_SIGNALS__STORE, `CONTROL_SIGNALS__STORE_W: begin
76                  `CONTROL__ALU_ADDRESS
77                  control_signals.mem_write = 1'b1;
78              end
79              `CONTROL_SIGNALS__OPIMM: begin
80                  control_signals.alu_insel1 = `CONTROL_SIGNALS__ALU1_RS;
81                  control_signals.alu_insel2 = `CONTROL_SIGNALS__ALU2_IM;
82                  `CONTROL__WRITE_ALU
83                  `CONTROL__NEXT_INST
84              end
85              `CONTROL_SIGNALS__OP: begin
86                  control_signals.alu_insel1 = `CONTROL_SIGNALS__ALU1_RS;
87                  control_signals.alu_insel2 = `CONTROL_SIGNALS__ALU2_RS;
88                  `CONTROL__WRITE_ALU
89                  `CONTROL__NEXT_INST
90              end
91              `CONTROL_SIGNALS__SYSTEM: begin
92                  `CONTROL__NEXT_INST
93              end
94          endcase
95      end
96
97      always_comb begin
98          case (mcp_addr)
99              `CONTROL_SIGNALS__LOAD,
100             `CONTROL_SIGNALS__LOAD_W:  mcp_next = control_signals.mem_complete ?
                    `CONTROL_SIGNALS__LOAD_1   : `CONTROL_SIGNALS__LOAD_W;
101             `CONTROL_SIGNALS__STORE,
102             `CONTROL_SIGNALS__STORE_W: mcp_next = control_signals.mem_complete ?
                    `CONTROL_SIGNALS__STORE_1  : `CONTROL_SIGNALS__STORE_W;
103             default:                   mcp_next = control_signals.mem_complete ?
                    `CONTROL_SIGNALS__DISPATCH : `CONTROL_SIGNALS__PROLOGUE;
104         endcase
105     end
106 endmodule
```

Б *Debug Module (DM)*

```
1  `include "debug.svh"
2  `include "debug_if.svh"
3  `include "dmi_if.svh"
4  `include "../system/arilla_bus_if.svh"
5
6  module dm #(
7      parameter logic [11:0] BaseAddress
8  ) (
9      input clk,
10     input rst_n,
11
12     input  n_trst,
13     input  n_rst,
14
15     output vt_ref,
16     output reset_n,
17     output hart_reset_n,
18     output dtm_reset_n,
19
20     dmi_if        dmi,
21     debug_if      debug,
22     arilla_bus_if bus_interface,
23     output        mem_hit
24 );
25     localparam logic [31:0] FullAddress  = {{20{BaseAddress[11]}},BaseAddress};
26     localparam logic [11:0] DataStart    = BaseAddress + (`DEBUG__DATA0_OFFSET * 12'd4);
27     localparam int          BusDataWidth = $bits(bus_interface.data_ctp);
28     localparam int          DmiDataWidth = $bits(dmi.data);
29     localparam int          NumWords     = 32;
30
31     reg dmactive;
32
33     always @(posedge clk) begin
34         if (!rst_n) begin
35             dmactive <= 1'b0;
36         end else begin
37             if (dmi.address == `DEBUG__DMCONTROL && dmi.write) begin
38                 dmactive <= `DEBUG__DMCONTROL_DMACTIVE(dmi.data);
39             end
40         end
41     end
42
43     wire dm_reset = !rst_n || !dmactive;
44
45     reg ndmreset, hartreset, hart_reset_tail;
46     reg havereset, resumeack;
47     reg haltreq, resethaltreq;
48
49     reg                     busy, exec;
50     reg [             2:0] cmderr, cmderr_next;
51     reg [DmiDataWidth-1:0] command;
52     reg [DmiDataWidth-1:0] abstractauto;
53
54     reg                     sbbusy_error;
```

```
55      reg                     sbbusy;
56      reg                     sbreadonaddr;
57      reg [             2:0] sbaccess;
58      reg                     sbautoincrement;
59      reg                     sbreadondata;
60      reg [             2:0] sberror, sberror_next;
61      reg [DmiDataWidth-1:0] sbdata;
62      reg [DmiDataWidth-1:0] sbaddr, sbaddr_next;
63      reg                     sb_read;
64      reg                     sb_write;
65      reg                     sb_readout;
66      reg                     sb_writeout;
67
68      `DEBUGGEN__FOREACH_SIMPLE(DEBUGGEN__GENERATE_INTERFACE)
69
70      wire resumereq    = dmi.address == `DEBUG__DMCONTROL && dmi.write &&
            `DEBUG__DMCONTROL_RESUMEREQ(dmi.data) && !`DEBUG__DMCONTROL_HALTREQ(dmi.data);
71      wire ackhavereset = dmi.address == `DEBUG__DMCONTROL && dmi.write &&
            `DEBUG__DMCONTROL_ACKHAVERESET(dmi.data);
72
73      wire system_reset = !rst_n       || ndmreset || !n_rst;
74      wire hart_reset   = system_reset || hartreset;
75      wire dtm_reset    = !rst_n       || !n_trst;
76
77      assign vt_ref       = rst_n;
78      assign reset_n      = !system_reset;
79      assign hart_reset_n = !hart_reset;
80      assign dtm_reset_n  = !dtm_reset;
81
82      wire available = !hart_reset;
83      wire running   = available && !debug.halted;
84      wire halted    = available && debug.halted;
85
86      wire [DmiDataWidth-1:0] dmstatus   = {9'd0,1'b1,2'd0,havereset,havereset,resumeack,
            resumeack,2'd0,!available,!available,running,running,halted,halted,2'b10,2'b10,4'
            h2};
87      wire [DmiDataWidth-1:0] dmcontrol  = {2'd0,hartreset,27'd0,ndmreset,dmactive};
88      wire [DmiDataWidth-1:0] hartinfo   = {`DEBUG__HARTINFO_VALUE,DataStart};
89      wire [DmiDataWidth-1:0] abstractcs = {3'd0,5'd16,11'd0,busy,1'b0,cmderr,4'd0,4'd12};
90      wire [DmiDataWidth-1:0] haltsum    = {31'd0,halted};
91      wire [DmiDataWidth-1:0] sbcs       = {3'd1,6'd0,sbbusy_error,sbbusy,sbreadonaddr,
            sbaccess,sbautoincrement,sbreadondata,sberror,7'd32,5'b00111};
92
93      wor [DmiDataWidth-1:0] data;
94
95      assign data = 32'd0;
96      assign data = dmi.address == `DEBUG__DMSTATUS     ? dmstatus     : 32'd0;
97      assign data = dmi.address == `DEBUG__DMCONTROL    ? dmcontrol    : 32'd0;
98      assign data = dmi.address == `DEBUG__HARTINFO     ? hartinfo     : 32'd0;
99      assign data = dmi.address == `DEBUG__ABSTRACTCS   ? abstractcs   : 32'd0;
100     assign data = dmi.address == `DEBUG__ABSTRACTAUTO ? abstractauto : 32'd0;
101     assign data = dmi.address == `DEBUG__HALTSUM0     ? haltsum      : 32'd0;
102     assign data = dmi.address == `DEBUG__SBCS         ? sbcs         : 32'd0;
103     assign data = dmi.address == `DEBUG__SBADDRESS0   ? sbaddr       : 32'd0;
104     assign data = dmi.address == `DEBUG__SBDATA0      ? sbdata       : 32'd0;
105
106     `DEBUGGEN__FOREACH_SIMPLE(DEBUGGEN__GENERATE_READ_ASSIGN)
107
108     assign dmi.data = dmi.read ? data : {DmiDataWidth{1'bz}};
109
110     assign debug.halt_req   = haltreq || (resethaltreq && hart_reset_tail);
111     assign debug.resume_req = resumereq;
112     assign debug.exec       = exec;
113     assign debug.command    = command;
114     assign debug.data0_in   = DATA0_reg;
115     assign debug.data1_in   = DATA1_reg;
116
```

```
117     assign DATA0_in    = debug.data0_out;
118     assign DATA0_write = debug.write;
119
120     wire [(BusDataWidth*NumWords)-1:0] memory;
121
122     wire [BusDataWidth-1:0] mem_out;
123     wire [    NumWords-1:0] mem_write;
124
125     wire [BusDataWidth-1:0] sb_out;
126     wire                    sb_complete;
127     wire                    sb_malign;
128     wire                    sb_fault;
129
130     `DEBUGGEN__FOREACH_SIMPLE(DEBUGGEN__GENERATE_MEMORY_ASSIGN)
131     `DEBUGGEN__GENERATE_MEMORY_GUARD_ASSIGN
132
133     wire aar = `DEBUG__AC_COMMAND(command) == `DEBUG__AC_COMMAND_ACCESS_REGISTER;
134     wire aqa = `DEBUG__AC_COMMAND(command) == `DEBUG__AC_COMMAND_QUICK_ACCESS;
135     wire aam = `DEBUG__AC_COMMAND(command) == `DEBUG__AC_COMMAND_ACCESS_MEMORY;
136
137     assign sbaddr_next = sbaddr + (3'd2 ** sbaccess);
138     assign DATA1_in    = DATA1_reg + (3'd2 ** `DEBUG__AC_AARSIZE(command));
139     assign DATA1_write = debug.done && aam && `DEBUG__AC_AARPOSTINC(command) &&
            cmderr_next == `DEBUG__AC_ERR_NO_ERR;
140
141     wire busy_err = busy &&
142     (  dmi.address == `DEBUG__COMMAND      && dmi.write
143     || dmi.address == `DEBUG__ABSTRACTCS   && dmi.write
144     || dmi.address == `DEBUG__ABSTRACTAUTO && dmi.write
145     `DEBUGGEN__FOREACH_SIMPLE(DEBUGGEN__GENERATE_BUSY_ERROR)
146     );
147
148     wire notsupported_err = (aar && `DEBUG__AC_TRANSFER(command) && `DEBUG__AC_AARSIZE(
            command) != 3'd2) || (aam && (`DEBUG__AC_AARSIZE(command) == 3'd3 ||
            `DEBUG__AC_AARSIZE(command) == 3'd4)) || (aam && `DEBUG__AC_AAMVIRTUAL(command));
149     wire haltresume_err   = (aar && !halted) || (aqa && !running && !exec) || (aam && !
            halted) || (debug.haltresume && exec);
150
151     always_comb begin
152         cmderr_next = cmderr;
153         if (dmi.address == `DEBUG__ABSTRACTCS && dmi.write) begin
154             cmderr_next = cmderr_next & ~(dmi.data[10:8]);
155         end
156         if (busy) begin
157             if (cmderr_next != `DEBUG__AC_ERR_NO_ERR) begin
158                 cmderr_next = cmderr_next;
159             end else if (busy_err) begin
160                 cmderr_next = `DEBUG__AC_ERR_BUSY;
161             end else if (notsupported_err) begin
162                 cmderr_next = `DEBUG__AC_ERR_NOT_SUPPORTED;
163             end else if (debug.exception && exec) begin
164                 cmderr_next = `DEBUG__AC_ERR_EXCEPTION;
165             end else if (haltresume_err) begin
166                 cmderr_next = `DEBUG__AC_ERR_HALT_RESUME;
167             end else if (debug.bus && exec) begin
168                 cmderr_next = `DEBUG__AC_ERR_BUS;
169             end else begin
170                 cmderr_next = cmderr_next;
171             end
172         end
173         sberror_next = sberror;
174         if (dmi.address == `DEBUG__SBCS && dmi.write) begin
175             sberror_next = sberror_next & ~`DEBUG__SBCS_SBERROR(dmi.data);
176         end
177         if (sbbusy) begin
178             if (sberror_next != `DEBUG__SB_ERR_NO_ERR) begin
179                 sberror_next = sberror_next;
```

```
180             end else if (sb_fault && (sb_readout || sb_writeout)) begin
181                 sberror_next = `DEBUG__SB_ERR_FAULT;
182             end else if (sb_malign && (sb_readout || sb_writeout)) begin
183                 sberror_next = `DEBUG__SB_ERR_MALIGN;
184             end else if (sbaccess > 2 && (sb_read || sb_write)) begin
185                 sberror_next = `DEBUG__SB_ERR_SIZE;
186             end else begin
187                 sberror_next = sberror_next;
188             end
189         end
190     end
191
192     always @(posedge clk) begin
193         if (dm_reset) begin
194             ndmreset        <= 1'b0;
195             hartreset       <= 1'b0;
196             havereset       <= 1'b0;
197             resumeack       <= 1'b0;
198             haltreq         <= 1'b0;
199             resethaltreq    <= 1'b0;
200             hart_reset_tail <= 1'b0;
201             busy            <= 1'b0;
202             exec            <= 1'b0;
203             cmderr          <= `DEBUG__AC_ERR_NO_ERR;
204             command         <= {BusDataWidth{1'b0}};
205             abstractauto    <= {BusDataWidth{1'b0}};
206
207             sbbusy_error    <= 1'b0;
208             sbbusy          <= 1'b0;
209             sbreadonaddr    <= 1'b0;
210             sbaccess        <= 3'd2;
211             sbautoincrement <= 1'b0;
212             sbreadondata    <= 1'b0;
213             sberror         <= 3'd0;
214             sbdata          <= {BusDataWidth{1'b0}};
215             sbaddr          <= {BusDataWidth{1'b0}};
216             sb_read         <= 1'b0;
217             sb_write        <= 1'b0;
218             sb_readout      <= 1'b0;
219             sb_writeout     <= 1'b0;
220
221             `DEBUGGEN__FOREACH_SIMPLE(DEBUGGEN__GENERATE_INITIAL_VALUE_SIMPLE)
222         end else begin
223             havereset       <= (havereset || hart_reset) && !ackhavereset;
224             resumeack       <= (resumeack || running)    && !resumereq;
225             hart_reset_tail <= hart_reset;
226             cmderr          <= cmderr_next;
227             exec            <= busy && (cmderr_next == `DEBUG__AC_ERR_NO_ERR ||
                    cmderr_next == `DEBUG__AC_ERR_BUSY);
228             busy            <= busy && (cmderr      == `DEBUG__AC_ERR_NO_ERR || cmderr
                        == `DEBUG__AC_ERR_BUSY);
229             if (dmi.address == `DEBUG__DMCONTROL && dmi.write) begin
230                 ndmreset     <= `DEBUG__DMCONTROL_NDMRESET(dmi.data);
231                 hartreset    <= `DEBUG__DMCONTROL_HARTRESET(dmi.data);
232                 haltreq      <= `DEBUG__DMCONTROL_HALTREQ(dmi.data);
233                 resethaltreq <= `DEBUG__DMCONTROL_CLRRESETHALTREQ(dmi.data) ? 1'b0 :
                        `DEBUG__DMCONTROL_SETRESETHALTREQ(dmi.data) ? 1'b1 : resethaltreq;
234             end
235             `DEBUGGEN__FOREACH_SIMPLE(DEBUGGEN__GENERATE_WRITE_SIMPLE)
236             if (dmi.address == `DEBUG__COMMAND && dmi.write && cmderr ==
                    `DEBUG__AC_ERR_NO_ERR && busy == 1'b0) begin
237                 command <= dmi.data;
238                 busy    <= 1'b1;
239             end
240             if (dmi.address == `DEBUG__ABSTRACTAUTO && dmi.write) begin
241                 abstractauto <= dmi.data && 32'hFFFF0FFF;
242             end
```

```
243             sberror <= sberror_next;
244             if (dmi.address == `DEBUG__SBCS && dmi.write) begin
245                 sbbusy_error    <= sbbusy_error & ~`DEBUG__SBCS_SBBUSYERROR(dmi.data);
246                 sbreadonaddr    <= `DEBUG__SBCS_SBREADONADDR(dmi.data);
247                 sbaccess        <= `DEBUG__SBCS_SBACCESS(dmi.data);
248                 sbautoincrement <= `DEBUG__SBCS_SBAUTOINCREMENT(dmi.data);
249                 sbreadondata    <= `DEBUG__SBCS_SBREADONDATA(dmi.data);
250             end
251             if (dmi.address == `DEBUG__SBADDRESS0 && dmi.write) begin
252                 if (sbbusy) begin
253                     sbbusy_error <= 1'b1;
254                 end else if (sberror == `DEBUG__SB_ERR_NO_ERR && !sbbusy_error) begin
255                     sbaddr  <= dmi.data;
256                     sbbusy  <= sbreadonaddr;
257                     sb_read <= sbreadonaddr;
258                 end
259             end
260             if (dmi.address == `DEBUG__SBDATA0 && dmi.write) begin
261                 if (sbbusy) begin
262                     sbbusy_error <= 1'b1;
263                 end else if (sberror == `DEBUG__SB_ERR_NO_ERR && !sbbusy_error) begin
264                     sbdata   <= dmi.data;
265                     sbbusy   <= 1'b1;
266                     sb_write <= 1'b1;
267                 end
268             end
269             if (dmi.address == `DEBUG__SBDATA0 && dmi.read) begin
270                 if (sbbusy) begin
271                     sbbusy_error <= 1'b1;
272                 end else if (sberror == `DEBUG__SB_ERR_NO_ERR && !sbbusy_error) begin
273                     sbbusy  <= 1'b1;
274                     sb_read <= sbreadondata;
275                 end
276             end
277             `DEBUGGEN__FOREACH_SIMPLE(DEBUGGEN__GENERATE_AUTOEXEC)
278             if (debug.done) begin
279                 busy <= 1'b0;
280                 exec <= 1'b0;
281                 if (aar && `DEBUG__AC_AARPOSTINC(command) && cmderr_next ==
                        `DEBUG__AC_ERR_NO_ERR) begin
282                     command <= {command[31:16],(command[15:0]+16'd1)};
283                 end
284             end
285             if (sbbusy) begin
286                 if (sb_write) begin
287                     sb_write    <= 1'b0;
288                     sb_writeout <= 1'b1;
289                 end
290                 if (sb_read) begin
291                     sb_read    <= 1'b0;
292                     sb_readout <= 1'b1;
293                 end
294                 if (!(sb_write || sb_read)) begin
295                     sbbusy <= 1'b0;
296                     if (sb_readout) begin
297                         if(sberror_next == `DEBUG__SB_ERR_NO_ERR)begin
298                         sbdata <= sb_out;
299                         end
300                         sb_readout <= 1'b0;
301                     end
302                     if (sb_writeout) begin
303                         sb_writeout <= 1'b0;
304                     end
305                     if (sberror_next == `DEBUG__SB_ERR_NO_ERR && sbautoincrement) begin
306                         sbaddr <= sbaddr_next;
307                     end
308                 end
```

```
309             end
310         end
311     end
312 
313     periph_mem_interface #(
314         .BaseAddress(FullAddress),
315         .SizeWords  (NumWords)
316     ) periph_mem_interface (
317         .clk              (clk),
318         .rst_n            (!dm_reset),
319         .bus_interface    (bus_interface),
320         .hit              (mem_hit),
321         .data_periph_in   (memory),
322         .data_periph_out  (mem_out),
323         .data_periph_write(mem_write)
324     );
325 
326     assign bus_interface.inhibit = (sb_read && sberror_next == `DEBUG__SB_ERR_NO_ERR) ||
        (sb_write && sberror_next == `DEBUG__SB_ERR_NO_ERR);
327 
328     mem_interface #(
329         .InhibitPolarity(1'b1)
330     ) mem_interface (
331         .clk           (clk),
332         .rst_n         (!dm_reset),
333         .bus_interface (bus_interface),
334         .address       (sbaddr),
335         .sign_size     (sbaccess),
336         .rd            (sb_read && sberror_next == `DEBUG__SB_ERR_NO_ERR),
337         .wr            (sb_write && sberror_next == `DEBUG__SB_ERR_NO_ERR),
338         .data_in       (sbdata),
339         .data_out      (sb_out),
340         .complete      (sb_complete),
341         .malign        (sb_malign),
342         .fault         (sb_fault)
343     );
344 
345 endmodule
```

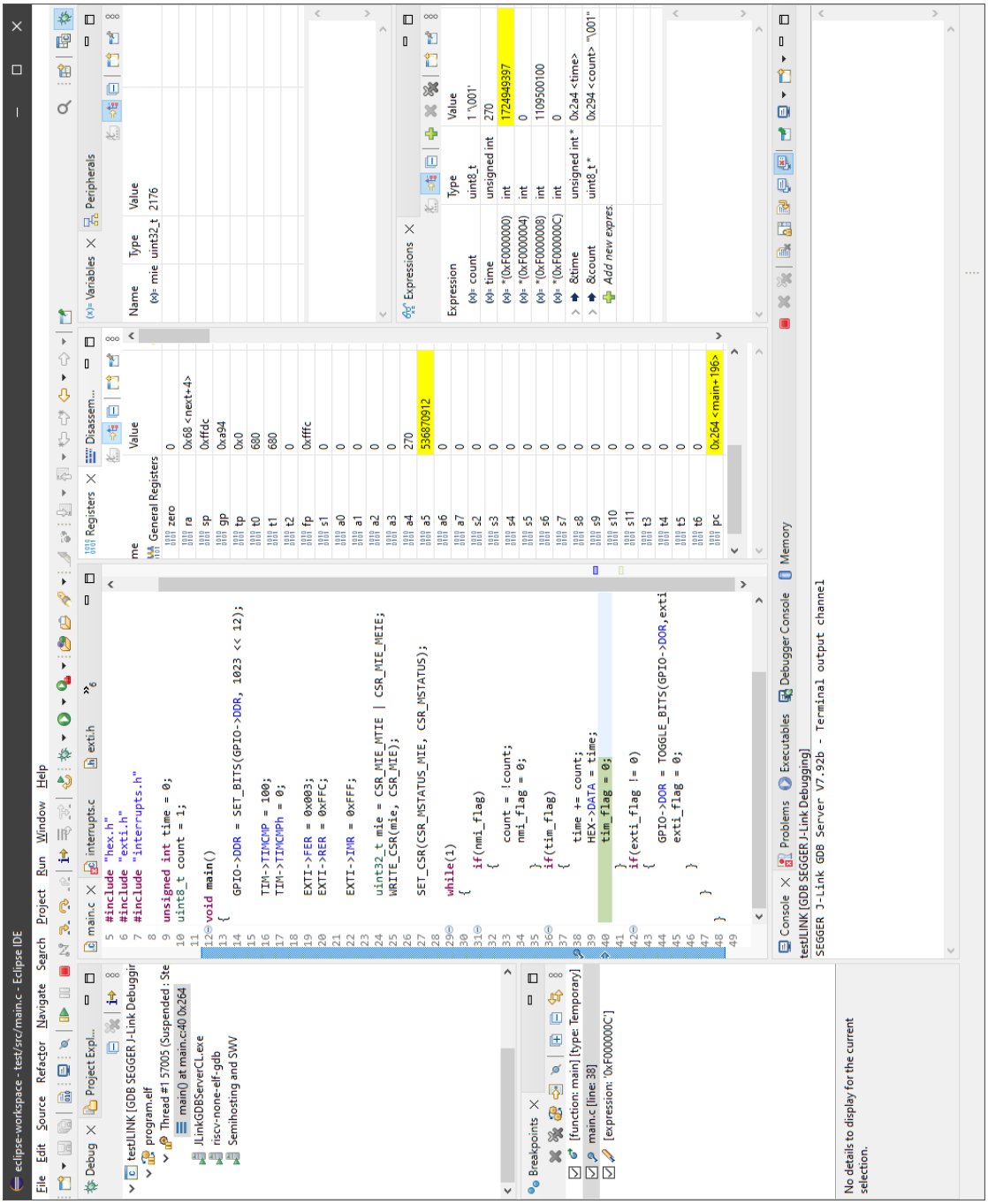
B D_CTL

```
1  `include "isa.svh"
2  `include "csr.svh"
3  `include "csr_if.svh"
4  `include "control_signals_if.svh"
5  `include "../debug/debug_if.svh"
6  `include "../debug/debug.svh"
7
8  module d_ctl (
9      input clk,
10     input rst_n,
11
12     input nmi,
13     input interrupt,
14     input exception,
15     input eb,
16     input trig,
17
18     input malign,
19     input fault,
20     input invalid_csr,
21
22     input [`ISA__XLEN-1:0] pc_reg,
23     input [`ISA__XLEN-1:0] pc_next,
24
25     output                  debug,
26     output                  halted,
27     output                  halted_ctrl,
28     output                  step_en,
29     output                  resuming,
30     output                  abstract,
31     output                  reg_error,
32     output [`ISA__XLEN-1:0] dpc_out,
33
34     csr_if             csrs,
35     control_signals_if ctrl,
36     debug_if           debug_if
37 );
38     wire progbuf;
39     wire ebreak            = eb && ctrl.write_pc_ne;
40     wire ctrl_halted       = ctrl.mcp_addr == `CONTROL_SIGNALS__HALTED;
41     wire ctrl_resuming     = ctrl.mcp_addr == `CONTROL_SIGNALS__RESUMING;
42     wire instruction_end   = (ctrl.write_pc && !ctrl_resuming) || interrupt;
43     wire trigger_cause     = trig;
44     wire ebreak_cause      = ebreak && `CSR__DCSR_EBREAKM(csrs.DCSR_reg) && !interrupt;
45     wire halt_cause        = debug_if.halt_req;
46     wire step_cause        = step_en && instruction_end;
47     wire quickaccess_cause = `DEBUG__AC_COMMAND(debug_if.command) ==
           `DEBUG__AC_COMMAND_QUICK_ACCESS && debug_if.exec;
48     wire halt_req=trigger_cause||ebreak_cause||halt_cause||step_cause||quickaccess_cause;
49
50     reg debug_reg;
51     reg halted_reg;
52     reg progbuf_reg;
53
```

```
54      always @(posedge clk) begin
55          if (!rst_n) begin
56              debug_reg  <= 1'b0;
57              halted_reg <= 1'b0;
58              progbuf_reg <= 1'b0;
59          end else begin
60              debug_reg   <= debug && !(debug_if.done && quickaccess_cause);
61              halted_reg  <= halted;
62              progbuf_reg <= progbuf;
63          end
64      end
65
66      assign debug  = (debug_reg    || halt_req)                   && !debug_if.resume_req;
67      assign halted = (halted_reg   || (debug_reg && ctrl_halted)) && !ctrl_resuming;
68      assign progbuf= (progbuf_reg  || ctrl.progbuf)               && !debug_if.done;
69
70      assign halted_ctrl = halted && !(ebreak && progbuf_reg);
71      assign step_en     = `CSR__DCSR_STEP(csrs.DCSR_reg);
72      assign resuming    = ctrl_resuming;
73      assign abstract    = debug_if.exec && !progbuf_reg && halted_reg;
74      assign dpc_out     = csrs.DPC_reg;
75
76      reg [           2:0] cause;
77      reg [`ISA__XLEN-1:0] dpc;
78
79      wire aar= `DEBUG__AC_COMMAND(debug_if.command) == `DEBUG__AC_COMMAND_ACCESS_REGISTER;
80      wire aqa= `DEBUG__AC_COMMAND(debug_if.command) == `DEBUG__AC_COMMAND_QUICK_ACCESS;
81      wire aam= `DEBUG__AC_COMMAND(debug_if.command) == `DEBUG__AC_COMMAND_ACCESS_MEMORY;
82
83      assign reg_error      = (aar && ctrl.mcp_addr == `CONTROL_SIGNALS__ABS_REG &&
            `DEBUG__AC_TRANSFER(debug_if.command)) && !(`DEBUG__AC_REG_GPR(debug_if.command)
            || (`DEBUG__AC_REG_CSR(debug_if.command) && !invalid_csr));
84      wire   postexec_error = exception && !ebreak;
85      wire   autohalt_error = aqa && cause != `DEBUG__CAUSE_HALTREQ && halted_reg;
86      wire   bus_error      = aam && (malign || fault);
87
88      assign debug_if.halted = halted;
89      assign debug_if.done=ctrl.abstract_done||(ebreak && progbuf_reg)||postexec_error;
90      assign debug_if.write = ctrl.abstract_write && !(reg_error || bus_error);
91      assign debug_if.bus = bus_error;
92      assign debug_if.haltresume = autohalt_error;
93      assign debug_if.exception = reg_error || postexec_error;
94
95      assign csrs.DCSR_in={csrs.DCSR_reg[31:9],cause,csrs.DCSR_reg[5:4],nmi,csrs.DCSR_reg
            [2:0]};
96      assign csrs.DCSR_write = 1'b1;
97      assign csrs.DPC_in     = dpc;
98      assign csrs.DPC_write  = debug && !debug_reg;
99
100     always_comb begin
101         if (trigger_cause) begin
102             dpc = pc_reg;
103         end else if (ebreak_cause) begin
104             dpc = pc_reg;
105         end else if (halt_cause || quickaccess_cause) begin
106             dpc = ctrl.mcp_addr == `CONTROL_SIGNALS__PROLOGUE ? pc_reg : pc_next;
107         end else if (step_cause) begin
108             dpc = pc_next;
109         end else begin
110             dpc = `ISA__ZERO;
111         end
112     end
113
114     always @(posedge clk) begin;
115         if (!rst_n) begin
116             cause <= 3'd0;
117         end else begin
```

```
118             if (debug && !debug_reg) begin
119                 if (trigger_cause) begin
120                     cause <= `DEBUG__CAUSE_TRIGGER;
121                 end else if (ebreak_cause) begin
122                     cause <= `DEBUG__CAUSE_EBREAK;
123                 end else if (halt_cause || quickaccess_cause) begin
124                     cause <= `DEBUG__CAUSE_HALTREQ;
125                 end else if (step_cause) begin
126                     cause <= `DEBUG__CAUSE_STEP;
127                 end
128             end
129         end
130     end
131
132 endmodule
```

Г Пример дебаговања коришћењем *Eclipse Embedded CDT*



Д Скрипте коришћене за тестирање критичних функционалности

```
force -freeze sim:/testbench/top/dmi_interface/address $1 0
force -freeze sim:/testbench/top/dmi_interface/read 1 0
run 100
noforce sim:/testbench/top/dmi_interface/address
noforce sim:/testbench/top/dmi_interface/read
examine -delta -1 sim:/testbench/top/dmi_interface/data
```

Исечак кода Д.1: dmi_read.do

```
force -freeze sim:/testbench/top/dmi_interface/address $1 0
force -freeze sim:/testbench/top/dmi_interface/data $2 0
force -freeze sim:/testbench/top/dmi_interface/write 1 0
run 100
noforce sim:/testbench/top/dmi_interface/address
noforce sim:/testbench/top/dmi_interface/data
noforce sim:/testbench/top/dmi_interface/write
```

Исечак кода Д.2: dmi_write.do

```
restart -f
run 1600
do dmi_write.do 10 00000001
run 400
do dmi_write.do 10 80000001
do dmi_write.do 10 00000001
do dmi_write.do 20 00508093
do dmi_write.do 4 53729
do dmi_write.do 5 A0000000
do dmi_write.do 17 $1
run 900
echo $1
pause
```

Исечак кода Д.3: prime.do

```
restart -f
run 1600
do dmi_write.do 10 00000001
do dmi_write.do 38 $1
do dmi_write.do 39 $2
run 500
do dmi_read.do  3c
run 500
do dmi_write.do 3c $3
run 500
echo $1
echo $2
pause
```

Исечак кода Д.4: primesb.do

```
do prime.do 002007b1
do prime.do 00200000
do prime.do 00201001
do prime.do 00201100
do prime.do 002107b1
do prime.do 00210000
do prime.do 00211001
do prime.do 00211100
do prime.do 002207b1
do prime.do 00220000
do prime.do 00221001
do prime.do 00221100
do prime.do 002307b1
do prime.do 00230000
do prime.do 00231001
do prime.do 00231100
do prime.do 002407b1
do prime.do 00240000
do prime.do 00241001
do prime.do 00241100
do prime.do 002507b1
do prime.do 00250000
do prime.do 00251001
do prime.do 00251100
do prime.do 002607b1
do prime.do 00260000
do prime.do 00261001
do prime.do 00261100
do prime.do 002707b1
do prime.do 00270000
do prime.do 00271001
do prime.do 00271100
do prime.do 002807b1
do prime.do 00280000
do prime.do 00281001
do prime.do 00281100
do prime.do 002907b1
do prime.do 00290000
do prime.do 00291001
do prime.do 00291100
do prime.do 002A07b1
do prime.do 002A0000
do prime.do 002A1001
do prime.do 002A1100
do prime.do 002B07b1
do prime.do 002B0000
do prime.do 002B1001
do prime.do 002B1100
do prime.do 002C07b1
do prime.do 002C0000
do prime.do 002C1001
do prime.do 002C1100
do prime.do 002D07b1
do prime.do 002D0000
do prime.do 002D1001
do prime.do 002D1100
do prime.do 002E07b1
do prime.do 002E0000
do prime.do 002E1001
do prime.do 002E1100
do prime.do 002F07b1
do prime.do 002F0000
do prime.do 002F1001
do prime.do 002F1100
do prime.do 003007b1
do prime.do 00300000
do prime.do 00301001
```

```
do prime.do 00301100
do prime.do 003107b1
do prime.do 00310000
do prime.do 00311001
do prime.do 00311100
do prime.do 003207b1
do prime.do 00320000
do prime.do 00321001
do prime.do 00321100
do prime.do 003307b1
do prime.do 00330000
do prime.do 00331001
do prime.do 00331100
do prime.do 003407b1
do prime.do 00340000
do prime.do 00341001
do prime.do 00341100
do prime.do 003507b1
do prime.do 00350000
do prime.do 00351001
do prime.do 00351100
do prime.do 003607b1
do prime.do 00360000
do prime.do 00361001
do prime.do 00361100
do prime.do 003707b1
do prime.do 00370000
do prime.do 00371001
do prime.do 00371100
do prime.do 003807b1
do prime.do 00380000
do prime.do 00381001
do prime.do 00381100
do prime.do 003907b1
do prime.do 00390000
do prime.do 00391001
do prime.do 00391100
do prime.do 003A07b1
do prime.do 003A0000
do prime.do 003A1001
do prime.do 003A1100
do prime.do 003B07b1
do prime.do 003B0000
do prime.do 003B1001
do prime.do 003B1100
do prime.do 003C07b1
do prime.do 003C0000
do prime.do 003C1001
do prime.do 003C1100
do prime.do 003D07b1
do prime.do 003D0000
do prime.do 003D1001
do prime.do 003D1100
do prime.do 003E07b1
do prime.do 003E0000
do prime.do 003E1001
do prime.do 003E1100
do prime.do 003F07b1
do prime.do 003F0000
do prime.do 003F1001
do prime.do 003F1100
```

Исечак кода Д.5: test.do

```
do prime.do 02000000
do prime.do 02010000
do prime.do 02080000
do prime.do 02090000
do prime.do 02100000
do prime.do 02110000
do prime.do 02180000
do prime.do 02190000
do prime.do 02200000
do prime.do 02210000
do prime.do 02280000
do prime.do 02290000
do prime.do 02300000
do prime.do 02310000
do prime.do 02380000
do prime.do 02390000
do prime.do 02800000
do prime.do 02810000
do prime.do 02880000
do prime.do 02890000
do prime.do 02900000
do prime.do 02910000
do prime.do 02980000
do prime.do 02990000
do prime.do 02A00000
do prime.do 02A10000
do prime.do 02A80000
do prime.do 02A90000
do prime.do 02B00000
do prime.do 02B10000
do prime.do 02B80000
do prime.do 02B90000
```

Исечак кода Д.6: test2.do

```
do primesb.do 00000000 0 53729
do primesb.do 00010000 0 53729
do primesb.do 00020000 0 53729
do primesb.do 00030000 0 53729
do primesb.do 00040000 0 53729
do primesb.do 00050000 0 53729
do primesb.do 00060000 0 53729
do primesb.do 00070000 0 53729
do primesb.do 00008000 0 53729
do primesb.do 00018000 0 53729
do primesb.do 00028000 0 53729
do primesb.do 00038000 0 53729
do primesb.do 00048000 0 53729
do primesb.do 00058000 0 53729
do primesb.do 00068000 0 53729
do primesb.do 00078000 0 53729
do primesb.do 00100000 0 53729
do primesb.do 00110000 0 53729
do primesb.do 00120000 0 53729
do primesb.do 00130000 0 53729
do primesb.do 00140000 0 53729
do primesb.do 00150000 0 53729
do primesb.do 00160000 0 53729
do primesb.do 00170000 0 53729
do primesb.do 00108000 0 53729
do primesb.do 00118000 0 53729
do primesb.do 00128000 0 53729
do primesb.do 00138000 0 53729
do primesb.do 00148000 0 53729
do primesb.do 00158000 0 53729
do primesb.do 00168000 0 53729
do primesb.do 00178000 0 53729

do primesb.do 00158000 1 53729
do primesb.do 00158000 A0000000 53729
```

Исечак кода Д.7: testsb.do

Ђ Конфигурациони фајлови за тестове *RISC-V* организације

```
adapter speed 1000
adapter driver jlink
transport select jtag

set _CHIPNAME riscv
jtag newtap $_CHIPNAME cpu -irlen 5 -expected-id 0x00537291

set _TARGETNAME $_CHIPNAME.cpu
target create $_TARGETNAME.0 riscv -chain-position $_TARGETNAME
$_TARGETNAME.0 configure -work-area-phys 0x00000000 -work-area-size 0x10000

gdb_report_data_abort enable
gdb_report_register_access_error enable

riscv expose_csrs 2288

init

halt
```

Исечак кода Ђ.1: custom.cfg

```
import targets

class customHart(targets.Hart):
    xlen = 32
    ram = 0x00000000
    ram_size = 64 * 1024
    instruction_hardware_breakpoint_count = 4
    misa = 0x40000100
    bad_address = 0xA0000000
    reset_vectors = [0x00000000]

class custom(targets.Target):
    harts = [customHart()]
    timeout_sec = 20
    supports_clint_mtime = False
    test_semihosting = False
    support_manual_hwbp = True
    skip_tests = [''Sv32Test'',''SemihostingFileio'',''EtriggerTest'',''IcountTest'',''
        TriggerDmode'',''DownloadTest'']
    support_hasel = False
```

Исечак кода Ђ.2: custom.py

```
ENTRY(_start)
MEMORY{SRAM (rwx) : ORIGIN = 0x00000000, LENGTH = 64K}
SECTIONS{
    . = 0x0000;
    PROVIDE (__executable_start = SEGMENT_START(''text-segment'', 0x00000));
    PROVIDE (__stack_pointer =  ORIGIN(SRAM) + LENGTH(SRAM) - 0x4);
    .ivt :
    {
        *(.ivt .ivt*)
    } > SRAM
    .text :
    {
        *(.startup .startup*)
        *(.text.startup .text.startup*)
        *(.text .text*)
    } > SRAM
    .rodata :
    {
        *(.rodata .rodata*)
    } > SRAM
    .data :
    {
        __DATA_BEGIN__ = .;
        *(.data .data*)
    } > SRAM
    .sdata :
    {
        __SDATA_BEGIN__ = .;
        *(.srodata .srodata*)
        *(.sdata .sdata*)
    } > SRAM
    _edata = .; PROVIDE (edata = .);
    . = .;
    . = ALIGN(32 / 8);
    __bss_start = .;
    .sbss :
    {
        *(.sbss .sbss*)
        *(.scommon)
    } > SRAM
    .bss :
    {
        *(.bss .bss*)
        *(COMMON)
        . = ALIGN(. != 0 ? 32 / 8 : 1);
    } > SRAM
    . = ALIGN(32 / 8);
    . = SEGMENT_START(''ldata-segment'', .);
    . = ALIGN(32 / 8);
    __BSS_END__ = .;
    __bss_end = .;
    __global_pointer$ = MIN(__SDATA_BEGIN__ + 0x800, MAX(__DATA_BEGIN__ + 0x800,
        __BSS_END__ - 0x800));
    __malloc_start = .;
    . = . + 512;
    _end = .; PROVIDE (end = .);
    .comment 0:
    {
        *(.comment)
    }
    .gnu.build.attributes :
    {
        *(.gnu.build.attributes .gnu.build.attributes*)
    }}
```

Исечак кода Ђ.3: custom.lds

Е Запис извршавања тестова *RISC-V* организације

```
lazar@ssVM:debug (master *)$ ./gdbserver.py ./targets/custom/custom.py
Using $misa from hart definition: 0x40000100
[CeaseStepiTest] Starting > logs/20230910-030451-custom-CeaseStepiTest.log
[CeaseStepiTest] not_applicable in 0.00s
[CheckMisa] Starting > logs/20230910-030451-custom-CheckMisa.log
[CheckMisa] pass in 4.87s
[CustomRegisterTest] Starting > logs/20230910-030456-custom-CustomRegisterTest.log
[CustomRegisterTest] not_applicable in 0.00s
[DebugBreakpoint] Starting > logs/20230910-030456-custom-DebugBreakpoint.log
[DebugBreakpoint] pass in 14.09s
[DebugChangeString] Starting > logs/20230910-030510-custom-DebugChangeString.log
[DebugChangeString] pass in 11.64s
[DebugCompareSections] Starting > logs/20230910-030522-custom-DebugCompareSections.log
[DebugCompareSections] pass in 5.31s
[DebugExit] Starting > logs/20230910-030527-custom-DebugExit.log
[DebugExit] pass in 5.88s
[DebugFunctionCall] Starting > logs/20230910-030533-custom-DebugFunctionCall.log
[DebugFunctionCall] pass in 13.74s
[DebugSymbols] Starting > logs/20230910-030546-custom-DebugSymbols.log
[DebugSymbols] pass in 10.10s
[DebugTurbostep] Starting > logs/20230910-030557-custom-DebugTurbostep.log
[DebugTurbostep] pass in 27.79s
[DisconnectTest] Starting > logs/20230910-030624-custom-DisconnectTest.log
[DisconnectTest] pass in 26.92s
[DownloadTest] Starting > logs/20230910-030651-custom-DownloadTest.log
[DownloadTest] not_applicable in 0.00s
[EbreakTest] Starting > logs/20230910-030651-custom-EbreakTest.log
[EbreakTest] pass in 11.29s
[EtriggerTest] Starting > logs/20230910-030703-custom-EtriggerTest.log
[EtriggerTest] not_applicable in 0.00s
[FreeRtosTest] Starting > logs/20230910-030703-custom-FreeRtosTest.log
[FreeRtosTest] not_applicable in 0.00s
[Hwbp1] Starting > logs/20230910-030703-custom-Hwbp1.log
[Hwbp1] pass in 13.10s
[Hwbp2] Starting > logs/20230910-030716-custom-Hwbp2.log
[Hwbp2] pass in 16.67s
[HwbpManual] Starting > logs/20230910-030732-custom-HwbpManual.log
[HwbpManual] not_applicable in 7.91s
[IcountTest] Starting > logs/20230910-030740-custom-IcountTest.log
[IcountTest] not_applicable in 0.00s
[InfoTest] Starting > logs/20230910-030740-custom-InfoTest.log
[InfoTest] pass in 2.08s
[InstantChangePc] Starting > logs/20230910-030742-custom-InstantChangePc.log
[InstantChangePc] pass in 6.90s
[InstantHaltTest] Starting > logs/20230910-030749-custom-InstantHaltTest.log
[InstantHaltTest] pass in 3.03s
[InterruptTest] Starting > logs/20230910-030752-custom-InterruptTest.log
[InterruptTest] not_applicable in 0.00s
[ItriggerTest] Starting > logs/20230910-030752-custom-ItriggerTest.log
[ItriggerTest] not_applicable in 0.00s
[JumpHbreak] Starting > logs/20230910-030752-custom-JumpHbreak.log
[JumpHbreak] pass in 8.33s
[MemTest16] Starting > logs/20230910-030801-custom-MemTest16.log
[MemTest16] pass in 3.73s
```

```
[MemTest32] Starting > logs/20230910-030804-custom-MemTest32.log
[MemTest32] pass in 3.13s
[MemTest64] Starting > logs/20230910-030807-custom-MemTest64.log
[MemTest64] pass in 3.34s
[MemTest8] Starting > logs/20230910-030811-custom-MemTest8.log
[MemTest8] pass in 3.32s
[MemTestBlock0] Starting > logs/20230910-030814-custom-MemTestBlock0.log
[MemTestBlock0] pass in 5.95s
[MemTestBlock1] Starting > logs/20230910-030820-custom-MemTestBlock1.log
[MemTestBlock1] pass in 5.79s
[MemTestBlock2] Starting > logs/20230910-030826-custom-MemTestBlock2.log
[MemTestBlock2] pass in 5.71s
[MemTestBlockReadInvalid] Starting > logs/20230910-030832-custom-MemTestBlockReadInvalid.
    log
[MemTestBlockReadInvalid] not_applicable in 0.00s
[MemTestReadInvalid] Starting > logs/20230910-030832-custom-MemTestReadInvalid.log
[MemTestReadInvalid] pass in 6.38s
[MemorySampleMixed] Starting > logs/20230910-030838-custom-MemorySampleMixed.log
[MemorySampleMixed] pass in 14.11s
[MemorySampleSingle] Starting > logs/20230910-030852-custom-MemorySampleSingle.log
[MemorySampleSingle] pass in 13.07s
[MulticoreRegTest] Starting > logs/20230910-030905-custom-MulticoreRegTest.log
[MulticoreRegTest] not_applicable in 0.00s
[MulticoreRtosSwitchActiveHartTest] Starting > logs/20230910-030905-custom-
    MulticoreRtosSwitchActiveHartTest.log
[MulticoreRtosSwitchActiveHartTest] not_applicable in 0.00s
[MulticoreRunAllHaltOne] Starting > logs/20230910-030905-custom-MulticoreRunAllHaltOne.
    log
[MulticoreRunAllHaltOne] not_applicable in 0.00s
[PrivChange] Starting > logs/20230910-030905-custom-PrivChange.log
[PrivChange] not_applicable in 3.19s
[PrivRw] Starting > logs/20230910-030908-custom-PrivRw.log
[PrivRw] pass in 11.15s
[ProgramHwWatchpoint] Starting > logs/20230910-030919-custom-ProgramHwWatchpoint.log
[ProgramHwWatchpoint] pass in 50.85s
[ProgramSwWatchpoint] Starting > logs/20230910-031010-custom-ProgramSwWatchpoint.log
[ProgramSwWatchpoint] pass in 120.51s
[Registers] Starting > logs/20230910-031211-custom-Registers.log
[Registers] pass in 35.04s
[RepeatReadTest] Starting > logs/20230910-031246-custom-RepeatReadTest.log
[RepeatReadTest] not_applicable in 0.00s
[Semihosting] Starting > logs/20230910-031246-custom-Semihosting.log
[Semihosting] not_applicable in 0.00s
[SemihostingFileio] Starting > logs/20230910-031246-custom-SemihostingFileio.log
[SemihostingFileio] not_applicable in 0.00s
[SimpleF18Test] Starting > logs/20230910-031246-custom-SimpleF18Test.log
[SimpleF18Test] pass in 12.81s
[SimpleNoExistTest] Starting > logs/20230910-031259-custom-SimpleNoExistTest.log
[SimpleNoExistTest] pass in 1.87s
[SimpleS0Test] Starting > logs/20230910-031301-custom-SimpleS0Test.log
[SimpleS0Test] pass in 7.64s
[SimpleS1Test] Starting > logs/20230910-031308-custom-SimpleS1Test.log
[SimpleS1Test] pass in 10.13s
[SimpleT0Test] Starting > logs/20230910-031318-custom-SimpleT0Test.log
[SimpleT0Test] pass in 10.15s
[SimpleT1Test] Starting > logs/20230910-031328-custom-SimpleT1Test.log
[SimpleT1Test] pass in 10.35s
[SimpleV13Test] Starting > logs/20230910-031339-custom-SimpleV13Test.log
[SimpleV13Test] pass in 6.18s
[SmpSimultaneousRunHalt] Starting > logs/20230910-031345-custom-SmpSimultaneousRunHalt.
    log
[SmpSimultaneousRunHalt] not_applicable in 0.00s
[StepTest] Starting > logs/20230910-031345-custom-StepTest.log
[StepTest] pass in 21.93s
[StepThread2Test] Starting > logs/20230910-031407-custom-StepThread2Test.log
[StepThread2Test] not_applicable in 0.00s
[Sv32Test] Starting > logs/20230910-031407-custom-Sv32Test.log
```

```
[Sv32Test] not_applicable in 0.00s
[Sv39Test] Starting > logs/20230910-031407-custom-Sv39Test.log
[Sv39Test] not_applicable in 0.00s
[Sv48Test] Starting > logs/20230910-031407-custom-Sv48Test.log
[Sv48Test] not_applicable in 0.00s
[TooManyHwbp] Starting > logs/20230910-031407-custom-TooManyHwbp.log
[TooManyHwbp] pass in 17.00s
[TriggerDmode] Starting > logs/20230910-031424-custom-TriggerDmode.log
[TriggerDmode] not_applicable in 0.00s
[TriggerExecuteInstant] Starting > logs/20230910-031424-custom-TriggerExecuteInstant.log
[TriggerExecuteInstant] pass in 6.58s
[TriggerLoadAddressInstant] Starting > logs/20230910-031431-custom-
    TriggerLoadAddressInstant.log
[TriggerLoadAddressInstant] pass in 17.32s
[TriggerStoreAddressInstant] Starting > logs/20230910-031448-custom-
    TriggerStoreAddressInstant.log
[TriggerStoreAddressInstant] pass in 13.12s
[UnavailableCycleTest] Starting > logs/20230910-031501-custom-UnavailableCycleTest.log
[UnavailableCycleTest] not_applicable in 0.00s
[UnavailableMultiTest] Starting > logs/20230910-031501-custom-UnavailableMultiTest.log
[UnavailableMultiTest] not_applicable in 0.00s
[UnavailableRunTest] Starting > logs/20230910-031501-custom-UnavailableRunTest.log
[UnavailableRunTest] not_applicable in 0.00s
[UserInterrupt] Starting > logs/20230910-031501-custom-UserInterrupt.log
[UserInterrupt] pass in 12.54s
[VectorTest] Starting > logs/20230910-031514-custom-VectorTest.log
[VectorTest] not_applicable in 0.00s
[WriteCsrs] Starting > logs/20230910-031514-custom-WriteCsrs.log
[WriteCsrs] pass in 11.74s
[WriteGprs] Starting > logs/20230910-031525-custom-WriteGprs.log
[WriteGprs] pass in 24.15s
::::::::::::::::::::::::::[ ran 72 tests in 658s ]::::::::::::::::::::::::::
27 tests returned not_applicable
45 tests returned pass
```