ПРИЛОЖЕНИЕ НА TL-VERILOG ЗА МОДЕЛИРАНЕ НА КОМПОНЕНТИ НА МИКРОПРОЦЕСОРНИ АРХИТЕКТУРИ

Илиан Върбов, Валентина Кукенска, Петър Минев, Матьо Динев

Технически университет – Габрово
{ivarbov, vally, pminev, mdinev}@tugab.bg

APPLICATION OF TL-VERILOG FOR MODELLING COMPONENTS OF MICROPROCESSOR ARCHITECTURES

Ilian Varbov, Valentina Kukenska, Petar Minev, Matyo Dinev

Technical University of Gabrovo
{ivarbov, vally, pminev, mdinev}@tugab.bg

Abstract

This report presents techniques for modelling components of processor architectures at a high level of abstraction, using the hardware description language TL-Verilog. The microarchitectural requirements have been defined, according to which a model of the RISC-V processor architecture has been developed. The model has been validated through simulation and visualization of its operation. Freely available online tools for open-source development, including online Makerchip IDE, were used for a model implementation and validation. By modelling the individual components in the RISC-V microarchitecture, the emerging extension of the Verilog language is presented in an accessible way.

Keywords: TL-Verilog; RISC-V; HDL; modelling; microprocessor

ВЪВЕДЕНИЕ

TL-Verilog, или Transaction-Level Verilog, е иновативен език за описание на хардуер (HDL), който предлага няколко предимства при проектирането и моделирането на цифрови схеми и системи. Той е разширение на традиционния език за описание на хардуер Verilog HDL, който е широко използван в полупроводниковата индустрия. В TL-Verilog се въвежда по-високо ниво на абстракция и по-интуитивен подход към описанието на цифрови системи. Подходът за моделиране на ниво транзакции, опростеният синтаксис и поддръжката на паралелизъм са особено подходящи за съвременни, сложни хардуерни проекти, включително микропроцесори, цифрови сигнални процесори и високоскоростни интерфейсни схеми [2].

В този доклад е представено приложението на TL-Verilog за моделиране на компонентите в процесорна микроархитектура, реализираща изпълнението на инструкции от системата RISC-V. Системата инструкции RISC-V е петата в серията от RISC архитектури, разработени в Калифорнийския университет в Бъркли. Голямата ѝ популярност се дължи на това, че е свободна за използване архитектура с отворен код [3].

ИЗЛОЖЕНИЕ

А. Микроархитектура на процесор, изпълняващ инструкции от системата RISC-V

Блоковата схема с моделираните компонентите е показана на фиг. 1 [1]. Тя се състои от:

- Логика на програмния брояч (PC)

Програмният брояч се използва като указател за следващата инструкция, която процесорът трябва да изпълни. В повечето случаи инструкциите се изпълняват последователно, което означава, че стойността на програмния брояч се увеличава с единица при всеки процесорен цикъл. При инструкции за преход PC се променя, така че да сочи целевата инструкция, която следва да се изпълни.

- Памет за инструкции (IMem)

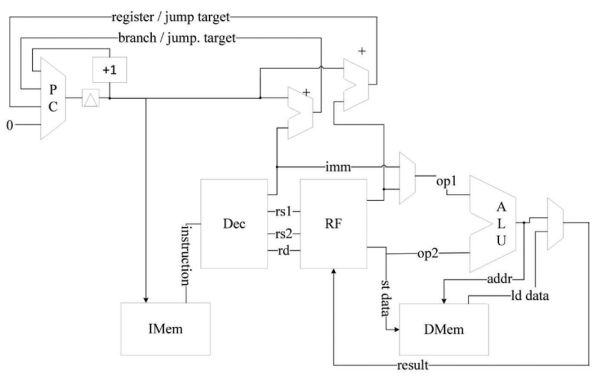
Паметта за инструкции съдържа програмата, подлежаща на изпълнение. Програмният брояч адресира паметта за инструкции, като посочва адреса на инструкцията, която трябва да се прочете от IMem (познато още като извличане на инструкция).

- Логика за декодиране (DEC)

За да се разбере вида на операцията, заложена в инструкцията и нейните операнди (регистри и константи), е необходимо декодиране или интерпретиране на прочетения двоичен код от паметта за инструкции.

- Регистров файл (RF)

Регистровият файл е малко локално пространство, състоящо се от набор от регистри, за съхранение на стойности, с които програмата работи активно. Декодирането на инструкцията показва с кои регистри следва да се работи. По конкретно, след декодирането е ясно кои са регистрите от регистровия файл, от които да се прочетат входните данни и кой е целевият регистър, в който да се запише резултатът след извършване на операцията.



Фиг. 1. *Блокова диаграма на RISC-V CPU*

- Аритметично-логическо устройство (ALU)

В зависимост от операцията, посочена в инструкцията, аритметично-логическото устройство извършва събиране, изваждане, умножение, побитово преместване и др.

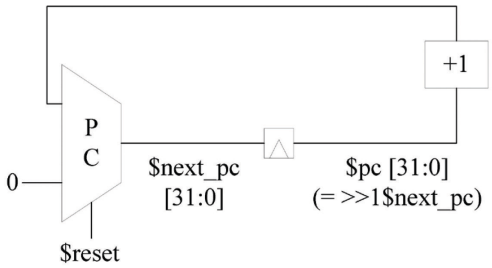
- Памет за данните (DMem)

В RISC архитектурите за достъп до паметта за данни се използват само два типа инструкции – load и store. Инструкциите от тип load четат от паметта и записват прочетеното в регистър, а – store четат от регистър и записват в паметта.

Б. Моделиране на компонентите от RISC-V микроархитектурата

Този доклад е фокусиран върху възможностите на TL-Verilog за моделиране на основните компоненти от микроархитектурата на RISC-V процесор [5]. В него не са засегнати схемите за прекъсвания, системните таймери, контролерите за вход/изход (I/O) и др.

Първоначално програмният брояч сочи адреса на първата инструкция в IMem. Инструкциите са с дължина 4 байта. Въпреки че на фиг. 2 увеличаването на PC е изобразено с "+1" (плюс една инструкция), реално PC нараства с 4. Така, при нормална работа най-младшите два бита на PC винаги са нула.



Фиг. 2. *Схема на PC*

В програмния код на модела, по-долу, са добавени и случаите, когато инструкцията е инструкция за условен или безусловен преход.

```
$pc[31:0] = >>1$next_pc;
$next_pc[31:0] = $reset ? 0 :
        $taken_br ? $br_targ_pc :
        $ins_jal   ? $br_targ_pc :
        $ins_jalr  ? $jalr_targ_pc :
        $pc + 4;
```

Моделиране на програмния брояч

При моделиране на логиката за декодиране на инструкции са взети предвид 31 от основните инструкции от базовия набор на RISC-V (RV32I). За да се определи видът на операцията в тези инструкции и нейните операнди е необходимо, 32-битовият двоичен код на инструкцията да се раздели на отделни полета според формата на инструкцията.

Инструкциите в RISC-V съдържат следните полета:

- **opcode** (код на операцията): Това поле осигурява общо класифициране на инструкцията и определя кои от останалите полета са необходими и как са подредени или кодирани в останалите битове.
- **function field** (поле за функция) (funct3/funct7): Указва точната функция, изпълнявана от инструкцията, ако тя не е напълно определена от кода на операцията (opcode).
- **rsrc1/rsrc2**: Това са индекси (0-31), които идентифицират регистър(ите) в регистровия файл, съдържащ(и) входните операнди, върху които се извършва операцията.
- **rdata**: Това е индекс (0-31), посочващ регистъра, в който се записва резултатът от инструкцията.
- **immediate** (непосредствена стойност): Константна стойност, съдържаща се в 32 битовия двоичен код на самата инструкция. Тази стойност може да послужи като отместване за индексиране в паметта или като стойност, върху която да се извърши операция (вместо стойността на регистъра, индексиран от полето rsrc2).

Инструкциите, чийто операнди са само регистри (и нямат непосредствена стойност) са от тип R. Тяхното кодиране предоставя общата структура на полетата и за останалите типове инструкции (фиг. 3). Преди да бъдат извлечени стойностите на полетата от кода на инструкцията, е необходимо да се определи нейният тип. В модела, това се реализира по следния начин:

```
$u_ins = $instr[6:2] ==? 5'b0x101;
$s_ins = $instr[6:2] ==? 5'b0100x;
$b_ins = $instr[6:2] ==? 5'b11000;
$j_ins = $instr[6:2] ==? 5'b11011;
$r_ins = $instr[6:2] == 5'b01011 ||
        $instr[6:2] == 5'b01100 ||
        $instr[6:2] == 5'b01110;
$i_ins = $instr[6:2] ==? 5'b0000x ||
        $instr[6:2] ==? 5'b001x0 ||
        $instr[6:2] == 5'b11001;
```

Моделиране на логиката за декодиране на инструкции

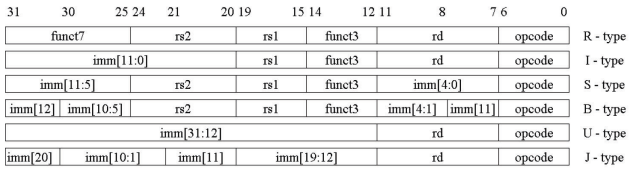
За да са валидни инструкциите в RV32I [6], битовете $instr[1:0] трябва да са 2'b11. Прието е, че всички инструкции са валидни и проверката на тези два бита се пренебрегва.

Програмният код за разделяне на инструкцията на отделни полета е представен по-долу:

```
$rsrc2[4:0] = $instr[24:20];
$rsrc1[4:0] = $instr[19:15];
$funct3[2:0] = $instr[14:12];
$rdata[4:0] = $instr[11:7];
$opcode[6:0] = $instr[6:0];
```

Стойността на immediate се извлича малко по-сложно. Тя се състои от битове от различни полета в зависимост от типа на инструкцията. Например, стойността на immediate за инструкции от тип I се формира от 21 копия на бит 31 на инструкцията, следвани от inst[30:20].

```
$imm[31:0] =
        $i_ins ? { {21{$instr[31]}},
$instr[30:20] } :
        $s_ins ? { {21{$instr[31]}},
$instr[30:25],   $instr[11:7]} :
        $b_ins ? { {20{$instr[31]}}, $instr[7],
$instr[30:25], $instr[11:8], 1'b0} :
        $u_ins ? { $instr[31:12], 12'b0} :
        $j_ins ? { {12{$instr[31]}},
$instr[19:12], $instr[20], $instr[30:21], 1'b0} :
        32'b0; // Default
```



Фиг. 3. *Формат на инструкциите в RISC-V, според техния тип*

За да се определи конкретната операция в инструкцията, следва да се декодират полетата opcode, instr[30] и funct3 (фиг. 4).

funct7[5]	funct3	opcode	instr
		0110111	LUT
		0010111	AUIPC
		1101111	JAL
	000	1100111	JALR
	000	1100011	BEQ
	001	1100011	BNE
	100	1100011	BLT
	101	1100011	BGE
	110	1100011	BLTU
	111	1100011	BGEU
	000	0000011	LB
	001	0000011	LH
	010	0000011	LW
	100	0000011	LBU
	101	0000011	LHU
	000	0100011	SB
	001	0100011	SH
	010	0100011	SW
	000	0010011	ADDI

funct7[5]	funct3	opcode	instr
	010	0010011	SLTI
	011	0010011	SLTIU
	100	0010011	XORI
	110	0010011	ORI
	111	0010011	ANDI
0	001	0010011	SLLI
0	101	0010011	SRLI
1	101	0010011	SRAI
0	000	0110011	ADDI
1	000	0110011	SUB
0	001	0110011	SLL
0	010	0110011	SLT
0	011	0110011	SLTU
0	100	0110011	XOR
0	101	0110011	SRL
1	101	0110011	SRA
0	110	0110011	OR
0	111	0110011	AND

Фиг. 4. *Определяне на вида на операцията според операционния код и полето funct3*

Следва TL-Verilog кодът, чрез който се определя видът на операцията в дадена инструкция:

```
$dec_bits[10:0] = {$instr[30],$funct3,$opcode};
$ins_beq = $dec_bits ==? 11'bx_000_1100011;
$ins_bne = $dec_bits ==? 11'bx_001_1100011;
$ins_blt = $dec_bits ==? 11'bx_100_1100011;
$ins_bge = $dec_bits ==? 11'bx_101_1100011;
$ins_bltu = $dec_bits ==? 11'bx_110_1100011;
$ins_bgeu = $dec_bits ==? 11'bx_111_1100011;
$ins_addi = $dec_bits ==? 11'bx_000_0010011;
$ins_add = $dec_bits ==? 11'b0_000_0110011;
$ins_lui = $dec_bits ==? 11'bx_xxx_0110111;
$ins_auipc = $dec_bits ==? 11'bx_xxx_0010111;
$ins_jal = $dec_bits ==? 11'bx_xxx_1101111;
$ins_jalr = $dec_bits ==? 11'bx_000_1100111;
$ins_slti = $dec_bits ==? 11'bx_010_0010011;
$ins_sltiu = $dec_bits ==? 11'bx_011_0010011;
$ins_xori = $dec_bits ==? 11'bx_100_0010011;
$ins_ori = $dec_bits ==? 11'bx_110_0010011;
$ins_andi = $dec_bits ==? 11'bx_111_0010011;
$ins_slli = $dec_bits ==? 11'b0_001_0010011;
$ins_srli = $dec_bits ==? 11'b0_101_0010011;
$ins_srai = $dec_bits ==? 11'b1_101_0010011;
$ins_sub = $dec_bits ==? 11'b1_000_0110011;
$ins_sll = $dec_bits ==? 11'b0_001_0110011;
$ins_slt = $dec_bits ==? 11'b0_010_0110011;
$ins_sltu = $dec_bits ==? 11'b0_011_0110011;
$ins_xor = $dec_bits ==? 11'b0_100_0110011;
$ins_srl = $dec_bits ==? 11'b0_101_0110011;
$ins_sra = $dec_bits ==? 11'b1_101_0110011;
$ins_or = $dec_bits ==? 11'b0_110_0110011;
$ins_and = $dec_bits ==? 11'b0_111_0110011;
$ins_load = $dec_bits ==? 11'bx_xxx_0000011;
```

Логика за определяне на вида на операцията.

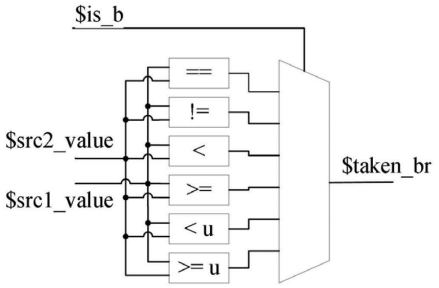
В АЛУ множество операционни устройства работят паралелно. Така, едновременно, за всяка аритметично-логическа операция, се изчислява резултатът, който би произвела. Кой от тези резултати да се използва се определя на базата на вида на операцията в инструкцията.

```
$result[31:0]= $ins_addi  ? $src1_value + $imm :
        $ins_add   ? $src1_value + $src2_value :
        $ins_andi  ? $src1_value & $imm :
        $ins_ori   ? $src1_value | $imm :
        $ins_xori  ? $src1_value ^ $imm :
        $ins_slli  ? $src1_value << $imm[5:0] :
        $ins_srli  ? $src1_value >> $imm[5:0] :
        $ins_and   ? $src1_value & $src2_value :
        $ins_or    ? $src1_value | $src2_value :
        $ins_xor   ? $src1_value ^ $src2_value :
        $ins_sub   ? $src1_value - $src2_value :
        $ins_sll   ? $src1_value << $src2_value[4:0] :
        $ins_srl   ? $src1_value >> $src2_value[4:0] :
        $ins_sltu  ? $sltu_rslt :
        $ins_sltiu ? $sltiu_rslt :
        $ins_lui   ? {$imm[31:12],12'b0} :
        $ins_auipc ? $pc + $imm :
        $ins_jal   ? $pc + 32'd4 :
        $ins_jalr  ? $pc + 32'd4 :
        $ins_slt ? (($src1_value[31] == $src2_value[31]) ? $sltu_rslt : {31'b0, $src1_value[31]}) :
        $ins_slti  ? (($src1_value[31] == $imm[31]) ? $sltiu_rslt : {31'b0, $src1_value[31]}) :
        $ins_sra   ? $sra_rslt[31:0] :
        $ins_srai  ? $srai_rslt[31:0] :
        $ins_load  ? $src1_value + $imm :
        $s_ins     ? $src1_value + $imm :
    32'b0;
```

Моделиране на аритметично-логическото устройство (АЛУ)

Инструкциите за преход са два вида – условен и безусловен. При инструкция за условен преход целевият адрес, заложен в този вид инструкция се зарежда в програмния брояч, само ако условието е изпълнено (вярно). В условието се сравняват стойностите на двата входни регистъра ($src1_value и $src2_value).

Подобно на логиката на АЛУ и тук едновременно се извършват всички възможни сравнения и се избира само един от получените резултати, според точния вид на инструкцията за условен преход (фиг. 5). Дали преходът в програмата на посочения в инструкцията адрес трябва да се осъществи зависи от това дали $taken_br е нула или единица.



Фиг. 5. *Логика за определяне на резултата от условието в инструкциите за преход*

Следва програмният код, реализиращ логиката от фиг. 5.

```
$taken_br = $ins_beq ?  ($src1_value == $src2_value) :
        $ins_bne   ?  ($src1_value != $src2_value) :
        $ins_blt ?  (($src1_value < $src2_value) ^ ($src1_value[31] != $src2_value[31])) :
        $ins_bge   ?  (($src1_value >= $src2_value) ^ ($src1_value[31] != $src2_value[31])) :
        $ins_bltu  ?  ($src1_value < $src2_value) :
        $ins_bgeu  ?  ($src1_value >= $src2_value) :
    1'b0;
```

Моделиране на необходимата логика за инструкциите за преход

Целевият адрес, сочен от инструкцията за преход, с който трябва да се обнови PC, ако условието е изпълнено, се определя като полето immediate (непосредствената стойност, като относително байтово отместване) се сумира с текущата стойност на PC. Следва кодът за пресмятане на целевия адрес за инструкциите за условен и безусловен преход.

```
$br_targ_pc[31:0] = $pc + $imm;
$jalr_targ_pc[31:0] = $src1_value + $imm;
```

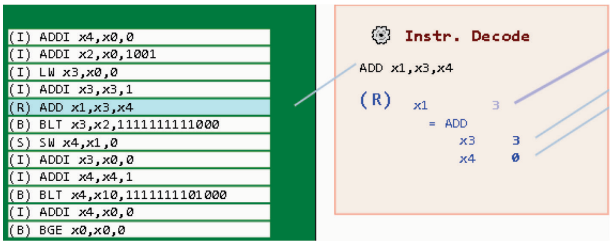
B. Валидация на моделираната RISC-V микроархитектура

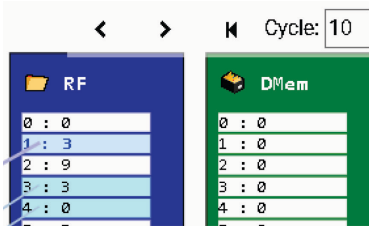
Валидирането на моделираната RISC-V микроархитектура е изпълнено в средата за разработка Makerchip. За целта е разработена следната тестова програма:

```
m4_asm(ADDI, x4, x0, 0)
m4_asm(ADDI, x2, x0, 1001)
m4_asm(LW, x3, x0, 0)
m4_asm(ADDI, x3, x3, 1)
m4_asm(ADD, x1, x3, x4)
m4_asm(BLT, x3, x2, 1111111111000)
m4_asm(SW, x4, x1, 0 )
m4_asm(ADDI, x3, x0, 0)
m4_asm(ADDI, x4, x4, 1)
m4_asm(BLT, x4, x10, 1111111101000)
m4_asm(ADDI, x4, x0, 0)
m4_asm(BGE, x0, x0, 0)
m4_asm_end()
m4_define(['M4_MAX_CYC'], 315)
```

С представената тестова програма се цели да се провери коректното изпълнение на аритметично-логическите инструкции, инструкциите за преход и инструкциите за достъп до паметта Load и Store. С програмата се реализира брояч до 99. За целта се използват два вложени един в друг цикъла. Във вложения цикъл един от регистрите нараства от 0 до 9. При следващата итерация този регистър се нулира и същевременно във външния цикъл друг регистър нараства с единица. За външния брояч се използва регистър x4, а за вътрешния – регистър x3. Числото 9, което е граничната стойност, до която достигат двата брояча се зарежда първоначално в регистър x2. При всяка итерация на вътрешния (младшия) брояч, резултатът от сумирането на x3 и x4 се записва в регистър x1. Когато младшият брояч достигне 9, той се нулира, а в старшия брояч се добавя 1 и стойността на регистър x1 се записва в паметта на адрес, указан в регистър x4.

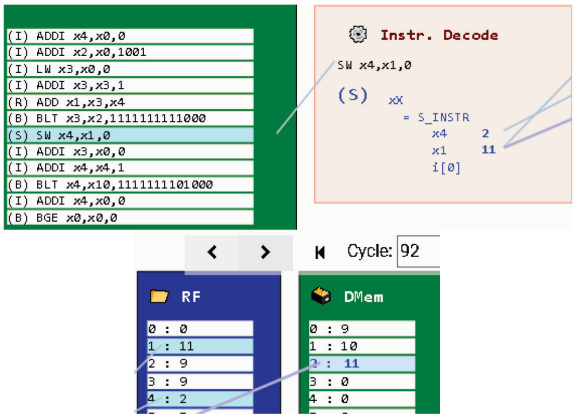
На фиг. 6 се вижда, че при достигане до десетия процесорен цикъл, инструкция ADD записва сбора на регистри x3(3) и x4(0) в регистър x1.





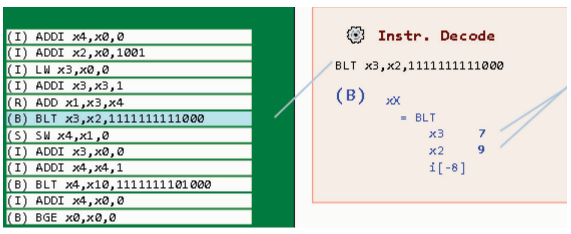
Фиг. 6. *Визуализация на резултата от симулацията на модела в процесорен цикъл 10*

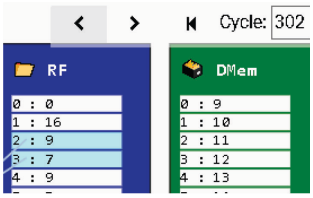
След всеки пълен цикъл на младшия брояч се изпълнява инструкция SW, която записва стойността на регистър x1(11) в паметта на адрес 2(x4). Това е представено на фиг. 7. При достигане до процесорен цикъл 92 са завършени 3 пълни цикъла на младшия брояч.



Фиг. 7. *Визуализация на резултата от симулацията на модела в процесорен цикъл 92*

На фиг. 8 се вижда, как се изпълнява инструкцията за условен преход BLT. Стойностите на регистрите x2 и x3 се сравняват и тъй като стойността на x3(7) е по-малка от стойността на x2(9), то програмата ще продължи не със следващата инструкция, а ще се върне с 2 инструкции назад, според отместването „111111111100“, записано като непосредствена стойност в инструкцията BLT.





Фиг. 8. *Визуализация на резултата от симулацията на модела в процесорен цикъл 302*

ЗАКЛЮЧЕНИЕ

В доклада бяха представени модели на основни компоненти от микроархитектурата на процесор, способен да изпълнява инструкции от системата RISC-V. Видно е, че TL-Verilog осигурява удобни езикови конструкции за моделиране на работата на микропроцесори. Освен това създадените модели са лесни за разбиране, което прави този език подходящ за използване в учебния процес по дисциплини, свързани с проектиране на цифров хардуер.

За валидацията на изградения модел на RISC-V микроархитектура е използвана средата за разработка Makerchip и вградените в нея средства за визуална симулация на работата на TL-Verilog моделите. За нуждите на валидацията е съставена тестова програма на асемблерен език за RISC-V архитектура. В програмата са включени основни инструкции от различен тип, за да се тества тяхното коректно изпълнение.

Визуалната валидация, която предлага TL-Verilog и Makerchip значително опростява процеса на тестване и симулация, като вместо стандартните времедиаграми, работата на моделите се представя, чрез подходящо изградени анимирани визуални елементи, които се променят според промяната на сигналите в моделите. За големи и сложни модели, каквито са моделите на микропроцесори, това е особено полезно, тъй като позволява лесно да се проследи цялостната работа на даден модел, без да е необходимо да се наблюдават десетки или стотици цифрови сигнали.

В TL-Verilog не са предвидени езикови средства за изграждане на памети. Поради тази причина за регистров файл и памет за инструкции и данни са използвани готови макроси [4], в които посочените памети се реализират, чрез SystemVerilog. Като бъдещо развитие на предложената разработка се предвижда създаване на собствени модели на памети със средствата на Verilog HDL, които да могат лесно да се вграждат и използват в TL-Verilog кода.

БЛАГОДАРНОСТИ

Този доклад е подготвен и осъществен като част от проект № 2209Е „Виртуална лаборатория за обучение по проектиране на цифров хардуер“, финансиран от средствата по бюджета за научни изследвания на Технически университет – Габрово.

REFERENCE

[1] Hennessy J. and D. Patterson, Computer Architecture, Sixth Edition: A Quantitative Approach (6th. ed.). San Francisco, CA: Morgan Kaufmann Publishers, 2017.

[2] Hoover S. F. and A. Salman, "Top-Down Transaction-Level Design with TL-Verilog", CoRR, vol. abs/1811.01780, 2018, [online] Available: http://arxiv.org/abs/1811.01780.

[3] Höller R., D. Haselberger, D. Ballek, P. Rössler, M. Krapfenbauer and M. Linauer, “Open-Source RISC-V Processor IP Cores for FPGAs — Overview and Evaluation,” 2019 8th Mediterranean Conference on Embedded Computing (MECO), Budva, Montenegro, 2019, pp. 1-6, doi: 10.1109/MECO.2019.8760205.

[4] Hoover S., “Stevehoover - Overview,” GitHub, https://github.com/stevehoover/ (accessed Jun. 12, 2023).

[5] Hoover S., “Building a RISC-V CPU Core”, Course from LinuxFoundationX, https://www.edx.org/learn/design/the-linux-foundation-building-a-risc-v-cpu-core (accessed Nov. 12, 2022).

[6] Waterman A., Y. Lee, D. A. Patterson, and K. Asanović, “The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA,” Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley, May 2011.