Pēteris Račinskis
pr20015

Mikrokontrolera izstrāde – kopsavilkums

1. Nostādītie mērķi, sasniegtie rezultāti

Šogad digitālo iekārtu projektēšanas kursa ietvaros tika jau pašā pirmajā nodarbībā nolemts, ka kurss tiks strukturēts kā praktisko darbu secība, kas kopā novedīs līdz gala produktam – uz Spartan6 FPGA strādājošam centrālajam procesoram RISC-V arhitektūrā, sasaistītam ar kāda veida ārēju fizisku saskarni. Procesora izstrāde tika sākta pakāpeniski, katru nedēļu ieviešot pa kādam funkcionālajam blokam un tālāku darbību plānojot atkarībā no sasniegtajiem rezultātiem un gūtās pieredzes. Precīzas prasības gala projektam tika noteiktas ap to pašu laiku, kad vientakts procesora projekts bija gandrīz pabeigts, un tās ir sekojošas:

- vientakts RISC-V procesora kodols, kas spēj izpildīt RV32I instrukciju kopas apakškopu – visas loģikas/aritmētikas, kontroles un atmiņas piekļuves instrukcijas;
- patvaļīgi izvēlētu ārējās saskarnes moduļu izstrāde, kas ļautu novērtēt procesora darbību uz fiziskas ierīces;
- demonstrācijām nepieciešamo RISC-V asemblera programmas kodu izstrāde un izpilde.

Ņemot vērā uz Digilent Anvyl prototipēšanas plates pieejamo ievades/izvades resursu klāstu, prasību izpildei tika izveidots mikrokontroleris ar sekojošajiem izpildelementiem:

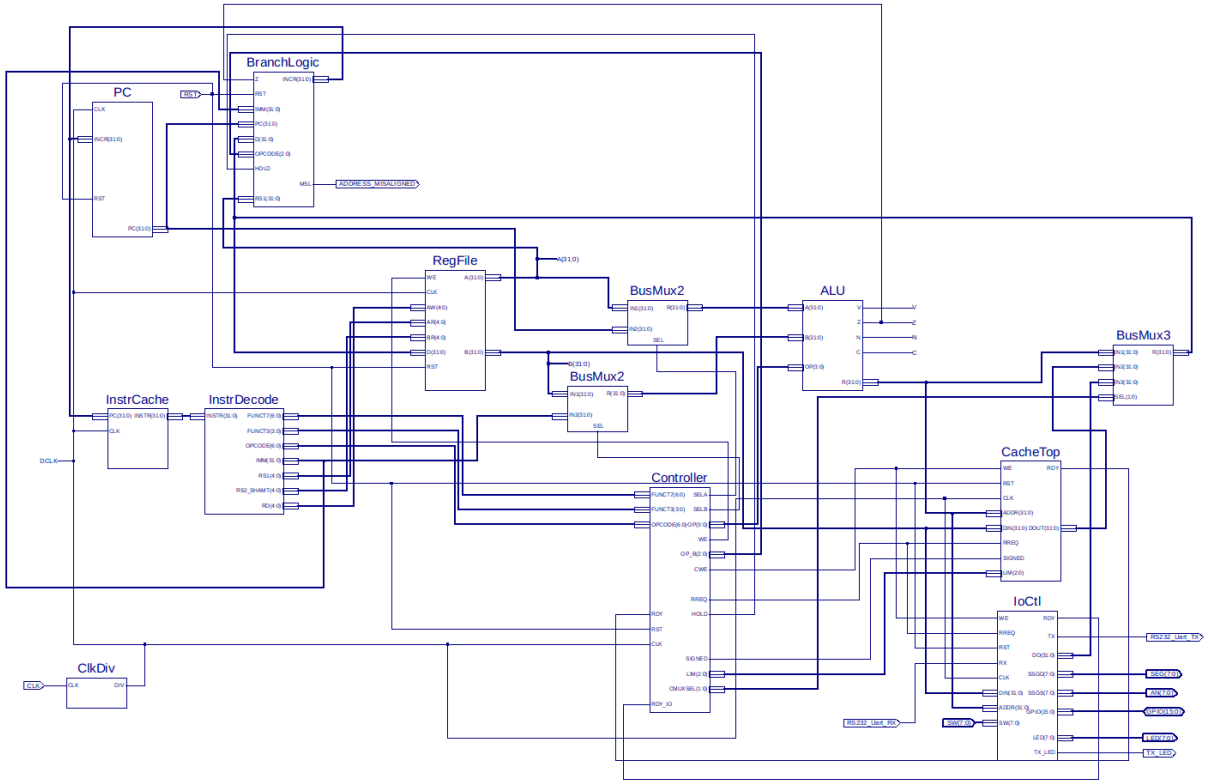
- ievades kontaktiem, kas savienoti ar 2-pozīciju slēdžiem uz plates;
- izvades kontaktiem, kas savienoti ar spīddiodēm;
- konfigurējamiem ievades/izvades kontaktiem, kas savienoti ar *breadboard* un JA PMOD konektoru;
- sešu ciparu 7-segmentu displeju;
- UART moduli, kas savienots ar platē integrēto FTDI USB-UART pārveidotāju

Mikrokontrolera uzbūves aspekti no lietotāja/programmētāja puses sīkāk aprakstīti dokumentā "*Datasheet*".

2. Risinājums, tā izstrādes process

Kaut gan praktiskā darbība tika sākta ar reģistru faila izstrādi, par ko nedaudz runāts zemāk, viens no svarīgākajiem un, varētu tiekt, pirmais lēmums FPGA projekta izstrādē, ir izvēle starp pieejamajām izstrādes valodām – Verilog, VHDL un zīmēšanu shēmtehnikā. VHDL apsvērta netika, jo autoram ir iepriekšēja pieredze ar

Verilog un tā ir arī HDL, kas tiek pasniegta kursa ietvaros. Shēmtehnika nav tik universāla kā HDL un ir piesaistīta ISE izstrādes rīkam, taču padara pārskatāmākus tādus moduļus, kas pamatā sastāv tikai no savienojumiem starp apakšmoduļiem. Tāpēc nolemts augšējā līmeņa moduli “top.sch” izstrādāt shēmtehnikā, bet visus pārējos - Verilog.



Att. 1. Centrālā procesora augšējā līmeņa modulis shēmtehnikā.

Sākumā gan šis lēmums bija paredzēts kā pagaidu risinājums – ar domu augšējo līmeni pārnest uz Verilog – taču šī izmaiņa netika veikta, jo konstatēts, ka, par spīti lielajam moduļu un savieonojumu skaitam kā arī mazliet haotiskajam izkārtojumam, shēma tik un tā palikusi samērā pārskatāma. Šāda veida shēma arī uzskatāmi parāda visus galvenos CPU apakšmoduļus, par kuriem atsevišķi un aptuveni hronoloģiskā secībā rakstīts tālāk.

2.1. Reģistru fails

Pirmais modulis, kas izstrādāts projekta ietvaros, bija reģistru fails – procesora kodola iekšējā atmiņa. Pirmā šī moduļa versija uzrastīta vēl pirms izveidots CPU projekts, un sākotnēji veidota kā vienkāršs reģistru masīvs ar sinhronām rakstīšanas un lasīšanas operācijām. Taču tālākās projekta fāzēs, veicot integrāciju ar citiem moduļiem, konstatēts, ka pareizai reģistru faila darbībai nepieciešams datu nolasīšanu veikt asinhroni, t.i., izmantojot tikai kombinacionālu loģiku, bez pulksteņa signāla. Tāpēc reģistru faila izmēri ir ierobežoti – to nevar realizēt kā FPGA integrēto BRAM atmiņas bloku, jo tie atbalsta tikai sinhronas datu piekļuves operācijas.

2.2. ALU

Nākamais solis procesā bijis ALU (Aritmētikas un Loģikas operāciju moduļa) izstrāde. Kaut gan virspusēji varētu šķist, ka šeit slēpjas kas sarežģīts, no tehniskās realizācijas viedokļa ALU ir gaužām vienkāršs – ar Verilog *case* konstrukciju pēc *opcode* signāla nodalīta kombinacionālas loģikas izteiksmju grupa, kuru vērtība piešķirta izvade signālam atkarībā no diviem ievadiem. Pašas operācijas iespējams viegli pievienot vai noņemt, tālāk attīstot projektu, taču svarīgs lēmums jāpieņem jau pašā sākumā – kuri signāli var parādīties kurā no ALU ieejām, lai samazinātu nepieciešamo atsevišķu izteiksmju un līdz ar to telpā apjomīgu rēķināšanas elementu skaitu. Jau šajā etapā ticis nolemts, kuras no RV32I instrukcijām paredzēts realizēt, un attiecīgi signāli sagrupēti pie A un B ieejām. Pie katras pievienots multipleksors. A ieejā tas izvēlas starp reģistru faila A izvadu un programmas skaitītāja vērtību, B – starp B izvadu un instrukcijas *immediate*.

2.3. Instrukciju atmiņa, dekoderis

Lai varētu sākt vispirms ar individuālu instrukciju izpildi, bet pēc tam jau ar vienkāršām kompilētām programmām (izmantojot ārējā testa failā ģenerētu instrukciju vai programmas skaitītāju), nepieciešams pievienot instrukciju dekoderi un instrukciju atmiņu. Pati atmiņa ir gaužām vienkāršs *read-only* RAM bloks, savukārt dekoderis – kombinacionālā loģika. Dekoderī galveno izaicinājumu sastāda *immediate* izdalīšana no instrukcijas, jo tā var parādīties dažādās vietās un ar dažādi sajauktiem bitiem. *Sign-extend* operācija, kas citās instrukciju kopas arhitektūrās var būt sarežģītāka problēma, gan RISC-V ir elementāra – vērtības msb vienmēr tiek izmantots kā zīmes indikators, t.i., *sign-extend* operācija vienkārši tiek veikta vienmēr.

Lielākā darba daļa šajā projekta etapā droši vien aizgāja tieši programmas kompilācijas procesa izstrādē un automatizācijā. Lai radītu HDL kompilatoram saprotamu instrukciju faila formātu, izmantots gnu-as kompilators, python skripts un Makefile, lai izveidotu čaulas komandu, kas secīgi:

- ģenerē izpildfailu 32-bitu RISC-V arhitektūrai no asemblera koda;
- izvelk no izpildfaila tikai pašas programmas instrukcijas;
- pārveido bināro failu par tekstu ar vērtībām “1”/”0”.

Šādu failu tad tālāk var izmantot HDL kompilācijas procesā, lai piešķirtu instrukciju atmiņas masīvam tā sākotnējo vērtību kā konstanti.

Paralēli dekodera izstrādei sākta arī procesora kontrolera izstrāde, kas gan sīkāk aprakstīta zemāk.

2.4. Programmas skaitītājs, kontroles pārneses loģika

Paralēli instrukciju atmiņai izstrādāts arī programmas skaitītājs un kontroles pārneses loģikas bloks, kas pēc tam gan vairākas reizes iteratīvi papildināts un

pārveidots. Pats programmas skaitītājs ir vienkāršs reģistrs, kam katrā pulksteņa taktī tiek ierakstīta jauna vertība (vai, reset signāla gadījumā – 0). Šo vērtību rēķina kontroles pārneses loģikas modulis, kas faktiski ir vēl viens – vienkāršāks – ALU. Parasti tas vienkārši rēķina tipisko PC + 4 izteiksmi, taču *Branch* vai *Jump* instrukciju gadījumā – PC + IMM, vai PC + IMM + Rs1. Praktisku apsvērumu dēļ *Jump* instrukcijās tad PC + 4 operāciju (atgriešanās adresi) rēķina galvenais ALU, jo šī vērtība nonāk nevis programmas skaitītājā, bet reģistru failā.

2.5. "Ārējā" atmiņa

Saskarne ar ārējiem atmiņas blokiem procesoros tiek realizēta ar kešatmiņas starpniecību, un tāda sākotnēji bijus paredzēta arī šajā projektā. Izvedots kešatmiņas modulis, kas sastāv no kešatmiņas kontrolera, pašas kešatmiņas un simulētas "ārējās" atmiņas, kas uzvedas analogi tipiskai ārējai SRAM mikroshēmai. Kešatmiņa tikusi realizēta ar īpatnēju *shift-register* metodi, jo darba autoram nebija ienācis prātā, ka var izmantot daļu no 32-bitu adreses kā "nejaušu" indeksu nelielai lokālai atmiņai – kolīzijas notiks, bet tas praksē daudz neatšķirsies no secīgas datu rakstīšanas pārbīdes reģistrā, jo varbūtiski tiks izmantoti visi reģistri.

Tomēr izveidotais modulis balstīts uz pieņēmumu, ka tiks strādāts ar 32-bitu vārdiem. Vēlāk pārveidojot visu atmiņas sistēmu uz baitu adresāciju konstatēts, ka, lai kešatmiņa būtu lietojama situācijās, kad iespējamas signed un unsigned atmiņas piekļuves dažādos garumos, būtu nepieciešams to papildināt ar sarežģītu pēcapstrādes loģiku lasīšanai un priekšapstrādes loģiku rakstīšanai (un dzēšanai), tāpēc beigās šis modulis no projekta vienkārši izņemts.

Līdz ar to atmiņas funkcionalitāti beigās nodrošina tikai simulētā "ārējā" atmiņa, kas īstenībā ir vienkārši pašā FPGA integrēts BRAM bloks ar sinhronu lasīšanu un rakstīšanu.

2.6. Kontroleris

Visu pārējo moduļu darbību kopā sasaista kontroleris, kura izstrāde sākta kopā ar dekodera izstrādi. Tā pamatuzdevums ir pārveidot dekodēto instrukciju saturu komandās izpildelementiem kā ALU, kontroles pārneses loģikai, reģistru failam, u.c. To lielākoties nodrošina kombinacionālās loģikas izteiksmes. Taču ieviests arī sinhrons galīgais automāts atmiņas piekļuves operāciju kontrolei, lai procesors būtu spējīgs tikt galā ar nenoteikta ilguma aizturēm šādās operācijās. Tas nodrošina programmas skaitītāja apturēšanu un visu datu maģistrāļu noturēšanu nepieciešamajā stāvoklī visā datu piekļuves instrukcijas izpildes garumā. Praktisku realizācijas iemeslu dēļ šis automāts realizēts ar 180 grādu fāzes nobīdi, t.i. – stāvokļa pārejas un reģistru vērtību nomaiņas notiek pie pulksteņa krītošās frontes.

2.7. Ievades-izvades kontroleris, integrācija aparatūrā

Pēdējais galvenais modulis, kas pievienots procesoram, ir ievades-izvades kontroleris. Tas satur reģistrus ievades-izvades kontaktu vērtību nolasīšanai un kontrolei, 7-segmentu displeja kontroleri, UART Tx un Rx kontrolerus. Tā kā speciālu I/O instrukciju RISC-V arhitektūrā nav, ievades-izvades kontroleris būvēts tā, lai tā saskarne ar pārējo procesoru būtu identiska atmiņas blokam. To, vai instrukciju izpildīs atmiņa vai I/O, nosaka adreses 31. bits. Ievades un izvades funkciju darbība sīkāk aprakstīta dokumentā "*Datasheet*".

Piešķirot projektam reālus ierobežojumus, konstatēti izpildes laika pārkāpumi. Signāliem izplatoties caur daudziem kombinacionālās loģikas slāņiem un gariem savienojumiem visa čipa apjomā, rodas aiztures, kas summējoties pārkāpj 100MHz frekvencei noteiktās robežas. Kaut gan praksē šķiet, ka arī ar šādiem pārkāpumiem procesors strādā (vismaz tās procesora daļas, ko šādos apstākļos bija izdevos notestēt uz aparatūras), tie noved pie ļoti ilgiem *place & route* procesa soļiem – 20+ minūtes, kompilējot projektu. Tāpēc ieviests vienkāršs frekvences dalītājs, kas dala ienākošo 100MHz frekvenci ar 10. Līdz ar to iegūts 10MHz procesors.

3. Secinājumi

Tīri apjoma ziņā šis ir viens no vērienīgākajiem projektiem, ar ko darba autoram sanācis saskarties studiju gaitās. Līdz ar to gūta neaizvietojama praktiska pieredze HDL izstrādē. Iespēja pašam izstrādāt savu centrālo procesoru, šķiet, sniedz ieskatu dauzos aparatūras aspektos, par kuriem, strādājot ar augsti abstrahētām programmēšanas valodām, bieži pat neiedomājamies. Turklāt beidzot pārvarēta konceptuālā barjera – priekštats par procesora darbību vairs nav kaut kas abstrakts un aptuvens, bet reāls, konkrēts un detalizēts.

Protams, kā jau vienmēr, strādājot pie ierobežotiem resursiem un termiņiem, jāatzīst, ka ir lietas, ko būtu iespējams darīt labāk. Agrīnos procesora izstrādes etapos pieņemti lēmumi par elementu un signālu savstarpējiem savienojumiem un izkārtojumu, ko nevienā brīdī nav šķitis vērts mainīt – vieglāk taču ir vienkārši pielāgoties – kas līdz projekta beigām jau sakrājušies ievērojamos sarežģījumos. Kā vienu piemēru varētu minēt neveiksmīgo kešatmiņas dizainu. Izveidots, lai strādātu pie vienmērīga platuma atmiņas un ar sarežģītu shift-register uzbūvi, turklāt abstrahēts kā papildus atmiņas bloks, tas nav bijis padevīgs pārveidošanai. Vēl viena problēma ir instrukciju adreses padošana no kontroles pārneses loģikas bloka, nevis programmas skaitītāja reģistra. Sākumā šķitis, ka tas tikai atvieglo dažādo procesora elementu sinhronizāciju, taču pievienojot reālus fiziskus ierobežojumus konstatēts, ka šādi tiek radīti garāki loģikas ceļi – kas, iespējams, ir galvenais iemesls nepieciešamībai samazināt procesora takts frekvenci.