Univerza v Ljubljani
Fakulteta za računalništvo in informatiko
Fakulteta za matematiko in fiziko

Fedja Beader

Implementacija igre Tetris v vezju FPGA

diplomsko delo
univerzitetni študijski program prve stopnje
računalništvo in matematika

Mentor: izr. prof. dr. Patricio Bulić

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V okviru diplomske naloge implementirajte popularno računalniško igro Tetris v vezju FPGA Xilinx Artix-7. Za uporabniški vmesnik uporabite stikala na dotik, 7-segmentni prikazovalnik ter v FPGA implementirajte vmesnik VGA. ZA implementacijo igre uporabite jezik VHDL, pri čemer ne uporabite procesorjev.

Kazalo

Seznam uporabljenih kratic

ASIC	application-specific integrated circuit
BDF	glyph bitmap distribution format
BRAM	block RAM
CLB	configurable logic block
EDID	extended display identification data
DPI	dots per inch
DVI	digital visual interface
FPGA	field-programmable gate array
HDL	hardware description language
HDMI	high-definition multimedia interface
LUT	lookup table
RAM	random-access memory
ROM	read-only memory
SCL	serial clock line
SDA	serial data line
VGA	video graphics array
VHDL	VHSIC HDL
VHSIC	very high speed integrated circuit

Povzetek

Naslov: Implementacija igre Tetris v vezju FPGA

V okviru diplomske naloge je bila razvita različica popularne računalniške igre Tetris. Igra je bila v celoti implementirana v jeziku za opis strojne opreme VHDL s poudarkom na modularni zgradbi. Razvit je bil modul za operacije na aktivnem elementu, ki implementira vodoravne premike, obračanje, padanje in hitre spuste. V podporo temu sta bila razvita modula za psevdonaključni izbor naslednjega elementa in modul za odstranjevanje polnih vrstic. Oba sta bila z modulom za aktivni element povezana preko razvitega modula za igralno polje. Izris na zaslon je bil narejen v modulu za izris znakov in v modulu za izrisni cevovod, ki je hkrati vrhnji strojno-neodvisni modul. Za komunikacijo z igralcem so bili razviti krmilnik zaslona VGA, modul za zaznavanje pritiskov na tipke in krmilnik 7-delnega prikazovalnika. Ti so bili z igro povezani preko razvitega strojno-specifičnega vrhnjega modula za razvojno ploščo Digilent Nexys4.

Ključne besede: tetris, tetrimino, strojna izvedba igre, končni avtomati, UNI-VGA, BDF, VGA, VHDL, FPGA.

Abstract

Title: Implementation of the Tetris game for FPGA

In this thesis, a version of the popular computer game Tetris was developed. The game was fully implemented in the VHDL hardware design language with a focus on modular design. A module was developed for operations on the active element that implements horizontal movement, rotations, falling and fast descent. Other game logic was implemented in the module for pseudorandom selection of next element and the module for full row removal. Both of these were connected to the developed module for the playing field. On screen display was implemented in the developed module for display of characters and the module for pipelined rendering, the later of which represents the top level hardware-independent module. Several hardware-dependent modules were developed for interacting with the player: a 7-segment display controller, a VGA controller and a module for tactile buttons. These modules were all connected with the game via a hardware-dependent top level module for the Digilent Nexys4 development board.

Keywords: tetris, tetrimino, hardware game implementation, finite state automata, UNI-VGA, BDF, VGA, VHDL, FPGA.

Poglavje 1

Uvod

Zakaj razvoj v digitalni logiki? V primeru izdelave tetrisa kot programa na mehkem jedru (angl. soft core) brez namenskih pospeševalnih enot, bi izris na navideznem zaslonu moral biti končan pred začetkom prenosa na zaslon. Posledično bi potrebovali relativno veliko pomnilnika glede na količino, razpoložljivo v ciljnih FPGA čipih. Za zaslon ločljivosti 640 stolpcev krat 480 vrstic točk, kjer ima vsak od treh barvnih kanalov (rdeča, zelena in modra) na ciljni plošči ločljivost 4 bite, bi tako potrebovali:

$$640 \times 480 \times (4 \times 3)/8 = 460800 \text{ bajtov} \tag{1.1}$$

Rezultat izračuna (1.1) nam pove, da bi za izris potrebovali 450 KiB (en kibibajt je enak 1024 bajtov) pomnilnika, kar je precej več kot velikost posameznega vgrajenega blokovnega pomnilnika (angl. BRAM) na FPGA čipu xc7a100t, ki znaša 36864 bitov [4]. Nastalo težavo bi brez razvoja namenskih pospeševalnih enot (tako da bi razvito igro lahko uporabili na cenovno ugodnih mikrokrmilnikih), lahko odpravili na več načinov:

- Prvi pristop bi bil povezovanje več blokovnih pomnilnikov skupaj, kar lahko s pomočjo proizvajalčeve programske opreme tudi enostavno naredimo. Natanko koliko jih moramo povezati skupaj nam določa enačba (1.2).

$$460800 \times 8/36864 = 100 \tag{1.2}$$

Vendar razvita igra zavzame le majhen del uporabljenega čipa, tako da bi se nepotrebno omejili na večje a mnogo dražje modele za vgradnjo v končni izdelek, saj poddružine 15t, 35t in 50t vsebujejo premalo blokovnih pomnilnikov. V primeru finančne nesmiselnosti proizvodnje namenskega integriranega vezja (angl. ASIC), bi z drugačnim pristopom morda lahko uporabili najmanjši in s tem tudi najcenejši FPGA iz družine Artix-7.

- Drugi pristop bi zahteval priklop namenskega pomnilnega vezja (angl. RAM). Poleg tega bi bilo potrebno razviti krmilnik za uporabljeno pomnilniško tehnologijo ali uporabiti že dokončano jedro intelektualne lastnine (angl. IP core)[1]. Za ta pristop se nismo odločili, saj je bilo namensko pomnilno vezje Cellular RAM, ki je prisotno na razvojni plošči Nexys4, umaknjeno iz proizvodnje[2]. Prav tako nobeno pomnilno vezje tega tipa ni na voljo pri velikih distributerjih elektronskih komponent, kar bi otežilo njegovo integracijo skupaj z FPGA ali ASIC v namensko tiskano vezje.

- Tretji pristop bi zahteval manjšo ločljivost in manj možnih barv. Konkretno bi se morali za uporabo čipa xc7a15t omejiti na manjšo, npr. standardno ločljivost QVGA [13], ki znaša 320×240. Pri tej ločljivosti in minimalni barvni globini 16, kolikor rabimo za tetris, bi z upoštevanjem podprtih širin podatkovnega vodila porabili 12 blokovnih pomnilnikov [4].

Glede na hitrost procesorja je povsem možno, da celotnega izrisa ne bi bilo mogoče opraviti v času med dvema izrisoma (več o tem v poglavju o vhodno-izhodnih enotah). V tem primeru bi morali uporabiti tehniko dvojnega medpomnenja (angl. double buffering), pri kateri bi v pomnilniku imeli dva navidezna zaslona. Tako bi program izrisoval tetris na prvega medtem ko bi krmilnik zaslona prenašal sliko iz drugega. Ko bi krmilnik zaslona in

[1] https://en.wikipedia.org/wiki/Semiconductor_intellectual_property_core
[2] https://blog.digilentinc.com/tag/cellular-ram/

naš program končala izrisovanje, bi se ta dva navidezna zaslona zamenjala in proces izrisovanja bi se začel od začetka.

Kaj so slabosti takšnega razvoja? Kot prvo bi izpostavili mnogo počasnejši razvoj in dejstvo, da se po prenosu iz FPGA na ASIC igre ne bi več dalo spreminjati. Poleg tega je izdelava ASIC velik začetni strošek [6], za povrnitev katerega bi bilo potrebno prodati veliko število igralnih aparatov. V primeru, da bi se odločili v slednje vgraditi FPGA namesto izdelave namenskih čipov, bi tudi naredili izgubo glede na ceno FPGA v primerjavi s ceno primernega mikrokrmilnika in dodatnega pomnilniškega čipa za izris slike.

Tukaj smo ubrali pot izdelave igre v digitalni logiki oziroma raziskali, kako se programska oprema prenese v strojno. Glavni cilj pri tem je bil raziskati, kako se programska oprema prenese v digitalno logiko.

Poglavje 2

Razvojni komplet Digilent Nexys4

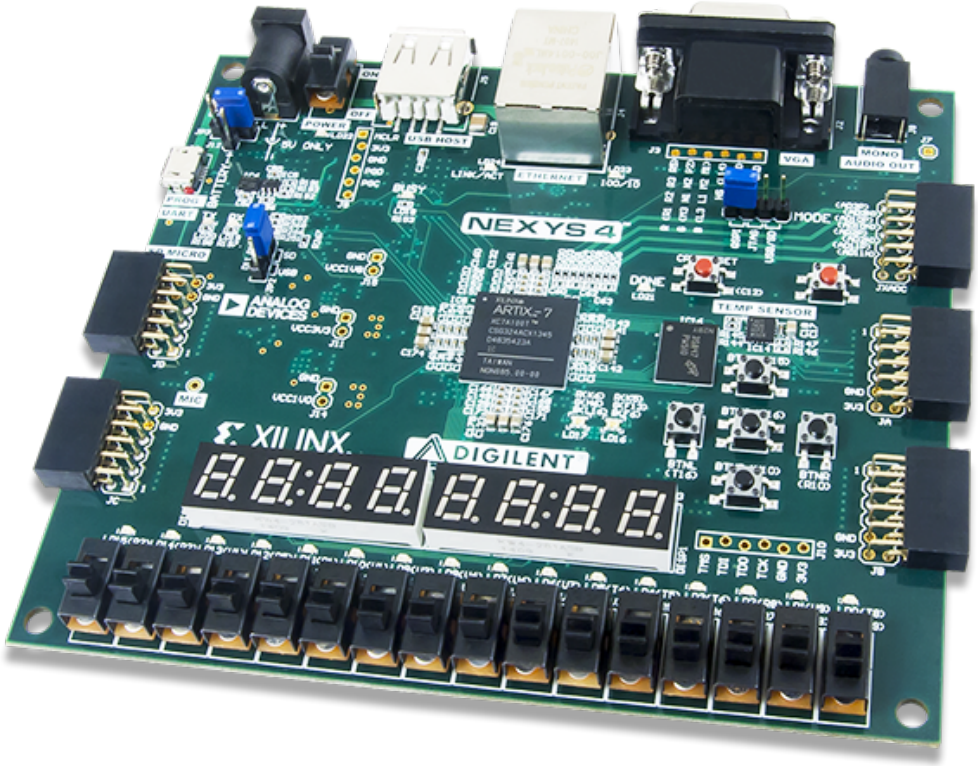
Za implementacijo igre smo uporabili razvojni komplet Nexys4 [7] revizije B podjetja Digilent, prikazan na sliki 2.1. Na tej plošči smo za komunikacijo z igralcem uporabili:

- izhod VGA za povezavo z zaslonom,
- 7-segmentni zaslon za izpis doseženih točk,
- 5 smernih tipk za upravljanje s padajočim elementom tetrisa,
- tipko "CPU RESET" za ponastavitev sistema.

V naslednjih razdelkih bomo na kratko opisali uporabljeno opremo, nakar bo v naslednjem poglavju sledil opis uporabljenih komponent na razvojni plošči in razvitih krmilnikov zanje.

2.1 Integrirano vezje FPGA

Na sredini plošče se nahaja čip FPGA podjetja Xilinx z oznako xc7a100t-1csg324. Čip je sestavljen iz konfigurabilnih logičnih blokov (angl. CLB)



Slika 2.1: Razvojna plošča Digilent Nexys4 [7]

in povezav med njimi. Dalje je vsak CLB sestavljen iz dveh rezin, od katerih vsaka vsebuje štiri poizvedbene tabele, 8 flip-flopov, prenosno logiko (za učinkovito implementacijo števcev) in multiplekserje med njimi. Vsak CLB je z drugimi povezan s pomočjo lokalne stikalne in globalne usmerjevalne matrike [5].

Vendar niti vsebina poizvedbenih tabel niti poti med njimi niso trajno določene, temveč jih lahko s pomočjo konfiguracijske datoteke, imenovane "bitstream", dokaj svobodno povezujemo. Vsebina te datoteke tako določa opis strojne opreme, realizirane z naštetimi elementi. Omeniti je vredno, da pred konfiguriranjem in po izgubi napajanja strojna oprema, ki smo jo definirali, ne obstaja. S pomočjo teh čipov torej lahko skoraj zastonj preizkušamo nove izvedbe digitalnih vezij.

2.2 Jezik VHDL

Z rastjo kompleksnosti digitalnih vezij so snovalci le-teh potrebovali način opisa vezij na višjem nivoju, ki jih ne bi vezal na točno določeno tehnologijo integriranih vezij. Kot odgovor na to so bili ustvarjeni jeziki za opis strojne opreme [9]. Eden od teh jezikov je tudi VHDL oziroma jezik za opis zelo hitrih integriranih vezij. V njem strojno opremo opisujemo v modulih oziroma entitetah, katere zopet v drugih modulih povezujemo med sabo. Na koncu pridemo do korenskega oziroma vrhnjega modula, katerega vhodno-izhodne vodnike povežemo z nogami (angl. pins) integriranega vezja. Vodnike lahko tudi združujemo glede na njihovo funkcijo, s čimer dobimo vodila (angl. bus).

Od tu naprej bomo za oboje uporabljali besedo "signal", razen ko bomo želeli posebej poudariti število vodnikov.

2.3 Razvojni orodji Xilinx ISE in Vivado

Z razvojem jezikov za opis strojne opreme in čipov FPGA so se razvijala tudi orodja okoli njih. Vendar je delo z čipi FPGA zaradi varovanja poslovnih skrivnosti podprto le s strani proizvajalčevih orodij. Eno od teh orodij, ki podpirajo uporabljen čip, je Xilinx ISE. To orodje je sprva tudi bilo uporabljeno za razvoj tetrisa, ki predstavlja praktični del te diplome. Tekom izdelave tetrisa pa smo ugotovili, da bi zaradi boljše abstrakcije bilo dobro uporabiti konstrukte iz standarda VHDL 2008. Standarda VHDL 2008 ne podpira nobena različica orodja ISE, zato je bil izveden prehod na novejše orodje Vivado [10]. Prednost novejšega orodja ni le podpora za novejši standard VHDL, temveč se odraža tudi pri nekoliko manjši porabi logičnih gradnikov na ciljnem FPGA čipu.

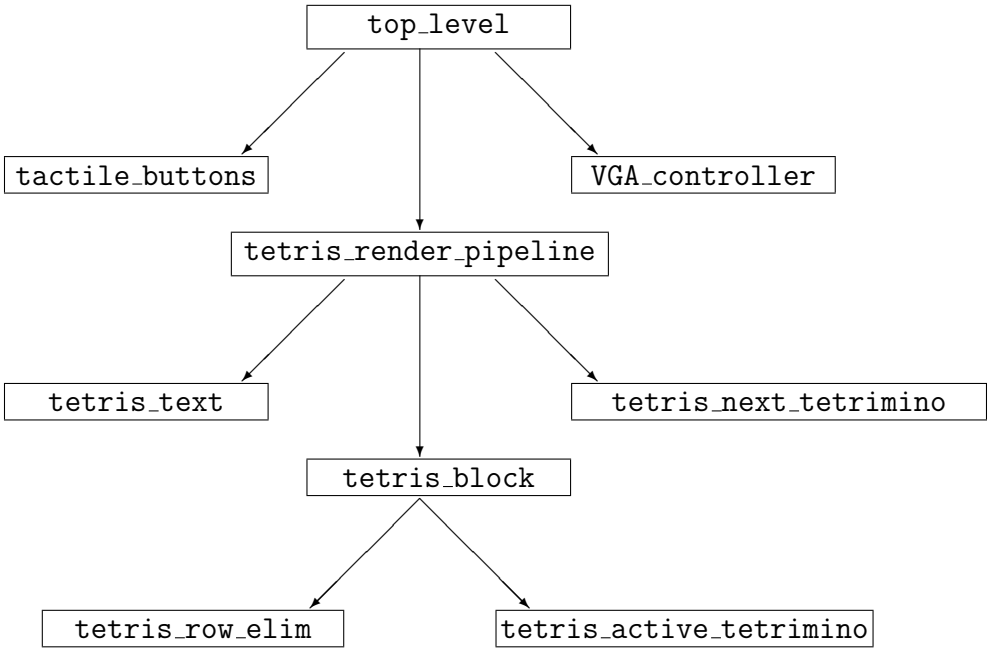
Poleg krovnega modula moramo za uspešno sintezo ustvariti tudi datoteko XDC (angl. Xilinx Design Constraints), ki signale krovnega modula preslika v noge čipa FPGA, definira napetosti na njih in za potrebe časovne analize oziroma verifikacije razvitega sistema definira hitrost sistemske ure. To smo pod imenom `pin_map.xdc` definirali v mapi `board/Nexys4`.

Poglavje 3

Vhodno-izhodne enote in podporna strojna oprema

Razviti projekt je razdeljen na tri dele. Prvi del predstavlja jedro projekta oziroma logiko, ki implementira osnovno funkcionalnost igre. Drugi del predstavlja strojno-specifično logiko. Ta je bila razvita, da smo lahko tetris pognali na uporabljeni razvojni plošči. Omenjena logika skrbi za povezovanje krovnega modula igre z uporabljeno strojno opremo. Med te sodijo krmilnik akcij in logika za branje tipk, krmilnik 7-segmentnega prikazovalnika in krmilnik VGA. Tretji del predstavlja razvita logika, ki ni del niti jedra igre niti podpore za posamezno razvojno ploščo. Vseeno pa je nujno potrebna za izvedbo obeh. Ta del predstavljajo razni števci, pomnilniški elementi, krmilnik tip in nadgradnje standarda.

V naslednjih razdelkih bomo tretji del opisali le okvirno, drugega pa bolj podrobno. Prvemu delu smo namenili svoje poglavje, kjer bo opisan zelo podrobno. Medsebojne odvisnosti glavnih razvitih entitet v projektu smo ponazorili na sliki 3.1.



Slika 3.1: Medsebojna povezanost pomembnejših entitet VHDL v projektu.

3.1 Branje tipk in krmilnik akcij

Preden smo lahko igralcu dali možnost upravljanja z igro, smo morali razviti krmilnik vhodnih tipk in na podlagi pritiskov le-teh v krmilniku akcij poslati ustrezno akcijo v logiko, ki upravlja z igro.

3.1.1 Krmilnik tipk

Krmilnik tipk smo implementirali v datoteki `tactile_button.vhd`. Vhodi vanj so poleg ure in signala za ponastavitev trije signali tipa `std_logic`:

1. `button_i`. Ta signal se poveže z ustreznim V/I blokom (angl. IOB) na čipu, ki je na tiskanem vezju povezan s tipko [8].

2. `press_ack_i`. Na tem signalu čakamo potrditev krmilnika akcij, da je prejel oziroma obdelal pritisk tipke.

3. `press_o`. Ta izhod predstavlja filtriran pritisk tipke. Po pritisku je aktiven do naslednje urine fronte po prejemu potrditve na zgornjem vhodu.

V krmilniku prvo z dvema flip flopoma tipa D vhodni signal `button_i` sinhroniziramo s sistemsko uro, s čimer omilimo problem metastabilnosti [2]. Ko krmilnik zazna pritisk tipke na vhodu, sproži signal `press_o` in začne čakati na potrditev iz nadrejenega avtomata in konec pritiska na tipko. Po izpolnitvi obeh pogojev se premakne v začetno stanje čakanja na pritisk.

Ker imamo več tipk, je bilo potrebno ustvariti ustrezno število instanc krmilnika za posamezno tipko, kar smo storili preko entitete, vsebovane v datoteki `tactile_buttons.vhd`. V tej entiteti smo vse tri zgornje signale pretvorili v obliko vektorjev, jih ustrezno preimenovali in posamezne bite priklopili na tukaj vzporedno ustvarjene instance krmilnika za posamezno tipko.

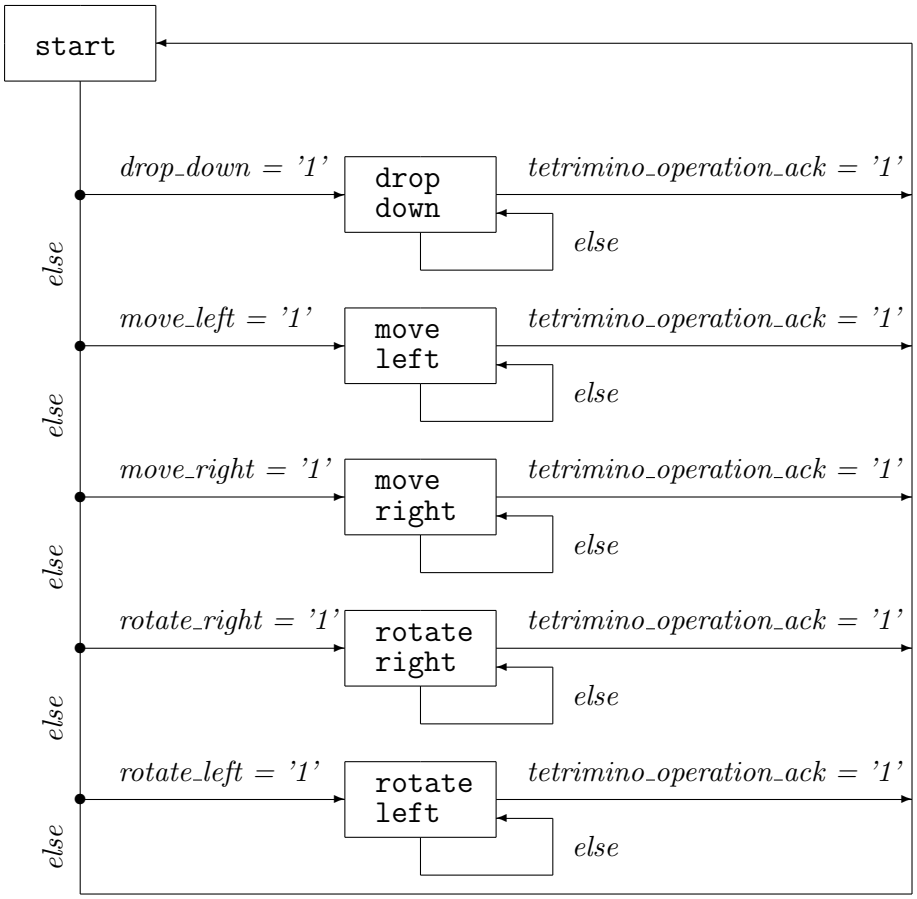
3.1.2 Krmilnik akcij

Krmilnik akcij smo implementirali kot končni avtomat, vsebovan v delu `INPUT_LOGIC` krovne strojno-specifične entitete `top_level`. Krmilnik je potreben iz dveh razlogov:

1. Krovna strojno-neodvisna entiteta igre pričakuje na vhodu `active_operation_i` enumeracijo akcije tipa `active_tetrimino_operations` oziroma se sama ne ukvarja s tipkami. Slednje zato, ker bi na drugi razvojni plošči lahko za krmiljenje aktivnega elementa uporabljali navadno računalniško tipkovnico, igralno palico ali pa kakšen drug kos strojne opreme.

2. Vsak krmilnik tipke deluje neodvisno, kar pomeni, da se lahko signal za pritisk sproži istočasno na dveh ali več tipkah. Zaradi tega moramo njihove izhode nekako multipleksirati.

Končni avtomat je prikazan na sliki 3.2. V vsakem od njegovih stanj sprožimo ustrezno akcijo s postavitvijo signala `tetrimino_operation` tipa



Slika 3.2: Končni avtomat krmilnika akcij.

`active_tetrimino_operations`, ki je vezan na istoimenski vhodni signal v strojno-specifičen krovni modul, v eno od veljavnih vrednosti. Stanja avtomata in vrednosti tega signala v njih so:

Stanje avtomata	akcija	opis akcije
`state_start`	`ATO_NONE`	Ni akcije
`state_drop_down`	`ATO_DROP_DOWN`	Hitri spust elementa
`state_move_left`	`ATO_MOVE_LEFT`	Premik levo
`state_move_right`	`ATO_MOVE_RIGHT`	Premik desno
`state_rotate_right`	`ATO_ROTATE_CLOCKWISE`	Obrat za $\frac{\pi}{2}$ v smeri urinega kazalca (desno)
`state_rotate_left`	`ATO_ROTATE_...` `...COUNTER_CLOCKWISE`	Obrat za $\frac{\pi}{2}$ v obratni smeri urinega kazalca

V začetnem stanju (`state_start`) čakamo na znak iz vsaj enega krmilnika tipke. Ob urini fronti bodisi ostanemo v tem stanju, če nobena tipka ni bila pritisnjena, bodisi preidemo v stanje za dodeljeno akcijo tiste izmed pritisnjenih tipk, ki ima najvišjo prioriteto. Prioritete posameznih signalov niso pomembne, temveč jih potrebujemo zaradi tega, ker naenkrat lahko preidemo le v eno naslednje stanje avtomata.

V vseh drugih stanjih tega avtomata dajemo krmilniku ustrezne tipke znak, da smo prejeli in obdelali pritisk. Poleg tega preverjamo vsebino signala `tetrimino_operation_ack`, ki je povezan na istoimenski izhod krovnega modula igre. V začetno stanje se premaknemo po prejemu znaka, da je zahtevana akcija bila izvršena. V nasprotnem primeru ostanemo v trenutnem stanju.

3.2 7-segmentni LED prikazovalnik

Za prikaz doseženih točk smo razvili krmilnik 7-segmentnih prikazovalnikov, ki so na uporabljeni razvojni plošči prisotni v vrsti osmih. Vsak od tukajšnjih prikazovalnikov je sestavljen iz 8 svetlečih diod (7 delov in pika). Povezan je z devetimi priključki: skupno anodo in 8 katodami. Dalje ima vrsta osmih prikazovalnikov vse katode priključene na skupno vodilo, tako da so npr. vse pike priključene na isti vodnik. Iz opisa sheme vezja v [8] lahko tudi razberemo, da je signal za anode obrnjen oziroma aktiven pri nizkem stanju

napetosti na izhodu čipa FPGA. Za osvetlitev enega segmenta moramo torej primerni par anode in katode hkrati držati nizko.

Tako vezje ima za posledico tudi, da ne moremo imeti hkrati prižganih vseh prikazovalnikov, temveč jih moramo časovno multipleksirati. To smo storili tako, da prižgemo posamezno anodo in istočasno v segmente pošljemo dekodirani pripadajoči vhodni 4-bitni signal, ki predstavlja binarno kodirano števko, na vodilo za katode. Signali za vse ostale anode so seveda neaktivni.

Razviti krmilnik se nahaja v datoteki `seven_seg_display.vhd`, v kateri uporabljamo entiteto iz datoteke `seven_seg_digit.vhd` za dekodiranje števke v segmente.

3.3 Krmilnik VGA

Glavni vmesnik v smeri uporabnika je zaslon standardne VGA [14] ločljivosti 640 točk v širino, 480 točk v širino in 16 barvami za vsakega od treh barvnih kanalov: rdečo, zeleno in modro (RGB). Slednje nam da na voljo $16 \times 16 \times 16 = 4096$ barv, kar je mnogo več kot 11 barv, kolikor smo jih dejansko uporabili za našo implementacijo igre tetris. Kljub temu pa nam večja zaloga barv omogoči izbrati očem bolj prijazne odtenke.

Za krmiljenje zaslona VGA je bilo potrebno ustvariti vertikalni (VSYNC) in horizontalni (HSYNC) sinhronizacijski signal. Trajanje teh dveh signalov je odvisno od uporabljene ločljivosti zaslona, osveževalne frekvence in frekvence signala za omogočanje ure slikovne točke (angl. pixel clock). Tukaj smo se zaradi frekvence ure 100 Mhz na uporabljeni razvojni plošči odločili izdelati krmilnik VGA za izbrano ločljivost pri natanko štirikrat manjši frekvenci ure 25 MHz s parametri v [2]. Ta frekvenca je lahko dosegljiva s pomočjo števca dolžine dveh bitov, katerega izhod za prekoračitev je vezan na signal za omogočanje ure slikovne točke.

Razvita entiteta za ustvarjanje sinhronizacijskega signala se nahaja v datoteki `sync_generator.vhd`. Za svoje delovanje uporablja števec, na podlagi katerega prek primerjalnikov z podanimi parametri krmili kontrolne signale.

Pri tem smo za začetek štetja izbrali levi oziroma zgornji rob izrisne površine. Krmilnik VGA v datoteki `VGA_controller.vhd` predstavlja zaporedno povezavo dveh takih modulov in s tem tudi povezavo vsebovanih števcev. Signal za omogočanje ure slikovne točke smo povezali s števcem v instanci generatorja za horizontalni sinhronizacijski signal. Nato smo njegov signal za prekoračitev obsega povezali na vhod za omogočanje ure instance generatorja za vertikalni sinhronizacijski signal.

S tako vezavo signalov smo dobili navidezni zaslon, ki v svojem zgornjem levem kotu vsebuje dejansko površino zaslona. Vse ostalo predstavlja čas, v katerem izris ni aktiven. Zaradi tega dobimo ogromno časovno okno za izvajanje drugih opravil, ki jih med izrisom ne bi smeli.

Poleg sinhronizacijskih signalov moramo izrisnem cevovodu igre, opisanem v razdelku 4.6, pošiljati tudi zaslonske koordinate. Slednje dosežemo preko signalov:

- `row_o` določa vrstico v navideznem zaslonu,
- `col_o` določa stolpec v navideznem zaslonu,
- `enable_draw_o` pove ali se nahajamo na izrisni površini navideznega naslova,
- `screen_end_o` da znak, da smo prešli v največje območje izven izrisne površine.

3.4 Definicije

V projektu skoraj vsepovsod uporabljamo konstante in tipe, ki so vezani na lastnosti uporabljene strojne opreme. Da bi se izognili dolgim delom deklaracije entitet za generično parametrizacijo in posledično pretiranemu ponavljanju in povezovanju, smo se odločili vse definicije shraniti v strojno-specifičen paket. Shranili smo ga v datoteko board/Nexys4/definitions.vhd pod imenom `definitions`. Tega nato z direktivo "`use work.definitions.all;`"

vključimo, kjer je to potrebno. Pred sintezo moramo le še vključiti definicije in krovni modul za podprto ploščo, s čimer bo sintezno orodje našlo pravi paket in entiteto.

3.5 Krovni modul

Krovni strojno-specifični in s tem tudi krovni modul našega projekta se nahaja v datoteki v `board/Nexys4/top_level.vhd`. V njem povežemo vse strojno-specifične komponente s krovnim strojno-neodvisnim modulom, ki je hkrati krovni modul igre.

Vhodi vanj so vsi signali smernih tipk, urin signal in signal tipke "CPU RESET", katero smo uporabili za ponastavitev igre. Signal slednje je na uporabljeni plošči invertiran [8] oziroma je ponastavitev aktivna takrat, ko iz signala preberemo vrednost `'0'`. Tudi tega smo prvo sinhronizirali na sistemsko uro z uporabo dveh flip-flopov tipa D. Nato smo sinhronizirani signal invertirali in ga ponovno shranili v tretji flip-flop. Izhod tretjega nato uporabljamo kot signal za ponastavitev sistema.

Izhodi so: signali za anode in katode 7-segmentnega prikazovalnika, sinhronizacijska signala zaslona in signali za vse tri komponentne barvnega vektorja, Prva dva pridobimo iz modula za 7-segmentni izris, preostala dva pa iz izrisnega cevovoda, kateri predstavlja strojno-neodvisni krovni modul.

Zaradi razvojnih odločitev v razdelku 3.3, moramo tukaj iz sistemske ure ustvariti ustrezen vhod za uro slikovne točke `pixelclock_i`. To smo storili prek števca `Inst_counter_pixelclockprescale`, katerega izhod za prekoračitev smo povezali z zgornjim vhodom. Preostane nam le še med seboj povezati krmilnik zaslona, krmilnik prikazovalnika, krmilnik akcij in V/I signale s krovno entiteto igre.

Poglavje 4

Opis jedra razvite strojne opreme

Pri zasnovi projekta je bil eden od temeljnih ciljev ločiti posamezne funkcije v lastne enote tako v projektu kot na končnem vezju. S tem se je poenostavilo testiranje posameznih komponent, bodisi prek samostoječe (angl. standalone) implementacije na čipu bodisi preko t.i. testnih klopi (angl. test bench) vsebovanih v datotekah s predpono `testbench_`. Tudi kompleksnost posamezne funkcijske enote je manjša, kar ima dva takojšnja pozitivna učinka in sicer:

- olajša razvoj,
- omogoča višjo hitrost posamezne enote.
  Ker je logičnih gradnikov v enoti manj, lahko razvojno orodje le-te položi bolj skupaj. Posledično je na FPGA med gradnikoma manjše število preklopnih elementov, kar pomeni, da je čas fronte signala od enega do drugega krajši. V tem primeru je možno skrajšati periodo uro oziroma povečati njeno frekvenco.

Kot primer ločenega testiranja komponent velja izpostaviti razviti projekt `VGA`, katerega namen je testiranje krmiljenja zaslona priklopljenega preko

vmesnika VGA. Primer testne klopi pa je `testbench_active_tetrimino.vhd`, katere namen je testiranje izrisa aktivnega tetrimina na zaslon.

Takšna zasnova prinese tudi nekaj slabosti, od katerih je najbolj izrazita večja poraba logičnih elementov. Prenos stanj in povezane logike (za določanje naslednjega stanja in izhodov) enega končnega avtomata v svojega nadrejenega (recimo avtomata za aktivni element v avtomat, ki krmili dostop do pomnilnika za odložene elemente) pomeni, da ne potrebujemo več logike za sinhronizacijo oziroma dogovarjanje med njima. Ta je nujno potrebna v kolikor avtomati dostopajo do deljenih resursov. Primer in delovanje takšne sinhronizacijske logike smo podrobno opisali v razdelku 4.1.3.

4.1 Igralno polje

Igralno polje in povezana logika sta implementirana v datoteki `tetris_block.vhd`. Do njega morajo poleg v tej datoteki vsebovane logike za brisanje imeti dostop še krmilnik za aktivni tetrimino (za shranitev novo odloženega tetrimina), krmilnik za odstranjevanje vrstic (brisanje polnih vrstic in premik zgornjih navzdol) in izrisni cevovod (prikazovanje pomnilnika na zaslonu).

4.1.1 Pomnilnik za igralno polje

Pomnilnik, ki predstavlja igralno polje mora pomniti osem različnih barv: po eno za vsako od sedmih tetrimin in barvo, ki označuje praznino in je enaka barvi ozadja polja. Za shranjevanje samih barv bi porabili preveč prostora, tako da smo se odločili definirani nov podatkovni tip `tetrimino_shape_type`, ki ima natanko 8 možnih vrednosti in enolično določa barvo.

Začetni naslov pomnilnika je enak 0 in predstavlja zgornji levi kot zaslona. Velikost pa je definirana s konstantama `config.number_of_rows` in `config.number_of_columns`, ki določata število vrstic in stolpcev. Pomnilnik naslavljamo z bitnim vektorjem, ki ga dobimo iz zaporedne povezave bitnih vektorjev naslova vrstice in stolpca, kar pomeni, da se naslovi v njem povečujejo prvo po stolpcih, nato pa po vrsticah:

```
to_integer (ram_write_address.row & ram_write_address.col)
```

Znak "&" v zgornjem izvlečku iz procesa za pisanje v pomnilnik predstavlja združitev teh dveh vektorjev. Funkcija `to_integer`, definirana v paketu `ieee.numeric_std_unsigned`, nato pretvori združen bitni vektor v spremenljivko tipa `natural` za potrebe indeksiranja.

Za implementacijo smo se odločili uporabiti pomnilnik s pisanjem, sinhroniziranim glede na sistemsko uro `clock_i`, in z asinhronim branjem. Prvo zato, ker vse implementacijske tehnologije ne podpirajo pomnilnika z asinhronim pisanjem [3], drugo pa zaradi lažje izvedbe dostopa vanj. Slednje pomeni, da nam ni potrebno prvo nastaviti naslov in nato v naslednji urini periodi brati, temveč lahko oboje naredimo naenkrat. V praksi to pomeni, da imamo namesto dveh stanj v končnem avtomatu le enega za posamezno opravilo, oziroma, da se nam zaporedje brezpogojno sledečih si stanj skrajša za eno.

Ker pri trenutni velikosti igralnega polja (30×16) pomnilnik ne izpolnjuje kriterijev, po katerih bi orodje Xilinx Vivado za njegovo implementacijo uporabilo večji blokovni pomnilnik (angl. BRAM), je le-ta implementiran kot distribuiran RAM pomnilnik. To je doseženo s povezavo osmih poizvedbenih tabel za vsakega od treh bitov, uporabljenih za predstavitev tipa `tetrimino_shape_type`, kar je razvidno v izvlečku iz dnevnika sinteze (angl. synthesis log):

Module Name	`tetris`
RTL Object	`Inst_tetris_render_pipeline/`
	`...Inst_tetris_block/RAM_reg`
Inference	`Implied`
Size (Depth x Width)	`512 x 3`
Primitives	`RAM64M x 8`

Preslikavo elementa v barve in pomnilnika na zaslon smo naredili v izrisnem cevovodu, opisanem v razdelku 4.6.

4.1.2 Štetje točk

Štetje točk smo implementirali z osemmestnim desetiškim števcem kvadratov, odloženih v igralnem polju. Dolžina števca smo izbrali tako, da se ujema s številom 7-segmentnih LED prikazovalnikov, ki so na voljo na uporabljeni razvojni plošči.

Ker se v binarnem svetu desetiškega števca ne da narediti direktno, smo ga sestavili iz osmih števcev `counter_until`, vsak od katerih šteje po modulu 10. Pri tem je vhod za omogočanje ure (angl. clock enable) prvega enak signalu `active_write_enable`, ki sicer omogoča uro za pisanje v igralno polje s strani avtomata za aktivni tetrimino. To pomeni, da vsak odloženi tetrimino nagradimo s štirimi točkami. Isti vhod pri ostalih števcih je enak izhodu za prekoračitev obsega prejšnjega enomestnega števca. Rezultat je osemmestni števec.

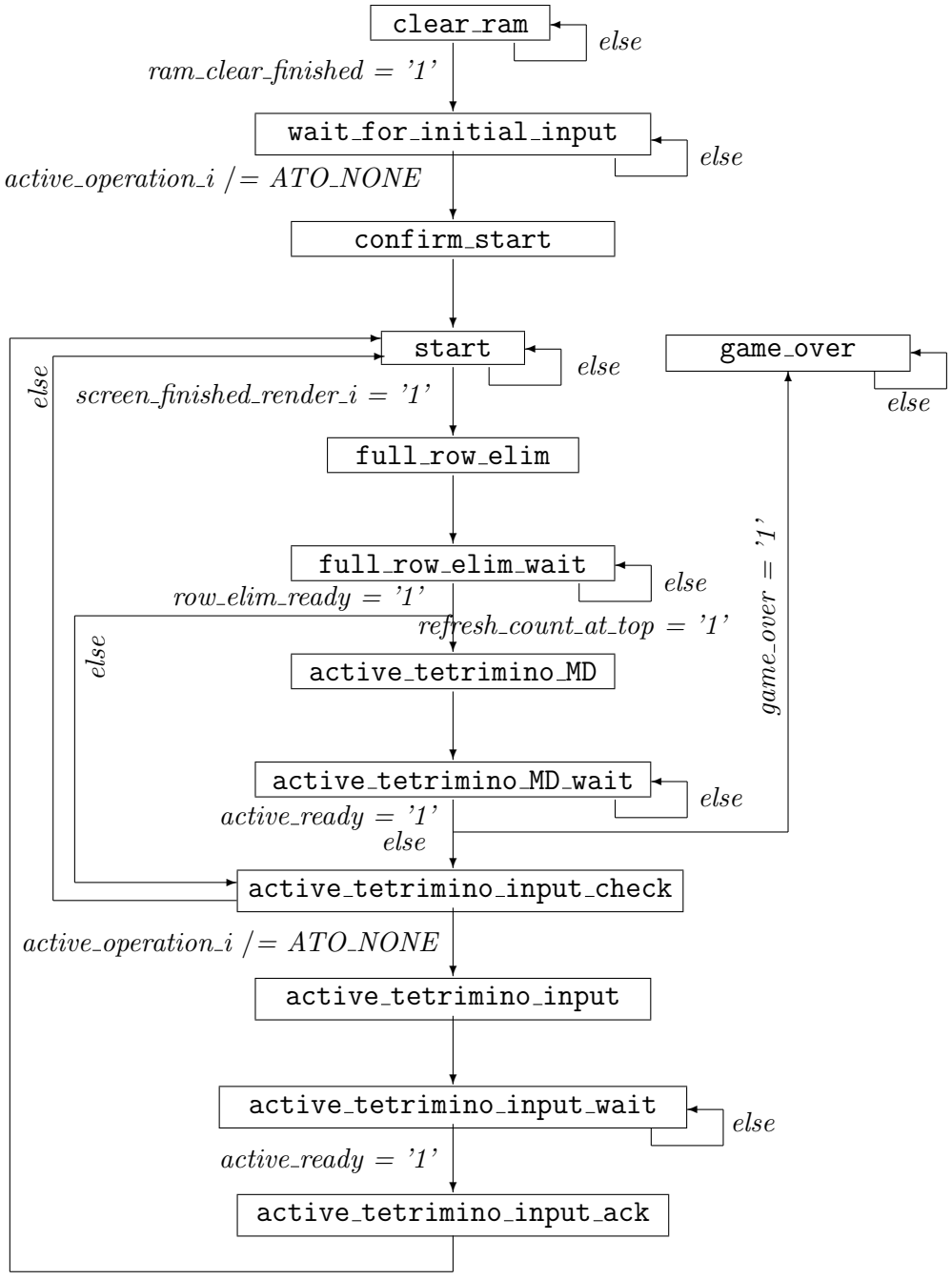
4.1.3 Krmilnik dostopa do pomnilnika

Ker imamo več kot eno enoto, ki mora dostopati do pomnilnika, smo morali multipleksirati njihov dostop. To smo naredili s končnim avtomatom, katerega stanja so prikazana na sliki 4.1.

Avtomat krmili dostop do pomnilnika preko štirih multiplekserjev 4v1. Potrebujemo jih za bralni naslov, pisalni naslov, podatke za pisanje in signal za omogočanje pisanja. Kot primer navajamo multiplekser za omogočanje pisanja:

```
with ram_access_mux select ram_write_enable <=
  '1'                       when MUXSEL_RAM_CLEAR,
  '0'                       when MUXSEL_RENDER,
  row_elim_write_enable     when MUXSEL_ROW_ELIM,
  active_write_enable       when MUXSEL_ACTIVE_TETRIMINO;
```

Tukaj signal `ram_access_mux` predstavlja izbirni signal, ki ga moramo krmiliti iz našega avtomata. Možne vrednosti zanj so torej `MUXSEL_RAM_CLEAR`, `MUXSEL_RENDER`, `MUXSEL_ROW_ELIM` in `MUXSEL_ACTIVE_TETRIMINO`.



Slika 4.1: Stanja pri končnem avtomatu, ki krmili dostop do igralnega polja.

Začetno in obenem tudi stanje po ponastavitvi (angl. reset) tega avto-

mata je `state_clear_ram`.

Brisanje pomnilnika

Preden se (nova) igra lahko začne, moramo pobrisati sestavne kvadrate prej odloženih tetriminov. Ker celotnega distribuiranega RAM pomnilnika zaradi tehnoloških omejitev v uporabljenem čipu FPGA ne moremo pobrisati (ponastaviti) v eni urini periodi, moramo to narediti za vsak element posebej.

Izbris smo implementirali s pomočjo števca `Inst_ram_clear_counter` in stanja `state_clear_ram` končnega avtomata. V tem stanju na izbirni signal multiplekserjev damo vrednost `MUXSEL_RAM_CLEAR` in omogočimo štetje števca s stavkom `ram_clear_enable <= '1'`. Prejšnje povzroči, da se pisalni naslov pomnilnika poveže z izhodom za vrednost števca. Poleg tega se pisalno podatkovno vodilo nastavi na vrednost `TETRIMINO_SHAPE_NONE`, ki predstavlja nezasedenost, in brezpogojno aktivira pisanje.

Števec šteje od prvega do zadnjega elementa v pomnilniku, pri čemer zgornje vrednosti na povezavah slednjega povzročijo, da v vsaki urini periodi pobrišemo drug element. Avtomat ves ta čas preverja, če je signal `ram_clear_finished` enak '1'. V primeru neresničnosti ostanemo v istem stanju, drugače pa nadaljujemo v stanje `state_wait_for_initial_input`, opisano v naslednjem razdelku. Pogoj je izpolnjen le, ko je števec tik pred prekoračitvijo svojega obsega. Takrat nam na izhodu `overflow_o`, ki je povezan z zgornjim signalom, da vrednost '1' namesto '0'. V sledeči urini periodi se stanje avtomata spremeni, števec pa preide nazaj na začetek štetja. Slednje se zgodi zaradi tega, ker je do naslednje urine fronte štetje še vedno omogočeno.

Začetek igre

Na začetku igre moramo počakati na znak, da je igralec pripravljen na igro. Pri tem za znak šteje pritisk na katerokoli od veljavnih tipk Dejanje, ki je vezano nanjo, pa se "prestreže" in se ne izvrši.

Neizvršitev smo dosegli tako, da v stanju `state_wait_for_initial_input`

čakamo dokler signal `active_operation_i`, ki prenaša enumeracijo aktivne akcije iz krmilnika akcij, ni različen od vrednosti `ATO_NONE`. Slednje pomeni, da uporabnik od nas ni zahteval nobene akcije. Ko se to zgodi, preidemo v stanje `state_confirm_start`, v katerem s postavitvijo signala `active_operation_ack_o` na '1' potrdimo prejem pritiska. V naslednji urini periodi preidemo v stanje `state_start`, s čimer popolnoma ignoriramo trenutno akcijo.

"Glavna zanka" igre

V stanju `state_start` čakamo na postavitev signala `screen_finished_render_i` iz izrisnega cevovoda. Ta signal nam pove, da je krmilnik VGA končal z izrisom na zaslon in se premaknil v območje navideznega zaslona, v katerem oddaja le sinhronizacijske signale, ki so nujni za delovanje protokola VGA. Parametri krmilnika VGA pri izbrani ločljivosti zaslona 640×480 nam v enačbi 4.1 povedo, da imamo za izvedbo logike igre na voljo približno 32800 period ure za VGA krmilnik oziroma 131200 period sistemske ure na plošči Nexys4.

$$(t_{\text{v\_fp}} + t_{\text{v\_bp}} + t_{\text{v\_pw}}) \times t_{\text{h\_line}} = (10 + 29 + 2) \times 800 = 32800 \tag{4.1}$$

Po prejemu signala `screen_finished_render_i` preidemo naprej v stanje `state_full_row_elim`.

Hkrati s čakanjem na zgornji signal dajemo izrisnemu cevovodu dostop do lokalnega pomnilnika z vpisom vrednosti `MUXSEL_RENDER` v izbiralni signal `ram_access_mux`.

Odstranjevanje polnih vrstic

V stanju `state_full_row_elim` pošljemo avtomatu za odstranjevanje polnih vrstic, vsebovanim v podrejeni entiteti `tetris_row_elim`, znak za začetek delovanja s postavitvijo signala `row_elim_start` na '1'. Da bi ta avtomat lahko naredil to kar od njega zahtevamo, mu moramo hkrati dodeliti do-

stop do pomnilnika za igralno polje. Podobno kot pri brisanju, to storimo s postavitvijo multiplekserjev v ustrezno stanje: `MUXSEL_ROW_ELIM`.

Z naslednjo urino fronto brezpogojno preklopimo v stanje `state_full_row_elim_wait`, v katerem čakamo znak '1' na izhodu `fsm_ready_o` podrejenega avtomata, ki označuje, da je ta končal z delom. Hkrati pa mu še vedno dajemo dostop do pomnilnika za odlaganje tetriminov. To se morda sliši nepotrebno, vendar bi v nasprotnem primeru dostop trajal le eno urino periodo, kar je bistveno premalo v primerjavi z časom, ki ga za zaključek dela potrebuje podrejeni avtomat.

V podrejenem avtomatu imamo stanje `state_start`, katerega funkcija je čakanje na znak za začetek in **hkrati** sporočanje, da je končal z delom. Hkratno sporočanje zaključka je tukaj v redu, saj tukajšnji avtomat šele v naslednjem stanju (naslednja urina perioda) po sprožitvi podrejenega avtomata začne čakati na zaključek njegovega dela.

Po zaključku opravil podrejenega avtomata nadaljujemo bodisi z zagonom padanja tetrimina v primeru izpolnitve pogoja, opisanega v naslednjem razdelku, bodisi s procesiranjem uporabniških akcij.

Padanje tetrimina

Padanje tetrimina je operacija, ki se sproži vsakih nekaj izrisov na zaslon. Natanko koliko izrisov mora preteči, preden se to zgodi, definira vrednost `refresh_count_top`. Kot že ime pove, je to vrhnja vrednost za števec `Inst_refresh_counter`, ki šteje osvežitvene (angl. refresh) cikle zaslona. Signal za omogočanje ure in s tem tudi znak za premik števca navzgor je že opisani signal `screen_finished_render_i`.

Pri tem števcu smo uporabili izhod `count_at_top_o`, ki predstavlja prenos štetja, namesto običajnega izhoda prekoračitve obsega (angl. overflow) štetja. Če tega ne bi naredili, bi morali padanje zagnati takoj za stanjem `state_start`. To bi pomenilo, da bi bil učinek za ponazoritev brisanja vrstic krajši za en zaslonski izris. Poleg tega za realizacijo uporabljenega izhoda potrebujemo ena vrata `in` manj, ker se prekoračitev zgodi takrat, ko je števec

pri vrhnji vrednosti in je aktiven signal za omogočanje ure. Ker je uporabljeni signal aktiven do konca naslednjega izrisa, imamo več kot dovolj časa za njegovo zaznavo.

Sinhronizacija med avtomatom za krmiljenje dostopa in tistim za krmiljenje aktivnega tetrimina je skoraj enaka. Implementirali smo jo prek stanj `state_active_tetrimino_MD` in `state_active_tetrimino_MD_wait`. Edina razlika je v tem, da tukaj krmilimo še multiplekser za aktivno akcijo:

```
with active_tetrimino_command_mux select active_operation <=
  ATO_MOVE_DOWN          when ATC_MOVE_DOWN,
  active_operation_i     when ATC_USER_INPUT;
```

Razvidno je, da ta multiplekser izbira aktivno akcijo med uporabniško akcijo, ki jo dobimo iz krmilnika akcij na vhodu `active_operation_i`, in akcijo premika navzdol, ki uporabniku ni neposredno dostopna. Slednjo v teh dveh sinhronizacijskih stanjih izberemo z vrednostjo izbirnega signala `ATC_MOVE_DOWN`.

Ko podrejeni avtomat konča z delom, moramo le še preveriti, če se je igra končala. Avtomat nam to sporoči z aktivnim signalom `fsm_game_over_o`. Takrat preidemo v stanje `state_game_over`, v katerem čakamo v nedogled oziroma dokler uporabnik ne pritisne tipke za ponastavitev igre. V nasprotnem primeru bodisi nadaljujemo s procesiranjem s strani uporabnika zahtevane akcije bodisi se vrnemo nazaj v stanje `state_start`, če avtomat še ni končal z delom.

Uporabniške akcije

Uporabnikovo zahtevo prejmemo iz krmilnika akcij po vhodnemu signalu `active_operation_i`, ki ga obdelujemo v stanju `state_active_tetrimino_input_check`. Če je ta signal različen od vrednosti `ATO_NONE`, preidemo v stanje `state_active_tetrimino_input`, kjer zaženemo avtomat za aktivni tetrimino. V nasprotnem primeru preskočimo sledeča stanja in se vrnemo v `state_start`.

Preverjanje pogoja je tukaj (namesto v avtomatu za aktivni tetrimino) potrebno zaradi tega, ker bi se nam lahko zgodilo, da bi se vhodni signal pojavil šele v drugem delu sinhronizacijske logike. Avtomat za aktivni tetrimino bi nam še vedno oddajal signal za pripravljenost (ker je v začetnem stanju), kar bi povzročilo, da se takoj ob naslednji urini fronti premaknemo v prvo stanje izven sinhronizacijske logike in mu odvzamemo dostop do pomnilnika igralnega polja. Ob isti urini fronti bi podrejeni avtomat odšel v ukazu primerno stanje, ki bi se nato zaradi pomanjkanja dostopa do pomnilnika izvršilo nepravilno.

To preverjanje bi lahko tudi izvedli v čakalnih stanjih sinhronizacijske logike delov avtomata za odstranjevanje vrstic in pomik tetrimina navzdol, od koder preidemo v del za izvrševanje uporabniških akcij. Vendar smo se zaradi dveh razlogov vseeno odločili dodati novo stanje:

1. Preverjanje pogoja na enem mestu namesto na dveh pomeni, da ob spremembah ni potrebno posodabljati ostalih kopij kode. Posledično je manj verjetno, da bi pri tem naredili napako.

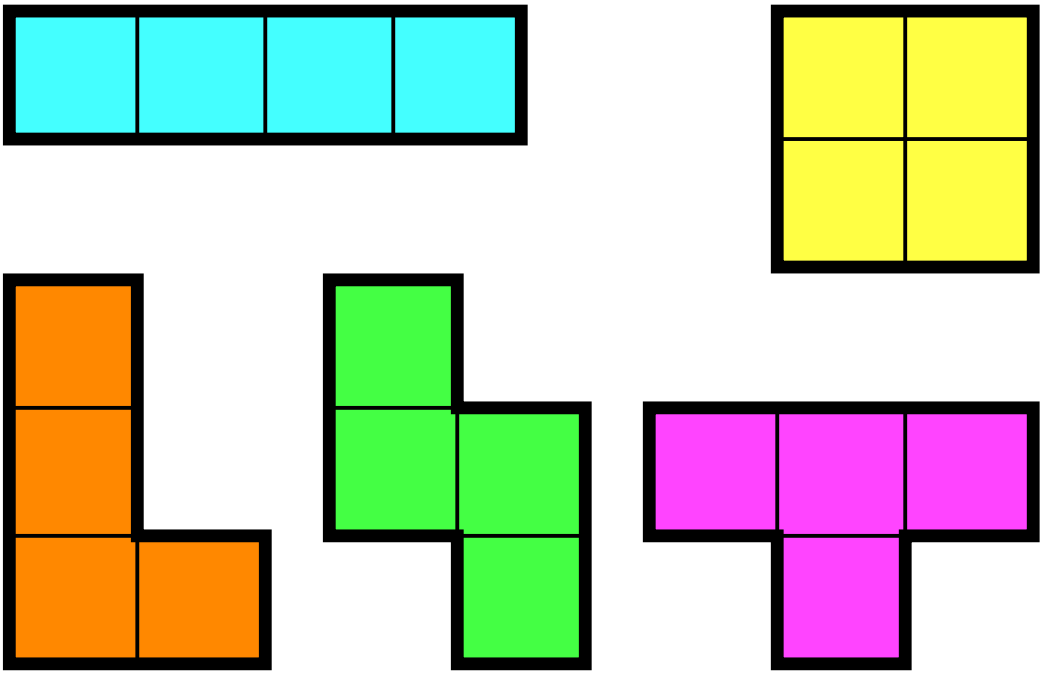
2. Dodano stanje nam zmanjša število pogojev, ki jih je potrebno preveriti v čakalnem delu sinhronizacijske logike, kjer že tako ali tako imamo tri izhodne povezave. Kot pri prejšnji točki to tudi tukaj zmanjšuje kompleksnost kode in s tem možnost za napake.

V stanjih `state_active_tetrimino_input` in `state_active_tetrimino_input_wait` smo implementirali zdaj že znano sinhronizacijsko logiko. Razlika je v tem, da sedaj prek izbirne konstante `ATC_USER_INPUT` multiplekserja za aktivno akcijo v krmilnik za aktivni tetrimino posredujemo vrednost vhodnega signala namesto znaka za spust tetrimina.

Preostane nam le še potrditi, da smo akcijo prejeli. To storimo v stanju `state_active_tetrimino_input_ack` s postavitvijo izhodnega signala `active_operation_ack_o` na vrednost '1'. Razlog, da tega nismo naredili že v prejšnjih dveh stanjih je v tem, da mora biti enumeracija akcije

`active_operation_i` za pravilno delovanje podrejenega avtomata nespremenjena do njegovega zaključka.

4.2 Operacije na aktivnem tetriminu



Slika 4.2: Vse možne oblike tetriminov razen dveh, katera sta zrcalni kopiji oranžnega in zelenega tetrimina [11]

Edini del tetrisa s katerim uporabnik lahko upravlja je geometrijski element sestavljen iz štirih s stranicami povezanih kvadratov in ga v tetrisu imenujemo tetrimino [11]. Različni tetrimini v tetrisu do rotacije natančno predstavljajo vse možne kombinacije štirih s stranicami povezanih kvadratov in so prikazani na sliki 4.2. Tu je vredno poudariti, da sta dva od sedmih v tetrisu prisotnih tetriminov zrcalni sliki zelenega in oranžnega tetrimina, saj nimamo operacije zrcaljenja.

Končni avtomat in povezana logika za izvajanje operacij na njem sta vsebovana v datoteki `tetris_active_tetrimino.vhd`. Uporabniku dostopna

dejanja so premik levo in desno, obračanje v smeri urinega kazalca in obratno ter hitri spust. Med nedostopne pa sodi padanje tetrimina.

Pomnjenje in izris aktivnega tetrimina

Za upravljanje z aktivnim tetriminom ga moramo najprej shraniti. Zaradi hitrega dostopa pri izrisovanju na zaslon moramo shraniti naslov znotraj igralnega polja za vsakega v njem vsebovanih štirih kvadratov. Vsak od teh naslovov je sestavljen iz dela za stolpec (`blockX_row`) in dela za vrstico (`blockX_column`), kjer `X` predstavlja kvadratu dodeljeno številko. Poleg teh štirih naslovov moramo za določitev barve kvadratov shraniti tudi obliko tetrimina, ki barvo enolično določa. To storimo s signalom `tetrimino_shape`.

V procesu `ACTIVE_RENDER` tako lahko implementiramo 8 primerjalnikov, ki preverjajo trenutni naslov izrisovanja `active_address_i` z vsakim od parov naslovov sestavnih kvadratov tetrimina. Na izrisni izhod `active_data_o` pošljemo bodisi obliko trenutnega tetrimina v primeru ujemanja v katerem-koli paru bodisi `TETRIMINO_SHAPE_NONE`, ki označuje praznino na naslovu izrisa.

Med razvojem krmilnika za operacijo vrtenja smo ugotovili, da bi bilo dobro čez cel tetrimino orisati navidezni okvir (angl. bounding box) kvadratne oblike. Za dolžino njegove stranice smo izbrali število 4, saj toliko meri najdaljši možni tetrimino. Za opis okvirja hranimo t. i. naslov kota v signalih `corner_row` in `corner_column`. Signala določata naslov zgornjega levega kvadrata v našem okvirju.

S pomočjo te predstavitve lahko vsakega od naslovov v tetriminu vsebovanih kvadratov predstavimo s parom odmikov od naslovov okvirja. Pri tem lahko za vseh sedem tetriminov in njihove orientacije te odmike preprosto naložimo iz bralnega pomnilnika, v katerega smo shranili vnaprej izračunane vrednosti. Še več, pri implementaciji akcije vrtenja smo ugotovili, da je tudi funkcijo prehoda iz ene orientacije v drugo najbolje predstaviti z vnaprej izračunanimi odmiki. V ta namen moramo shraniti tudi orientacijo trenutnega tetrimina, kar storimo s signalom `tetrimino_rotation`. Bralni

pomnilnik uporabimo pri izračunih novih naslovov vsebovanih štirih kvadratov.

Dostop do pomnilnika igralnega polja

Za uspešno shranitev odloženih tetriminov in preverjanje nezasedenosti posameznih naslovov smo potrebovali dostop do pomnilnika igralnega polja. Hkratni dostop do vseh štirih kvadratov zaradi njegove organizacije ni bil mogoč, zato smo vrstice in stolpce bralnih in pisalnih naslovov multipleksirali s štirimi multiplekserji, ki izbirajo med naslovi vseh štirih vsebovanih kvadratov. Krmili jih izbiralni signal `block_select`, v katerega lahko zapišemo izbirne konstante `BLOCK0`, `BLOCK1`, `BLOCK2` in `BLOCK3`.

Pisalnega vodila ni potrebno multipleksirati, saj so vsi vsebovani kvadrati iste barve oziroma imajo isto obliko, s katero je barva natančno določena.

Začetno stanje končnega avtomata

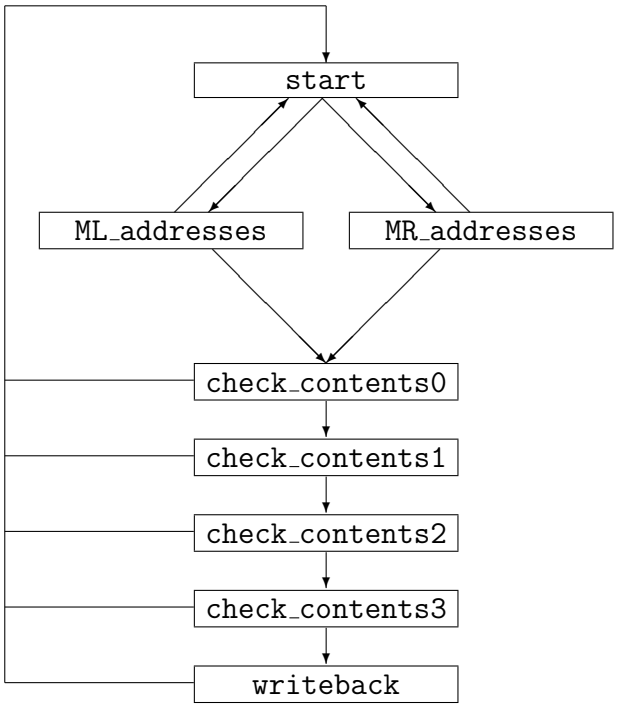
Krmilnik za aktivni tetrimino na znak za začetek čaka v stanju `state_start`, kjer v skladu z opisano sinhronizacijsko logiko sporočamo svojo pripravljenost nadrejenemu avtomatu. Od tega skupaj s postavitvijo signala `operation_i` v eno od možnih stanj, dobimo na vhodu `fsm_start_i` tudi znak za zagon avtomata. Po prejemu znaka preidemo v ustrezno stanje za izračun novih naslovov.

4.2.1 Premiki levo in desno

Vodoravna premika sta najpreprostejši akciji, ki ju lahko izvede uporabnik. Podobno kot pri hitrem spustu se premik požene s pritiskom na tipke in sicer leva tipka za premik levo in desna tipka za premik desno. Krmilnik akcij zatem začne pošiljati vrednost `ATO_MOVE_LEFT` za prvo in `ATO_MOVE_RIGHT` za drugo operacijo.

Za preverbo izpolnitve potrebnih pogojev za premik in izvedbo ustreznih dejanj nato poskrbijo stanja v končnem avtomatu krmilnika aktivnega

tetrimina, prikazana na sliki 4.3:



Slika 4.3: Stanja dela avtomata za izvajanje vodoravnih premikov aktivnega tetrimina

Izračun novih in preveritev starih naslovov

V stanju `state_ML_addresses` za premik levo in `state_MR_addresses` za premik desno moramo izračunati nove naslove in preveriti veljavnost premika glede na stare naslove.

Pri premiku levo moramo preveriti, da noben naslov stolpca štirih sestavnih kvadratov aktivnega tetrimina ni enak najmanjšemu možnemu naslovu, ki ga predstavlja konstanta `column0`, saj bi v nasprotnem primeru premik povzročil izhod tetrimina iz igralnega polja. Podobno pri premiku desno preverimo, da noben od teh naslovov ni enak najvišjemu možnemu, ki je določen

s konstanto `columnNm1`. V primeru kršitve v vsaj enem od stolpcev premik ignoriramo s prehodom nazaj v stanje `state_start`, drugače pa nadaljujemo v stanje `state_check_contents0`.

Hkrati s preverjanjem tudi zaženemo izračun novih naslovov navideznega okvirja. To storimo tako, da v signal `corner_column_operation`, ki predstavlja operacijo nad naslovom stolpca, naložimo konstanto `MINUS_ONE` v primeru premika levo in `PLUS_ONE` za premik desno. Poleg tega omogočimo zapis v registre za nove naslove (*new_address_write_enable ¡= '1'*), s čimer se v procesu `SAVE_NEW_DATA` poleg novega naslova navideznega stolpca zapišejo tudi nespremenjen naslov navidezne vrstice in novi naslovi posameznih kvadratov v tetriminu. Naslove pridobimo iz navideznih naslovov in vnaprej izračunanih odmikov, shranjenih v bralnem pomnilniku `tetrimino_init_rom`.

Preverjanje zasedenosti

V stanjih `state_check_contents0`, `state_check_contents1`, `state_check_contents2` in `state_check_contents3` moramo preveriti, da se pod novimi naslovi vsakega od štirih kvadratov, ki sestavljajo aktivni tetrimino, ne nahaja v igralnem polju že odloženi kvadrat. To storimo tako, da v signal `block_select` naložimo eno od konstant `BLOCK0` do `BLOCK3`. Signal med drugim izbira, kateri od štirih naslovov kvadratov se naloži v bralni naslov igralnega polja `block_read_address_o`.

V isti urini periodi dobimo v signalu `block_i` vsebino igralnega polja pod tem naslovom. Če je njegova vsebina enaka `TETRIMINO_SHAPE_NONE`, kar pomeni, da prostor ni zaseden, se premaknemo v stanje za preverbo naslednjega od štirih naslovov oziroma v `state_writeback`, če smo preverili že vse. V nasprotnem primeru ignoriramo premik z vrnitvijo v stanje `state_start`.

Shranitev novih naslovov tetrimina

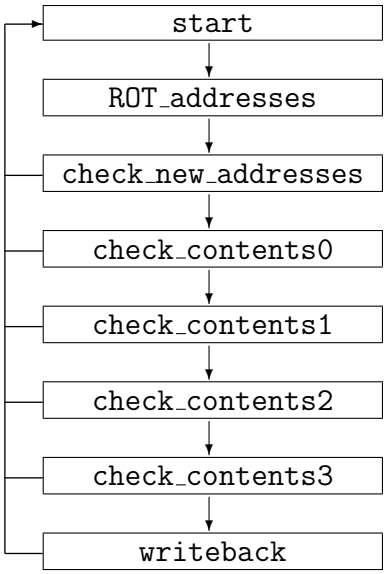
Ko smo že vse preverili in izračunali nove naslove, je vse kar nam preostane zapis novih naslovov in ostalih atributov v registre aktivnega tetrimina. To naredimo v stanju `state_writeback` s postavitvijo signala

`active_address_write_enable`, ki omogoči uro procesu `SAVE_ACTIVE_DATA`. Zaporedje stanj tukaj zaključimo s prehodom nazaj v stanje `state_start`.

4.2.2 Vrtenje tetrimina

Igralcu je na voljo obračanje tetrimina v korakih $\frac{\pi}{2}$. Pri tem se zasuk v smeri urinega kazalca požene s pritiskom na zgornjo tipko, v nasprotni smeri pa s pritiskom na spodnjo. Krmilnik akcij zatem začne pošiljati vrednost `ATO_ROTATE_CLOCKWISE` za prvo in `ATO_ROTATE_COUNTER_CLOCKWISE` za drugo operacijo.

Za preverbo izpolnitve potrebnih pogojev in izvedbo ustreznih dejanj nato poskrbijo stanja v končnem avtomatu krmilnika aktivnega tetrimina, prikazana na sliki 4.4.



Slika 4.4: Stanja dela avtomata za vrtenje aktivnega tetrimina

Izračun novih naslovov

Stanje za izračun novih naslovov se pri nas imenuje `state_ROT_addresses`. Podobno kot pri vodoravnih premikih tetrimina tudi tukaj izračunamo in

shranimo nove naslove. Za razliko od prejšnjih stanj starih naslovov ne preverjamo, temveč le določimo naslednjo orientacijo aktivnega tetrimina (`tetrimino_rotation_next` in nove naslove štirih kvadratov. To kombinatorično storimo v procesu `COMB_NEXT_ROTATION` glede na trenutno orientiranost, shranjeno v `tetrimino_rotation` in vhodno operacijo (`operation_i`). Tako dobimo iz bralnega pomnilnika nove, za vsako orientacijo in tetrimino vnaprej izračunane odmike. Nove naslove v procesu `SAVE_NEW_DATA` prištejemo nespremenjenemu navideznemu naslovu in jih skupaj z novo orientacijo shranimo za obdelavo v naslednjih stanjih končnega avtomata, v katere tudi brezpogojno nadaljujemo.

Preveritev novih naslovov

Za razliko od vodoravnih premikov smo se pri obračanju odločili dodati novo stanje (`state_check_new_addresses`), namen katerega je zgolj preverjanje pogojev za rotacijo. Razlog je v tem, da se naslovi vsebovanih štirih kvadratov ne premikajo lahko predvidljivo glede na navidezni naslov (ki se tu ne premika). Ocenili smo, da je mnogo lažje in zanesljivejše preverjati, ali so novi naslovi še vedno vsebovani v igralnem polju. Če bi poleg določanja novih naslovov le-te tudi preverjali v isti urini periodi, bi dobili zelo dolgo kombinatorično pot med dvema registroma, kar bi nam znižalo dosegljivo frekvenco.

Preveriti moramo, da nobeden on naslovov štirih kvadratov ni izven igralnega polja. Zaradi skrbne izbire odmikov pri prehajanju iz ene orientacije tetrimina v drugo se ne more zgoditi primer, ko bi vsi štirje naslovi končali dve ali več vrstic ali stolpcev izven igralnega polja. Zato je zadosti preveriti, da nobeden od novih naslovov ni enak prvemu zunanjemu naslovu.

Lahko se zgodi, da je ena od dimenzij igralnega polja enaka potenci števila dve. V tem primeru zaradi prekoračitve obsega prvi zunanji naslov postane prvi notranji gledano iz druge smeri. Za potrebe preverjanja pogojev smo tako nove naslove vsebovanih kvadratov razširili za ena v obeh dimenzijah. Te po potrebi shranjujemo v obliki razširjenega bitnega vektorja. Slednje

dobimo z eno-bitno levo razširitvijo vektorja bitov (`std_logic_vector`), ki predstavlja posamezni naslov stolpca ali vrstice.

V primeru kršitve pogojev akcijo ignoriramo, drugače nadaljujemo v stanje `state_check_contents0`, katerega delovanje je opisano v razdelku 4.2.1.

4.2.3 Padanje tetrimina

Padanje je edina operacija, ki je uporabnik ne more direktno pognati. Ko je pogoj za padanje tetrimina, opisan v razdelku 4.1.3 izpolnjen, nam nadrejeni avtomat da znak, naj izvršimo padanje za eno vrstico. V naslednji urini periodi iz začetnega stanja avtomata preidemo v stanje `state_MD_addresses`.

Izračun novih in preveritev starih naslovov

Kot pri podobno imenovanih stanjih za vodoravne premike, moramo tudi v stanju `state_MD_addresses` preveriti ali smo na robu igralnega polja. Za razliko od vodoravnih premikov pa tukaj gledamo spodnji in ne levega oziroma desnega roba. Če je vrstica kateregakoli vsebovanega kvadrata enaka najvišji možni (spomnimo se, da je začetek igralnega polja v zgornjem levem kotu), ki je definirana s konstanto `number_of_rows` pomanjšano za 1, potem smo prišli do dna igralnega polja. Tukaj s prehodom v stanje `state_MD_fill_contents0` začnemo vrsto stanj za odlaganje tetrimina v igralno polje. V nasprotnem primeru nadaljujemo s preverjanjem vsebine pod novimi naslovi v stanju `state_MD_check_contents0`.

Hkrati s preverjanjem izvedemo akcijo prištevanja vrednosti ena k vrstici navideznega okvirja. To storimo s postavitvijo izbiralnega signala `corner_row_operation` v stanje `PLUS_ONE` in omogočimo uro registrom za nove naslove s signalom `new_address_write_enable`. Enako kot pri vodoravnih premikih s tem povzročimo izračun vseh štirih parov naslovov vsebovanih kvadratov.

Preverjanje zasedenosti

Zasedenost pri padanju preverjamo v stanjih `state_MD_check_contentsX`, kjer `X` predstavlja številko kvadrata. Ta stanja imajo skoraj isto funkcijo kot v razdelku 4.2.1 opisana `state_check_contentsX`. Razlika je v tem, da pri neizpolnitvi pogoja od tukaj preidemo v serijo stanj za odlaganje tetrimina namesto ignoriranja akcije. Poleg tega po zaključenem preverjanju preidemo v stanje `state_MD_writeback` namesto `state_writeback`.

Shranitev novih naslovov tetrimina

Stanje `state_MD_writeback` je podobno kot stanje za splošno shranjevanje novih naslovov, ki je opisano v razdelku 4.2.1. Potrebno je zaradi implementacije hitrega spusta, opisanega v sledečem razdelku. V primeru združitve obeh stanj bi ob prevzemu novega tetrimina, opisanem v razdelku 4.2.6, tudi tukaj bila aktivna vhodna operacija `ATO_DROP_DOWN`, kar bi povzročilo takojšen nastanek stebra tetriminov na sredini polja in s tem konec igre.

4.2.4 Hitri spust

Hitri spust je funkcija, ki ni nujna za igranje igre. Pomembna je zato, ker bi bilo brez nje igralcu dolgčas. To bi bilo najbolj izrazito na začetku igre, ko je igralno polje še prazno. Omenjeno akcijo igralec požene s pritiskom na srednjo tipko razvojne plošče Nexys4. Krmilnik akcij tedaj začne prek krmilnika igralnega polja pošiljati vrednost `ATO_DROP_DOWN` skozi vhod `operation_i` v tukajšnji krmilnik, nakar začne čakati na potrditev o izvršitvi.

Postopek izvedbe hitrega spusta je skoraj natanko enak in tudi uporablja isti niz stanj končnega avtomata kot postopek izvajanja navadnega padanja tetrimina. Razlika med njima je v tem, da se v stanju `state_MD_writeback` pri hitrem spustu ne vrnemo v začetno stanje avtomata (`state_start`), temveč s prehodom v stanje `state_MD_addresses` ponovno začnemo spust elementa za eno vrstico navzdol. To zaporedje stanj se ponavlja vse dokler je premik tetrimina navzdol še možen.

4.2.5 Shranitev starega tetrimina v igralno polje

Shranitev oziroma odlaganje starega tetrimina v igralno polje je notranja operacija, ki smo jo implementirali prek stanj `state_MD_fill_contentsX`, kjer `X` predstavlja številko vsebovanega kvadrata, ki ga shranjujemo. Operacija se začne po neuspešnem premiku tetrimina navzdol, ko eno od tamkajšnjih stanj za preverjanje veljavnosti ugotovi kršitev pogojev in povzroči prehod v stanje `state_MD_fill_contents0`.

Podobno kot pri preverjanju pogojev tudi tukaj krmilimo izbirne multiplekserje za dostop do igralnega polja. Pri vsakem stanju te nastavimo s postavitvijo izbirnega signala `block_select` v ustrezno stanje in omogočimo pisanje z aktivacijo signala `block_write_enable_o`.

Ob urini fronti preidemo bodisi v naslednje od teh štirih stanj bodisi v stanje `state_NT_new_addresses`, če smo v stanju `state_MD_fill_contents3`. Tako v štirih urinih periodah shranimo vse štiri vsebovane kvadrate tetrimina na primerne naslove v igralnem polju in preidemo v serijo stanj za prevzem novega tetrimina.

4.2.6 Prevzem novega tetrimina

Prevzem novega tetrimina je druga notranja operacija in se zažene bodisi po ponastavitvi čipa bodisi po shranitvi starega tetrimina v igralno polje.

Nalaganje novih naslovov

V stanju `state_NT_new_addresses` moramo izbrati nov aktivni tetrimino. To storimo s preklopom izbirnega signala `tetrimino_select` v stanje `TETRIMINO_NEW`. Ta signal krmili proces `COMB_NT`, ki ima funkcijo multiplekserja za operand aritmetične enote za premik naslova navideznega okvirja. Pri vseh ostalih stanjih avtomata ta proces za operand določi kar naslove trenutnega tetrimina, tukaj pa za operand izbere konstanto `tetris.tetrimino_start_row` za vrstico in `tetris.tetrimino_start_col` za stolpec. Ti dve konstanti sta definirani v definicijah in prek naslova okvirja

določata začetni naslov tetrimina v igralnem polju.

Poleg naslova je potrebno izbrati tudi orientacijo in obliko novega tetrimina. Za prvo izberemo kar orientacijo starega tetrimina, obliko pa naložimo preko vhodnega signala `nt_shape_i` iz entitete za naslednji tetrimino. Tukaj moramo tudi sprožiti signal `nt_retrieved_o`, s katerim entiteti za naslednji tetrimino povemo, da bomo ob naslednji urini fronti prevzeli novo obliko. Vse kar nam še preostane, je sprožitev pisanja v registre novega tetrimina preko signala `new_address_write_enable` in prehod v serijo stanj za preverjanje zasedenosti.

Preverjanje zasedenosti

Preverjanje zasedenosti pri nalaganju novega tetrimina smo implementirali v stanjih `state_NT_check_contentsX`, kjer `X` določa številko v novem tetriminu vsebovanega kvadrata. Preverjanje je tukaj isto kot pri stanjih, opisanih v razdelku 4.2.1. Razlika je le v tem, da ob neizpolnitvi vsaj enega pogoja namesto ignoriranja akcije preidemo v stanje za zaključek igre, saj v igralnem polju ni več prostora za novi tetrimino. V primeru izpolnitve vseh pogojev iz zadnjega stanja preidemo v stanje `state_writeback`, ki je opisan v razdelku 4.2.1 in poskrbi za zapis novega tetrimina v registre trenutnega.

Konec igre

V primeru neuspešnega preverjanja zgornjih pogojev pristanemo v stanju `state_NT_game_over`. Tukaj s sprožitvijo signala `fsm_game_over_o` nadrejenemu avtomatu pošljemo znak, da nadaljevanje igre ni več mogoče, in ob naslednji urini fronti preidemo v stanje `state_start`. Mogoče se sliši narobe, toda v razdelku 4.1.3 opisano stanje povzroči, da se krmilnik za igralno polje zaklene v trenutno stanje in do ponastavitve več ne prepušča ukazov.

4.3 Odstranjevanje polnih vrstic

Končni avtomat in povezana logika v entiteti `tetris_row_elim`, ki je vsebovana v datoteki `tetris_row_elim.vhd`, določata dogajanje po zapolnitvi vsaj ene vrstice v igralnem polju. Avtomat se zažene po vsakem zaključenem izrisu na zaslon, kot je opisano v razdelku 4.1.3. Na znak za začetek čaka v stanju `state_start`.

4.3.1 Učinek "fade-in"

Pri odstranjevanju vrstic smo se odločili uporabiti učinek "fade-in", ker je na ta način možno zelo elegantno poudariti odstranitev vrstice. V nasprotju z učinkom "fade-out", pri katerem uporabnik vidi postopno izginevanje vrstice, se tukaj vrstica postopno spremeni v popolnoma belo, nato pa pri nas "v trenutku" izgine oziroma se spremeni v črno. S tem dosežemo največji kontrast in pritegnemo pozornost igralca.

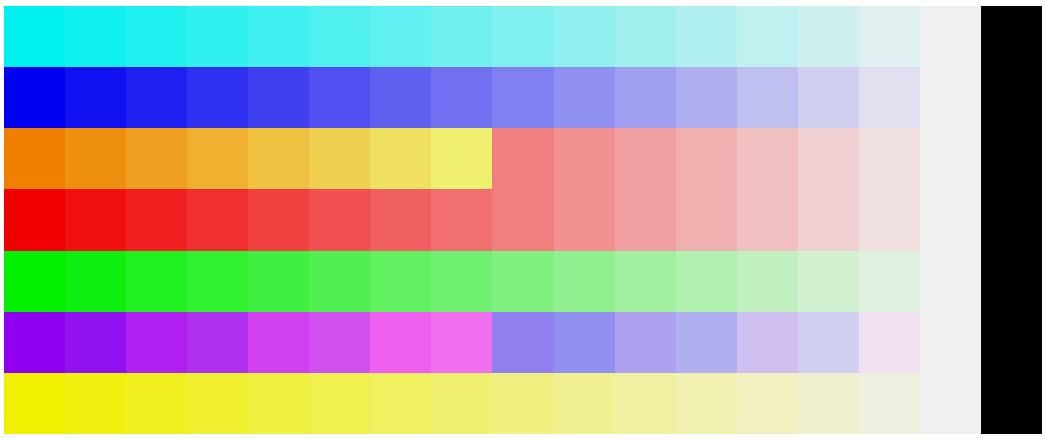
V projektu smo to dosegli tako, da vsaki vrstici pridružimo po en števec. Dolžino tega in s tem tudi njegov obseg štetja smo izbrali glede na osveževalno frekvenco zaslona in zaželeno trajanje efekta. Pri slednjem je bila eksperimentalno določena optimalna vrednost, ki se giblje okoli polovice sekunde. To nam skupaj da števec dolžine 5 bitov in učinek trajajoč približno 33 sličic. Dodatno sličico dobimo zaradi vrstnega reda klicanja tega avtomata in tistega za operacijo padanja v nadrejenem avtomatu.

Zaradi pri vmesniku VGA uporabljenega aditivnega model barv RGB[1] nam najnižja možna vrednost na vseh treh barvnih kanalih da črno barvo, najvišja možna pa belo. Tako lahko za želeni učinek naš števec enostavno združimo skupaj z vsakim od treh barvnih kanalov preko funkcije maksimuma. Rezultat je barva, ki je na začetku štetja enaka originalni, proti koncu pa limitira proti beli. Problem pri tem pa je, da je funkcija maksimuma relativno draga v smislu porabe gradnikov na našem čipu in porabe logičnih vrat pri izdelavi namenskega integriranega vezja. Prav tako ne zah-

[1] https://en.wikipedia.org/wiki/Rgb

tevamo popolne vizualne pravilnosti, saj igralec tako ali tako ne more zaznati manjših nepravilnosti.

Odločili smo se uporabiti funkcijo bitnega `ALI`. Ta nam pri večini barv tetriminov daje popolne rezultate, pri ostalih, kot je npr. oranžna, pa nepravilnosti niso dovolj hude, da bi jih igralec opazil. Na sliki 4.5 je prikazano



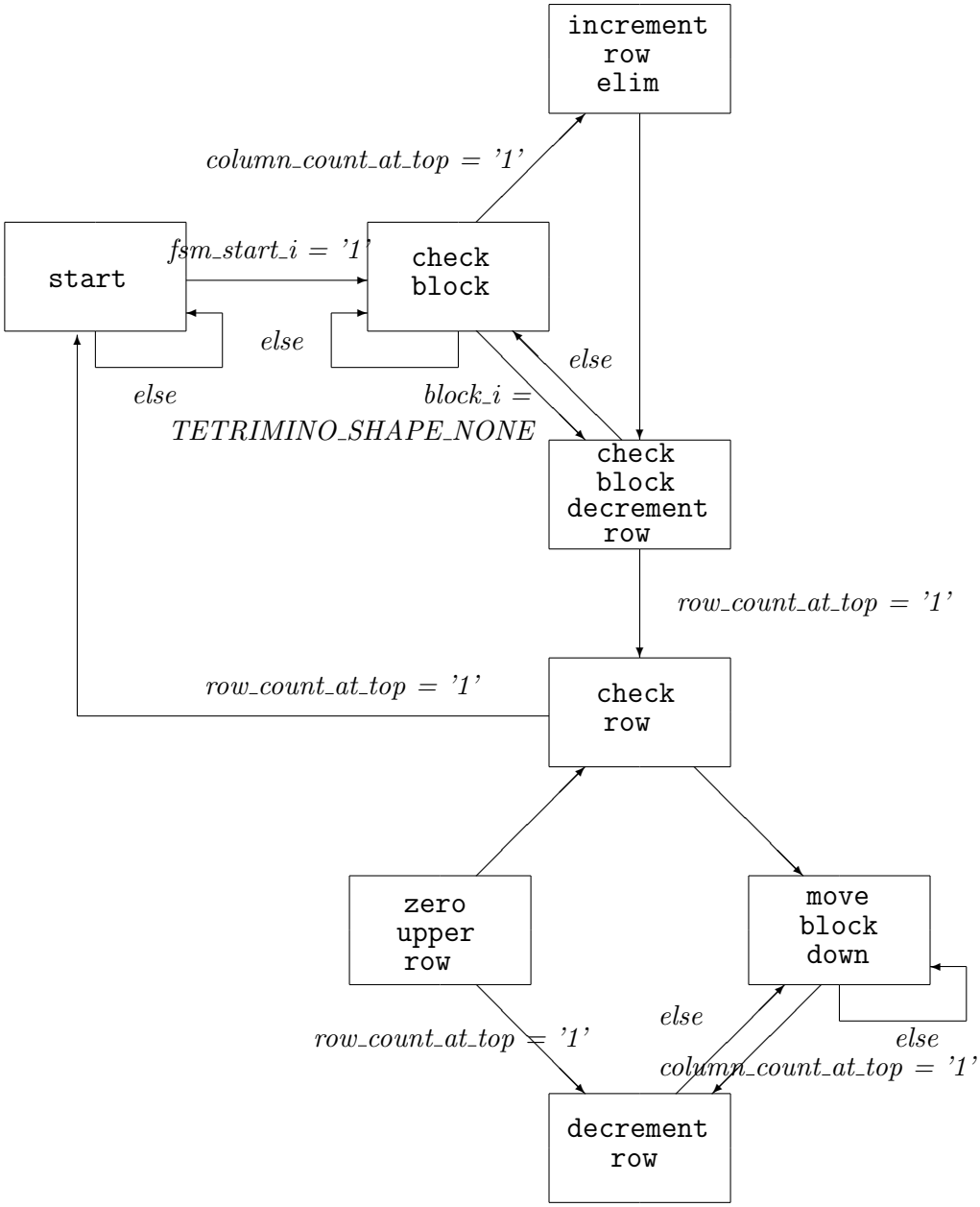
Slika 4.5: Odtenki pri učinku "fade-in" za vseh sedem barv tetriminov

spreminjanje barv na zaslonu med izginjanjem posameznega kvadrata v polni vrstici. Pri tem stolpci prikazujejo barvo v posameznem osvežitvenem ciklu zaslona po začetku odstranjevanja, kjer so na skrajno levi strani slike prikazane originalne barve tetriminov. Z izbiro te funkcije namesto maksimuma smo na čipu xc7a100t privarčevali 4 elemente LUT6.

Opisano zlivanje pridruženega števca z barvo trenutnega kvadrata je implementirano v izrisnem cevovodu.

4.3.2 Krmilnik za odstranitev polne vrstice

Krmilnik za odstranjevanje smo zasnovali kot dvodelni končni avtomat. Prvi del poskrbi za zaznavo polnih vrstic in povečevanje pridruženih števcev za učinek "fade-in". Drugi del skrbi za dejansko odstranitev polne vrstice in premik vseh zgornjih vrstic za eno navzdol. Celotni avtomat je prikazan na sliki 4.6.



Slika 4.6: Končni avtomat za odstranjevanje polnih vrstic. Trojica stanj zgoraj desno predstavlja prvi del, spodnja štiri pa drugi del.

Poleg avtomata smo potrebovali tudi generator naslovov za dostop do igralnega polja in tukajšnjih pridruženih števcev. V ta namen smo uporabili dva števca. Prvi se imenuje `Inst_column_counter` in šteje po stolpcih od prvega do zadnjega, drugi nosi ime `Inst_row_counter` in šteje od najnižje do najvišje vrstice. Ker se naslovi igralnega polja začnejo v zgornjem levem kotu in ker premik vrstic navzdol naravno poteka od spodaj navzgor, ima najnižja vrstica v igralnem polju največji naslov. Zaradi tega števec po vrsticah začne šteti pri vrhnji vrednosti in šteje do ničle.

Za potrebe premika vrstice navzdol smo potrebovali naslov vrstice, ki je za eno večji od trenutne vrednosti števca po vrsticah. Tukaj smo uporabili register, kateremu smo nadeli ime `Inst_reg_old`. Njegov podatkovni vhod je vezan na izhod števca za vrstice, izhod pa je na voljo v signalu `row_count_old`. Vhod za omogočanje ure smo seveda vezali na signal za omogočanje ure števca `row_count_enable`.

Izhod registra `row_count_old` smo povezali na vrstični del izhoda za pisalni naslov igralnega polja `block_write_address_o`. Da bi lahko uspešno premaknili kvadrat navzdol, smo vrstični del izhoda za bralni naslov `block_read_address_o` povezali na vrstični števec. Poleg tega smo del za naslov stolpca pri obeh izhodih vezali na izhod števca stolpcev. Vse te povezave so trajne, saj drugačnega dostopa do igralnega polja ne potrebujemo.

Pri premiku navzdol moramo najvišjo vrstico pobrisati oziroma vanjo vpisati vrednost za prazne kvadrate. V ta namen smo pred izhod `block_o`, ki predstavlja podatkovno pisalno vodilo igralnega polja, postavili multiplekser. Krmilimo ga preko izbiralnega signala `ram_write_data_mux`. Izbirna vrednost `MUXSEL_MOVE_DOWN` povzroči, da se na izhodu pojavi vhod `block_i`, ki predstavlja podatkovno bralno vodilo igralnega polja. Vrednost `MUXSEL_NONE` nam na izhod postavi konstanto `TETRIMINO_SHAPE_NONE`.

Poleg tega smo potrebovali multiplekserje za bralni naslov, pisalni naslov in podatkovno pisalno vodilo pomnilnika vrsticam pridruženih števcev. Te tri krmilimo z izbiralnim signalom `row_elim_mode`.

Tako kot vsi avtomati prej se tudi ta zažene na znak krmilnika dostopa

do igralnega polja, ki ga prejmemo na vhodnem signalu `fsm_start_i`. Na znak čakamo v stanju `state_start`, od koder ob prejemu preidemo v stanje za preverjanje zasedenosti kvadrata `state_check_block`.

Hkrati s čakanjem na znak moramo izrisnemu cevovodu pošiljati vrednost števca, ki je pridružen vrstici na vhodu `row_elim_address_i`. To storimo s preklopom multiplekserjev v stanje `MUXSEL_ROW_ELIM_RENDER`.

Zaznava polne vrstice

Če bi tetris zasnovali kot program, bi zaznava najbrž potekala v obliki dveh zank. Zunanja zanka bi tekla od najnižje do najvišje vrstice, notranja pa po stolpcih. V notranji bi preverjali ali je kvadrat pod tem naslovom prazen in če je temu tako, takoj nadaljevali z naslednjo iteracijo zunanje zanke. Ko bi se notranja zanka končala, bi pridružen števec povečali za ena. Program smo v končnem avtomatu implementirali s pomočjo naslovnih števcev in s tremi stanji:

state_check_block Prvo stanje predstavlja notranjo zanko programa. Tukaj povečujemo vrednost števca stolpcev z aktivacijo signala `column_count_enable` in preverjamo ali je v igralnem polju kaj odloženega pod trenutnim naslovom. V primeru da ni, nadaljujemo v stanje `state_check_block_decrement_row`. V nasproten primeru preverjamo ali je števec stolpcev že prišel do vrha, kar pomeni, da je celotna vrstica zasedena. Ko se to zgodi, preidemo v stanje `state_increment_row_elim`. Če nobeden od teh dveh pogojev ni izpolnjen, ostanemo v tem stanju in preverjamo naslednji kvadrat v vrstici.

state_increment_row_elim Drugo stanje predstavlja preostali del telesa zunanje zanke. in je namenjeno povečevanju vrstici pridruženega števca. To storimo tako, da multiplekserje preklopimo v stanje `MUXSEL_ROW_ELIM_INCREMENT` in omogočimo pisanje v pomnilnik pridruženih števcev s signalom `row_elim_write_enable`. Multiplekserji poskrbijo, da se

na vodilu za pisanje pojavi vrednost iz vodila za branje, povečana za ena. Prav tako bralni in pisalni naslov nastavimo na izhod števca za vrstice. Ob naslednji urini fronti nato brezpogojno preidemo v stanje `state_check_block_decrement_row`.

state_check_block_decrement_row .

Tretje stanje predstavlja zunanjo zanko programa. V njem števcu za vrstice omogočimo uro in ponastavimo števec za stolpce. Istočasno tudi preverimo ali smo že v zadnji vrstici. Če je temu tako, potem nadaljujemo s stanjem `state_check_row` v delu avtomata za odstranitev polne vrstice. V nasprotnem primeru se vrnemo nazaj v `state_check_block`, s čimer začnemo preverjati novo vrstico.

Odstranitev polne vrstice

Podobno kot pri zaznavi, bi tudi program za odstranitev polne vrstice implementirali s pomočjo ugnezdenih zank. Tukaj imamo tri ugnezdene zanke, saj smo registra vezali na isti signal za omogočanje ure kot števec, zaradi česa lahko vrstico naenkrat premaknemo le za eno navzdol. Poleg tega imamo dodatno zanko za brisanje zgornje vrstice.

Prva zanka teče od spodnje do zgornje vrstice in preverja, če je kakšen od pridruženih števcev pri svoji vrhnji vrednosti. Ko se to zgodi, zaženemo dve ugnezdeni zanki za premik vrstic navzdol. Ti dve zanki premikata eno po eno vrstico navzdol, po zaključku katerih znotraj prve zanke nadaljujemo še z zanko za brisanje zgornje vrstice. Prvo zanko sicer zaključimo takrat, ko smo v vrhnji vrstici, vendar nam prekoračitev števca vrstic povzroči, da se zopet znajdemo na začetku. Do prekoračitve pride le, če smo vrstico premikali navzdol, kar se po pravilih naše implementacije igre lahko zgodi največ štirikrat (ko najdaljši tetrimino zaključi štiri vrstice naenkrat).

state_check_row V tem stanju implementiramo zunanjo zanko. Multiplekserje preklopimo v stanje `MUXSEL_ROW_ELIM_INCREMENT`, ki je sicer mišljeno

za prištevanje števcu, vendar se v njem na bralnem naslovu pomnilnika pridruženih števcev pojavi izhod števca vrstic. Istočasno mu omogočimo uro.

Hkrati s tem preverjamo, ali je vrednost števca, pridruženega trenutni vrstici, enaka najvišji možni vrednosti. V tem primeru nadaljujemo s stanjem `state_move_block_down`. Če temu ni tako, preverimo ali je števec vrstic že pri svoji vrhnji vrednosti. Ko se to zgodi smo zaključili z delom in se vrnemo v začetno stanje `state_start`. V nasprotnem primeru ostanemo v trenutnem stanju vse dokler eden od zgornjih pogojev ni izpolnjen.

state_move_block_down V drugem stanju smo implementirali najbolj notranjo zanko, v kateri izvajamo premik kvadrata navzdol. Ker smo naslove za igralno polje že trajno povezali s števcema in registrom, je vklop štetja po stolpcih, pisanja v igralno polje s signalom `block_write_enable_o` in preklop multiplekserja za podatkovno pisalno vodilo v stanje `MUXSEL_MOVE_DOWN` vse kar nam še preostane.

Sočasno preverimo ali je števec po vrsticah prišel do konca. V tem primeru ob naslednji urini fronti preidemo v stanje `state_decrement_row`, drugače pa ostanemo v trenutnem stanju.

state_decrement_row Tretje stanje smo uporabili za izvedbo srednje zanke. Podobno kot prva tudi ta šteje po vrsticah in svoje štetje konča v najvišji. Razlika je v tem, da moramo tukaj poleg sprožitve števca po vrsticah tudi premakniti pridruženi števec za eno navzdol. To storimo z aktivacijo signala `row_elim_write_enable` in preklopom multiplekserjev v stanje `MUXSEL_ROW_ELIM_MOVE_DOWN`. Slednje poskrbi, da se za pisalni naslov v pomnilnik preko registra izbere stara vrednost števca in trenutna vrednost za bralni naslov. Prav tako se podatkovno pisalno vodilo poveže z podatkovnim bralnim vodilom. To povzroči, da se ob naslednji urini fronti vrednost zgornjega števca zapiše v spodnji pridruženi števec.

Hkrati seveda preverimo ali je števec že pri vrhu. V tem primeru preidemo v stanje `state_zero_upper_row`, drugače pa nadaljujemo s premiki navzdol v stanju `state_move_block_down`.

state_zero_upper_row V zadnjem stanju smo implementirali zanko za izbris zgornje vrstice. Zanka je potrebna, saj moramo sedaj v zgornjo vrstico vpisati vrednost, ki predstavlja odsotnost kvadrata. To storimo podobno kot v stanju `state_move_block_down`, le da tukaj preklopimo multiplekser za podatkovno pisalno vodilo igralnega polja v stanje `MUXSEL_NONE`. Načeloma bi tudi tukaj morali izbrisati zgornji vrstici pridružen števec, vendar se zapolnitev zgornje vrstice niti v teoriji niti v praksi ne more zgoditi.

Ob naslednji urini fronti bodisi preidemo v stanje `state_check_row`, če je števec po stolpcih že prišel do konca, bodisi stanja ne spreminjamo.

4.4 Naključno izbiranje naslednjega tetrimina

Pri implementaciji tetrisa kot programa na osebnem računalniku imamo običajno na voljo generator naključnih števil uporabljenega operacijskega sistema. Nasprotno pri razvoju tetrisa kot prosto stoječega programa na mikrokrmilnikih ali v obliki logičnih vrat na FPGA oziroma namenskega integriranega vezja tega nimamo. Prav tako na uporabljenem čipu FPGA ni namenskega logičnega gradnika, ki bi implementiral generator naključnih števil.

Tako smo primorani razviti enoto za izbiro naslednjega tetrimina, kar smo storili v datoteki `tetris_next_tetrimino.vhd`. Sprva smo želeli implementirati pravi generator naključnih števil, ki bi temeljil na posplošenem krožnem oscilatorju za naključnost in linearnem hibridnem celičnem avtomatu za filtriranje dobljenih števil [1]. To smo izpustili, saj je bil namen te diplomske naloge raziskati prenos programske opreme, ki običajno ima na voljo zgornje resurse, v digitalno logiko. Poleg tega bi implementacija generatorja naključnih števil in predvsem njegovo testiranje močno presegalo okvirje te diplomske naloge.

Poleg same izbire naslednjega tetrimina pa ga je potrebno tudi prikazati igralcu, kar smo implementirali v treh delih. V tej datoteki je bila razvita logika za prikaz naslednjega tetrimina v zato namenjenem okvirju,

ki ga izrišemo v izrisnem cevovodu. Tretji del predstavlja napis "Naslednji tetrimino", katerega izpišemo v entiteti za izris znakov.

4.4.1 Izbira naslednjega tetrimina

Izbiranje naslednjega tetrimina oziroma naslednje oblike tetrimina smo implementirali dvodelno. Prvi del predstavlja proces `CYCLE_SHAPES`, ki za novo obliko ob vsaki urini fronti izbere kar naslednjo po vrsti. Pri tem se oblike menjavajo po vrstnem redu identifikatorjev v definicijah, pri čemer se osma, neveljavna, oblika ne more pojaviti. To sicer ni niti blizu kriptografsko varnemu naključnemu izbiranju, vendar je zaradi načina implementacije avtomata za odstranjevanje vrstic in igralčevega naključnega odlaganja tetriminov v igralno polje dovolj dobro za naše potrebe.

Drugi del smo implementirali s flip-flopom tipa D, opisanem v procesu `SAVE_NEW`. Tukaj naslednjo obliko tetrimina shranimo za potrebe prikaza igralcu. Signal za omogočanje ure tega procesa je signal `nt_retrieved_i`, po katerem nam avtomat aktivnega tetrimina sporoči, da je prevzel novo obliko.

4.4.2 Izris naslednjega tetrimina

Izbiri naslednjega tetrimina sledi le še njegov izris. To smo storili podobno kot v procesu za izris aktivnega tetrimina, opisanem v razdelku 4.2. V ta namen smo v procesu `RENDER_NEXT` ustvarili 8 primerjalnikov za istočasno preverjanje vseh štirih parov naslovov vsebovanih kvadratov naslednjega tetrimina z naslovom na zaslonu, ki ga predstavlja vhodni signal `render_address_i`.

Da bi imeli ta naslov s čim primerjati, je bilo potrebno najprej shraniti naslove vsebovanih kvadratov. Te pridobimo iz bralnega pomnilnika odmikov `tetrimino_init_rom`, ki ga naslovimo z obliko naslednjega tetrimina in njegovo orientacijo. Za slednjo smo izbrali kar konstantno orientacijo `TETRIMINO_ROTATION_90`, saj ta podatek ni tako pomemben za igralca.

Za razliko od krmilnika za aktivni tetrimino je bilo tukaj možno lokacijo prikaza izbrali tako, da je potrebno preveriti le dva bita v naslovu. Izbrali

smo naslov okvirja, ki je poravnan na štiri kvadrate v obeh razsežnostih zaslona. Tako sta spodnja dva bita naslova kvadrata kar enaka odmikom v bralnem pomnilniku oziroma z dobljenimi odmiki ni potrebno računati. Zaradi tega smo naslove vsebovanih kvadratov v procesu `SAVE_ADDRESSES` shranili v osem registrov po dva bita, ki smo jih predstavili s signali `rows` in `cols`.

Na izhodni signal `render_shape_o` postavimo bodisi vrednost signala `tetrimino_shape_next` v primeru, ko eden od parov primerjalnikov zazna ujemanje, bodisi vrednost `TETRIMINO_SHAPE_NONE`, ko tega ujemanja ni.

4.5 Izris znakov

Za izris znakov smo se odločili uporabiti font, pri katerem je velikost posameznih znakov oziroma njihovih glifov enaka 16 točk v višino in 8 točk v širino. Razlog za to leži v preprosti preslikavi med naslovom točke na zaslonu, ki ga dobimo iz krmilnika VGA, in naslovom posameznega znaka. Zaradi dejstva, da sta širina in višina glifa potenci števila 2, je ta naslov preprosto zgornji del bitnega vektorja, v katerem je podan naslov točke na zaslonu. To velja tako za vodoravni kot tudi za navpični odmik. Pri standardni ločljivosti zaslona VGA, ki znaša 640x480 [13], tako dobimo na voljo $\frac{640}{8} = 80$ znakov v širino in $\frac{480}{16} = 30$ znakov v višino, kar nam da skupno $80 \times 30 = 2400$ znakov.

Pri 19 palčnem zaslonu, katerega smo uporabili za razvoj te igre, bi uporaba večjega ali manjšega fonta, pri katerem je velikost stranice v točkah potenca števila 2, pomenila občutno prevelike črke v prvem in premajhne v drugem primeru.

V primeru uporabe fonta, pri katerem stranica posamezne črke ni potenca števila 2, tovrstna preprosta preslikava ni več mogoča. Za zgled lahko vzamemo font velikosti 12, ki je predpisan s strani Fakultete za računalništvo in informatiko za diplomska dela [23]. Pri tem privzamemo velikost zaslona 72 točk na inč (angl. DPI). Ker so velikosti fontov podane v točkah namiznega založništva (angl. DTP oz. desktop publishing point oz. PostScript point),

pri katerem je velikost ene točke enaka $\frac{1}{72}$ inča [22], je višina posamezne črke natanko 12 točk na zaslonu. Za preračun naslova točke v naslov črke bi sedaj imeli na voljo več možnosti:

- Naslov točke bi lahko delili z 12. V tem primeru bi morali razviti delilno vezje, kar ni enostavno v smislu porabe logičnih elementov. Vsekakor bi ga potrebovali, če bi dopustili prilagajanje velikost črk med delovanjem skupaj s skaliranjem celotne igre glede na velikost zaslona.

- Ker vemo, da naslov točke vedno narašča s korakom 1 v intervalih, ki so točno določeni s točkovno uro (angl. pixel clock), bi lahko imeli dva dodatna števca. Prvi bi štel s hitrostjo točkovne ure in bi določal naslov točke v predstavitvi znaka, drugi pa bi bil vezan na prekoračitev obsega prvega in bi štel naslov znaka na zaslonu. Ta postopek zahteva veliko manj resursov na samem čipu kot zgornja a še vedno več kot direktna uporaba naslova točke iz modula VGA. Kljub temu bi rezultat bil enak: prvi števec predstavlja ostanek, drugi pa količnik pri deljenju z 12.

4.5.1 Kodiranje znakov

Po odločitvi o velikosti črk smo jih morali še ustrezno predstaviti v bralnem pomnilniku. Na voljo je bilo več različnih načinov zapisa posameznih znakov [17], od katerih smo v ožji izbor vzeli naslednja dva:

- Bitni (angl. bitmap) oziroma rasterski fonti. To je najbolj enostavna predstavitev črk, saj so zapisane v obliki, ki je enaka njihovi končni predstavitvi na zaslonu. Ključna slabost take predstavitve je v zahtevnosti spreminjanja velikosti in posledično slabemu izgledu dobljenih črke [17].

- vektorski (angl. vector) ali obrisni (angl. outline) fonti. Ta oblika zapisa je prevladujoča na sodobnih računalnikih in za zapis izgleda črke uporablja Bezierjeve krivulje. Glavna prednost je, da lahko črko lepo skaliramo na poljubno velikost. Od tu sledi glavna slabost in sicer, da

moramo črko pred izrisom na zaslon pretvoriti v bitno sliko. Slednjo lahko po spremembi velikosti naenkrat izračunamo za vse črke in ta vmesni zapis predpomnimo v delovnem pomnilniku.

Ker nismo imeli namena spreminjati velikosti črk med delovanjem igre, smo se odločili font v bralnem pomnilniku shraniti kar v obliki bitnih slik ustrezne velikosti. Naredili smo kratek pregled dostopnih fontov, med katerimi sta bila paketa črk oz. fonta GNU Unifont in UNI-VGA na voljo v obliki bitnih slik ravno prave velikosti. Za izris znakov smo se odločili uporabiti slednjega, saj ima prvi poleg glifov nam ustrezne velikosti 8x16 tudi veliko glifov velikosti 16x16.

4.5.2 Paket UNI-VGA in datotečni format BDF

Za izris znakov smo uporabili paket UNI-VGA oziroma Unicode VGA. Paket je sicer namenjen uporabi v tekstovni konzoli operacijskega sistema in v programskemu paketu XDosEmu [16], vendar pa je zaradi črk v obliki 8x16 bitnih slik primeren za takojšnjo integracijo.

Font je distribuiran v datotečnem formatu BDF za distribuiranje bitnih slik glifov (angl. Adobe Glyph Bitmap Distribution Format) podjetja Adobe [19]. V glavi datoteke `u_vga16.bdf`, v kateri so shranjene vse bitne slike fonta, se pred prvo direktivo `STARTCHAR` nahajajo splošne informacije kot npr. `SIZE 16 75 75`, kar nam pove, da je to font velikosti 16 za zaslon ločljivosti 75x75 točk. Za tem se nahajajo opisi posameznih črk.

Med njimi za malo črko `a` najdemo vrstice `STARTCHAR a`, ki nam pove, da se začenja glif za to črko. Temu sledi `ENCODING 97` oziroma podatek, da ima črka a v tem kodiranju decimalni indeks 97 (kar je enako kot črka a v računalništvu prevladujočem kodiranju ASCII [18]) Naslednja za nas pomembna informacija `DWIDTH 8 0` nam pove, da se naslednja (desna) črka začne čez 8 točk vodoravne in 0 točk navpične razdalje oziroma, da je glif te črke velikosti 8x16 [20].

V paketu so vsi glifi zahtevane velikosti 8x16, kar nam olajša rokovanje

pri izrisu. Same slike črk so zapisane v ASCII zapisu v šestnajstih vrsticah po dva šestnajstiška znaka.

4.5.3 Prenos zapisa glifov iz BDF v VHDL

Direktno shranjevanje glifov v formatu BDF ne bilo smiselno, saj bi datotečni format in vezje za njegovo branje zavzela mnogo več prostora kot bitni zapis potrebnih znakov. Zato smo pretvorbo glifov v primernejši zapis naredili vnaprej. Odločili smo se vsak glif posebej pretvoriti v svojo konstanto jezika VHDL in jih nato shraniti v paket `uni_vga`, vsebovan v datoteki `uni_vga.vhd`.

Seveda je to datoteko pred sintezo potrebno ustvariti iz `.bdf` datoteke fonta. V ta namen smo predelali orodje *bdf2c* [21] v *bdf2vhdl*, katerega izhod je želen paket (angl. package) v jeziku VHDL. Ta paket nadomesti vključitveno in izvorno datoteko v programskem jeziku C, za ustvarjanje katerih je namenjeno prvotno orodje. Na začetku paketa smo najprej definirali nekaj osnovnih tipov in konstant, odvisnih od lastnosti pretvorjenega fonta:

```
constant character_width  : positive := 8;
constant character_height : positive := 16;
constant number_of_characters : natural  := 2899;
subtype pixel_row is std_logic_vector (0 to character_width - 1);
type pixel_rows is
  array (0 to character_height - 1) of pixel_row;
```

Temu sledijo za vsak glif posebej definirane konstante tipa `pixel_rows`, ki predstavlja seznam vrstic tipa `std_logic_vector`. Tako za vsako črko dobimo poizvedbeno tabelo, naslovljeno sprva po vrsticah nato po stolpcih. Pri tem smo začetek črke postavili v zgornji levi kot, tako kot začetek štetja vrstic in stolpcev pri razvitem krmilniku VGA.

4.5.4 Shranjevanje črk

Za primer morebitne kasnejše zamenjave fonta smo se odločili, da identifikatorje posameznih črk v projektu predstavimo v fontu nevtralni obliki. Izbrali smo predstavitev s pri nas definirano enumeracijo, saj podatkovni tip `Character`, definiran v paketu `std.standard` jezika VHDL [3], ni zadosti za predstavitev vseh možnih znakov izbranega fonta. Prav tako jezik VHDL ne podpira zapisa črk v kodiranju Unicode oziroma kateremu drugemu od razširjenih zapisov.

Enumeracijo smo definirali s tipom `object` v paketu `definitions.letter` in v njo vstavili vse potrebne identifikatorje znakov. Primer vsebovanega identifikatorja je npr. `letter.N_upper` za veliko pisano črko n. Tukaj smo bili primorani dodati pripono `_upper` (lahko bi izbrali tudi kaj drugega), saj jezik VHDL ne loči med velikimi in malimi črkami v identifikatorjih [3] in je tako zapis `letter.n` ekvivalenten npr. zapisu `LeTTer.N`.

Celoten font smo shranili v bralnem pomnilniku z enobitnim izhodom, saj na zaslon naenkrat pošljemo le eno točko. Pomnilnik smo definirali s konstantnim seznamom `font_data` v paketu `definitions.font` in predstavlja preslikavo med enumeracijo črke in posameznimi glifi fonta. Poizvedbo po njem izvedemo s pomočjo sintezibilne funkcije `get_dot`, definirane v istem paketu in prikazane na sliki 4.7. Zasnovali smo jo tako, da pomnilnik na-

```
function get_dot (l : in letter.object;
                  r : in row.object; c : in col.object
                 ) return std_logic is
  constant prows : data.pixel_rows := font_data (l);
  constant prow  : data.pixel_row  := prows (to_integer (r));
begin
  return prow (to_integer (c));
end get_dot;
```

Slika 4.7: Funkcija za preslikavo črk v posamezno vsebovano točko

slovimo prvo s črko, nato z naslovom znotraj znaka. Pri tem je identifikator črke vsebovan v parametru `l`, s katerim naslovimo pomnilnik `font_data`. Pridobljeni element `prows` je pomnilnik glifa, ki ga nato naslovimo z vrstico (4 biti) in stolpcem (3 biti). Končni rezultat je element tipa `std_logic`, ki nam pove ali se pod trenutnim naslovom nahaja del črke ali ne.

4.5.5 Izris znakov

Napise oziroma položaje črk na zaslonu smo definirali v entiteti `tetris_text`, vsebovani v datoteki `tetris_text.vhd`. Instanca te entitete je vsebovana v izrisnem cevovodu, iz katerega pridobimo naslovna vektorja `s0_read_address_i` in `s1_read_subaddress_i`.

Za shranjevanje položaja črke smo prvo potrebovali podatkovne tipe za hrambo naslovov vrstic in stolpcev. Te smo definirali v paketu `letters`, vsebovanem v paketu `definitions`. Nadaljevali smo z definicijo podatkovnega tipa `storage_object` za predstavitev dvorazsežnega seznama črk na zaslonu:

```
type storage_object is array
  (0 to 2**letters.row.width-1, 0 to 2**letters.col.width-1) of
  letter.object;
```

Podatkovni tip smo nato uporabili za definicijo konstantnega pomnilnika napisa “Naslednji tetrimino”, ki smo ga umestili levo od prostora za prikaz naslednje tetrimine. Pri tem smo prazne prostore zapolnili z konstanto `letter.space`.

Izris črke poteka v treh delih, od katerih sta dva implementirana tukaj, zadnji pa v izrisnem cevovodu. V prvem delu z naslovom iz prve stopnje cevovoda, vsebovanem v vhodnem signalu `s0_read_address_i`, naredimo poizvedbo v zgornjem pomnilniku. Dobljeno črko nato ob naslednji urini fronti shranimo v registru `s1_read_letter`. V drugem delu jo skupaj z naslovom `s1_read_subaddress_i`, ki določa točko znotraj glifa, uporabimo za klic funkcije `get_dot`. Rezultat pošljemo v izrisni cevovod preko signala `s1_read_dot_o`.

4.6 Cevovodni izris na zaslon

Cevovodni izris smo implementirali v datoteki `tetris_render_pipeline.vhd` in predstavlja strojno nevtralno krovno entiteto našega projekta, ki je podrejena entiteti `top_level`, definirani za vsaki plošči posebej. Cevovodenje je tukaj potrebno zato, ker imamo od prejema novih zaslonskih naslovov iz krmilnika VGA do izbire barv zelo dolgo kombinatorično pot. Poleg tega Xilinx priporoča uporabo cevovodenja za izboljšanje zmogljivosti vezja, kar dosežemo s pomočjo na čipu vsebovanega izobilja flip-flopov [5].

Drugi problem, ki omilimo s cevovodno izvedbo, je problem preveč bralnih bremenitev istega registra. Vsak tranzistor (s katerimi je implementirana vsa logika) na vodniku predstavlja kapacitivno breme in se "upira" spremembi logičnega stanja. Več elementov kot imamo povezanih na dani vodnik, več časa in energije potrebuje premik v del napetostnega razpona, ki predstavlja nasprotno logično vrednost. Seveda to pomeni daljši čas razširjanja signala in nam omejuje dosegljivo urino frekvenco. Problematični vodili sta tukaj naslovna vektorja točke na zaslonu, ker moramo nanju povezanih veliko primerjalnikov za določitev ali je izris posameznih zaslonskih elementov igre potreben ali ne.

To smo omilili tako, da smo te primerjalnike razporedili po več stopnjah cevovoda. V prvih dveh stopnjah tako v posameznih modulih določamo izhode za izris, v predzadnji pa izračunamo preostanek izbiralnih signalov za glavni multiplekser. S tem smo sicer dodali bremena v obliki registrov, vendar smo hkrati bistveno zmanjšali število bremen na registrih, ki hranijo naslovne vektorje.

V naslednjih podrazdelkih bomo opisali delovanje cevovoda. Za razliko od prejšnjih razdelkov, kjer smo delovanje vsega opisovali zelo podrobno, bomo tukaj le opisali, kaj se v posameznih stopnjah cevovoda dogaja. V nasprotnem primeru bi sledilo vsaj deset strani mukotrpnega navajanja vseh signalov. Prav tako ne bomo navajali katere signale shranjujemo na meji med stopnjama.

Zaradi izobilja flip-flopov in ker nismo omejeni s časom, smo si za glavni

kriterij pri razporejanju opravil po posameznih stopnjah izbrali omenjeno zmanjšanje bremenitev posameznih vodil. Tega nismo storili popolno, temveč smo se ustalili na pet stopenjskemu cevovodu, pri čemer se številčenje teh začne od prvih registrov. Seveda lahko signale uporabimo tudi pred prvo stopnjo, kar smo tudi storili.

4.6.1 Pred stopnja

Vhod v naš cevovod so vsi izhodi krmilnika VGA: naslov na zaslonu, signal za omogočanje izrisa, signal za konec izrisa in sinhronizacijska signala. Tukaj povežemo zgornji del naslovnega vektorja s pripadajočim vhodom entitete za izris črk.

4.6.2 Prva stopnja

V tem stanju naslovna vektorja povežemo z entiteto za določitev naslednjega tetrimina in s krmilnikom igralnega polja. V slednjega pošljemo vse razen spodnjih štirih bitov, s čimer smo definirali velikost kvadratov, ki sestavljajo tetrimino. Ta velikost je tako 16×16 slikovnih točk. Podrejeni moduli za določitev izhodov ne potrebujejo spodnjega dela teh dveh vektorjev, zato smo ga tukaj povezali tudi z vhodom za spodnji del naslova entitete za izris črk, zaradi vsebovanega dela cevovoda pričakuje naslov, zakasnjen za eno urino periodo. Ob koncu te urine periode na podlagi podanih naslovov iz podrejenih entitet pridobimo:

- Signala za "Obliki" kvadratov znotraj igralnega polja in aktivnega tetrimina, s katerima je njuna barva enolično določena.
- Bit, ki določa ali se pod trenutnim naslovom nahaja črka ali ne.
- Zgornjemu delu trenutne vrstici pridružen števec. Pri tem nas zanima le levi del števca, ki je enak dolžini najdaljšega vektorja barvne komponente (4 bite).

Ker smo modul za naslednji tetrimino vključili tukaj namesto v hierarhiji krmilnika igralnega polja, smo s slednjim povezali signal za naslednjo obliko in kontrolni signal za prevzem. Preostane nam le še povezati modul igralnega polja z nadrejenim modulom preko sledečih povezav:

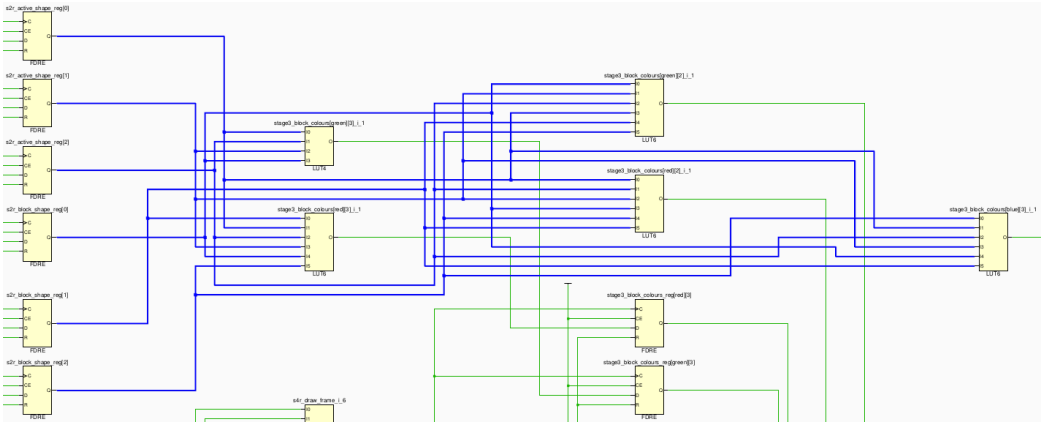
- signala za aktivno akcijo iz krmilnika akcij in tisti za potrditev le-te,
- signal za konec zaslonskega izrisa oziroma pričetek izvajanja opravil iz krmilnika VGA,
- binarno zakodirano desetiško število točk za izris na 7-segmentnemu prikazovalniku.

4.6.3 Druga stopnja

V drugi stopnji iz shranjenih "oblik" kvadrata iz igralnega polja in aktivnega tetrimina izračunamo barvo. Ker se ne moreta prekrivati, smo prvo z multipleksiranjem teh dveh pridobili novo "obliko". Za to vzamemo bodisi tisto od kvadrata aktivnega tetrimina, če je različna od `TETRIMINO_SHAPE_NONE`, bodisi od kvadrata, vsebovanega v igralnem polju. Nato preko funkcije `definitions.get_colour` pridobimo barvo v zapisu RGB.

Opisano zaradi arhitekture uporabljenega čipa predstavlja najbolj učinkovito izrabo poizvedbenih tabel, saj se za izračun vsakega bita barve funkcija in multiplekser zapišeta v le eno poizvedbeno tabelo. Pri tem trije naslovni vhodi predstavljajo "obliko" enega kvadrata in preostali trije drugega. Na sliki 4.8 je prikazanih pet poizvedenih tabel (štiri LUT6 in ena LUT4) za določitev barve. Ostale je zaradi podobnosti v dvojiškem zapisu osmih uporabljenih barv sintezno orodje optimiziralo stran.

Poleg tega smo zaradi zmanjšanja bremen na naslovnih vodilih šele tukaj povezali signale izrisnega dela modula za naslednji tetrimino. Ti predstavljajo naslov na zaslonu in na podlagi teh pridobljeno njegovo obliko.



Slika 4.8: Shema vezja izrisnega cevovoda za izračun barve tetrimina. Trije vodniki na skrajno levi strani predstavljajo obliko aktivnega tetrimina, spodnji pa "obliko" odloženega kvadrata v igranem polju.

4.6.4 Tretja stopnja

Tretja stopnja je uporabljena za izračun krmilnih signalov izrisnega multiplekserja, za katerega moramo poleg iz krmilnika VGA pridobljenega signala za omogočanje izrisa in signala za omogočanje izrisa slikovne točke v črki izračunati še:

- signal za omogočanje izrisa okvirja okoli izrisne površine zaslona in igralnega polja,
- signal za omogočanje izrisa elementov v igralnem polju,
- signal za omogočanje izrisa okvirja okoli naslednjega tetrimina in omogočanje prikaza le-tega.

Preostane nam le še izračun barv naslednjega tetrimina in združevanje barv iz prejšnje stopnje z vrstici pridruženim števcem za potrebe učinka "fade-in".

4.6.5 Četrta stopnja

Pred to stopnjo smo že izračunali vse, kar je potrebno za izris na zaslon. Zato smo sem umestili glavni izrisni multiplekser. Sicer se zaradi razporeditve elementov nobena dva elementa ne moreta prekrivati, vseeno pa se lahko prekrivajo s svojimi okvirji. V multiplekserju nato izrisujemo po naslednjem vrstnem redu:

1. Izrisna površina. Če se ne nahajamo na tej, potem moramo zaradi standarda VGA na barvni izhod poslati črno barvo.
2. Okvir zaslona. V primeru aktivnega signala za izris zaslonskega okvirja pošljemo na izhod svetlo vijolično barvo.
3. Okvir naslednjega tetrimina. Tu smo uporabili svetlo modro.
4. Izris črk. Če je aktiven signal za zaslonsko piko, potem na izhod pošljemo sivo barvo.
5. Polje tetrisa. Za izhod izberemo v prejšnji stopnji dobljen rezultat združevanja pridruženega števca z barvo v trenutni točki.
6. Naslednji tetrimino. Na izhod pošljemo v tretji stopnji izračunano barvo.
7. Vse ostalo. Podobno kot na začetku, tukaj izrišemo črno barvo.

V vseh razen v prvem primeru so barve bile izbrane poljubno. Za tetrimine smo uporabili barve, standardizirane s strani “The Tetris Company” [12].

4.6.6 Peta stopnja

Peto stopnjo potrebujemo za sinhronizacijo izhodnih signalov. Tukaj shranimo sinhronizacijska signala, ki smo ju nespremenjena peljali čez vse stopnje in malo prej izbrano končno barvo oziroma izhod glavnega multiplekserja.

Poglavje 5

Sklepne ugotovitve

Razvita je bila računalniška igra tetris v jeziku za opis strojne opreme VHDL, brez uporabe procesorjev. Pri tem so bile razvite osnovne funkcije igre, kot so premiki, vrtenje in padanje. Opisi datotek v projektu so na voljo v dodatku A. V nadaljevanju bi bilo primerno izvesti še sledeče izboljšave in razširitve:

1. Glasbena spremljava

   Na razvojni plošči Nexys4 se nahaja tudi konektor za izhod zvoka in podporno vezje za pretvorbo digitalnega izhoda čipa FPGA v analogni signal [8]. Za glasbeno spremljavo bi bilo potrebno razviti vezje za pulzno širinsko modulacijo, ki bi bralo s strani igre določeni digitaliziran zvočni zapis v bralnem pomnilniku. Tako bi lahko imeli melodijo za neprekinjeno spremljavo, ki bi jo prekinila spremljava za brisanje vrstice ali spremljava pri položitvi aktivnega tetrimina na končno mesto v igralnem polju.

2. Dinamično prilagajanje velikosti zaslona.

   Poleg vodnikov za tri barve, dva sinhronizacijska signala in pripadajoče ozemljitve imamo na konektorju VGA tudi vodnika SDA in SCL, ki sta uporabljena za vodilo i2c. Na tem vodilu se običajno nahaja pomnilno vezje EEPROM, v katerem je shranjena podatkovna struktura EDID, v kateri so krmilniku VGA na voljo podatki o priključenem zaslonu.

Med temi se nahajajo tudi podatki o podprtih ločljivostih, osveževalnih frekvencah in pripadajočih časovnih parametrov.

3. Moderni vmesnik za zaslon.

   Skupaj s podporo prilagajanja ločljivosti priključenemu zaslonu, bi lahko za naslavljanje večjih zaslonskih površin in višje kvalitete pri prenosu uporabili enega od digitalnih protokolov za naslavljanje zaslona:

   - DVI
   - DisplayPort
   - HDMI

   V vsakem primeru bi takšna razširitev zahtevala drugo razvojno ploščo, saj Nexys4 nima nobenega od naštetih priključkov. Prav tako nimamo splošno namenskih visokofrekvenčnih razširitvenih vodil, preko katerih bi lahko priklopili razširitvene kartice.

4. Izris števila točk in zgodovine prvih deset največjih dosežkov.

   Izris igralnega polja in naslednjega tetrimina nam ne zavzame celotnega zaslona, temveč imamo na voljo še nekaj prostora. V ta prostor bi lahko namesto uporabe sedem segmentnega prikazovalnika vrisali še trenutno število točk in prvih deset največjih dosežkov. Sledeče bi zahtevalo razvoj avtomata za sortiranje desetih pomnilniških lokacij

5. Izdelava načrta namenskega integriranega vezja

   V ta namen poleg komercialnih obstaja prosto dostopna programska oprema Qflow. Težava pri uporabi le-te je v tem, da podpira le mnogo preprostejši jezik za opis strojne opreme Verilog, v katerega bi bilo potrebno prevesti našo igro. Zaradi zamudnosti in možnosti logičnih napak pri ročnem prevajanju obstaja orodje *vhdl2verilog*, ki nam to delo olajša, vendar to orodje zaenkrat ne podpira celotnega nabora konstruktov jezika VHDL [24]. Med temi je najbolj pomemben podatkovni tip `record`, ki je v uporabi po celotnem projektu.

Dodatek A

Opis datotek v projektu

FPGA_projects – repozitorij

library/

- basic/
  - **basic.vhd** – paket z osnovnimi pripomočki
  - **counter.vhd** – osnovni števec navzgor
  - **flip_flop_jk.vhd** – flip-flop tipa JK (jump-kill)
  - **generic_register.vhd** – seznam flip-flopov tipa D (register)
- **counter_until.vhd** – števec po poljubnem modulu
- **seven_seg_digit.vhd** – pretvorba 4-bitne vrednosti v segmente 7-delnega zaslona
- **seven_seg_display.vhd** – krmilnik 7-delnega LED prikazovalnika
- **tactile_button.vhd** – krmilnik tipke
- **tactile_buttons.vhd** – seznam krmilnikov tipk
- **util.vhd** – pomožne funkcije
- vga/

  - **sync_generator.vhd** – generator posameznega sinhronizacijskega signala in pripadajoča logika
  - **VGA_controller.vhd** – krmilnik VGA
  - **vga.vhd** – definicija paketa s krmilnikom VGA

tetris/

- board/
  - Nexys4/
    - **definitions.vhd** – definicije podatkovnih tipov in njihove konfiguracije
    - **pin_map.xdc** – preslikava V/I vodnikov krovne datoteke na V/I bloke
    - **top_level.vhd** – krovna datoteka projekta, preslikava lastnosti strojne opreme na logiko igre
- **README** – navodila za sintezo
- **tetris_active_tetrimino.vhd** – krmilnik aktivnega tetromina
- **tetris_block.vhd** – krmilnik igralnega polja
- **tetris_render_pipeline.vhd** – izrisni cevovod
- **tetris_next_tetrimino.vhd** – logika za določanje in izris naslednjega tetromina
- **tetris_row_elim.vhd** – logika za odstranjevanje polnih vrstic
- **tetris_text.vhd** – logika za izris znakov na zaslon
- **uni_vga.vhd** – paket z glifi znakov

bdf2vhdl – repozitorij pomožnega programa

- **bdf2vhdl.c** – program za pretvorbo datotečnega formata BDF v paket VHDL

- **Makefile** – Makefile za prevedbo programa bdf2vhdl v izvršljivo kodo

- **README.txt** – opis programa

Literatura

[1] C. Baetoniu. "High Speed True Random Number Generators in Xilinx FPGAs." [Online]. Dosegljivo: http://forums.xilinx.com/xlnx/attachments/xlnx/EDK/27322/1/HighSpeedTrueRandomNumberGeneratorsinXilinxFPGAs.pdf. [Dostopano 12. 8. 2016].

[2] P. Bulić. *Načrtovanje in sinteza digitalnih in mikroprocesorskih sistemov v FPGA, 1. del, 2011* [Online]. Dosegljivo: https://ucilnica1415.fri.uni-lj.si/mod/resource/view.php?id=12351. [Dostopano 15. 10. 2014].

[3] P. J. Ashenden. *The Student's Guide to VHDL*. Morgan Kaufmann Publishers, 2008.

[4] 7 Series FPGAs Overview, Product Specification. [Online]. Dosegljivo: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf. [Dostopano 11. 8. 2016].

[5] 7 Series FPGAs Configurable Logic Block, User Guide. [Online]. Dosegljivo: http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf. [Dostopano 25. 9. 2016].

[6] 2016 General Europractice MPW runs: Schedule and Prices. [Online]. Dosegljivo: http://www.europractice-ic.com/docs/160826_MPW2016-general-v16.pdf. [Dostopano 2. 8. 2016].

[7] Nexys 4 Reference Manual. [Online]. Dosegljivo: https://reference.digilentinc.com/reference/programmable-logic/nexys-4/reference-manual. [Dostopano 4. 8. 2016].

[8] Nexys 4^{TM} FPGA Board Reference Manual. [Online]. Dosegljivo: https://reference.digilentinc.com/_media/reference/programmable-logic/nexys-4/nexys4_rm.pdf. [Dostopano 4. 8. 2016].

[9] Hardware description language. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Hardware_description_language. [Dostopano 28. 9. 2016].

[10] Vivado Design Suite. [Online]. Dosegljivo: http://www.xilinx.com/products/design-tools/vivado.html. [Dostopano 12. 7. 2016].

[11] Tetromino. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Tetromino. [Dostopano 30. 8. 2016].

[12] Tetris. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Tetris. [Dostopano 30. 8. 2016].

[13] Computer display standard. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Computer_display_standard. [Dostopano 21. 8. 2016].

[14] Video Graphics Array. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/VGA. [Dostopano 5. 9. 2016].

[15] GNU Unifont Glyphs. [Online]. Dosegljivo: http://www.unifoundry.com/unifont.html. [Dostopano 20. 8. 2016].

[16] Unicode VGA font. [Online]. Dosegljivo: http://www.inp.nsk.su/∼bolkhov/files/fonts/univga. [Dostopano 20. 8. 2016].

[17] Computer font. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Computer_font. [Dostopano 20. 8. 2016].

[18] ASCII. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/ASCII. [Dostopano 3. 9. 2016].

[19] Glyph Bitmap Distribution Format. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Glyph_Bitmap_Distribution_Format. [Dostopano 20. 8. 2016].

[20] Glyph Bitmap Distribution Format (BDF) Specification. [Online]. Dosegljivo: https://www.adobe.com/content/dam/Adobe/en/devnet/font/pdfs/5005.BDF_Spec.pdf. [Dostopano 20. 8. 2016].

[21] bdf2c - BDF Font to C source convertor. [Online]. Dosegljivo: https://sourceforge.net/projects/bdf2c. [Dostopano 20. 8. 2016].

[22] Point (typography). [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Traditional_point-size_names. [Dostopano 20. 8. 2016].

[23] Izdelava diplomske naloge, Navodila študentom pred diplomo na prvostopenjskih študijih. [Online]. Dosegljivo: http://www.fri.uni-lj.si/file/190523/2016_maj_navodila-za-diplomsko-delo.pdf. [Dostopano 20. 8. 2016].

[24] VHDL To Verilog Translator. [Online]. Dosegljivo: http://www.edautils.com/vhdl2verilog.html. [Dostopano 7. 9. 2016].