DIVS 2016/17 – Laboratorijski projekt

Grafična kartica – GPU

Michel Adamič

Februar 2017

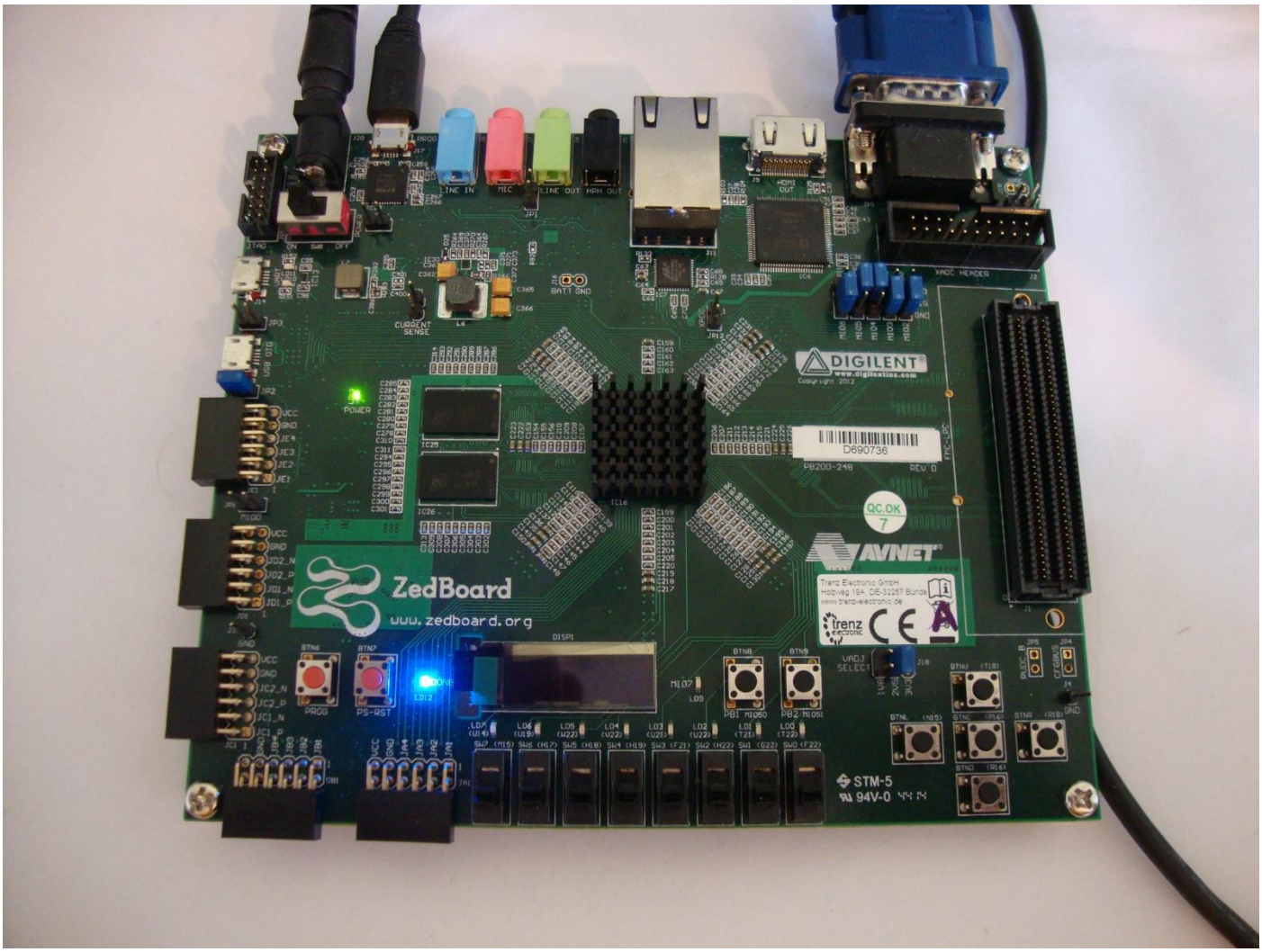
Uvod

Za zaključni projekt pri predmetu Digitalna integrirana vezja in sistemi sem se odločil na FPGA narediti zelo enostavno grafično kartico. Ideja je v tem, da imamo na FPGA implementirano posebno vezje, ki skrbi za celotno računalniško grafiko (kreiranje slike in njeno izrisovanje), CPE pa se torej s tem ni treba ukvarjati. Na CPE teče aplikacija, ki prek ustreznih gonilnikov grafični kartici pove, kaj naj se izriše na zaslon, implementirano vezje (GPU – graphics processing unit) pa potem to izvrši.

Sistem sem razvil na plošči *ZedBoard*, ki uporablja zmogljiv SoC Zynq-7000 od proizvajalca Xilinx. Čip pod istim pokrovom vsebuje dvojedrni ARM Cortex-A9 procesor in programirljivo logiko (FPGA). Moje grafične kartice sicer ne moremo primerjati s pravimi tovrstnimi zadevami v računalniku, saj je tam namen GPU, da razbremeni CPE ter naredi sistem veliko hitrejši in učinkovitejši (strojno pospeševanje). V mojem primeru bi izris grafike verjetno hitreje opravila sama CPE, ker je dvojedrni ARM procesor zelo zmogljiv, implementirana GPU pa nadvse primitivna. Gre bolj za demonstracijo koncepta, kjer imamo poleg CPE še namensko vezje, specializirano za določeno nalogo, programska oprema pa je prek gonilnikov ločena od strojne opreme.

Projektna naloga, ki je nastala, je precej obširna. Zato se v tem poročilu ne bom preveč spuščal v detajle, ker bi tekst postal zelo nepregleden, porabil pa bi tudi ogromno časa. Za podrobnosti mora bralec odpreti kodo, ki ni revna s komentarji. Gre bolj za predstavitev vezja, na nivoju blokovnih shem, s poudarkom na funkcionalnosti, kako zadeva deluje in kako se jo uporablja. Najprej se bom lotil jedra naloge, torej implementiranega vezja na FPGA, v katerega sem vložil večino časa in truda. Opisoval bom od spodaj navzgor (od posameznega vezja do celotnega sistema), tako kot je projekt tudi nastajal. Na koncu bom predstavil še programsko opremo, se pravi gonilnik, testno aplikacijo in igro Snake (Kača), ki uporablja vse podprte elemente grafike.

Kodo za hardware sem pisal v jeziku VHDL, za software pa v C.



Na sliki: Zedboard

Prvi del – implementirano vezje (GPU)

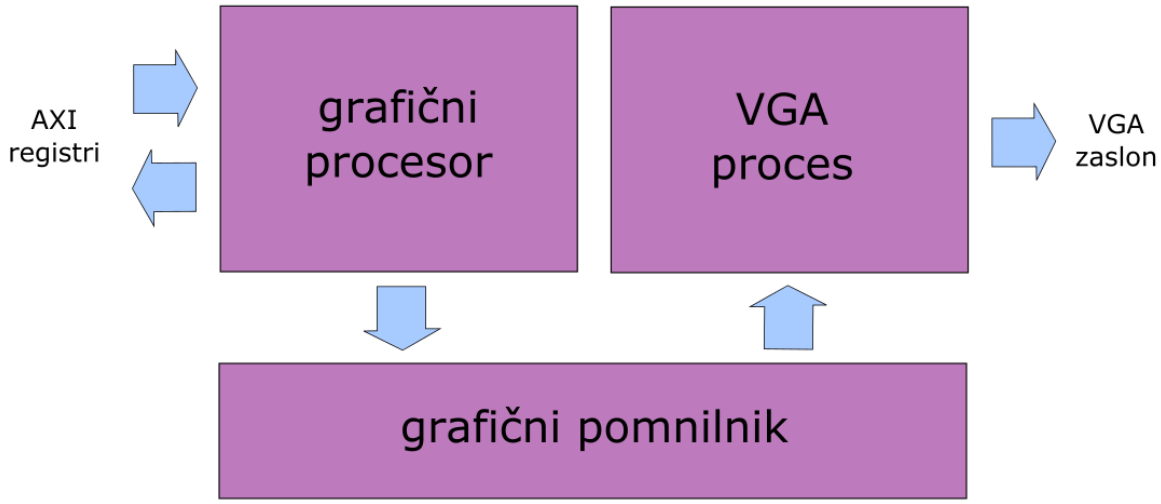
Predstavitev

Na vajah smo že nekaj delali z grafiko oziroma izrisovanjem slike, uporabljali pa smo tudi že komunikacijo med CPE in periferijo prek AXI vodila. Prav tako smo predstavili koncept grafičnega pomnilnika.

Problem pri tej grafiki na vajah je bil v tem, da so bili posamezni elementi slike »zapečeni« v FPGA. Ko smo na zaslonu prikazovali krog, je za njegov izris skrbela posebna hardverska enota. Če bi hoteli narisati še en krog, bi potrebovali še eno identično komponento. In tako naprej. Z drugimi besedami, ti primeri niso imeli nobene fleksibilnosti; če smo želeli na zaslonu narisati kaj drugega, je bilo treba spremeniti VHDL kodo. Edino vezje, ki je bilo že bolj univerzalno in je uporabljalo grafični pomnilnik, pa je znalo risati le posamezne piksle, ki jih je pošiljala CPE.

Zadevo sem se odločil razširiti na univerzalno vezje, ki temelji na grafičnem pomnilniku in zna po ukazih iz CPE generirati posamezne enostavne elemente slike. Procesor torej vezju preko AXI vodila pošlje ukaz, kaj je treba narisati, GPU pa potem to preračuna v ustrezne slikovne točke (piksle) na sliki ločljivosti 256x256 in jih shrani v grafični pomnilnik. Posebna enota potem periodično bere iz tega pomnilnika in generira VGA signal, ki vsebino pomnilnika prikaže na zaslonu. Princip delovanja je torej podoben kot pri pravih grafičnih karticah. Glavna ideja je, da CPE nič ne ve o kakšnih pikslih in risanju slike.

Vezje, ki uresniči to idejo in predstavlja jedro moje naloge, je opisano v datoteki *GPU.vhd* Vivado projekta *GPU.xpr*. Njegovo bistvo prikazuje spodnja shema:



Celotno vezje *GPU* je sinhrono in deluje na enotni uri 50 MHz. Na začetku datoteke je opisan proces *VGA*, ki izrisuje sliko. Ta del kode je kopija komponente *VGAstream* iz laboratorijskih vaj, ki pa sem jo nekoliko modificiral za vgradnjo v svoje vezje. Še vedno prikazujemo 256x256 slikovnih točk v oknu velikosti 512x512 v VGA načinu 800x600. Frekvenca osveževanja in sinhronizacijski signali so nespremenjeni, prav tako en piksel ostaja 8-bitni (3 rdeča, 3 zelena, 2 modra). Ni več signalov *din* ter *dout,* VGA proces pa namesto tega na izhod *rgb* direktno pošilja točke, shranjene v grafičnem pomnilniku. Prav tako je zaradi branja iz pomnilnika, ki povzroči zamik enega cikla ure, nekoliko spremenjena logika za štetje položaja kurzorja (x,y).

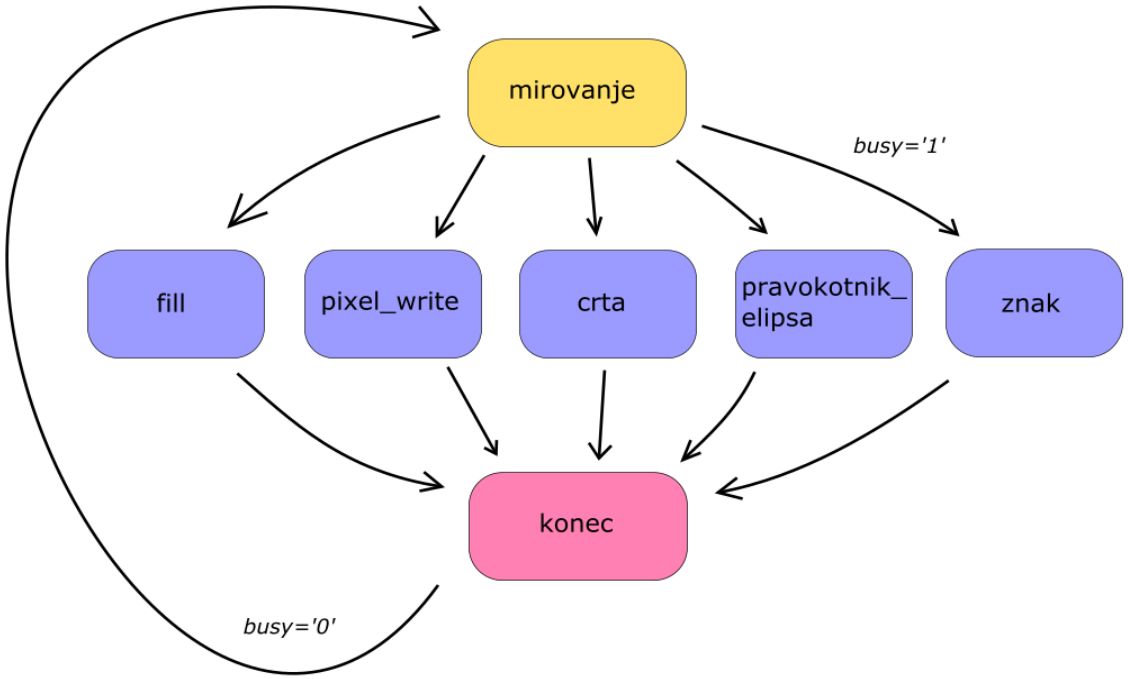
Grafični pomnilnik je tipa BRAM (blokovni RAM), ki omogoča hkratno branje in pisanje. To dovoljuje grafičnemu procesorju in procesu VGA, da delujeta nemoteno eden od drugega. V mojem vezju grafični pomnilnik sestavljata kar dva bloka 256x256 8-bitnih vrednosti (*videoRAM1* in *videoRAM2*), ki predstavljata dve sliki – ozadje in glavno sliko. V kodi ju označujem kot *layer 1* in *2*. Ideja je v tem, da damo recimo na ozadje neko nepremično sliko, v ospredju pa se nekaj dogaja ali premika. Predmeti se potem lahko mirno vozijo čez ozadje, brez da bi ob tem kaj »povozili« oziroma pobrisali. VGA proces se zelo enostavno odloči, katero sliko bo pošiljal na zaslon. Privzeto je to glavna (sprednja) slika, razen če je vrednost piksla 0 (to pomeni, da tam nič ni). V tem primeru prikaže ozadje. Tega se je treba zavedati, namreč da vrednost 0x00 na glavni sliki ne pomeni črne, ampak prazno. Prava črna barva (oprana s Perwollom) lahko obstaja le na ozadju.

Grafični procesor (v kodi proces *GPU*) izvršuje opravila, ki mu jih preko AXI Slave registrov pošilja CPE. Ukazi, ki jih GPU trenutno pozna, so:

- *Fill* : Podamo *barvo* in *layer*, ki ga želimo pobarvati. Podprti layerji so 0 (okvir okoli okna), 1 (ozadje) in 2 (slika). Ekvivalent brisanja zaslona je barvanje z vrednostjo 0x00.
- *Pixel_write*: Najpreprostejše opravilo, ki pobarva en piksel. Sprejeti parametri sta koordinati *p* in *q* ter že poznana barva in layer, slednji pa je od zdaj naprej lahko le 1 ali 2 (po okvirju ne moremo risati, zato z njim dela samo funkcija *fill*). To opravilo je malo nadgrajena verzija naše četrte vaje.
- *Crta*: GPU na layerju 1 ali 2 nariše črto izbrane barve od točke (p,q) do točke (*a,b*).
- *Pravokotnik*: GPU na layerju 1 ali 2 nariše pravokotnik izbrane barve s središčem v (p,q), pri čemer sta *a* in *b* njegovi stranici. Enota omogoča tudi vrtenje lika okoli središča za poljuben kot. Več o tem kasneje.
- *Elipsa*: GPU na layerju 1 ali 2 nariše elipso izbrane barve s središčem v (p,q), pri čemer sta *a* in *b* njeni polosi. Prav tako je omogočeno tudi vrtenje.
- *Znak*: GPU na layerju 1 ali 2 nariše ASCII znak (kode 32 do 95) izbrane barve z levim spodnjim robom v (p,q) velikosti 8x8 pikslov, ki se lahko skalira (povečuje) s celim številom.

To so trenutno podprte funkcije, zlahka pa se dodajajo nove. Ena izmed takšnih očitnih manjkajočih je recimo risanje trikotnikov, ki so osnova vsake prave grafične kartice. Vezje je od samega začetka zasnovano za popolno univerzalnost, zato lahko grafičnemu procesorju poljubno dodajamo nove enote oziroma funkcije, brez spreminjanja ostale VHDL kode. Omejitev je le resolucija 256x256 točk in močno zasidrana 8-bitna arhitektura vezja.

Da razumemo univerzalnost komponente GPU, je treba poznati njeno delovanje. Ogrodje vsega je kontrolna enota, ki bazira na sinhronem avtomatu. Njegova stanja vidimo na sledečem grafu:



Takšna je ponavadi situacija, ko je vse v redu. V resnici je možen še direkten prehod iz *mirovanja* v *konec*, če pride do kritične napake (neveljaven ukaz), kar pa sem zaradi preglednosti na sliki izpustil.

Zadeva deluje takole: najprej stroj čaka v stanju *mirovanje*, kar pomeni, da GPU nič ne dela (prosti tek, »idle«). Še vedno paralelno teče proces VGA, ki periodično osvežuje sliko. Ko pride po AXI vodilu od CPE zahteva po risanju (*busy* bit se postavi na '1' – o tem še kasneje), avtomat glede na podan ukaz izbere naslednje stanje, ki bo izvršilo dano nalogo (eden od modrih okvirčkov). Prav tako še v istem ciklu pripravi podatke, ki jih tista enota potrebuje (inicializacija). Nato gre GPU v eno izmed modrih stanj. Lahko si mislimo, da ima trenutna GPU pet hardverskih enot (*fill, pixel_write, crta, pravokotnik_elipsa* in *znak*), vsako za določeno opravilo. Vedno je aktivna samo ena enota, ki s frekvenco ure (50 MHz) barva ustrezne piksle v grafičnem pomnilniku. Vsak urin cikel se obdela en piksel, tako da se morajo vse računske

operacije, zahtevane za tisti piksel, izvršiti znotraj ene urine periode. To se dogaja brez uporabe vmesnih registrov (cevovoda). V kodi se to vidi tako, da se vsi vmesni rezultati prirejajo spremenljivkam, ne signalom. Cevovoda nisem uporabil, ker bi se moral ukvarjati še z zamiki, ki ob tem nastanejo. Enostavno sem predpostavil, da je vezje dovolj hitro, da lahko izvede zahtevane operacije znotraj 20 ns.

Ko posamezna enota konča z risanjem, gre avtomat v stanje *konec*. To poskrbi, da se *busy* bit postavi nazaj na '0', s čimer signalizira CPE, da je GPU končala, in se vrne nazaj v *mirovanje*.

Grafični procesor – podrobneje

Zdaj, ko razumemo osnovno strukturo in funkcionalnost grafičnega procesorja, gremo lahko še malo bolj v detajle, potem pa bomo vse skupaj zapakirali v IP in si pogledali izsek iz simulacije.

Komponenta GPU je direktno vezana na AXI vmesnik s štirimi 32-bitnimi axi slave registri, ki predstavljajo stik med CPE in našim vezjem. Vsi štirje registri nastopajo kot vhodni signali v GPU (*reg0, reg1, reg2* in *reg3*), kjer se takole povežejo na notranje signale:

```
-- graficni procesor
ukaz <= reg0(3 downto 0);
layer <= reg0(5 downto 4);
barva <= reg0(15 downto 8);
char <= reg0(22 downto 16);
p <= reg1(7 downto 0);
q <= reg1(15 downto 8);
a <= reg1(23 downto 16);
b <= reg1(31 downto 24);
sin1024 <= signed(reg2(11 downto 0));
cos1024 <= signed(reg2(23 downto 12));
busy <= reg3(0);
```

Prvi register (reg0) vsebuje 4-bitni ukaz, ki specificira zahtevano operacijo, torej 0-fill, 1-pixel_write, 2-crta, 3-pravokotnik, 4-elipsa, 5-znak. Prostora je torej že zdaj še za 10 ukazov. Sledi 2-bitni layer, 8-bitna barva in 7-bitna ascii koda, če hočemo risat znak.

V drugem registru (reg1) so razne dimenzije, vse 8-bitne. *p* in *q* sta ponavadi koordinati, kje naj se nekaj nariše, *a* in *b* pa še dodatna parametra dimenzij (pri pravokotniku stranici, pri elipsi polosi, pri črti drugo krajišče). *a* ima pri risanju znakov vlogo faktorja skaliranja (povečave).

Tretji register (reg2) je relevanten za stanje *pravokotnik_elipsa* in vsebuje sinus in kosinus kota zasuka, množena s 1024, torej dve predznačeni 12-bitni števili.

Zadnji register (reg3) je ključen za komunikacijo s CPE oziroma gonilniki in je edini, v katerega GPU tudi piše, medtem ko ostale samo bere. Zadnji bit (LSB) tega registra je prej omenjeni *busy*, ki ga CPE postavi na '1', ko želi, da grafična kartica nekaj nariše. Nato čaka, da GPU postavi ta bit nazaj na '0', kar pomeni, da je naloga opravljena. Kako vezje to naredi, bo opisano pri stanju *konec* in predstavljeno na simulaciji. Vsi ti registri so namreč samo vhodi v vezje, vanje pa prek AXI vodila načeloma lahko piše le CPE. Zato je treba malo popraviti avtomatsko generirano kodo za AXI vmesnik.

Še prej nekaj komentarjev o posameznih stanjih oziroma enotah, ki opravljajo določene naloge. Kot že rečeno, je vsaka enota specifična, inicializira pa se že v stanju *mirovanje*, ko je znana zahtevana operacija. To pomeni, da se vrednosti, ki jih GPU prebere iz vhodnih registrov in so potrebne za izvršitev dane naloge, ustrezno obdelajo in pripravijo za posamezno enoto, če je to potrebno. To se vedno zgodi v enem urinem ciklu, saj v naslednjem že skočimo v stanje, ki bo izvršilo nalogo. Velja omeniti še, da ima kontrolna enota GPU 3-bitni signal *error*, v katerega zapisuje kakšne težave pri izvrševanju opravil. Ta *error* se potem na koncu poleg bita *busy* tudi pošlje v reg3, da CPE ve, če je bilo vse v redu. Vrednost signala se v stanju *mirovanje* resetira na ''000'', tekom izvajanja pa lahko postane ''001'' ali ''111''. ''111'' je kritična napaka in pomeni, da smo poslali nekaj neveljavnega, bodisi neznano kodo ukaza ali napačen layer (pri ascii znakih tudi neveljavno kodo znaka). V tem primeru GPU ne naredi nič, avtomat pa gre takoj v stanje *konec*, ki CPE sporoči, da je šlo nekaj narobe. Napaka ''001'' ni kritična in pomeni, da je del risanega objekta padel izven okna 256x256. Gre bolj samo za opozorilo, naloga pa se nemoteno izvrši do konca. Če na koncu v registru ostane ''000'', je bilo vse OK.

Stanje *fill* pobarva vse piksle na izbranem layerju, enega za drugim. Razen če barvamo okvir, to opravilo traja najdlje, saj moramo pisati v vseh 256x256=65536 točk, kar pri frekvenci delovanja 50 MHz traja dobro milisekundo.

Za stanje *pixel_write* velja ravno nasprotno, saj vpišemo samo en piksel določene barve na izbrano mesto. Izvrši se v enem urinem ciklu. Vseeno to ne pomeni, da lahko CPE s periodo 20 ns pošilja nove piksle, saj traja več ciklov, da avtomat pride nazaj v *mirovanje*, kjer sprejema nova opravila. Konkretno mislim da rabimo vsaj še 3 cikle za administracijo (1 *mirovanje* + 2 *konec*), se pravi skupno 4 za izris enega piksla, če ga rišemo posebej z uporabo tega ukaza. Nalaganje velikih bitnih slik v grafični pomnilnik je torej najbolj zamudno.

Crta izriše črto od točke (p,q) do (a,b) s pomočjo Bresenhamovega algoritma. Je direktna kopija C kode iz četrte vaje (*Test2.c*), spremenjena v VHDL. Mislim da sem uporabljal celo ista imena spremenljivk. Je pa potrebno dobro paziti pri takšnem kopiranju, ker se C koda na procesorju izvaja zaporedno, VHDL in FPGA pa sta nekaj čisto drugega.

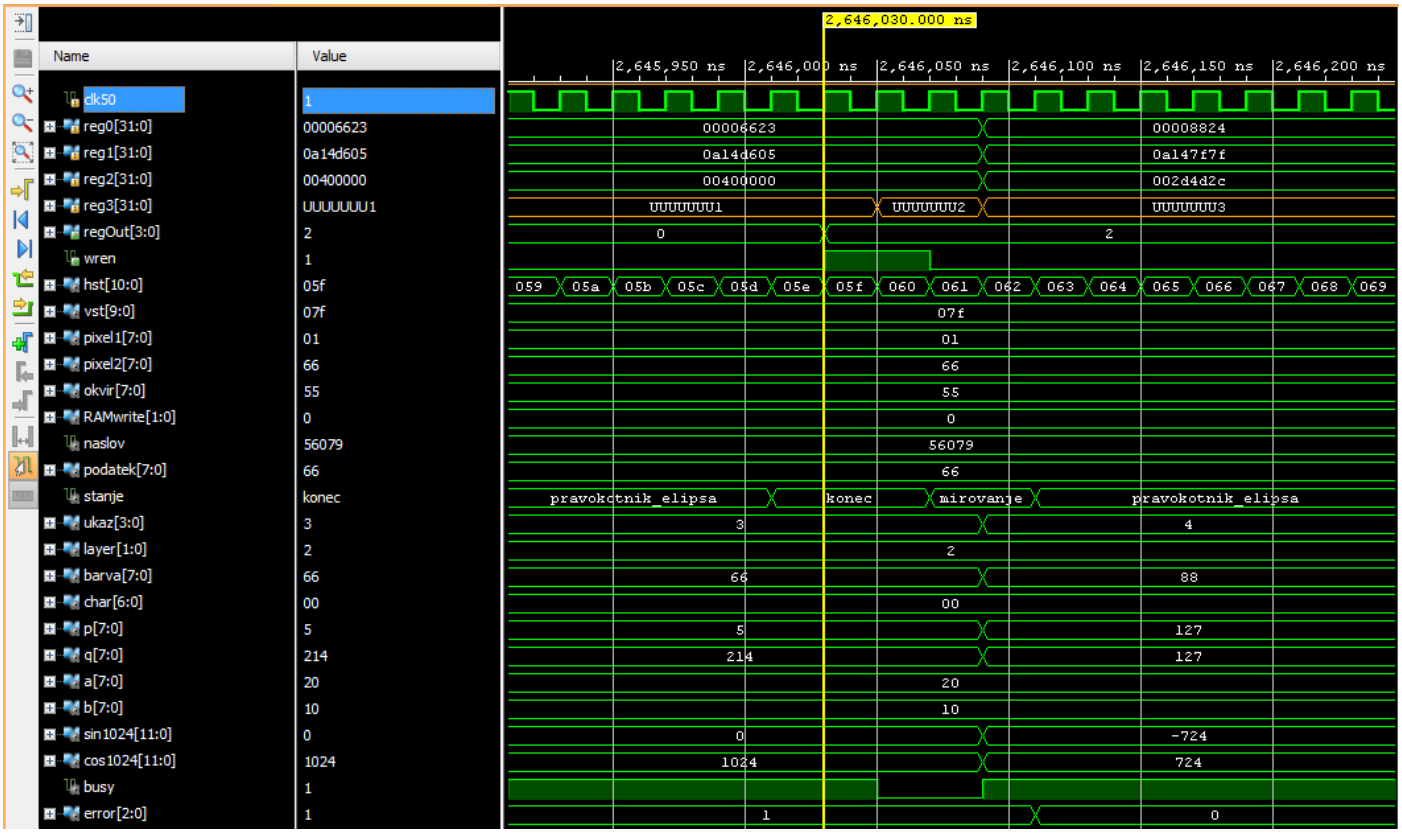
Stanje *pravokotnik_elipsa* je zadolženo za risanje pravokotnikov in elips. Sprva sta bili to dve različni enoti, ki pa sem ju združil, ker sta si bili zelo podobni. Še posebej zdaj v najnovejši različici, ki uporablja univerzalen algoritem za vrtenje okoli središča (p,q). Deluje tako, da se sprehodi po vseh točkah kvadrata, ki je dovolj velik, da se izrisano telo v njem lahko zavrti za 360 stopinj. Ob tem nad vsako točko deluje z inverzom rotacije za zahtevan kot in preveri, če rezultat pade znotraj podanega lika (pravokotnika ali elipse). Če da, jo pobarva, sicer pa ne. Risanje pravokotnikov in elips se razlikuje samo po inicializaciji enote in pogoju, ki preverja, če je zavrtena točka znotraj lika. Vse ostalo je identično, zato se tudi računa v istem stanju. Zanimivo pri tem problemu je, kako točke množiti z rotacijsko matriko, ki vsebuje sinus in kosinus kota. Obe te vrednosti sta realni števili, s katerimi na FPGA ne znamo delati, saj nimamo FPU. Zato tukaj uporabimo en trik. CPE nam v reg2 pošlje obe kotni funkciji, množeni s 1024 in zaokroženi na celo število. GPU vse računske operacije izvede s tema velikima celima številoma, rezultat pa nato premakne za 10 mest v desno, kar je ekvivalentno deljenju s 1024. Na koncu tako dobimo pravilen rezultat, brez da bi na hardveru kjerkoli delali z realnimi števili. S tem, ko smo na CPE množili s 1024, zaokrožili in nato na FPGA delili s 1024, smo v bistvu delali s tremi mesti (za decimalno vejico) realnega števila. Na primer, če imamo kot 45°, sta sinus in kosinus enaka 0,7071067812…, kar po množenju in zaokroževanju da 724, se pravi smo s tem obdržali natančnost ranga promila. Tej tehniki se reče aritmetika s fiksno vejico, za razliko od plavajoče.

Stanje *znak* na zaslon izpiše enega izmed 64 ASCII znakov s kodami od 32 do 95. Pri tem uporablja komponento *CROM.vhd*, ki sem jo »ukradel« s spletne strani letošnje izvedbe predmeta DES in vključil v svoj projekt. V osnovi ostaja vse isto, samo malo sem spremenil signala *cx* in *cy*, da ustrezata mojim zahtevam. Znaki so po naravi veliki 8x8, kar pa se da povečati za faktor poljubnega celega števila. Dodatni zunanji signali, ki jih GPU potrebuje za povezavo s CROM so: 7-bitni *ascii*, 3-bitna *cx* in *cy* ter bit *pix*.

Ko določena enota konča s svojim delom, gre GPU v stanje *konec*, ki je ključno za usklajeno delovanje GPU in CPE. V tem stanju se signal *error* združi z bitom '0' in pošlje na 4-bitni izhodni register *regOut*, ob tem pa na '1' postavimo še izhodni signal *wren*. To dvoje peljemo kot vhod na modificirano komponento AXI vmesnika, ki vsebuje 32-bitne axi slave registre. Komponento sem spremenil tako, da ob aktivnem *wren*, če je CPE tiho, v zadnja mesta četrtega slave registra vpiše vsebino *regOut*. Tako lahko GPU posredno piše v reg3 in procesorju sporoči, da smo konec (*busy* bit je spet na '0') in zraven pošlje še kodo napake, ki jo lahko CPE potem prebere. Ko avtomat vidi, da se je *busy* bit postavil na '0', onemogoči *wren* in vrne GPU v stanje *mirovanje*.

Simulacija

Tako, prišli smo do konca opisa *GPU.vhd*. Vezje sem skupaj s *crom.vhd* vključil v testno strukturo *GPUtest.vhd* (še vedno pod projektom *GPU.xpr*) in preveril delovanje svoje grafične kartice. Struktura GPUtest simulira obnašanje CPE in AXI vmesnika, tako da ustrezno nastavi *reg0-3* (vključno s *busy='1'*) in počaka, da GPU opravi zadano nalogo. Ko zazna omogočen signal *wren*, prepiše vsebino *regOut* v *reg3*, *busy* gre s tem na '0' in testna struktura GPU pošlje novo nalogo. To ponavlja, dokler ne preveri vse funkcionalnosti grafične kartice. Testna struktura je realizirana kot avtomat stanj.

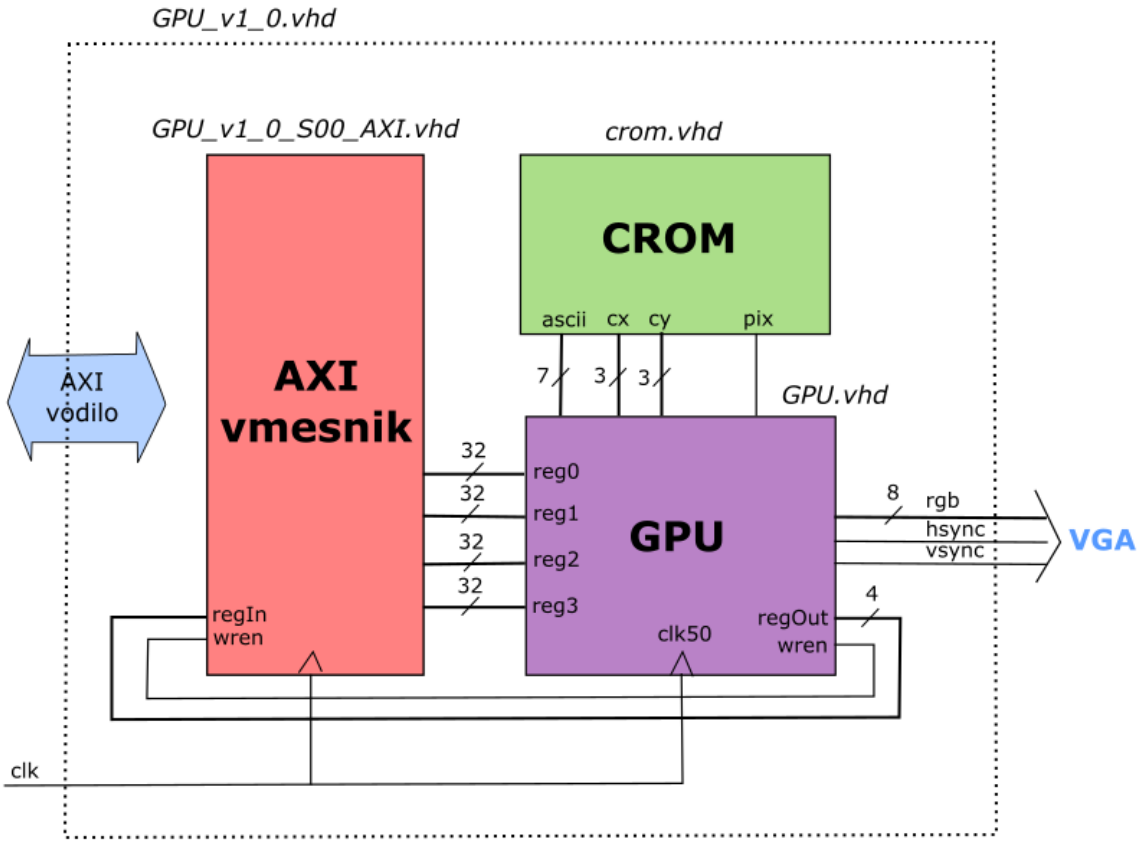


Zgoraj lahko vidimo zelo kratek izsek iz simulacije, ko GPU konča z risanjem pravokotnika in dobi novo nalogo risanja elipse. Ker je pravokotnik padel malo izven okvirja, je error flag (čisto na dnu ekrana) na 1 oziroma ''001''. Rumena črta nakazuje zagon stanja konec (na fronto ure), kar aktivira *wren*, *regOut* pa gre na 2 oziroma ''0010''; torej združena *error&'0'*. Ko testna struktura to zazna, prepiše *reg3, busy* pade na '0' (čisto spodaj), kar spravi GPU nazaj v stanje *mirovanje*. Nato v *reg0-2* naložimo novo nalogo, kar se vidi kot sprememba parametrov (*ukaz* gre s 3 na 4, spremenimo barvo in položaj, *a* in *b* ostaneta ista, elipso pa hočemo zavrteti za -45°, kar opazimo v vrednosti *cos1024* in *sin1024*). Hkrati postavimo *busy* nazaj na '1' in v naslednjem ciklu se vezje spravi risat elipso. Ob tem na dnu opazimo še reset napake na 0.

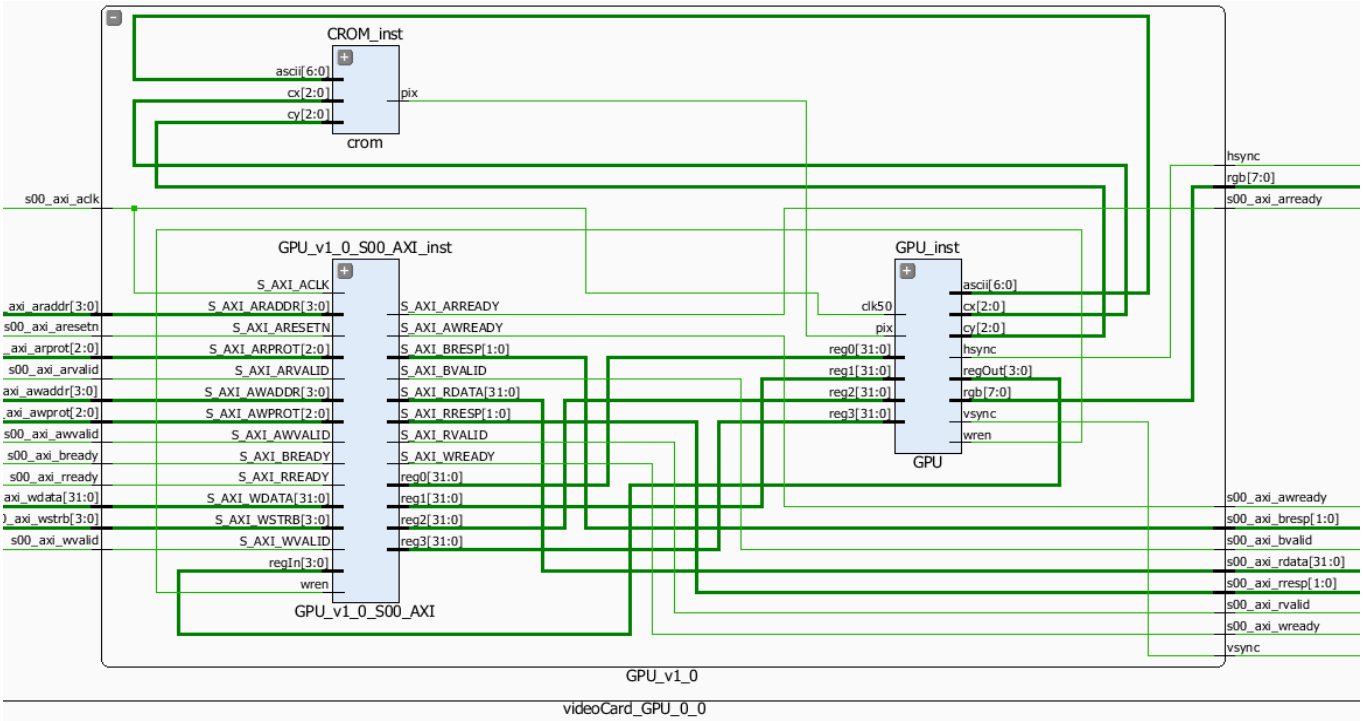
Ta simulacija je sicer glomazna, signalov pa še precej več kot na tej sliki. To je samo tak prijazen primer, ki prikazuje nek zanimiv trenutek.

Komponenta IP

Vezje GPU in CROM sem skupaj z AXI vmesnikom zapakiral v komponento IP pod imenom *GPU_v1_0.vhd* v sklopu (skoraj)istoimenskega projekta. Trenutna različica komponente je *GPUv2_3*, tako da je ime projekta malo zavajajoče. Ampak nič ne de. Struktura IP je torej sledeča:

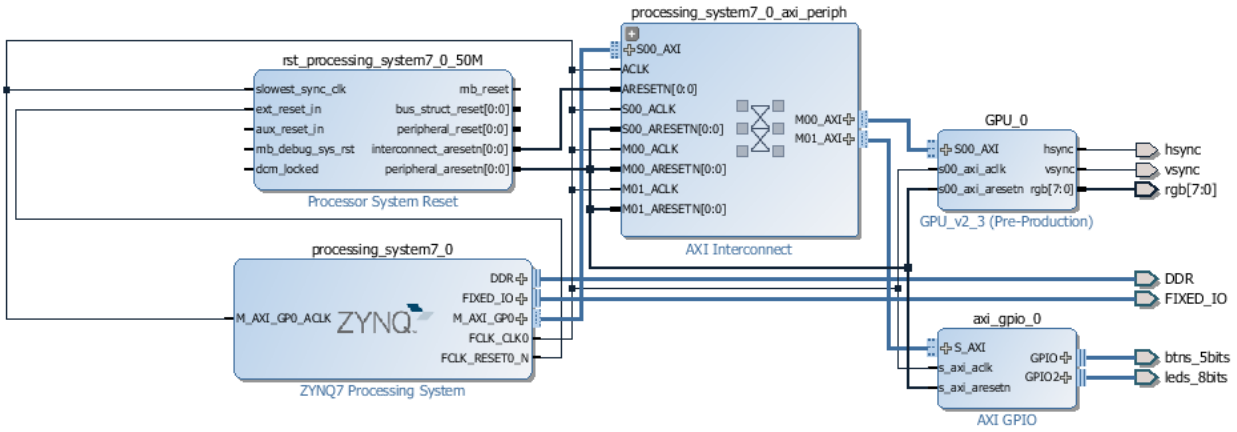


Nekaj podobnega, a mogoče nekoliko manj preglednega dobimo, če malo kopljemo po shemi finalnega projekta:

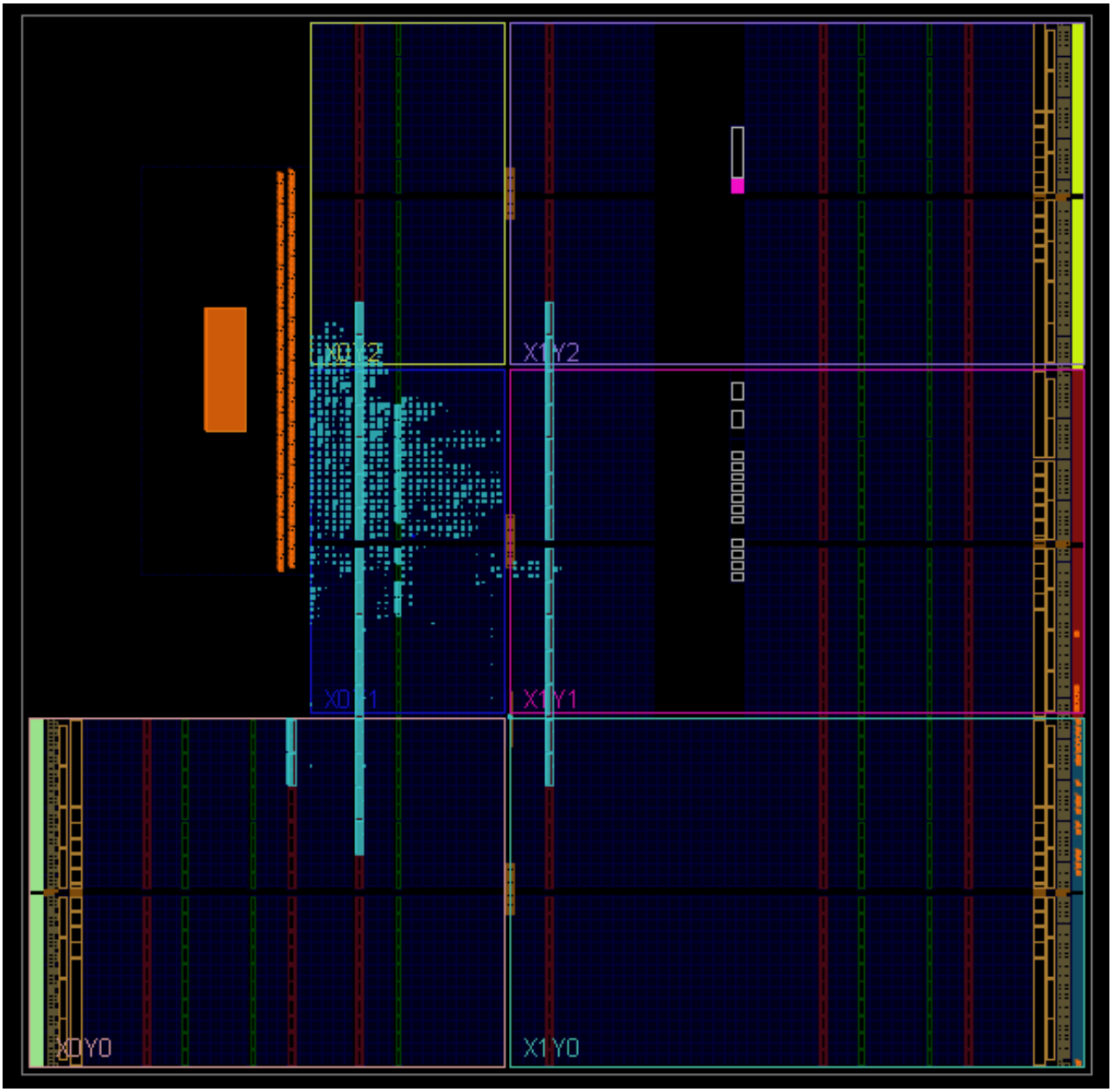


Implementacija

Projekt *VideoCard.xpr* vsebuje celoten digitalen sistem, ki je implementiran na FPGA. Vanj sem vključil svojo komponento IP in dodal še GPIO z gumbki in LEDicami, da je zadeva bolj interaktivna. Blokovna shema:



Desno najdemo svojo komponento GPU_v2_3. Odpremo še *Implemented Design*:



Na prvi pogled kar lepa uporaba vezja. Vidimo, da smo porabili precej BRAM blokov in nekaj DSPjev. Poglejmo še poročilo (*Utilization Report*) po implementaciji:

Site Type	Used	Fixed	Available	Util%
Slice LUTs	1667	0	53200	3.13
LUT as Logic	1599	0	53200	3.01
LUT as Memory	68	0	17400	0.39
LUT as Distributed RAM	0	0		
LUT as Shift Register	68	0		
Slice Registers	1214	0	106400	1.14
Register as Flip Flop	1214	0	106400	1.14
Register as Latch	0	0	106400	0.00
F7 Muxes	31	0	26600	0.12
F8 Muxes	1	0	13300	<0.01

Dobra 3-odstotna zasedenost Slice LUT, kar ni tako hudo. Nižje najdemo še informacijo, da smo zasedli 32 blokov BRAM (23 %) in 10 DSPjev (5 %).

Vivado se po implementaciji sicer pritoži, da nekatere časovne zahteve niso izpolnjene. Našel je neke čudne poti, ki rabijo dlje od meje 20 ns (najdaljša je 30 ns). Vseeno vezje normalno deluje.

Drugi del – programje

Gonilnik

CPE se torej z grafično kartico, implementirano na FPGA, pogovarja preko štirih 32-bitnih AXI slave registrov. To komunikacijo in vse, kar spada zraven, sem lepo zapakiral v nekakšen gonilnik, ki je v bistvu zbirka funkcij, ki jih kličemo iz aplikacije, ki uporablja grafiko. V resnici ne gre za gonilnik v pravem pomenu besede, ker nimamo operacijskega sistema, ampak zgolj za nek vmesnik med programsko in strojno opremo, da se programerju ni treba ukvarjati z delovanjem GPU. V bistvu sploh ne rabi nič vedeti o tem. Funkcije so zbrane v datoteki *gfx_driver.c*, njihove deklaracije pa v *gfx_driver.h*. Header moramo vključiti v program, ki uporablja grafično kartico.

Najbolje, da pogledamo primer ene funkcije iz *gfx_driver.c*, recimo za risanje pravokotnika:

```
int pravokotnik(int layer, int barva, int p, int q, int a, int b, int zasuk){

    Xil_Out32(REG0, (barva << 8) | (layer << 4) | 3);   // ukaz = 3
    Xil_Out32(REG1, (b << 24) | (a << 16) | (q << 8) | p);

    double pretvorba = PI / 180;
    int sin1024 = (int)round(sin(zasuk*pretvorba) * 1024);
    int cos1024 = (int)round(cos(zasuk*pretvorba) * 1024);
    sin1024 = sin1024 & 0x00000FFF; // maska --> rabimo le zadnji 12 bitov
    cos1024 = cos1024 & 0x00000FFF;
    Xil_Out32(REG2, (cos1024 << 12) | sin1024);

    Xil_Out32(REG3, 1); // execute
    while (Xil_In32(REG3) & 1){}    // cakaj dokler je busy = 1

    int error = (Xil_In32(REG3) >> 1) & 0x00000007; // zadnji 3 biti
    if (error == 7){ error = -1;}

    return error;
}
```

Funkcija sprejme parametre, ki jih že poznamo, torej layer, barvo, koordinate in dimenzije ter kot zasuka v stopinjah. Vsi parametri razen zasuka se vpišejo direktno v registra 0 in 1, vključno s kodo ukaza 3 za risanje pravokotnika. Nato je treba pripraviti obe kotni funkciji, množeni s 1024 in zaokroženi, kot sem razlagal v prejšnjem delu pri fiksni vejici. To gre potem v REG2. Ko so vsi podatki poslani, se vpiše še 1 v REG3, kar pomeni *busy='1'*, s čimer poženemo GPU. V naslednji vrstici v *while* zanki beremo REG3 in čakamo, da se *busy* vrne na '0'. S tem CPE čaka, da GPU konča, kar ni ravno optimalno, je pa bolj varno, ker poskrbi, da se vrnemo iz funkcije šele ko je naloga opravljena, prav tako pa na ta način ne moremo klicati nove funkcije, ko GPU morda še ne bi bila pripravljena na sprejem novih ukazov.

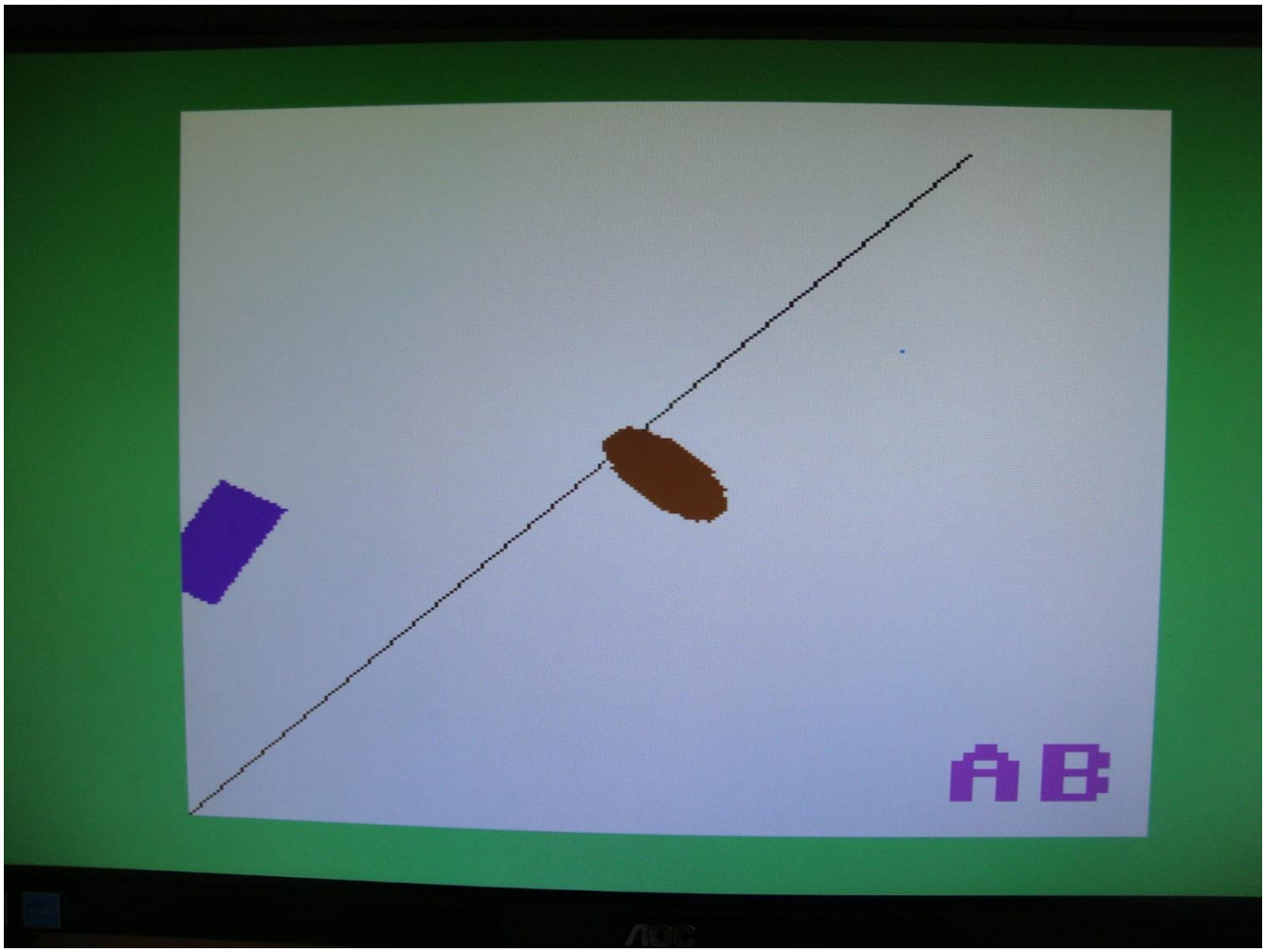
Ko je opravilo končano, v spremenljivko *error* zapišemo še morebitno napako, ki nam jo je GPU pustila v »nabiralniku« in jo vrnemo glavnemu programi, ki je funkcijo poklical. Tam lahko kaj naredimo glede tega, ali pa se ne sekiramo in sploh ne preberemo napake. Stvar programerja.

Pomembno opozorilo: kljub temu, da so vhodni parametri funkcije deklarirani tipa *integer*, morajo njihove vrednosti ustrezati obsegu, ki ga podpira GPU! To pomeni, da morajo recimo *barva, p, q, a* in *b* imeti vrednosti od 0 do 255, torej obseg 8-bitnega nepredznačenega števila. Če je ta obseg prekoračen, se ne bosta niti driver niti GPU nič pritožila, izrisan rezultat pa bo napačen. Poznavanje obsega vrednosti, ki jih grafična kartica sprejema, je odgovornost programerja.

Pri drugih funkcijah ni kaj dodati. Mogoče samo pri *text*, ki na zaslon izpiše besedilo. HW sicer zna risati le znak po znak, vendar lahko funkciji vseeno pošljemo celo besedilo, ki ga potem sama razbije na znake. Paziti je treba še, da od črk uporabljamo le velike tiskane, ker male trenutno (še) niso podprte s strani GPU (oziroma CROM).

Testna aplikacija

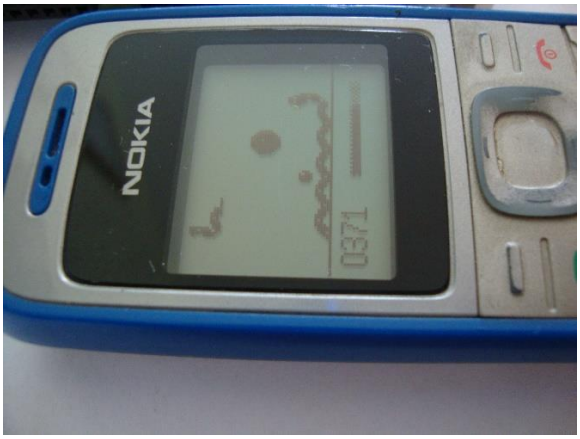
Za test delovanja celotnega sistema na ZedBoardu sem spisal eno enostavno sekvenco, ki preveri vse funkcije grafičnega procesorja. Koda se nahaja v datoteki *aplikacija.c* SDK projekta *testnaAplikacija*, oziroma v *helloworld.c* projekta *test+uart*. Gre za isto stvar, samo da zna slednja še izpisovati vrnjene napake prek UART vmesnika. Rezultat na ekranu:



Pobarvali smo okvir na zeleno, nato sliko na belo, čez pa narisali en svetlomoder piksel (v 1. kvadrantu, če človek dobro pogleda), črto neke … kakšna je to barva?, vijoličen pravokotnik, zarotiran za 60 stopinj, elipso, obrnjeno za -45 stopinj in tekst »AB«, skaliran s 3. Napak ni bilo, razen pri pravokotniku, ki je pričakovano vrnil 1 (del izven okna). Prav tako sem poskusil pobarvati še layer 3, ki ne obstaja, kar ni naredilo nič in je vrnilo napako -1. Vse dela!

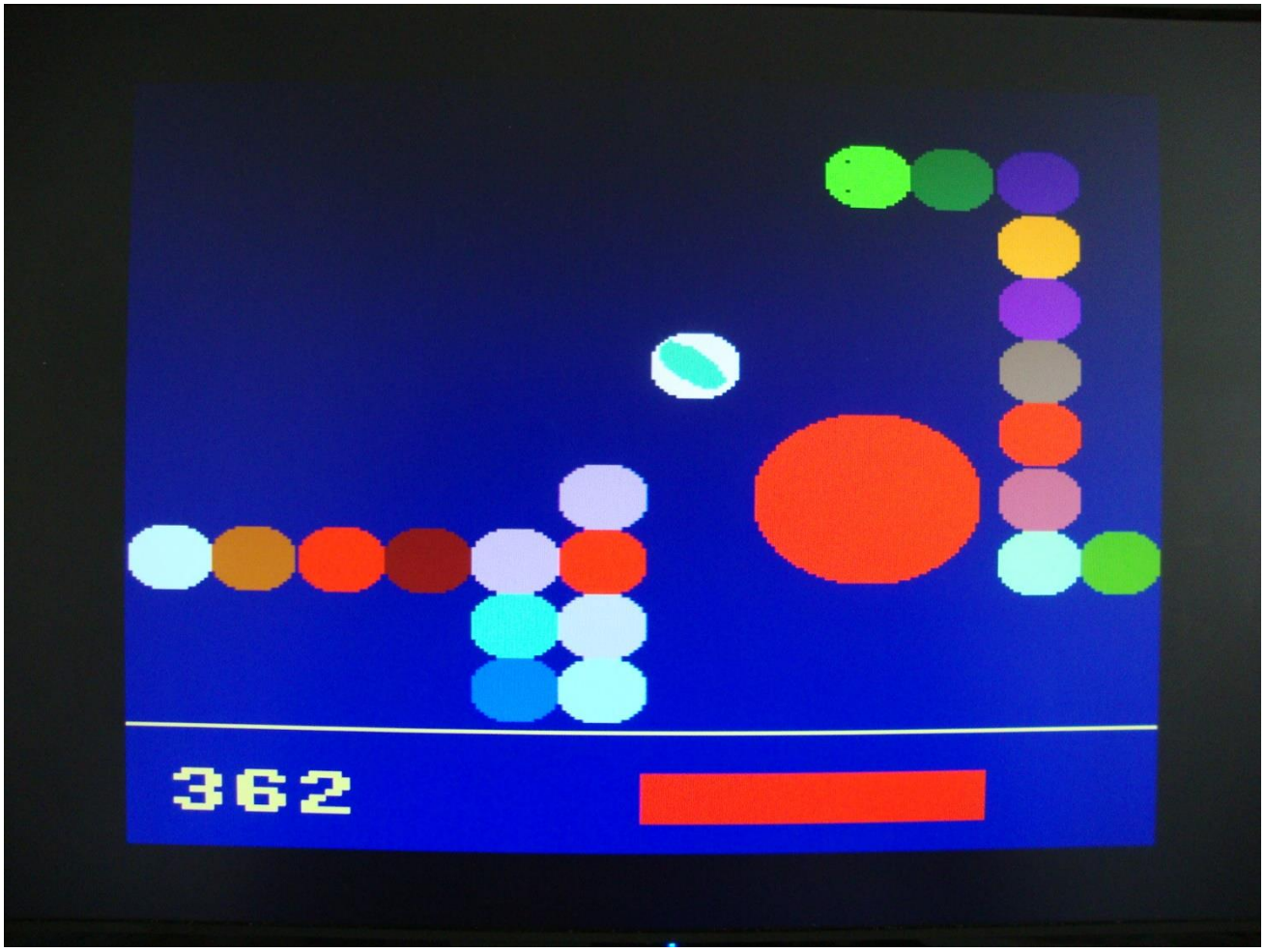
Igra – Snake

Za konec sem se kot aplikacijo, ki bo uporabljala vso to grafiko, odločil narediti preprosto igro, in sicer Kačo (Snake). Igra temelji na legendarni *Snake Xenzia*, ki sem jo rad igral na svojem prvem telefonu, Nokii 1200.



Na sliki: Nokia 1200 in Snake Xenzia

Moja igra je tej zelo podobna, v bistvu naj bi bila njena kopija, samo grafika ni tako napredna, kača pa se giblje diskretno (po korakih), ne zvezno. Igralec uporablja gumbke na ZedBoardu za premikanje kače, ki je hrano in ob tem raste. Vsaka peta pobrana hrana proizvede veliko hrano, ki jo je treba čimprej pojest, da dobimo več točk. Lahko prehajamo tudi skozi stene. Igre je konec, ko se kača zaleti v lastno telo. Srednja tipka je za novo igro in pavzo. Screenshot:



Navadno hrano prepoznamo po belem krogu, v katerem se vrti elipsa naključne barve. Ko to pojemo, kači zraste nov člen iste barve. Na zgornji sliki je še velika rdeča hrana in indikator, koliko točk še vsebuje (0-100). Navadna hrana je 10 točk.

Igra uporablja vse elemente grafike, ki jih GPU podpira (to je tudi namen). Imamo črn okvir in temnomodro ozadje. Oči kače so narisane s *pixel_write*, njeni členi in hrana pa so krogi. Vrtenje objektov demonstrira elipsa, ki se vrti v hrani. Pravokotnik predstavlja rdeč indikator točk velike hrane na dnu ekrana. Imamo še črto, ki ločuje igralno okno od ostalih informacij in točke, ki so izpisane kot besedilo iz zbirke *crom*.

Za konec še dve opombi: igra za koordinate in barve uporablja generator psevdonaključnih števil (C funkcija *rand()* iz standardne knjižnice), ki mu je treba na začetku podati neko na videz naključno »seme«. Ponavadi je to trenutni čas. Ker to na Zynq čipu ni podprto, sem tja vpisal kar konstantno število, kar je za demonstracijske namene čisto OK. Pač se vsakič, ko program znova naložimo, igra začne enako. Prav tako ima Snake en bug, ki ga še nisem našel in ga trenutno tudi ne mislim iskati. Gre za to, da igra pri zelo veliko dolžini kače (točke 1200+) zmrzne. Najprej sem mislil, da sem kakšen seznam deklariral prekratek, ampak ne gre za to. Napako je odkrila moja sestra, ki je ekspert za takšne igre in ji je uspelo priti do tako neverjetno visokega rezultata. Zadeva je dokazano ponovljiva, baje ko poješ hrano, ampak to ni čisto prepričana.

Mimogrede, koda se nahaja v datoteki *snake.c* SDK projekta *Igra*. Vse to programje pa najdemo v Xilinx projektu *VideoCard.*