STŘEDOŠKOLSKÁ ODBORNÁ ČINNOST

Obor č. 10: Elektrotechnika, elektronika a telekomunikace

RISC-V Procesor

Dominik Liberda
Moravskoslezský kraj

Sedliště 2021

STŘEDOŠKOLSKÁ ODBORNÁ ČINNOST

Obor č. 10: Elektrotechnika, elektronika a telekomunikace

RISC-V Procesor
RISC-V Processor

Autoři: Dominik Liberda
Škola: Střední průmyslová škola elektrotechniky a informatiky, Kratochvílova 7/1490, 702 00 Ostrava – Moravská Ostrava
Kraj: Moravskoslezský kraj
Konzultant: Mgr. Daniel Merta
Sedliště 2021

Prohlášení

Prohlašuji, že jsem svou práci SOČ vypracoval/a samostatně a použil/a jsem pouze prameny a literaturu uvedené v seznamu bibliografických záznamů.

Prohlašuji, že tištěná verze a elektronická verze soutěžní práce SOČ jsou shodné.

Nemám závažný důvod proti zpřístupňování této práce v souladu se zákonem č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších předpisů.

V Sedlištích dne 18. 3. 2021 .....

Dominik Liberda

Poděkování

Chtěl bych poděkovat celé mé rodině – Anetě Liberdové, Janě Liberdové a Romanovi Liberdovi.

Chtěl bych poděkovat i mé třídní učitelce Pavlíně Pavlové a vedoucímu práce Danielu Mertovi.

Velké poděkování patří i mým kumpánům – Markovi Pekárkovi, který pro mě byl vždy oporou a dobrým kamarádem, Vojtěchovi Macurovi, Tomáši „Kotynovi“ Kotáskovi, Výtkovi Žáčkovi, Josefu Hranitzkému a Matyáši Jarčíkovi.

Poděkování si dále zaslouží i Daniel Figa, Antonín Zápalka, Viki a Bára Tolaszovy, Darina Kovalová, Dominik Schmit, Eliška Pekárková, Daniel Augustín, Radek Plaček, Honza Mička a Tereza Pavlisková.

Anotace

Tato práce se zabývá vývojem 32bitového RISC-V procesoru a jeho implementací na FPGA.

Klíčová slova

RISC-V; CPU; procesor; instrukce; FPGA;

Annotation

This project is focused on development of a 32 bit RISC-V processor and it’s implementation on a FPGA

Keywords

RISC-V; CPU; processor; instruction; FPGA;

OBSAH

1. Úvod

Skoro každý dnes používá počítač nebo mobilní telefon, a každé takové zařízení je poháněno procesorem. Procesory se nacházejí i v mnohem jednodušších zařízeních, jako jsou například mikrovlnky nebo lednice. Dalo by se říct, že ve skoro každém moderním zařízení se nachází hned několik procesorů – od jednoduchých mikrořadičů až přes výkonné a velmi žravé procesory.

Úkolem procesorů je zpracovávat data, ovládat a řídit.

Od jakživa mě fascinovalo, jak můžou věci jako počítač fungovat – něco tak neskutečně komplikovaného a složitého, a toužil jsem jednou jejich fungování porozumět. Tato touha mě jednou donutila do Googlu zadat „How to design a CPU“. K mému překvapení jsem narazil na opravdu skvělou učebnici [22], která se věnovala právě návrhu procesoru – začínala vysvětlením jednoduchých digitálních obvodů, přes obvody napsané v HDL až po navrhnutí vlastního MIPS procesoru.

Rozhodl jsem se tedy, že si jednou vlastní procesor navrhnu. Tato myšlenka stála na počátku této práce a definovala její cíl – vytvořit 32bitový procesor na bázi instrukční sady RISC-V (přesněji její varianty RV32I). Proč zrovna 32bitový? Moderní vývojové nástroje značně usnadňují návrh procesoru, a není tudíž moc velký rozdíl mezi návrhem 16 a 32 bitového jádra procesoru. Přišlo mi tedy vhodné navrhnout rovnou 32bitový procesor na bázi instrukční sady používané v reálných produktech než navrhnout 16bitový procesor na bázi instrukční sady, která dnes už dávno není aktuální.

Abych mohl předvést funkčnost procesoru, rozhodl jsem se ho doplnit o rozhraní PS/2 pro připojení klávesnice a o VGA výstup pro připojení monitoru.

2. CO JE TO PROCESOR?

Obecně se dá říct, že procesor je digitální obvod, jehož úkolem je zpracování dat. Procesor vykonává program, který je složen z mnoha instrukcí. (např. sečti dvě proměnné, načti proměnnou z paměti, podmíněné skoky).

Digitální obvody pracují s binárními signály – jedničkami a nulami, kde logická 1 je obvykle reprezentována kladným napětím a logická 0 je obvykle reprezentována napětím 0V. Tyto obvody jsou složeny z tranzistorů, které jsou zapojeny do logických hradel – AND, OR, NAND, XOR, NOR. Digitální obvody dále obsahují např. paměti typu flip flop, které slouží pro uchovávání dat a tvoří např. registry.

V procesoru se vyskytují dva typy čísel – Signed (se znaménkem) a Unsigned (bez znaménka).

Např. takto vypadá 8bitové unsigned binární číslo, kde jednotlivé bity mají váhu určenou dle jejich pozice:

```
  1   0   0   1     0   0   1   1
x128 x64 x32 x16   x8  x4  x2  x1
-------------------------------------
 128 + 0 + 0 + 16  +  0 + 0 + 2 + 1 = 147
```

Tímto způsobem jsme schopni do 8bitového unsigned čísla zakódovat desítkové hodnoty 0 až 255.

Podobně vypadají signed čísla, s tím rozdílem, že nejvyšší bit má váhu se záporným znaménkem:

```
  1   0   0   1     0   0   1   1
x-128 x64 x32 x16   x8  x4  x2  x1
-------------------------------------
 -128 + 0 + 0 + 16  +  0 + 0 + 2 + 1 = -109
```

Tímto způsobem jsme schopni zakódovat desítková čísla v rozmezí od -128 až 127.

3. JAK SE LIŠÍ TTL, CPU, FPGA A ASIC?

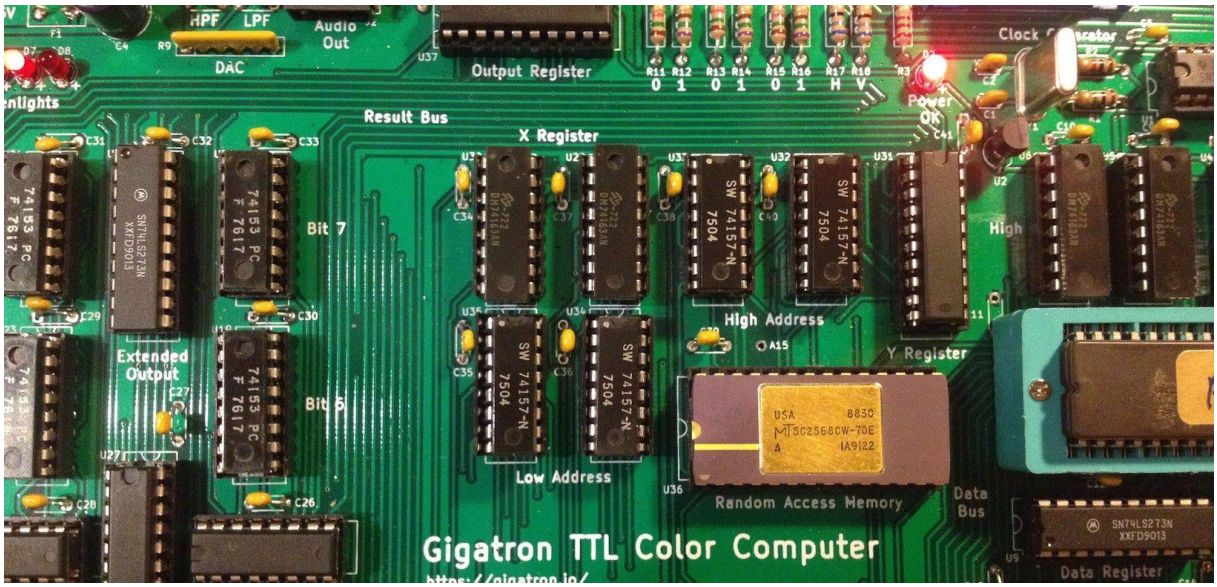
Digitální logiku lze implementovat hned několika způsoby. Tato kapitola slouží jako krátké shrnutí rozdílů mezi TTL, FPGA a ASIC obvody, a zdůvodnění, proč jsem si pro svůj procesor vybral právě FPGA a ne např. TTL obvody nebo rovnou ASIC.

3.1. TTL

Když se řekne digitální obvod, většina si představí právě TTL (popřípadě CMOS) čipy, které mají charakteristický vzhled „černých švábů“ (viz obrázek č. 1).

Velkou výhodou TTL obvodů je jejich jednoduchost – stačí na nepájivém poli zapojit několik drátků a během okamžiku máme funkční logické hradlo.

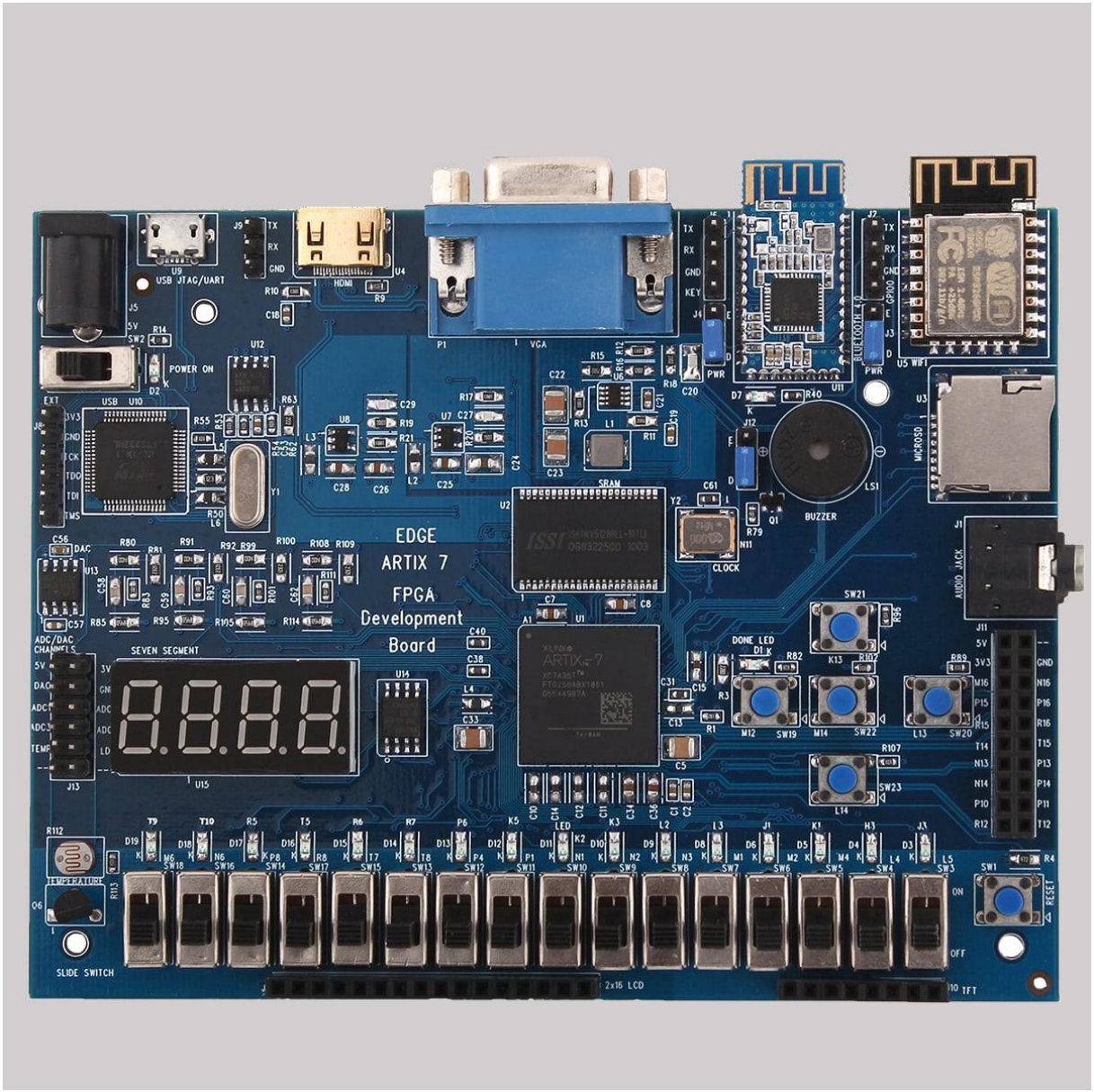
Nevýhody TTL obvodů se začnou projevovat hlavně u větších (a složitějších) zapojení – postavit TTL obvod z pár desítek nebo ze stovky logických hradel není problém, ale postavit TTL obvod z tisíce logických hradel je takřka nemožné – takové zapojení by zabíralo obrovské množství místa, potřebovalo by vysoké množství energie a pravděpodobně by mělo problém fungovat při vysokých frekvencích.



Obrázek č. 1 - Ukázka TTL obvodu, zdroj: https://hackaday.io/project/20781-gigatron-ttl-microcomputer

3.2. FPGA

Programovatelné hradlové pole (FPGA – Field Programmable Gate Array) na rozdíl od TTL, CPU či PLC nenese žádnou fixní logiku – když přijdou z továrny, jsou „prázdné“, nic neumí. Je na uživateli, aby navrhl digitální obvod a nahrál ho do FPGA.



Obrázek č. 2 – Foto FPGA desky, zdroj: https://allaboutfpga.com/product/edge-artix-7-fpga-development-board/

Velká výhoda oproti obvodům TTL je vysoká míra integrace – FPGA jsou schopné nést logické obvody s několika miliony logických hradel.

Výhody FPGA jsou:

- Rychlé prototypování – nemusí se čekat na výrobu digitálního obvodu/PCB v továrně, stačí nahrát kód na FPGA a během pár minut máte na stole funkční prototyp.
- Obrovská flexibilita

Nejčastější aplikace FPGA jsou např. armádní technika, vesmírné mise (např. rakety společnosti SpaceX jsou řízeny FPGA od výrobce Xilinx [1]), ale FPGA lze nalézt také v hračkách nebo automobilech.

3.3. CPU

Pod pojem CPU spadají jak velmi výkonné serverové procesory, tak i různé mikrořadiče a mikrokontrolery. Základní vlastností CPU je schopnost vykonávat program, který je tvořen instrukcemi.

Jelikož dnešní CPU jsou velmi výkonná a dnešní mikrokontrolery jsou velmi levné, může vyvstat otázka, zda mají FPGA smysl, a co je jejich výhoda oproti koupi běžných CPU a mikrokontrolerů.

FPGA oproti CPU umožňují, aby si uživatel navrhl vlastní specifický obvod a přidal ho např. k procesorovému jádru na FPGA (tak jako já jsem si k procesorovému jádru přidal obrazový procesor).

Například pokud potřebujete zpracovat data z kamery specifickým způsobem, můžete na FPGA navrhnout obvod specializovaný právě pro tuto činnost – takový obvod zvládne práci rychleji a levněji, než kdybyste se stejnou činnost pokusili udělat na CPU, které na tuto aplikaci není specializované.

FPGA jsou neskutečně flexibilní a lze na nich navrhnout například A/D převodníky nebo rovnou celý osciloskop.

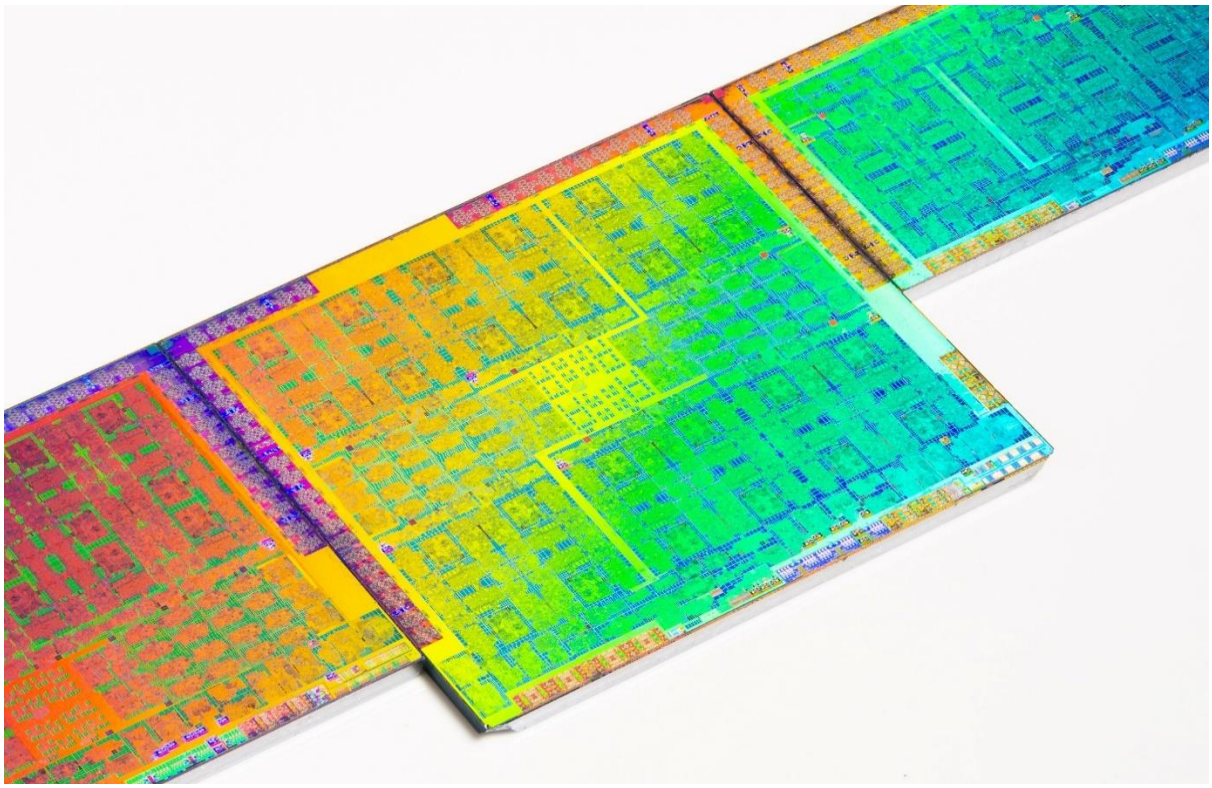
Obrázek č. 3 – Fotka CPU a mikrokontroleru, zdroj: https://components101.com/articles/difference-between-microprocessor-and-microcontroller

3.4. ASIC

ASIC (neboli Application Specific Integrated Circuit) je digitální obvod vyvinutý na míru specifické aplikaci, který se následně nechá vyrobit v továrně (oproti tomu, aby se nahrál na FPGA). ASICy mají mnoho výhod oproti FPGA – nižší kusová cena, nižší spotřeba, vyšší integrace a zároveň vyšší rychlost. Obecně řečeno, na míru navrhnutý ASIC bude vždy ve všem lepší než stejný obvod na FPGA.

Obrovskou nevýhodou ASICů je ale cena a složitost jejich návrhu – navrhnout ASIC čip trvá několikrát déle (řádově roky) než navrhnout stejný obvod pro FPGA. Aby se ASIC mohl nechat vyrobit v továrně, je potřeba vyhotovit masky pro výrobní linku, prototypy, provést validaci a mnoho jiných kroků – výsledná cena se u moderních výrobních procesů může vyšplhat až na stovky milionů dolarů [2]. Z těchto důvodů se ASIC obvody využívají pouze pro sériovou výrobu, kde nevadí vysoká cena fixních nákladů.[1]

Zástupci ASIC obvodů jsou např. grafické karty, akcelerátory umělé inteligence, obvody na těžení kryptoměn nebo také CPU.



Obrázek č. 4- ASIC GPU nVidia, zdroj: https://www.flickr.com/photos/130561288@N04/50914796651/

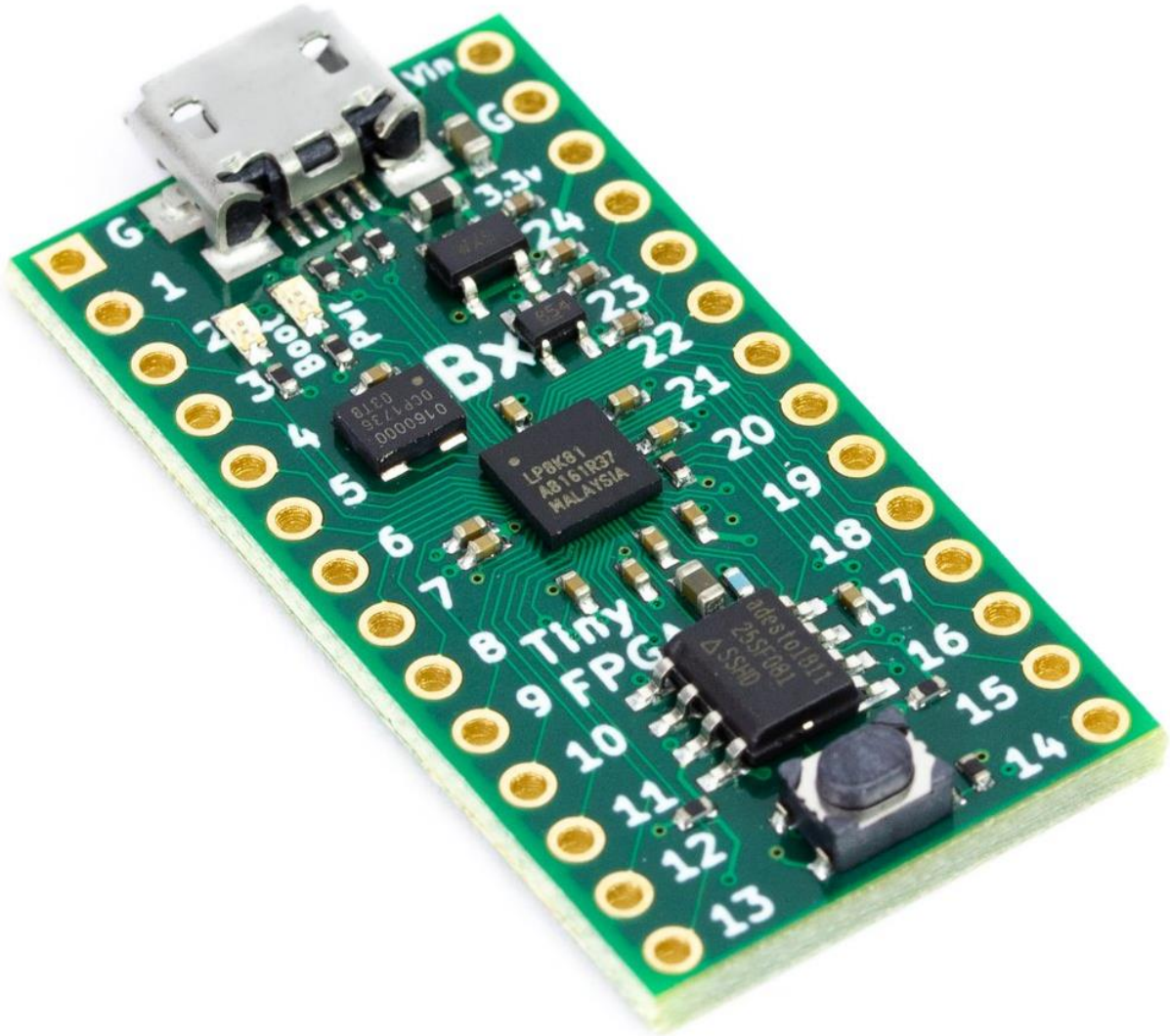
[1] V současné době se začínají objevovat příležitosti, jak si může i „běžný smrtelník“ díky skupinovým projektům navrhnout vlastní ASIC obvod. Např. efabless nabízí možnost výroby 10mm^2 ASICu na 130nm procesu za 10 000$ (v ceně jsou zahrnuty i vývojové nástroje, PDK atd.) - ačkoliv se stále jedná o obrovskou sumu, běžně je tato suma 10-100x vyšší.

4. VÝBĚR FPGA

Jelikož jsem k projektu návrhu CPU přišel jako úplný nováček (co se týče práce s FPGA), neměl jsem moc dobrou představu, jaké FPGA potřebuji – FPGA mají mnoho parametrů, a je těžké se v nich zorientovat bez zkušenosti.

Asi nejzásadnějším parametrem je počet tzv. LUTů. Množství LUTů omezuje, jak velký procesor se na FPGA vleze a pohybuje se od několika stovek, až do několika milionů u nejvýkonnějších FPGA [3].

Nakonec jsem se rozhodl pro FPGA desku TinyFPGA BX, a to kvůli její malé velikosti (připomíná Arduino Nano), vysokému počtu LUTů (~8000) za poměrně nízkou cenu (~1300Kč).



Obrázek č. 5- TinyFPGA BX, zdroj: https://www.crowdsupply.com/tinyfpga/tinyfpga-ax-bx

Zvolená FPGA deska – TinyFPGA BX – je založena na FPGA čipu iCE40 LP8K od výrobce Lattice [4].

Parametry FPGA tedy jsou:

1. 7680 LUT4 (rozdělených do 960 PLB)
2. 128 kb Blokových paměti (32 bloků, každý má kapacitu 4 kb)
3. 1x PLL
4. 41 uživatelských I/O pinů (31 dedikovaných, 10 sdílených)
5. 1MB Flash paměť Adesto AT25SF081

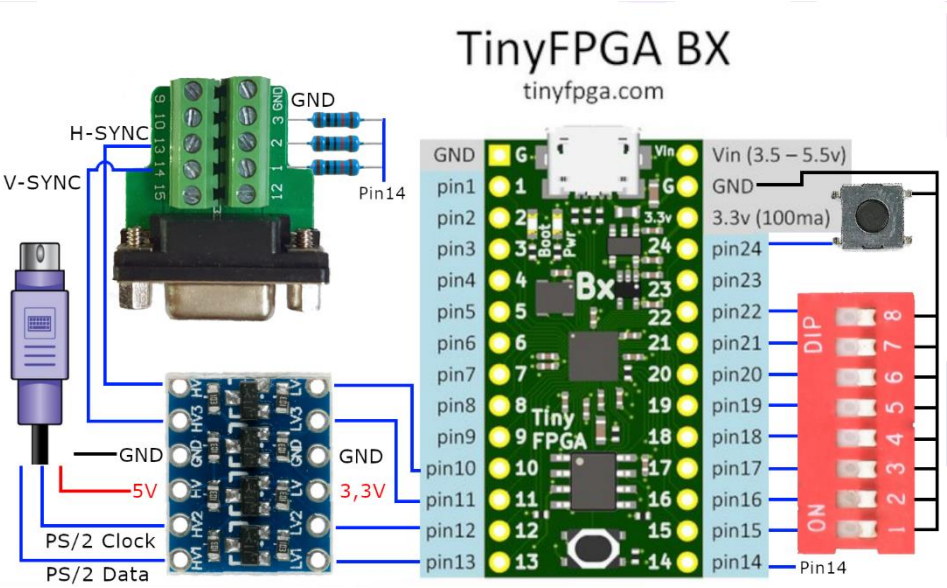
4.1. Architektura FPGA

Moderní FPGA se skládají z mnoha logických bloků [5], které následně slouží pro implementaci logických funkcí. Tato podkapitola obsahuje krátký přehled bloků, které se nacházejí ve většině FPGA.

Obecně lze říct, že FPGA obsahují tyto bloky:

1. PLL – obvod používaný pro generování hodinových signálů o různých frekvencích.
2. Výpočetní bloky – každý výrobce je pojmenovává jinak (např. Lattice PLB, ale Xilinx CLB). Obsahují několik tzv. LUTů (Look Up Table), které jsou používány pro vykonávání logických (AND, NOT, OR…) a aritmetických operací. Zároveň obsahují několik (typicky 4 až 16) bitů paměti typu Flip-Flop.
3. Bloková paměť (Block RAM) – velké bloky paměti (orientačně 4 kb až 36 kb) určené pro implementaci velkých paměťových struktur – paměti cache, FIFO. Block RAM nebývají moc flexibilní – mají fixní kapacitu a podporují pouze omezené datové šířky. FPGA obsahují desítky až stovky blokových pamětí.
   Blokové paměti v iCE40 LP8K mají 4 kb a podporují tyto datové šířky: 256 adres po 16 bitech, 512 adres po 8 bitech, 1024 adres po 4 bitech nebo 2048 adres po dvou bitech.
4. Distribuovaná RAM (Distributed RAM) – umožňuje využít LUTy jako paměť. Tento typ paměti je mnohem flexibilnější než bloková paměť, ale zabírá velké množství LUTů.
5. DSP (Digital Signal Processing) – různé HW bloky používané pro zpracování obrazu.
6. Fixní HW – různé řadiče, sběrnice nebo rovnou celá procesorová jádro (u velmi výkonných FPGA).

5. ZAPOJENÍ

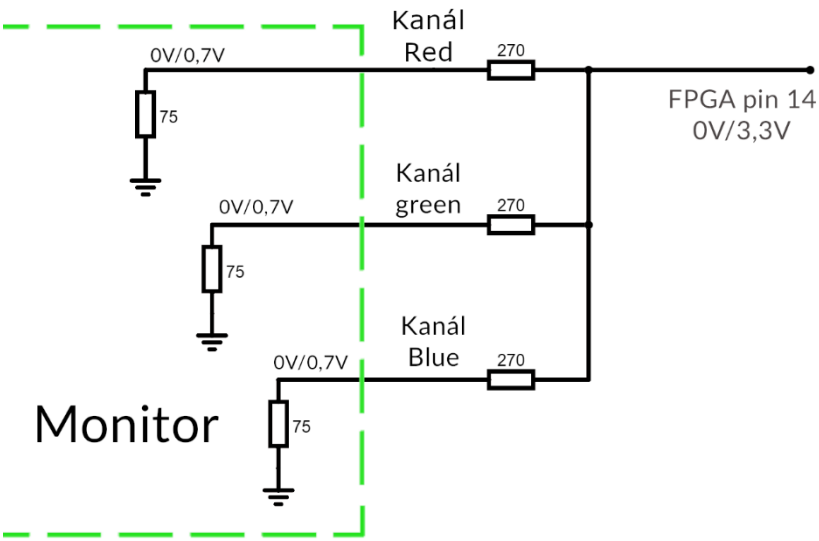


Obrázek č. 6 - Schéma zapojení FPGA, zdroj fotky FPGA: https://www.crowdsupply.com/tinyfpga/tinyfpga-bx/updates/manufacturing-continues

Na pravé straně FPGA na obrázku č. 6 lze vidět debugovací logiku – osmipinový DIP switch a jeden mikrospínač. DIP switch slouží pro nastavení debugovacího režimu (viz kapitola *Debugovací logika*) a mikrospínač slouží pro krokování procesoru v debugovacím režimu. DIP switch a tlačítko jsou připojeny na GND a využívají interní pull up rezistory v FPGA.

Na levé straně FPGA je umístěn VGA a PS/2 konektor. Jelikož FPGA pracuje s 3,3V logikou, musí signály pro PS/2 a VGA konektor projít skrz převodník.

PS/2 konektor je připojen na 5V (které poskytuje pin Vin od FPGA) a GND, jeho linka s hodinovým signálem a datová linka procházejí převodníkem z 5V na 3,3V logiku a následně na piny 12 a 13.

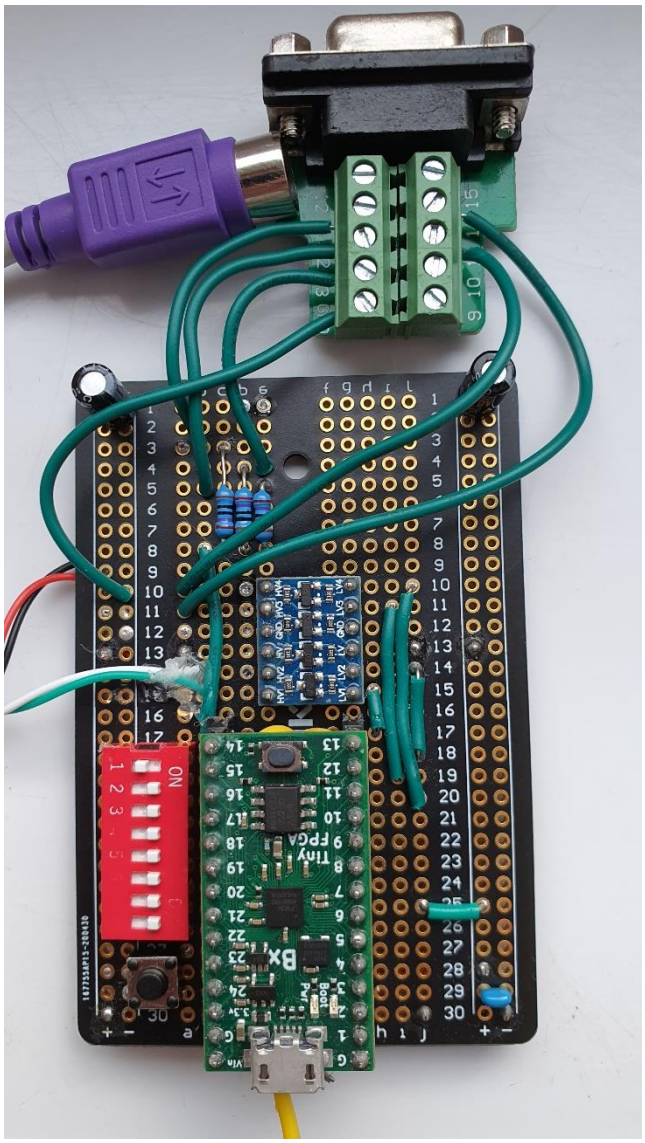


Obrázek č. 7- Schéma odporového děliče pro kanály RGB

Časovací signály rozhraní VGA (signály V-Sync a H-Sync) jsou generovány na pinech 11 respektive 10, následně projdou skrz převodník a pokračují na piny 14 respektive 13 VGA konektoru.

Pin GND od VGA konektoru je připojen na GND. Piny 1, 2 a 3 (kanály Red, Green a Blue) jsou analogové signály (rozmezí 0 V až 0,7 V) ovládající barvu pixelu posílaného na displej. Proto jsou připojeny přes odporové děliče (viz obrázek č. 8) na Pin 14 od FPGA. Pokud je na pinu 14 log 1 (3,3 V), na všech RGB kanálech je 0,7 V, pixel tedy svítí bíle. Pokud je na pinu 14 log 0 (0 V), na všech kanálech je 0 V a pixel tedy nesvítí (je černý).

Na obrázku č. 8 lze vidět reálné zapojení FPGA na pájivém poli.[2]



Obrázek č. 8 - Zapojení FPGA na pájivém poli

[2] Pod FPGA si lze povšimnout žlutých vodičů – jedná se o vodiče SPI sběrnice, které jsem používal pro připojení osciloskopu při zprovozňování SPI rozhraní.

6. INSTRUKČNÍ SADA

Instrukční sada (ISA – Instruction Set Architecture) představuje něco jako jazyk procesoru, a popisuje programátorovi, jak může procesor naprogramovat.

ISA zajišťuje, že rozdílné procesory (např. procesory Intel a AMD), jsou schopné spustit stejný kód, protože jsou postaveny na stejné instrukční sadě a mluví tedy „stejným jazykem“.

Instrukční sada obvykle definuje tyto vlastnosti procesoru:

- Počet, šířku a určení registrů
- Seznam dostupných instrukcí
- Uspořádání paměti

Tradičně se ISA rozdělují na CISC (Comnplex Instruction Set Computer) a RISC (Reduced Instruction Set Computer).

6.1. CISC

Instrukční sady CISC se vyznačují velmi vysokým počtem instrukcí (několik stovek až víc jak tisíc), rozdílnou šířkou instrukcí (např. instrukce x86 můžou být široké od 8 bitů až po 128 bitů) a podporou mnoha adresovacích módů. Ukázkou této ISA je např. právě instrukční sada x86.

6.2. RISC

Instrukční sady RISC jsou typicky mnohem jednodušší. Šířka instrukcí je často fixní, počet instrukcí je několik desítek (maximálně pár stovek). Většinou se jedná o tzv. load-store architektury – číst a zapisovat do paměti můžou pouze instrukce load a store. Zástupci jsou např. MIPS, ARM nebo RISC-V.

6.3. Instrukční sada RISC-V

RISC-V je open source instrukční sada (její vývoj tedy neřídí žádná velká společnost, ale nezisková asociace se sídlem ve Švýcarsku) jejíž vývoj začal v roce 2010 (je tedy dost mladá). Její využití (i komerční) je zdarma, takže není nutné platit licenční poplatky (na rozdíl od instrukční sady ARM).

Díky své otevřenosti si tato instrukční sada získala velkou podporu (tuto instrukční sadu využívá například firma nVidia nebo Western Digital), a její popularita v současné době stále roste – z tohoto důvodu jsem si pro můj procesor vybral právě instrukční sadu RISC-V, jelikož se jedná o velmi perspektivní volbu.

Existují čtyři varianty této instrukční sady:

1. RV32E – osekaná 32bitová verze určená pro embedded systémy (vestavěné systémy – bankomaty, ovládání průmyslových linek, automobilový průmysl). Hlavní rozdíl oproti variantě RV32I je snížení počtu registrů z 32 na 16. Její specifikace v době psaní práce ještě nebyla uzavřena.
2. RV32I –32bitová verze
3. RV64I – 64bitová verze
4. RV128I – 128bitová verze, specifikace v době psaní práce nebyla ještě uzavřena

Pro můj procesor jsem si vybral právě variantu RV32I, jelikož nepotřebuji 64bitový adresní prostor (a navíc 64bitový procesor by zabíral o dost více logiky) a RV32E nemá uzavřenou specifikaci.

Varianta RV32I má dostupných i několik instrukčních rozšíření – např. F (pro operace s floating point (desetinnými) čísly), M (instrukce pro násobení a dělení celočíselných hodnot) nebo C (obsahuje 16bitové verze některých instrukcí, díky čemuž umožňuje zmenšit velikost programu).

Na obrázku č. 9 lze vidět seznam instrukcí v instrukční sadě RISC-V, přesněji její variantě RV32I. Tato varianta obsahuje 40 instrukcí, z nichž procesor implementuje 37 (vyjma instrukcí FENCE, ECALL a EBREAK, které nejsou potřeba, jelikož na procesoru nepoběží žádný operační systém).

Každá instrukce má 32 bitů. Malá tabulka na vrchní straně obrázku ilustruje, jak vypadá šest šablon typů instrukcí v RV32I (R, I, J, S, U a B) a jaká jsou jejich jednotlivá pole. Velká tabulka umístěná níže vypisuje, jak vypadají konkrétní instrukce –procesor podporuje všechny instrukce, až na tři nejspodnější.

- Sedmibitové pole opcode stanovuje typ instrukce
- Pole rd stanovuje adresu cílového registru (kam se ukládají data)
- Pole rs1 a rs2 jsou adresy registrů z kterých se čte
- Pole funct3 a funct7 blíže specifikují typ instrukce
- Pole imm (které je různě rozházené ve všech formátech) tvoří konstantu (např. instrukce „ADDI rd, rs1, 5“ přičte k registru rs1 konstantu 5 a výsledek uloží do registru rd)

Detailní popis všech instrukcí lze najít v manuálu k instrukční sadě RISC-V [7], z kterého jsem při tvoření procesoru čerpal.

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Instruction Set

imm[31:12]					rd	0110111	LUI
imm[31:12]					rd	0010111	AUIPC
imm[20|10:1|11|19:12]					rd	1101111	JAL
imm[11:0]			rs1	000	rd	1100111	JALR
imm[12|10:5]		rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]		rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]		rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]		rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]		rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]		rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]			rs1	000	rd	0000011	LB
imm[11:0]			rs1	001	rd	0000011	LH
imm[11:0]			rs1	010	rd	0000011	LW
imm[11:0]			rs1	100	rd	0000011	LBU
imm[11:0]			rs1	101	rd	0000011	LHU
imm[11:5]		rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]		rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]		rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]			rs1	000	rd	0010011	ADDI
imm[11:0]			rs1	010	rd	0010011	SLTI
imm[11:0]			rs1	011	rd	0010011	SLTIU
imm[11:0]			rs1	100	rd	0010011	XORI
imm[11:0]			rs1	110	rd	0010011	ORI
imm[11:0]			rs1	111	rd	0010011	ANDI
0000000		shamt	rs1	001	rd	0010011	SLLI
0000000		shamt	rs1	101	rd	0010011	SRLI
0100000		shamt	rs1	101	rd	0010011	SRAI
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB
0000000		rs2	rs1	001	rd	0110011	SLL
0000000		rs2	rs1	010	rd	0110011	SLT
0000000		rs2	rs1	011	rd	0110011	SLTU
0000000		rs2	rs1	100	rd	0110011	XOR
0000000		rs2	rs1	101	rd	0110011	SRL
0100000		rs2	rs1	101	rd	0110011	SRA
0000000		rs2	rs1	110	rd	0110011	OR
0000000		rs2	rs1	111	rd	0110011	AND
fm	pred	succ	rs1	000	rd	0001111	FENCE
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK

Obrázek č. 9 - List RV32I instrukcí, zdroj: https://github.com/riscv/riscv-isa-manual/releases/tag/draft-20210316-ad5f04d

7. BLOKOVÉ SCHÉMA PROCESORU

Procesor se dělí na čtyři části, které lze vidět na obrázku č. 10 s blokovým schématem procesoru[3]:

1. Jádro procesoru (CPU core)
2. Paměťový subsystém (Memory subsystem)
3. Obrazový procesor (Display engine)
4. Obvod pro klávesnici (Keyboard)

Celý procesor běží na frekvenci 16 MHz (základní frekvence na mnou vybraném FPGA), pouze VGA obvod běží na frekvenci 40 MHz, jak stanovuje standard VGA pro dané rozlišení.

Na pravé straně schématu se nachází ta nejdůležitější část – jádro procesoru. Zde se provádí všechny aritmetické operace a zde se vykonává program.

Jádro procesoru je napojeno na instrukční cache, z které se do jádra načítají instrukce, a na datovou sběrnici umístěnou v paměťovém subsystému.

Paměťový subsystém propojuje dohromady jednotlivé části procesoru. Umožňuje jádru přístup do datové cache (datová cache obsahuje data vykonávaného programu), zapisovat do videopaměti, přečtení aktuálního času uplynulého od spuštění procesoru nebo přečtení zmáčknuté klávesy z bufferu pro klávesnici.

Obvod obsluhující klávesnici funguje jednoduše. PS/2 řadič komunikuje s klávesnicí a hlídá, zda byla zmáčknuta některá klávesa. Zmáčknutá klávesa se následně přeloží z tzv. scancode (kód zmáčknuté klávesy, který zašle klávesnice) do ASCII. Tento znak se následně uloží do bufferu, který může procesor kdykoliv přečíst a zjistit tak, která klávesa byla zmáčknuta. Pokaždé když procesor buffer přečte, dojde k vyčištění tohoto bufferu.

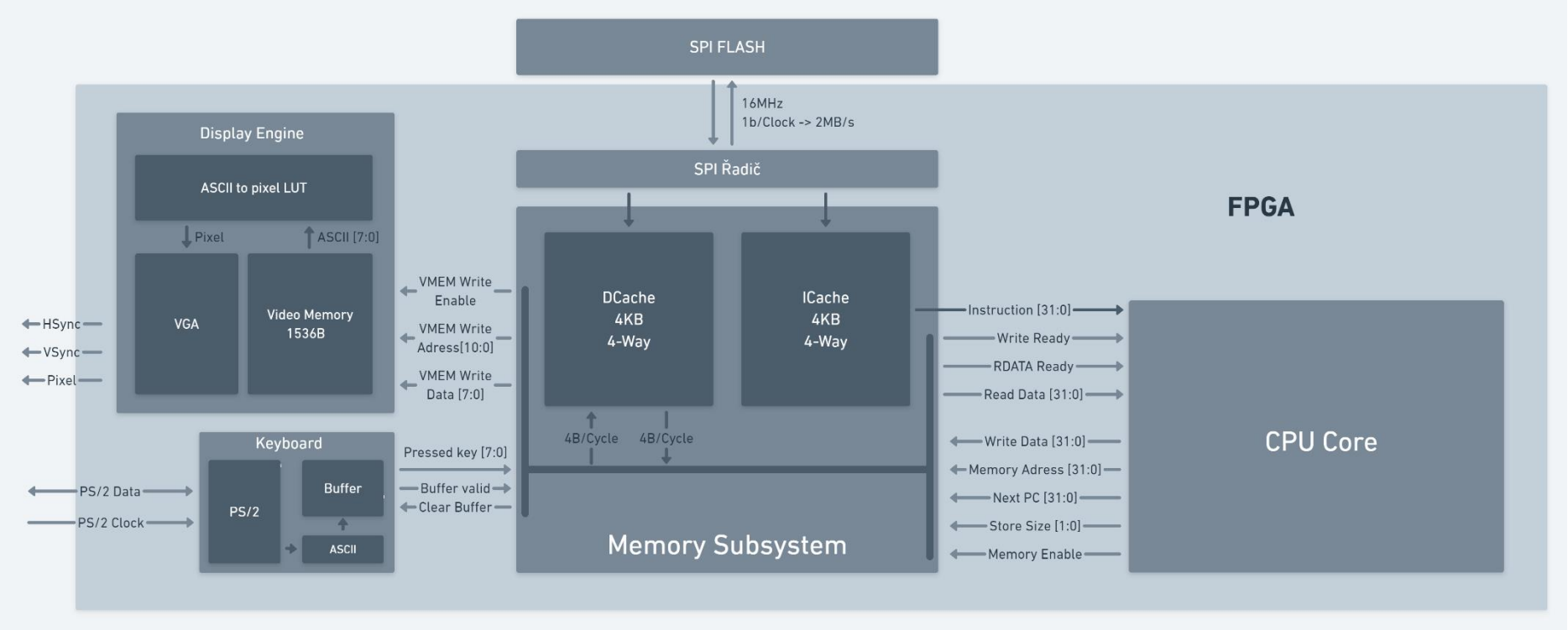
Display Engine (obrazový procesor) se skládá ze tří částí:

1. Videopaměť, do které procesor zapisuje ASCII znaky, které se mají tisknout na displej. Tato paměť má kapacitu 1500 Bytů.
2. Převodník z ASCII znaků na řádky pixelů.
3. VGA obvod, který řídí komunikaci s monitorem a vysílá na něj pixely a časovací signály V-Sync a H-Sync.

[3] Jedná se o zjednodušené blokové schéma, které plně neodpovídá reálnému zapojení procesoru (např. pro přehlednost jsou vynechány některé nepodstatné signály).

Datová a instrukční cache jsou napojeny na SPI řadič, který v případě potřeby komunikuje s SPI Flash pamětí o velikosti 1 MB a načítá z ní uživatelský program a data.

Více informací k jednotlivým obvodům lze nalézt v jejich příslušných kapitolách.



Obrázek č. 10 - Blokové schéma procesoru

8. HDL

Původně byly procesory a jiné logické obvody navrhovány ručně – inženýři kreslili schémata, podle kterých se následně vyráběly integrované obvody v továrnách. Avšak kvůli rostoucí míře integrace začaly integrované obvody postupně obsahovat tisíce, nebo i desetitisíce, tranzistorů, a tak kreslení schémat přestalo být vhodným řešením.

Schémata byla postupně nahrazena počítači – inženýři začali procesory obvody psát v tzv. HDL jazycích (Hardware Description Language) – inženýr napsal HDL kód pro kombinační/sekvenční obvod, a počítač sám jejich kód převedl na logická hradla, registry a tranzistory.

Ačkoliv se HDL jazyky na první pohled podobají programovacím jazykům, jsou konceptuálně odlišné. Základní vlastností programovacích jazyků bývá, že se vykonávají řádek po řádku – sekvenčně. Oproti tomu v HDL jazycích se všechny řádky vykonávají najednou.

Nejpoužívanější jsou dva HDL jazyky – Verilog (potažmo System Verilog) a VHDL.

8.1. VHDL

Počátky VHDL se datují přibližně do roku 1983. Oproti Verilogu má VHDL o dost striktnější syntax a je hodně „upovídaný“ (logický obvod napsaný ve VDHL má více řádků kódu, než stejný obvod napsaný v System Verilogu), z těchto důvodů jsem pro vývoj upřednostnil právě System Verilog.

8.2. (System) Verilog

Verilog pochází přibližně ze stejné doby jako VHDL. Jeho syntax je velmi podobný programovacímu jazyku C.

V roce 2005 byl vytvořen System Verilog, který představuje aktualizovanou verzi Verilogu. System Verilog je supersetem Verilogu, tj. obsahuje celý Verilog, je s ním plně kompatibilní, opravuje některé jeho nedostatky a přidává pokročilé funkce. Dnešní komerční (a tím pádem i velmi drahé) vývojové nástroje nemají s podporou System Verilogu problém, ale podpora v open source nástrojích dost pokulhává a často úplně chybí.

Na straně 26 je ukázka modulu napsaného v System Verilogu. Na začátku je keyword „`module`“ následovaný názvem modulu „`branch_unit`“ (jednotka pro výpočet větví). V závorce za názvem modulu následuje výčet portů, jejich směr (vstup/výstup, respektive `input` / `output`), jejich typ (ve Verilogu `reg` / `wire`, v System Verilogu prakticky vždy `logic`) a jejich bitová šířka (v hranatých závorkách).

Můžeme tedy vidět, že tento modul má dva 32bitové vstupy `rd1` a `rd2` (jedná se o dvě 32bitová čísla přečtená z registrů), jeden tří bitový vstup `funct3` (identifikuje typ instrukce) a jeden jednobitový výstup `branch_taken` (pokud nabývá log 1, větev je přijata, pokud nabývá log 0, větev není přijata).

Za definicí portů následuje **`always_comb begin....end`**, který říká, že logika mezi slovy **`begin`** a **`end`** je kombinační logika – nepoužívá tedy registry ani žádný jiný typ paměti (pro sekvenční obvody se používá **`always_ff`** `@ (`**`posedge`** `CLK)` **`begin....end`**, kde **`posedge`** `CLK` znamená, že se data do registrů ukládají při vzestupné hraně hodinového signálu CLK).

V bloku **`always_comb`** se nachází **`case`**`(funct3)`, který je velmi podobný klasickému switch/case z jazyka C.

Pokud vstup `funct3` nabývá logické hodnoty:

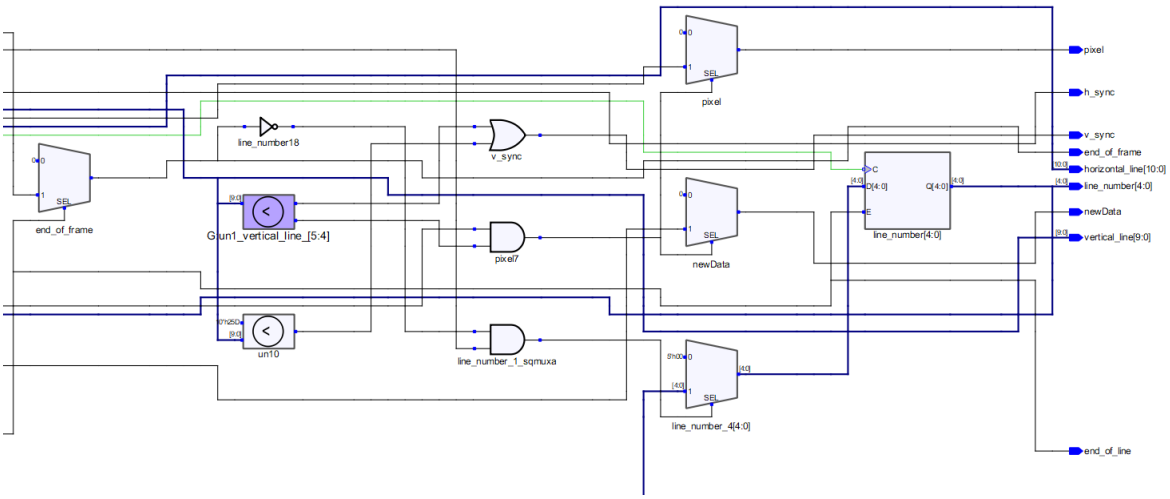
1. „000“, jedná se o instrukci BEQ (Branch on Equal) – výstup `branch_taken` je log 1, pokud se vstupy `rd1` a `rd2` rovnají.
2. „001“, jedná se o instrukci BNE (Branch on Not Equal) – výstup `branch_taken` je log 1, pokud se vstupy `rd1` a `rd2` nerovnají).
3. Další instrukce nepodstatné pro ukázku kódu
4. „111“, jedná se o instrukci BGEU (Branch on Greater Than or Equal, Unsigned) – výstup `branch_taken` je log 1, pokud je vstup `rd1` větší nebo roven vstupu `rd2`, s tím že `rd1` a `rd2` jsou typu Unsigned, tj. jsou nezáporné.
5. „**`default`**“ default definuje co udělat, pokud se hodnota funct3 nerovná ani jedné z výše zmíněných kombinací.

Na konci modulu jsou slova **`endcase`**, **`end`** a **`endmodule`**, které ukončují **`case`**, respektive **`always_comb`** a **`module`**.

```
module branch_unit(
  input logic [31:0] rd1, rd2,
  input logic [2:0] funct3,
  output logic branch_taken
);


always_comb begin

  case(funct3)

    3'b000: begin        //BEQ

      if( $signed(rd1) ==  $signed(rd2) ) branch_taken = 1;
      else branch_taken = 0;

    end

    3'b001: begin       //BNE

      if( $signed(rd1) ==  $signed(rd2) ) branch_taken = 0;
      else branch_taken = 1;

    end
          .
          .
          .
          .
          .

    3'b111: begin       //BGEU

      if(rd1 >= rd2) branch_taken = 1;
      else branch_taken = 0;

    end

    default: branch_taken = 0;

  endcase

end
endmodule
```

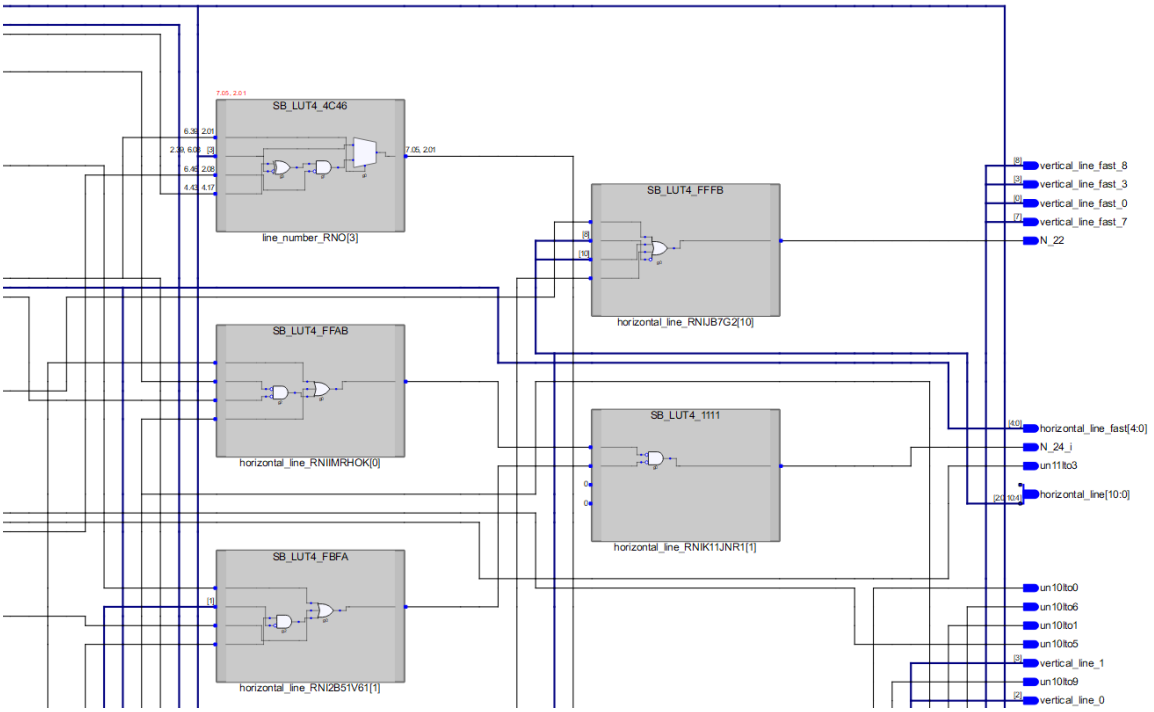
9. JAK FUNGUJE „PŘEKLAD“ KÓDU PRO FPGA

HDL kód je potřeba „přeložit“ tak, aby se mohl nahrát na FPGA. Tento překlad probíhá v několika krocích. Na začátku je HDL kód zpracován syntetizátorem, který z něj vytvoří obvod složený z logických hradel. Na obrázku č. 11 lze vidět malou část VGA obvodu.



Obrázek č. 11 - Výsledek syntézy logických hradel

Následně syntetizátor vezme schéma s logickými hradly, a přeloží ho do technologie daného FPGA – LUTy, blokové paměti, PLL obvody a další bloky. Na obrázku č. 12 lze vidět čtyřvstupové LUTy na FPGA iCE40 LP8K.



Obrázek č. 12- Výsledek syntézy technologie FPGA

Po syntetizátoru si obvod vezme do péče placer („umisťovač“), který LUTy a jiné bloky umístí na FPGA. Pokud dané FPGA nemá dostatek prostředků pro implementování

daného obvodu, placer bude hlásit chybu. Po placeru je na řadě router („propojovač“), který jednotlivé bloky propojí (placer a router jsou často spojeny do jednoho kroku P&R). Podobně jako u placeru, pokud router nebude schopný propojit všechny bloky, zahlásí také chybu.

Výstupem P&R je tzv. netlist. Po P&R se provede analýza, zda je daný obvod schopný běžet na zadané frekvenci – pokud ne, je potřeba obvod upravit/optimalizovat, nebo v krajním případě koupit silnější FPGA.

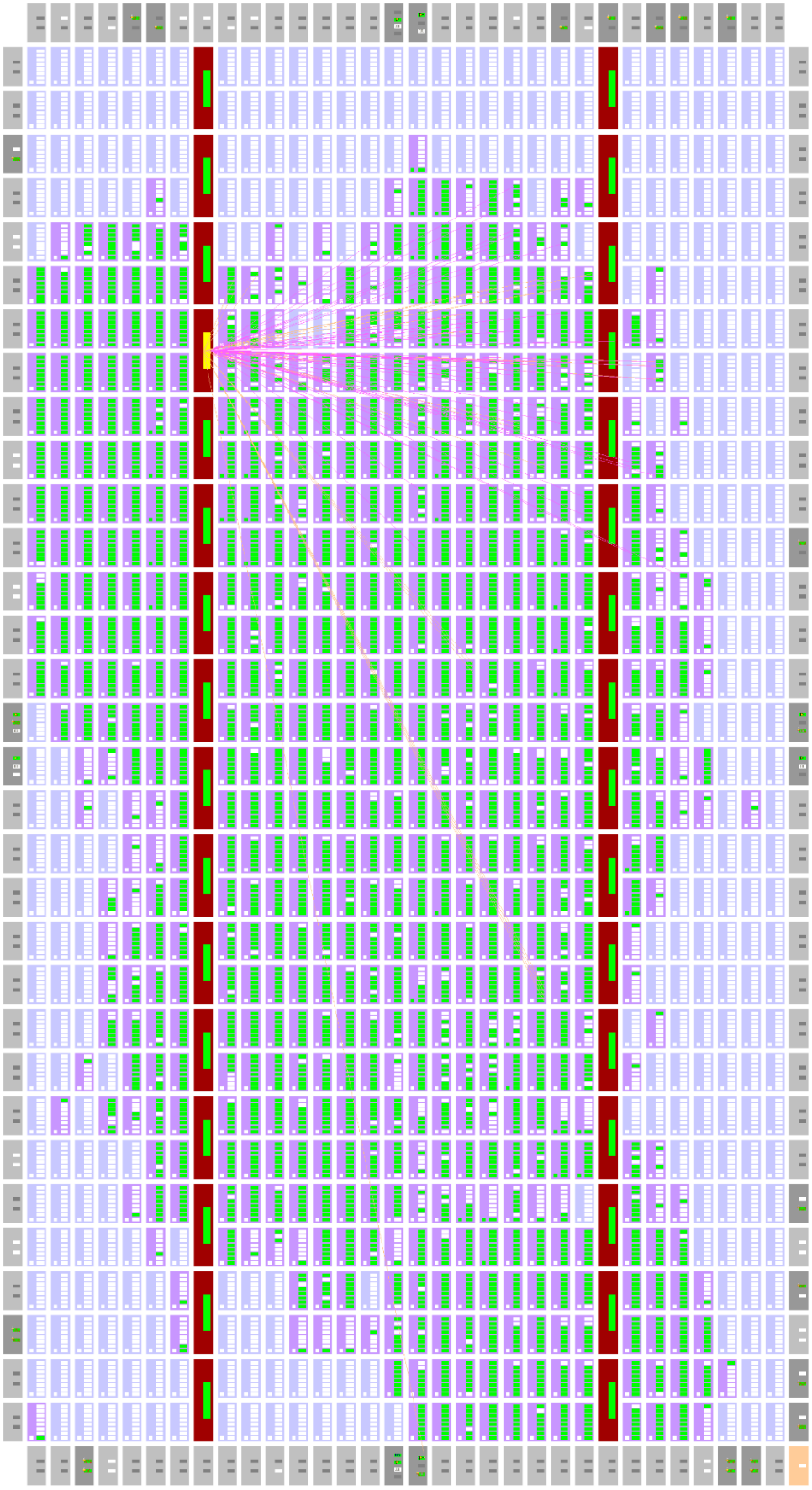
Na úplném konci procesu překladu kódu se z netlistu vytvoří bitstream – soubor jedniček a nul, který se nahraje na flash paměť přítomnou na FPGA desce. FPGA bitstream při každém startu přečte, a nastaví podle něj jednotlivé LUTy a všechny programovatelné části.

Na obrázku č. 13 lze vidět výsledek P&R pro můj procesor – světle a tmavě fialové bloky jsou jednotlivé PLB, malé zelené tečky jsou jednotlivé LUTy. Protáhlé červené bloky jsou blokové paměti – je jich 32, a všechny jsou využity. Jedna bloková paměť (část pole registrů v jádře) je i rozkliknuta, takže lze vidět, kam z ní vedou jednotlivé vodiče.

Procesor na FPGA zabírá:

- 4101 z 7680 LUTů
- 544 z 960 PLB
- 32 z 32 blokových pamětí
- 1 z 1 PLL

Jde tedy vidět, že FPGA ještě není úplně plné a zvládlo by i jemně vylepšený procesor.



Obrázek č. 13- Výsledek P&R procedury

10. VÝVOJOVÉ NÁSTROJE

Vývoj obvodů na FPGA jde ruku v ruce s vývojovými nástroji – bez syntetizátorů logiky by bylo nemožné vytvořit bitstream pro FPGA a bez simulátorů by bylo nemožné ladit funkci logických obvodů.

10.1. ModelSim PE Student Edition

ModelSim je RTL (Register Transfer Level – simulace na úrovni registrů) simulátor s podporou velkého množství HDL jazyků. Já jsem používal jeho zdarma dostupnou studentskou verzi.

Simulátor se používá, jak už název napovídá, k simulaci kódu před nahráním na FPGA – v obvodu nahraném na FPGA se opravdu těžko hledají chyby, simulátor má tu výhodu, že umožňuje zobrazit všechny signály a jejich přesný průběh v čase.

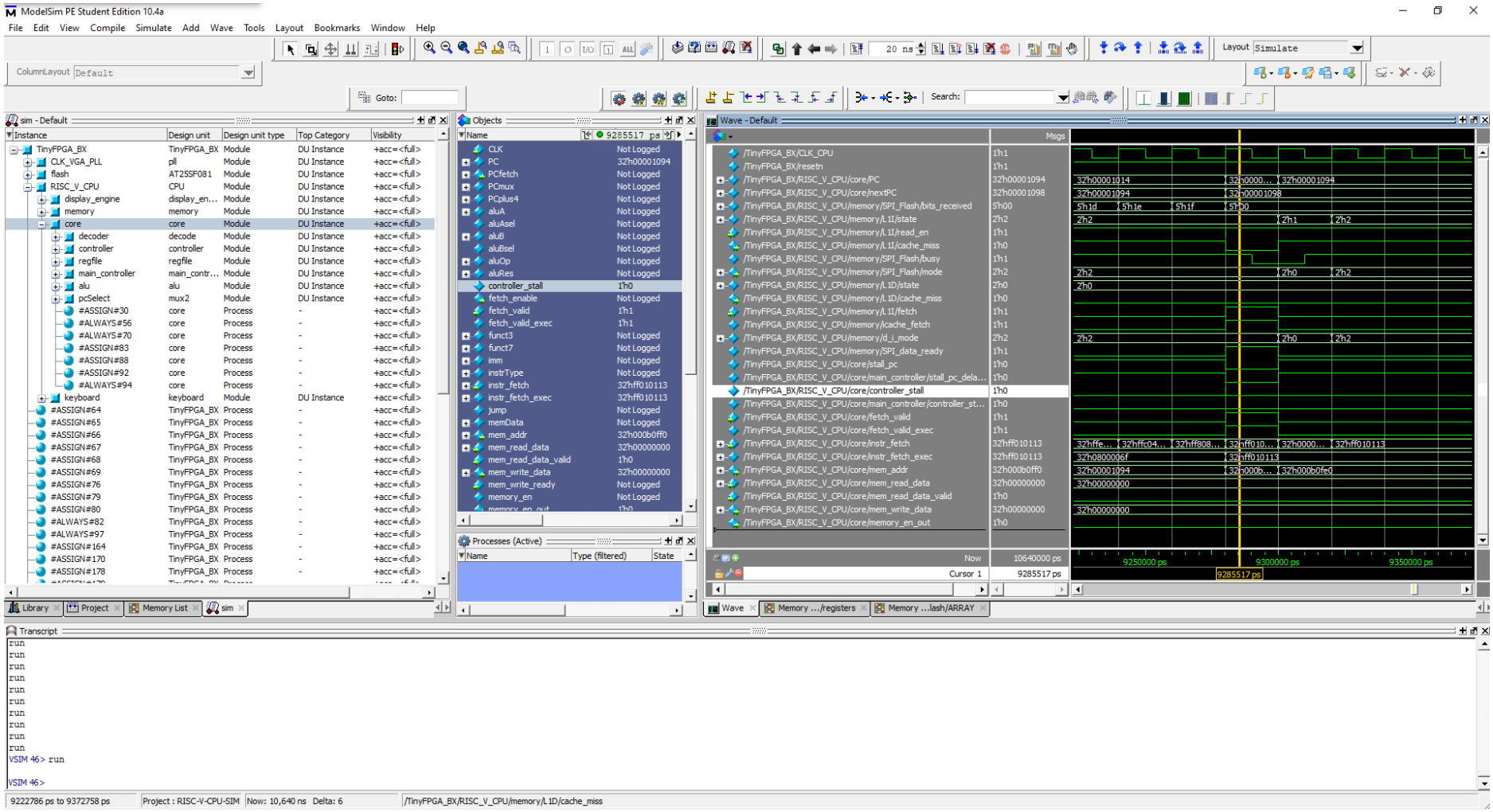
Simulaci lze provádět v různých fázích překladu HDL kódu:

1. Před syntézou (Pre-synthesis) – asi nejobvyklejší způsob simulace. Simulátor zkompiluje HDL kód a následně simuluje jeho chování. Největší nevýhodou této simulace je fakt, že může dojít k tzv. „simulation missmatch“, kdy se simulovaný obvod chová jinak, než ten syntetizovaný na FPGA (a je tedy nemožné v simulátoru objevit některé chyby). Takovéto chyby je následně obzvláště těžké odhalit a odstranit.
2. Po syntéze (Post-synthesis) – simulátor nesimuluje HDL kód, ale simuluje jeho syntetizovanou verzi (kterou vytvořil syntetizátor)[4], a nemůže tedy dojít k „simulation missmatch“.
3. Po P&R (Post place and route) – simulátor simuluje přesně takový obvod, který se nahraje na FPGA. Zároveň toto je jediný typ simulace, který je schopný simulovat časování obvodu (a odhalit tedy např. že obvod není schopný běžet na vybrané hodinové frekvenci).

[4] V rámci ladění procesoru jsem vyzkoušel i tento typ simulace, v naději, že v něm půjdou jednoduše odhalit chyby, které se neukazují v běžné simulaci. Ale zjistil jsem, že i simulace syntetizovaného obvodu má své nevýhody:

1. Hlavním problémem je, že syntetizátor během syntézy změní různým signálům jejich názvy, a obecně všechny obvody úplně „překope“ k nepoznání – měl jsem velké problémy se v procesoru následně zorientovat, a nemohl jsem lokalizovat signály, které mě zajímaly.
2. Tento typ simulace je velmi pomalý.

Nakonec jsem od tohoto typu simulace odstoupil, a vymyslel jsem lepší řešení ladění procesoru – viz kapitola *Debugovací logika*



Obrázek č. 14 - Simulace v programu Modelsim

10.2. iCEcube2

iCEcube2 je balíček nástrojů od výrobce Lattice pro FPGA řady iCE40. Vyjma toho, že je schopný vygenerovat bitstream pro FPGA řady iCE40, obsahuje také syntetizátor Synopsys Synplify (který jsem hojně využíval pro kontrolu, jak se kód syntetizoval) a simulátor Active-HDL, který jsem nevyužíval, jelikož jsem již měl dřívější zkušenosti se simulátorem Modelsim, a připadal mi jako lepší volba.

Další součástí je generátor parametrů pro PLL modul – bohužel ten v mém případě nefungoval. Nástroj má fungovat tak, že mu zadáte vstupní frekvenci PLL modulu (16MHz) a požadovanou výstupní frekvenci PLL modulu (40MHz pro VGA obvod), a nástroj vám sám vygeneruje PLL modul se správnými parametry. Takto vygenerovaný PLL obvod ale nefungoval (na jeho výstupu nebyl žádný hodinový signál) – toto mě donutilo dočasně přejít z iCEcube2 na open source syntetizátor Yosys (viz níže), kde vygenerovaný PLL obvod fungoval správně.

Nakonec se mi díky nástroji „icepll“ z projektu IceStorm [6] povedlo vygenerovat PLL obvod, který fungoval i v iCEcube2, a tak jsem se vrátil zpět k tomuto vývojovému nástroji.

10.3. Yosys + nextpnr

Yosys je open source syntetizátor, který je používán v kombinaci s open source place and route nástrojem „nextpnr“, pokud z nějakého důvodu nechcete využívat nástroje od výrobce (např. když generují neoptimální design nebo když obsahují chyby).

V budoucnu bych chtěl vyzkoušet „portnout“ procesor zpět na tuto sadu nástrojů, a porovnat, která sada nástrojů vygeneruje lepší obvod (bude zabírat méně HW prostředků, poběží na vyšší frekvenci).

10.4. Verilator

Verilator je open source simulátor pro obvody napsané ve Verilogu (s omezenou podporou System Verilogu[5]). V mém případě jsem Verilator nepoužíval jako simulátor, ale jako tzv. Linter – nástroj pro kontrolu chyb v napsaném kódu. Díky Verilatoru jsem v návrhu CPU našel několik závažných chyb, které by jinak bylo velmi složité odhalit.

[5] Verilator podporuje velkou část System Verilogu, ale chybí mu podpora některých funkcí. Jelikož já při vývoji procesoru tyto pokročilé funkce nevyužíval, Verilator neměl s mými zdrojovými kódy problém. S čím ale problém měl, byly knihovny od výrobce FPGA – ty sice jsou napsané v čistém Verilogu, ale používají některé konstrukty (např. weak, strong), pro které Verilator nemá podporu. Musel jsem tedy tyto knihovny projít pomocí CTRL + F (asi 27 000 řádků kódu) a tyto konstrukty odstranit.

11. JÁDRO PROCESORU

Jak už bylo řečeno, procesor je postaven na instrukční sadě RISC-V. Právě proto jsem při návrhu vycházel z manuálu k této instrukční sadě [7], kde je přesně popsáno, co má která instrukce dělat. Zároveň jsem se nechal inspirovat základní strukturou jádra (rozdělení do jednotlivých bloků) z knihy „Digital Design and Computer Architecture“ [8], kde autoři postavili osekaný 32bitový procesor na bázi instrukční sady MIPS, který uměl 5 instrukcí.

Původním cílem bylo navrhnout standardní jednoduché jedno cyklové jádro (každá instrukce se vykonává jeden cyklus), avšak toto se změnilo v poslední fázi vývoje. Přidáním debugovací logiky (viz kapitola *Debugovací logika*) narostla velikost procesoru natolik, že se nevlezl na FPGA, a byl jsem tedy nucen udělat změny, které vedly k tomu, že se z jádra stalo jádro více cyklové.

Pomocí Synopsys Synplify jsem se tedy snažil odhalit, která část procesoru zabírá nejvíce prostředků (a je tedy nejlepším kandidátem k optimalizaci), a k mému překvapení jsem zjistil, že se jedná o jednu z nejjednodušších komponent – pole třiceti-dvou 32bitových registrů. Tato komponenta si sama o sobě brala více než 2000 LUTů, a k tomu velké množství pamětí Flip Flop.

Řešením bylo změnit architekturu pole registrů – nahradit paměti Flip Flop blokovými pamětmi. Toto řešení přineslo velké množství výhod:

1. Velikost procesoru se zmenšila na polovinu – uvolnilo se tedy obrovské množství LUTů.
2. Zlepšilo se časování, a jádro procesoru je tedy schopné běžet na vyšší frekvenci.
3. Vznikl potenciál pro předělání jádra a využití tzv. pipeliningu (zřetězení instrukcí), kdy jádro procesoru pracuje na více instrukcích najednou.

Zároveň toto řešení přineslo určité nevýhody:

1. Neefektivní využití blokových pamětí. Kvůli potřebné propustnosti (64 bitů za cyklus) bylo nutné využít čtyři blokové paměti (každá má propustnost 16 bitů za cyklus a kapacitu 4 kilobity, dohromady mají propustnost 64 bitů za cyklus a kapacitu 16 kilobitů).
   Pole registrů má velikost 31x 32 bitů[6] = 992 bitů. Je tedy poměrně neefektivní takovou to paměť implementovat z blokových pamětí s kapacitou 16 384 bitů, ale bohužel TinyFPGA BX jinou možnost nenabízí.

[6] Ačkoliv ISA RISC-V má 32 registrů, jelikož registr 0 je v této ISA vždy roven nule (a data v něm se tedy nemusí nijak ukládat), nezapočítává se do velikosti pole registrů (zabírá 0 bitů). „Reálně" tedy RISC-V obsahuje 31 registrů a konstantu 0.

2. Zatímco paměti Flip Flop mají latenci 0 cyklů a jejich data jsou okamžitě dostupná ke zpracování (a tudíž instrukce může být dokončena během jednoho cyklu), blokové paměti mají latenci 1 cyklus, a zpracování instrukcí se tedy roztáhne na 2 cykly.
   V prvním cyklu dojde vždy k načtení instrukce z instrukční cache, k dekódování instrukce a nastavení kontrolních signálů. Zpracování instrukce dále postupuje následovně:
   - Pokud instrukce nenačítá data z registrů, je dokončena v tomto cyklu.
   - Pokud instrukce načítá data z registrů (což drtivá většina instrukcí dělá), čeká se jeden cyklus na načtení dat, a až v následujícím cyklu je instrukce dokončena.

Schéma syntetizovaného jádra procesoru lze vidět na obrázku č. 15.

Každá instrukce nejdříve projde skrz dekodér. Dekodér je čistě kombinační obvod. Jeho úkolem je rozdělit instrukci do jednotlivých polí, shodně jako je zobrazeno na obrázku číslo 9 v kapitole *Instrukční sada*.

Následuje vedlejší kontrolér, který dle typu instrukce nastavuje kontrolní signály – zda se bude číst/zapisovat do paměti, zda se bude zapisovat do registrů nebo zda je daná instrukce skok (těchto kontrolních signálů je přibližně 10). Vedlejší kontrolér je taktéž čistě kombinační obvod.

Hlavní kontrolér již obsahuje sekvenční prvky. Úkolem hlavního kontroléru je procesor stallovat (zastavovat), v případě kdy např. čeká na novou instrukci, čeká na načtení dat z paměti, čeká na načtení dat z registrů nebo v debugovacím režimu čeká na stisk tlačítka.

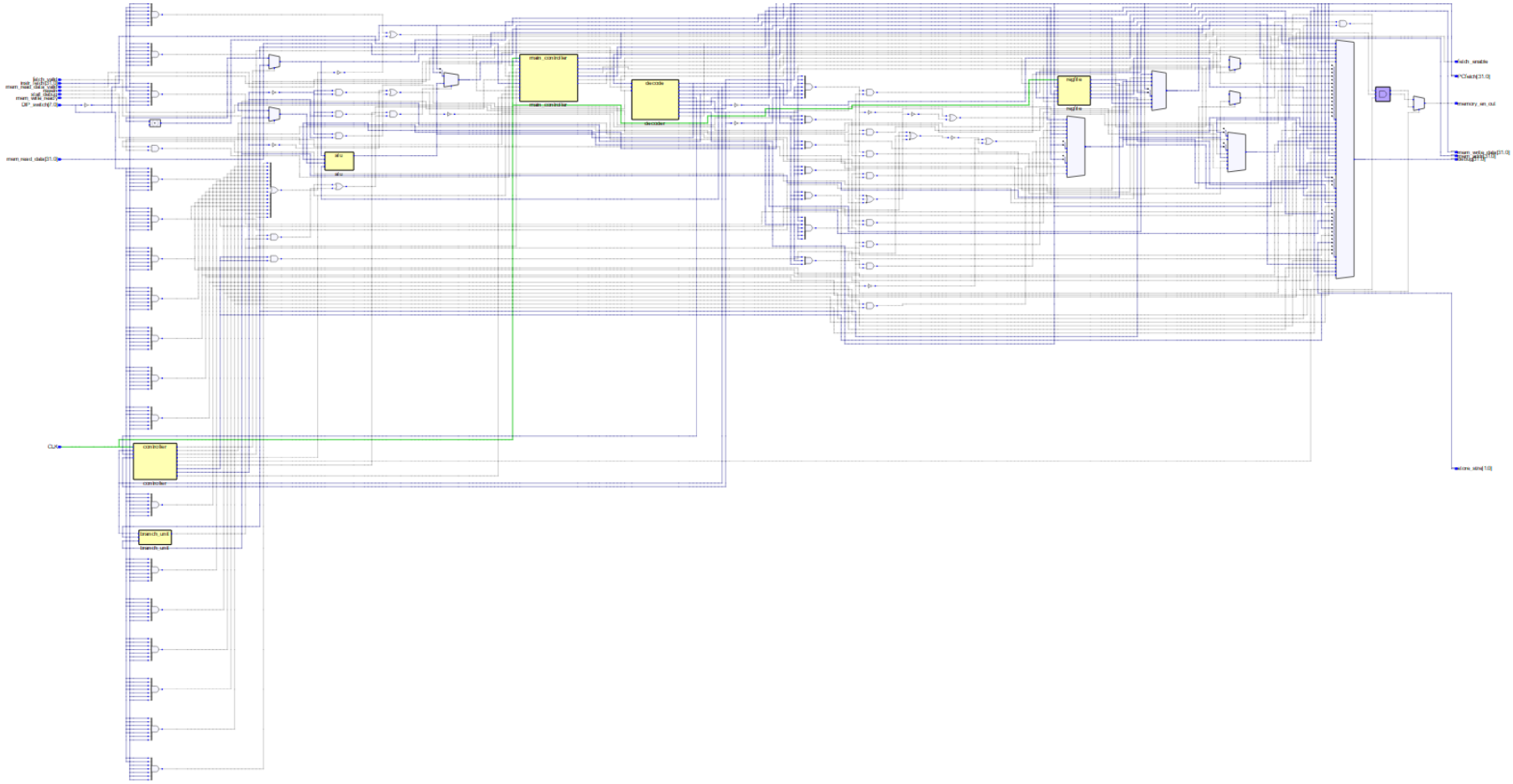
Dalším sekvenčním obvodem je pole registrů, kam si programy ukládají své proměnné. Instrukční sada RV32I má 32x 32bitových registrů. Implementace pole registrů již byla řešena výše v této kapitole.

V momentě, kdy je instrukce dekódovaná a všechny operandy z registrů jsou načtené, může dojít k provedení aritmetických operací v ALU (Arithmetic Logic Unit, Aritemticko-Logická jednotka).

ALU podporuje všechny operace přítomné v instrukční sadě RV32I – sčítání, odčítání, bitový AND, OR, XOR, bitové posuny (logický posun vlevo, aritmetický posun vpravo i vlevo) a porovnávání. ALU je čistě kombinační obvod.

Když ALU dokončí výpočty, je buď výsledek uložen zpět do pole registrů, nebo je předán load store jednotce, která ho uloží do paměti.

Poslední jednotkou v jádře je jednotka pro řešení větví (podmíněných skoků), ta je také čistě kombinační obvod, a zjišťuje, zda daná větev je či není přijata.

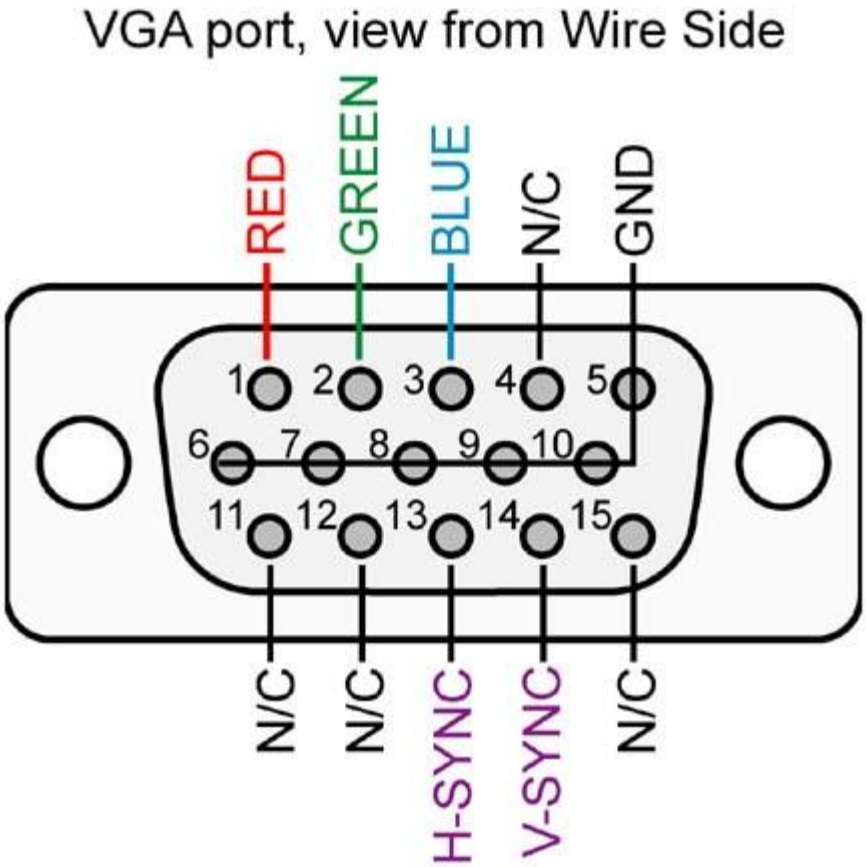


Obrázek č. 15 - Schéma syntetizovaného jádra

12. DISPLAY ENGINE

Návrh Display engine (displejového procesoru, dal by se přirovnat k extrémně jednoduché grafické kartě) byl asi nejprekérnější částí.

Pro výstup na displej jsem zvolil rozhraní VGA, a to zejména kvůli jeho jednoduchosti. Výstup na displej je černobílý, a to proto, aby se snížily nároky na velikost videopaměti. Rozlišení je fixních 800x600 pixelů.



Obrázek č. 16 - Pinout VGA konektoru, zdroj: https://www.bukalapak.com/p/kamera/converter-adapter/28i2ajl-jual-konektor-vga-male-plug-breakout-terminal-connector-15-pin-soket-vga

Rozhraní je tvořeno těmito piny:

1. Piny 1-3 – Analogové kanály Red, Green, a Blue s rozmezím napětí od 0V až do 0,7 V pro nastavení barvy posílaného pixelu. Jelikož výstup na displej je monochromatický, jsou tyto tři kanály spojeny do jednoho, a jejich napětí je buď 0V (posílá se černý pixel) nebo 0,7 V (posílá se bílý pixel)
2. Piny 5-10 – GND
3. Pin 13 – Horizontální synchronizace
4. Pin 14 – Vertikální synchronizace

Na obrázku č. 17 lze vidět znázornění, jak vypadá jeden snímek poslaný na monitor přes VGA rozhraní. Pixely (oblast označená „Display Time“) se na displej posílání zleva doprava, od shora dolů.[7]

Spolu s pixely se posílají i signály V-SYNC a H-SYNC, podle kterých monitor pozná, v jakém rozlišení vysíláme data. Tyto signály se generují podle standardu VGA [9] pro dané rozlišení, viz tabulka č. 1.

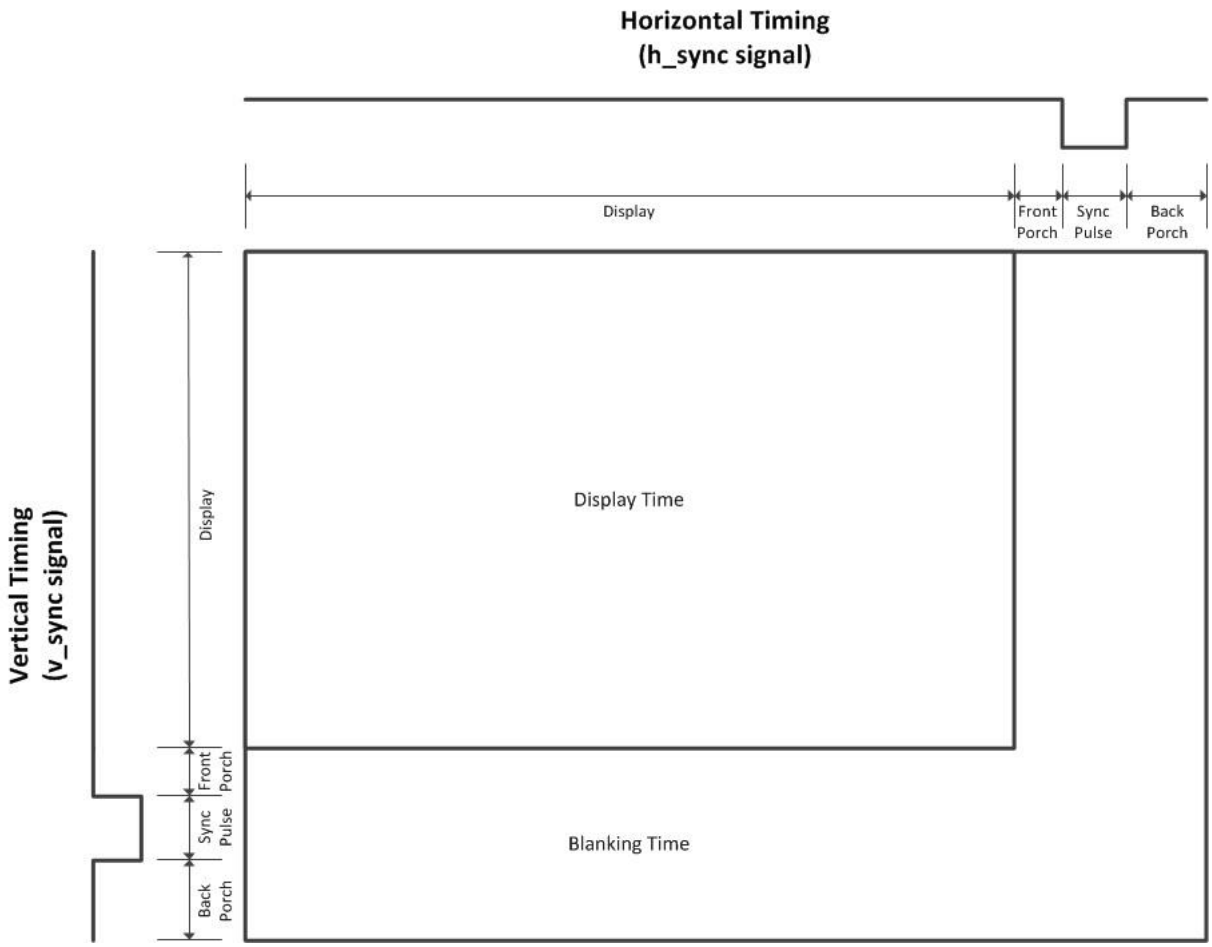
Rozlišení	800x600 pixelů
Obnovovací frekvence	60Hz
Pixel clock	40MHz

Horizontální pixely celkově	1056
Horizontální pixely viditelná část	800
Horizontální Front Porch	40
Horizontální Sync Pulse	128
Horizontální Back Porch	88

Vertikální pixely celkově	628
Vertikální pixely viditelná část	600
Vertikální Front Porch	1
Vertikální Sync Pulse	4
Vertikální Back Porch	23

Tabulka č. 1 - VGA časování

[7] Rozhraní VGA bylo původně navrženo pro CRT monitory, a proto je na obrázku č. 17 vidět tzv. „blanking time“. Tato oblast, kdy se na monitor neposílají žádné pixely, představuje čas určený pro návrat elektronového děla (emitoru elektronů) zprava doleva, respektive ze spodu nahoru, aby emitor mohl vysílat nový řádek pixelů.



Obrázek č. 17 - Diagram VGA časování, zdroj: https://www.hackster.io/dhq/fpga-camera-system-14d6ea

Původním cílem byl monochromatický (černobílý) VGA výstup na monitor, ve fixním rozlišení 640x480 pixelů. Ale toto řešení se ukázalo jako neproveditelné.

Jelikož se pixely musejí kontinuálně posílat na displej (každý pixel se musí na displej poslat ±60krát za vteřinu), je nutné mít pixely uložené ve videopaměti. Velikost této videopaměti je přímo diktována rozlišením displeje. Jelikož výstup na displej je monochromatický, pixely se zakódují do videopaměti jako logická 0 pokud nemají svítit, a logická 1 pokud mají svítit – každý pixel tedy zabírá jeden bit.

Tím pádem se velikost videopaměti rovná:

$$Velikost\ videopaměti = horizontální\ rozlišení * vertikální\ rozl. * počet\ bitů\ na\ pixel$$

$$Velikost\ videopaměti = 640 * 480 * 1 = \ 307\ 200b = 300Kb$$

Pro displej o rozlišení 640x480 pixelů by byla potřeba videopaměť o velikosti 300 kb. FPGA iCE40 obsahuje celkem 128 kb blokových pamětí – tj. přinejlepším třetinu potřebné velikosti, a to nezohledňujeme, že tuto blokovou paměť budou využívat např. i paměti cache.

Řešení bylo na první pohled jasné – použít externí EEPROM paměť. Jejich kapacita se pohybuje až v řádu Megabitů, takže by nebyl problém využít je jako videopaměť.

Ve skutečnosti ale použití externí videopaměti přináší jiný problém – propustnost. Standard VGA pro rozlišení 640x480 stanovuje Pixel clock 25,175 MHz [10], to znamená, že každou sekundu se na displej pošle 25,175 milionů pixelů (cca 73 % z toho jsou reálné pixely, zbylých 27 % je tzv. blanking, kdy se neposílají data na displej).

Vzhledem k tomu, že každý pixel zabírá jeden bit, potřebná propustnost pro zaslání 25,175 milionů pixelů se rovná 25,175 Mb/s. To není až tak nereálný požadavek, ale k tomuto údaji je potřeba připočítat určitý overhead způsobený komunikačními protokoly (je nutné posílat adresy a instrukce) – reálně by byla potřebná propustnost spíše 40-50 Mb/s.

Další komplikaci představuje fakt, že zatímco VGA obvod čte data z videopaměti, tak procesor musí do videopaměti zároveň zapisovat, aby aktualizoval data na displeji – a jelikož se data z EEPROM většinou čtou i zapisují na stejné sběrnici, procesor by se s VGA obvodem musel „prát“ o každý bit propustnosti – byla by tedy potřeba ještě vyšší propustnost.

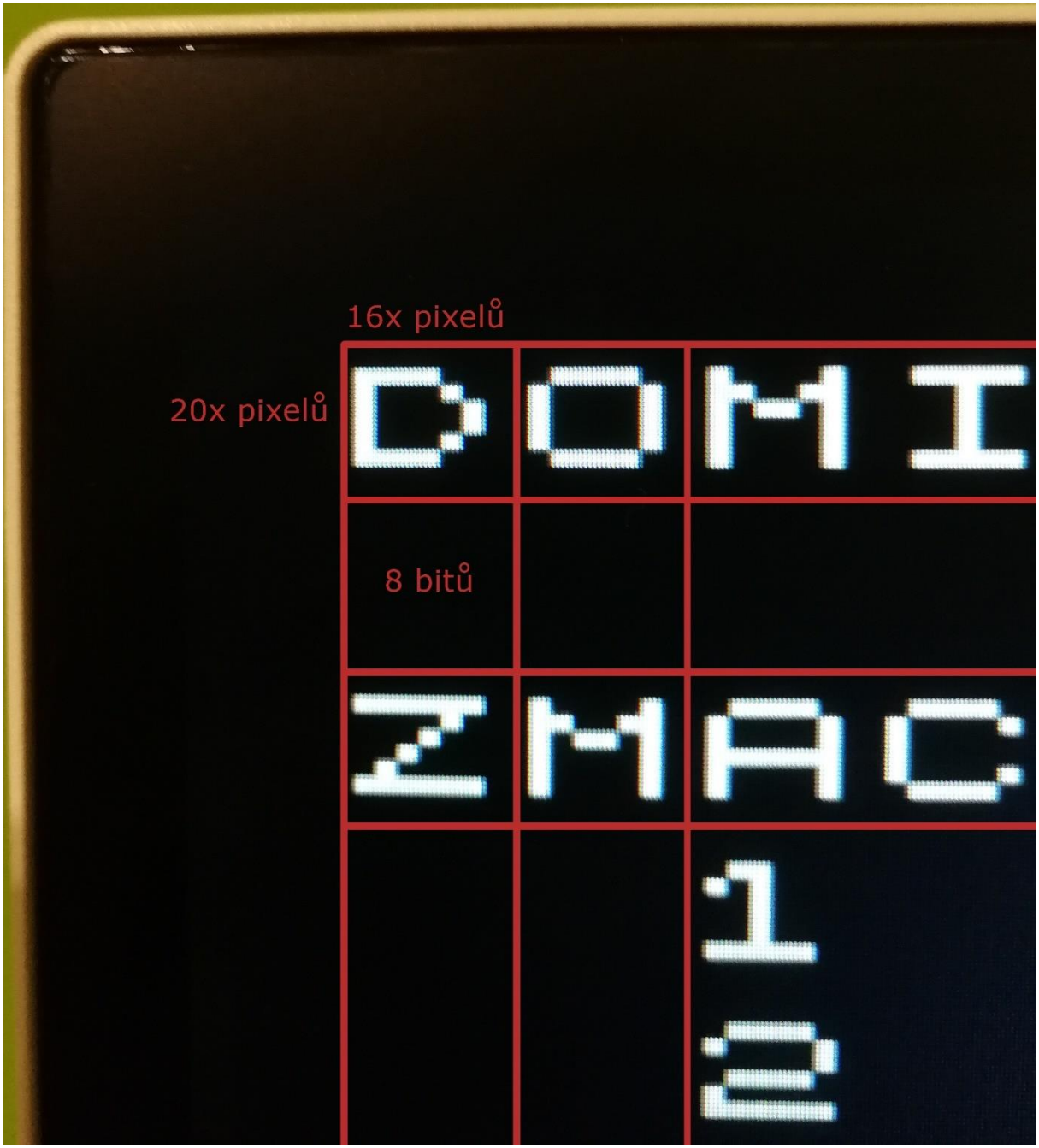
Tento problém by byl o to horší, že VGA obvod pracuje na rozdílné frekvenci než jádro procesoru (viz kapitola *Hodinové domény v procesoru*) – musel by se tedy velmi komplikovaně řešit přechod dat mezi jednotlivými frekvenčními doménami.

Bylo mi jasné, že toto řešení zdaleka není ideální, a že by mohlo způsobit velké množství potenciálních problémů (bylo by velmi nepříjemné navrhnout takto celý displejový procesor a řadič pro EEPROM paměť, a následně zjistit, že toto řešení je nepoužitelné). Proto jsem se snažil najít jiné řešení tohoto problému. Nabízela se možnost razantně snížit rozlišení, ale k tomu jsem nechtěl přistoupit, jelikož by to představovalo velké omezení funkčnosti.

Klíčem k řešení tohoto problému je podstata dat zobrazovaných na displeji – v drtivé většině případů se bude jednat o písmena abecedy, číslice a různé speciální znaky (např. +, -, !). Není tedy nutné do videopaměti ukládat jednotlivé pixely, stačí do ní uložit zakódované znaky, které pak HW na FPGA přeloží na pixely.

Na obrázku č. 18 lze vidět, jak toto řešení vypadá – displej je rozdělen na bloky po 16x20 pixelech. Každý znak zabírá ve videopaměti 8 bitů (znaky jsou zakódovány podle ASCII tabulky) místo 16 * 20 = 320 bitů, kdyby se znak uložil pixel po pixelu – tj. nároky na paměť se snížily 40x.

Obrazový procesor tedy umí na displej vytisknout pouze předdefinované znaky (abeceda, číslice, +, -, …), a k tomu obsahuje malou paměť, ve které si každý program může za běhu nadefinovat až 12 vlastních znaků tisknutelných na displej (toto je použito např. pro tisknutí tanků, viz. kapitola *Demo program*).



Obrázek č. 18 - Ukázka výstupu na displej

13. PAMĚŤOVÝ SUBSYSTÉM

Pro správnou funkci procesoru je nutné ho doplnit o správně navrhnutý paměťový subsystém, který nebude jádro procesoru nijak omezovat. Cílem bylo co nejvíce snížit složitost paměťového subsystému – to byl hlavní důvod, proč procesor funguje čistě na fyzických adresách, a ne na virtuálních (a neobsahuje tedy stránkování).

Jelikož můj procesor je 32bitový, má adresový prostor pro až 4GB paměti (2^{32} Bytů). Prakticky je tak velké množství paměti zbytečné a procesor ho nikdy nevyužije, a tak jsem se rozhodl omezit počet adresních bitů na 20 – procesor má tedy dostupných 2^{20} Bytů paměti, čili 1 MB.

Tato hodnota byla zvolena z důvodu, že TinyFPGA BX je vybaveno právě 1MB flash pamětí, z které je přibližně 0,3 MB využito pro bitstream FPGA a zbylých 0,7 MB je volných pro uživatelský program. 20 adresních bitů je tedy dostatečných pro naadresování celé flash paměti.

13.1. Paměťová mapa

Paměťová mapa rozděluje paměť procesoru do různých oblastí. Její účel je informovat programátora o tom, které adresy může používat pro ukládání dat, které adresy se používají pro I/O a které adresy jsou použity pro různé interní sběrnice.

ROM – adresy 0x0000 0000 až 0x000A EFFF

RAM – adresy 0x000A F000 až 0x000A FFFF

Videopaměť – adresy 0xF000 0000 až 0xF000 0FDC

LUT paměť obrazového procesoru – adresy 0x7F80 0000 až 0x7F80 00FF

Počitadlo milisekund – adresa 0xFFFF FFF0

Klávesnice – adresa 0xFFFF FFFF

ROM představuje oblast paměti s uživatelským programem, tato část paměti je pouze ke čtení, nelze do ní zapisovat. Velikost je přibližně 0,7 MB. RAM je oblast v paměti o velikosti 4 kB, z které procesor může číst a zároveň do ní může ukládat proměnné.

Videopaměť má 1500B a procesor do ní ukládá znaky, které se mají tisknout na displej. Z této paměti nelze číst. LUT je paměť pro uživatelsky definované znaky, má 512B a vleze se do ní až 12 znaků.

Klávesnice je buffer o velikosti jednoho Bytu, z kterého procesor může přečíst poslední zmáčknutou klávesu. Podobně může procesor přečíst aktuální počet milisekund od startu procesoru, když přečte Počitadlo milisekund. Zbytek adres je nedefinovaný a programátor k nim nemá přístup.

13.2. Paměť Flash a SPI řadič

Paměť flash na TinyFPGA BX je Adesto AT25SF081[23]. Její kapacita je 1 MB a paměť komunikuje přes sběrnici SPI. Do paměti flash nelze zapisovat, lze z ní pouze číst uživatelský program.

Paměťový řadič běží na frekvenci 16 MHz, stejně jako jádro procesoru (a tak není potřeba řešit přechod mezi frekvenčními doménami). Každý cyklus paměťový řadič zašle nebo přijme jeden bit z flash paměti – hrubá propustnost je tedy 16 Megabitů za sekund (efektivní je kvůli komunikační režii mnohem nižší).

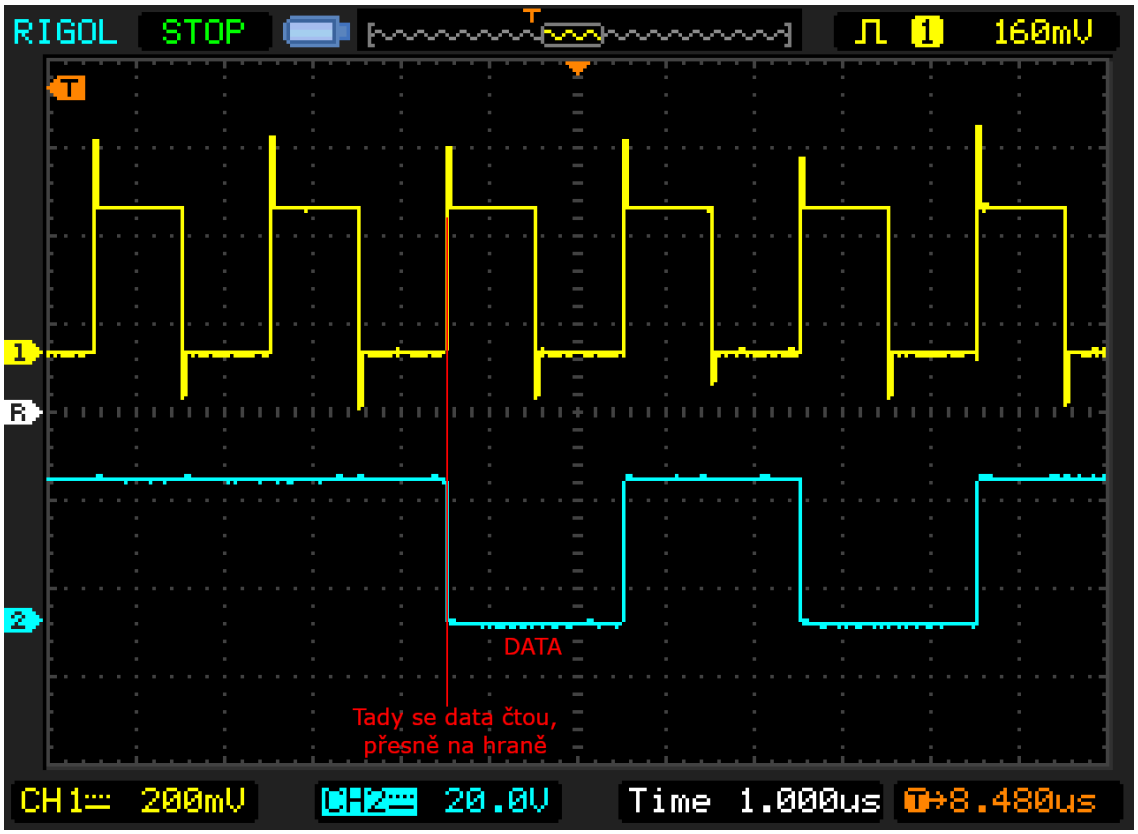
Paměťový řadič je minimalistický a podporuje pouze čtení z paměti, a to v tom nejjednodušším režimu, při kterém je propustnost z/do paměti flash 1 bit za cyklus – v budoucnu bych chtěl přidat podporu pro pokročilejší režimy, které umožňují zvýšit propustnost až na 4 bity za cyklus a zároveň snížit komunikační režii/overhead (jelikož rychlost čtení z paměti flash není v současné době nijak limitující, dostalo toto poměrně jednoduché vylepšení nízkou prioritu).

Při startu procesoru vyšle paměťový řadič do paměti flash instrukci „AB“, která probudí paměť flash ze spánku. Následně pokud paměťový řadič dostane zprávu z instrukční nebo datové cache o tom, že došlo k tzv. cache missu (hledaná data se nenachází v paměti cache), pošle do flash paměti instrukci „03“ následovanou 24bitovou adresou. Paměť flash postupně bit po bitu pošle 32 bitů dat uložených na dané adrese. Paměťový řadič poskládá všech 32 bitů dohromady a vyšle je do instrukční/datové cache.

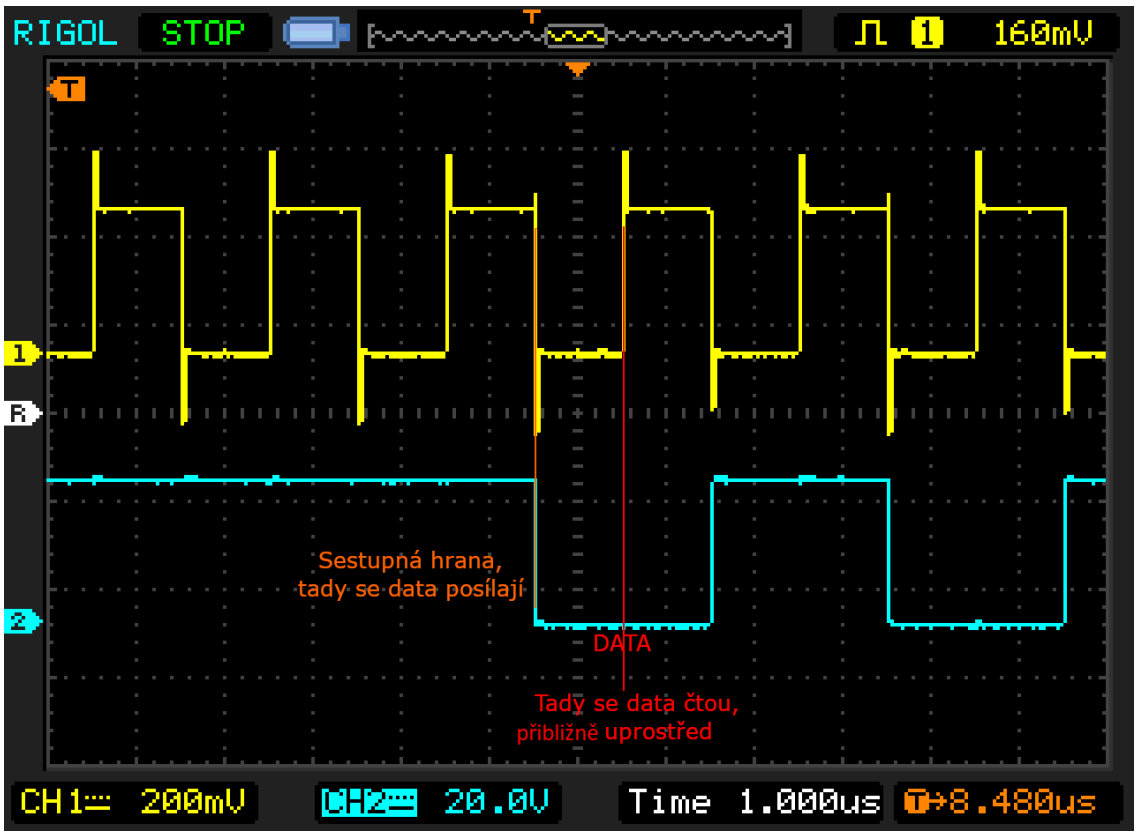
Paměťový řadič při každém čtení z paměti tedy nejdříve zašle 32 bitů (8bitová instrukce + 24bitová adresa) a následně přijme 32 bitů dat – celkem tedy 64 bitů. Při rychlosti přenosu 1 bit za cyklus dostaneme tedy latenci 64 cyklů.

Přesněji popsanou komunikaci lze najít v datasheetu paměti flash. Efektivní propustnost (včetně konkrétních hodnot), latence a možná vylepšení paměťového řadiče jsou více rozepsány v kapitole *Budoucí možná vylepšení procesoru*.

Při zprovozňování SPI řadiče jsem narazil na zajímavý problém, kdy původní implementace zasílala data při vzestupné hraně hodinového signálu (viz obrázek č. 19), a až později jsem si uvědomil, že paměť flash data čte také při vzestupné hraně – data se tedy nemohla přenést do paměti flash včas. Na obrázku č. 20 lze vidět opravenou verzi, kdy se data posílají na sestupné hraně hodinového signálu, a mají tedy spoustu času dorazit do paměti flash.



Obrázek č. 19 – SPI rozhraní na osciloskopu 1, žlutá křivka je hodinový signál, modrá křivka jsou data



Obrázek č. 20 – SPI rozhraní na osciloskopu 2, žlutá křivka je hodinový signál, modrá křivka jsou data

13.3. Datová a Instrukční Cache

Datová i instrukční Cache mají přibližně stejnou strukturu, liší se jen v detailech (do instrukční cache nelze zapisovat a lze z ní číst pouze po 32 bitech, zatímco do datové cache lze zapisovat a lze z ní číst po 8 bitech).

Úkolem pamětí cache je zásobovat procesor daty – aby nemusel čekat na načítání dat z pomalé paměti flash (respektive paměti RAM u jiných procesorů).

Obě paměti cache mají velikost 4 kB, další 2 kB zabírají tagy. Takže ačkoliv instrukční i datová cache mají efektivní kapacitu 4 kB, fyzicky zabírají 6 kB.

Tagy mají velikost 16 bitů a představují doprovodná data potřebná pro každý blok dat (tzv. Cache blok). Tag je tvořen 10bitovou adresou, dvěma LRU bity, jedním valid bitem a jedním dirty bitem – celkem tedy 14 bitů (jelikož blokové paměti nepodporují datovou šířku 14 bitů, jsou do tagu přidány dva prázdné bity, aby tagy měly 16 bitů -> shodná velikost jako je datová šířka blokových pamětí).

Latence obou pamětí cache je 1 cyklus, což znamená, že když v cyklu 1 požádáme o data, paměť cache nám je zašle v cyklu 2. Latence při zápisu jsou 2 cykly.

Obecně se dá říct, že paměti cache pro svou práci využívají dvou principů – temporální lokality a prostorové lokality (temporal and spatial locality) [22].

Prostorová lokalita znamená, že pokud procesor čte data např. z adresy 4, je velmi pravděpodobné, že v budoucnu bude číst data z adresy 8 (jelikož se tato adresa nachází hned vedle adresy 4). Z tohoto důvodu se data do cache nenačítají po jednotlivých Bytech, ale po tzv. Cache blocích – kdykoliv chce procesor načíst jakákoliv data z paměti flash (resp. paměti RAM), tak procesor nenačte pouze 32 bitů nebo 8 bitů, načte rovnou celý Cache blok.

Velikost Cache bloků (někdy se také říká cache line size) se pro každý procesor liší, ale moderní procesory (Intel, AMD a ARM) se s velikostí cache bloků ustálily na 64 Bytech, tedy 512 bitech.

Velikost cache bloků představuje trade off – ne vždy platí, že vyšší velikost cache bloků je lepší. Za určitou hranicí se zvyšování velikosti cache bloků přestane vyplácet, a naopak zvýšení velikosti vede ke snížení výkonu procesoru. Vyšší velikost cache bloků benefituje hlavně práci s poli (kde jsou jednotlivé prvky pole v paměti umístěny hned za sebou. Při načtení jednoho prvku z pole se tedy načtou i okolní prvky pole).

Já jsem si pro svůj procesor zvolil velikost cache bloků 4 Byty, tedy 32 bitů, a to z důvodu, že se jedná o nejmenší logickou velikost (jelikož instrukce mají šířku právě 32 bitů, je vhodné, aby cache bloky měly minimálně tuto velikost) a zároveň se tato velikost cache bloků nejsnadněji implementuje z blokových pamětí.

Velikost cache bloků implikuje i to, že se data z flash paměti do paměti cache načítají po 4Bytových blocích.

V budoucnu bych chtěl vytvořit verzi pamětí cache s velikostí cache bloků 8 Bytů (64 bitů), a otestovat, zda dojde ke zvýšení či ke snížení výkonu procesoru. Toto je více rozebráno v kapitole *Budoucí možná vylepšení procesoru*.

Temporální lokalita znamená, že pokud procesor nedávno načítal data např. z adresy 12, je velmi pravděpodobné, že v brzké době bude tato data využívat znovu. Z tohoto důvodu se do pamětí cache zavádí asociativita (n-cestně asociativní cache, n-way set associative) – to znamená, že každá adresa (každá data) se mohou uložit na *n* cest (na *n* setů) uvnitř paměti cache.

Paměť cache následně vyhodnocuje, která z *n* cest obsahuje nejstarší a nejméně využívaná data, a nová data zapíše právě na tento nejméně využívaný set (takže často využívaná data zůstávají v paměti cache, zatímco málo využívaná data jsou nahrazena novými daty).

Moje instrukční a datová cache se tedy obě skládají ze čtyř setů adres, kam se může každá proměnná/instrukce uložit. Konkrétní set se vždy vybere pomocí principu LRU (Least Recently Used) – každý tag má v sobě 2 LRU bity, které indikují, kdy naposledy byla tato adresa využita. Když paměť cache potřebuje uložit nová data, uloží je na set, který je nejméně užívaný (má nejvyšší hodnotu LRU).

Každý set paměti cache je tvořen z blokových pamětí, které na FPGA iCE40 LP8K mají velikost 4 kb – pro paměti cache jsem zvolil organizaci blokové paměti 256 záznamů po 16 bitech (vyšší datovou šířku bohužel blokové paměti nepodporují). Abych tedy dosáhl propusti 32 bitů/cyklus (což je potřeba, aby bylo možné každý cyklus načíst jednu 32bitovou instrukci z instrukční cache, respektive jedno 32bitové číslo z datové cache), musel jsem využít dvě blokové paměti zapojené paralelně (jedna dodává spodních 16 bitů dat, druhá dodává vrchních 16 bitů dat).

Každý set tedy obsahuje dvě blokové paměti, do kterých se ukládají data, a k tomu ještě jednu blokovou paměť pro ukládání tagů – celkem tedy tři blokové paměti na každý set. Vzhledem k tomu, že obě cache jsou složeny ze 4 setů, tak instrukční i datová cache využívají každá 12 blokových pamětí (celkem tedy využívají 24 z dostupných 32 blokových pamětí).

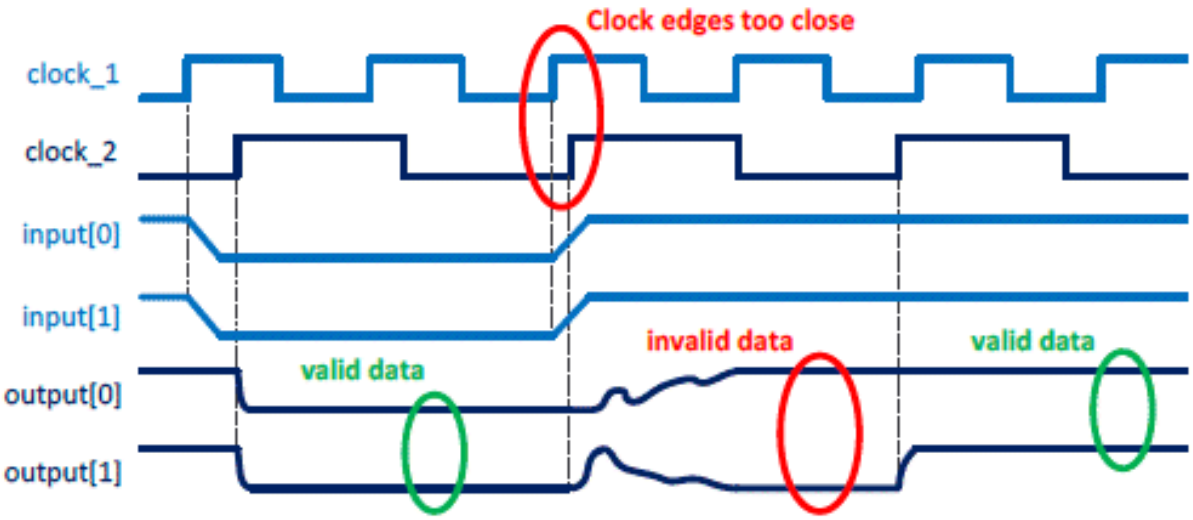
14. HODINOVÉ DOMÉNY V PROCESORU

V procesoru se nacházejí tři hodinové (frekvenční) domény (clock domain):

1. 16MHz doména – zde spadá celé jádro procesoru a paměťový subsystém
2. 40MHz doména – zde spadá obrazový procesor
3. ~16KHz hodinový signál z klávesnice

Paměti flip flop mají výrobcem deklarované tzv. setup a hold times (nastavovací a držící časy), které deklarují, jak dlouho (řádově kolik nanosekund/pikosekund) musí být data na vstupu paměti flip flop stabilní před (setup time) a po (hold time) vzestupné hraně hodinového signálu. Pokud dojde k nedodržení těchto časů, začne se digitální obvod chovat nevyzpytatelně a začnou vznikat chyby.

Na některých místech v procesoru dochází k přechodu dat mezi výše zmíněnými hodinovými doménami (clock domain crossing). Taková místa jsou velmi kritická, jelikož při nesprávném návrhu přechodu mezi frekvenčními doménami se data nestihnou ustálit a může dojít právě k nedodržení setup/ hold časů pamětí flip flop a dojde tedy k chybám a nestabilitě. Taková situace je ilustrována na obrázku č. 21.



Obrázek č. 21 – Ukázka nestability frekvenčního přechodu. Vzestupné hrany hodinových signálů clock_1 a clock_2 se nachází moc blízko a dojde k chybnému přenosu dat. Zdroj: https://www.design-reuse.com/articles/29080/clock-domain-crossing-verification-in-do-254.html

Přechod z 16KHz domény klávesnice do 16MHz domény jádra procesoru (při načítání dat z klávesnice) nepředstavoval problém, jelikož cílová frekvenci je mnohem vyšší než zdrojová.

Přechod z 16MHz domény jádra procesoru do 40MHz domény obrazového procesoru (při zápisu a čtení z/do videopaměti) naštěstí prochází skrz blokové paměti – blokové paměti na FPGA iCE40 LP8K mají separátní čtecí a zapisovací hodinový signál. Do videopaměti je

tedy zapisováno ve frekvenční doméně 16MHz, a čteno je z ní ve frekvenční doméně 40MHz. Blokové paměti tvoří ideální přechod mezi hodinovými doménami.

Další přechod mezi 16MHz a 40MHz doménou se nachází v debugovacím rozhraní (které „tiskne“ signály z 16MHz domény na monitor, viz kapitola *Debugovací logika*). Tento přechod je o dost nepříjemnější, jelikož neprochází skrz blokové paměti a obě frekvence mají řádově porovnatelnou velikost.

Původně jsem tento přechod řešil jen skrz synchronizaci pomocí pamětí flip flop. Data prošly skrz dvě paměti flip flop před frekvenčním přechodem a skrz další dvě paměti flip flop po frekvenčním přechodu (aby se zajistila synchronizace dat), jenže toto řešení frekvenčního přechodu se ukázalo jako nedostatečné – při zapnutí debugovacího rozhraní bylo na monitoru jasně vidět šumění.

Řešení jsem našel v práci Clifforda E. Cummingse *Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog* [24]. Data z 16MHz domény jsem před posláním nejdříve zpomalil na frekvenci ~2MHz, a až následně po zpomalení prošly stejnou sadou pamětí flip flop. Tato malá změna stačila k tomu, aby ve frekvenčním přechodu nedocházelo k chybám (respektive počet chyb se snížil na minimum) a na monitoru nebyl vidět žádný šum.

15. KOMPILÁTOR (A LOADER)

Ačkoliv tématem práce je návrh procesoru, samotný procesor bez programu, který by na něm šel spustit, je k ničemu. Pro napsání programu existuje hned několik možností:

1. Napsat program v assembly (ve strojovém kódu, instrukci po instrukci). Tato možnost je vhodná pro jednoduché programy, ale je dost zdlouhavá a nevhodná pro napsání větších programů.
   Takto jsem napsal několik prvních programů, které měly otestovat základní funkčnost procesoru, když ještě byl plný chyb.

2. Napsat program v programovacím jazyce (např. jazyk C), a následně použít kompilátor, který za nás přeloží programovací jazyk do strojového kódu.
   Tuto možnost jsem si vybral pro napsání finálního programu, jelikož umožňuje jednoduché (a uživatelsky přívětivé) naprogramování procesoru.

Jelikož k naprogramování procesoru bude potřeba dostat se blízko k HW, zvolil jsem pro naprogramování procesoru nízko úrovňový programovací jazyk C, který je pro toto nejvhodnější.

Jedna z výhod použití otevřené a populární instrukční sady RISC-V je, že pro ni je volně dostupné velké množství nástrojů – já použil kompilátor GCC [11].

Bohužel se mi nepodařilo postavit si sám nejnovější verzi GCC pro RISC-V z oficiálního GitHubu, a tak jsem použil předem postavenou verzi [12].

15.1. Nastavení kompilátoru

Základní nastavení kompilátoru je jednoduché – stačí specifikovat použitou variantu instrukční sady *rv32i*, a tzv. ABI *ilp32* (ABI – Application Binary Interface, specifikuje, že datové typy Int, Long a Pointery mají 32 bitů. Druhou možností je *ilp64* pro 64 bitové systémy a procesory RISC-V).

Takto nastavený kompilátor je sice schopný kompilovat kód pro můj procesor, ale neví nic např. o struktuře paměti procesoru.

Vzhledem k tomu, že jsem neměl moc velkou představu o tom, co vše je nutné v kompilátoru nastavit, byl jsem moc rád, když jsem narazil na tento článek [13], podle něhož jsem byl schopný upravit linkovací skript (definoval jsem v něm jednotlivé paměťové oblasti, viz kapitola *Paměťový subsystém*, velikost a umístění stacku – stack je oblast paměti, kde si programy ukládají proměnné) a zároveň jsem podle tohoto článku vytvořil vlastní „startovací“ sekci programu „crt0.s“. Zároveň jsem od autora přejal několik vlajek pro kompilátor, které jsem doplnil o několik vlastních vlajek, abych zaručil že se kód zkompiluje správně a optimálně.

Konkrétně se jedná o tyto vlajky:

1. `-Wall` (můj dodatek) – říká kompilátoru, aby hlásil veškerá (tj. i nepodstatná) varování, která by jinak „zamlčel“.

2. `-O3` (můj dodatek) – jedná se o optimalizační vlajku, která udává, jak moc má kompilátor program optimalizovat. Nejnižší stupeň optimalizace je -O0 (defaultní nastavení, žádné optimalizace), nejvyšší stupeň optimalizace je -O3.

3. `-flto` (můj dodatek) – link time optimization. Standardně kompilátor provádí optimalizace pouze při kompilaci programu, vlajka -flto umožňuje program optimalizovat i při linkování programu („spojování dohromady“).

4. `-mstrict-align` (můj dodatek) – zakazuje kompilátoru použít nezarovnané zápisy/čtení paměti (jejich podpora je v RISC-V dobrovolná, já se rozhodl je nepodporovat v mém procesoru).

5. `-ffreestanding` – informuje kompilátor, že standardní knihovna není na cílovém systému přítomna. Standardní knihovnu běžně poskytuje operační systém, ale jelikož na mém procesoru žádný operační systém neběží, není standardní knihovna přítomna, a je potřeba o tomto informovat kompilátor.

6. `-Wl--gc-sections` – instruuje linker, aby odstranil nepoužívané sekce programu.

7. `-nostartfiles` – instruuje linker ať nepoužívá žádné standardní „startovací“ soubory, jelikož my si definujeme vlastní „startovací“ soubor „crt0.s“.

8. `-Wl,-T,riscv32-virt.ld` – specifikuje linkovací skript, který jsme si vytvořili.

Kompletní příkaz pro kompilaci tedy vypadá takto:

```
riscv32-unknown-elf-gcc -Wall -mstrict-align -O3 -flto -
ffreestanding -Wl,--gc-sections -nostartfiles -Wl,-T,riscv32-
virt.ld crt0.s main_define.c -o a3.bin
```

Na obrázku č. 22 je zobrazen výstup kompilátoru – nahoře je výpis všech sekcí přítomných v programu, a pod tímto výpisem už jsou jednotlivé instrukce programu (obrázek ukazuje prvních asi 20 instrukcí z 1600).

15.2. Loader

Momentálně má kompilace programu jeden háček – nelze využívat globální proměnné (a také např. C funkci rand()). V případě použití globálních proměnných kompilátor totiž vytvoří programovou sekci „.sdata“ – tato sekce je specifická v tom, že kompilátor

předpokládá, že na cílovém procesoru běží operační systém, který obsahuje loader. Úkolem loaderu je data ze sekce „.sdata“ vzít, a načíst je do paměti na kompilátorem specifikovanou adresu.

Ale jelikož můj procesor nemá žádný operační systém, neobsahuje ani žádný loader – sekce „.sdata“ se tedy nikdy do paměti nenačte a program se nevykoná správně. Pokoušel jsem se toto obejít pomocí vlajky pro kompilátor „-static-pie“, která by měla linker instruovat, aby vytvořil statický, pozičně nezávislý program (**P**osition **I**ndependent **E**xecutable – **pie**), bohužel tato vlajka vrátila chybu „-pie not supported“, kterou se mi nepodařilo nijak obejít.

Proto bych v budoucnu chtěl buď vytvořit vlastní loader, nebo zprovoznit kompilaci programu s vlajkou „-static-pie“ a odstranit tedy tento nedostatek – použití globálních proměnných by značně zjednodušilo funkce pro tisknutí na displej.

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .init         0000001c  00000000  00000000  00001000  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .text         00001a00  0000001c  0000001c  0000101c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .rodata       00000054  00001a1c  00001a1c  00002a1c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .eh_frame     0000002c  00001a70  00001a70  00002a70  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      00000012  00000000  00000000  00002a9c  2**0
                  CONTENTS, READONLY
  5 .riscv.attributes 0000001c  00000000  00000000  00002aae  2**0
                  CONTENTS, READONLY
Disassembly of section .init:

00000000 <_start>:
   0:   00003197                auipc   gp,0x3
   4:   29c18193                addi    gp,gp,668 # 329c <__global_pointer$>
   8:   000b0117                auipc   sp,0xb0
   c:   ff810113                addi    sp,sp,-8 # b0000 <__stack_top>
  10:   00010433                add     s0,sp,zero
  14:   008000ef                jal     ra,1c <main>
  18:   00000073                ecall

Disassembly of section .text:

0000001c <main>:
      1c:       f6010113                addi    sp,sp,-160
      20:       00200793                li      a5,2
      24:       02f12a23                sw      a5,52(sp)
      28:       000105b7                lui     a1,0x10
      2c:       00400793                li      a5,4
      30:       08112e23                sw      ra,156(sp)
      34:       08812c23                sw      s0,152(sp)
      38:       08912a23                sw      s1,148(sp)
      3c:       09212823                sw      s2,144(sp)
      40:       09312623                sw      s3,140(sp)
      44:       09412423                sw      s4,136(sp)
      48:       09512223                sw      s5,132(sp)
      4c:       09612023                sw      s6,128(sp)
      50:       07712e23                sw      s7,124(sp)
      54:       07812c23                sw      s8,120(sp)
      58:       07912a23                sw      s9,116(sp)
      5c:       07a12823                sw      s10,112(sp)
      60:       07b12623                sw      s11,108(sp)
      64:       02f12823                sw      a5,48(sp)
      68:       040105a3                sb      zero,75(sp)
      6c:       00000613                li      a2,0
      70:       00f00813                li      a6,15
      74:       01300893                li      a7,19
      78:       fcb58593                addi    a1,a1,-53 # ffcb <__global_pointer$+0xcd2f>
```

Obrázek č. 22 - Výstup kompilátoru

16. STANDARDNÍ KNIHOVNY

Jak už bylo řečeno v předchozí kapitole, jelikož procesor nebude mít žádný operační systém, nelze se spoléhat na standardní systémové knihovny, je nutné je všechny navrhnout od píky – jedná se zejména o funkce printf() a scanf( ), které slouží pro tisknutí dat na displej, respektive jejich načítání z klávesnice.

Vytvořil jsem si tedy vlastní knihovnu „std_fce.c“, která obsahuje funkce:

1. „scan_char“ – načte zmáčknutou klávesu z klávesového bufferu
2. „print_char“ – vytiskne jeden znak na monitor
3. „print_text“ – vytiskne string (řetězec) textu na monitor
4. „print_int“ – vytiskne na monitor číslo typu int
5. „clear_display“ – vymaže z displeje všechny znaky
6. „pwr“ – jednoduchá funkce, která počítá zadanou mocninu zadaného čísla
7. „millis“ – vrátí počet milisekund od startu procesoru, podobně jako funkce millis na Arduinu
8. „delay“ – čeká zadaný počet milisekund

Knihovna tedy poskytuje všechny funkce potřebné pro vytvoření i poměrně složitých programů.

V knihovně „std_math.c“ lze najít funkce pro výpočet absolutní hodnoty a vybraných goniometrických funkcí – sin128x, cos128x, cos128x^2, tan128x.

Jelikož procesor neumí operace s desetinnými čísly float, podporuje pouze celočíselné hodnoty int, a jelikož funkce sinus a cosinus ze své podstaty vrací desetinné číslo z intervalu <0; 1>, bylo potřeba tyto funkce implementovat speciálním způsobem.

Funkce ve skutečnosti vrací zaokrouhlený 128 násobek dané hodnoty sin/cos. S touto vynásobenou hodnotou se následně udělají všechny potřebné výpočty, a jako finální krok se výsledek vydělí 128. Programátor musí dbát zvýšené opatrnosti, aby tyto funkce použil správně – jejich použití lze vidět v aplikaci tanky, kde se pomocí goniometrických funkcí počítá trajektorie výstřelu s uspokojivou přesností.

V budoucnu bych knihovnu chtěl doplnit o funkce pro generování náhodných čísel (nebo zprovoznit standardní C funkci rand(), která momentálně nefunguje kvůli nepřítomnosti loaderu).

Knihovny si lze prohlédnout v příloze č. 2.

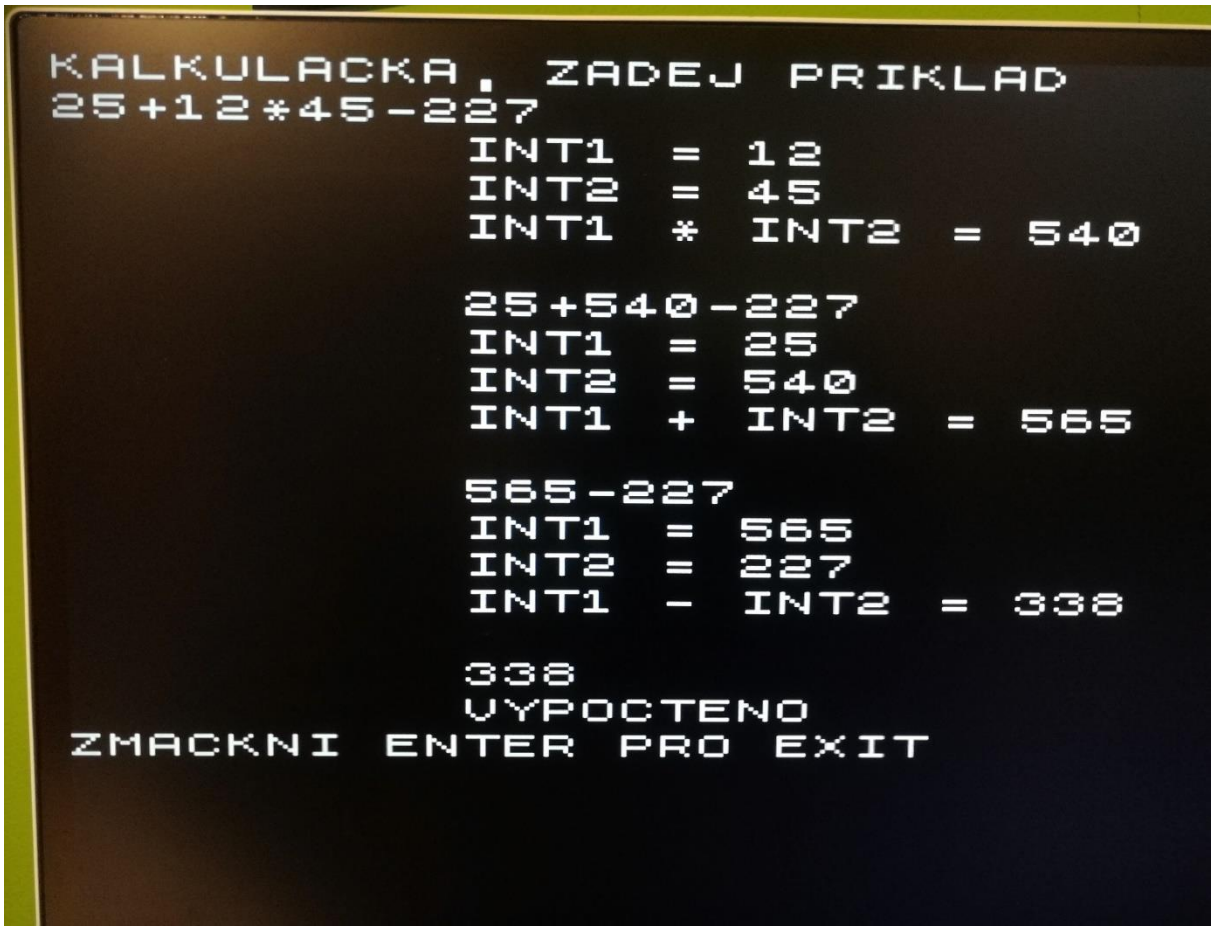
17. DEMO PROGRAM

Pro ukázku funkce procesoru jsem si vybral program jednoduché kalkulačky a hru tanky.

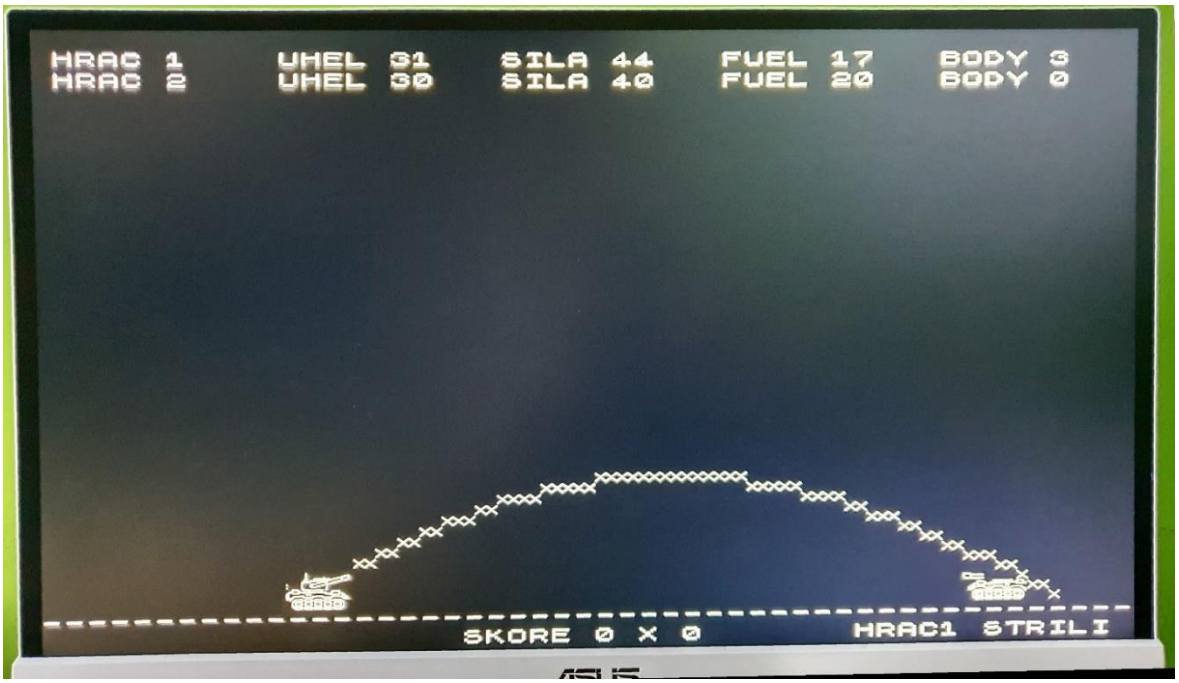
Kalkulačka je schopná jako vstup vzít libovolný počet čísel, jejich hodnota může být od -1000000 až do +1000000. Kalkulačka umí čísla sčítat, odečítat a násobit – s tím, že respektuje přednost násobení před sčítáním a odčítáním. Kalkulačka umí pracovat i se zápornými čísly (ačkoliv tato funkce není plně otestována).

Jak jde vidět z obrázku č. 23, program pracuje iterativně. Zadaný příklad je „25+12*45-227“. Program nejdříve vynásobí čísla 12 a 45, a výsledek zapíše zpět do příkladu – příklad je nyní „25+540-227“. Následně program sečte čísla 25 a 540 – výsledek znovu zapíše do příkladu, který je nyní „565–227“. V poslední iteraci procesor vypočítá výsledek (číslo 338) a vytiskne ho na displej.

Program bych v budoucnu chtěl vylepšit o podporu desetinných čísel nebo např. dělení. Zároveň bych v budoucnu chtěl vytvořit o dost složitější program který předvede funkci procesoru.



Obrázek č. 23 - Ukázka aplikace Kalkulačka na procesoru

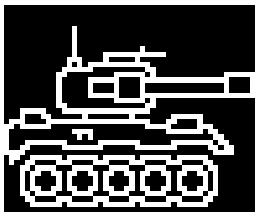


Obrázek č. 24. – Ukázka aplikace tanky

Minihra tanky je o poznání pokročilejší. Dva hráči bojují proti sobě, nastavují vždy rychlost a sílu výstřelu, a procesor na monitoru vykreslí trajektorii střely. Trajektorie se počítá dle následujícího vzorce[8], kde α je úhel výstřelu, v je síla výstřelu, g je tíhové zrychlení ($\sim 10\ m * s^{-2}$) a x je souřadnice na horizontální ose:

$$y = \frac{tan128(\alpha) * x - \frac{3 * g * x^2 * 128 * 128}{2 * v^2 * cos128^2(\alpha)}}{128}$$

Počáteční pozice tanků je náhodná. Tanky se mohou pohybovat pomocí šipek vlevo a vpravo, než jim dojde omezený benzín. Vyhrává hráč, který jako první eliminuje nepřítelův tank.

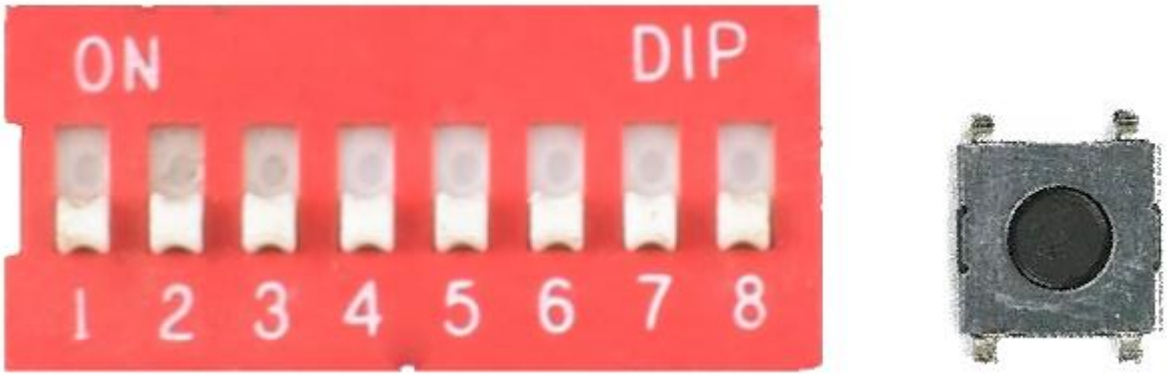


Obrázek č. 25. - Tank

[8] Jedná se o vzorec pro výpočet trajektorie šikmého vrhu upravený pro mé potřeby. Jak už bylo zmíněno výše, procesor (jako všechny mikrokontrolery a jednoduché procesory) nepodporuje desetinná čísla typu float. Z tohoto důvodu musel být vzorec upraven.
Funkce tan128 a $cos128^2$ představují 128 násobek daných funkcí. Různě po vzorci najdete rozmístěné násobení a dělení právě číslem 128 – to je zde z důvodu dodržení matematické korektnosti. Pokud se ve jmenovateli zlomku nachází 128 násobek cos^2, je nutné čitatel zlomku vynásobit číslem 128 aby byla dodržena ekvivalence. Násobení/dělení je rozmístěno tak, aby bylo dosaženo co nejvyšší přesnosti.
Pro většinu úhlů má tento vzorec na procesoru velmi uspokojivou přesnost, odchylka oproti verzi s float čísly a standardními goniometrickými vzorci spuštěné na standardním stolním počítači je pod 1 % (tj. naprosto zanedbatelná a na obrazovce neviditelná). Jedinou výjimkou jsou úhly 1° a 2°, u kterých odchylka u funkce cos^2 dosahuje asi 10 %

18. DEBUGOVACÍ LOGIKA

Abych si usnadnil proces ladění procesoru, tak jsem si navrhl vlastní poměrně rozsáhlou debugovací logiku, která umožňuje procesor krokovat instrukci po instrukci[9] a zobrazovat na monitoru interní proměnné (respektive spíš dráty).



Obrázek č. 26 - DIP switch a mikrospínač

Debugovací logika je ovládána DIP switchem, který je přítomný na pájivém poli vedle FPGA.

Přepínač číslo 1 ovládá debugovací režim – pokud je ve spodní poloze, procesor funguje v normálním režimu s frekvencí 16 MHz. Pokud ho ale přepneme do vrchní pozice, frekvence sice stále zůstane na 16 MHz, ale procesor bude s vykonáním každé instrukce čekat do zmáčknutí tlačítka – a je tedy možné vidět chod programu instrukci po instrukci.

Ostatní přepínače slouží pro nastavení, co za interní signály („dráty“) se budou tisknout na monitor, viz tabulka č. 2.

[9] Alternativně procesor podporuje možnost krokování po jednotlivých cyklech, ale ta není moc praktická, jelikož např. pro načtení instrukce z paměti flash je potřeba vyčkat 64 cyklů – tj. 64 zmáčknutí tlačítka, a tak je debugování po jednotlivých cyklech neprakticky zdlouhavé.

DIP_switch[6:0]	Kde	Signál
7'b0000000	Nic	Nic
7'b1000000	CORE	PC
7'b1000001	CORE	nextPC
7'b1000010	CORE	instr_fetch_exec
7'b1000011	CORE	stall_pc, fetch_valid_exec, decoder_stall,
		stall_debug, resetn, mem_write_ready, we_reg
7'b1000100	CORE	aluRes
7'b1000101	CORE	memData
7'b1000110	CORE	mem_write_data
7'b1000111	CORE	nextPC
7'b1001000	CORE	instr_fetch_exec
7'b1001001	CORE	Směska
7'b1001010	CORE	aluRes
7'b1001100	CORE	memData
7'b1001101	CORE	nextPC
7'b1001110	CORE	instr_fetch_exec
7'b1001111	CORE	Směska
7'b1010000	CORE	aluRes
7'b0100001	ICACHE	read_addr_old
7'b0100010	ICACHE	RDATA_OUT
7'b0100011	ICACHE	state, cache_miss, fetch_valid, set_used, read_en
7'b0100100	ICACHE	RADDR_CACHE, WADDR_CACHE, tag
7'b0100101	ICACHE	RDATA_setA
7'b0000001	SPI_CONTROLLER	flash_addr
7'b0000010	SPI_CONTROLLER	SPI_DATA
7'b0000011	SPI_CONTROLLER	flash_byte, receiving, startup, SPI_CS, SPI_SCK, SPI_SI, SPI_SO
7'b0000100	SPI_CONTROLLER	icache_miss, dcache_miss, startup, busy, bits_received
7'b1111110	KEYBOARD	keyboard_valid, pressed_key
7'b1111111	KEYBOARD	poslední ctyri scancode

Tabulka č. 2 - Debugovací kódy

19. BUDOUCÍ MOŽNÁ VYLEPŠENÍ PROCESORU

I když procesor je již hotový, stále existuje mnoho způsobů, jak ho vylepšit, jmenovitě jak mu zvýšit výkon.

19.1. Pipelining

Pipelining (česky zřetězení) je technika pro návrh jádra procesoru (a digitálních obvodů obecně). Pipelining spočívá v rozdělení zpracování instrukce do několika kroků.

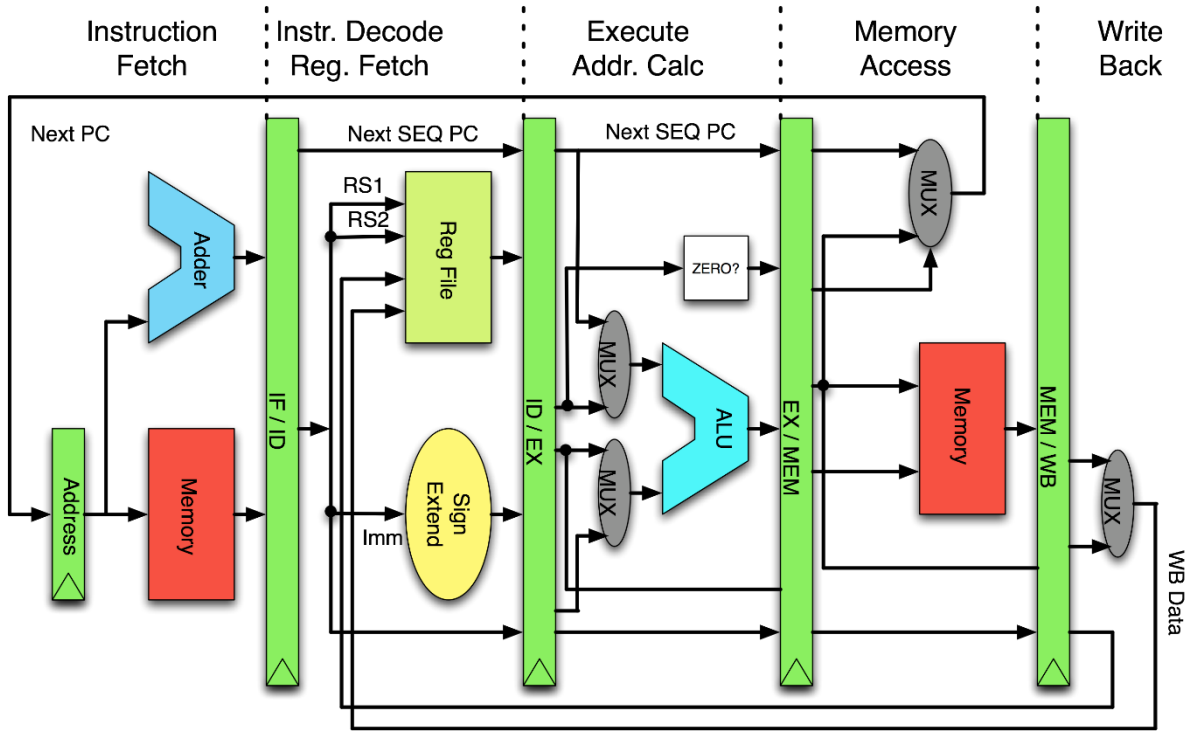
Pipelining lze nejlépe vysvětlit na produkční lince v továrně.

Obrázek č. 27. – Výrobní linka aut, zdroj: https://www.behance.net/gallery/106658417/Car-Factory-illustration

V továrně na auta jednotlivé vozy postupně za sebou projíždí výrobní linkou. Na každém stanovišti se vždy auto zastaví, aby se vykonala určitá činnost – osazení kol, osazení karoserie, nástřik barvy, osazení oken. Po dokončení operace výrobní linka popojede a auto se ocitne na dalším stanovišti. Než auto opustí továrnu, projede postupně celou výrobní linkou od začátku až po konec.

Úplně stejně funguje pipelined procesorové jádro (ukázka na obrázku č. 28).

Instrukce postupně putuje procesorovým jádrem skrz jednotlivé *kroky*. Stejně jako se na auto osadí karoserie nebo okna, tak shodným způsobem se v každém kroku na instrukci provede určitá operace – načtení z paměti, dekódování instrukce, aritmetické operace, zapsání výsledků do paměti.



Obrázek č. 28. – Ukázka pěti stupňového pipelined jádra. Zdroj: https://commons.wikimedia.org/wiki/File:Pipeline_MIPS.png

Jednotlivé kroky v jádře jsou od sebe odděleny tzv. pipeline registry (dlouhé zelené obdélníky na obrázku č. 28). Každý krok se vykonává jeden cyklus. Pokud tedy procesor obsahuje pěti krokovou pipeline, bude se instrukce vykonávat pět cyklů, protože musí projít každým krokem. [10]

To je zásadní rozdíl oproti standardnímu jednocyklovému procesoru, který celou instrukci zpracovává v jediném cyklu.

Využití pipeliningu má několik následků – zpracovává se vždy několik instrukcí najednou. Mezitím, co se výsledek jedné instrukce zapisuje do paměti, druhá instrukce je dekódována a třetí instrukce se načítá z paměti.

[10] Podobně jako pipelining lze na příkladu výrobní linky vysvětlit superskalární jádro – superskalární jádro je defacto několik výrobních linek paralelně vedle sebe. Pracuje se tedy vždy na dvou autech/dvou instrukcích paralelně. Toto umožňuje každý cyklus dokončit dvě auta/instrukce najednou.
Moderní procesory, jako ten v počítači, na kterém čtete tuto práci, nebo ten v telefonu, který leží kousek od vás, jsou všechny bez výjimky superskalární pipelined procesory. Tyto dva principy tvoří pomyslné základní stavební kameny všech dnešních výkonných procesorů.
Někdy se počtu kroků v pipeline referuje jako „hloubka“ pipeline, a počtu paralelních „výrobních linek“ se referuje jako „šířka“ pipeline.

Toto velmi komplikuje načítání instrukcí, jelikož se velmi často stává, že výsledek jedné instrukce závisí na výsledku jiné instrukce. Jinými slovy, pokud procesor vidí za sebou tyto instrukce:

1. A = B + 5
2. X = A + Y

Tak procesor musí zajistit, že instrukce č. 2 použije pro výpočet aktualizovanou hodnotu A. Zajistit něco takového v mnoha krokovém pipelined procesoru je velmi komplikované.

Ještě větší problém představují větve – pokud určitá podmínka neplatí, procesor načte následující instrukci. Pokud ale podmínka platí, procesor skočí a načte instrukci z úplně jiného místa v paměti. Pipelined procesor se dozví až velmi hluboko v pipeline, zda je větev přijata či ne. To znamená, že když pipelined procesor narazí na jakoukoliv větev, tak několik následujících cyklů neví, co za instrukci má načíst – a naše výrobní linka začne být prázdná.

Výkonné procesory mají velmi hluboké pipeline – např. procesory ve stolních počítačích mívají pipeline hluboké 15 až 20 kroků, procesory v telefonech spíše 10 až 15 kroků (rekordmanem je nechvalně známé jádro Intelu Prescott, které mělo pipeline dlouhou 31 kroků. Dnes panuje spíše trend krátkých a velmi širokých pipeline).

Využívají proto velmi složité prediktory větvení, které mají za úkol odhadnout, zda daná větev bude přijata či ne, aby procesor mohl začít spekulativně vykonávat instrukce a načítat je do pipeline předem (prázdná pipeline je noční můra každého návrháře procesorů). Vývojem prediktorů větvení se zabývala má loňská práce SOČ **KVN Prediktor Větvení** [26].

Druhým efektem použití pipeliningu je zkrácení každého kroku – jednocyklový procesor má jeden dlouhý a pomalý krok, a proto běží na nízké frekvenci. Pipelined procesory mají mnohem kratší kroky, které se vykonávají mnohem kratší dobu, a tak je možné, aby procesor běžel na mnohem vyšší frekvenci.

Moje procesorové jádro již nyní nese náznaky pipeliningu a mělo by být možné ho předělat na tří až čtyř krokový pipelined procesor. Proč jen tři až čtyři kroky? Nemyslím si, že by prodlužování pipeline za tři až čtyři kroky přineslo nějaký benefit, protože by se následně začalo narážet na množství LUTů v FPGA – pokud je FPGA moc plné, dosažitelná frekvence se začne rapidně snižovat. Pěti kroková pipeline by velmi pravděpodobně byla možná, ale vzhledem k interní struktuře jádra si myslím, že tři až čtyři kroky jsou přirozenější a lepší volba.

Příliš dlouhá pipeline by tedy mohla být kontraproduktivní z hlediska dosažitelné frekvence. Navíc je velmi složité naplnit dlouhou pipeline instrukcemi, při stejné frekvenci by tedy např. pětistupňová pipeline byla vždy pomalejší než čtyřstupňová, jelikož by většinu času byla prázdnější.

Navíc mám poněkud svázané ruce, co se týče frekvence, na které jádro může běžet – PLL obvody na FPGA již došly, a je tedy problém vytvořit další hodinovou doménu. Jediná přijatelná možnost je vytvořit 20MHz doménu pomocí děličky 1:2 z 40MHz domény pro VGA obvod (myslím si, že dostat jádro na 40MHz je velmi nerealistické, ale tato možnost také existuje).

Alternativou by bylo generovat hodinový signál externě např. pomocí krystalového oscilátoru. U tohoto řešení ale vzniká problém *jak* tento hodinový signál dostat do FPGA – hodinové signály nelze do FPGA přivést skrz standardní digitální vstupy (respektive lze, ale takto vznikne mnoho problémů ohledně kvality daného hodinového signálu a hlavně jeho napojení uvnitř FPGA).

Hodinové signály by se do FPGA měly přivádět skrz speciálně přizpůsobené vstupy – když jsem prohledával rozsáhlou dokumentaci FPGA čipu iCE40 LP8K, nenašel jsem jejich lokaci. Navíc si obecně myslím, že řešení externího generování hodinového signálu by bylo velmi problematické.

Z těchto důvodů je tedy podle mě tří až čtyř kroková pipeline a frekvence jádra 20MHz nejlepší možnost. Toto řešení by zároveň umožnilo, aby paměť flash a paměťový řadič běžely na frekvenci plných 40MHz[11] (frekvenční přechod mezi jádrem a paměťovým řadičem by nebyl problém, jelikož by frekvence byly synchronizované a v přesném poměru 1:2). Toto navýšení frekvence by více než zdvojnásobilo propustnost z paměti flash, tato změna je více rozvinuta v následující kapitole.

[11] Dle datasheetu by flash paměť AT25SF081 měla zvládat frekvence až 85MHz, 40MHz by tudíž neměl být problém.

19.2. Paměťový řadič

Momentální paměťový řadič pracuje v nejjednodušším módu (0x03), kdy se po SPI sběrnici posílá z/do paměti flash 1 bit za cyklus (při frekvenci 16MHz).

Při každém čtení z flash paměti se musí poslat 8bitová opcode (specifikující typ operace), 24bitová adresa, z které se čte, a následně se musí přenést 32 bitů dat z flash paměti do paměťového řadiče. Při rychlosti přenosu 1 bit za cyklus je latence čtení z paměti 64 cyklů, to znamená, že při každém čtení z flash paměti musí procesor 64 cyklů čekat na data a nic nedělat. Pro porovnání, latence při čtení z paměti cache je pouze 1 či 2 cykly.

Paměť flash podporuje pokročilejší režim čtení (0xEB), kdy se některé data mohou posílat po SPI sběrnici rychlostí 4 bity za cyklus (díky zvýšení šířky sběrnice z 1 bitu na 4 bity). V tomto režimu se 8 bitová opcode posílá stále 8 cyklů, ale 24bitová adresa se posílá již jen 6 cyklů. Za ní následuje 8bitový prázdný Byte (dummy Byte), který se posílá 8 cyklů. 32bitová data se posílají jen 8 cyklů. Latence tedy spadne z původních 64 cyklů na 30 cyklů, tj. snížení latence na polovinu a zdvojnásobení propustnosti.

Zároveň by bylo možné využít „Continuous Read Mode“, kdy paměťový řadič nemusí stále dokola posílat 8bitovou opcode a může ji vynechat po prvním poslání. Díky tomu by se latence čtení dále snížila na 22 cyklů.

Dalším možným vylepšením by bylo zvýšení velikosti Cache bloků z 4B na 8B (z 32 bitů na 64 bitů. Toto vylepšení by mělo dva následky:

1. Lepší načítání dat z paměti. Pokud procesor zažádá o instrukci na adrese 44, automaticky by se z paměti flash načetla instrukce z adres 44 a 48. Instrukce 48 by tedy byla připravená v paměti ještě dříve, než si o ni procesor vůbec zažádá.[12]

2. Paměťový řadič nyní bude pokaždé načítat 64 bitů z paměti (místo 32 bitů). To by mělo následující efekty:

Velikost	Latence	**Efektivní** propustnost
4B	22 cyklů	32/22 = 1,45 bitů za cyklus
8B	30 cyklů	64/30 = 2,13 bitů za cyklus

Tabulka č. 3. – Vliv zvýšení velikosti Cache bloků na latenci a efektivní propustnost

[12] Velikost Cache bloků představuje trade off. Zatímco větší Cache bloky „před načítají“ procesoru data, zároveň větší Cache bloky zhoršují „granularitu“ čtení a zápisů a můžou mít tedy negativní dopad na výkon. Velikost 8B by, pro můj procesor, dle mého názoru měla zvýšit výkon. Ale už si netroufám odhadnout jaký efekt by měla velikost např. 12B.
Každopádně bude velmi zajímavé otestovat jak velký efekt (pozitivní či negativní) zvětšení velikosti Cache bloků na 8B bude mít na můj procesor.

Došlo by k navýšení latence o 8 cyklů (~36 %), ale zároveň k navýšení efektivní propustnosti o ~47 %. Ačkoliv procentuálně jsme získali více než ztratili, myslím si, že z pohledu paměťového řadiče se nejedná o úplně výhodný trade off – jelikož procesor ani nyní nenaráží na limit propustnosti, je latence o něco cennější než propustnost.

Možná vylepšení paměťového řadiče jsou shrnuta v tabulce číslo 4. Pro informaci jsem zahrnul i hypotetické varianty velikostí Cache bloků 12B a 16B.

Lze vidět, že paměťový řadič má velmi velký potenciál pro vylepšení. Jen použitím pokročilejšího režimu čtení lze latenci snížit na třetinu a zároveň 3x zvýšit propustnost. Velmi zajímavý je i efekt navýšení frekvence z 16MHz na 40MHz (umožněnému díky využití pipeliningu), který pro každou konfiguraci přináší přibližně 2,5× nižší latenci a 2,5× vyšší propustnost.

Zvýšení velikosti Cache bloků přineslo vždy navýšení propustnosti za cenu zhoršení latence.

Zároveň lze vidět zajímavý trend, kdy latence se v nejlepším případě zlepšila pouze ~7×, zatímco propustnost se v nejlepší případě zlepšila rovnou ~14×. Tento trend lze vidět i u moderních a velmi pokročilých procesorů a grafických karet, kdy je mnohem jednodušší zvyšovat propustnost (víceméně stačí použít širší sběrnici), než snižovat latenci.

Režim	Velikost	Frekvence	Latence	Latence %	**Efektivní** Propustnost	**Ef.** BW %
0x03	4B	16MHz	64 cyklů/4 µs	100,0 %	32/64 = 0,5 b/cyklus = 0,95 MB/s	100 %
0xEH	4B	16MHz	30 cyklů/1,88 µs	46,9 %	32/30 = 1,06 b/cyklus = 2,03 MB/s	213 %
0xEH + cont.	4B	16MHz	22 cyklů/1,38 µs	34,4 %	32/22 = 1,45b/cyklus = 2,77 MB/s	291 %
0xEH + cont.	8B	16MHz	30 cyklů/1,88 µs	46,9 %	64/30 = 2,13 b/cyklus = 4,07 MB/s	427 %
0xEH + cont.	12B	16MHz	38 cyklů/2,38 µs	59,4 %	96/38 = 2,52 b/cyklus = 4,82 MB/s	505 %
0xEH + cont.	16B	16MHz	46 cyklů/2,88 µs	71,9 %	128/46 = 2,78 b/cyklus = 5,31 MB/s	557 %
0xEH + cont.	4B	40MHz	22 cyklů/0,55 µs	13,8 %	32/22 = 1,45 b/cyklus = 6,94 MB/s	727 %
0xEH + cont.	8B	40MHz	30 cyklů/0,75 µs	18,8 %	64/30 = 2,13 b/cyklus = 10,17 MB/s	1067 %
0xEH + cont.	12B	40MHz	38 cyklů/0,95 µs	23,8 %	96/38 = 2,52 b/cyklus = 12,05 MB/s	1263 %
0xEH + cont.	16B	40MHz	46 cyklů/1,15 µs	28,8 %	128/46 = 2,78 b/cyklus = 13,27 MB/s	1391 %

Tabulka č. 4. – Shrnutí možných vylepšení paměťového řadiče

Legenda:

0x03	Jednoduchý čtecí mód, propustnost SPI sběrnice 1 bit za cyklus
0xEH	Lepší čtecí mód, propustnost SPI sběrnice 4 bity za cyklus
0xEH + cont.	Lepší čtecí mód + "Continous read mode"
Velikost	Velikost Cache bloků
b/cyklus	bitů za cyklus
Ef. BW	Efektivní propustnost
Efektivní propustnost	Maximální reálně dosažitelná efektivní propustnost Efektivní = hodnota je snížena o komunikační režii (overhead) - např. posílání adres Představuje tedy reálné množství užitečných dat, které z paměti dostaneme

Tabulka č. 5. – Legenda k tabulce č. 4.

20. ZÁVĚR

Na začátku projektu jsem měl poměrně dobrou představu, jak by mělo vypadat procesorové jádro v mém procesoru. Ale neměl jsem vůbec žádnou představu o tom, co všechno obnáší věci okolo – když jsem během měsíce až dvou občasné práce navrhl (skoro) celé procesorové jádro, netušil jsem, že zbytek procesoru a jeho odladění mi bude trvat dalších 7 až 8 měsíců.

Když se na to zpětně podívám, přijde mi až neskutečné, že všechny ty jednotlivé části do sebe zapadly a všechno funguje tak jak má. Ještě půl roku zpět jsem měl FPGA zapojené na nepájivém poli spolu s osmi LEDkami a snažil se zprovoznit rozhraní pro klávesnici, dnes ho mám připojené k monitoru a jsem na něm schopný spustit programy napsané v jazyce C.

Nikdy jsem si nemyslel, že na mém procesoru někdy poběží tak velké programy. Vždycky jsem si říkal: „Navrhnu procesor, a spustím na něm *nějaký* program“. Zároveň jsem nikdy netušil, kolik instrukcí je potřeba i na tak relativně jednoduchý program – aktuální program (kalkulačka + poznámkový blok + tanky) má přes **6 tisíc** instrukcí. Očekával jsem, že toto číslo bude 2-3x menší.

Jelikož se instrukce nevykonávají jen jednou, programy jsou plné opakujících se cyklů, lze očekávat, že počet zpracovaných instrukcí bude mnohem vyšší – vytvořil jsem tedy speciální registr, který ukládá počet zpracovaných instrukcí, a napojil ho na debugovací logiku. Vypočítání příkladu na kalkulačce – **10 tisíc** instrukcí. Tři výstřely v jediné hře tanků – **300+ tisíc** instrukcí (toto číslo zní až neuvěřitelně a několikrát jsem ho kontroloval). A každá z těchto instrukcí musí být vykonána bezchybně.

Pro ilustraci, jak moc ambice tohoto projektu narostly – když jsem si někdy v září na papír čmáral různé možné konfigurace pamětí cache, došel jsem k závěru, že instrukční i datovou cache implementuji jakožto čtyřcestně asociativní s velikostí 4 kB, ale zastával jsem názor, že obě paměti cache budou zbytečně velké a „výkonné“, a že se bude jednat spíše o akademické cvičení a o „frajeřinu“, kterou se budu moct chlubit.

Tato práce dokonce v jednu chvíli obsahovala větu „Bylo mi jasné, že procesor nikdy tak velké paměti cache nevyužije“. Když jsem si ale později přepočítal, že do 4kB instrukční cache se vleze maximálně 1024 instrukcí, ale můj program má už více jak 6000 unikátních instrukcí, okamžitě jsem tuto větu smazal.

Nejvíce pyšný jsem na návrh obrazového procesoru (Display engine), který je díky chytré optimalizaci ukládání znaků zakódovaných v ASCII, místo ukládání jednotlivých pixelů, velmi jednoduchý a efektivní.

Nejtěžší částí pro mne byl návrh pamětí cache. Vzal jsem si na sebe poměrně velké břímě, jelikož jsem se s nulovými zkušenostmi (co se týče návrhu pamětí cache) pustil rovnou do čtyřcestně asociativní cache. Na té jsem se několikrát posekal a musel ji asi třikrát kompletně přepisovat. V jednu chvíli jsem dokonce zvažoval, že se čtyřcestně asociativní cache vzdám a implementuji jednoduchou direct mapped cache. Ale nakonec se mi podařilo obě paměti cache úspěšně zprovoznit jakožto čtyřcestně asociativní – na toto jsem také náležitě pyšný.

Další nesmírně obtížnou částí vývoje bylo testování a ladění chyb v procesoru – hledání chyb bylo neuvěřitelně obtížné. S tím mi značně pomohl simulátor ModelSim, linter Verilator a v neposlední řadě debugovací rozhraní, které jsem si sám navrhnul.

Procesor bych chtěl v budoucnu vylepšit na mnoha frontách, jak už bylo v práci naznačeno:

1. Chtěl bych na 3D tiskárně vytisknout krabičku, aby procesor působil jako malý přenosný počítač.
2. Doplnit jádro o podporu instrukčního rozšíření C, které zvyšuje výkon skrz menší velikost programu.
3. Předělat jádro procesoru tak, aby využívalo pipelining [22] (řetězení instrukcí), což by vedlo ke zvýšení výkonu.
4. Vylepšit SPI řadič, aby podporoval pokročilejší (a rychlejší) režimy čtení z SPI paměti.
5. Zvýšit velikost cache bloků na 64 bitů (oproti současným 32 bitům) a otestovat vliv na výkon.

Dále bych chtěl pro procesor vytvořit výpočetně o dost komplikovanější uživatelský program, který by plně předvedl schopnosti mého procesoru (chtěl bych otestovat jeho limity).

Co mě na vývoji velmi překvapilo, bylo, jak často jsem narážel na podobné problémy/poznatky které jsou totožné s velkými (nebo i jednoduchými) procesory a grafickými kartami, i přes to, jak jednoduchý můj procesor je. Například fakt, že zvýšit propustnost paměťového subsystému je mnohem jednodušší než snížit jeho latenci. Ačkoliv se jedná o běžně známý fakt, nikdy bych nečekal, že na něj čistě náhodou narazím i u vývoje mého procesoru.

Nebo debugovací rozhraní – FPGA leželo na stole přede mnou, ale já měl problém zprovoznit procesor, jelikož do FPGA nevidím a nevím co je špatně. Tudíž jsem si navrhl debugovací rozhraní, abych viděl, co se uvnitř děje. A později mi došlo, že úplně stejný princip debugovací sběrnice se musí využívat u návrhu procesorů a grafických karet – jinak by je inženýři v životě nebyli schopní zprovoznit, když se jim z továrny vrátí první prototypy plné chyb. Fakt, že jsem osamoceně a izolovaně došel ke stejnému řešení problému, mě velmi potěšil.

Můj procesor by se dal použít jako pomůcka pro výuku programování mikrokontrolerů – atraktivnost vidím zejména ve velmi jednoduché práci s klávesnicí a displejem (pro zápis na displej stačí pouhé tři instrukce). Zároveň by se na mém procesoru dala jednoduše vyučovat architektura a stavba procesorů.

Myslím si, že jsem dosáhl svého cíle a jsem se svou prací spokojený a těším se na návrh budoucích procesorů.

CITACE

[1] - *Xilinx* [online]. [cit. 2021-03-18]. Dostupné z: https://www.xilinx.com/applications/aerospace-and-defense.html

[2] – *Fixní cena vývoje ASIC* [online]. [cit. 2021-03-18]. Dostupné z: https://www.extremetech.com/computing/272096-3nm-process-node

[3] – *Xilinx Virtex Ultrascale+* [online]. [cit. 2021-03-18]. Dostupné z: https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html#productTable

[4] – *ICE40* [online]. [cit. 2021-03-18]. Dostupné z: http://www.latticesemi.com/iCE40

[5] – *HW bloky v FPGA* [online]. [cit. 2021-03-18]. Dostupné z: https://www.nandland.com/articles/block-ram-in-fpga.html

[6] – *IceStorm* [online]. [cit. 2021-03-18]. Dostupné z: https://github.com/YosysHQ/icestorm

[7] – *RISC-V Manual* [online]. [cit. 2021-03-18]. Dostupné z: https://github.com/riscv/riscv-isa-manual/releases/tag/draft-20210316-ad5f04d

[8] *Onur Mutlu Lectures* [online]. [cit. 2021-03-18]. Dostupné z: https://www.youtube.com/channel/UCIwQ8uOeRFgOEvBLYc3kc3g

[9] – *VGA časování* [online]. [cit. 2021-03-18]. Dostupné z: http://tinyvga.com/vga-timing/800x600@60Hz

[10] – *VGA časování* [online]. [cit. 2021-03-18]. Dostupné z: http://tinyvga.com/vga-timing/640x480@60Hz

[11] – *GNU Toolchain* [online]. [cit. 2021-03-18]. Dostupné z: https://github.com/riscv/riscv-gnu-toolchain

[12] – *Prebuilt GCC* [online]. [cit. 2021-03-18]. Dostupné z: https://github.com/stnolting/riscv-gcc-prebuilt

[8] – *RISC-V from scratch* [online]. [cit. 2021-03-18]. Dostupné z: https://twilco.github.io/riscv-from-scratch/2019/04/27/riscv-from-scratch-2.html

[12] – *Scancodes tabulka* [online]. [cit. 2021-03-18]. Dostupné z: https://www.digikey.com/eewiki/pages/viewpage.action?pageId=28278929#PS/2KeyboardInterface(VHDL)-ExampleTransaction

[13] – *PS/2 řadič* [online]. [cit. 2021-03-18]. Dostupné z: http://www.burtonsys.com/ps2_chapweske.htm

[14] – *VGA časování* [online]. [cit. 2021-03-18]. Dostupné z: https://youtu.be/l7rce6IQDWs

[15] – *Zip CPU* [online]. [cit. 2021-03-18]. Dostupné z: https://zipcpu.com/

[16] – *Verilog Coding Style* [online]. [cit. 2021-03-18]. Dostupné z: https://github.com/lowRISC/style-guides/blob/master/VerilogCodingStyle.md#signal-naming

[22] *Digital design and computer architecture*. 2nd ed. Waltham: Morgan Kaufmann, c2013. ISBN 978-0123944245.

[23] *Adesto AT25SF081 Datasheet* [online]. [cit. 2021-03-25]. Dostupné z: https://www.adestotech.com/wp-content/uploads/DS-AT25SF081_045.pdf

[24] CUMMINGS, Clifford E. *Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog* [online]. Sunburst Design, Inc., 2008 [cit. 2021-5-17]. Dostupné z: http://www.sunburst-design.com/papers/CummingsSNUG2008Boston_CDC.pdf

[25] CUMMINGS, Clifford E. *Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!* [online]. Sunburst Design, Inc., 2000 [cit. 2021-5-17]. Dostupné z: http://www.sunburst-design.com/papers/CummingsSNUG2008Boston_CDC.pdf

[26] LIBERDA, Dominik. *KVN Prediktor Větvení* [online]. Sedliště, 2020 [cit. 2021-5-23]. Dostupné z: https://github.com/Dom324/SOC-2020/blob/main/KVN%20Prediktor%20V%C4%9Btven%C3%AD.pdf

SEZNAM ZKRATEK A POJMŮ

ABI – Application Binary Interface, rozhraní mezi procesorem a programem, definuje, jak se mají programy „chovat“

ASCII – tabulka kódů se znaky abecedy

Bitstream – binární soubor určený k nahrání na FPGA

Cache – paměť mezi procesorovým jádrem a hlavní pamětí, slouží jako odkládací buffer pro často užívané proměnné

FIFO – blok paměti, používá se pro přechod mezi frekvenčními doménami

Flash paměť – typ paměti

Flip-Flop – malá registrová paměť

FPGA – Field Programmable Gate Array, programovatelné hradlové pole

HDL – Hardware Description Language, jazyk pro popis digitálních obvodů

I/O – vstup/výstup

LUT – Look Up Table. Jednotka v FPGA pro realizaci logických funkcí

Overhead – přebytečná režie, neefektivita

PS/2 – rozhraní pro klávesnici/myš

RTL – Register Transfer Level, simulace na úrovni registrů

TTL – Transistor Transistor Logic, druh digitálních obvodů

VGA – rozhraní pro připojení monitoru

SEZNAM OBRÁZKŮ A TABULEK

Seznam Obrázků

Seznam Tabulek

Seznam Příloh

1. Složka procesor, obsahující zdrojové kódy k procesoru
2. Složka program, obsahující zdrojové kódy k programu