ČVUT

Fakulta elektrotechnická

Katedra mikroelektroniky

Elektronika a komunikace

TUTORIÁL ZÁKLADŮ VHDL NA ARDUINO KITU

Bakalářská práce

Autor: Daniel Krysa
Vedoucí práce: Ing. Vladimír Janíček, Ph.D.
Rok: 2021

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Krysa** Jméno: **Daniel** Osobní číslo: **456941**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra mikroelektroniky**

Studijní program: **Elektronika a komunikace**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Tutoriál základů VHDL na Arduino kitu

Název bakalářské práce anglicky:

Basic VHDL Tutorial with Arduino kit

Pokyny pro vypracování:

1) Analyzujte dostupné metody výuky základů VHDL (principy, syntaxe zápisu) na jednočipových kitech Arduino.
2) Zvolte vhodný kit a vytvořte pro něj sadu online tutoriálů pro výuku základů VHDL.
3) Využijte volně dostupné IDE.

Seznam doporučené literatury:

1) Datasheet a white papers ARDUINO MKR VIDOR, https://store.arduino.cc/mkr-vidor-4000
2) Tutorial - Introduction to VHDL,
https://www.nandland.com/vhdl/tutorials/tutorial-introduction-to-vhdl-for-beginn
ers.htm

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Vladimír Janíček, Ph.D., katedra mikroelektroniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **01.10.2019** Termín odevzdání bakalářské práce: **05.01.2021**

Platnost zadání bakalářské práce: **30.09.2021**

Ing. Vladimír Janíček, Ph.D.	prof. Ing. Pavel Hazdra, CSc.	prof. Mgr. Petr Páta, Ph.D.
podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.

Datum převzetí zadání Podpis studenta

© VUT v Praze, Design: VUT v Praze, VIC

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne ....

Jméno, příjmení

Poděkování

Rád bych poděkoval vedoucímu mé práce Ing. Vladimíru Janíčkovi, Ph.D. za odborné vedení, za připomínky a konstruktivní kritiku, které mě vedly vpřed.

Abstrakt

Tato bakalářská práce se věnuje vytvoření tutoriálu programovacího jazyka VHDL. Je zaměřena na studenty prvních ročníků. Jako platforma pro testování úkolů a programů byl použit kit Arduino MKR VIDOR 4000. Práce přibližuje VHDL a programování FPGA čipů začátečníkům.

Klíčová slova: VHDL, FPGA, Arduino MKR Vidor

Abstrakt

This bachelor thesis describes the creation of VHDL programming language tutorials. Is focused on first-year students at university. As a platform for testing tasks and programs is used Arduino MKR Vidor board. The work introduces VHDL and FPGA chip programming to beginners.

Key Words: VHDL, FPGA, Arduino MKR Vidor

Obsah

Úvod a motivace **4**

1 Arduino **5**

1.1 Desky Arduino 6

1.2 MKR Vidor 4000 7

1.2.1 Komponenty 8

2 FPGA **10**

2.1 Historie FPGA 11

2.1.1 SPLD 11

2.1.2 HPLD 12

2.2 Architektura FPGA 13

2.2.1 Základní stavba FPGA 14

2.2.2 Typy FPGA 17

2.3 Intel Cyclone 10CL016 20

3 VHDL **24**

3.1 Materiály 24

3.2 Vlastnosti 24

3.3 Historie 25

3.4 Knihovny a návrhové jednotky 26

3.4.1 Typy 28

3.4.2 Entita 31

3.4.3 Architektura 32

3.4.4 Ostatní návrhové jednotky 34

3.5 Základní datové objekty 35

3.5.1 Konstanty 35

3.5.2 Proměnné 35

3.5.3 Signály 35

3.6 Podmínky 36

3.6.1 Case 36

3.6.2 If 36

3.6.3 With-select 37
3.6.4 When-else 37
3.7 Podprogramy 38
3.7.1 Funkce 38
3.7.2 Procedury 38
3.8 Programy 39
3.8.1 Realizace základních hradel 39
3.8.2 Sekvenční a kombinační obvody 40

4 Výukové prezentace 45
4.1 1. prezentace - základní seznámení s Arduinem 45
4.2 2. prezentace - základy VHDL 47
4.3 3. prezentace - nahrání kódu 48
4.4 4. prezentace - realizace semaforu 48
4.5 5. prezentace - realizace stopek 49
4.6 6. prezentace - realizace hodin 51
4.7 Struktura prezentací 52

5 Realizace a vzniklé překážky 56

Závěr 57

Seznam použitých zkratek 58

A Příloha 60

References 62

Seznam obrázků 66

Seznam tabulek 67

Úvod a motivace

Tato práce se zabývá vytvořením podkladů k základnímu programování čipů FPGA v jazyce VHDL. K tomu je využita deska Arduino, jenž je platforma s grafickým vývojovým prostředím, která je legálně volně dostupná široké veřejnosti. Pro programování na Arduinu se využívá volně nainstalovatelné integrované vývojové prostředí Arduino IDE. Ve své podstatě je Arduino ideálním jednodeskovým mikrokontrolerem pro začínající či pokročilé programátory.

V této práci je použita deska Arduino MKR Vidor 4000, která se vymyká ostatním Arduino deskám a to hlavně díky programovatelnému hradlovému poli FPGA, k jehož programování se využívají programovací jazyky HDL, které slouží k popisu hardwaru.

Cílem mé práce je vytvořit tutoriál základů programovacího jazyka VHDL a seznámit studenty a veřejnost s prototypovací deskou Vidor. Tyto prezentace tak mohou sloužit k výuce základů jazyka VHDL a programování desek s FPGA čipem.

Motivací pro vytvoření mé práce je obohacení vědomostí v programování VHDL. Jelikož jsem studoval na gymnáziu, kde jsem nezískal žádné zkušenosti s programováním, byly pro mě poměrně složité začátky na vysoké škole. Chtěl bych tak studentům, jako jsem byl já, zjednodušit začátky s programováním a vytvořit materiály, které budou velmi názorné, detailně popsané a vysvětlené.

1 Arduino

Arduino je otevřená (open source) elektronická platforma založená na snadno použitelném hardwaru a softwaru [3]. Společnost nabízí řadu harwerových platforem, softwarových nástrojů a dokumentace, které všem umožňují být kreativní. Díky volné dostupnosti se dá vyhledat na webu spoustu návodů, článků a dalších užitečných informací o této platformě, které mohou pomoci začátečníkům ale i profesionálům. Jejich desky umí jednoduše proměnit vstupy, jako je použití tlačítka, teplota na senzoru a mnoho dalších, na výstupy, které například spustí servomotor nebo rozsvítí světlo. Díky tomu mohou být použity v nespočtu různých projektů a aplikací. Arduino je výborným nástrojem pro učení se něčemu novému a zároveň podporuje kreativitu.

Arduino vzniklo za účelem vytvoření jednoduchého nástroje pro prototypování, který byl primárně určen studentům designu bez znalosti elektroniky či programování [3]. Počátky Arduina začaly ve městě Ivrea v roce 2000 jako výzkumný projekt. V roce 2005 už byla vydána první deska, která byla založena na jednoduchém 8bitovém mikrokontroleru: Atmel AVR [4]. Ve skutečnosti byla původní deska prvním velmi úspěšným hardwarovým projektem s otevřeným zdrojovým kódem. Díky nízkým nákladům a jednoduchému Arduino IDE se Arduinu dostalo značné popularity, což vyústilo ve vznik dalších verzí desek, které se přizpůsobovaly novým potřebám uživatelů jako jsou studenti, vývojáři, fanoušci, apod. Arduino se dostalo do špičky mezi výrobci elektroniky a to především pro řešení IoT [4].

1.1 Desky Arduino

Postupem času byla vytvořena řada desek, které se mezi sebou liší využitelností, porty, komponenty, rozměry, cenou, ale také výkonem (viz tab. 1). Většina desek je založená na 8bitových AVR mikrokontrolerech ATmega od firmy Atmel [10].

Tabulka 1: Arduino desky [10]

Arduino	Procesor	Operační napětí	Vstupní napětí	Taktovací frekvence	Digitální I/O piny	Analog. I/O piny	SRAM	EEPROM	Flash
Uno	ATmega328P	5 V	7-12 V	16 MHz	20	6	2 kB	1 kB	32 kB
Ethernet	ATmega328P	5 V	7-12 V	16 MHz	18	6	2 kB	1 kB	32 kB
Nano	ATmega328	5 V	7-12 V	16 MHz	22	8	2 kB	1 kB	32 kB
Leonardo	ATmega32U4	5 V	7-12 V	16 MHz	20	12	2,5 kB	1 kB	32 kB
Micro	ATmega32U4	5 V	7-12 V	16 MHz	20	12	2,5 kB	1 kB	32 kB
Mega 2560	ATmega2560	5 V	7-12 V	16 MHz	54	16	8 kB	4 kB	256 kB
Due	ATSAM3X8E	3,3 V	7-12 V	84 MHz	54	18	96 kB	- kB	512 kB
Zero	ATSAMD21G18	3,3 V	7-12 V	48 MHz	20	7	32 kB	- kB	256 kB
MKR1000	SAMD21 Cortex-M0+	3,3 V	5 V	48 MHz	8	8	32 kB	- kB	256 kB
MKR VIDOR 4000	SAMD21 Cortex-M0+	3,3 V	5 V	48 MHz	8	8	32 kB	- kB	256 kB

Nejoblíbenější a nejrobustnější deskou je bezesporu Arduino Uno (viz obr. 1) [11]. Tato deska je založená na ATmega328P, je vybavená USB portem a je nejvhodnější pro začátek práce s elektronikou a kódováním. Téměř totožnou deskou je Arudino Ethernet u níž byl USB port nahrazen rozhraním Wiznet Ethernet pro připojení k síti. Další velmi podobnou deskou, ovšem 5krát menší, je Arduino Nano se stejným mikrokontrolerem jako Uno [9].

Obrázek 1: Arduino Uno [11]

Arduino Leonardo je obdobnou verzí předchozích zmíněných desek, jen pracuje na ATmega32U4 [5], která má vestavěnou USB komunikaci, což eliminuje potřebu sekundárního procesoru. Díky tomu se objeví v počítači připojená stejně jako klávesnice nebo myš. Dalším je Arduino Micro [7], které je nejmenší Arduino deskou a stejně jako u typu Leonardo pracuje na ATmega32U4. Pro komplexnější projekty je vhodná Arduino Mega 2560 [6], která disponuje 54 digitálními vstupními a výstupními piny (jako PWM výstupy může být využito 15 z nich). Disponuje větší pamětí a větším výkonem než například typ Uno. Jako první Arduino využívající 32-bitové ARM jádro bylo vytvořeno Arduino Due. Dalšími typy jsou desky určené pro IoT.

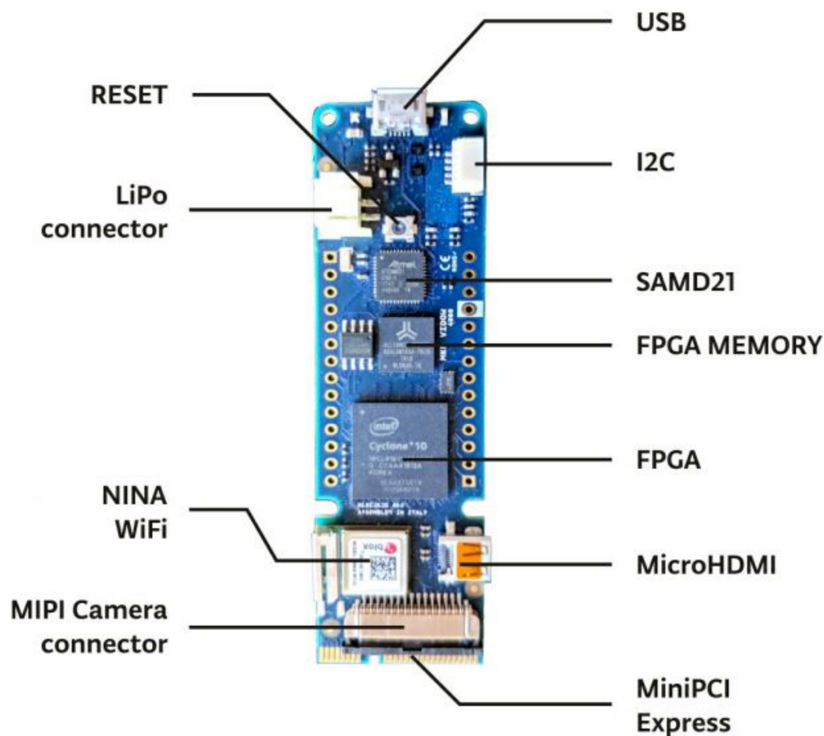
1.2 MKR Vidor 4000

Může se stát, že pro některé aplikace již hardware klasických Arduino desek není dostačující. Tuto problematiku lze vyřešit použitím přeprogramovatelných čipů FPGA. Cílem vývojářů tedy bylo vytvořit desku, která dá uživatelům kombinaci flexibility a vysokého výkonu FPGA s jednoduchostí, na které si Arduino tak zakládá. Z hlediska hardwaru přidali FPGA do architektury desky. Dochází tak k rozšíření periferních zdrojů mikrokontroleru [8].

Deska, která tímto spojením vznikla, se nazývá MKR Vidor 4000. Jedná se o jednu z nejpokročilejších a nejnovějších desek z celého portfolia, které Arduino nabízí [8]. Deska kombinuje nízkoenergetický mikrokontroler firmy Atmel (spadající pod firmu Microchip) SAM D21 s FPGA čipem Intel Cyclone 10CL016.

1.2.1 Komponenty

Deska disponuje mnoha komponentami (viz obr. 2).



Obrázek 2: Arduino MKR VIDOR 4000 [10]

Mini PCIe: Jedná se o standard, který slouží k připojení desky k dalším zařízením. Klasické.

SAM D21: Tento mikrokontroler využívá na práci 32bitový ARM Cortex-M0+ procesor [8]. Jedná se o mikrokontroler disponující 256kB flash paměti a 32kB paměti SRAM s pracovním taktem 48MHz.

FPGA Cyclone 10CL016 a FPGA Memory (SDRAM): Tento čip a SDRAM jsou popsány v kapitole 2.3.

CSI/MIPI: Rozhraní, jenž je pro připojení kamery. Je to standartní formát, který lze najít u několika produktů na trhu. Ovšem né všechny kamery, které lze připojit, fungují v kombinaci s Vidorem. Kompatibilní kamerou je například Omnivision OV5647.

Micro HDMI: Klasický video konektor umožňující připojení monitoru. V případě připojení kamery k MIPI camera konektoru, lze přenášet obraz, který kamera snímá.

NINA-W102 u-blox: Umožňuje bezdrátovou komunikaci desky. Tato komponenta poskytuje podporu Wi-fi 802.11b/g/n a Bluetooth dual-mode v4.2 [12].

USB a Li-Po: K napájení této desky se využívá microUSB port [8], jenž je přímo napojen na SAM D21 . Na počítači je vidět jako COM port a proto může být použit k odesílání a přijímání zpráv pomocí **Serial()** funkce a Arduino Software (IDE) . Další možností je napájení přes konektor pro lithium-polymerové baterie. Vidor je schopen pracovat díky baterii s napětím 3,3V. Tato baterie se sama dobíjí při připojení k USB. Čip, který je využívaný ke kontrole nabíjecího procesu, komunikuje přímo se SAM D21.

I2C: Tímto konektorem lze připojit nízkorychlostní periferie k desce a pomocí I2C sběrnice probíhá komunikace mezi propojenými zařízeními [8].

Následující kapitola se bude věnovat technologii FPGA jakožto celku a poté i konkrétnímu čipu, kterým deska disponuje. Bude se snažit FPGA přiblížit lidem, kteří zatím nemají jakoukoli zkušenost s těmito čipy.

2 FPGA

FPGA neboli programovatelná hradlová pole jsou speciální polovodičová zařízení. Jsou to ve své podstatě integrované obvody, které obsahují spoustu hradel a I/O (vstupních/výstupních) obvodů, které umožňují přijímat data ze zdroje a předat je na druhém konci jinému systému nebo subsystému [33]. Díky své flexibilitě, jednoduchosti, programovatelnosti a dalším výhodám se tyto obvody v dnešním světě uplatňují ve velkém množství aplikací. Uplatňují se v hudebních, medicínských zařízeních, také v telekomunikacích, automobilovém, vesmírném nebo vojenském průmyslu. Pomocí FPGA lze vytvořit spoustu obvodů například komplexní procesor.

V současné době je o FPGA velký zájem a stále roste, protože je hojně využíván ve spojení s AI nebo IoT, dále při přechodu na 5G síť [31] je využíváno v datových centrech pro aplikace, které vyžaduji velký výpočetní výkon. FPGA je na vzestupu co se týče trhu a to by mělo také pokračovat v následujících letech.

Pomocí FPGA vytváříme programováním hardware. Kdybychom měli realizovat, nějaké složitější zapojení, reálně bychom potřebovali spoustu komponent propojit a dát dohromady, FPGA to udělá za nás [28]. FPGA je poměrně výkoné a například obvody zákaznické integrované obvody ASIC (Application Specific Integrated Circuit) jsou sice také výkonné, nicméně nejsou tak univerzální jako FPGA. Také v porovnání s další konkurencí v podobě mikrokontrolerů si stojí velmi dobře. Mikrokontrolery jsou oproti FPGA poměrně pomalé.

2.1 Historie FPGA

Počátky historie obvodů FPGA můžeme datovat od roku 1847 [23], kdy Booleovou algebru ve své knize představil George Boole. Ta nepoložila základy jen obvodům FPGA, ale celé výpočetní technice. Dalším milníkem bylo vytvoření logického obvodu Booleovy algebry z elektromechanických relé, za kterým stál Claude Shannon ve 30. letech minulého století. Mezi další historické milníky ve vzniku FPGA můžeme počítat vznik MOSFETU (r.1960) [23], který představuje základní element čipů FPGA (typy P-FET a N-FET). Dále vznik standardu logických (TTL - Tranzistor-Tranzistor Logic) integrovaných obvodů (r.1962). Tyto integrované obvody nejsou vytvořeny technologií MOSFET, nýbrž křemíkovými bipolárními tranzistory, které nebyly tak energeticky náročné. Jako další stojí za zmínku první CMOS obvody (r.1963), které byly tvořené kombinací MOSFETů, což umožnilo vytvářet větší integrované obvody s menšími náklady. Poté přišel na svět první SRAM (r.1964) (viz. kapitola 2.2.2) nebo Moorův zákon, za kterým stal Gordon Moore spoluzakladatel firmy Intel [23].

Pro přehlednost dalšího vývoje si rozdělíme programovatelné logické obvody do dvou kategorií a to SPLD (Simple Programmable Logic Device) [28], neboli jednoduché programovatelné logické zařízení a HPLD (High-Density Programmable Logic Device), což je programovatelné logické zařízení s vyšší hustotou logických bloků.

2.1.1 SPLD

- PROM (Programmable Read Only Memory)
- F-PLA (Field-Programmable Logic Array)
- PAL (Programmable Array Logic)
- GAL (Genetic Array Logic)

Historie modernějších programovatelných polích (PLD) [28], začala v roce 1970 spolu s vytvořením programovatelných pamětí PROM [23] firmou Radiation, což můžeme považovat za velký milník. PROM je sice programovatelná paměť, ale právě jednou (kdysi fungovala na principu přepalování propojek). Dnes se kvůli své vlastnosti moc nevyužívá. Za

účelem vícenásobného přeprogramování vznikla o rok později paměť EPROM v dílnách firmy Intel, u které před novým naprogramováním muselo dojít ke smazání paměti pomocí UV záření, čímž se liší od paměti EEPROM, která byla vyrobena roku 1983 stejnou firmou, jenž k přeprogramování využívá elektrický způsob mazání (signály resp. napětím, jenž jsou větší než napětí napájecí). Před vytvořením znovu přeprogramovatelných pamětí vznikl F-PLA obvod (r.1970). Tento obvod se zakládal na technologii PROM. Skládal se ze dvou programovatelných matic a to konkrétně z AND a OR, přičemž tyto matice vytvářely požadovaný výstup. Toto řešení bylo pomalé a poněkud finančně náročné. Stále se zdaleka nepřibližovalo flexibilitě, kterou dnes požadujeme od FPGA [21, 2].

V roce 1978 byl vytvořen PAL, jenž také využíval matic AND a OR, ovšem matice OR již nebyla programovatelná. Toto řešení bylo rychlejší a levnější, proto bylo poměrně hojně využíváno. Jenže stále bylo vytvořeno pomocí pamětí PROM, tudíž bylo programovatelné pouze jednou [21, 2, 28].

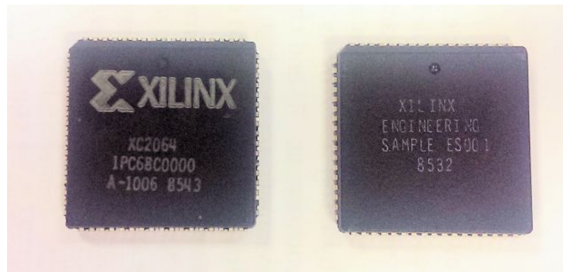
Logickým vývojovým krokem bylo vytvoření podobného obvodu jako PAL, ale vícekrát programovatelného. Proto byly vytvořeny obvody GAL (r.1983), které využívaly paměť EEPROM a technologii CMOS, což je umožňovalo až 10 000krát vymazat a přeprogramovat. Tyto logické obvody měly také programovatelné I/O konfigurace. Díky kombinaci EEPROM a CMOS bylo dosaženo výrazně rychlejší odezvy [25, 23, 28].

2.1.2 HPLD

- CPLD (Complex Programmable Logic Device)
- FPGA (Field-Programmable Gate Array)

S vývojem technologií rostla postupně složitost obvodů a s tím i počet neznámých. Seskupením PLD obvodů vznikly obvody CPLD, založené na EEPROM. Tyto obvody lze považovat jako kombinaci obvodů PAL a GAL [23]. CPLD obvody jsou vhodné pro malé až střední počty bran. Oproti FPGA nemá tak složitou architekturu. To znamená, že zpoždění jsou předvídatelnější než u architektury FPGA určenou pro vysoký počet bran. CPLD se nejčastěji využívá v jednodušších a středně složitých logických aplikacích [17]. Od CPLD už nebylo daleko k FPGA. Ale ještě předtím se na trhu v roce 1984 objevila

paměť FLASH, která se od EPROM a EEPROM lišila možností vymazání pouze části paměti, nemusela být přemazána jako její předchůdci [2]. V následujícím roce se již na trhu objevil první čip FPGA s 800 hradly od firmy Xilinx XC2064 (viz obr. 3) [34]. Pro vývojáře a návrháře elektronických systémů jsou CPLD a FPGA díky své vhodnosti a dostupnosti obecně nejpoužívanější. Ovšem potenciál do budoucna je u CPLD poměrně omezen a rychlejší růst a vývoj zažívá FPGA a se stejnou či podobnou tendencí se počítá i do budoucna.



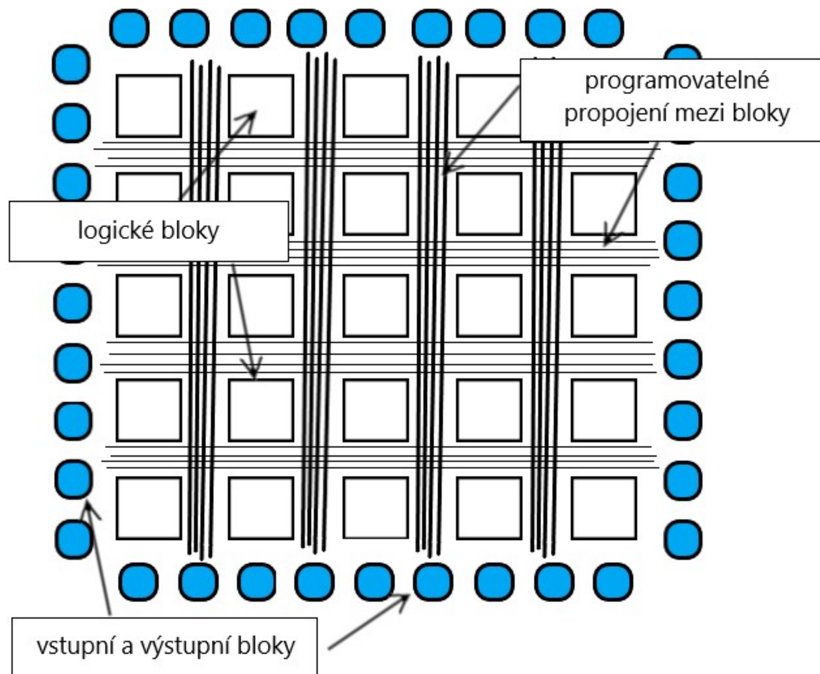
Obrázek 3: Xilinx XC2064 [34]

2.2 Architektura FPGA

Obvody FPGA jsou nejpoužívanější polovodičová zařízení, která lze elektronicky naprogramovat, aby se chovala jako systém či digitální obvod. Jedná se o pole přeprogramovatelných hradel. Základem architektury FPGA je pravidelná struktura sestavená z logických bloků [2]. FPGA bylo určeno, jak napovídá jeho název field programmable gate array, pro naprogramování u koncového uživatele, nikoli u výrobce. Je důležité zmínit, že každý výrobce má svou vlastní specifickou FPGA architekturu, ale principiálně jsou všechny velmi podobné. Existují 3 základní typy programovatelných elementů, které se využívají viz. 2.2.2 [21, 28].

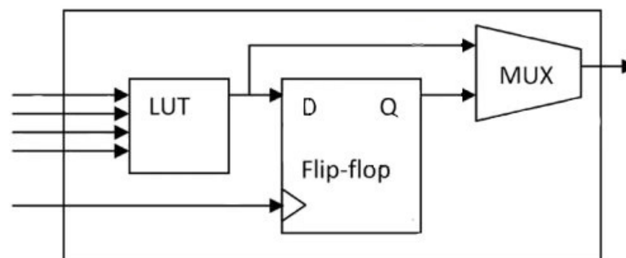
2.2.1 Základní stavba FPGA

Architektura FPGA má základ v konfigurovatelných logických blocích tzv. LAB (Logic Array Block) (viz obr. 4) [2], které jsou tvořeny logickými buňkami LE (Logic Element).



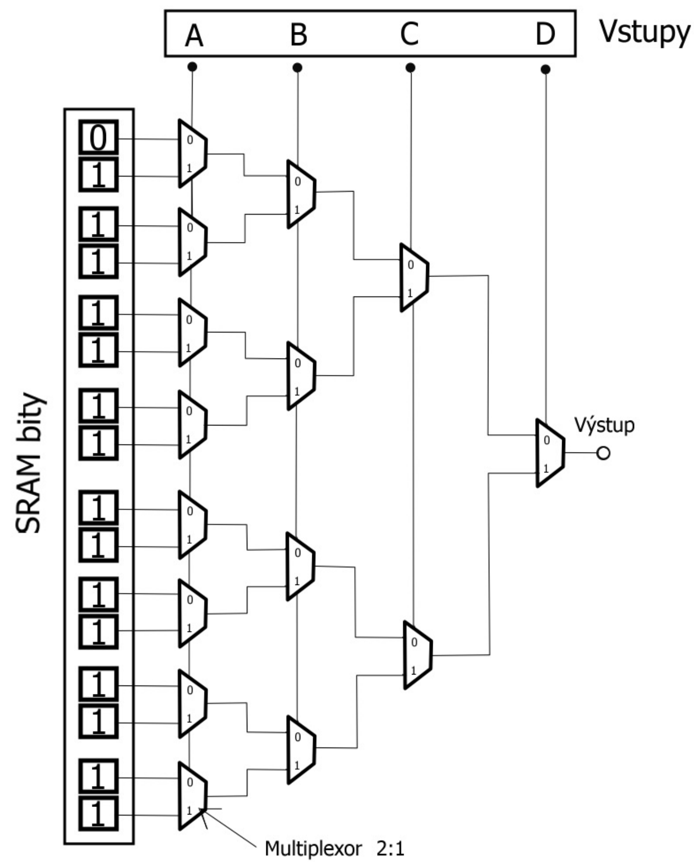
Obrázek 4: Struktura FPGA

Logické buňky jsou v jednoduchosti pak nadále složeny z LUT (Look up table), klopného obvodu a multiplexoru, který má za úkol vybrat adekvátní výstup (viz obr. 5). Vybírá z kombinačního nebo registrového výstupu [25, 28].



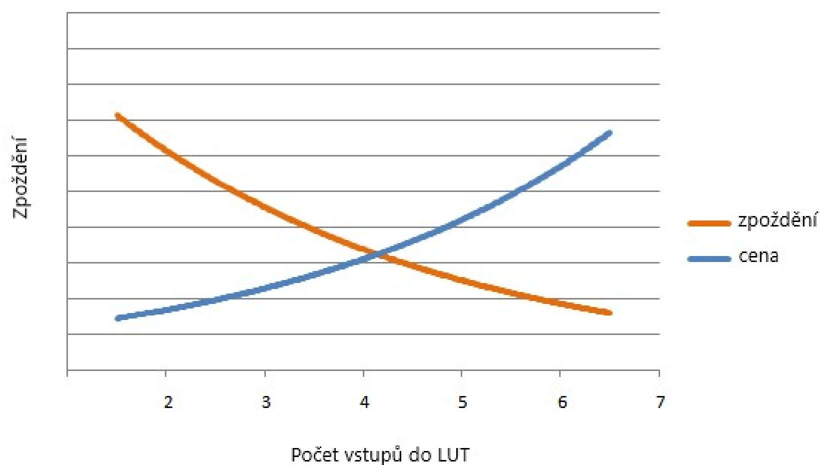
Obrázek 5: Logická buňka [23]

Zmíněný LUT (viz obr. 6) je vyhledávací tabulka [2], která dokáže uskutečnit všechny kombinační logické funkce vstupů. Ve smyslu kombinační logiky se jedná o pravdivostní tabulku. V levé části jsou SRAM bity, kde jsou uloženy funkce a jednotlivé vstupy pak vytvoří adresu konkrétní buňky. Takže v případě, že všechny vstupy budou log. 0, pak výsledný výstup bude log. 0 a v jakémkoli jiném případě bude výstup tohoto LUT log. 1.



Obrázek 6: 4vstupý LUT [2]

Vyhledávací tabulka je nejčastěji založena na SRAM [2]. Důležité je zmínit, že využití 4vstupého typu není nutnou podmínkou pro realizaci vyhledávací tabulky, lze dosáhnout vyššího počtu vstupů. Ovšem s počtem vstupů roste pořizovací cena a jak si můžeme všimnout na obrázku (viz obr. 7) tak i logická složitost s dobou zpoždění logiky [21].



Obrázek 7: Počet vstupů do LUT a jejich zpoždění [2]

Pro jakoukoli realizaci vyhledávací tabulky je potřeba 2^n počtu SRAM bitů, kde n je počet vstupů [2].

Nejmenší úložnou jednotkou celého FPGA je klopný obvod typu D (D flip-flop), které se používají jako paměťové registry [2]. Někdy také umožňují hladinové řízení.

Konfigurovatelné logické bloky tvoří "jádro", jak je vidět na obrázku (viz obr. 4), který znázorňuje blokové schéma čipu FPGA. Toto "jádro" je obklopeno vstupně-výstupními bloky, které mají za úkol zprostředkování oboustranného toku do FPGA a naopak [2]. Navzájem jsou logické bloky i vstupně-výstupní bloky propojeny programovatelnými propojeními (routing channels) (viz obr. 4), jenž jsou velmi flexibilní a rychlé. Každý I/O pin může být nastaven jako vstupní, výstupní, obousměrný nebo nezapojený. Tyto piny mohou podporovat různé standardy, které vyžadují jiné napájecí napětí (například LVCMOS, LVTTL, HSTL, PCIe, atd.) [28]. Je tedy nutné vždy tyto piny dobře nastavit.

2.2.2 Typy FPGA

Různé firmy užívají odlišných technologií při realizaci FPGA. V této části jsou zmíněny typy FPGA podle použité technologie pro uložení konfigurace obvodu FPGA. Na trhu se pohybuje mnoho firem vyrábějící tyto čipy, mezi hlavní výrobce na poli obchodu s FPGA čipy jsou Xilinx Inc., Intel Corporation, Microchip Technology Inc., Lattice Semiconductor Corporation [2]. Jednotlivé technologie (viz tab. 2) fungují samostatně, ale existují v kombinacích (například obvod založený na SRAM v kombinaci s pamětí flash) [24].

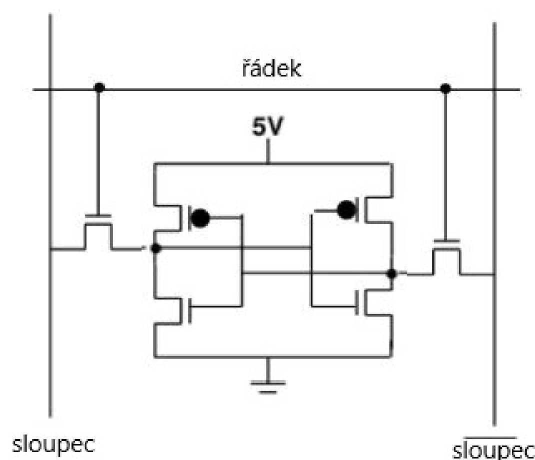
Tabulka 2: Srovnání jednotlivých technologií [2]

	Flash paměť	Antifuse	Statická paměť
Energetická nezávislost	Ano	Ano	Ne
Přeprogramovatelnost	Ano	Ne	Ne
Paměť	Střední	Malá	Velká
Technologie	Flash	CMOS + Antifuse	CMOS
Využitelnost (%)	100	>90	100
Počet přeprogramování	10000	1	Nekonečno

SRAM

Paměť RAM má 2 základní vlastnosti. Zápis a čtení dat probíhá velmi rychle, přičemž nezáleží na místě, kde jsou data uložena, ale po vypnutí se z ní uložená data ztratí. Paměť RAM se dělí na dva hlavní typy: SRAM a DRAM [29]. SRAM je statická a DRAM dynamická. V paměti DRAM se vyskytují kondenzátory [30]. Nežádoucí vlastností u kondenzátorů je elektrický svod, kvůli kterému musí být celý DRAM obnovován s určitou periodicitou, což ho v porovnání se SRAM značně zpomaluje. Paměť SRAM pracuje na principu bistabilního klopného obvodu pro zachování 1 bitu. Buňka je tvořena 4 až 6 tranzistory (viz obr. 8) a je schopna si udržet stav při připojeném napětí. To je důvod, proč nemusí být obnovován jako DRAM. Jsou na křemíkových plátech za sebou ve sloupcích (bit line) a řádcích (word line). Průnik těchto sloupců a řádků pak tvoří jedinečnou adresu paměťové buňky [30].

Výhodou u tohoto typu paměti oproti konkurenčním technologiím je využívání standardního procesu výroby a také přeprogramovatelnost SRAM [30], čím umožňuje libovolněkrát přeprogramovat celý FPGA čip. Na druhou stranu u FPGA čipů založené na této bázi existují i nevýhody. Například oproti ostatním technologiím mají méně paměti na čip, větší rozměry, také jsou energeticky náročnější při běžné činnosti zařízení, horší zabezpečení proti okopírování (čemuž se dá dnes předcházet zašifrováním dané konfigurace), vyšší bitovou chybovost, která je vyšší než u ostatních nebo také skutečnost potenciálního poškození při závadě na napájení [21].



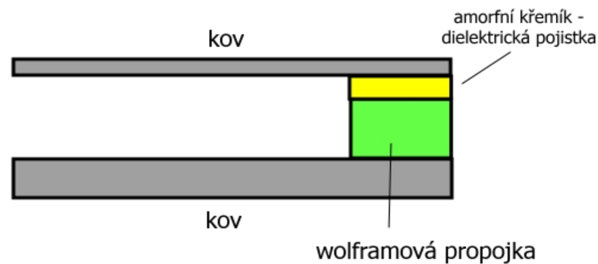
Obrázek 8: Buňka SRAM [30]

Anti-fuse

Antifuse nebo antipojistka je poměrně energeticky nenáročná paměť s velmi dobrým zabezpečením. Tato paměť nevyžaduje externího zařízení pro naprogramování po zapnutí jako tomu je u SRAM [30].

Princip této technologie je již patrný ze samotného názvu. Jednoduše řečeno před naprogramováním jsou mezi 2 kovovými vodiči amorfní křemík (dielektrická antipojistka) a wolframová propojka (viz obr. 9). Pokud projde tímto spojem proud větší než 17 mA, dojde ke změně struktury amorfního křemíku a ten se stane vodivým. To vytvoří trvalý spoj mezi kovy na obou stranách. To ovšem může být výhodou nevýhodou, jelikož musí být naprogramována na počátku a pak již přeprogramovat nejde. Oproti SRAM

využívá složitější výrobní proces a technologie se nevyvíjí tak rychle, což jsou u této paměti stěžejní faktory, proč byla po několik let méně využívána [29, 2].

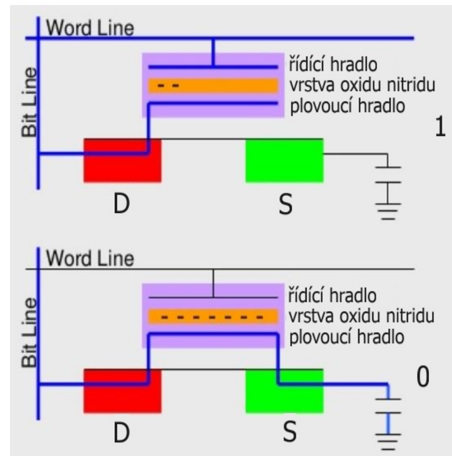


Obrázek 9: Princip Anti-fuse FPGA

Flash/EEPROM

Princip zařízení na bázi paměti EEPROM je přibližně stejný jako u využití paměti flash [2]. Jedná se o energeticky nezávislé paměti, jenž můžeme elektricky mazat a nebo také přepisovat [21]. Jediný markantnější rozdíl je ve způsobu zápisu nebo mazání paměti. U EEPROM probíhá tento proces po bytech a u flash po jednotlivých blocích.

Hlavní jednotkou je unipolární tranzistor MNOS, který mezi řídícím hradlem (control gate) a tělem má vrstvu oxidu nitridu (oxide layer) a plovoucí hradlo (viz obr. 10) [13]. Tranzistor s velkým prahovým napětím je zavřen a s nízkým napětím otevřen. Prahové napětí je ovlivněno elektrickým nábojem, jenž se do plovoucího hradla tuneluje při samotném zápisu.



Obrázek 10: Buňka EEPROM [22]

Můžeme říct, že tento způsob řešení FPGA kombinuje lepší vlastnosti obou předchozích technologií [21]. Využívá standardní proces výroby, nechybí mu možnost přeprogramovatelnosti jako u SRAM a je bezpečnější, energeticky méně náročné ve srovnání se SRAM stejně jako antifuse [2].

FPGA založená na principu flash paměti je momentálně na vzestupu a co se týče vývoje trhu je této technologii přisuzován největší potenciální růst na celosvětovém trhu FPGA v příštích letech [24].

2.3 Intel Cyclone 10CL016

Základní vlastnosti, kterými FPGA čip Cyclone 10CL016 disponuje:

Technologie

- nízkoenergetický FPGA čip s napájením jádra [19]

Základní architektura

- logický blok (LE) - 4vstupý LUT a registr

Vnitřní paměťové bloky

- M9K - 9 kilobitů vložených bloků SRAM [19]
- konfigurovatelné jako RAM, vyrovnávací paměti FIFO nebo ROM [19]

Hodinový signál

- globální hodiny, na kterých běží všechny části desky [19]

Fázové závěsy (PLL)

- až 4 PLL pro všeobecné využití, které poskytují správu a syntézu hodin [19]

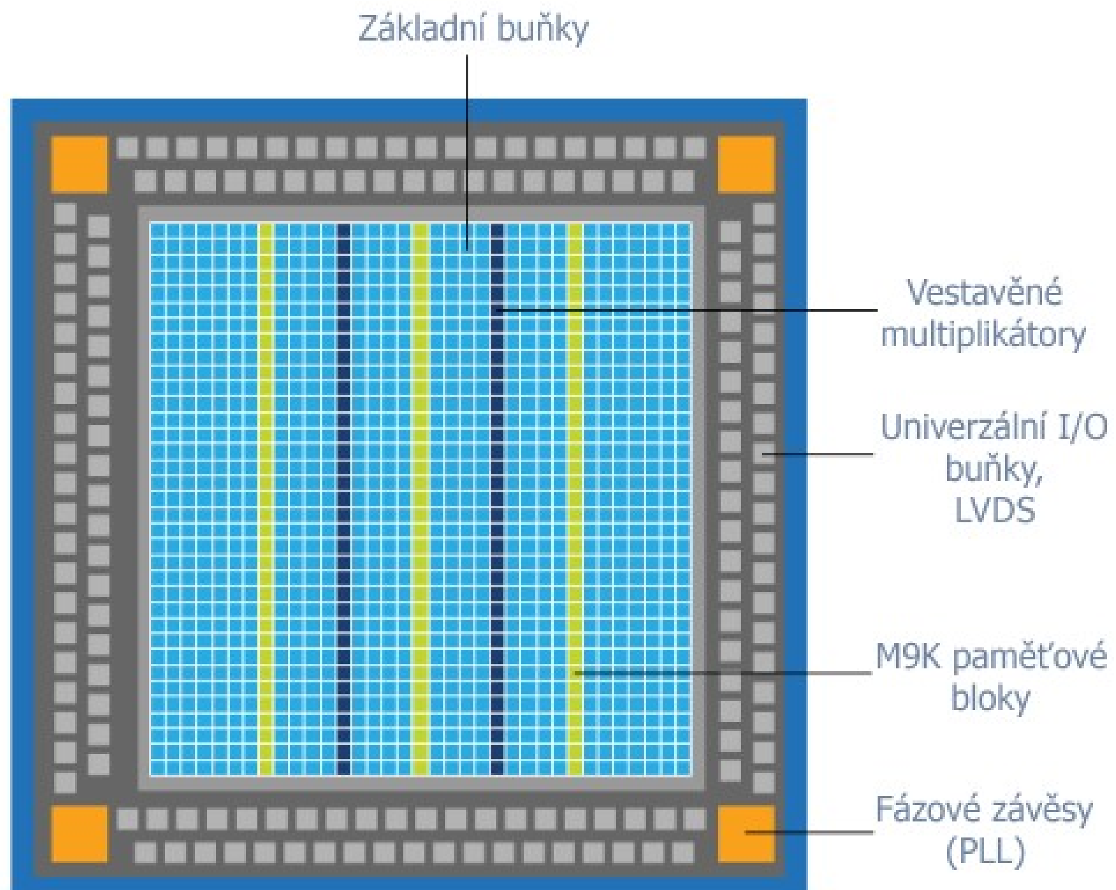
Univerzální I/O piny (GPIO)

- podpora více I/O standardů, pull-up rezistory [19]
- maximální počet pinů je 340

Konfigurace

- konfigurace přes SRAM pro uložení konfiguračních dat

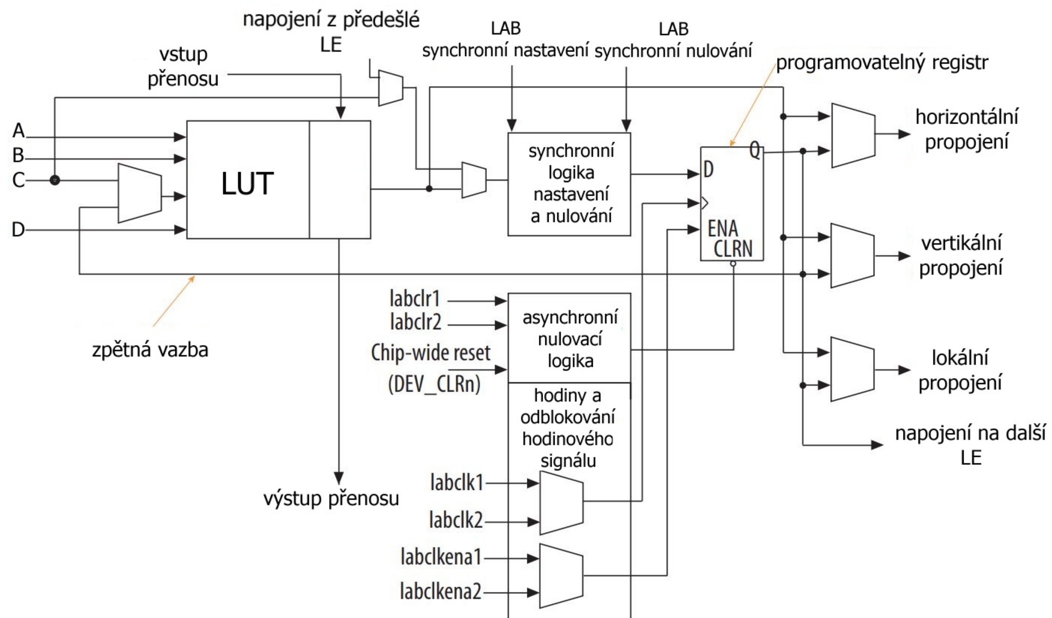
Tento typ čipu FPGA je použit v Arduino MKR VIDOR 4000. Je tam umístěn proto, aby uživatelům umožnil širší využití desek.



Obrázek 11: Blokové schéma čipu Cyclone 10CL016 [18]

Hlavní síla FPGA čipů spočívá v počtu logických členů, které vzájemným skládáním mohou vytvářet složité systémy. Čip Cyclone 10CL016 je založen na 15408 logických buňkách (LE) (viz obr. 12) [19]. Disponuje 56 M9K paměťovými bloky (zabudovaných bloků paměti SRAM) o celkové kapacitě 504 Kb, 56 multiplikátory 18 x 18 bit, maximálním počtem 340 pinů a maximálním počtem 137 diferenčních kanálů. Dále má 4 fázové závěsy (Phase-Locked Loops), které poskytují správu a syntézu hodin. Lze dynamicky měnit fáze nebo frekvence hodin díky rekonfiguraci PLL. Co se týče univerzálních pinů, tak podporuje několik I/O standardů, piny jsou přeprogramovatelné. Dále disponuje nízkonapěťovou diferenciální signalizací LVDS (Low-Voltage Differential Dignaling), která funguje na principu přenosu informací jakožto rozdílu napětí dvou drátů. [19, 18].

Co se týče paměti tak tento čip využívá paměťových buněk desky Arduino MKR VIDOR 4000 založených na slušných 8 MB SRAM, které podporují práci s audiem a videem. Ke konfiguraci FPGA čipu se využívá 2 MB QSPI flash paměť, ze které je polovina paměti vyhrazena pro uživatelské operace [19, 8].



Obrázek 12: Logická buňka Cyclone 10CL016 [19]

Je patrné, že se již nejedná o základní logickou buňku (viz obr. 12) jak byla popsána v kapitole 2.2.2. Lze vidět jednotlivá propojení na další logické buňky (napojení z předešlé logické buňky, napojení do horizontálního směru, vertikálního a lokální napojení). Dále je vidět nastavování a nulování jak synchroním tak v asynchroním režimu. Asynchronní nulovací logika je rozvod významného a často používaného signálu po celé ploše čipu, ať se na jeho rozvod nemusí plýtvat inteligentními buňkami. Například centrální reset (Chip-wide reset), signál hromadného zápisu do klopných obvodů, atd. Zpětná vazba je na vstup LUT zavedena proto, aby v rámci jedné buňky šlo realizovat obecný synchronní stavový automat - tedy kombinační logika a za ní registr. Část výstupních pinů registru může tvořit vstupy do kombinační logiky a část výstupních pinů registru vytváří požadovanou logickou funkci (synchronní). Do této doby sloužily kapitoly jako teoretický základ, kde byl popsán hardware. FPGA čipy se programují právě v jazyce VHDL, kterému se bude věnovat následující kapitola.

3 VHDL

VHDL nebo také VHSIC-HDL je zkratka pro název Very High Speed Integrated Circuit Hardware Description Language [28]. Jak z anglického názvu vyplývá, jedná se o programovací jazyk, kterým popisujeme hardware. Využívá se k návrhu a simulaci programovatelných hradlových polí a logických obvodů.

3.1 Materiály

V dnešní době se objevuje spousta materiálů, pomocí kterých se lze naučit programovat v jazyce VHDL. Je spoustu knih, které si však musí člověk koupit nebo aspoň půjčit z knihovny. Existují však i knihy, které jsou dostupné online. Dle průzkumu fór mezi jednu z nejlepších a volně dostupných jednoznačně patří Circuit Design and Simulation with VHDL, jejímž autorem je prof. Volnei A. Pedroni [1]. Kniha detailně popisuje jazyk VHDL a obsahuje i základní aplikace. Existují i online kurzy. Například se jedná o výukový kurz VHDL od Intelu. Stačí se pouze zaregistrovat na stránkách Intelu. Jde o soubor prezentací s výkladem [20].

Nejlepší literaturou zabývající se VHDL v češtině je kniha Číslicové systémy a jazyk VHDL napsaná autory Jiřím Pinkerem a Martinem Pupou a vydaná nakladatelstvím BEN - technická literatura [26].

3.2 Vlastnosti

Jazyk VHDL má volně dostupný a otevřený standard. Pomocí VHDL lze popsat i velmi složité obvody poměrně jednoduchým způsobem. Tento jazyk klade důraz na funkcionalitu. Velkou výhodou je možnost popisu paralelně probíhajících dějů, kterým se tento jazyk odlišuje od sekvenčně řešených kódů. Umožňuje také opakovaně používat jednotlivé modely pomocí knihoven. Je vhodný pro návrh i pro fázi testování [23].

Jakýkoli název u objektu, návrhové jednotky, proměnných apod. nesmí být žádné z rezervovaných slov (viz tab. 3).

Tabulka 3: Klíčová slova VHDL [23]

abs	access	after	alias	all	and	architecture
array	assert	attribute	begin	block	body	buffer
bus	case	component	configuration	constant	disconnect	downto
else	elsif	end	entity	exit	file	for
function	generate	generic	group	entity	exit	file
for	function	generate	generic	group	guarded	if
impure	in	inertial	inout	is	label	library
linkage	literal	loop	map	mod	nand	new
next	nor	not	null	of	on	open
or	others	out	package	port	postponed	procedure
process	pure	range	record	register	reject	rem
report	return	rol	ror	select	severity	shared
signal	sla	sll	sra	srl	subtype	then
to	transport	type	unaffected	units	until	use
variable	wait	when	while	with	xnor	xor

V jazyce VHDL se nerozlišují malá a velká písmena (například označení typu **std_logic** je to samé jako **STD_LOGIC**). Tento jazyk má striktní typovou kontrolu, pro porovnání dvou odlišných typů je nutnost jeden z nich přetypovat.

V zápisech kódů se v kapitole 3 budou objevovat tmavě modré části textu za dvěmi pomlčkami (–), které symbolizují, že cokoli následujícího je komentář. Zelenou barvou budou zbarveny klíčová slova VHDL. Světle modrou barvou budou názvy knihoven, balíků, entit a architektur. Červenou barvou boudou hodnoty a typy. Názvy proměnných, portů, signálů, atd. budou černou barvou.

3.3 Historie

Historie VHDL se začala psát v roce 1981 a primárně byl tento jazyk vyvíjen na příkaz ministerstva obrany Spojených Států Amerických jako program s názvem VHSIC [1]. Cílem tohoto programu bylo navrhnout jazyk vysoké úrovně s rozsáhlými vyjadřovacími schopnostmi, pomocí kterého bude možno simulovat a navrhovat číslicové obvody nezávisle na metodice návrhu či cílové technologii. Základy jazyka VHDL, na kterých se

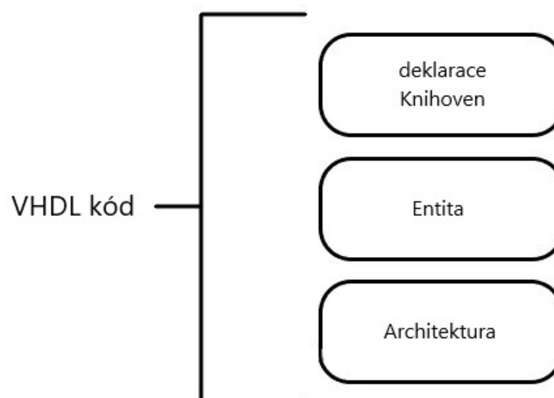
podílely firmy Intermetrics, TI a IBM, vznikly mezi lety 1983 a 1985, kdy byly publikovány. Poté v roce 1986 předalo ministerstvo obrany práva organizaci IEEE a v roce 1987 byl vydán první standard pod hlavičkou této organizace. Do současné doby bylo vydáno několik revizí a rozšíření tohoto standardu [26].

Vývoj VHDL

- 1981 zahájení projektu VHSIC
- 1983-1985 položení základů jazyka VHDL
- 1986 předání práv organizaci IEEE
- 1987 vydání standardu IEEE (VHDL 1076-1987)
- 1994 revize standardu (VHDL 1076-1993)
- 2000 revize standardu (VHDL 1076-2000)
- 2002 revize standardu (VHDL 1076-2002)
- 2008 revize standardu (VHDL 1076-2008)
- 2019 revize standardu (VHDL 1076-2019)

3.4 Knihovny a návrhové jednotky

Knihovny a návrhové jednotky jsou základními částmi VHDL kódu (viz obr. 13).



Obrázek 13: Základní stavba VHDL kódu

Knihovna je skupinou funkcí a procedur, jenž mají definované chování. Určuje, jak se chovají různé funkce, konstanty, datové typy apod. Návrhové jednotky se nacházejí v knihovně. Knihovnu lze připojit pomocí příkazu **library** a poté jednotlivé balíky (package blíže popsán v 3.4.4) pomocí příkazu **use** (viz zdrojový kód 1). Existují dvě třídy knihoven, zdrojové a pracovní (work) [26]. Připojení zdrojových knihoven není jakkoli omezeno, ovšem pracovní knihovnu můžeme mít připojenu právě jednu. Do té se pak ukládají všechny překládané objekty.

Aktivace knihovny a jejich balíků:

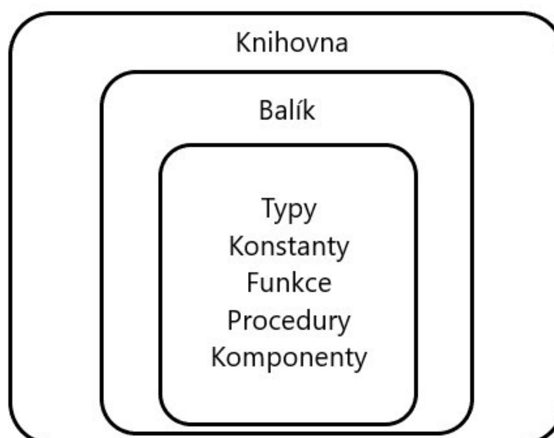
```
library knihovna1;  -- načtení knihovny "knihovna1"

use knihovna1.balik1.all;
  -- načtení konkrétního balíku "balik1" z knihovny

  -- pomocí "all" se načte vše z balíku nebo může být načtena jen část a to
  -- přídáním konkrétního názvu části balíku
```

Zdrojový kód 1: Inicializace knihoven a balíků

Nejčastěji používaná knihovna je standardní knihovna **IEEE**, která obsahuje knihovní balíky standardizované organizací IEEE.



Obrázek 14: Knihovna a její součásti

3.4.1 Typy

Existuje spousta knihovních balíků, ale mezi ty nejvyužívanější patří:

- **std_logic_1164**
- **std_logic_signed**
- **std_logic_unsigned**
- **numeric_std**
- **numeric_bit**
- **math_real**

Balíky obsahují předdefinované typy, které se ve VHDL využívají. V knihovním balíku **std_logic_1164** jsou definovány různé datové typy (**std_logic**, **std_logic_vector**, **std_ulogic**, **std_ulogic_vector**), které jsou VHDL často používány. Typy **std_logic** a **std_ulogic** mohou nabývat různých hodnot (viz tab. 4) a zbylé dva datové typy z tohoto balíčku jsou jednorozměrná pole obsahující tyto hodnoty. Pokud chceme zapsat takové pole je nutné nastavit jeho velikost. Například 4místné pole **std_logic_vector** zapíšeme jako **pole:in std_logic_vector (3 downto 0)**. Slovo **downto** nám určuje sestupné číslování, které se upřednostňuje kvůli uspořádání binárního čísla. Pokud však chceme vzestupné řazení, stačí použít **to**. Důležité je zmínit, že **std_logic_vector** a všechny jiné typy s částí ”vector” v názvu neslouží jako místo pro uložení dat, nýbrž jako jakési spojení. Můžeme si ho tedy představit jako drát v obvodu. Přiřazení hodnot u těchto polí probíhá pomocí uvozovek, kdežto u typů jenž nejsou polemi, probíhá přiřazení hodnoty pomocí apostrofů.

Tabulka 4: Hodnoty typu std_logic a std_ulogic [26]

’U’	Hodnota nebyla inicializována.
’X’	Hodnota je neznámá.
’0’	Hodnota je logická 1.
’1’	Hodnota je logická 0.
’Z’	Vysoká impedance
’W’	Slabý signál s neurčitostí stavu.
’L’	Slabý signál pravděpodobně logická 0.
’H’	Slabý signál pravděpodobně logická 1.
’-’	Hodnota, na které nám nezáleží.

Dalším využívaným balíkem je **numeric_std**. Tento balík obsahuje typy **signed**, **unsigned** a k tomu operátory (viz tab. 5), které jsou nedílnou součástí většiny VHDL programů.

Tabulka 5: Matematické a relační operátory [26]

abs(x)	Absolutní hodnota	=	Operace ”rovná se”.
mod	Operace modulo.	>	Operace ”je větší”.
rem	Hodnota je logická 1.	<	Operace ”je menší”.
+	Operace sčítání.	<=	Operace ”je menší nebo rovno”.
−	Operace odčítání.	=>	Operace ”je větší nebo rovno”.
*	Operace násobení	/ =	Operace ”je různé”.
/	Operace dělení.		

V standardní knihovně, jsou předdefinovány další typy:

- **bit:** Nabývá pouze hodnoty ’0’ a ’1’.
- **bit_vector (délka):** Jedná se o bitový vektor nebo můžeme říci jednorozměrné pole bitů.
- **boolean:** Tento typ nabývá hodnoty ’true’ a ’false’.

- **character:** Nabývá jakékoli hodnoty z ASCII tabulky.
- **string:** Jedná se o pole nabývající hodnot z ASCII tabulky.
- **integer range:** Jedná se o jakékoli číslo v rámci rozsahu, který se v tomto případě nepíše do závorek. Například ho můžeme zapsat jako **cislo1:in integer range 0 to 30;**. Maximální rozsah tohoto typu je od -2147483647 až do 2147483647 [26].
- **natural range:** Jedná se o přirozené číslo, jenž se deklaruje stejným způsobem jako integer, ale jeho rozsah je od 0 do 2147483647 [26].
- **positive range:** Jedná se o kladná čísla deklarovaná stejným způsobem jako předchozí natural a jeho rozsah je od 1 do 2147483647 [26].
- **real range:** Jedná se o reálná čísla, deklarace opět probíhá stejným způsobem a jeho rozsah je od $-1 * 10^{308}$ do $1 * 10^{308}$ [26].

Může být vytvořen také vlastní typ, například typ barva **type barva is (cervena, zelena, zluta)**, který bude maximálně nabývat tří hodnot (červená, zelená, žlutá).

3.4.2 Entita

Existuje pět návrhových jednotek a dělí se také na primární a sekundární [23]. Pokud je návrhová jednotka sekundární, váže se na primární a definuje její chování (viz zdrojový kód 2). Entita je jednou z hlavních návrhových jednotek, určuje vstupy a výstupy, jejich typ a směr toku dat (viz obr. 15).

Blok entity:

```
entity entita1 is -- deklarace bloku entity entita1

  generic (genericka_konstanta1: typ1 |:= vyraz|;);
    -- deklarace generické konstanty, kde typ může být integer,natural apod. a
    -- hodnota či výraz nemusí být deklarovány

  port ( port1 : in/out/inout/buffer) typ1 |:= vyraz|;);
    -- deklarace jednotlivých portů

end entita1; -- ukončení entity
```

Zdrojový kód 2: Deklarace entity

Část generic deklaruje konstanty, které mohou být použity k ovládání struktury nebo chování celé entity. Nepředstavuje však žádný signál a používá se jako parametr pro zlepšení čitelnosti. Mohou se využít například k určení bitové šířky dat nebo operandů [26]. Port je užíván k definici vstupů a výstupů (vnější signály entity). Definuje se zde název portu, způsob jakým se bude chovat a typ.

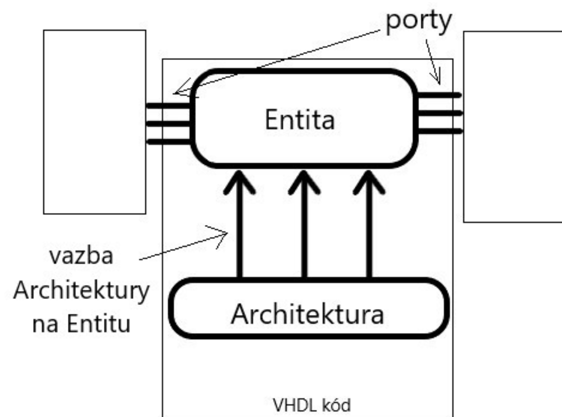
Chování portů:

- **in** : může být pouze čten a lze použít pouze jako vstup (například nulování, hodiny, atd.)
- **out** : může být přiřazena pouze hodnota a lze použít pouze jako výstup (například datové výstupy)
- **inout** : může mít vlastnosti jako oba předchozí typy (například obousměrné datové sběrnice)

- **buffer** : má prakticky stejný význam jako out, nicméně entita může výstupní data zpětně číst (například výstupy čítačů)

3.4.3 Architektura

Architektura je sekundární návrhová jednotka, která je vázána na primární jednotku (viz obr. 15). Jejím účelem je popsat chování entity.



Obrázek 15: Návaznost entity a architektury

Skládá se z deklarační a příkazové části (dále nazýváno jako tělo architektury). Deklarační část obsahuje definice signálů, konstant, různých komponentů nebo podprogramů. Příkazová část je ohraničena **begin** a **end** a zde jsou příkazy, jak má entita fungovat (viz zdrojový kód 3). Návaznost na jednu entitu může mít více architektur (pokud máme definovanou entitu, kde jsou 2 vstupy a 1 výstup, můžeme definovat například architekturu pro hradlo AND a pro hradlo OR), opačně toto tvrzení neplatí [26].

Deklarace architektury:

```
architecture architektura1 of entita1 is -- deklarace architektury entity 1
  -- deklarační část
begin -- tělo architektury začíná klíčovým slovem begin
  -- funkcionalita se popisuje v rámci architektury

  -- po begin následují jednotlivé příkazy nebo procesy
end architektura1; -- ukončení architektury pomocí klíčového slova end
```

Zdrojový kód 3: Deklarace architektury

V praxi je VHDL kódování rozděleno do 3 stylů, ovšem je nutné zmínit, že neexistuje žádná podmínka výběru pouze jednoho stylu. Běžný je výskyt všech 3 stylů v jednom kódu.

- **Behaviorální (Behavioral)** - Určuje, jak se má daný systém chovat v čase. Jde tedy o sekvenční programování. Tento styl překladači neposkytuje žádné informace, jak bude design vypadat. Překladač si nejprve musí kód přeložit a až poté vytvoří obvod. To sebou nese nevýhodu, že výsledný systém může být velmi složitý. Využití najde hlavně u simulací [1].
- **Tok dat (Dataflow)** - Jak název napovídá, jedná se o styl, kde popisujeme pohyb jednotlivých dat v systému. Dataflow se skládá z příkazů kudy a kam mají data téct. Již se jedná o nižší úroveň abstrakce, než tomu bylo u behaviorálního stylu [26].
- **Strukturální (Structural)** - Pomocí tohoto stylu lze popsat systém na nižších úrovních, ale i na těch vyšších, jelikož definuje obvody na základě dataflow i behavioral. Definuje strukturální implementaci využitím deklarace komponentů a jejich instancí. Hodí se hlavně na návrh větších projektů [26].

Pouze v architektuře se může nacházet paralelní příkaz **process** (viz zdrojový kód 4). Paralelní znamená, že pokud je v architektuře více procesů spustí se všechny zároveň. Příkazy, jenž jsou obsaženy v těle procesu, se vykonávají sekvenčně. Je tu tedy určitá analogie s jinými programovacími jazyky, jako je jazyk C a další. Procesy se využívají například pro realizaci děliček, multiplexorů, klopných obvodů atd.

Deklarace procesu:

```
process |(citlivostni_seznam)|
  -- proces může mít tzv. citlivostní seznam, který obsahuje porty
  -- nebo signály, při každé změně tohoto signálu se začne proces provádět
begin
  -- sekvenční příkazy se píší v těle procesu
  -- tělo procesu začíná klíčovým slovem begin

end process; -- proces musí být ukončen klíčovým slovem end
```

Zdrojový kód 4: Deklarace procesu

3.4.4 Ostatní návrhové jednotky

Dalšími návrhovými jednotkami jsou **Configuration** (konfigurace), **Package** (balík) a **Package body** (tělo balíku).

Configuration - Jelikož uživatel chce využít různé typy zapojení obvodu, má pro jednu entitu více architektur. Konfigurace umožňuje rychlou změnu těchto zapojení. Nicméně tato jednotka není povinná [26].

Package - Jedná se o primární návrhovou jednotku. Hlavním účelem balíků, je obsahovat deklarace datových typů, hlaviček funkcí, procedur nebo například konstant a signálů, aby je mohly jednotlivé návrhové jednotky spolu jakkoli sdílet a využívat je. Stanou se tedy globálními. [26].

Package body - Tato jednotka je sekundární k balíku, ve kterém jsou často jen deklarována těla hlaviček funkcí a procedur nebo konstanty, které jsou v obsahu balíku následně definovány detailně (viz zdrojový kód 5). Smysl oddělení balíku a těla balíku je stejný jako u entity a architektury. Znamená to, že tělo balíku popisuje chování balíku. [26].

```
package balik1 is
  -- deklarace balíku balik1

  -- zde může být například deklarována konstanta
end balik1;

package body balik1 is
  -- deklarační část těla balíku balik1

  -- zde se definuje daná hodnota kontanty z balíku
end balik1;
```

Zdrojový kód 5: Deklarace balíku

3.5 Základní datové objekty

Ve VHDL existují datové objekty **konstanty**, **proměnné** a **signály**. Jejich definice musí být v místech určených pro jejich deklaraci.

3.5.1 Konstanty

Tento datový objekt je identifikátor obsahující nějakou konstantu (viz zdrojový kód 6).

```
constant konstanta1 : typ1 := vyraz;  -- deklarace konstanty
  -- deklarace konstanty konstanta1 typu typ1 a vyraz je hodnota dané konstanty
```

Zdrojový kód 6: Deklarace konstanty

3.5.2 Proměnné

Jedná se o identifikátor, který obsahuje nějakou hodnotu (viz zdrojový kód 7). Může se ovšem měnit za chodu programu, tudíž se využívá k držení dočasné hodnoty v procesu.

```
|shared| variable promenna1 : typ |:= vyraz|; -- deklarace proměnné
  -- pokud je použito "shared", pak tuto proměnnou můžeme definovat
  -- pouze mimo proces a v opačném případě pouze v procesu
  -- deklarace proměnné promenna1 typu typ1 a vyraz může být nějaká hodnota
```

Zdrojový kód 7: Deklarace proměnné

3.5.3 Signály

Signály jsou ve VHDL využívány jako komunikace mezi jednotlivými komponenty programu. Slouží k uchování hodnot minulých a budoucích. Její definici lze vidět v kódu (viz zdrojový kód 6).

```
signal signal1 : typ1 |:= vyraz|;
  -- deklarace signálu signal1 typu typ1
```

Zdrojový kód 8: Deklarace signálu

3.6 Podmínky

3.6.1 Case

Sekvenční podmínka **case** se používá pouze v procesu, kde v citlivostním seznamu je nějaký vstupní signál. Vezme si daný vstupní signál a podívá se na každou podmínku (viz zdrojový kód 9), aby našel tu, kterou vstupní signál splňuje. Používá se pro kontrolu vstupu při mnoha kombinacích (například v implementaci multiplexoru).

```
process (vstup) -- proces se vstupem v citlivostním seznamu
  case vstup is
    when '0' => vystup <= a;
      -- pokud je vstupní signál '0' hodnota a se přiřadí na výstup
    when '1' => vystup <= b;
      -- pokud je vstupní signál '1' hodnota b se přiřadí na výstup
    when others => x <= 'Z';
      -- pokud bude vstupní signál nabývat jiné hodnoty než '1' nebo '0'
      -- , přiřadí se na výstup vysoká impedance, to se
      -- používá jako ošetření ostatních možností
  end case; -- ukončení case klíčovým slovem end
end process;
```

Zdrojový kód 9: Deklarace podmínky case

3.6.2 If

Podmínka **if** (viz zdrojový kód 10) se stejně jako podmínka **case** používá pouze v paralelním příkazu proces. Jedná se o sekvenční podmínku.

```
process (vstup)
  if vstup = '0' then
    vystup <= a;
      -- pokud je vstupní signál log. 0 pak hodnotu a přiřadí na výstup
  elsif vstup = '1' then
    -- pomocí elsif vytváříme další podmínky
    vystup <= b;
  else
    vystup <= 'Z'; -- ošetření výjimek stejně jako u case
end process;
```

Zdrojový kód 10: Deklarace podmínky if

3.6.3 With-select

Podmínka **with-select** se používá k přiřazení signálů (viz zdrojový kód 11). Oproti **case** a **if** se nemusí nacházet v procesu. Použití může být například při zobrazování čísel na 7segmentovém displeji, kdy každému číslu odpovídá určitá kombinace segmentů.

```
with vstup select
  -- pokud je vstupní signál log. 0, pak se přiřadí na výstup "00"
  -- pokud log. 1, pak se přiřadí "01"
  vystup <= "00" when '0',
            "01" when '1',
            "11" when others; -- ošetření vyjimek
```

Zdrojový kód 11: Deklarace podmínky with-select

3.6.4 When-else

Má stejný význam použití jako **with-select** (viz zdrojový kód 12).

```
vystup <=
  -- pokud je vstupní signál log. 0, pak se přiřadí na výstup "00"
  -- pokud log. 1, pak se přiřadí "01"
    "00" when vstup = '0' else
    "01" when vstup = '1' else
    "11" when others;
```

Zdrojový kód 12: Deklarace podmínky when-else

3.7 Podprogramy

Podprogramy ve VHDL jsou procedury a funkce. Mohou být volány opakovaně z různých částí kódu. Kódy uvnitř těchto podprogramů se vykonávají sekvečně [28].

3.7.1 Funkce

Funkce vrací vždy jen jednu návratovou hodnotu (viz zdrojový kód 13). Funkce může mít v citlivostním seznamu konstanty, proměnné nebo signály. Musí tam však být všechny, které bude funkce používat. Funkce jsou převážně krátké části kódu, sloužící k zpřehlednění kódu.

```
function funkce1 |(citlivostni_seznam)| return typ1 is
  -- deklarace funkce procedura1, která může mít citlivostní seznam a
  -- návratová hodnota je typu typ1
begin -- tělo funkce začíná klíčovým slovem begin
  -- zde jsou jednotlivé příkazy
  return vystup; -- návratová hodnota vystup
end funkce1;
```

Zdrojový kód 13: Deklarace funkce

3.7.2 Procedury

Procedury jsou stejně jako funkce malé části kódu (viz zdrojový kód 14). Mohou přijímat vstupy a generovat výstupy. Mohou být komplikovanější než funkce [28].

```
procedure procedura1 |(citlivostni_seznam)| is
  -- deklarace procedury procedura1, která může mít citlivostní seznam
begin -- tělo procedury začíná klíčovým slovem begin
  -- zde jsou jednotlivé příkazy
end procedura1;
```

Zdrojový kód 14: Deklarace procedury

3.8 Programy

3.8.1 Realizace základních hradel

Pomocí logických hradel vytváříme kombinační logiku. Na následujícím obrázku č. 15 je vidět realizaci hradel AND a OR v jazyce VHDL.

```
library ieee;
use ieee.std_logic_1164.all;

entity gates_and_or is
  port (
    vstup1 : in std_logic; -- první vstup
    vstup2 : in std_logic; -- druhý vstup
    vystup_AND : out std_logic; -- výstup logické funkce and
    vystup_OR : out std_logic -- výstup logické funkce or
  );
end gates_and_or;

architecture dataflow of gates_and_or is

begin
  vystup_AND <= vstup1 and vstup2;
    -- hodnota logické funkce AND vstupů vstup1 a vstup2 se uloží
    -- na výstup vystup_AND
  vystup_OR <= vstup1 or vstup2;
    -- hodnota logické funkce OR vstupů vstup1 a vstup2 se uloží
    -- na výstup vystup_OR
end dataflow;
```

Zdrojový kód 15: Realizace hradel AND a OR

Realizace ostatních základních hradel není nijak složitá (viz tab. 6). Provádí se analogicky jako u hradel AND a OR.

Tabulka 6: Ostatní hradla ve VHDL [28]

Logická operace	Zápis operace ve VHDL
NOT	**vystup <= not(vstup);**
NOR	**vystup <= vstup_1 nor vstup_2;**
NAND	**vystup <= vstup_1 nand vstup_2;**
XOR	**vystup <= vstup_1 xor vstup_2;**
XNOR	**vystup <= vstup_1 xnor vstup_2;**

3.8.2 Sekvenční a kombinační obvody

V této podkapitole jsou realizovány vybrané základní sekvenční a kombinační obvody. Tyto obvody jsou hojně užívány jako jednotlivé části většiny programů v jazyce VHDL.

Multiplexor Jedná se o datový selektor jenž přepíná mezi vstupy přivedených signálů (viz zdrojový kód 16), podle vstupů adres. Hodnota z vybraného vstupu je přivedena na výstup. Vnitřně je multiplexor tvořen hradly AND, OR a NOT.

Příklad realizace multiplexoru:

```
library ieee;
use ieee.std_logic_1164.all;
entity multiplexor4_1 is
  port (
    a_in : in std_logic; -- 1. vstup do muxu
    b_in : in std_logic; -- 2. vstup do muxu
    c_in : in std_logic; -- 3. vstup do muxu
    d_in : in std_logic; -- 4. vstup do muxu
    sel1 : in std_logic; -- 1. vstupní adresa
    sel2 : in std_logic; -- 2. vstupní adresa
    vystup  : out std_logic -- výstup multiplexoru
  );
end multiplexor4_1;

architecture behavioral of multiplexor4_1 is
  -- určitou kombinací adres je přiveden určitý vstup na výstup
begin
  process(a,b,c,d)
    -- proces je spuštěn na základě svého citlivostního seznamu
  begin
    if(sel_1 = '0' and sel_2 = '0') then
      -- podívá se nejpve na obě vstupní adresy a poté pošle jednu ze
      -- vstupních hodnot na výstup multiplexoru
      vystup <= a_in; -- přiřazení vstupu a_in na vystup
    elsif(sel_1 = '0' and sel_2 = '1') then
      -- analogicky jako předchozí krok
      vystup <= b_in;
    elsif(sel_1 = '1' and sel_2 = '0') then
      vystup <= c_in;
    elsif(sel_1 = '1' and sel_2 = '1') then
      vystup <= d_in;
  end process;
end behavioral;
```

Zdrojový kód 16: Realizace multiplexoru 4:1

Demultiplexor

Demultiplexor funguje na opačnému principu jako multiplexor. Disponuje několika výstupy, jedním vstupem s přivedeným signálem a vstupy adres (viz zdrojový kód 17). Demultiplexor vybere vstup, který přivede na určitý výstup podle adres, které má k dispozici. Vnitřně je demultiplexor tvořen hradly not a and.

Příklad realizace demultiplexoru:

```
entity demultiplexor1_4 is
  port (
    vstup : in std_logic_vector (3 downto 0); -- vstup do multiplexoru
    sel : in std_logic (1 downto 0);
      -- 2bitová vstupní adresa, podle kterého volí odpovídající výstup
    vystup1 : in std_logic_vector (3 downto 0);
      -- jednotlivé 4bitové výstupy multiplexoru
    vystup2 : in std_logic_vector (3 downto 0);
    vystup3 : in std_logic_vector (3 downto 0);
    vystup4 : in std_logic_vector (3 downto 0);
  );
end demultiplexor1_4;

architecture behavioral of demultiplexor1_4 is
begin
  process(a,b,c,d)
    -- proces je spuštěn na základě citlivostního seznamu
  begin
    vstup1 <= vstup when sel = "00" else "0000";
      -- vstup se přiřadí na výstup1, pokud odpovídá adresa selektoru tomuto
      -- výstupu pokud ne, pak se na tento výstup zapíší samé nuly
    vstup2 <= vstup when sel = "01" else "0000";
    vstup3 <= vstup when sel = "10" else "0000";
    vstup4 <= vstup when sel = "11" else "0000";
      -- vstup je na jeden z výstupů přiřazen na základě binární adresy selektoru
  end process;
end behavioral;
```

Zdrojový kód 17: Realizace demultiplexoru 1:4

Dekodér

Princip digitálního logického zařízení dekodéru je založen na překládání z binárního formátu na jiný (viz zdrojový kód 18). Využívá se například pro výběr segmentů u 7segmentového displeje pro zobrazení čísel. Každému číslu odpovídá určitá kombinace segmentů.

Příklad realizace dekodéru:

```
library ieee;
use ieee.std_logic_1164.all;

entity dekoder is
  port (
    vstup : in std_logic_vector(1 downto 0); -- 2bitový vstup
    vystup : out std_logic_vector(3 downto 0) -- 4bitový výstup
  );
end dekoder;

architecture behavioral of dekoder is
begin
  process(vstup)
    -- proces se vstupem v citlivostním seznamu
  begin
    if(vstup = "00") then
      -- podmínka if otestuje vstup a podle toho vybere výstup
      vystup <= "1110"
        -- pokud se vstup rovná "00", pak se hodnota "1110" přiřadí na výstup
    if(vstup = "01") then
      vystup <= "1101"
    if(vstup = "10") then
      vystup <= "1011"
    if(vstup = "11") then
      vystup <= "0111"
    end if;
  end process;
end behavioral;
```

Zdrojový kód 18: Realizace dekodéru

Registr

Registr lze realizovat klopným obvodem. V případech, kdy je požadováno, aby klopný obvod udržel hodnotu po nějaký čas a nereagoval na vstup, využijí se synchronní klopné obvody (viz zdrojový kód 19). Ty disponují hodinovým vstupem, který řídí zápis vstupu na výstup. Na hodinový vstup je přiveden synchronizační či hodinový signál.

Příklad realizace registru:

```
library ieee;
use ieee.std_logic_1164.all;

entity Posuvny_registr is
  port(
    hodiny : in  std_logic; -- hodinový signál
    reset : in  std_logic; -- signál, který vynuluje registr
    enable : in  std_logic; -- povolení funkce registru -> když log. 1,
      -- pak se ukládá vstup na výstup
    vstup : in  std_logic; -- vstupní signál registru
    vystup : out std_logic_vector(7 downto 0) -- 8bitový výstupní signál registru
  );
end Posuvny_registr;

architecture behavioral of Posuvny_registr is
signal cnt : std_logic_vector (7 downto 0);        -- pomocný signál cnt
begin
  posuvny_reg : process (hodiny, reset, enable)
    -- proces pojmenovaný "posuvny_registr", který má v citlivostním
    -- seznamu hodiny, reset a enable
  begin
    if hodiny'event and hodiny = '1' then -- reakce na hodinový signál
      if reset = '1' then -- detekce reset signálu
        cnt <= (others => '0'); -- vynulování pomocného signálu
      elsif enable = '1' then
        -- detekce signálu povolení funkce registru
        cnt <= cnt(6 downto 0) & vstup;
          -- posunutí jednotlivých bitů v pomocném signálu a na pozici (0)
          -- se přiřadí vstup
      end if;
    end if;
  end process;
  vystup <= cnt; -- přiřazení pomocného signálu na výstup
end behavioral;
```

Zdrojový kód 19: Realizace posuvného registru

Sčítačka

Jedná se o kombinační logický obvod. Poloviční sčítačka realizuje sčítání dvou binárních čísel a sčítačka (viz zdrojový kód 20) počítá i s výstupem předchozího řádu.

Příklad realizace sčítačky:

```
library ieee;
use ieee.std_logic_1164.all;

entity scitacka is
  port (
    vstup1 : in std_logic; -- 1. číslo
    vstup2 : in std_logic; -- 2. číslo
    vstup3 : in std_logic; -- 3. číslo (výstup z předchozího řádu)
    suma : out std_logic;
      -- suma 3 vstupů se rovná log. 1, pokud je lichý počet vstupů roven log. 1
    vystup : out std_logic
      -- výstup sčítačky se rovná log. 1, pokud je sudý počet vstupů roven log. 1
  );
end scitacka;

architecture dataflow of scitacka is
signal signal1, signal2, signal3 : std_logic;
  -- definujeme signály pro uložení výstupů
  -- jednotlivých hradel pro jejich opětovné použití
begin
  signal1 <= vstup1 and vstup2;
  signal2 <= vstup1 xor vstup2;
  suma <= vstup3 xor signal1;
  signal3 <= signal1 and vstup3;
  vystup <= signal2 or signal3;
end dataflow;
```

Zdrojový kód 20: Realizace sčítačky

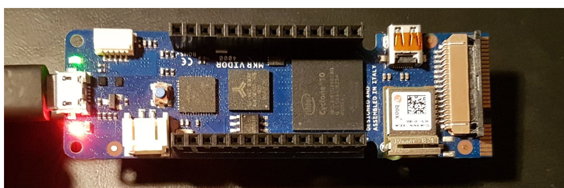
4 Výukové prezentace

Za účelem přiblížení desky Arduino MKR Vidor 4000 a FPGA čipů studentům byla vytvořena skupina šesti prezentací, které by měly studenta seznámit s programováním Arduina v programovacím jazyce Arduino a jazyce VHDL.

4.1 1. prezentace - základní seznámení s Arduinem

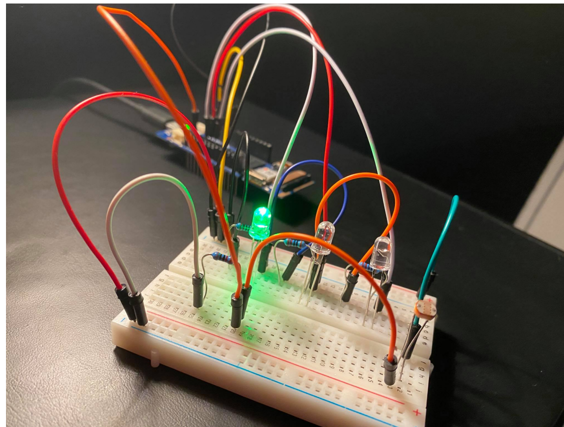
V první prezentaci se nachází detailní pohled na Arduino MKR Vidor 4000. Tato prezentace má desku studentům přiblížit. Poté následuje část, kde je student seznámen s Arduino vývojovým prostředím. Student si vyzkouší manipulaci se softwarem Arduino IDE a také jeho některé nástroje.

Prezentace vede studenty od nejzákladnějších úkonů. Je důležité, aby si vyzkoušeli základní manipulaci s deskou. Prvním vytvořeným programem je rozblikání LED diody (viz obr. 16), jenž je zabudovaná v desce.



Obrázek 16: Rozblikání LED diody

Dále jsou popsány základní dovednosti z jazyka Arduino. student si naprogramuje semafor (viz obr. 17), který se bude programovat i v jazyce VHDL v prezentaci č.3. Tímto si ověří základní rozdíl mezi programováním VHDL a jiného sekvenčně řešeného jazyka jako je jazyk Arduino.



Obrázek 17: Semafor reagující na hodnotu fototranzistoru

Orientační body prezentace:

- Seznámení s Arduinem a jeho zapojením.
- Stáhnutí prostředí Arduino IDE a jeho instalace.
- Instalace potřebných knihoven a vysvětlení jak funguje IDE.
- Založení projektu, vysvětlení stavby kódu programovacího jazyka Arduino a připojení desky k PC.
- Rozblikání zabudované LED (viz obr. 16).
- Vysvětlení sériového monitoru, základních cyklů.
- Rozblikání externí LED diody s určitou frekvencí.
- Vytvoření kódu realizující semafor.

4.2 2. prezentace - základy VHDL

Tato prezentace studenty provede instalací programu Quartus, který je stěžejní pro programování Arduina ve VHDL. Po instalaci Quartu přechází k základnímu popisu jazyka VHDL, vysvětluje základní stavbu kódu, jednotlivé návrhové jednotky, základní typy a objekty. Jakmile si student projde základní informace, založí si projekt, kde implementuje základní hradla. Na této implementaci si vyzkouší simulaci obvodu, která se využívá pro testování.

Následuje část, kde jsou popsány a vysvětleny základní sekvenční a kombinační obvody ve světě VHDL jako jsou dekodér, multiplexor, sčítačka nebo registry.

Na konci prezentace je realizována základní logická buňka VHDL. Díky vlastní implementaci této buňky by měl student pochopit, jak se tato buňka chová a na čem celkově stojí FPGA čipy.

Orientační body prezentace:

- Seznámení se softwarem Quartus, jeho instalace a základní seznámení.
- Vysvětlení základů stavby kódu VHDL.
- Založení projektu.
- Vysvětlení a vytvoření kódu hradel AND a OR, kompilace kódu a simulace napsaného kódu.
- Popis a vytvoření kódů pro základní sekvenční a kombinační obvody.
- Vysvětlení a vytvoření kódu logické buňky a její následná simulace.

4.3 3. prezentace - nahrání kódu

V pořadí 3. prezentace je návod o 6 snímcích, ve kterých je vysvětleno nahrání kódu do desky. Jedná se o složitější postup než jak je tomu u většiny FPGA čipů. Vidor bohužel nelze naprogramovat přímou cestou pomocí softwaru Quartus.

Orientační body prezentace:

- Stáhnutí šablony pro tvorbu VHDL kódu pro Vidor.
- Stáhnutí kompilátoru MSYS2, aby bylo možno kompilovat v příkazovém řádku v operačním systému Windows.
- Překlad výstupního souboru Quartu, vytvořeného kompilací, na binární sekvenci pro Vidor.
- Stáhnutí kódu, který pomocí Arduino IDE nahraje binární sekvenci do desky.

4.4 4. prezentace - realizace semaforu

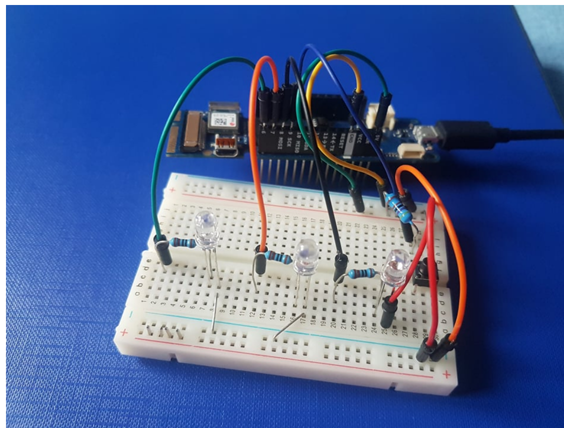
Tato prezentace slouží jako tutoriál k zapojení obvodu a vytvoření kódu pro realizaci semaforu.

Nejprve je popsána funkčnost a zapojení obvodu. Poté se studenti seznámí s tím, jak by měl vypadat blokový návrh semaforu. Následují kroky, které popisují jak jednotlivé bloky vytvořit a jak je vzájemně propojit. Každý krok je popsán tak, aby student věděl, co dělá a hlavně proč to dělá. Programováním bloků si postupně osvojí základy programování VHDL. V praxi naprogramuje například frekvenční děličku nebo dekodér, což jsou ve VHDL velmi používané moduly.

Na konci celé prezentace bude student mít hotov program a obvod realizující semafor (viz obr. 18). Zde vzniká prostor pro porovnání kódu programovacího jazyka Arduino z 4.1 s kódem VHDL.

Orientační body prezentace:

- Popis funkce semaforu a seznam potřebných součástek.
- Vytvoření obvodu pro realizaci semaforu a zapojení k Vidoru.
- Blokový návrh.
- Popis a implementace jednotlivých částí kódu.



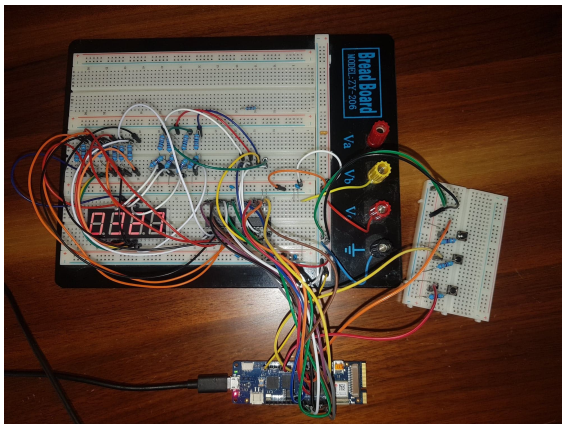
Obrázek 18: Realizace semaforu

4.5 5. prezentace - realizace stopek

Účelem této prezentace je provést studenty zapojením a naprogramováním stopek. Nejprve se studenti seznámí se zapojením 7segmentového displeje, na který si zobrazí čísla. Poté již začnou tvořit jednotlivé části kódu. Na konci budou mít vytvořeny stopky (viz obr. 19), které jsou ovládány 3 mikrospínači pro spuštění, zastavení a následné vynulování stopek k opakovanému použití.

Orientační body prezentace:

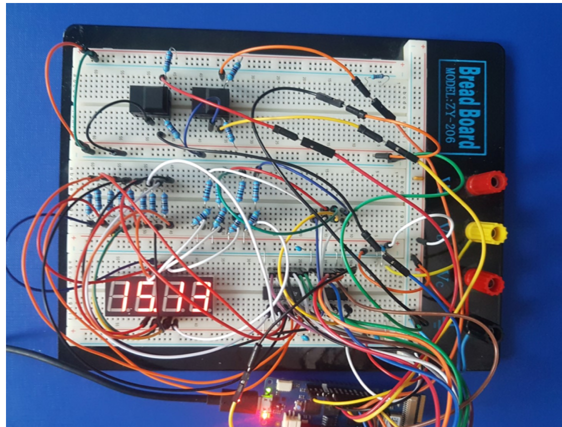
- Popis a implementace kódu pro otestování 7segmentového displeje.
- Zapojení 7segmentového displeje k Vidoru a zobrazení čísel.
- Popis funkce stopek a seznam potřebných součástek.
- Vytvoření obvodu pro realizaci stopek a zapojení k Vidoru.
- Blokový návrh.
- Popis a implementace jednotlivých částí kódu.



Obrázek 19: Realizace stopek

4.6 6. prezentace - realizace hodin

Prezentace č. 6 se věnuje vytvoření hodin. Studenti si vyzkouší složitější aplikace než tomu bylo například u stopek.



Obrázek 20: Realizace hodiny

Orientační body prezentace:

- Popis funkce hodin a seznam potřebných součástek.
- Vytvoření obvodu pro realizaci hodin a zapojení k Vidoru.
- Blokový návrh.
- Popis a implementace jednotlivých částí kódu.

4.7 Struktura prezentací

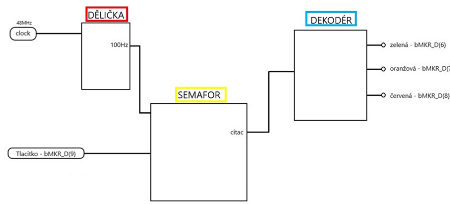
Struktura prezentací je popsána na prezentaci č. 3. V přehledu se nenachází všechny snímky prezentace. Jedná se pouze o stručný přehled. Prezentace s projekty mají až 30 snímků.

Nejprve je funkce semaforu popsána a graficky znázorněna.

Potřebný je seznam jednotlivých součástek s požadovanou dokumentací. Je zde také popsán postup volby rezistoru při zapojení LED diody.

Nejprve si student zapojí potřebný obvod podle schématu. Na snímku je konkrétní zapojení semaforu s tabulkou jednotlivých pinů a obrázkem pinů desky, aby bylo jednoznačně vidět propojení s Vidorem.

Blokový návrh semaforu

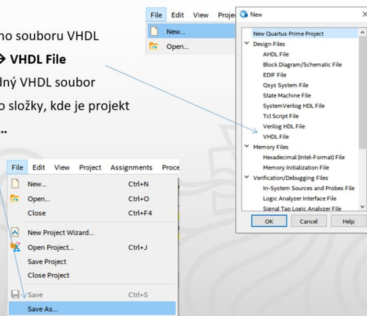


U každého projektu se nachází blokové schéma, které má přiblížit studentovi, jak by měl program vypadat a jak by jednotlivé části spolu měly být propojeny.

Dělička

Postup

- vytvoření nového souboru VHDL
- **File → New... → VHDL File**
- vytvoří se prázdný VHDL soubor
- uložit soubor do složky, kde je projekt
- **File → Save As...**



Napříč prezentacemi je detailně popsána i práce s nástroji jako je Quartus nebo Arduino IDE. Všechny kroky pro úspěšné napsání kódu a jeho nahrání do desky jsou názorně ukázány pomocí jednotlivých výstřižků v prezentaci. Zde je popis, jak přidat VHDL soubor k hlavnímu návrhu.

Dělička

- nahrání knihovny a knihovních balíků

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

- vytvoření entity **Delicka** - bude obsahovat porty –
  - vstupní port **Masterclock** typu **std_logic**
  - výstupní port **clock_100** typu **std_logic**
- **Masterclock** bude obsahovat hodinový signál Vidoru
- **clock_100** bude obsahovat vydělený hodinový signál na 100Hz

```
entity Delicka is
  port(
     Masterclock : in std_logic;
     clock_100 : out std_logic
  );
end Delicka;
```

- vytvoření architektury **behavior** vázané na entitu **Delicka**

```
architecture behavior of Delicka is

begin

end behavior;
```

U každého je detailně popsán zápis. Například je vidět jak nahrát knihovny, entitu, atd. Jedná se o první větší projekt, proto se věnuje úplným základům. V následujících projektech se třeba se znalostí nahrání knihoven již počítá.

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Dělička

- výsledná dělička

```
library ieee;

use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Delicka is
   port(
      Masterclock : in std_logic;    -- hodinový signál Vidoru 48MHz
      clock_100   : out std_logic    -- výstupní hodinový signál 100Hz
   );
end Delicka;

architecture behavior of Delicka is


begin
   process(Masterclock)
      variable MasterClockCounter: unsigned(18 downto 0);
   begin
      if Masterclock'event and Masterclock = '0' then
         if MasterClockCounter = 479999 then       -- testování, zda je čítač roven dané hodnotě
            MasterClockCounter := (others => '0'); -- vynulování čítače
         else
         MasterClockCounter := MasterClockCounter + 1; -- přičtení 1 do čítače
         end if;
      end if;
      clock_100 <= std_logic(MasterClockCounter(18));
               -- jelikož se jedná o unsigned, je nutné ho přetypovat na typ výstupu
   end process;

end behavior;
```

Tento snímek ukazuje celou implementaci děličky pro přehlednost.

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Stavový automat

- testování „aktualni_stav" – pokud bude „cervena" pak se provedou příkazy

```
case aktualni_stav is

   when cervena =>
      if (cnt = 99) then                    -- cnt = 99 odpovídá 1 sekundě
         cnt := (others => '0');            -- vynulování cnt
         aktualni_stav <= cervena_oranzova;   -- přechod na další stav
         citac <= "00";                     -- přiřazení kombinace na výstup
      else
         cnt := cnt + 1;       -- inkrementace čítače cnt
      end if;
```

- pokud se „cnt" nerovná 99 probíhá inkrementace čítače
- v momentě kdy dosáhne 99 uběhla 1 sekunda → „cnt" se vynuluje → přechod na další stav (cervena _oranzova) →na výstup „citac" se dostane kombinace, kterou další komponenta dekodér převede na kombinaci LED diod

V prezentacích jsou použity barevné prvky, pro přiblížení studentovi co právě dělá a s čím to souvisí. Zde je žlutý blok, kde je část stavového automatu a na ní je vysvětlen princip funkce kódu tohoto automatu.

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Přidání komponent do hlavního VHDL souboru

- komponenty se deklarují v architektuře hlavního návrhu před „begin"

```
component název komponenty
    port (
      jednotlivé porty komponenty
    );
end component;
```

- deklarace všech komponent programu

```
component Delicka is
   port(
      Masterclock : in std_logic;
      clock_100 : out std_logic
   );
end component;

component Semafor is
   port(
      clock_100 : in std_logic;
      tlacitko :in std_logic;
      citac : out std_logic_vector(1 downto 0)
   );
end component;

component Dekoder is
   port(
      citac    : in  std_logic_vector (1 downto 0);
      LED   : out std_logic_vector (2 downto 0)
   );
end component;
```

Jednotlivé vytvořené komponenty je potřeba připojit do hlavního souboru VHDL a vytvořit v něm pomocné signály, které fungují jako propojení mezi bloky.

Přiřazení portů a signálů

- přiřazení portů a signálů z hlavního souboru VHDL do jednotlivých komponent

```
pojmenování přiřazení : název komponenty port map (jednotlivé porty nebo signály);
```

- přiřazení portů a signálů musí být v pořadí v jakém jsou definovány v komponentě

```
-- přiřazení iCLK (hodinový signál Vidoru) na port děličky Masterclock
-- přiřazení výstupní hodinový signál děličky clock_100 do signálu clock_100
delicka_1 : Delicka port map (iCLK, clock_100);
```

- stejným způsobem mapovat zbytek komponent

```
-- přiřazení signálu clock_100 na port hodinového signálu semaforu
-- přiřazení portu bMKR_(9) (pin mikrospínače) na port tlačítka v semaforu
-- přiřazení výstupního portu Semaforu citac do signálu citac v hlavním souboru VHDL
semafor_1 : Semafor port map (clock_100, bMKR_D(9), citac);

-- přiřazení signálu citac na port citac v dekodéru
-- přiřazení výstupní kombinace LED na konkrétní piny Vidoru
dekoder_1 : Dekoder port map (citac, bMKR_D(8 downto 6));
```

Nakonec je potřeba přiřadit signály a porty jednotlivým komponentám, což je realizováno na tomto snímku. Poté už následuje pouze kompilace a nahrání kódu (viz. kapitola 4.3).

5 Realizace a vzniklé překážky

Vidor je deska, která měla udělat průlom a přiblížit FPGA jednodušším způsobem možným uživatelům. Firma Arduino měla v plánu vytvořit vývojové prostředí pro programování desky v HDL jazycích, ale k jeho vytvoření nedošlo. Vidor tak nenaplnil očekávání, které měla spousta uživatelů desek Arduino.

Arduino nevytvořila žádný návod či tutoriál, jak naprogramovat Vidor ve VHDL. Existují návody, které Arduino sdílí na svých stránkách [14]. Ovšem ty jsou poměrně nepřehledné a pro začátečníka nevhodné, jak zmiňuje vývojář Arduina Dario Pennisi na fóru [15].

Arduino odkazuje na platformu GitHub [16], kde je vytvořena výchozí šablona pro projekty a pár napsaných projektů v programovacím jazyce Verilog. Dá se říct, že se jedná o jediné materiály poskytnuté Arduinem. Na fórech lze však najít pár užitečných příspěvků jako například přeloženou výchozí šablonu pro projekty [27] do jazyka VHDL. Jednalo se tak o první výraznější krok k vytvoření první aplikace.

Následovala komplikace s nahráním kódu do desky. Jelikož nelze programovat Vidor přímo z Quartu, bylo nutné najít způsob, jak dostat kód do desky. Na již zmíněném GitHubu [16] jsou uvedeny 3 způsoby jak toho docílit, nicméně tyto způsoby u většiny uživatelů vedly pouze k neúspěchu. Proto byl použit trochu jiný způsob [32]. Po úspěšné kompilaci kódu v Quartu ve složce výstupních souborů vygeneruje soubor **.ttf** (Tabular-Text file). Tento soubor se musí převést na binární výstup (bitstream), který pak lze nahrát do desky pomocí kódu v jazyce Arduino. V tuto chvíli již bylo možné začít s projekty.

Závěr

V mé práci jsem se věnoval desce Arduino MKR Vidor 4000, která jako jediná z desek Arduino umožňuje práci s FPGA. První dvě kapitoly se zabývají se teoretickým rozborem. V první se jedná o rozbor platformy Arduino a ve druhé jsou popsány čipy FPGA. Následuje kapitola, kde jsou popsány základy VHDL, které jsou následně použity ve výukových prezentacích. Poslední dvě kapitoly se věnují věcnému popisu vytvořených materiálů a rozboru realizace i vzniklých problémů. Byl vytvořen soubor prezentací, na základě kterých si studenti vyzkouší VHDL na reálných příkladech. Celkově se podařilo vytvořit přehledný manuál, popisující skladbu kódu, jednotlivé základní objekty, podmínky, atd. Od těchto základů se postupně studenti dostanou až ke komplexnější realizaci hodin. Vytvořené prezentace budou použity při výuce studentů bakalářského studia a měly by jim pomoci k pochopení a přiblížení FPGA.

Seznam použitých zkratek

VHDL Very High Speed Integrated Circuit Hardware Description Language

ARM Advanced reduced instruction set computing machines

ASCII American Standard Code for Information Interchange

ASIC Application Specific Integrated Circuit

CMOS Complementary Metal–Oxide–Semiconductor

CPLD Complex programmable logic device

CSI Camera Serial Interface

EPROM Erasable Programmable Read-Only Memory

EEPROM Electrically Erasable Programmable Read-Only Memory

FIFO First In First Out

FPGA Field Programmable Gate Array

F-PLA Field Programmable Logic Array

GAL Genetic Array Logic

GPIO General-Purpose Input/Output

HDL Hardware Description Language

HDMI High-Definition Multimedia Interface

HSTL High-Speed Transceiver Logic

I/O Input/Output

I2C Inter-Integrated Circuit

IDE Integrated Development Environment

IEEE Institute of Electrical and Electronics Engineers

IoT Internet of Things

LAB Logic Array Block

LE Logic Element

LED Light Emitting Diode

LiPo Lithium Polymer

LUT Look-Up Table

LVCMOS Low Voltage Complementary Metal Oxide Semiconductor

LVDS Low Voltage Differential Signaling

TTL Transistor Transistor Logic

LVTTL Low Voltage Transistor Transistor Logic

MIPI Mobile Industry Processor Interface

MNOS Metal Nitride Oxide Semiconductor Transistor

MUX Multiplexor

N-FET N channelField Effect Transistor

P-FET P channel Field effect transistor

MOSFET Metal Oxide Semiconductor Field Effect Transistor

PAL Programmable Array Logic

PC Personal Computer

PCI Peripheral Component Interconnect

PCIe Peripheral Component Interconnect express

PLD Programmable Logic Device

HPLD High-Density Programmable Logic Device

SPLD Simple Programmable Logic Device

PLL Phase Locked Loop

ROM Read Only Memory

PROM Programmable Read Only Memory

PWM Pulse Width Modulation

QSPI Quad Serial Peripheral Interface

RAM Random Access Memory

DRAM Dynamic Random Access Memory

SRAM Static Random Access Memory

SDRAM Synchronous Dynamic Random Access Memory

USB Universal Serial Bus

UV Vltraviolet

A Příloha

Obsah přiloženého CD:

1. Šablona
2. Convertor
   - vidorcvt
   - vidorcvt.c
3. Sketch
   - Sketch.ino
   - jtag.h
   - jtag.c
   - defines
4. Vidor_Hodiny
   - MKRVIDOR4000_top_vhdl.vhd
   - DebouncedSwitch.vhd
   - Counter06.vhd
   - Counter10.vhd
   - Decoder7Segment.vhd
   - mapovani_pinu.qsf
5. Vidor_Stopky
   - MKRVIDOR4000_top_vhdl.vhd
   - Decoder7Segment.vhd
   - mapovani_pinu.qsf

6. Vidor_Semafor
    - MKRVIDOR4000_top_vhdl.vhd
    - Delicka.vhd
    - Dekoder.vhd
    - Semafor.vhd
    - mapovani_pinu.qsf
7. Arduino Vidor 4000 PART1.pptx
8. Arduino Vidor 4000 PART2.pptx
9. Arduino Vidor 4000 PART3.pptx
10. Arduino Vidor 4000 PART4.pptx
11. Arduino Vidor 4000 PART5.pptx
12. Arduino Vidor 4000 PART6.pptx

References

[1] PEDRONI Volnei A. *Circuit Design and Simulation with VHDL*. English. 2010th ed. [Online]. [cit. 2020-9-23]. Cambridge, Massachusetts: The MIT Press, 2017. ISBN: 978-0-262-01433-5. Dostupné z: `http://www.pld.ttu.ee/~alsu/Pedroni_2010_Circuit%5C%20Design%5C%20and%5C%20Simulation%5C%20with%5C%20VHDL.pdf`.

[2] HIDEHARU Amano. *Principles and Structures of FPGAs*. English. 1st. [Online]. [cit. 2020-05-07]. Singapore: Springer, 2018. ISBN: 9789811308246. Dostupné z: `https://link-springer-com.ezproxy.techlib.cz/book/10.1007%5C%2F978-981-13-0824-6`.

[3] Arduino. *Arduino*. English. [Online]. [cit. 2020-03-05]. Dostupné z: `https://www.arduino.cc/en/Main/AboutUs`.

[4] Arduino. *Arduino Intorduction*. English. [Online]. [cit. 2020-03-05]. Dostupné z: `https://www.arduino.cc/en/guide/introduction`.

[5] Arduino. *Arduino Leonardo*. English. [Online]. [cit. 2020-03-05]. Dostupné z: `https://www.arduino.cc/en/Main/Arduino_BoardLeonardo`.

[6] Arduino. *Arduino Mega*. English. [Online]. [cit. 2020-03-05]. Dostupné z: `https://store.arduino.cc/arduino-mega-2560-rev3`.

[7] Arduino. *Arduino Micro*. English. [Online]. [cit. 2020-03-05]. Dostupné z: `https://store.arduino.cc/arduino-micro`.

[8] Arduino. *Arduino MKR Vidor 4000*. English. [Online]. [cit. 2020-03-05]. Dostupné z: `https://store.arduino.cc/arduino-mkr-vidor-4000`.

[9] Arduino. *Arduino Nano*. English. [Online]. [cit. 2020-03-05]. Dostupné z: `https://store.arduino.cc/arduino-nano`.

[10] Arduino. *Arduino Store*. English. [Online]. [cit. 2020-03-10]. Dostupné z: `https://store.arduino.cc`.

[11] Arduino. *Arduino Uno*. English. [Online]. [cit. 2020-03-05]. Dostupné z: `https://store.arduino.cc/arduino-uno-rev3`.

[12] Arduino. *NINA-W10 series (open CPU) Stand-alone multiradio modules*. English. [Online]. [cit. 2020-05-05]. Dostupné z: https://www.u-blox.com/en/product/nina-w10-series-open-cpu.

[13] FROHMAN-BENTCHKOWSKY D. „The metal-nitride-oxide-silicon (MNOS) transistor—Characteristics and applications". *Proceedings of the IEEE* 58.8 (1970). [Online]. [cit. 2020-04-13], 1207–1219 s.

[14] HERTZ Daniel. *How to Program the Arduino MKR Vidor 4000's FPGA with Quartus IDE*. English. [Online]. [cit. 2020-12-15]. Dostupné z: https://maker.pro/arduino/tutorial/how-to-program-the-arduino-mkr-vidor-4000s-fpga-with-intel-quartus-ide.

[15] PENNISI Dario. *How to code and run VHDL examples*. [Online]. [cit. 2020-12-15]. 2018. Dostupné z: https://forum.arduino.cc/index.php?topic=584089.0.

[16] PENNISI Dario. *VidorFPGA*. [Online]. [cit. 2020-12-15]. 2018. Dostupné z: https://github.com/vidor-libraries/VidorFPGA.

[17] HARRIS David M. a HARRIS Sarah L. *Digital Design and Computer Architecture*. English. 1st. [Online]. [cit. 2020-05-05]. San Francisco: Morgan Kaufmann Publishers, 2007. ISBN: 978-0-12-370497-9. Dostupné z: https://ebookcentral.proquest.com/lib/techlib-ebooks/detail.action?docID=404196.

[18] Intel. *Intel Cyclone 10 LP FPGA Features*. English. [Online]. [cit. 2020-06-24]. Dostupné z: https://www.intel.la/content/www/xl/es/products/programmable/fpga/cyclone-10/lp/features.html.

[19] Intel. *Intel Cyclone 10LP Device Overview*. English. Tech. rep. [Online]. [cit. 2020-05-13]. Dostupné z: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-10/c10lp-51001.pdf.

[20] Intel. *VHDL Basics*. English. [Online]. [cit. 2020-09-15]. Dostupné z: https://www.intel.com/content/www/us/en/programmable/customertraining/webex/VHDL/presentation_html5.html.

[21] ŠŤASTNÝ Jakub. *FPGA prakticky: realizace číslicových systémů pro programovatelná hradlová pole*. 1. vyd. Praha: BEN - technická literatura, 2010. ISBN: 978-80-7300-261-9.

[22] WAKERLY John. *Digital Systems Design Textbook*. English. [Online]. [cit. 2020-09-26]. Dostupné z: `https://www.slideshare.net/abhilash128/lec-23`.

[23] AIKEN Pang a MEMBREY Peter. *Beginning FPGA: Programming Metal: Your brain on hardware*. English. 2013th ed. [Online]. [cit. 2020-03-14]. New York: Apress, 2017. ISBN: 978-1-4302-6248-0. DOI: `10.1007/978-1-4302-6248-0`. Dostupné z: `https://link-springer-com.ezproxy.techlib.cz/book/10.1007%5C%2F978-1-4302-6248-0`.

[24] PR Newswire. *Field Programmable Gate Array Market Research Analysis Forecast 2027: Global Field Programmable Gate Array (FPGA) Information, By Types (High-end, mid end, low end), by Technology (Static RAM, Anti Fuse technology, EPROM/EEPROM), by Application (Consumer electronics, IT Telecom, Industrial, Automotive, Defence & Aerospace) - Forecast 2016-2027*. English. [Online]. [cit. 2020-05-08]. July 2016. Dostupné z: `http://ezproxy.techlib.cz/login?url=https://search-proquest-com.ezproxy.techlib.cz/docview/1807609409?accountid=119841`.

[25] WILSON Peter. *Design Recipes for FPGAs*. 1st. [Online]. [cit. 2020-05-05]. Oxford: Elsevier Science & Technology, 2007. ISBN: 978-0-7506-6845-3. Dostupné z: `https://ebookcentral.proquest.com/lib/techlib-ebooks/detail.action?docID=4011801`.

[26] PINKER Jiří a POUPA Martin. *Číslicové systémy a jazyk VHDL*. 1. vyd. Praha: BEN - technická literatura, 2006. ISBN: 80-7300-198-5.

[27] Riscvdev. *How to code and run VHDL examples*. [Online]. [cit. 2020-12-15]. 2018. Dostupné z: `https://forum.arduino.cc/index.php?topic=584089.msg4009287#msg4009287`.

[28] VAIBBHAV Taraate. *PLD Based Design with VHDL: RTL Design, Synthesis and Implementation*. English. [Online]. [cit. 2020-04-26]. Singapore: Springer, 2017. ISBN: 978-981-10-3296-7. DOI: `10.1007/978-981-10-3296-7`. Dostupné

z: https://link-springer-com.ezproxy.techlib.cz/book/10.1007%5C%2F978-981-10-3296-7.

[29] 1-Core Technologies. *FPGA Architectures Overview*. English. [Online]. [cit. 2020-05-05]. Dostupné z: https://www.pdx.edu/nanogroup/sites/www.pdx.edu.nanogroup/files/FPGA-architecture.pdf.

[30] KAMATH D. V. *FPGA programming technology and Interconnect architecture*. English. [Online]. [cit. 2020-09-26]. Dostupné z: https://www.slideshare.net/anishgupta94801/ddhdl-fpga-programmingtechinterconnect.

[31] CHAMOLA Vinay et al. „FPGA for 5G: Re-configurable Hardware for Next Generation Communication“. *IEEE Wireless Communications* (2020). DOI: 10.1109/MWC.001.1900359.

[32] Wd5gnr. *VidorFPGA*. [Online]. [cit. 2020-12-15]. 2018. Dostupné z: https://github.com/wd5gnr/VidorFPGA.

[33] MOORE Andrew a WILSON Ron. *FPGAs For Dummies*. English. 2nd. [Online]. [cit. 2020-03-10]. Hoboken: John Wiley & Sons, Inc., 2017. ISBN: 978-1-119-39049-7. Dostupné z: https://www.intel.com/content/www/us/en/products/programmable/fpga/new-to-fpgas/resource-center/overview.html.

[34] Xilinix. *Xilinx XC2064*. English. [Online]. [cit. 2020-05-05]. Dostupné z: https://forums.xilinx.com/t5/Xcell-Daily-Blog-Archived/Xilinx-XC2064-world-s-first-commercial-FPGA-inducted-into-IEEE-s/ba-p/775806.

Seznam obrázků

1 Arduino Uno [11] 6
2 Arduino MKR VIDOR 4000 [10] 8
3 Xilinx XC2064 [34] 13
4 Struktura FPGA 14
5 Logická buňka [23] 14
6 4vstupý LUT [2] 15
7 Počet vstupů do LUT a jejich zpoždění [2] 16
8 Buňka SRAM [30] 18
9 Princip Anti-fuse FPGA 19
10 Buňka EEPROM [22] 20
11 Blokové schéma čipu Cyclone 10CL016 [18] 22
12 Logická buňka Cyclone 10CL016 [19] 23
13 Základní stavba VHDL kódu 26
14 Knihovna a její součásti 27
15 Návaznost entity a architektury 32
16 Rozblikání LED diody 45
17 Semafor reagující na hodnotu fototranzistoru 46
18 Realizace semaforu 49
19 Realizace stopek 50
20 Realizace hodiny 51

Seznam tabulek

1 Arduino desky [10] 6
2 Srovnání jednotlivých technologií [2] 17
3 Klíčová slova VHDL [23] 25
4 Hodnoty typu **std_logic** a **std_ulogic** [26] 29
5 Matematické a relační operátory [26] 29
6 Ostatní hradla ve VHDL [28] 39

Seznam kódů

1 Inicializace knihoven a balíků . . . 27
2 Deklarace entity . . . 31
3 Deklarace architektury . . . 32
4 Deklarace procesu . . . 33
5 Deklarace balíku . . . 34
6 Deklarace konstanty . . . 35
7 Deklarace proměnné . . . 35
8 Deklarace signálu . . . 35
9 Deklarace podmínky case . . . 36
10 Deklarace podmínky if . . . 36
11 Deklarace podmínky with-select . . . 37
12 Deklarace podmínky when-else . . . 37
13 Deklarace funkce . . . 38
14 Deklarace procedury . . . 38
15 Realizace hradel AND a OR . . . 39
16 Realizace multiplexoru 4:1 . . . 40
17 Realizace demultiplexoru 1:4 . . . 41
18 Realizace dekodéru . . . 42
19 Realizace posuvného registru . . . 43
20 Realizace sčítačky . . . 44