

MISKOLCI EGYETEM



GÉPÉSZMÉRNÖKI ÉS INFORMATIKAI KAR
AUTOMATIZÁLÁSI ÉS INFOKOMMUNIKÁCIÓS INTÉZET

FPGA-s EEPROM programozó eszköz tervezése

KÉSZÍTETTE:

Szabó Ferenc

KONZULENS:

Konzulens neve

beosztása

Miskolc, 2025.

EREDETISÉGI NYILATKOZAT

Alulírott **Szabó Ferenc**; Neptun kód: *JODV94* a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős, *gépészmérnök* szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy

FPGA-s EEPROM programozó eszköz tervezése

című szakdolgozatom/diplomatervem saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, 2025. május 15.

(Szabó Ferenc)

Tartalomjegyzék

1. Idegen nyelvű összefoglaló	6
2. Bevezetés	7
2.1. Téma választás Háttér	7
2.2. A Project terv bemutatása	8
3. Tartalmi rész	10
3.1. Analízis képek magyarázata és az analízis elrendezés	10
3.2. Technicai Hátter	14
3.2.1. Mi a FPGA?	14
3.2.2. Mi a VHDL?	17
3.2.3. Kommunikációs protokollok	19
3.2.4. I2C	23
3.3. NYÁK tervezés	23
3.3.1. A NYÁK tevező program bemutatása	23
3.3.2. A KiCad ban végzett munka bemutatása	24
3.4. A Kész NYÁK	35
3.5. A Programozó al-entitásai	36
3.5.1. A Int_Osc entitás	36
3.5.2. A write8bit entitás	39
3.5.3. UART_TX entitás	40
3.5.4. A UART_RX entitás	41
3.5.5. A writePage entitás	43

3.5.6. A I2C entitás	46
3.6. A fő (MAIN) entitás	50
3.6.1. A main entity állapotgépe	50
3.7. C++ ban megírt utasítás küldő program	52
3.7.1. A Terminal app program	52
3.7.2. A File app program	53
3.7.3. A programozó beállítása a belső memória segítségével	53
3.7.4. Utasítások és utasítás sor példa és magyarázat.	54

Ábrák jegyzéke

2.1. Projekt terv blokkvázlata	9
3.1. Vizualizált analizátor rögzítés példa	11
3.2. Analízis elrendezés blokkvázlata	12
3.3. Analízis elrendezés breadboardon	13
3.4. A MAchXO2 FPGA családból egy CLB [1]	16
3.5. UART adatkeret	20
3.6. UART blokkvázlat	21
3.7. SPI blokkvázlat	22
3.8. A programozó kapcsolási rajza	24
3.9. Ellenállás Jumper-ek	25
3.10. Ellenállósokon lévő feszültség összefüggése	25
3.11. VCC és VBUS közösítő jumper	26
3.12. Fontos feszültségek és a GND kivezetése	27
3.13. A programozó NYÁK-ja	27
3.14. KiCad limit beállítások	28
3.15. A NYÁK alsó rétege	29
3.16. SPI kimenet	29
3.17. I2C kimenet	30
3.18. A rounded tracks plugin UI-ja	31
3.19. Egy trace a plugin használata előtt	31
3.20. Egy trace a plugin használata után	32

3.21. Példa egy csúnyán tervezett trace-re	32
3.22. Teardrop vias UI	33
3.23. Teardrop plugin használata előtt	33
3.24. Teardrop plugin használata után	34
3.25. A Kész NYÁK	35
3.26. A kész NYÁK összerakva és forrasztva	36
3.27. Elérhető CLK-frekvenciák az FPGA adatlapja alapján [1]	38

Táblázatok jegyzéke

3.1. Állapototkhoz tartozó hex kódok	54
--	----

1. fejezet

Idegen nyelvű összefoglaló

Here comes the summary...

2. fejezet

Bevezetés

2.1. Téma választás Háttér

Szinte naponta hallhatunk újabb és újabb, egyre gyorsabb és hatékonyabb Intel illetve AMD hagyományos desktop processzorokról. Ugyanakkor létezik egy másik típusú integrált áramkör, amelyről kevesebb szó esik, mégis számos területen meghatározó szerepet játszik: ezek az FPGA-k.

Az FPGA-k széles körben alkalmazottak. Az FPGA-k egyik legnagyobb előnye, hogy lehetővé teszik a mérnökök számára saját, célzottan megtervezett logikai műveletek végrehajtását, rendkívül nagy sebességgel. Nem feltétlenül azért gyorsabbak, mert több GHz-es órajellel rendelkeznek, mint egy mai hagyományos CPU, hanem mert az áramkörök működése nagyon specifikusan, az adott feladathoz igazítva valósítható meg. Egy FPGA-ban akár több százezer ugyanolyan logikai blokkot lehet tervezni és megvalósítani. Emiatt kiválóak lehetnek a párhuzamos számításban. Így minden terület, ahol ez fontos lehet, használva vannak, legalább a fejlesztési szakaszban.

Például valós idejű rendszerekben vagy bonyolult jelfeldolgozási feladatokban. A mesterséges intelligenciáknál is fontos az ultragyors párhuzamos adat átvitel vagy feldolgozás. Például I/O feladatoknál. De az FPGA-kat jellemzően AI gyorsítók vagy AI processzorok ként is alkalmazzák. Mesterséges intelligencia betanítási feladatokban az FPGA-alapú gyorsítók gyorsabb teljesítményt tudnak nyújtani, mint a hagyományos

GPU-k. Hasonló előnyök miatt az FPGA technológia kiemelt szerepet kapott a kép- és videófeldolgozás területén is. Számos fogyasztói elektronikai eszközben, például virtuális valóság szemüvegekben FPGA-k végzik a beérkező videójelek feldolgozását és a kijelzőkre történő leképezését.

Az automata járművek fejlesztése során is használtak, hiszen alapvető fontosságú a környezet gyors és pontos érzékelése ezeknél a rendszereknél, ami nagymértékű párhuzamos adatfeldolgozást igényel. Jelentős szerepet töltenek be a hagyományos processzorok és az ASIC-ek fejlesztései során. Gyakran vannak használva a tervezők által új dizájnok validálására, debuggolására, mielőtt megrendelik az első prototípusokat egy félvezetőgyártó vállalatól. Bonyolult digitális rendszerek működése gyorsan és költséghatékonyan tesztelhetők, ha FPGA-kon vannak szimulálva. Mindezek alapján elmondható, hogy az FPGA egy olyan technológia, amely meghatározó szerepet tölt be a különböző iparágakban, és várhatóan a jövőben még inkább.

Tanulmányaim során különösen megfogott ez a terület, ezért választottam szakdolgozatom két fő témájául az FPGA-k és a VHDL hardverleíró nyelv mélyebb megismerését. A projektemben a NYÁK tervezése is kiemelt szerepet kapott. Úgy gondolom, hogy a NYÁK-tervezés mindig is keresett szaktudás marad a villamosmérnöki szakmában, hiszen bármilyen modern beágyazott rendszer létrehozása elképzelhetetlen megfelelő áramköri integráció nélkül.

A jövőben szeretnék beágyazott rendszerek fejlesztésével foglalkozni, hol a hardveres és szoftveres ismeretek egyaránt fontosok. A NYÁK-tervezés tudása ebben alapvető kompetenciának számít.

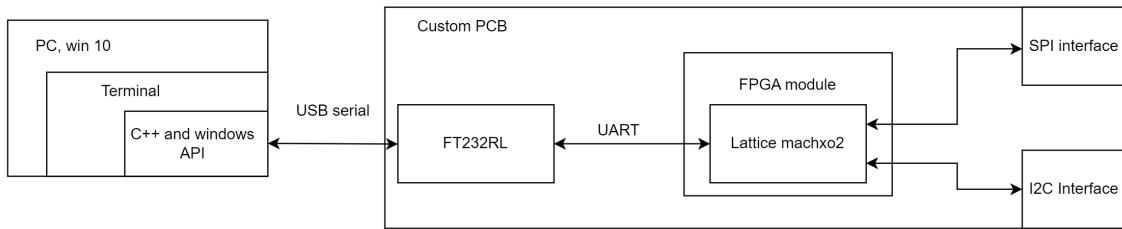
A szakdolgozat témája, miről szól(?), mi a célkitűzés(?), a szakdolgozat felépítése (?) mi az egyes fejezetek rövid tartlama (?)

2.2. A Project terv bemutatása

Projectembe egy összetett, de egyedül megvalósítható beágyazott rendszert akartam megvalósítani. Es használni, illetve próbára szerettem volna tenni a VHDL-ben való tervezési tudásomat. Szerettem volna egy számomra új programozási nyelvel is megis-

merkedni, és végül a C++-t választottam.

Ezek miatt egy EEPROM programozó megvalósítását választottam. A project saját ötlet volt, és úgy gondolom megfelelő szakmai kihívás volt.



2.1. ábra. Projekt terv blokkvázlat

Ez a block diagramm volt a project kiinduló ötlete. A felhasználó a c++-ban írt program fűtatlásával tud kommunikálni a programozóval terminálon keresztül. A programtól kapott USB kommunikációt a FT232RL váltja át UART csomagokká az FPGA module számára. Az FPGA module-ön megvalósított VHDL-ben megírt logikai áramkör pedig a kapott parancsok alapján küld SPI és I2C parancsokat a programozandó EEPROM chipnek. Az EEPROM chip válaszait esetleges válaszait pedig visszakonver-tálja UART csomagokké a FT232RL-nek, amely visszaküldi a PC-nek.

3. fejezet

Tartalmi rész

3.1. Analízis képek magyarázata és az analízis elrendezés

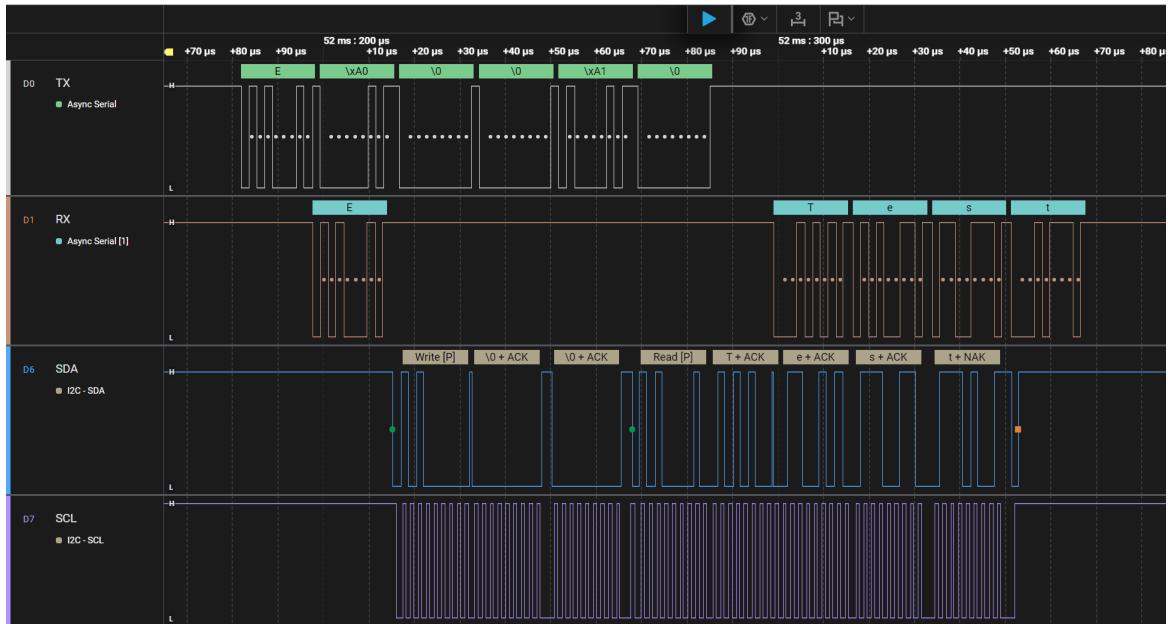
A projekt során egy logikai analizátort sokat használtam. A logikai analizátor egy olyan eszköz, amely lehetővé teszi a digitális jelek időbeli viselkedésének megfigyelését és elemzését. Az analizátor képes rögzíteni és megjeleníteni a digitális jelek állapotát, lehetővé téve a tervezők számára, hogy megértsék a rendszer működését és hibáit. Rendkívül hasznos volt debug-olásra.

A Diplomamunkában sok képet használok, amit egy 24MS/s-es OEM logikai analizátorral rögzítettem. Használtam 2 memória module-t is, egy I2C EEPROM-ot meg egy SPI Flash-t is, hogy tudjam ellenőrizni programozó működését.

A I2C EEPROM cikkszáma: AT24C256. Egy 256Kb-es EEPROM [2]

A SPI Flash cikkszáma W25Q64FV. Egy 64Mb-es Flash [3]

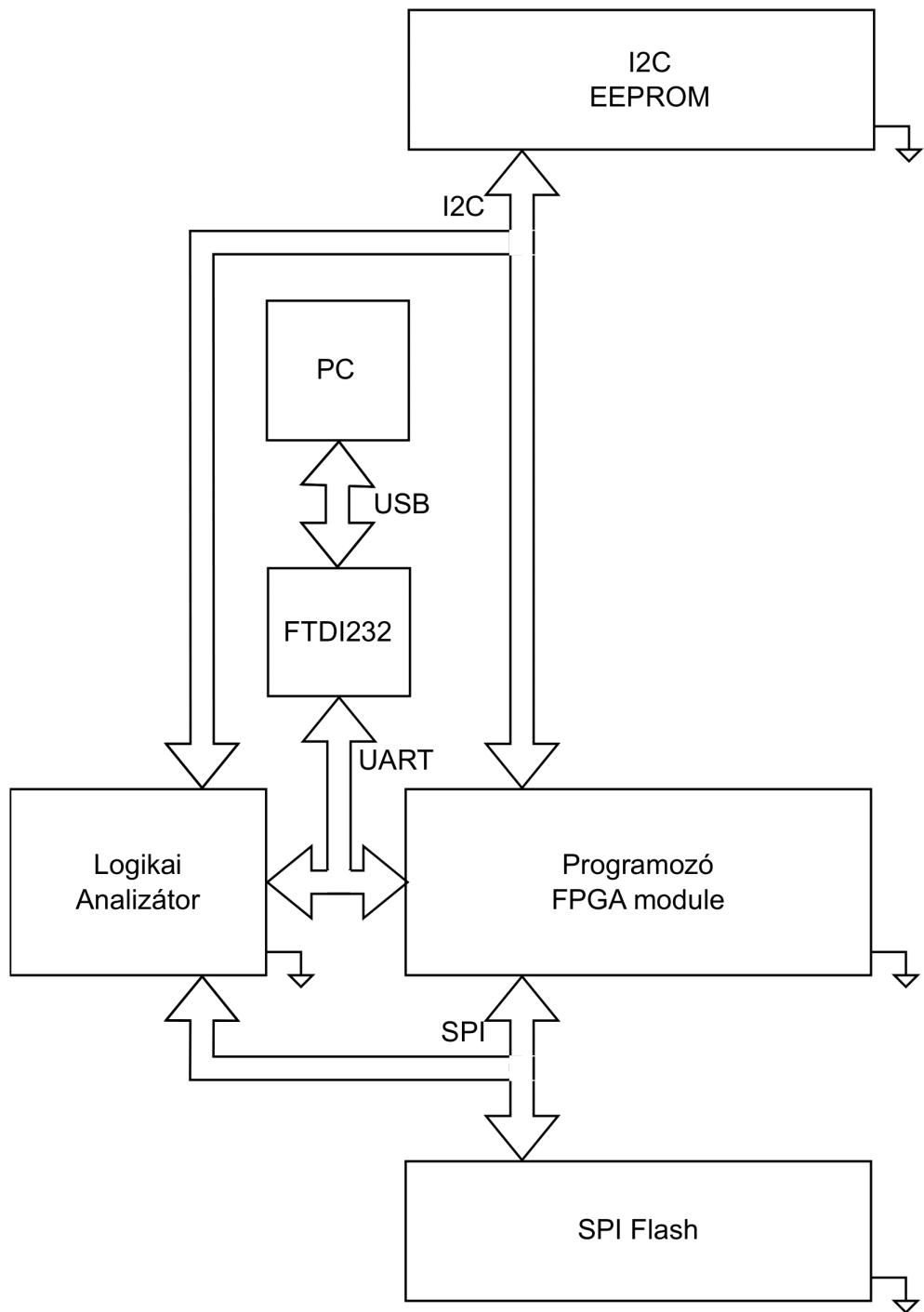
A készített rögzítések Logic nevezetű applikációval vannak vizualizálva és így fognak kinézni:



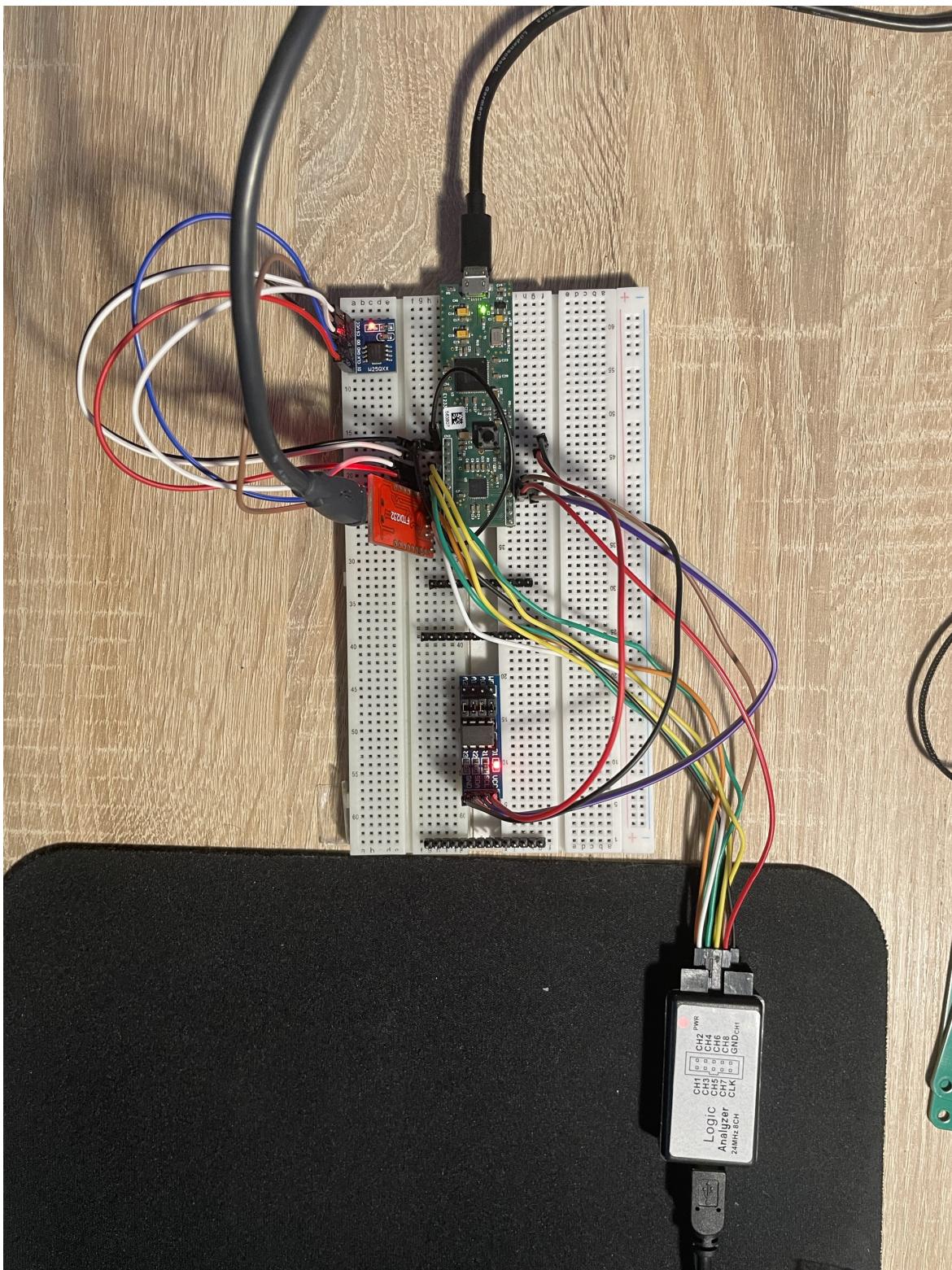
3.1. ábra. Vizualizált analizátor rögzítés példa

A rögzítésekhez a mérési elrendezés breadboard-on történt, mert a NYÁK-on nem tudom egyszerre csatlakoztatni a memória module-okat meg az analizátort.

Az elrendezés:



3.2. ábra. Analízis elrendezés blokkvázlata



3.3. ábra. Analízis elrendezés breadboardon

3.2. Technicai Hátter

3.2.1. Mi a FPGA?

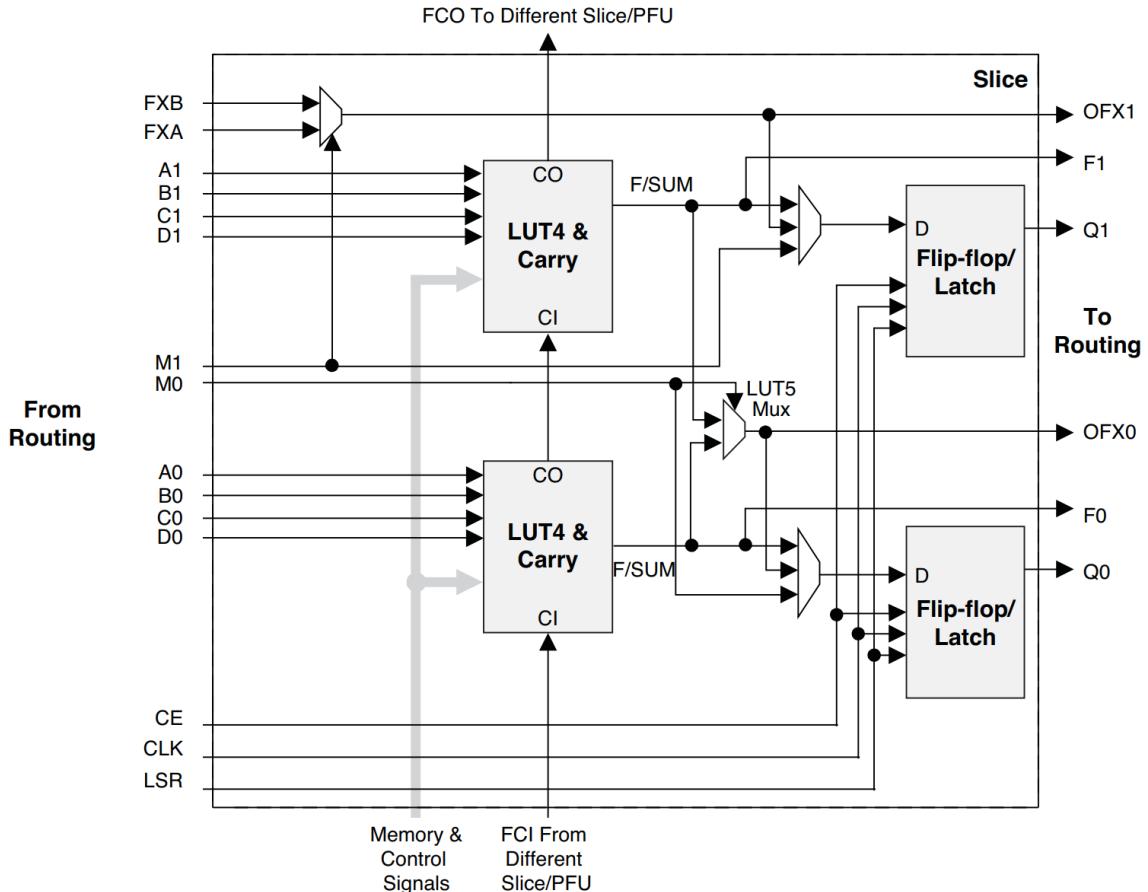
A FPGA egy rövidítés, teljes nevükön ezeket a chipeket, Field-programmable gate array-ként, magyarul a felhasználás helyén programozható logikai kapumátrixokként ismerjük. Az FPGA-ra HDL nyelveken lehet tervezni digitális áramköröket, amik megvalósulnak fizikailag a chip belséjében.

Az FPGA-k tartalmaznak logikai cellákat, különböző tartalmakkal: flip-flop-ok, LUT-ok, illetve logikai kapuk stb. Ezeket a cellákat lehet programozás során tetszőlegesen összeköttetni, így alakulnak ki a tervezett digitális áramkörök. Az FPGA-kon „futó kódot”, azaz a designt, firmware-nek szoktuk nevezni. A firmware szónak más az eredete, mégpedig a csak olvasható memóriába írt program, de az FPGA-k használata körül is elterjedt. A szó maga jól leírja, hogy a FPGA programozható, de közben egy tényleges digitális áramkör valósul meg.

Az FPGA-k megtalálhatók fogyasztói termékekben is, de nagy szerepet játszanak, sok fejlesztési környezetben is, főleg ASIC fejlesztésnél. Termékekben azért lehet hasznos egy FPGA, mert specifikus feladatokat nagyon jól tud elvégezni, és ha a gyártó cég akar változtatni a chip működésén, akkor van rá lehetőség egy firmware frissítéssel. Jó példa az FPGA-k használatára a fogyasztói virtuális valóság készülék. A készülékbe küldött videó adatot fel kell dolgozni, és helyesen rátérképezni a készülék kijelzőre. Mindez minél gyorsabban, hogy a felhasználó minél kevesebb jelkéést érezzen a fejmozgatás és a kép változása között. Ilyen feladatokra tökéletes az FPGA, hiszen ugyanazt a matematikai műveleteket kell elvégezni, ugyanolyan formátumú adatra, több milliószor egymás után. Egy hagyományos CPU ugyan-ebben a feladatban, rosszabbul szerepelne, még ha hasonlóan gyors az órajele is. A CPU-k arra vannak tervezve, hogy minél több használati esetet fedjenek le, és sok feladat elvégzésére könnyen lehessen őket programozni. Az ilyen feladatoknál éppen emiatt, nem a legoptimálisabban végzik el a számításokat, mivel nem erre lettek tervezve. Egy példán keresztül bemutatva: ha egy ember kap egy utasítássort kenyérsütshez, akkor

végre tudja hajtani az utasításokat, és képes egy kenyeret sütni, de lassabban, mint egy kenyérsütéshez tervezett automata gép. Ugyanennek az embernek írhatunk utasításokat egy villanykörté becsavarására is, de a kenyérsütő gép soha nem lesz képes villanykörtét becsavarni. Ehhez egy másik gépet kellene tervezni. A hasonlatban az ember egy CPU, az utasítássor a hagyományos kód és a tervezett automata gép egy FPGA-n megvalósított design.

A legtöbb fejlett FPGA manapság SRAM-alapú. Ezek az eszközök konfigurálható logikai blokkokat (CLB-k), és programozható I/O blokkokat tartalmaznak, a jelek chipbe történő be- és kivezetéséhez, valamint programozható útválasztó mátrixot, a CLB-k összekapcsolásához. Ahogy az elnevezés is sugallja, ezeknek az eszközöknek a konfigurációját, a belső SRAM tárolja a chipen. Mivel a programozható összeköttetések nem ideálisak (ellenállásuk és kapacitásuk van), az egyes útvonalak hossza, nagy hatással van az elkészült design elérhető teljesítményére. Az összekapcsolások okozta késések jelentősek, ezért előfordulhat, hogy nehéz megjósolni a sebességet a tényleges útválasztás befejezése előtt. Ahogy az eszközök kihasználtsága megközelíti a 100%-ot, az útválasztás jelentősen nehezebbé válik. A netlist generáló folyamat, kritikus fontosságú az elkészült design kívánt teljesítményének eléréséhez. Ebben a fejlettebb szintetizátorok nagyban segíthetnek.



3.4. ábra. A MachXO2 FPGA családból egy CLB [1]

A fenti diagram egy egyszerű konfigurálható logikai blokkot / logikai elemet mutat be. Ez a CLB, a MachXO2 FPGA termékcsalád adatlapjában található, ehhez a termékcsaládhoz tartozik az az FPGA is, amivel én dolgoztam. A modern és komolyabb FPGA-k bonyolultabb megvalósításokat tartalmaznak további erőforrásokkal, például blokk RAM-okkal, shift regisz-terekkel, összeadókkal stb. A kis LUT-ok (általában 4-6 bemenet) a kombinációs logika meg-valósítását szolgálják. A logikai blokkok általában tartalmaznak LUT-okat és néhány flip-flop-ot. A CLB-n belüli kapcsolatok meglehetősen rövidek, így a késleltetésük általában jelentéktelen. A CLB-keket általában az FPGA-n belül szigetekként helyezik el, amelyeket az összekap-csolási mátrix vesz körül, ahol a szomszédos CLB-k, általában rövid és gyors kapcsolatot tar-talmaznak közöttük. Ezen kapcsolatok alkalmazásával bonyolultabb funkciók is megvalósíthatók, a szomszédos blokkokból történő erőforrások „kölcsönvételével”. Ez azonban, csak

meg-felelő elhelyezés mellett történhet meg. Például egy összeadó bitjeit, sorrendben egymás mellé kell helyezni. Ez a szintetizátor felelőssége, de általában a tervezőnek ismernie kell a cél FPGA belső architektúráját, az erőforrások legjobb kihasználása érdekében. Manapság eléggyé elter-jedtek a beépített oszcillátorok, tápegységek, melyek a rendszerszintű tervezést segítik, és he-lyet spórolnak. Nem szabad figyelmen kívül hagyni azt a tényt, hogy a megfelelő tápegység megtervezése meglehetősen összetett feladat a modern FPGA-k számára, ahol sok jel, nagy frekvencián működik, és több tápsínre van szükség, nagy áramerősséggel. Az indítási blokkok segíthetnek a terv minden részének megfelelő inicializálásában, rendszerindítás után, vagy alaphelyzetbe állítás esetén.

Az FPGA-kat jelfeldolgozásra is használják, ahol a dedikált szorzók / digitális jelfeldolgozási blokkok hatékonyabbak, mint a CLB-alapú blokkok. Ezek a dedikált blokkok sokkal kevesebb helyet foglalnak el az IC belsőjében, és magasabb frekvencián, kisebb teljesítménnyel tudnak működni. A belső memória flip-flop-okból való szintetizálása, pazarló folyamat. A modern FPGA-k beépített memóriablokkokat tartalmaznak, amelyekre általában szükség van, mivel a legtöbb tervnek valamilyen FIFO-ra, vagy bufferre van szüksége az átmeneti adatok tárolására. A legtöbb FPGA legalább két-portos memóriát tartalmaz, ahol az olvasási és írási műveletek egyszerre történhetnek, két dedikált porton. Az órajelgenerátorok, manapság egy-aránt elterjedt erőforrások.

3.2.2. Mi a VHDL?

A VHDL az egyik legelterjedtebb HDL, a VERILOG mellet. A HDL, azaz hardware description language, magyar neve, hardware leíró nyelv. A VHDL egyben egy standard, és egy programozói nyelv. Programozói nyelv, mert segítségével absztrakt módon lehet leírni egy digitális áramkör működését, amit majd szintetizálás után, meg lehet valósítani egy FPGA segítségével. Egyben egy standard is, amit az IEEE tart fent. minden 5 év után frissül, hogy lépést tartson az ipari szükségekkel. A VHDL nehézsége, mint minden más leíró nyelvnek is, nem csak az, hogy meg kell tanulni a programozó nyelvet, hanem az is, hogy tudni kell, hogy a különböző szerszámok, amiket haszná-

lunk a fejlesztés során, hogyan értelmezik majd a kódot. Amikor egy design működik szimulátorban, nincs rá garancia, hogy szintetizálás után is működni fog. Komolyabb designok tervezése ezeket a lépéseket szokta követni:

- (a) Az elvárt digitális működés leírása HDL nyelven, egyben, majd a szimulációhoz szükséges „testbench” megírása.
- (b) A design szimulálása szimulátorral, debug-olás.
- (c) A design szintetizálása a tényleges a FPGA-ra.

A testbench szerepe az, hogy szimuláció közben, a design bemeneteit szimulálja, illetve kimeneteire válaszoljon, ha kell. Például egy I2C mester tervezése után, a testbench viselkedhet úgy, mint egy szolga. A VHDL tartalmaz szintetizálhatatlan részt is, nagyrészt azért, hogy testbench írásakor egyszerűbben és gyorsabban lehessen haladni. VHDL-nek azt a részét, ami szintetizálható, RTL-nek (register transfer level) hívjuk.

A szintetizálás az utolsó és legkritikusabb lépés, mielőtt egy működő design-t megvalósítunk.

1. A szintetizátor először generál egy kapu szintű netlist-et. Utána a generált netlist-et rátérképezi a specifikus logikai cellákra az FPGA-ban, flip-flop-okra, look-up table-okra és logikai kapukra stb.
2. Ezután készül egy netlist, ami a specifikus FPGA modellre tartozik, a szintetizátor eldönti az összes szükséges cella legjobb elrendezését, összecsatolását.
3. Végül következik az időzítés analízis. Komolyabb designoknál fontos a helyes működéshez, hogy adott értékeknél ne legyen nagyobb a „slack”. Ezen azt értjük, hogy egy logikai jel később ér a netlist-ben megadott helyére, mint kellene, és emiatt hibás a design. Ilyesmi akkor történik, ha például egy jel túl sok logikai kapun megy át anélkül, hogy az órajelhez újra időzítve lenne. Ilyenkor a megoldás az, hogy a logikai kapuk közepénél, újra szinkronizáljuk a jelet egy flipfloppal.
4. Ezután a szintetizátor előállít egy bit fájlt, amit rá lehet programozni az FPGA SRAM-jébe.

A VHDL entitás, egy darab hardware-nek lehet mondani, aminek definiált kimenetei is bemenetei vannak. Egy tervezett entitás lehet komplex, egyszerű, lehet szimpla ÉS kapu vagy akár SPI mester is. Egy entitáson belül lehet több, már megtervezett entitás is összekapcsolva.

Egy entitásnak két része van, amit VHDL-ben meg kell tervezni:

1. Az entitás definíció, ahol meg kell adni az entitás nevét, bemeneteit és kimeneteit
2. A szerkezet definíció, ahol az entitás működését kell leírni.

3.2.3. Kommunikációs protokollok

Ahhoz, hogy adatot tudjunk átadni egy feladótól egy fogadónak, fontos, hogy ugyanazt a „nyelvet” beszéljék. Ha két ember különböző nyelven beszél egymással, akkor nem fogják egymást érteni. Ehhez hasonlóan, ha egy feladó IC, az adatot egy bizonyos protokoll szerint kódolja, de a fogadó nem ismeri a protokollt, akkor nem történik meg az adatátvitel. Tehát egy protokoll szabályai leírják az adat kódolását/dekódolását, és a csatornát is, amin folyik a kommunikáció.

UART

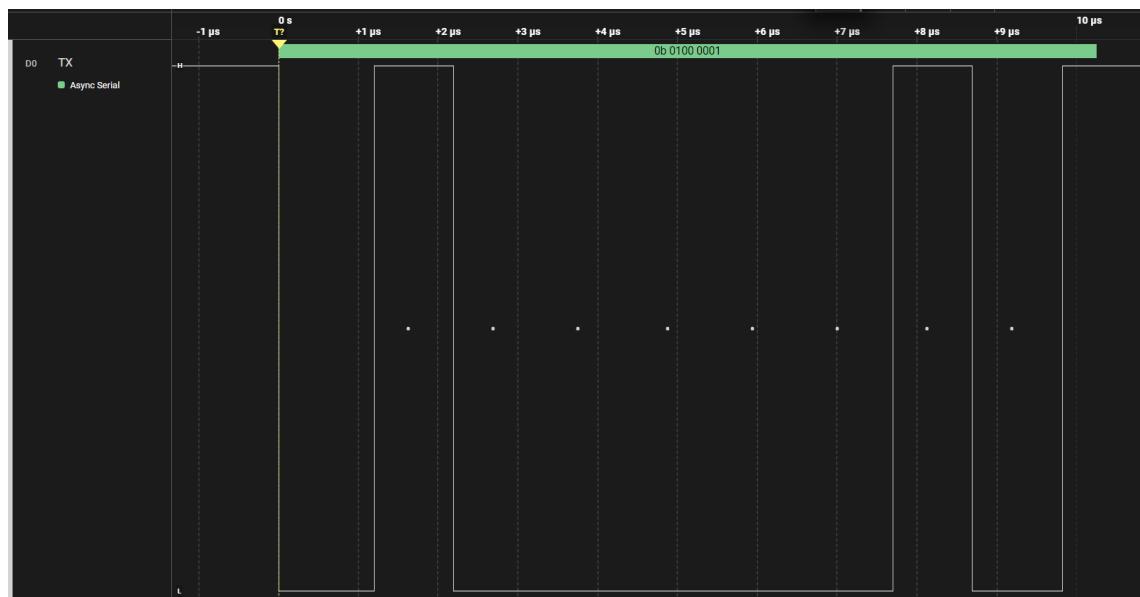
Az általam készített programozó sok kommunikációs protokolلت használ, köztük az UART-ot. A számítógéppel való kommunikálás UART-on keresztül történik, a FT-DI232 UART-USB átalakító segítségével.

Az UART egy rövidítés teljes neve: Universal asynchronous receiver transmitter. Egy aszinkron, soros protokoll, ami azt jelenti, hogy nincs órajel vonal, hanem az időzítést a küldő és fogadó külön-külön követi, és az adatot bitenként kell küldeni/fogadni. Ezért fontos, hogy minden fél ugyanazzal az időzítéssel dolgozzon. Ezt az időzítést szimbólum per szekundumban mérjük, és Baud-nak hívjuk. Ha a kommunikáció időzítése 9600 Baud, akkor a fogadó tudja, hogy egy szimbólum 1/9600 másodpercig lesz megtartva a vonalon, és helyesen tudja mintavételezni az adatkeretben levő adatot. De a Baud ráta nem egyenlő a maximum adatsebességgel, tehát 9600 Baud az nem 9.6

kb/sec. Ez az adatkeret, és egy pár kötelező időzítés miatt van. Az adatkeret azért szükséges, hogy a fogadó tudja mikor kezdődik egy adat, és hol van a vége. Az adatkeret tartalmazhat egy paritás bitet is. Az adatkeret tulajdonságai is egyezni kell az adó és fogadó között.

Tehát a fontos tulajdonságok:

- Baud ráta
- Adatbitek száma
- Paritás bit használata.



3.5. ábra. UART adatkeret

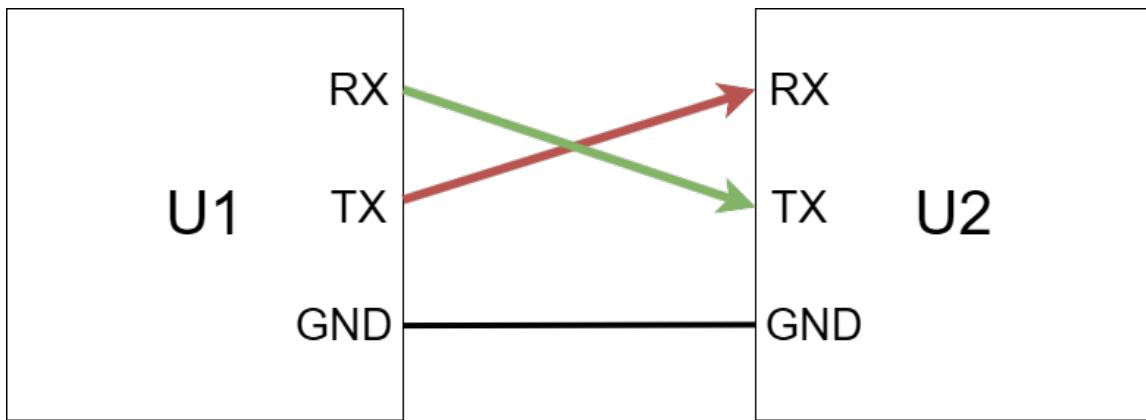
A képen látható egy adatkeret, amit az FPGA küld az FTDI232-nak. A Baud ráta ebben az esetben 921600 Baud, azaz egy szimbólum ideje: $1 / 921600 \text{ Baud} = 1.09 \mu\text{s}$. Ezt az analizátor program segítségével meg is jelöltem a képen.

A tétlen állapotot megbontja az indító bit.

Az indító bit után látható az adat bájt LSB bit sorrenddel, ez azt jelenti, hogy az első bit értéke 2^0 a második 2^1 , stb.

Ezután látható a stop bit, a stop bit 1 db szimbólum ideéjig magas szinten tartja a vonalat. Ha van paritás bit akkor az a stop bit előtt van volna, én viszont nem használok hiba detektálást.

Ezután a vonal visszatér a tétlen állapotba (logikai magas).

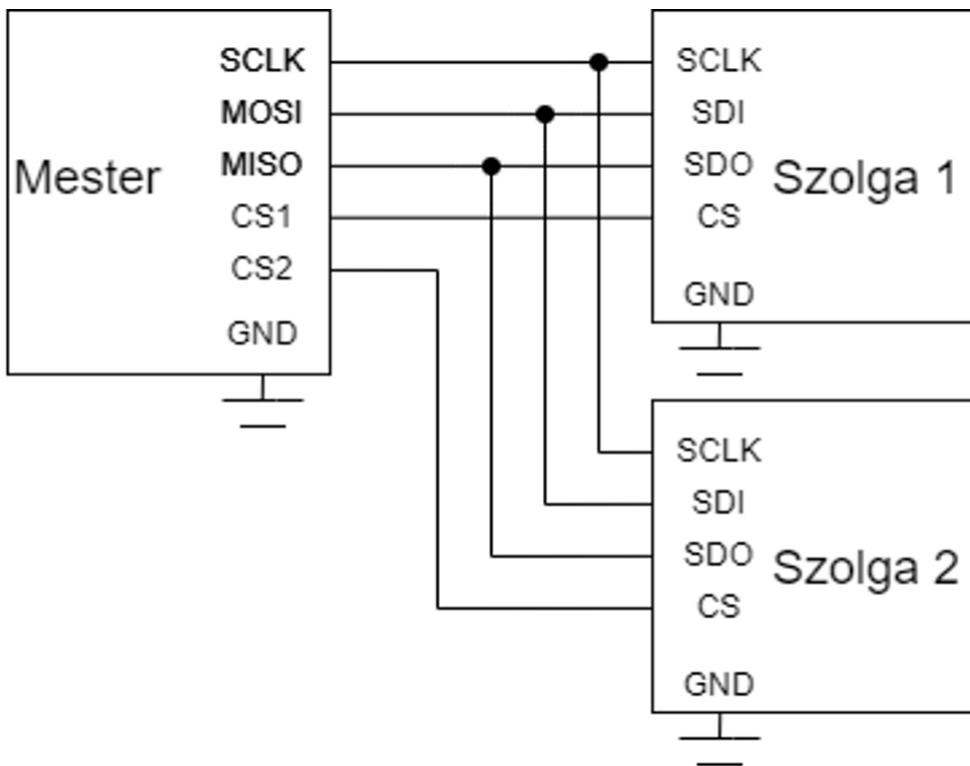


3.6. ábra. UART blokkvázlat

Az UART protokollban a vonalakat RX-nek és TX-nek hívjuk. A küldő és a fogató rendelkezik saját RX és TX lábbal. A TX lab a küldő- és a RX lab a fogadó lab. Nem kötelező az összes vonalakat használni, ezért a kommunikáció lehet szimplex, vagy fél/teljes duplex. Szimplex módban, csak az U1 Tx lába csatlakozna, az U2 RX lábához, és csak küldeni lenne képes az adatot. A fél/teljes duplex konfigurációban, hasonlóan lenne kapcsolva a képhez, és attól függően, hogy az U1/U2 képes-e egyszerre küldeni és olvasni adatot, a kommunikáció fél, illetve teljes duplex lenne. Az én nyákomon a kommunikáció teljes duplex.

SPI

Az SPI kommunikációs protokoll, egy rendkívül elterjedt és rugalmas protokoll, teljes neve serial peripheral interface. Az UART-al ellentétben szinkronizált protokoll, és teljes duplex.



3.7. ábra. SPI blokkvázlat

SPI protokollban több, mint két IC tudja használni ugyanazokat a vonalakat, de egyszerre csak kettő lehet aktív, ezért van mester és szolga. A mester szerepe az, hogy kiválasszon egy szolgát, és kezdje a kommunikációt. A szolga csak akkor képes kommunikálni, ha ki van éppen választva. A kiválasztást a CS „chip select” vonalokon keresztül végzi el a mester. minden szolgának külön CS vonal kell, de a többi vonalat meg tudják osztani. Emiatt minden szolgának elég 4 láb (referencia pontot kivéve), még a mesternek kell 3db, plus minden szolgaként egy láb (ismét referencia pontot kivéve).

Mivel az SPI szinkron protokoll a szimbólum sebességet itt nem az előre beállított Baud ráta határozza meg, hanem a mester előállít egy órajelet a SCLK vonalon, amit figyelnek a szolgák a kommunikálás alatt. A szimbólumsebesség tehát csak a SCLK frekvenciájától függ.

A MOSI vonal egy rövidítés. Teljes neve magyarra fordítva „mester ki szolga be”, ez a vonal hordozza a mestertől küldött soros adatot, a szolga oldalán a lábat SDI-

nak (soros adat be) hívjuk. A MISO vonal mester be szolga ki és a szolga oldalon SDO-nak (soros adat ki) hívjuk. Az adatkeretet bemutatásához használom ismét a prgramozómat, mégpedig egy kommunikációt egy spi FLASH és az FPGA között.

3.2.4. I2C

3.3. NYÁK tervezés

3.3.1. A NYÁK tevező program bemutatása

A KiCad egy ingyenes, nyílt forráskódú szoftvercsomag. Elsősorban kapcsolásai rajzok és nyomtatott áramkörök tervezésére szolgál. Egy KiCad projecten belül lehet tervezni nyákokat, kapcsolási rajzokat, lábnyomokat, és szimbólumokat is.

A KiCad sok hasznos segédprogramot is tartalmaz, amelyek segítenek az kapcsolási rajzok és a nyomtatott áramkörök tervezésében.

Támogatja a kapcsolási rajzok elkészítését, az alkatrészekhez tartozó lábnyomok hozzárendelését, valamint a nyomtatott áramkörök fizikai elrendezésének megtervezését. A PCB tervezés során több réteg kezelésére is lehetőség van, akár 32 rétegű áramkörök tervezése is lehetséges.

A tervezett áramkörök vizuális ellenőrzését egy integrált 3D nézet segíti, amelyben a felhasználó valósághű képet kaphat a végső NYAK-ról.

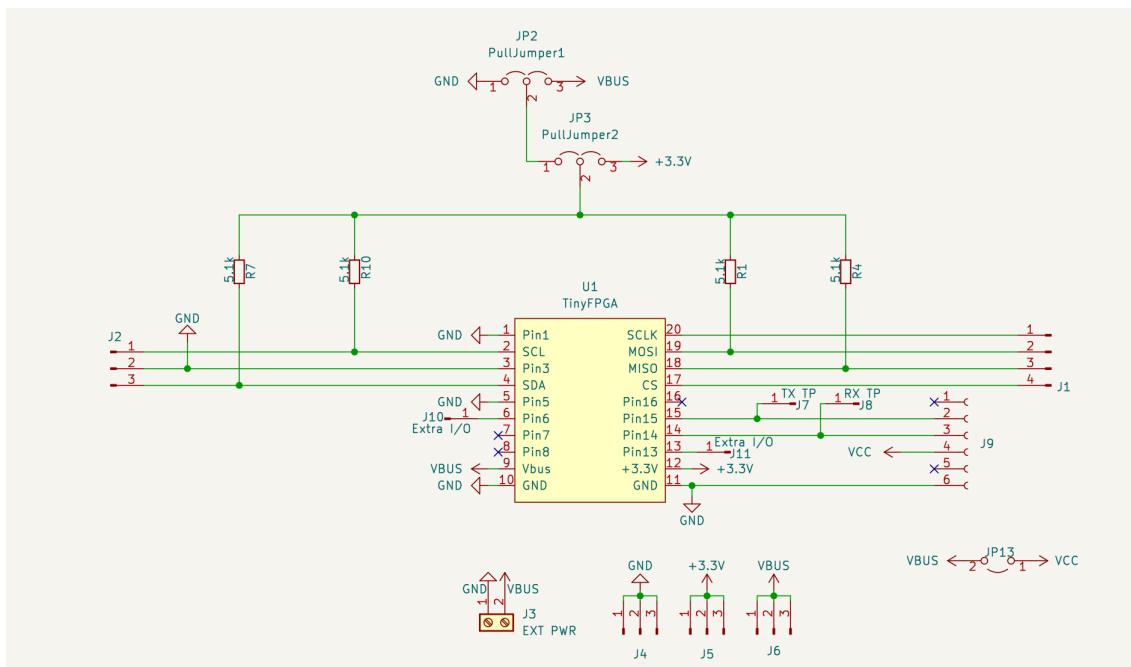
A tervezési folyamat biztonságát a beépített elektromos szabályellenőrző (ERC) és tervezési szabályellenőrző (DRC) rendszerek garantálják, amelyek képesek a gyakori tervezési hibák, például az elmaradt összeköttetések vagy a nem megfelelő távolságok automatikus felismerésére.

A gyártási előkészítést a KiCad támogatja gyártási fájlok (például Gerber, drill fájlok) generálásával, valamint beépített anyagjegyzék/BOM készítő funkcióval is rendelkezik. Ezen felül a KiCad lehetővé teszi saját szimbólum- és lábnyomkönyvtárak létrehozását, valamint 3D modellek hozzárendelését az alkatrészekhez. Amit és is kihasználtam, az, hogy a KiCad szoftver Python szkriptek támogatásával bővíthető,

amely különösen hasznos mivel nyílt forráskódú a KiCad. A kiterjedt, közösségi alapú dokumentáció és a hozzáférhető tanulási források tovább növelik a program használhatóságát mind kezdő, mind haladó felhasználók számára. Ezek miatt választottam a KiCad-ot.

3.3.2. A KiCad ban végzett munka bemutatása

Kapcsolási rajz

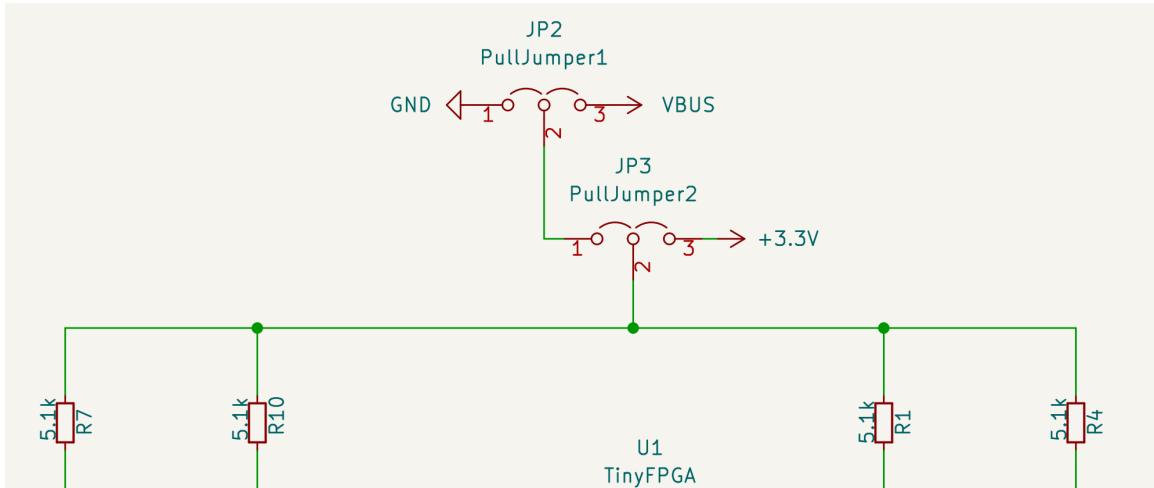


3.8. ábra. A programozó kapcsolási rajza

A kapcsolási rajz viszonylag egyszerű, nincs sok alkatrész. A felhúzó ellenállásokat kivéve csak csatlakozók meg jumper-ek vannak a NYÁK-on. Mégis igyekeztem sok időt fordítani a tervezésre.

A céлом az volt, hogy ez egy általánosan használható EEPROM programozó legyen, ehhez rugalmasnak kellet megterveznem. A jumper-ek segítségével az ellenélésök 3V3 vagy 5V felhúzó ellenállások, vagy akár lehúzó ellenállások ként is tudnak szolgálni. Így sok féle EEPROM module-okat tud támogatni az áramkőr. Ha a JP2 jumper 1-es és 2-es pin-je van közösítve, akkor a GND van kiválasztva. Ha a JP2 jumper 2-es és 3-es

pin-je van közösítve, akkor 5V van kiválasztva. Ha a JP3 jumper 2-es és 3-es pin-je van közösítve, akkor 5V van kiválasztva.



3.9. ábra. Ellenállás Jumper-ek

	JP2	JP3	Ellenállósokon lévő feszültség
Pin-ekkörönként	1,2	1,2	GND (0V)
	2,3	1,2	5V
	1,2	2,3	3.3V
	2,3	2,4	3.3V

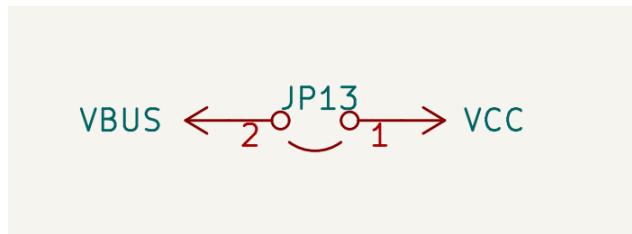
3.10. ábra. Ellenállósokon lévő feszültség összefüggése

A áramkörnek 4 lehetséges tápja van

- A TinyFPGA 5V VBUS PIN-je. A pin közvetlen össze van kötve a module mikró USB 5V tápjával. Tehát amikor a TinyFPGA module csatlakoztatva van a mikró USB-n keresztül a VBUS pin kimenet ként szolgálhat. A TinyFPGA module a betápját is a VBUS látja el. Szóval, ha a TinyFPGA nincs csatlakoztatva az USB-hez akkor a VBUS pin bemenetként is működhet.
- A TinyFPGA 3V3 PIN. Ez a kimenet a TinyFPGA module power management IC-je generálja a VBUS 5V feszültégéből.

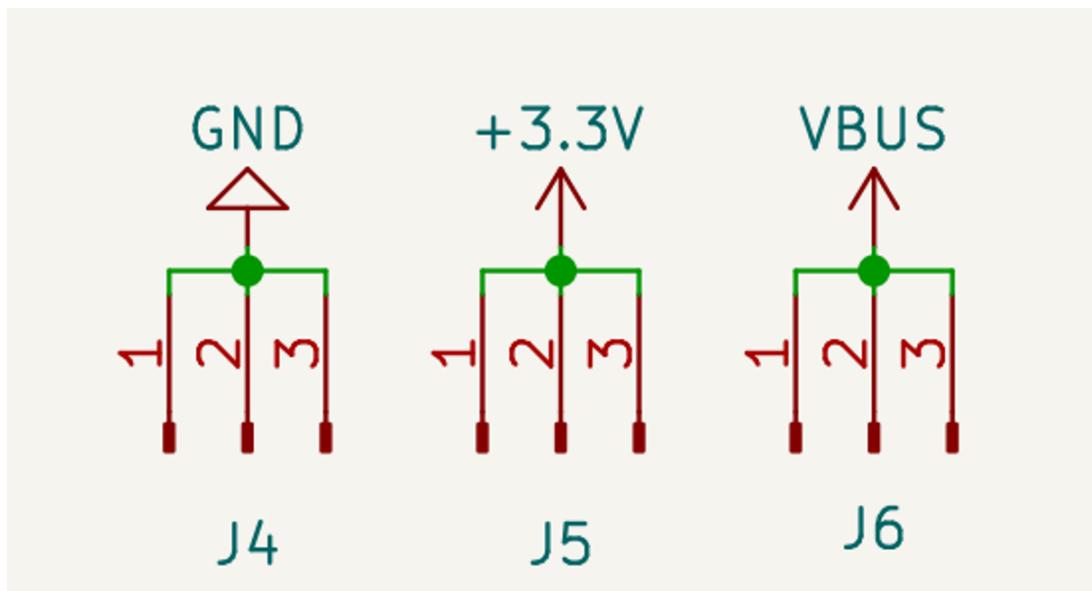
- Az FTDI232 VCC kimenete. Ami alítható 3V3 és 5V között a FTDI232 module-ön egy jumper-el.
- Az EXT PWR csatlakoztatón, amit biztonság kedvéért raktam hozzá az áramkörhöz, ha szükséges lenne a jövőben egy külső táp..

A Jumper-ek úgy vannak megtervezve, hogy bármijeik 5V-od bemenet szolgálhasson táp ként a NYÁK-nak meg a hozzá csatolt memória module-oknak. De a tervezett legtöbbet használt konfiguráció az a FTDI232 VCC kimenetét használja az 5V módban. Ilyenkor persze a JP13 jumper két pin-je közösítve kell, hogy legyen, hogy a TINY FPGA megkapja a 5V tápot.



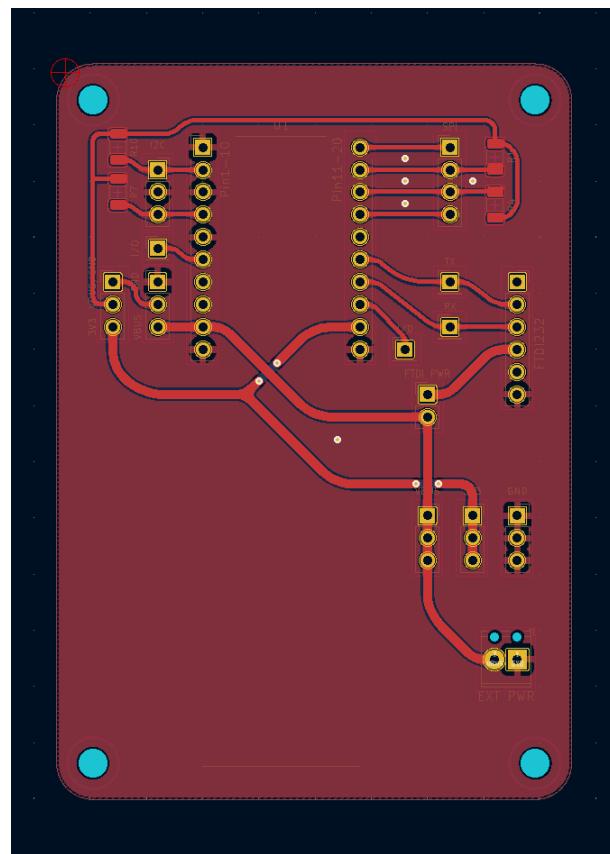
3.11. ábra. VCC és VBUS közösítő jumper

A NYÁK-on ki van vezetve 3-3 pin-en a GND, 3V3, és a VBUS. Ezek a csatlakozok tápként szolgálnak a csatlakoztatott memória module-oknak, illetve merés pontoknak is alkalmasak.



3.12. ábra. Fontos feszültségek és a GND kivezetése

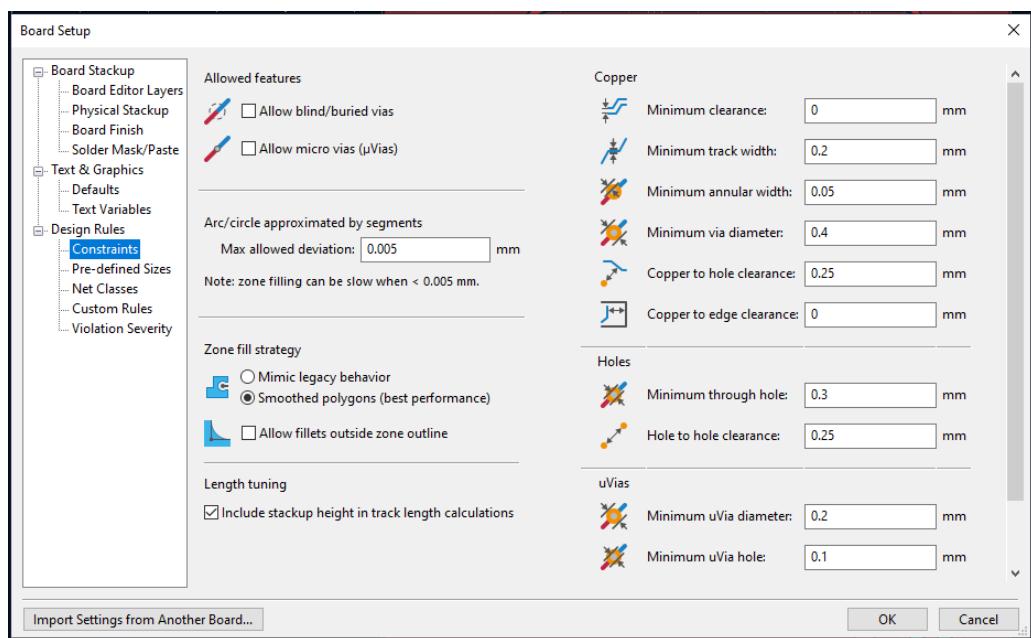
NYÁK tervezés



3.13. ábra. A programozó NYÁK-ja

Ez a végleges NYÁK terv, ez az a design, aminek gyártását megrendeltem. Egy két réteges egyszerű nyák. De egy ilyen egyszerűbb NYÁK tervezése közben fontos betartani az alapszabályokat.

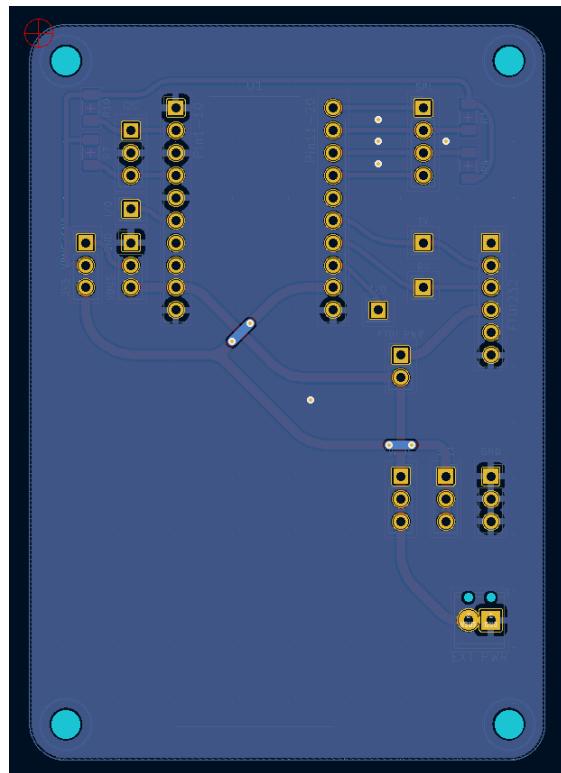
Tervezés előtt érdemes beállítani azokat a limiteket, amiket a NYÁK gyártó cég képes gyártani. Ha tudjuk, hogy melyik céget akarjuk használni, akkor a weboldalukon megtalálhatjuk az értékeket. A limiteket a Board Setup funkcióval lehet megadni. Ha jól meg vannak adva az értékek akkor a KiCad nem enged olyan nyákot tervezni, amit nem tud a NYÁK gyártó cég legyártani. Az én esetemben:



3.14. ábra. KiCad limit beállítások

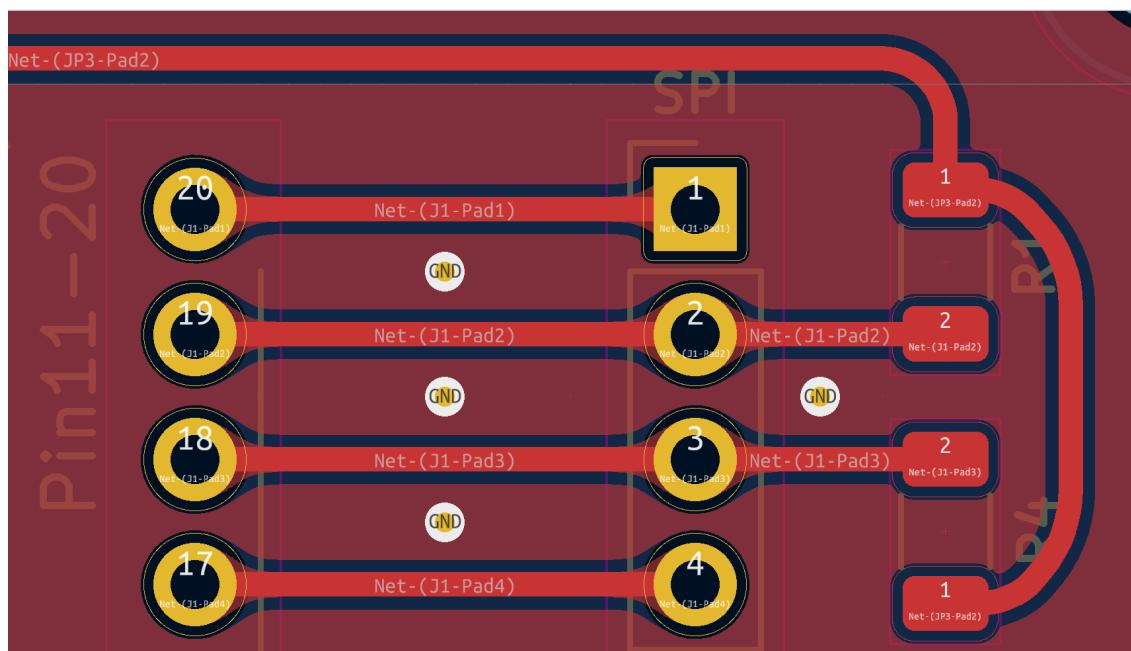
A NYÁK alsó rétege

A hátsó réteg itt egyszerű. Az egész egy GND réteg, minél kevesebb megszakítással, hogy minimalizálva legzenek az áram visszaútvai, ez ezáltal az arám hurkok területe is.



3.15. ábra. A NYÁK alsó rétege

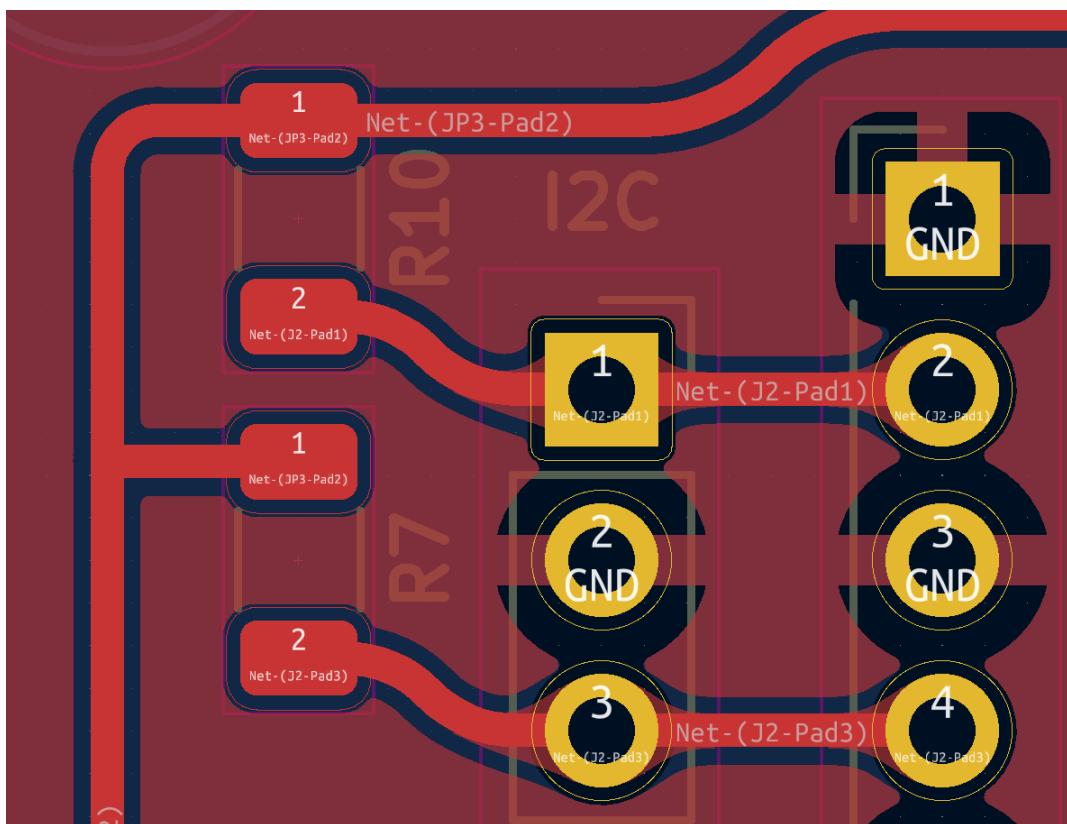
Az SPI csatlakozó



3.16. ábra. SPI kimenet

Kissé túl van tervezve az SPI csatlakozó, hiszen nem óriási frekvenciákon használom. De a túl tervezés ebben az esetben nem árt. A chip select, MISO, MOSI, és CLK vonalak mind ugyan olyan hosszúak. Ezt „length matching” nek hívjuk. Akkor fontos, ha akkora az adatátviteli alap frekvencia, hogy két vonal hossz különbségéből származó kettő közötti fázistolás gondot okoz a mintavételezésnél. A vonalak egymástól GND-vel le vannak árnyékolva, ez segíti, hogy a vonalak elektromágneses zaja kevésbé befolyásolják egymást.

Az I2C csatlakozó

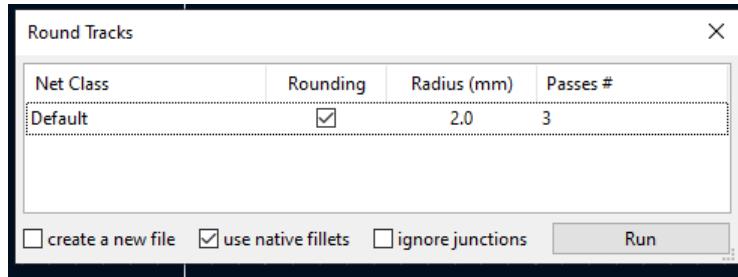


3.17. ábra. I2C kimenet

Itt is inkább túl terveztem a kimenetet, mint alul. Itt is a SCL IS SDA vonal „length match”-elve van. Illetve az árnyékolást még a csatlakozókra is kiterjed. Így egészen a FPGA pin-ekhez az árnyékolást végig lehet vinni ha a SCL és SCL-hez tartozó lábak közötti lábat logikai '0'-ra alítom.

Plugin-ek

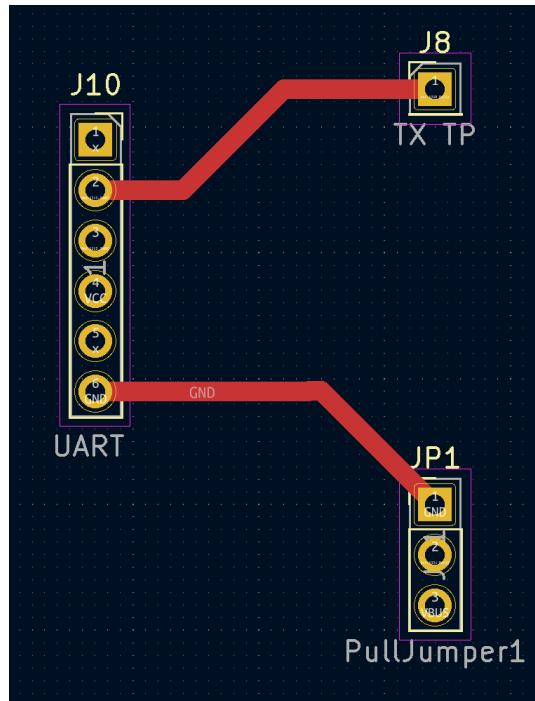
Én két plug-in-t használtam. Az egyik a rounded tracks plug-in. Ami igazából csak esztétikai változásokat kreál a nyákon. A plugin-t nagyon egyszerű használni. A következő kép a UI-ja.



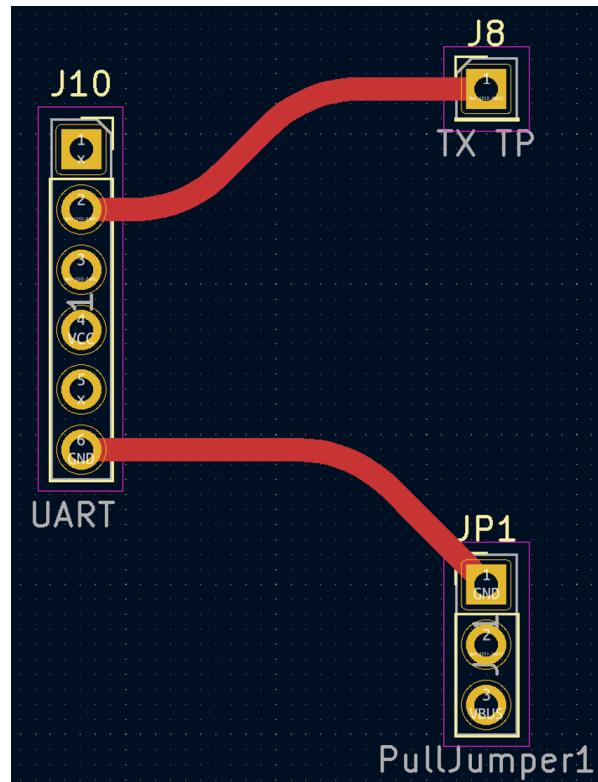
3.18. ábra. A rounded tracks plugin UI-ja

A UI egyszerű. Netlista kent ki lehet választani, hogy kerekítse a plugin a netlisthez tartozó trace-eket vagy nem. Ki lehet választani célzott rádiumszt, illetve, hogy hányszor fussen a programm. A Run gombbal kell elindítani.

Ezek pedig arról képek hogy hogyan néz ki egy trace a plugin használata előtt és utána Előtte:

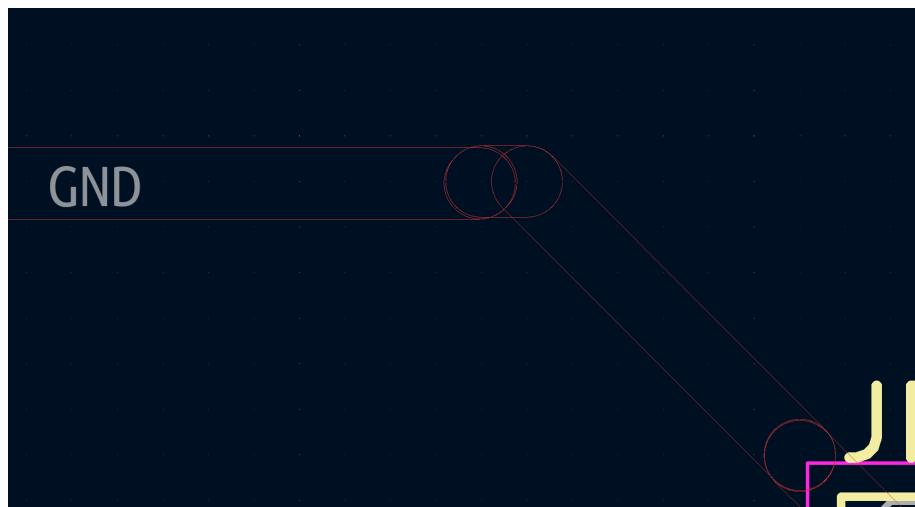


3.19. ábra. Egy trace a plugin használata előtt



3.20. ábra. Egy trace a plugin használata után

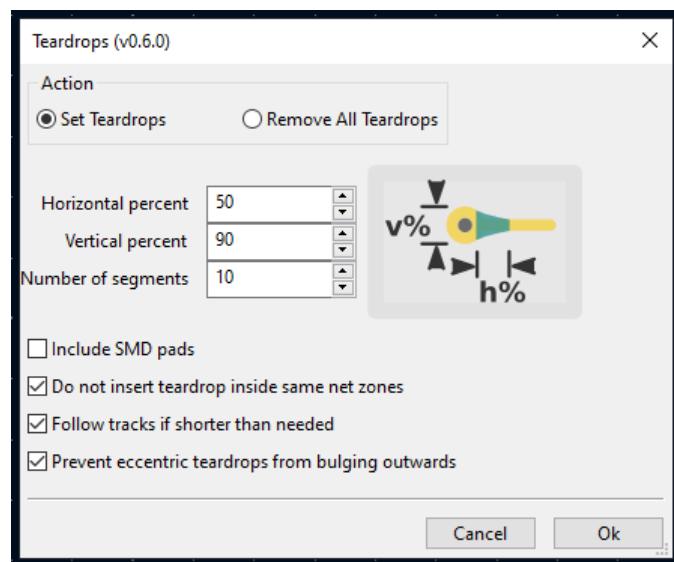
Fontos hogy a plugin használata előtt nézzük át a tarce-einket. A plugin nem működik jól, ha túl sok elemre bontott csúnyán tervezett trace-et akarunk kerekíteni.



3.21. ábra. Példa egy csúnyán tervezett trace-re

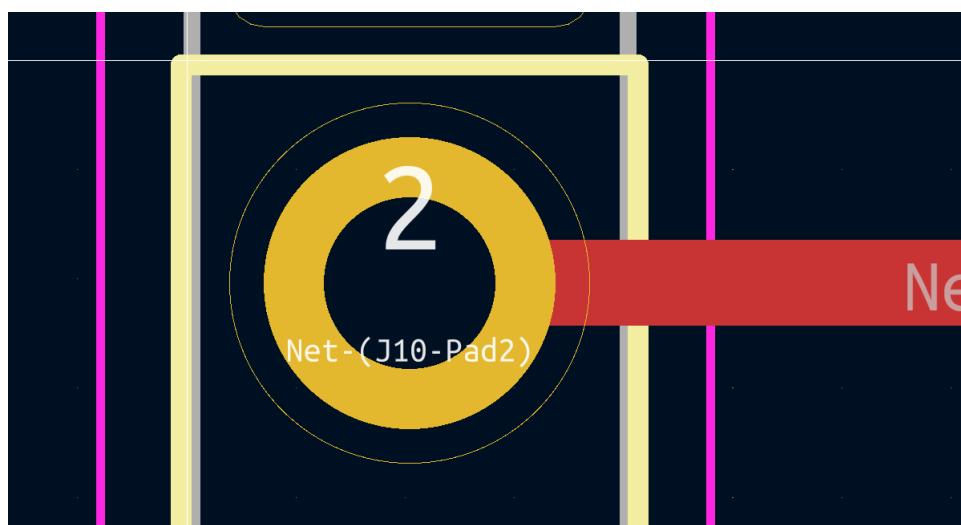
A másik plug-in pedig a Teardrop vias plug in. Ez is egy egyszerű plug-in ami rész-

ben Esztétikailag szépíti a nyák tervet, de fontos, hogy az éles sarkokat minimalizálja is. Hiszen az éles sarkok savcsapdákat okozhatnak, ahol a maratáshoz használt savak egy része megmarad, és az éles sarkoknál tovább korrodálja a rezet. A következő kép a UI-ja.

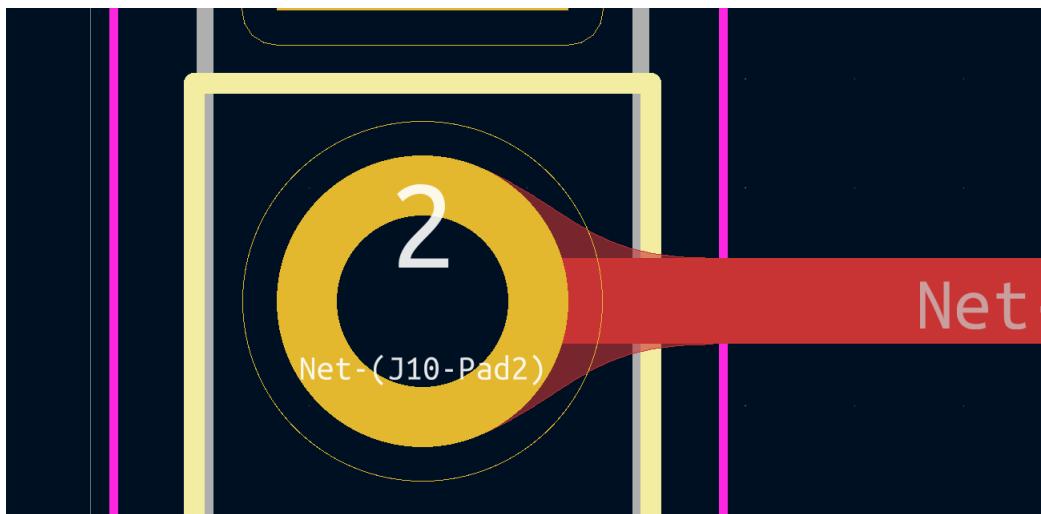


3.22. ábra. Teardrop vias UI

Ez pedig arról kép hogy hogyan néz ki egy via a plugin használata előtt és utána.



3.23. ábra. Teardrop plugin használata előtt

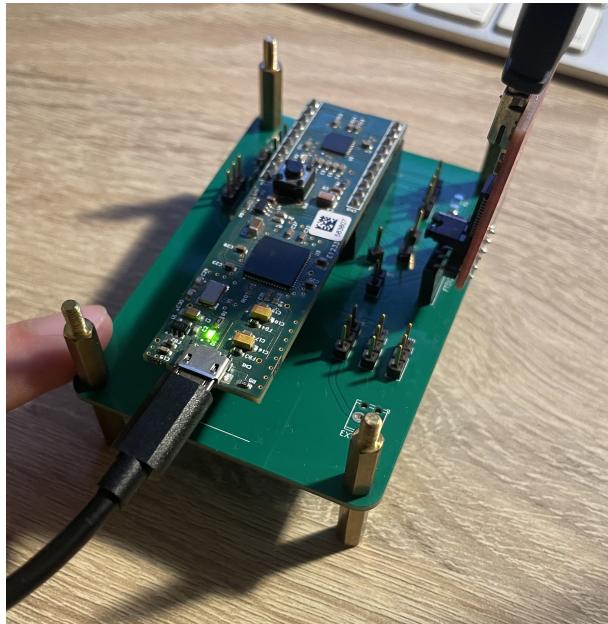


3.24. ábra. Teardrop plugin használata után

3.4. A Kész NYÁK



3.25. ábra. A Kész NYÁK



3.26. ábra. A kész NYÁK összerakva és forrasztva

3.5. A Programozó al-entitásai

A következőkben bemutatom az al-entitásokat, amiket használtam a programozóban. Mindet én terveztem és írtam meg, kivéve az Int_Osc entitást, amit az FPGA-m gyártója által adott módon írtam meg.

3.5.1. A Int_Osc entitás

Az Int_Osc egy belső oszcillátor modult valósít meg, amely egy órajelet generál a rendszer számára. Ez a modul a MachXO2 FPGA belső oszcillátorát használja, amely egy konfigurálható frekvenciájú órajelet biztosít.

Az Int_Osc entitás két portot tartalmaz:

- StdBy (in std_logic): Ez a bemeneti jel az oszcillátor készenléti (standby) állapotát vezérli. Ha a jel aktív, az oszcillátor leáll, és nem generál órajelet. Ha inaktív, az oszcillátor működik és órajelet generál.
- Clk (out std_logic): Ez a kimeneti jel az oszcillátor által generált órajelet biztosítja a rendszer többi részének.

Az architektúra az FPGA belső oszcillátorát (OSCH komponens) használja. Az OSCH komponens a következőket tartalmazza:

- NOM_FREQ (generikus paraméter): Az oszcillátor névleges frekvenciáját határozza meg. Az alapértelmezett érték itt "9.17", ami 9,17 MHz-es órajelet jelent.
- STDBY (bemenet): Az oszcillátor készenléti állapotát vezérli.
- OSC (kimenet): Az oszcillátor által generált órajel.
- SEDSTDBY (kimenet): Egy diagnosztikai jel, amely az oszcillátor készenléti állapotát jelzi (ebben az implementációban nem használják).

Az architektúra a következőképpen működik:

1. Az OSCH komponens egy példányát (OSCInst0) hozza létre.
2. A NOM_FREQ generikus paraméter értéke "9.17", amely meghatározza az oszcillátor frekvenciáját. Ez választhatóan módosítható a tervező által.
3. A STDBY bemenet az Int_Osc entitás StdBy portjához van kötve.
4. Az OSC kimenet az Int_Osc entitás Clk portjához van kötve.

Az Int_Osc modul a rendszer órajelének biztosítására szolgál. Az óra jelet a rendszer összes része használja a szinkron működéshez.

Az órajel sebeségének válastása

A FPGA belő órajel sebességét adott értékekből lehet választani. Az FPGA termékcatalógus adatlapja szerint ezek a sebességek közül lehet:

MCLK (MHz, Nominal)	MCLK (MHz, Nominal)	MCLK (MHz, Nominal)
2.08 (default)	9.17	33.25
2.46	10.23	38
3.17	13.3	44.33
4.29	14.78	53.2
5.54	20.46	66.5
7	26.6	88.67
8.31	29.56	133

3.27. ábra. Elérhető CLK-frekvenciák az FPGA adatlapja alapján [1]

Az óra jelet úgy kell választani, hogy elég gyors legyen, hogy a design megoldja a feladatot amire tervezve lett. Gyorsabba nem érdemes, mert az felesleges slack problémákhoz vezethet.

A programozóban az órajelnek legalább 4-szer gyorsabbnak kell lennie, mint a célzott I2C órajel sebessége, mert az I2C órajelenként 4 műveletet végzik a I2C entitás. És 2 szer gyorsabbnak, mint a célzott SPI órajel, mert az SPI órajelenként 2 műveletet végzik a SPI entitás. Az én célzott I2C órajelsebességem nagyobb mint a célzott uart baud ráta de lassab mint 1MHz-nel. Ez azért van, mert a legtöbb modern EEPROM képes 1MHz órajelsebességre, és ki szeretném használni ezt a sebességet. És azért kell hogy gyorsabb legyen az I2C mint az UART mert az I2C tud "várni" az UART adatra órajelnyújtással, de a UART nem tud "várni" a I2C-re mert az UART aszinkron. Az SPI célsabosság pedig 5MHz körül van, hasonló okok miatt.

A programozóm UART-ot is használ, amint előbb volt tárgyalva, egy aszinkron kommunikációs protokoll. Ez azt jelenti, hogy a belső óra jelet úgy kell választani, hogy jól leosztható legyen a megcélzott baud rátához. Tegyük fel, hogy a célzott baud rátá 23040. Ha a belső órajel 2.46MHz akkor az órajelosztó számlálónak 11-ig kell számolnia, mivel $230400\text{baud}/2.46\text{MHz} = 10.68$. Egy órajel számláló csak egész számokkal tud számolni. így minden egyes periódus alatt fél óra jelet késne a belső UART számlálá a tényleges 230400 baud-hoz. Úgy kellett választanom az óra jelet, hogy minimális kerekítéssel le lehessen osztania és ez a késés minimális legyen. A célzott AURT sebesség 576000 baud.

Ezek miatt választottam a 9.17MHz-t belső órajelnek. Így a UART osztó hiba 0.08

órajel per baud periódus. A I2C maximum sebessége 2.29MHz. Az SPI Max sebessége 4.59MHz.

3.5.2. A write8bit entitás

A write8bit e8 bites adat írására szolgál az SPI vonalokon. Az entitás bemeneti és kimeneti portjai a következők:

Portok:

- iCS: Bemeneti vezérlőjel, amely az írási folyamat indítását jelzi ('0' értékkel aktiválódik).
- cs: Kimeneti vezérlőjel, amely az írási folyamat állapotát jelzi ('1' az alapértelmezett érték).
- Clk: Órajel bemenet, amely az állapotgép működését vezéri.
- Done: Kimeneti jel, amely az írási folyamat befejezését jelzi ('1' az alapértelmezett érték).
- bit8SCL: Órajel kimenet a soros kommunikációhoz ('1' az alapértelmezett érték).
- nReset: Reset bemenet, amely az állapotgépet alaphelyzetbe állítja ('0' értékkel aktiválódik).
- bit8SDA: Adat kimenet a soros kommunikációhoz ('1' az alapértelmezett érték).
- SDA_data: 8 bites adat bemenet, amelyet a modul sorasan továbbít.

A write8bit entitás egy állapotgépet tartalmaz, amely három állapotból áll:

1. Idle: Alapértelmezett állapot, ahol az órajel (bit8SCL) és az adatvonal (bit8SDA) magas szinten van, és a Done jel aktív ('1'). Ha az iCS bemenet alacsony szintre kerül ('0'), az állapotgép a write8 állapotba lép.

2. write8: Az adat írásának állapota. Az adatot a SDA_data bemenetről bitenként továbbítja a bit8SDA kimenetre. Az írási folyamatot a BitIndex számláló vezérli, amely 0-tól 7-ig számol. Ha az összes bitet elküldte, az állapotgép a stop állapotba lép.

3. stop: Az írási folyamat lezárása. A cs_jel visszaáll magas szintre ('1'), és az állapotgép visszatér az Idle állapotba.

Az állapotgép működését az órajel (Clk) vezérli, és az nReset bemenet bármikor alaphelyzetbe állíthatja. A Main entitásban ez az entitás felelős az egyszerű csupán 8 bites egyirányú (MOSI) SPI kommunikációkért.

3.5.3. UART_TX entitás

A UART_TX az UART protokoll szerinti adatküldést valósítja meg. Ez a modul soros adatátvitelre szolgál, ahol a bemeneti adatokat bitenként továbbítja egy kimeneti vonalon

Portok:

- TXData: 8 bites bemeneti adat, amelyet a modul sorasan továbbít.
- Clk: Órajel bemenet, amely az állapotgép működését vezérli.
- SendTX: Bemeneti vezérlőjel, amely az adatküldés indítását jelzi ('1' értékkel aktiválódik).
- TXDone: Kimeneti jel, amely az adatküldés befejezését jelzi ('1' az alapértelmezett érték).
- nReset: Reset bemenet, amely az állapotgépet alaphelyzetbe állítja ('0' értékkel aktiválódik).
- TXOut: Kimeneti adatvonal, amelyen a soros adatátvitel történik.
- TXActive: Kimeneti jel, amely az adatküldés aktív állapotát jelzi ('1', ha a modul éppen adatot küld).

Generikus paraméter:

- ClkRatio: Az órajel osztására szolgáló paraméter, amely meghatározza az UART adatátviteli sebességét (baud rate).

A UART_TX entitás egy állapotgépet tartalmaz, amely az UART adatküldési folyamatát valósítja meg. Az állapotgép az alábbi állapotokból áll:

1. Idle: Alapértelmezett állapot, ahol a modul várakozik a SendTX jel aktiválására. Az órajel számláló (ClkCount) nullázódik, és a kimeneti vonal (TXOut) magas szinten van ('1').
2. StartBit: Az adatküldés kezdő bitje ('0') kerül a kimeneti vonalra. Az órajel számláló növekszik, és ha eléri a ClkRatio értéket, az állapotgép a DataBits állapotba lép.
3. DataBits: Az adatbitek soros továbbítása történik. A TXData bemenet bitjeit a BitIndex számláló segítségével küldi ki a modul. Ha az összes bit elküldésre került, az állapotgép a StopBit állapotba lép.
4. StopBit: Az adatküldés záró bitje ('1') kerül a kimeneti vonalra. Az órajel számláló növekszik, és ha eléri a ClkRatio értéket, az állapotgép a FrameEnd állapotba lép.
5. FrameEnd: Az adatküldés befejeződik, a TXActive jel inaktívvá válik ('0'), és a TXDone jel aktívvá válik ('1'). Az állapotgép visszatér az Idle állapotba.

Az állapotgép működését az órajel (Clk) vezérli, és az nReset bemenet bármikor alaphelyzetbe állíthatja.

3.5.4. A UART_RX entitás

A UART_RX entitás a protokoll szerinti adatfogadást valósítja meg. Ez a modul soros adatokat fogad egy bemeneti vonalon, és azokat párhuzamos formátumba alakítja.

- Clk: Órajel bemenet, amely az állapotgép működését vezéri.

- RXIn: Soros adat bemenet, amelyen keresztül az UART adatokat fogadja.
- Ack: Bemeneti vezérlőjel, amely az adatfogadás befejezésének visszaigazolására szolgál.
- RXData: 8 bites kimeneti adat, amely a fogadott adatot tartalmazza.
- RXDataReady: Kimeneti jel, amely az adatfogadás befejezését jelzi ('1' értékkel aktiválódik).
- nReset: Reset bemenet, amely az állapotgépet alaphelyzetbe állítja ('0' értékkel aktiválódik).
- RXActive: Kimeneti jel, amely az adatfogadás aktív állapotát jelzi.

Generikus paraméterek:

- ClkRatio: Az órajel osztására szolgáló paraméter, amely meghatározza az UART adatátviteli sebességét (baud rate).
- CRHalf: Az órajel osztásának felezett értéke, amely a bitközép detektálására szolgál.

A UART_RX entitás egy állapotgépet tartalmaz, amely az UART adatfogadási folyamatát valósítja meg. Az állapotgép az alábbi állapotokból áll:

1. Idle: Alapértelmezett állapot, ahol a modul várakozik az adatfogadás kezdetére. Ha a RXIn jel alacsony szintre kerül ('0'), az állapotgép a StartBit állapotba lép.
2. StartBit: Az adatfogadás kezdő bitjének ('0') detektálása történik. Az órajel számláló (ClkCount) növekszik, és ha eléri a ClkRatio értéket, az állapotgép a DataBits állapotba lép.
3. DataBits: Az adatbitek fogadása történik. A RXIn bemenet bitjeit a SData regiszterbe menti a modul, a BitIndex számláló segítségével. Ha az összes bitet fogadta, az állapotgép a StopBit állapotba lép.

4. StopBit: Az adatfogadás záró bitjének ('1') detektálása történik. A RXDataReady jel aktívvá válik ('1'), jelezve, hogy az adat fogadása befejeződött. Ha az Ack jel aktív ('1'), az állapotgép visszatér az Idle állapotba.

Az állapotgép működését az órajel (Clk) vezérli, és az nReset bemenet bármikor alaphelyzetbe állíthatja.

3.5.5. A writePage entitás

A writePage entitás a komplexebb SPI kommunikációkat kezeli. A neve writepage maradt, mert eredetileg a tervem az volt, hogy két entitást csinálok. Egy entitást, ami az SPI írásért felelős, egy pedig ami az olvasásért felelős. De végül egyszerűbb volt egy entitásban megvalósítani mind két funkciót. Ez az entitás azért is komplexebb mert, eltérően a write8bi-től képes a kommunikációs protokoll kerete hosszán alítani.

Portok:

- iCS: Bemeneti vezérlőjel, amely az adatátviteli folyamat indítását jelzi ('0' értékkel aktiválódik).
- RW: Bemeneti jel, amely az írási ('0') vagy olvasási ('1') műveletet határozza meg.
- Rdy: Kimeneti jel, amely az adatátviteli folyamat készenlétét jelzi ('1' az alapértelmezett érték).
- cs: Kimeneti vezérlőjel, amely az adatátvitel állapotát jelzi ('1' az alapértelmezett érték).
- from_SDO: Bemeneti adatvonal, amelyen keresztül az olvasott adat érkezik.
- Done: Kimeneti jel, amely az adatátviteli folyamat befejezését jelzi ('1' az alapértelmezett érték).
- toSCL: Órajel kimenet a soros kommunikációhoz ('1' az alapértelmezett érték).

- nReset: Reset bemenet, amely az állapotgépet alaphelyzetbe állítja ('0' értékkel aktiválódik).
- toSDA: Adat kimenet a soros kommunikációhoz ('1' az alapértelmezett érték).
- MISO_DR: Kimeneti jel, amely az olvasási művelet befejezését jelzi.
- next8bits: 8 bites bemeneti adat, amelyet a modul írni fog.
- read8bits: 8 bites kimeneti adat, amely az olvasott adatot tartalmazza.
- RW_length: Az SPI művelet teljes hosszát meghatározó bemeneti paraméter (bájtokban).
- RH_length: Az olvasási művelet címző keret hosszát meghatározó bemeneti paraméter (bájtokban).

Az állapotgép működését az órajel (Clk) vezérli, és az nReset bemenet bármikor alaphelyzetbe állíthatja. Az állapotgép négy fő állapotot tartalmaz:

1. Idle (Alapállapot)

- Funkció: Az állapotgép várakozik, amíg az iCS jel alacsony szintre kerül ('0'), ami az SPI kommunikáció kezdetét jelzi.
- Tevékenységek:
 - Az összes kimeneti jel alaphelyzetbe állítása.
 - Az RW jel értéke a RWlatch jelbe kerül.
 - Ha iCS = '0', akkor:
 - * Az állapot write8-ra vált
 - * cs = '0' (chip select aktív).
 - * Done = '0' (művelet folyamatban).
 - * A next8bits bemeneti adatot a current8bits jel tárolja.

2. write8 (Írási állapot)

- Funkció: Az aktuális 8 bites adat (current8bits) bitenként kerül kiküldésre az SPI buszra.
- Tevékenységek:
 - Az toSDA jel az aktuális bit értékét veszi fel (current8bits(7-BitIndex)).
 - Az toSCL jel váltakozik ('0' → '1'), hogy az órajelet biztosítsa.
 - A BitIndex növekszik minden bit kiküldése után.
 - Ha az összes bit kiküldésre került (BitIndex = 7), akkor:
 - * Az állapot stop-ra vált.
 - * A BitIndex alaphelyzetbe áll.

3. read8 (Olvasási állapot)

- Funkció: Az SPI buszról érkező adatot bitenként olvassa be.
- Tevékenységek:
 - Az toSCL jel váltakozik ('0' → '1'), hogy az órajelet biztosítsa.
 - Az from_SDO bemeneti jel értéke a read8bits jel megfelelő bitjébe kerül (read8bits(7-BitIndex)).
 - A BitIndex növekszik minden bit beolvasása után.
 - Ha az összes bit beolvasásra került (BitIndex = 7), akkor:
 - * Az állapot stop-ra vált.
 - * A MISO_DR jel '1'-re áll, jelezve, hogy az adat érvényes.

4. stop (Befejezési állapot)

- Itt történik az állapotok közötti váltási logika az iCS, RW, BitIndex, és ByteIndex jelek alapján. Mindig várja az iCS jel alacsony szintjét, mielőtt állapotot vált. Ha a RWlatch = '1', akkor felelős annak ellenőrzéséért, hogy az SPI üzenet elérte-e a megadott fejléc hosszúságát. Ha igen, akkor write8 helyett read8-be ugrik. Illetve, ha az üzenet elérte a teljes hosszát, visszatér az Idle állapotba.

3.5.6. A I²C entitás

Az I²C entitás az IC2 protokoll szerinti adatátvitelt valósítja meg. Ez az entitás képes adatokat írni és olvasni egy I²C buszon, és támogatja a protokollohoz szükséges vezérlőjelek generálását. Ez az entitás is komplex, képes írni/olvasni és a kommunikációs protokoll kerete hosszán alítani.

Portok:

- DataIn: 8 bites bemeneti adat, amelyet a modul az I²C buszra ír.
- DataOut: 8 bites kimeneti adat, amely az I²C buszról olvasott adatot tartalmazza.
- RW_length: Az írási művelet hosszát meghatározó bemeneti paraméter (bájtokban).
- RH_length: Az olvasási művelet címző keret hosszát meghatározó bemeneti paraméter (bájtokban).
- Clk: Órajel bemenet, amely az állapotgép működését vezéri.
- Go: Bemeneti vezérlőjel, amely az I²C művelet indítását jelzi.
- Send_i2c: Bemeneti jel, amely az I²C adatátvitel indítását szabályozza.
- nReset: Aszinkron reset bemenet, amely az állapotgépet alaphelyzetbe állítja ('0' értékkal aktiválódik).
- RW: Bemeneti jel, amely az írási ('0') vagy olvasási ('1') műveletet határozza meg.
- rSDA: Kétirányú adatvonal az I²C buszhoz.
- rSCL: Órajel kimenet az I²C buszhoz.
- dataready: Kimeneti jel, amely az adatátvitel befejezését jelzi.
- atstop: Kimeneti jel, amely az I²C stop feltétel elérését jelzi.

- Done: Kimeneti jel, amely az I2C művelet befejezését jelzi.

A célzott órajelsebesség mint előbb említettem 576kHz és 1MHz között van. A belső órajel 9.17MHz, így 15-tel leosztva az I2C órajel 661kHz.

Állapotok Leírása:

- off
 - Cél: Ez az alapértelmezett, nyugalmi állapot, ahol az I2C modul várakozik az indító jelre.
 - Műveletek: Az összes belső számláló (Bitcount, Bytecount, count) és retesz (RWlatch,dataready) alaphelyzetbe állítása.
 - Átmenet: Send_i2c = '1' esetén az állapot start-ra vált.
- start
 - Cél: Az I2C start feltétel generálása.
 - Műveletek:
 - * A SDAen és SCLen jelek változatának a start feltétel létrehozásához.
 - * A count jel növelése az időzítés nyomon követésére.
 - Átmenet: 15 órajelciklus után az állapot PFHW-ra vált.
- PFHW (Packet Frame Header and Write):
 - Cél: Az adatcsomag fejlécének és az adatok bitenkénti továbbítása.
 - Műveletek:
 - * A SDAen jel meghajtása a DataIn aktuális bitjével.
 - * A SCLen változatának órajelek generálásához.
 - * A bit- és bájtszámlálók (Bitcount, Bytecount) nyomon követése.
 - Átmenet:
 - * 8 bit (1 bájt) továbbítása után az állapot wack-ra vált.

- * Több bájt esetén a Bytecount növekszik.
- wack (Write Acknowledge):
 - Cél: Az írási művelet után a slave eszköz visszaigazolásának (ACK) várakozása.
 - Műveletek:
 - * A SDAen jel '1' értékre állítása az SDA vonal felszabadításához.
 - * A SCLEN változatára órajelek generálásához.
 - * A golatch jel ellenőrzése a következő művelet meghatározásához.
 - Átmenet:
 - * Ha golatch = '1', az állapot PFHW-ra vált további adatátvitelhez.
 - * Ha az írási művelet befejeződött (Bytecount = RW_length), az állapot stop-ra vált.
 - * Ha olvasási műveletre váltás történik (Bytecount >= RH_length), az állapot readB-re vált.
- readB (Read Byte):
 - Cél: Egy bájt adat olvasása a slave eszkösről.
 - Műveletek:
 - * A SDAen jel '1' értékre állítása az SDA vonal felszabadításához.
 - * Az rSDA értékének rögzítése a DataOut jelbe a 8. órajelciklus alatt.
 - * A SCLEN változatára órajelek generálásához.
 - * A bit- és bájtszámlálók (Bitcount, Bytecount) nyomon követése.
 - Átmenet:
 - * 8 bit (1 bájt) olvasása után az állapot rack-ra vált.
- rack (Read Acknowledge):

- Cél: Visszaigazolás (ACK) küldése a slave eszköznek az olvasási művelet után.
- Műveletek:
 - * A SDAen jel '0' értékre állítása ACK küldéséhez, vagy '1' értékre NACK küldéséhez (ha az olvasási művelet befejeződött).
 - * A SCLen váltogatása órajelek generálásához.
 - * A golatch jel ellenőrzése a következő művelet meghatározásához.
- Átmenet:
 - * Ha golatch = '1', az állapot readB-re vált további adatfogadáshoz.
 - * Ha az olvasási művelet befejeződött (Bytecount = RW_length), az állapot stop-ra vált.
- restart:
 - Cél: Ismételt start feltétel generálása több részből álló tranzakciókhöz.
 - Műveletek:
 - * A SDAen és SCLen váltogatása az ismételt start feltétel létrehozásához.
 - * A count jel alaphelyzetbe állítása.
 - Átmenet: 15 órajelciklus után az állapot PFHW-ra vált.
- stop:
 - Cél: Az I2C stop feltétel generálása a kommunikáció befejezéséhez.
 - Műveletek:
 - * A SDAen és SCLen váltogatása a stop feltétel létrehozásához.
 - * Az összes belső számláló (count, Bytecount) alaphelyzetbe állítása.
 - * A dataready jel '0' értékre állítása.
 - Átmenet: 15 órajelciklus után az állapot off-ra vált.

3.6. A fő (MAIN) entitás

A main entitás biztosítja a programozó funkcionalitását, az összes alentitást használva. A felhasználó által küldött utasításokat értelmezi, és az USB-n keresztül küldött adatfolyamot továbbítja a I2C és SPI alentitásoknak. Azért is felelős, hogy a I2C és SPI adatfolyamok szinkronizálva maradjanak az UART adatfolyamatok.

3.6.1. A main entity állapotgépe

- 000 (Base State):
 - Cél: Az alapállapot, ahol a rendszer várakozik a bemeneti jelekre.
 - Műveletek:
 - * Az összes vezérlőjel alaphelyzetbe állítása.
 - * Ha az UART fogadási művelet befejeződött, az állapot a StateHolder értékére vált.
 - Átmenet: Az UART fogadási művelet (RXDR) és az SPI írási művelet (DoneW) befejeződése után.
- 001 (SPI Configure):
 - Cél: Egy bájtos SPI kommunikációk végrehajtása.
 - Műveletek:
 - * Az SPI vonalak vezérlése a write8bit modul segítségével.
 - * Az UART fogadási adatainak (DataFromRx) továbbítása az SPI modul felé.
 - Átmenet: Az SPI művelet befejezése után visszatérés az alapállapotba.
- 010 (SPI Read):
 - Cél: Az SPI olvasási művelet végrehajtása.
 - Műveletek:

- * Az SPI olvasási művelet indítása a writePage modul segítségével.
- * Az olvasott adat (SDO_datap) továbbítása az UART felé.
- Átmenet: Az SPI olvasási művelet befejezése után visszatérés az alapállapotba.
- 011 (SPI Write):
 - Cél: Az SPI írási művelet végrehajtása.
 - Műveletek:
 - * Az SPI írási művelet indítása a writePage modul segítségével.
 - * Az UART fogadási adatainak (DataFromRx) továbbítása az SPI modul felé.
 - Átmenet: Az SPI írási művelet befejezése után visszatérés az alapállapotba.
- 100 (Get Settings):
 - Cél: A rendszer konfigurációs beállításainak frissítése.
 - Műveletek:
 - * Az UART fogadási adatainak (DataFromRx) tárolása a settings vektorban.
 - Átmenet: A settings vektor feltöltése után visszatérés az alapállapotba.
- 101 (I2C Read):
 - Cél: Az I2C olvasási művelet végrehajtása.
 - Műveletek:
 - * Az I2C olvasási művelet indítása az I2C modul segítségével.
 - * Az olvasott adat (i2c_DataOut) továbbítása az UART felé.
 - Átmenet: Az I2C olvasási művelet befejezése után visszatérés az alapállapotba.

- 110 (I2C Write):
 - Cél: Az I2C írási művelet végrehajtása.
 - Műveletek:
 - * Az I2C írási művelet indítása az I2C modul segítségével.
 - * Az UART fogadási adatainak (DataFromRx) továbbítása az I2C modul felé.
 - Átmenet: Az I2C írási művelet befejezése után visszatérés az alapállapotba.

3.7. C++ ban megírt utasítás küldő program

3.7.1. A Terminal app program

A program célja, hogy hexadecimális formátumban megadott adatokat küldjön a számítógéphez csatlakoztatott programozónak egy COM soros porton keresztül, majd fogadja és megjelenítse a programozótól visszakapott adatokat.

A program működése

1. Soros port megnyitása és beállítása.
 - A program a CreateFile függvényel megnyitja a COM portot olvasásra és írásra.
 - A SetupComm függvényel 1 MB-os bemeneti és kimeneti puffer állít be a soros porthoz.
 - A DCB struktúrában beállítja a kommunikációs paramétereket. Például: Baud rate: 576000, 8 adatbit, 1 stopbit, és paritás bit nélküli adatkeret.
 - A COMMTIMEOUTS struktúrában beállítja az időzítéseket az olvasáshoz és íráshoz.
2. Felhasználói adatbekérés és feldolgozás.

- A program egy végtelen ciklusban várja a felhasználó bemenetét.
- A felhasználó hexadecimális karakterláncot írhat be (például: AABBCC), vagy az exit szót a kilépéshez.
- Az üres sorokat figyelmen kívül hagyja.
- A beírt hexadecimális karakterláncot bájt tömbbé alakítja (két karakter = 1 bájt).

3. Adatküldés a programozónak.

- Az adatokat 4096 bájtos (4 KB) blokkokban küldi el a soros porton keresztül.
- minden elküldött blokk után kiírja, hány bájtot sikerült elküldeni.

4. Válasz fogadása a programozótól.

- 50 ms várakozás után lekérdezi, mennyi adat érkezett vissza a soros port bemeneti bufferébe.
- A beérkezett adatokat szintén 4096 bájtos blokkokban olvassa be.
- Az olvasott adatokat hexadecimális formátumban jeleníti meg a képernyőn.

3.7.2. A File app program

Ez a program hasonló a Terminal app-hoz, de fájlakkal dolgozik. A felhasználó megadhat egy fájlnevet, amely utasításokat tartalmaz. A fájlban lévő utasításokat a program feldolgozza, és a válaszokat kiírja, meg lementi egy log.txt fájlba.

3.7.3. A programozó beállítása a belső memória segítségével

A programozónak van egy 7 bájtos belső memóriája, ahol a SPI és I2C adatkeret beállításokat tárolja. Az alapértelmezett értéke x00001904000403. Az elő 3 bajt, azaz alapból a x000019, A SPI adatkeret teljes hosszát (bájtokban) mondja meg, U3 adattípusban. Így alapértelmezett esetben a SPI kommunikáció teljes hossza 25 bájt, minden beleérte. A következő 2 bájt pedig a SPI olvasás esetén az olvasási fejléc hossza U1

adattípusban. Alapértelmezett esetben 4 bajt a hossz. A következő 4 bajt pedig a I2C adatkeret teljes hosszát (bájtokban) mondja meg, U2 adattípusban. Alapértelmezett esetben 4 bajt a hossz. Az utolsó 2 bajt pedig a I2C olvasás esetén az olvasási fejléc hossza U1 adattípusban. Alapértelmezett esetben 3 bajt a hossz. Tegyük fel, hogy egy SPI olvasást, meg egy I2C olvasást akarunk végrehajtani. Mégpedig egy teljes page olvasást a W25Q64JV SPI Flash-ből, és egy 4 bajtos olvasást szeretnénk az AT24C256 EEPROM-ból. A page hossza összesen 256, egy utasítás bájtnak kell lennie, és 3 cím bájtnak [3]. Tehát a teljes adatkeret hossz 260 bajt lesz, és az olvasási fejléc hossza 4 bajt[2]. Választott címen való olvasáshoz 4 fejlécbajt kell az EEPROM-nak. Így a beállítás belső memóriát x00010404000804 re kell alítani. Az utasítás, ami ezt meg teszi az ez: 4400010404000804. Az elő bájt a programozót a megfelelő „get setting” állapotba rakja. Az utasítás többi része, az, amit be akarunk írni a belső memóriába. Az utasítás után a programozó automatikusan visszakerül az „Idle” alapállapotba.

3.7.4. Utasítások és utasítás sor példa és magyarázat.

Mint korábban említettem a programozónak összesen 7 fő állapotja van. Az alap állapot a base state. A base state-ból 6 állapotba lehet lépni. minden utasítás első bájtja, azaz első két hexadecimális karaktere azt határozza meg hogy melyik állapotba lépjen a programozó. Az állapotok és hozzájuk tartozó hexadecimálisan kifejezett bajt:

Állapot	Hexkód	Ascii ban
SPI configure	41	A
SPI read	42	B
SPI write	43	C
Get settings	44	D
I2C read	45	E
I2C write	46	F

3.1. táblázat. Állapotokhoz tartozó hex kódok

Irodalomjegyzék

- [1] [A MachXO2 FPGA család adatlapja](#)
- [2] [A AT24C256 EEPROM adatlapja](#)
- [3] [A W25Q64FV FLASH adatlapja](#)