

MISKOLCI EGYETEM



GÉPÉSZMÉRNÖKI ÉS INFORMATIKAI KAR
AUTOMATIZÁLÁSI ÉS INFOKommunikációs INTÉZET

**Konfigurálható EEPROM programozó tervezése
FPGA-ra**

KÉSZÍTETTE:

Szabó Ferenc Lőrinc

KONZULENS:

Bartók Roland

Automatizálási és Infokommunikációs Intézet Tanársegéd

Miskolc, 2025.

EREDETISÉGI NYILATKOZAT

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

Konfigurálható EEPROM programozó tervezése FPGA-ra

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 22.



(Szabó Ferenc Lőrinc)

Tartalomjegyzék

1. Rövidítések és angol kifejezések	6
2. Bevezetés	9
3. Feladat leírása, célok, követelmények	11
4. Irodalmi háttér, FPGA-k, a VHDL, kommunikációs protokollok	13
4.1. FPGA-k, és a VHDL	13
4.1.1. Mi a FPGA?	13
4.1.2. Mi a VHDL?	15
4.2. A NYÁK tervező program rövid bemutatása	17
4.3. Kommunikációs protokollok	18
4.3.1. Analízis képek magyarázata	18
4.3.2. UART	19
4.3.3. SPI	21
4.3.4. I2C	23
5. Tervezés	26
5.1. A terv bemutatása	26
6. Megvalósítás	27
6.1. Kapcsolási rajz tervezése	27
6.1.1. NYÁK tervezése	30
6.2. A Kész nyák	37

6.3. A Programozó alentitásai	38
6.3.1. Az Int_Osc entitás	38
6.3.2. A write8bit entitás	41
6.3.3. UART_TX entitás	42
6.3.4. A UART_RX entitás	43
6.3.5. A writePage entitás	45
6.3.6. A I2C entitás	48
6.4. A fő (MAIN) entitás	52
6.4.1. A main entitás állapotgép	52
6.5. C++-ban megírt utasítás küldő program	54
6.5.1. A Terminal app program	54
6.5.2. A "File app" program	55
6.5.3. A programozó beállítása a belső memória segítségével	55
6.5.4. Utasítások és utasítás sor példa és magyarázat.	56
7. Tesztelés	58
7.1. Tesztelési és analízis elrendezés	58
7.2. Tesztelés	60
7.2.1. SPI configure állapot tesztelése	60
7.2.2. SPI write állapot tesztelése	61
7.2.3. SPI read állapot tesztelése	63
7.2.4. I2C write és I2C read állapot tesztelése	65
7.2.5. Get settings állapot tesztelése	67
8. Összegzés	68
8.1. Magyar összefoglaló	68
8.2. Jövőbeli tervek	69
8.3. English summary	69
9. Melléklet	71

Ábrák jegyzéke

4.1.	vizualizált analizátor rögzítés példa	19
4.2.	UART blokkvázlat	20
4.3.	UART adatkeret	21
4.4.	SPI blokkvázlat	22
4.5.	SPI-al kiolvasott gyártó azonosító és típusszám	23
4.6.	I2C elrendezés példa blokkvázlat	24
4.7.	I2C 1 bájtos olvasás a programozával	25
5.1.	Projekt terv blokkvázlata	26
6.1.	A programozó kapcsolási rajza	27
6.2.	Ellenállás Jumper-ek	28
6.3.	Ellenállósokon lévő feszültség összefüggése	28
6.4.	VCC és VBUS közösítő jumper	29
6.5.	Fontos feszültségek és a GND kivezetése	30
6.6.	A programozó NYÁK-ja	30
6.7.	KiCad limit beállítások	31
6.8.	A nyák alsó rétege	32
6.9.	SPI kimenet	32
6.10.	I2C kimenet	33
6.11.	A rounded tracks bővítmény UI-ja	34
6.12.	Egy trace a bővítmény használata előtt	34
6.13.	Egy trace a bővítmény használata után	35

6.14. Példa egy csúnyán tervezett trace-re	35
6.15. Teardrop vias UI	36
6.16. Teardrop bővítmény használata előtt	36
6.17. Teardrop bővítmény használata után	37
6.18. A Kész nyák	37
6.19. A kész nyák összeszerelve és forrasztva	38
6.20. Elérhető CLK-frekvenciák az FPGA adatlapja alapján [7]	40
 7.1. Analízis elrendezés blokkvázlata	59
7.2. Analízis elrendezés kenyérszenellen	60
7.3. SPI configure állapot tesztelése rögzítés	61
7.4. SPI write állapot tesztelése rögzítés	63
7.5. SPI read állapot tesztelése rögzítés	64
7.6. I2C write állapot tesztelése rögzítés	66
7.7. I2C read állapot tesztelése rögzítés	66

Táblázatok jegyzéke

6.1. Állapotokhoz tartozó hexadecimális kódok	57
---------------------------------------------------------	----

1. fejezet

Rövidítések és angol kifejezések

- FPGA: Field Programmable Gate Array: Helyben programozható logikai kapumátrix.
- HDL: Hardware Description Language: Hardverleíró nyelv.
- VHDL: VHSIC Hardware Description Language: A HDL egyik elterjedt szabvánnya.
- LUT: Look Up Table: Kikereső tábla, amely logikai műveleteket valósít meg előre definiált értékek alapján az FPGA-n belül.
- CLB: Configurable Logic Block: Konfigurálható logikai blokk, amely flip-flopokat és LUT-okat tartalmaz.
- I/O: Input/Output: Bemeneti/kimeneti interfész, amelyen keresztül az eszköz adatokat fogad vagy továbbít.
- I2C: Inter-Integrated Circuit: Kommunikációs protokoll, jelentése magyarul: "IC-k közötti kapcsolat".
- SPI: Serial Peripheral Interface: Kommunikációs protokoll, jelentése magyarul: "Soros periféria interfész".

- UART: Universal Asynchronous Receiver-Transmitter: Kommunikációs protokoll, jelentése magyarul: "Univerzális aszinkron vevő-adó".
- EEPROM: Electrically Erasable Programmable Read-Only Memory: Elektronikusan törölhető és újraprogramozható nemfelejtő memória.
- ASIC: Application-Specific Integrated Circuit: Kifejezetten egy adott alkalmazásra tervezett integrált áramkör.
- VR: Virtual Reality: Virtuális valóság.
- Flash memory: Flashmemória: gyors nemfelejtő tárolóeszköz.
- CPLD: Complex Programmable Logic Device: Komplex programozható logikai eszköz.
- Bitstream: Bitfolyam: Az FPGA konfigurációs adatfolyama.
- Netlist: Network List: Kapcsolati lista, amely megadja az áramköri elemek közötti összeköttetéseket.
- CPU: Central Processing Unit: Egy számítógép agya, jelentése magyarul: "Központi feldolgozóegység".
- IC: Integrated Circuit: Integrált áramkör, amely több elektronikus alkatrészt tartalmaz egyetlen chipen.
- DRC: Design Rule Check: Tervezési szabályellenőrzés.
- ERC: Electrical Rule Check: Elektromos szabályellenőrzés.
- GND: Ground: Földelés
- CLK: Clock: Órajel.
- Trace: Egy nyákon rajzolt vezeték.

- COM port: Communication port: Kommunikációs port, soros interfész egy számítógépen.
- UI: User Interface: Felhasználói felület.

2. fejezet

Bevezetés

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

Az FPGA-k széles körben alkalmazottak. Egy FPGA-ban akár több százezer azonos logikai blokkot is 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 videofeldolgozás területén is. Számos fogyasztói elektronikai eszközben, például virtuális valóság szemüvegekben FPGA-k végezik 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. Az FPGA-k gyakran használtak tervezők által új dizájnok validálására, hibakeresésre, 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 témajának az FPGA-k és a VHDL hardverleíró nyelv mélyebb megismerését.

3. fejezet

Feladat leírása, célok, követelmények

Elsősorban szerettem volna egy összetett, de egyedül is megvalósítható beágyazott rendszert tervezni, amely során lehetőségem nyílik elmélyülni az FPGA-k és a VHDL hardverleíró nyelv használatában. A céлом az volt, hogy egy olyan eszközt hozzak létre, amely képes különböző típusú EEPROM, és más memóriák programozására két kommunikáció protokollt alkalmazásával, és amely rugalmasan alkalmazható különféle feladatokra, esetleg más IC-k felkonfigurálására.

Fontos szempont volt, hogy a projekt során ne csak a hardveres, hanem a szoftveres oldal is hangsúlyt kapjon, ezért döntöttem úgy, hogy a kommunikációs és vezérlő szoftvert C++ nyelven írom meg. Ezzel egyrészt bővíthettem a programozási ismereteimet, másrészt egy jól használható, parancssoros interfészt tudtam biztosítani a programozónak.

A projekt során kiemelt cél volt a NYÁK (nyomtatott áramkör) tervezésének gyakorlása is, hiszen ez a villamosmérnöki szakmában alapvető kompetenciának számít, és minden modern beágyazott rendszer alapja a megfelelő áramköri integráció.

Tehát a konkrét kitűzött célok/követelmények:

- Egy Windows-ra írt program, ami képes egy felhasználótól parancsokat értelmezni és azok alapján parancsokat küldeni egy COM port-on.

- Egy VHDL-ban megírt hardver, ami FPGA-n valósul meg, és képes a COM portról érkező, UART-ba alakított parancsokat értelmezni.
- Az FPGA-s hardver része legyen egy SPI és I2C mester.
- Az FPGA-s hardver legyen képes parancsok alapján SPI és I2C üzeneteket írni és olvasni.
- Az FPGA-s hardver legyen képes beállító parancsokat fogadni, amelyekkel a SPI és I2C keretek különböző részeinek hosszúságát a felhasználó állíthatja.
- Az FPGA-s hardvernek legyen reset logikája.
- Az FPGA-s hardver legyen képes a SPI és I2C vonalakon érkező adatokat UART kommunikációkkal visszaküldeni a felhasználónak.
- A Windows-os program legyen képes a kapott adatokat rögzíteni (log-olni).
- A programozó működése teszteléssel legyen bemutatva tényleges memóriaegységekkel
- A programozó rendszernek legyen egy NYÁK-ja.

4. fejezet

Irodalmi háttér, FPGA-k, a VHDL, kommunikációs protokollok

4.1. FPGA-k, és a VHDL

4.1.1. Mi a FPGA?

Az FPGA, magyarul: helyben programozható logikai kapumátrix, olyan integrált áramkör, amelyet a felhasználó a gyártás után konfigurálhat. A digitális áramkörök tervezése HDL nyelveken történik, és a megvalósítás fizikailag a chip-ben belül történik.

Az FPGA-k logikai cellákból épülnek fel, amelyek flip-flop-okat, LUT-okat és logikai kapukat tartalmaznak. Ezek a cellák tetszőlegesen összekapcsolhatók, így hozhatók létre a kívánt áramkörök. A konfigurációt, amely az FPGA-ban megvalósított hardver működését meghatározza, gyakran firmware-nek nevezik, bár a kifejezés eredetileg más jelentéssel bírt. Itt arra utal, hogy a "programozás" révén egy valós, hardveres működésű áramkör jön létre. Más megnevezés is elterjedt, például "design", vagy "gateware". A dolgozatban én legfőbbképpen a design angol szót használom, ha a HDL-ben írt "kódról" beszélek, és a hardver kifejezést, ha már a konfigurált FPGA-ról beszélek.

Az FPGA-k elterjedtek mind fogyasztói és ipari termékekben, mind fejlesztési környezetekben, mint például az ASIC termék tervezéséhez. Előnyük, hogy speci-

fikus feladatokat hatékonyan képesek végrehajtani, és működésük firmware frissítéssel módosítható. Például egy VR-eszközben az FPGA gyorsan és pontosan dolgozza fel a videó jelet, minimalizálva a késleltetést a fejmozgás és a képfrissítés között. Az ilyen ismétlődő, párhuzamos műveletekben az FPGA hatékonyabb, mint egy általános célú CPU, amelyet sokféle feladatra terveztek, de nem optimalizáltak ezekre a számításokra.

Egy szemléletes hasonlat: míg egy ember képes kenyérsütesi utasításokat követni, egy erre tervezett gép gyorsabban és hatékonyabban végzi el ugyanazt a feladatot. Ugyanakkor a gép nem alkalmas más típusú feladatokra írt utasításokat értelmezni, például villanykörte becsavarására. Erre az új feladathoz egy új gép tervezése szükséges. A hasonlatban a személy a CPU, az utasítássor a hagyományos kód, míg a gépek az FPGA-n megvalósított designek.

A modern FPGA-k többsége SRAM-alapú, és konfigurálható logikai blokkokat, programozható I/O blokkokat, valamint útválasztó mátrixot tartalmaz. A konfigurációt a chipen belüli SRAM tárolja. De vannak más megoldások is, például a munkahez-lyemen találkoztam egy másik gyakran használt megoldással a magas szintű FPGA-s termékek közt: egy CPLD, egy külső memória chipből konfigurált egy FPGA-t a termék indításakor

A programozható összeköttetések fizikai jellemzői (ellenállás, kapacitás) befolyásolják a teljesítményt, különösen nagy kihasználtság esetén. A végső elérhető órajel sebesség gyakran csak a "place and route" után becsülhető meg pontosan. Ezért fontos, hogy pontosan melyik szintetizálási programot használunk. A hatékony netlist generálás és fejlett szintetizáló eszközök kulcsfontosságúak az optimális működés eléréséhez, főleg, amikor a tervezett design közelíti a FPGA használtságának 100%-át. Mivel ahogy a felhasználás közelíti ezt az értéket, a jel útválasztás jelentősen nehezebbé válik.

Használt forrás: [3]

4.1.2. Mi a VHDL?

A VHDL napjaink egyik legszélesebb körben alkalmazott hardverleíró nyelve (HDL), amely a Verilog mellett ipari szabványnak számít. A VHDL különlegessége, hogy nem csupán egy "programozási" nyelv, hanem egyben egy, az IEEE által karbantartott és frissített szabvány is, amely folyamatosan igazodik az ipari igényekhez.

A VHDL nyelvben minden hardverkomponens egy entitás, amely jól definiált bemenetekkel és kimenetekkel rendelkezik. Az entitás lehet egészen egyszerű, például egy logikai kapu, de akár komplex alrendszer, mint egy teljes SPI vagy I2C kommunikációs vezérlő is. Egy-egy entitáson belül további alentitások integrálhatók. Ez hierarchikus és moduláris tervezési szemléletet követel, amely alapvető a nagyobb rendszerkomplexitások kezelésében.

A HDL-ek lehetővé teszik digitális áramkörök működésének absztrakt, magas szintű leírását, amelyből szintetizálás útján valós, FPGA-n megvalósítható hardver születhet. A nyelv használatának egyik fő kihívása, hogy nem elegendő csupán a szyntaxt és a nyelvi elemeket elsajátítani. Elengedhetetlen annak ismerete is, hogy a különböző fejlesztői eszközök és szintetizátorok, hogyan értelmezik a leírt kódot. Gyakori tapasztat, hogy egy szimulációban helyesen működő terv nem feltétlenül viselkedik ugyanúgy a szintetizált, fizikai környezetben.

Használt forrás: [2]

HDL-ben digitális rendszerek tervezéséhez a munkafolyamat:

- (a) Maga a design megtervezése és leírása a HDL nyelvben
- (b) A testbench megírása, ami a szimulációhoz szükséges.
- (c) Szimulátor program használata szimulációhoz, hibakeresés.
- (d) A cél hardverre design szintetizálása, hibakeresés.

A testbench alapvető feladata a szimuláció során a rendszer bemeneteinek előállítása, valamint a kimenetekre adott reakciók pontos leképezése. Például egy I2C mester implementációjának esetében a testbench egy alarendelt egységként működhet, ezzel

segítve a rendszer funkcionálisának ellenőrzését. A VHDL nyelv tartalmaz olyan, szintetizálhatatlan nyelvet is, melyek célja, hogy a tesztkörnyezet létrehozása során egyszerűbbé és gyorsabbá tegyék a fejlesztést. Ezzel szemben a nyelv azon részeit, amelyek szintetizálhatók, RTL-nek (Register Transfer Level) nevezik, és ezek képezik a végleges hardverimplementációt.

Használt forrás: [1]

Szimulálás és hibakeresés után következik a szintetizálási folyamat, ahol VHDL-ben megírt leírásból fizikai hardverstruktúrát állít elő. A cél, hogy a logikai tervet az adott cél-FPGA architektúrájához optimalizálva valós, működőképes hardverelemekre fordítsuk le. A folyamat során az alábbi lépések történnek:

1. A szintetizáló eszköz először egy hardver független netlist-et generál a megadott VHDL leírás alapján. Ez a netlist logikai kapuk és kapcsolatok absztrakt reprezentációja.
2. Ezt követően a netlistet leképezi az adott FPGA technológia konkrét erőforrásaira, például look-up táblákra (LUT), flip-flop okra, szállító multiplexer-ekre és belső routing hálózatra.
3. Az eszköz egy modell-specifikus optimalizációt hajt végre, amely során figyelembe veszi az időzítési követelményeket, az erőforrás használatot és a jelutak minimális késleltetését, így jön létre az FPGA-ra illesztett végleges netlist.
4. Ezt követi az időzítés elemzése, amely az időzítési korlátok betartását vizsgálja. A "slack" értékek és a kritikus útvonalak elemzése alapján meghatározható, hogy a rendszer képes-e a megadott órajel sebességgel megbízhatóan működni. Ha szükséges, a jeleket regiszteres úton szinkronizálják újra a stabil működés érdekében.
5. Végül generálódik a konfigurációs állomány (bitstream), amely a programozó eszköz segítségével betölthető az FPGA memóriájába (pl. SRAM), lehetővé téve a fizikai áramkör felprogramozását és működtetését.

Az időzítési analízis során alkalmazott technikák különösen fontosok akkor, amikor a tervezett rendszer közelíti az adott FPGA erőforrásainak maximális kihasználtságát.

4.2. A NYÁK tervező program rövid bemutatása

A KiCad egy ingyenes, nyílt forráskódú szoftvercsomag. Elsősorban kapcsolásai rajzok és nyomtatott áramkörök tervezésére szolgál. Egy KiCad projekten 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 nyák 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 nyákról.

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 én 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.

4.3. Kommunikációs protokollok

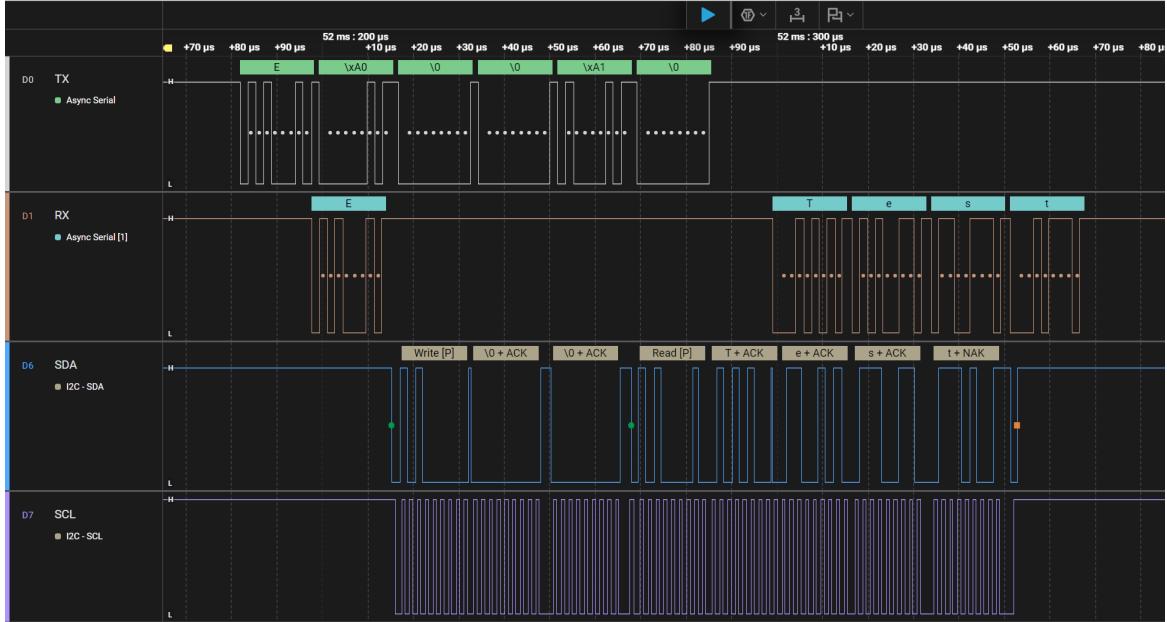
A kommunikációs protokoll egy szabályrendszer, amely meghatározza, hogyan cserélnek információt az eszközök egy adott kommunikációs csatornán keresztül. Ezek a szabályok tartalmazzák az adatcsomagok szerkezetét, az átvitel sorrendjét, a hibakezelést, valamint a szinkronizációs és vezérlőjeleket is. Az adatátvitelhez elengedhetetlen, hogy a feladó és a fogadó ugyanazt a kommunikációs protokollt használja. Ha a feladó egy adott protokoll szerint kódolja az adatot, de a fogadó nem ismeri ezt a szabványt, az információ nem értelmezhető. A protokoll meghatározza az adat kódolásának és dekódolásának szabályait, valamint a kommunikáció fizikai csatornáját is.

A digitális rendszerekben számos különböző kommunikációs protokoll létezik, a programozóm az UART-ot, SPI-t és I2C-t használja. A következőkben röviden bemutatom ezeket a protokollokat, kiemelve működésük alapelveit és a projekt szempontjából releváns tulajdonságait.

4.3.1. Analízis képek magyarázata

A diplomamunkámban sok képet használok, amit analizátorral rögzítettem. Használtam 2 memória modult is, egy I2C EEPROM-ot meg egy SPI Flash-t is, hogy tudjam ellenőrizni programozó működését, illetve, hogy a rögzítésekben látszódjon a kommunikáció válasza. Ezekről van több szó a "tesztelési és analízis elrendezés" szekcióban.

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



4.1. ábra. vizualizált analizátor rögzítés példa

4.3.2. UART

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

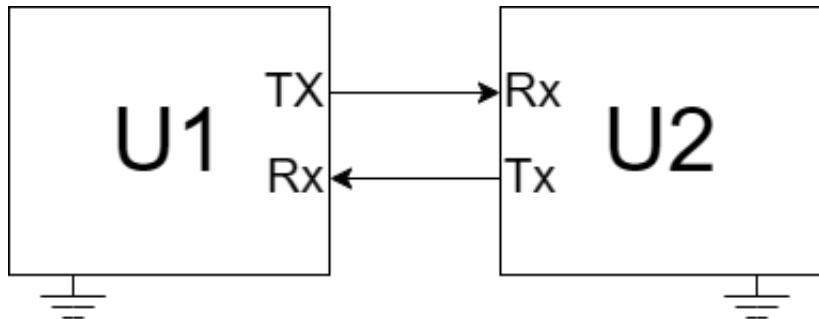
Az UART (Universal Asynchronous Receiver Transmitter) egy kommunikációs protokoll, amely aszinkron, teljes duplex és soros. Az, hogy soros azt jelenti, hogy az adatátvitel bitenként történik, nem párhuzamos vonalokon. Az, hogy aszinkron azt jelenti, hogy nem létezik külön órajel vonal; az időzítést a küldő és a fogadó egymástól függetlenül szabályozza. És az, hogy teljes duplex azt jelenti, hogy két irányú adatátvitel valósul meg egyidejűleg. Az adatokat "keretek" alakjában továbbítja, ahol egy tipikus adatkeret tartalmaz egy kezdő (start) bitet, adatbiteket, opcionális paritásbitet a hiba-kezeléshez, majd egy stop bitet. A start és stop bitek segítik a fogadó eszközt abban, hogy pontosan meghatározza az adatcsomagok kezdetét és végét, és ezáltal megbízható adatátvitelt biztosítanak.

A Baud ráta a kommunikációs rendszer azon paramétere, amely megadja, hogy másodpercenként hány szimbólum van a vonalon küldve, azaz hány darab jelváltás

történik a vonalon. A beállított Baud ráta alapján van a fogadó vonal mintavételezés, és az küldő vonalon a jel váltás időzítve.

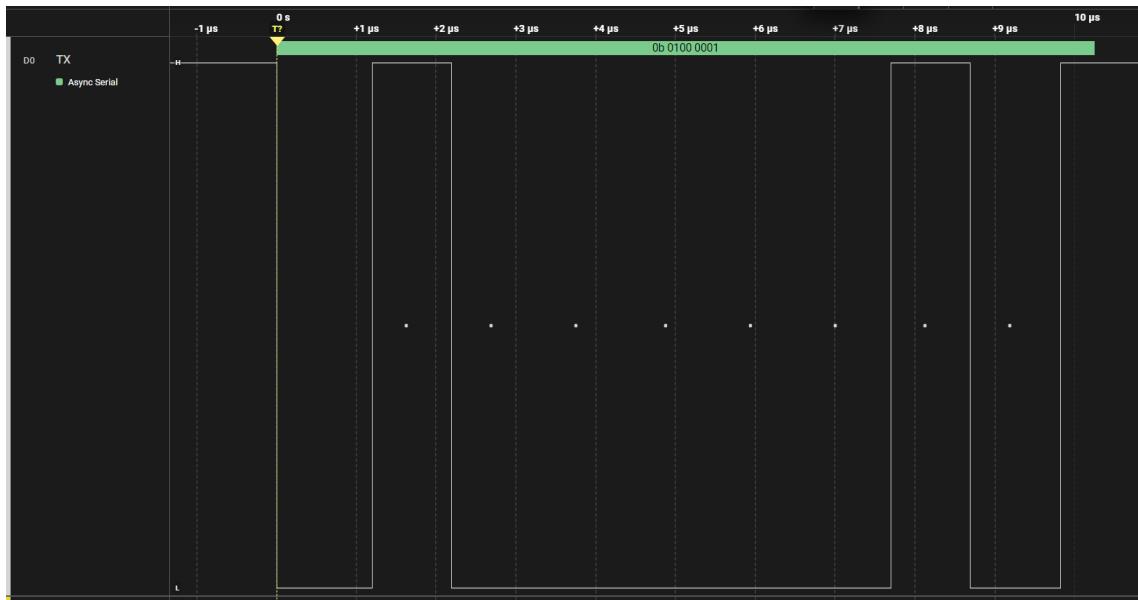
Az UART egyszerűsége és hatékonysága miatt elterjedt megoldás a mikrovezérlők, számítógépek és egyéb beágyazott rendszerek közötti kommunikációban.

Használt forrás: [4]



4.2. ábra. UART blokkvázlat

A fenti blokkvázlaton egy UART elrendezés van rajzolva. A TX jelölés a "Transmit" (küldés) jelölésé, míg az RX a "Receive" (fogadás) jelölésé. Nem szükséges minden vonalat használni, lehet szimplex, fél-duplex vagy teljes duplex a kommunikáció. Szimplex lenne, ha az csak a U1 Tx és U2 RX lába között lenne kapcsolat, ebben az esetben az adatátvitel egyirányú lenne, tehát csak küldésre lenne lehetőség. Fél- vagy teljes duplex konfiguráció esetén az áramkört a mellékelt ábrának megfelelően kell elrendezni. Ha az U1 és U2 képes egyidejűleg adatot küldeni és fogadni, a kommunikáció teljes duplex-ként működik, ha nem akkor fél duplex. A programozón teljes duplex a kommunikáció.



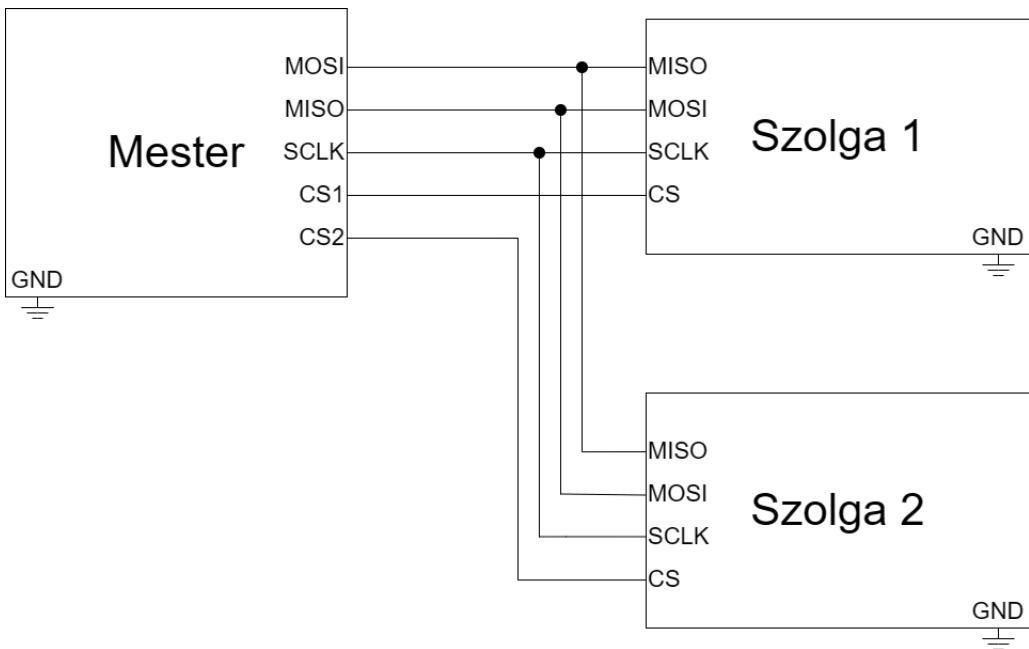
4.3. ábra. UART adatkeret

A fenti képen egy adatkeret szerepel. Ezt az adatkeretet az FPGA küldte a programozónak. A Baud ráta ebben az esetben 921600 Baud, egy szimbólum ideje: $1/921600 \text{ Baud} = 1.09 \mu\text{s}$. Ez az analizátor program segítségével látható a képen.

Látható a start bit. A start bit után következik 8 adatbit, amik a programozó által küldött parancsot tartalmazzák. És végül a stop bit után a vonal visszatér a tétlen állapotba.

4.3.3. SPI

Az SPI (Serial Peripheral Interface) egy széles körben használt és rugalmas kommunikációs protokoll. Teljes duplex, és szinkron módon működik, az UART-tal ellentétben.



4.4. ábra. SPI blokkvázlat

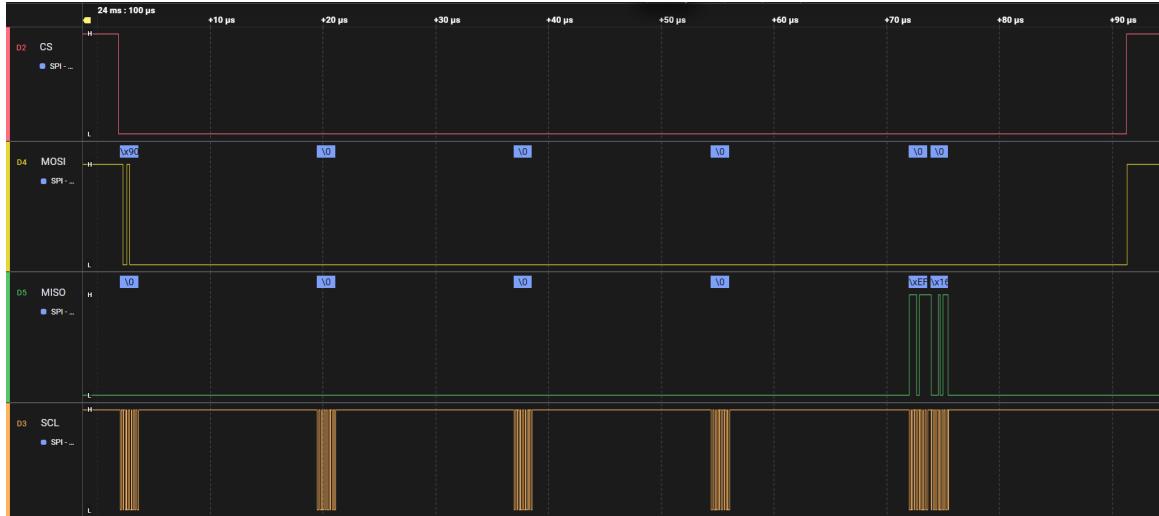
Az SPI protokoll előnyé, hogy lehetővé teszi, hogy több IC megossza ugyanazokat a kommunikációs vonalakat, de egyszerre csak egy mester és egy szolga lehet aktív. minden szolgához külön CS vonal szükséges, míg az egyéb vonalak (MISO, MOSI, SCLK) közösek lehetnek. A mester választja ki az adott szolgát az adott szolga chip select vonalán keresztül, és indítja a kommunikációt. A szolga kizárolag kiválasztott állapotban képes adatátvitelre. Ennek megfelelően egy szolgának négy, a mesternek három lábra, plusz egy további lábra per darab szolga, van szüksége.

Az SPI szinkron protokoll. Ezért kell az SCLK, vagyis az órajel vonal. Ez azt jelenti, hogy nincs előre beállított adatátviteli sebesség. A mester előállít egy órajelet a SCLK vonalon, a szolgák pedig az órajel éleit figyelve mintavételeznek és küldenek adatot. Így az adatátviteli sebesség az órajel frekvenciájától függ. Emiatt az órajelnek nem kell a kommunikáció alatt ugyan azon a frekvencián maradni. Ezt ki is használom a programozómban. Az órajel ki van "nyújtva" és megállítva amikor a programozó várja a következő adatot a felhasználótól az UART vonalon.

A MISO magyarul "mester be szolga ki", a MOSI pedig "mester ki szolga be". SPI-t használó IC-k lábai lehetnek jelölve SDI-ként (soros adat be) meg SDO-ként (soros

adat ki).

Használt forrás: [6]

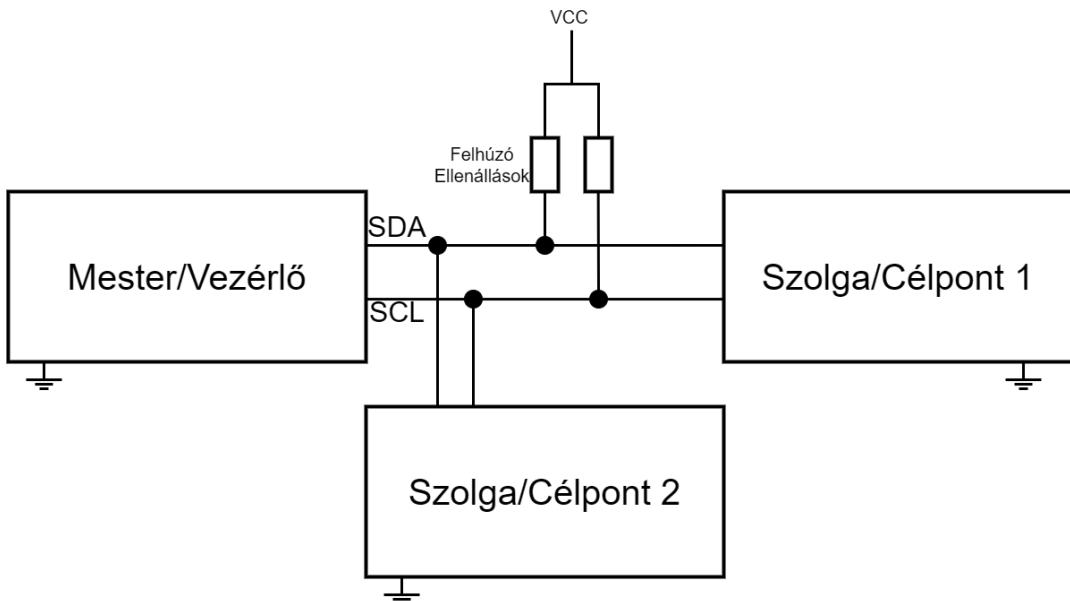


4.5. ábra. SPI-al kiolvasott gyártó azonosító és típusszám

Itt látható egy példa egy SPI kommunikációra. Ez a kommunikáció a programozóm és egy W25Q64JV Flash memória között történt. A termék gyártó azonosítóját (xEF) és típusszámát (x16) olvasom ki [9]. A CS vonal kiválasztja a Flash modult, majd kezdődik a kommunikáció. Ahhoz, hogy kiolvassuk az azonosítókat a modulnak a x90 instrukciót kell küldeni, majd 3 bájtin x00 címet [9]. Látszik az órajel nyújtás is, ami azért kell, mert a programozó várja a következő írandó bájtot az UART vonalon a PC-től. Többet írok erről a következő fejezetekbe. Amint az olvasás kész, és a programozó elküldte az adatot a PC-nek UART-on, a CS vonal visszalép magas szintbe és véget ér a kommunikáció.

4.3.4. I2C

Az I2C egy soros, szinkron, kétvezetékes kommunikációs protokoll. Az I2C célja, hogy egyszerű és hatékony adatcserét tegyen lehetővé rövid távolságon belül egy mester, másnéven vezérlő, és több szolga, másnéven célpont, eszköz között. Az egyszerűsége miatt széles körben alkalmazott.



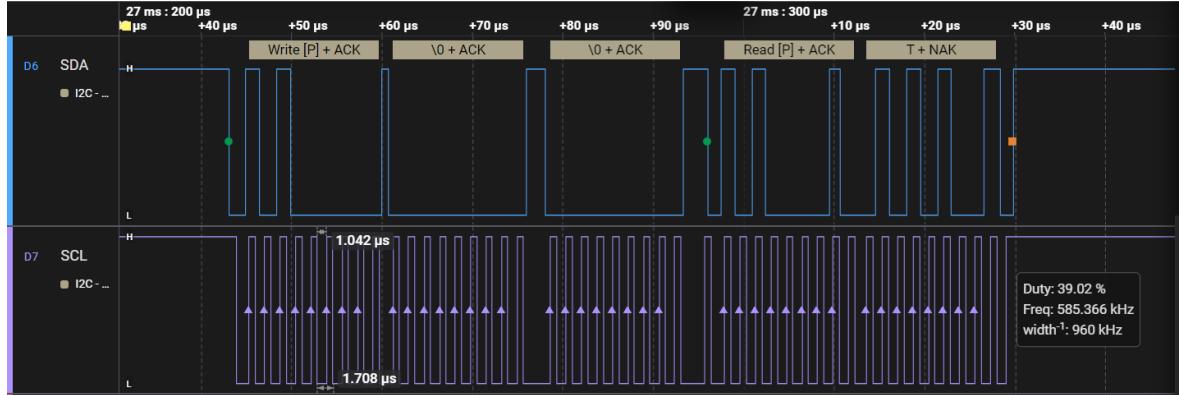
4.6. ábra. I2C elrendezés példa blokkvázlat

A fenti képen egy általam rajzolt példa elrendezés látható. A protokoll két vonalat használ: az SDA, magyarul soros adat, és az SCL, magyarul soros órajel, vezetéket. A szabvány szerint a vonalak nyitott kollektoros kialakításúak, ezért minden eszköz csak lehúzni tudja őket. Felhúzó ellenállások biztosítják az alapértelmezett magas szintet. Az I2C szinkron protokoll, vagyis az adatátvitel órajel vezérelt; az SCL vonalat mindig a mester generálja, így a tempót is ő szabja meg.

A protokoll a cím-alapú kommunikáción alapul: minden szolga eszköz rendelkezik egy egyedi címmel, és a mester ennek megfelelően kezdeményezi az adatcserét. Ezért nem kell minden szolgának egy külön CS vonal, mint az SPI-nál.

A programozóban az I2C protokollt úgy implementáltam, hogy az órajel nyújtás is lehetséges legyen. Ez lehetővé teszi, hogy a programozó várjon a felhasználói parancsra UART-on keresztül, mielőtt az I2C kommunikáció folytatódna.

Használt forrás: [5]



4.7. ábra. I2C 1 bájtos olvasás a programozóval

A fenti analizátor képen egy egybájtos I2C olvasási művelet látható, amely az EEPROM memóriamodul és a programozó közötti adatcserét rögzíti.

Az adatátvitel egy START jellel kezdődik: a programozó először a SDA vonalat logikai '0'-ra húzza majd a SCL vonalat is. Utána folytatódik a kommunikáció írási művelettes: a programozó az EEPROM címét (7 bites cím + írási bit) továbbítja. A memória eszköz ACK jellel válaszol. Ezután a programozó elküldi a memóriacímét (x0000), amely a kívánt olvasási cím regisztere. Ezt is ACK követi. Ez egy "dummy write" [8], ami azért szükséges, hogy az EEPROM betöltsé az adott regiszter értékeit, hogy később ki tudja őket küldeni.

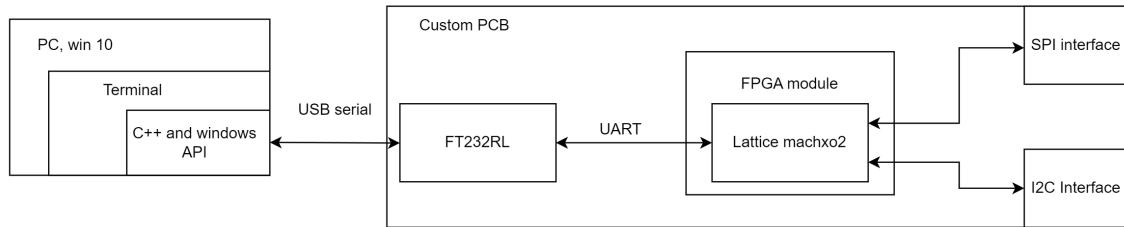
A címzés után a programozó ismételt START jelet küld, egy úgynevezett "repeated START" [8] jelet. Majd újra elküldi az EEPROM címét, ezúttal olvasási műveletet (read bit = 1) kérve. A szolga ACK jelzéssel visszaigazolja a fogadást, majd elküldi a kívánt adatbájtot. A programozó egy NAK jellel zárja az adatátvitelt, jelezve, hogy nem kíván további bájtokat olvasni, majd STOP jellel befejezi a kommunikációt: a SCL Vonalat logikai magasba engedi, majd a SDA vonalat is.

A SCL vonalon a mért frekvencia 585 kHz, ami a programozó beállított belső órajeléből számított értéknek megfelel. A duty cycle 39.02%, ez az órajelnyújtás, illetve a logikai műveletek időzítése miatt jelentkezik meg. A programozóban a I2C sebessége közelebb van az UART sebességéhez, mint az SPI sebessége, ezért nem olyan dramatikus az órajelnyújtás, mint az SPI rögzítésen látszódott.

5. fejezet

Tervezés

5.1. A terv bemutatása



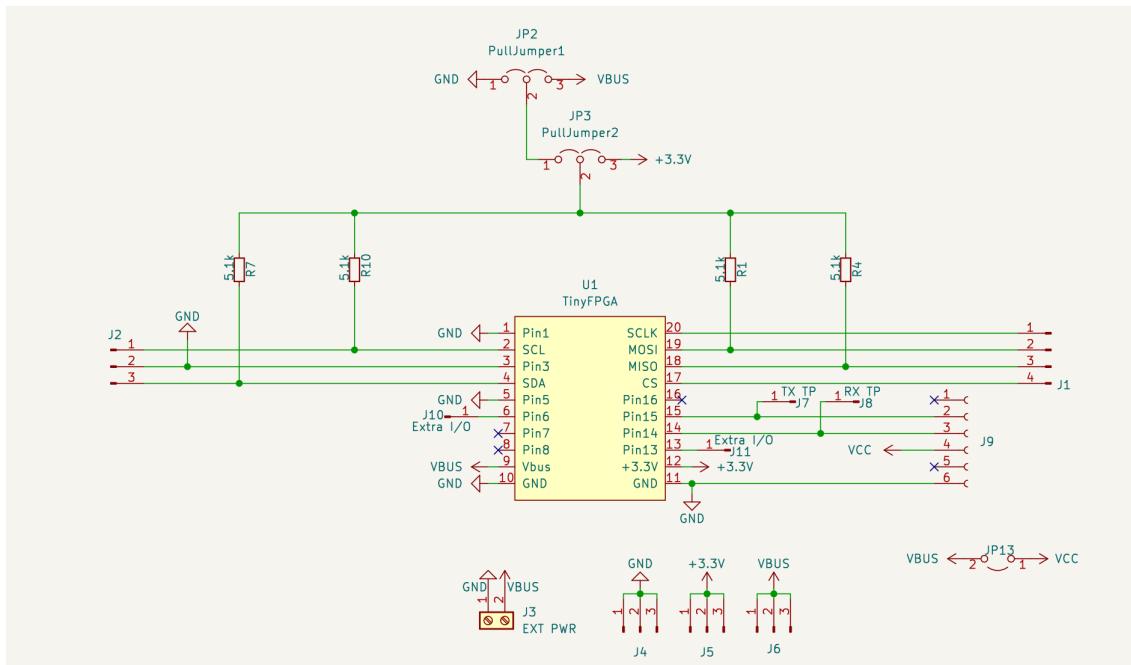
5.1. ábra. Projekt terv blokkvázlata

Ez a blokk diagramm volt a project kiinduló terve. A felhasználó a C++ -ban írt program futtatásával tud kommunikálni a programozóval terminálon keresztül. A programtól kapott USB kommunikációt a FT232RL váltja át UART csomagokká az FPGA modul számára. Az FPGA modulon megvalósított VHDL-ben megírt logikai áramkör pedig a kapott parancsok alapján küld SPI és I2C parancsokat a programozandó EEPROM chipnek. Az EEPROM chip esetleges válaszait pedig visszakonvertálja UART csomagokká az FT232RL-nek, amely visszaküldi a PC-nek.

6. fejezet

Megvalósítás

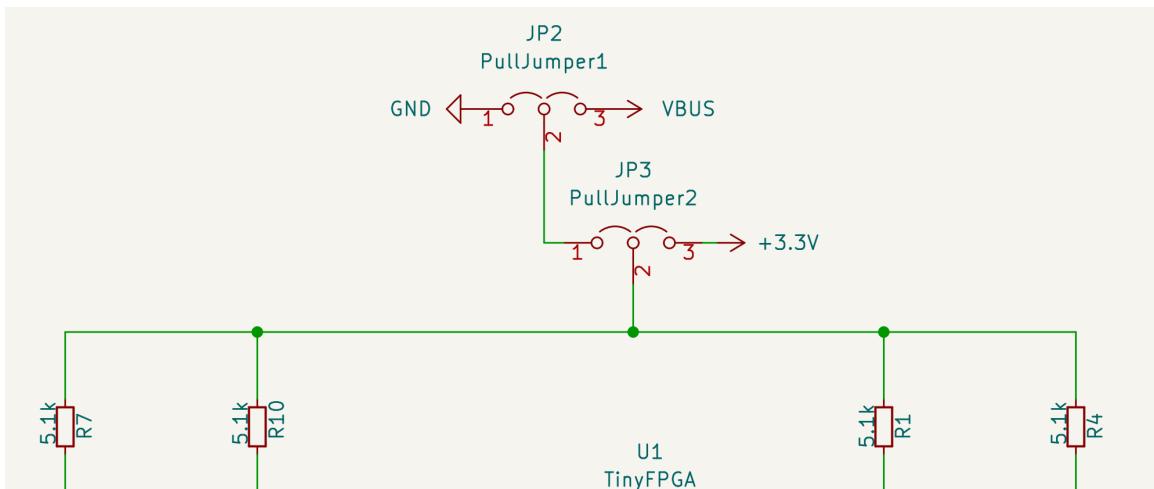
6.1. Kapcsolási rajz tervezése



6.1. ábra. A programozó kapcsolási rajza

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

A céлом az volt, hogy ez egy általánosan használható EEPROM programozó legyen, ehhez rugalmasnak kellet megterveznem a nyákot. A jumper-ek segítségével az ellenállások 3V3 vagy 5V felhúzó ellenállások, vagy akár lehúzó ellenállások ként is tudnak szolgálni. Így sok féle memória modult tud támogatni az áramkör. Ha a JP2 jumper 1-es és 2-es pin-je van közösítve, akkor 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 3.3V van kiválasztva.



6.2. ábra. Ellenállás Jumper-ek

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

6.3. ábra. Ellenállósokon lévő feszültség összefüggése

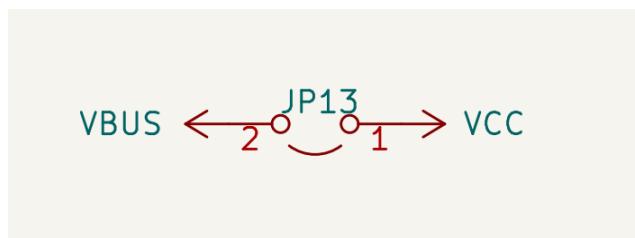
Az áramkörnek 4 lehetséges tápja van

- A TinyFPGA 5V VBUS PIN-je. A pin közvetlen össze van kötve a modul mikró USB 5V tápjával. Tehát amikor a TinyFPGA modul csatlakoztatva van a mikró

USB-n keresztül, a VBUS pin kimenet ként szolgálhat. A TinyFPGA modul a betápját is a VBUS látja el. Szóval, ha a TinyFPGA nincs csatlakoztatva az USB-hez akkor a VBUS pin bemenetként is működhet.

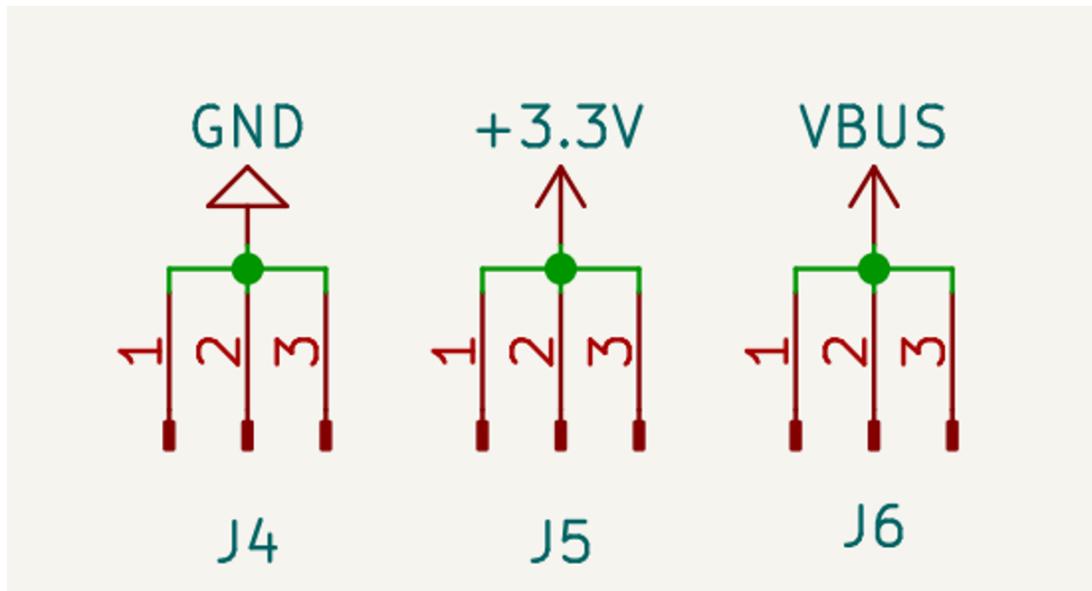
- A TinyFPGA 3V3 PIN-je. Ezt a kimenetet a TinyFPGA modul LDO IC-je generálja a VBUS 5V feszültégéből.
- Az FTDI232 VCC kimenete. Ami állítható 3V3 és 5V között a FTDI232 modulon egy jumper-el.
- Az EXT PWR csatlakoztatás, amit biztonság kedvéért raktam hozzá az áramkörhöz, ha szükséges lenne a jövőben egy külső táp.

A Jumper-ek úgy vannak megtervezve, hogy bármijeik 5V-os bemenet szolgálhasson tápként a nyáknak meg a hozzá csatolt memória moduloknak. 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.



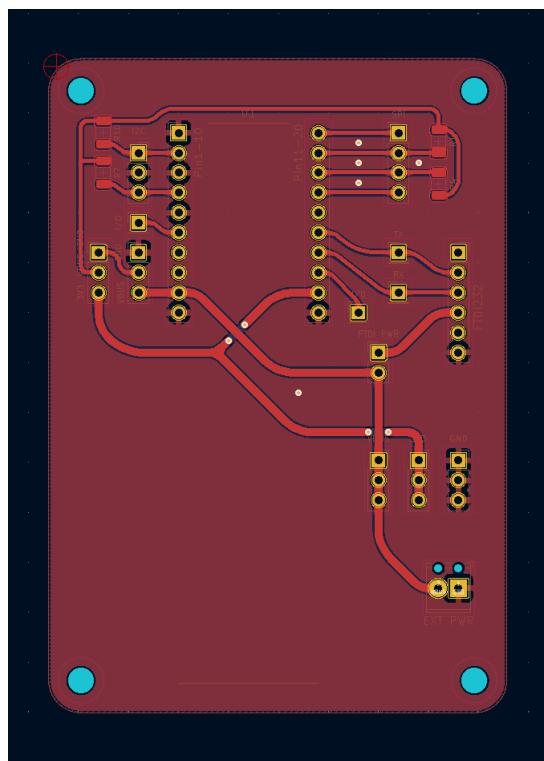
6.4. ábra. VCC és VBUS közösítő jumper

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



6.5. ábra. Fontos feszültségek és a GND kivezetése

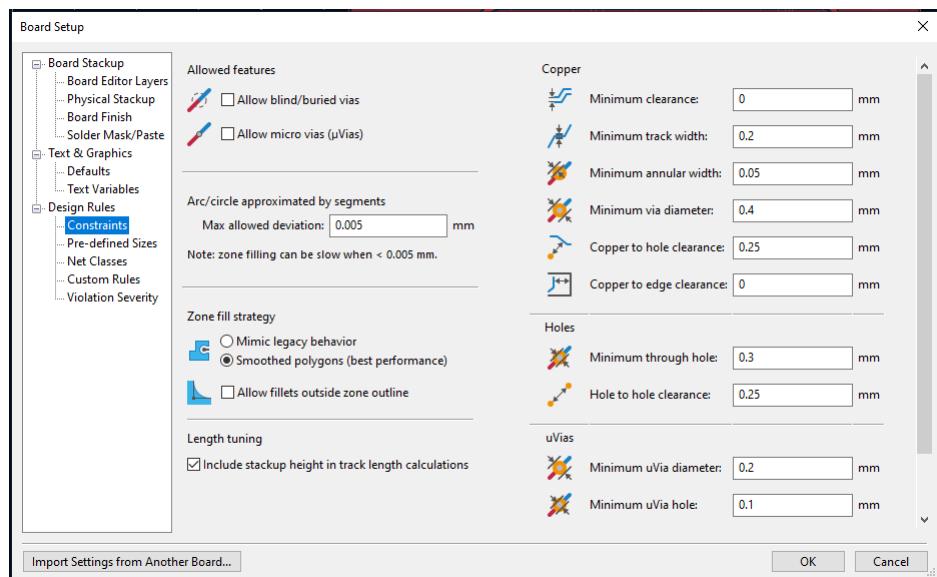
6.1.1. NYÁK tervezése



6.6. ábra. A programozó NYÁK-ja

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

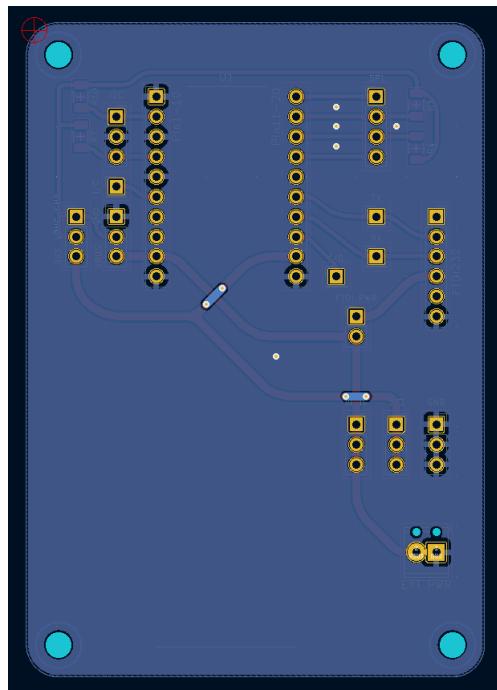
Tervezés előtt érdemes beállítani azokat a limiteket, amiket a nyák gyártó cég képes gyártani. Ha tudjuk, hogy melyik céget akarjuk használni, akkor a weboldalukon megtalálhatjuk az értékeket. A limiteket a "Board Setup" funkcióval lehet megadni. Ha jól meg vannak adva az értékek akkor a KiCad nem enged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



6.7. ábra. KiCad limit beállítások

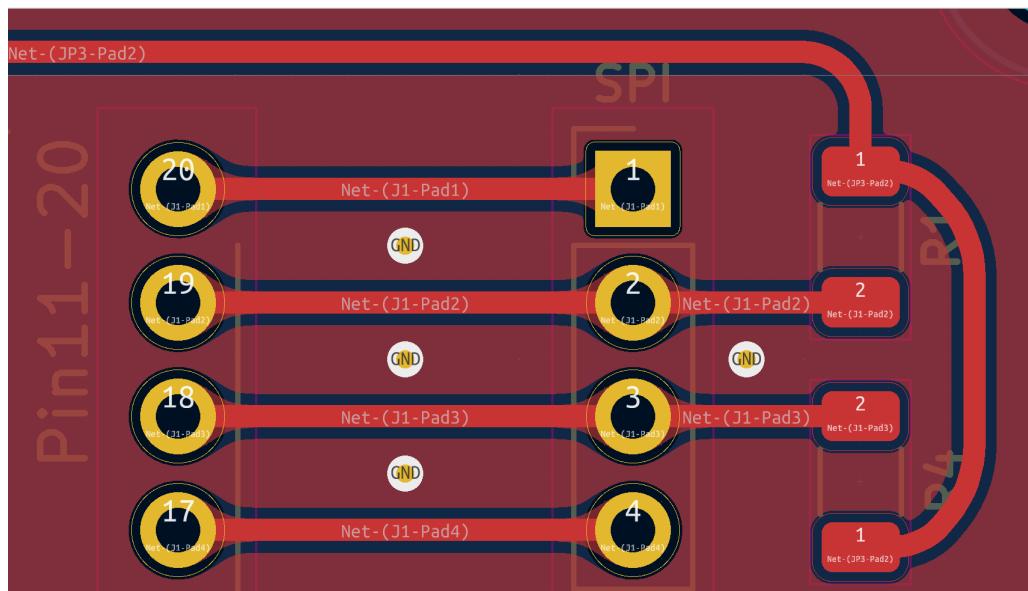
A NYÁK alsó rétege

A hátsó réteg itt egyszerű. Az egész egy GND réteg, minél kevesebb megszakítással, hogy minimalizálva legzenek az áram visszaújtjai, és ezáltal az arámhurkok területeit is.



6.8. ábra. A nyák alsó rétege

Az SPI csatlakozó

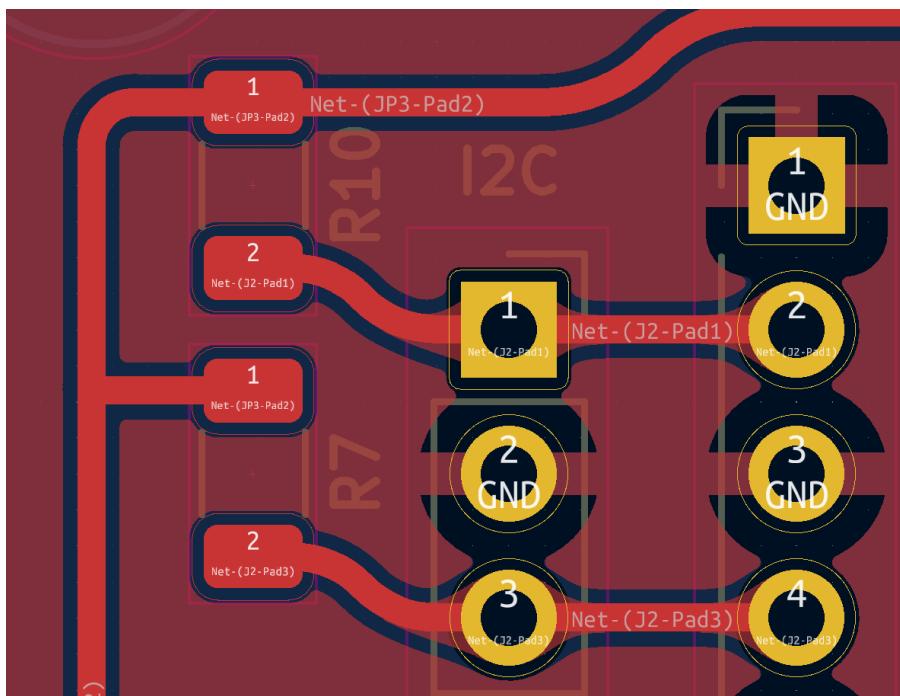


6.9. ábra. SPI kimenet

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 minden 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 árnyékoltva, ez segíti, hogy a vonalak elektromágneses zajai kevésbé befolyásolják egymást.

Az I2C csatlakozó



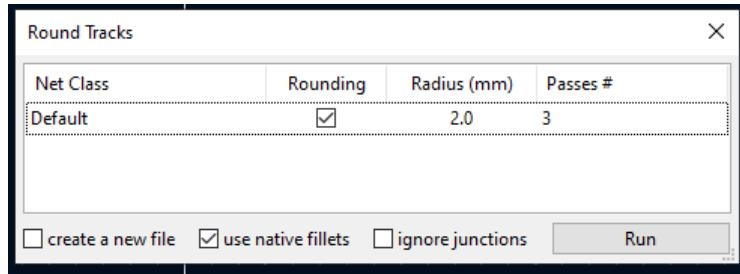
6.10. ábra. I2C kimenet

Itt is inkább túlterveztem a kimenetet, mint alul. Itt is a SCL és SDA vonal "length match"-elne van. Illetve az árnyékolást még a csatlakozókra is kiterjed. Így egészben a FPGA lábaihoz az árnyékolást végig lehet vinni, ha a SCL és SDA-hez tartozó lábak közötti lábat logikai '0'-ra állítom.

Bővítmények

Én két bővítményt használtam. Az egyik a rounded tracks bővítmény. Ami igazából csak esztétikai változásokat kreál a nyákon. A bővítményt nagyon egyszerű használni.

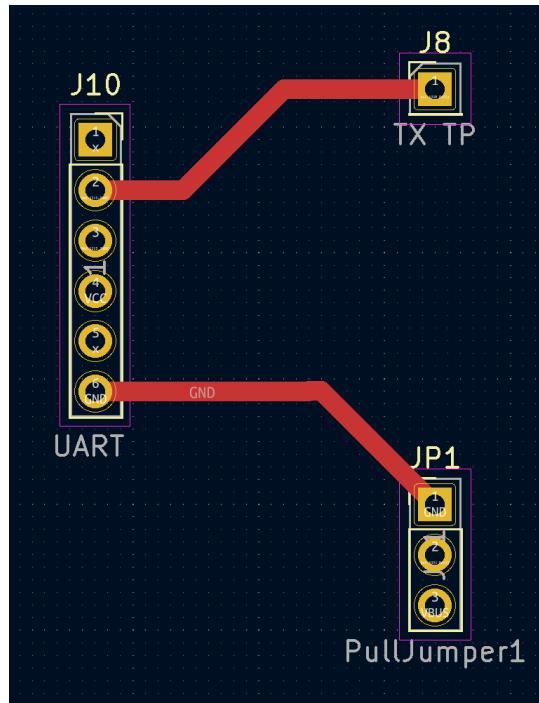
A következő kép a UI-ja.



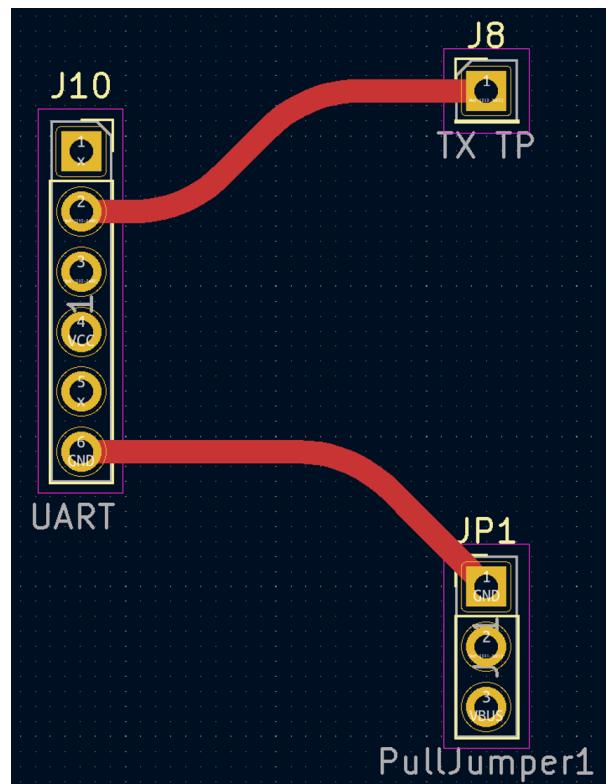
6.11. ábra. A rounded tracks bővítmény UI-ja

A UI egyszerű. Netlista kent ki lehet választani, hogy kerekítse a bővítmény a netlistához tartozó trace-eket vagy nem. Ki lehet választani célzott rádiumszt, illetve, hogy hányszor fusson a programm. A "run" gombbal kell elindítani.

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

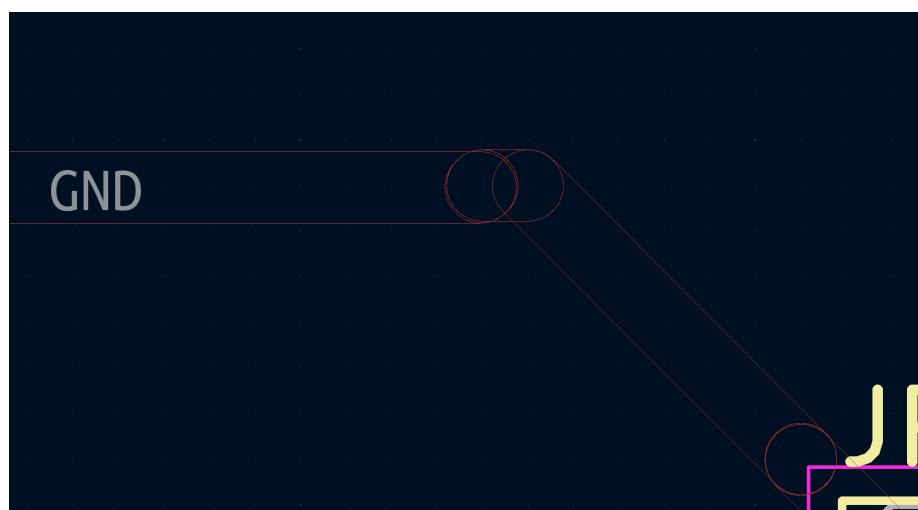


6.12. ábra. Egy trace a bővítmény használata előtt



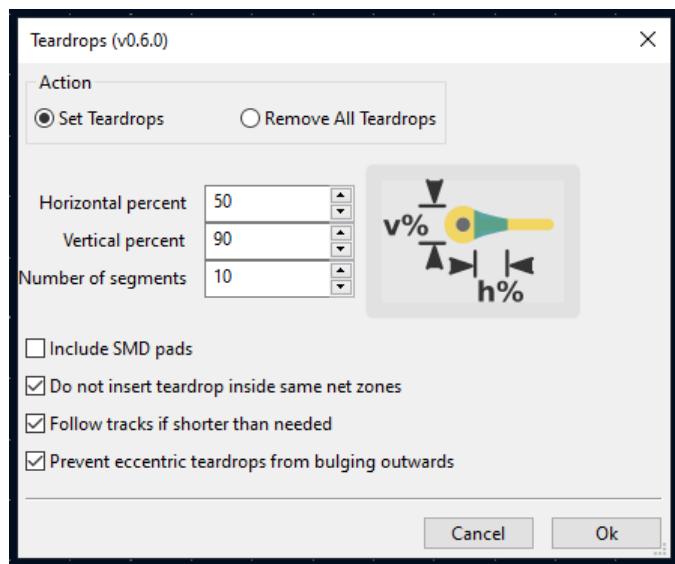
6.13. ábra. Egy trace a bővítmény használata után

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



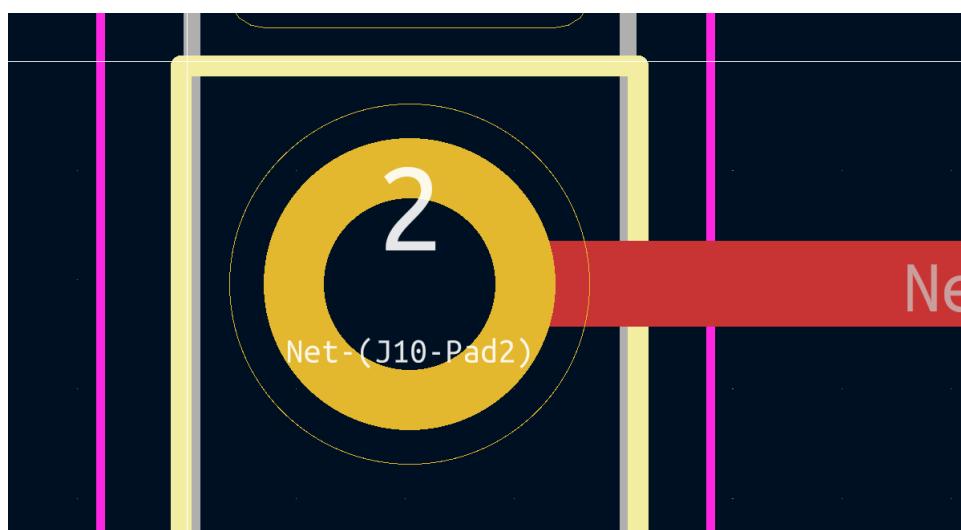
6.14. ábra. Példa egy csúnyán tervezett trace-re

A másik bővítmény pedig a Teardrop vias bővítmény. Ez is egy egyszerű bővítmény, ami részben esztétikailag szépíti a nyák tervet, de fontos, hogy az éles sarkokat is minimalizálja. 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.

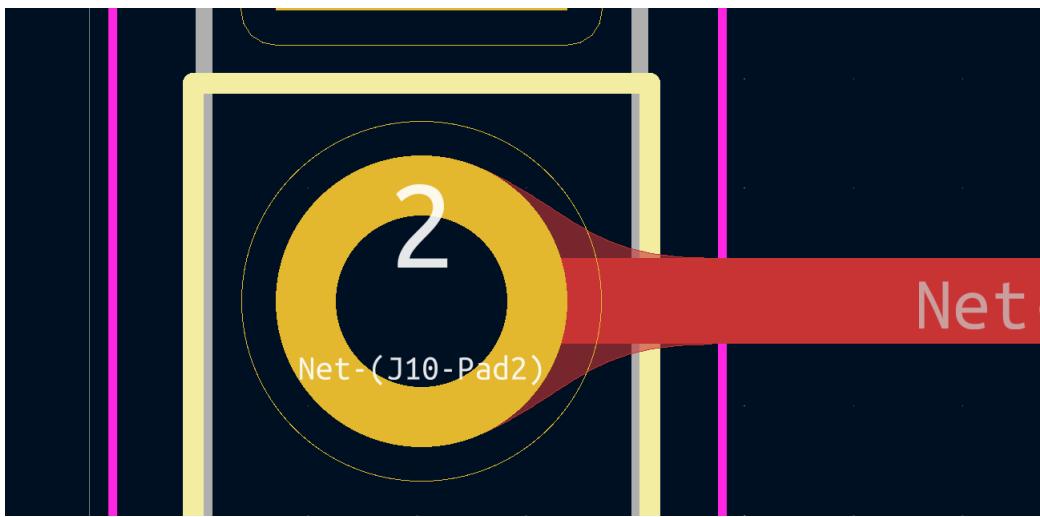


6.15. ábra. Teardrop vias UI

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

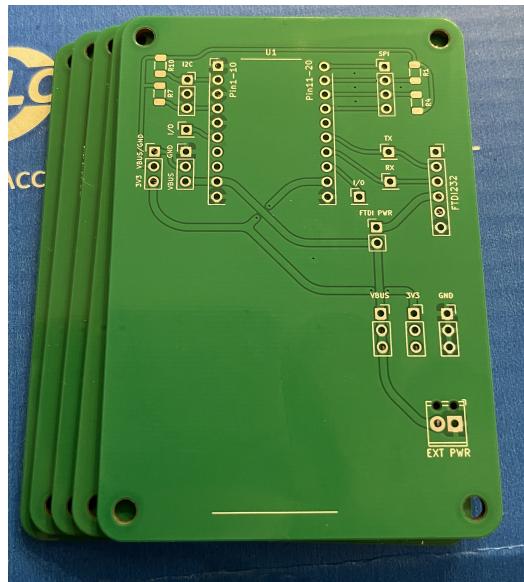


6.16. ábra. Teardrop bővítmény használata előtt

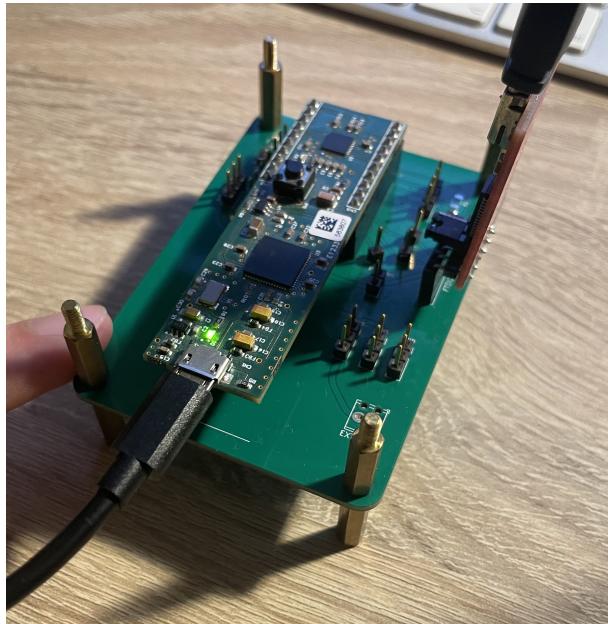


6.17. ábra. Teardrop bővítmény használata után

6.2. A Kész nyák



6.18. ábra. A Kész nyák



6.19. ábra. A kész nyák összeszerelve és forrasztva

6.3. A Programozó alentitásai

A következőkben bemutatom az alentitásokat, 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.

6.3.1. Az 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 port-ot 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 port-jához van kötve.
4. Az OSC kimenet az Int_Osc entitás Clk port-jához 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 választása

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

6.20. ábra. Elérhető CLK-frekvenciák az FPGA adatlapja alapján [7]

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, órajelenként 4 műveletet végez a I2C entitás. És 2 szer gyorsabbnak, mint a célzott SPI órajel, mert órajelenként 2 műveletet végez az SPI entitás.

Az én célzott I2C órajelsebességem nagyobb, mint a célzott UART Baud ráta, de lassabb, mint 1MHz-nel. Ez azért van, mert a legtöbb modern EEPROM képes 1MHz órajelsebességre, és ki szeretném használni ezt a sebességet. Ez 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. Ez azért van, mert az UART sebességét kommunikáció közben nem lehet változtatni, mert az UART aszinkron. Az SPI célsebesség 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 ráta 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 számláló a tényleges 230400 Baud-hoz képest. Úgy kellett választanom az óra jelet, hogy minimális kerekítéssel le lehessen osztania,

és ez a késés minimális legyen. A célzott UART 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.

6.3.2. A write8bit entitás

A write8bit 8 bites adat írására szolgál az SPI vonalokon. Az entitás bemeneti és kimeneti port-jai a következők:

Port-ok:

- 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.

6.3.3. 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

Port-ok:

- 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.

6.3.4. 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.

6.3.5. 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.

Port-ok:

- 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.

6.3.6. 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.

Port-ok:

- 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' értékkel 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 adat bájtok biten kent való 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 szolga eszköz visszaigazolásának (ACK) vára-kozása.
 - 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 szolga 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 szolga 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 változatára 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 változatára 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 változatára 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.

6.4. 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.

6.4.1. A main entitás állapotgép

- 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.

6.5. C++-ban megírt utasítás küldő program

6.5.1. A Terminal app program

A program célja, hogy hexadecimális formátumban megadott adatokat küldjön a számítógéphez csatlakoztatott programozónak egy COM soros port-on 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 port-ot olvasásra és írásra.
 - A "SetupComm" függvényel 1 MB-os bemeneti és kimeneti puffert állít be a soros port-hoz.
 - A DCB struktúrában beállítja a kommunikációs paramétereket. Például: Baud ráta: 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 port-on keresztül.
- minden elküldött blokk után kiírja, hány bájtot sikerült elküldeni.

4. Válasz fogadása a programozótól.

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

6.5.2. A "File app" program

Ez a program hasonló a Terminal app-hoz, de fájlokkal 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.

6.5.3. 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 bajt, minden beleértve.

A következő 2 bajt 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 [9]. Tehát a teljes adatkeret hossz 260 bajt lesz, és az olvasási fejléc hossza 4 bajt [8]. 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.

6.5.4. Utasítások és utasítás sor példa és magyarázat.

Mint korábban említettem a programozónak összesen 7 fő állapotja van. Az alap állapot a base state. Ebből az állapotból 6 másik 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 ki- fejezett bajt:

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

6.1. táblázat. Állapotokhoz tartozó hexadecimális kódok

7. fejezet

Tesztelés

7.1. Tesztelési és 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 hibakeresésre.

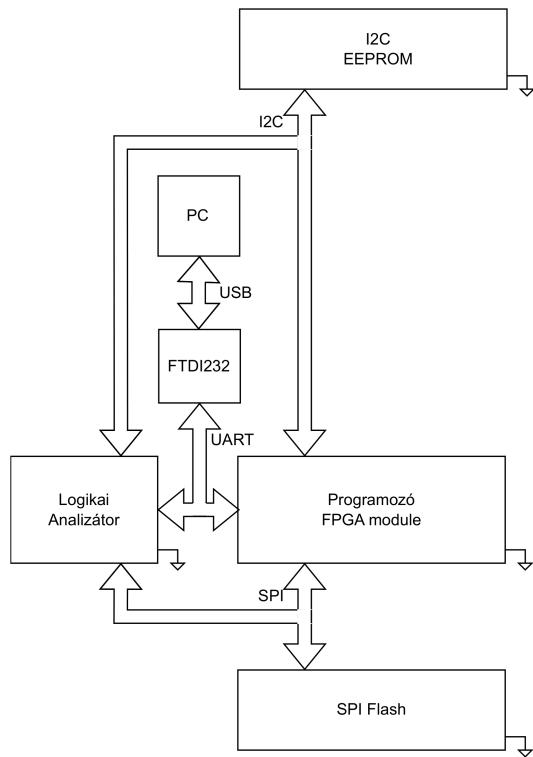
Egy Sealeae logic 24MS/s-es logikai analizátort használtam. Használtam 2 memória modult is, egy I2C EEPROM-ot meg egy SPI Flash-t is, hogy tudjam ellenőrizni programozó működését.

A I2C EEPROM cikkszáma: AT24C256. Egy 256Kb-es EEPROM [8]

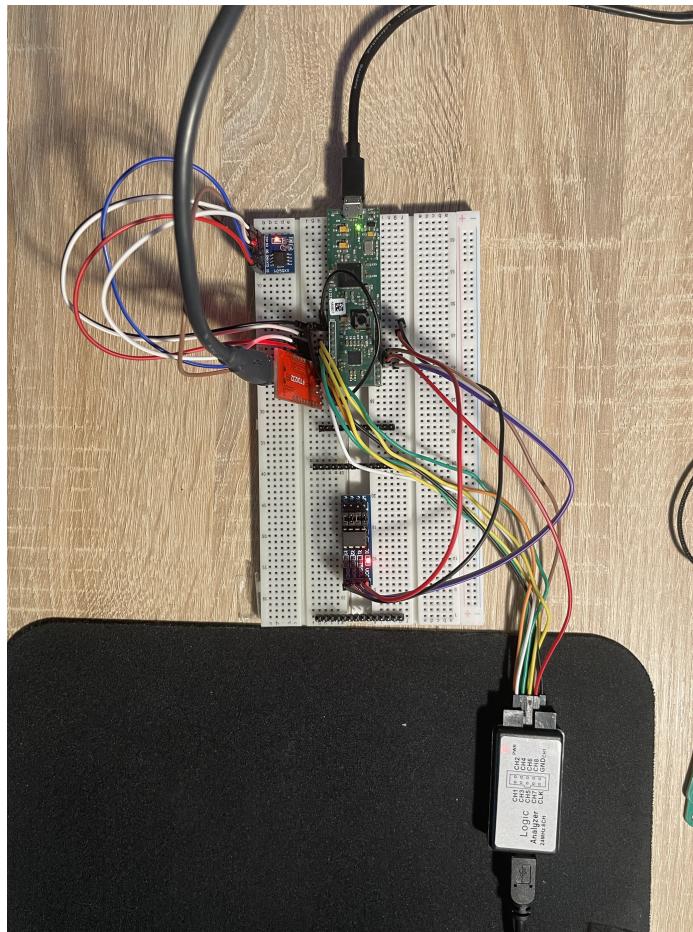
A SPI Flash cikkszáma W25Q64FV. Egy 64Mb-es Flash [9]

A teszteléshez és analízishez a mérési elrendezés kenyérpanelet történt, mert a nyákon nem lehet egyszerre csatlakoztatni a memória modulokat meg az analizátort is.

Az elrendezés:



7.1. ábra. Analízis elrendezés blokkvázlata



7.2. ábra. Analízis elrendezés kenyérpanelen

7.2. Tesztelés

7.2.1. SPI configure állapot tesztelése

Az SPI configure mindenkor egy bájtos SPI írás, a programozó beállításaitól függetlenül. Ezt az állapotot azért hoztam létre, hogy egy bájtos utasításokat könnyen tudjak írni, mint például a "chip erase", vagy "write enable".

Teszteléshez is "write enable" utasítást küldök a SPI Flash memóriának. Használt utasítássor az utasítás fájlban:

4106

exit

Teszt eredménye:

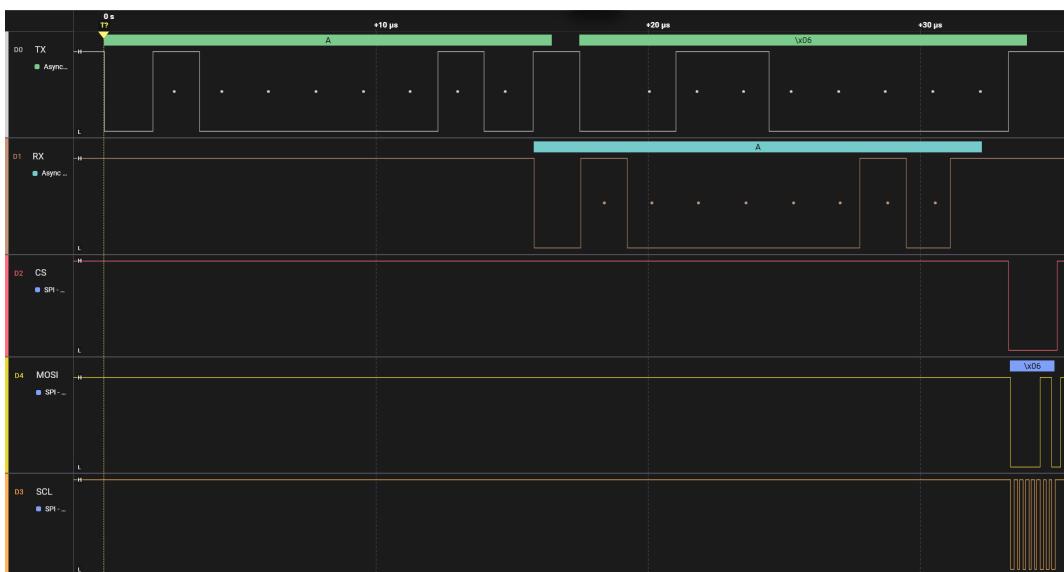
Log fájl tartalma:

Bytes written: 2

Bytes in queue: 1

Received: 41

Rögzítés:



7.3. ábra. SPI configure állapot tesztelése rögzítés

A rögzítésen látható, hogy a programozó fogadja a x42 értékű bájtot, és visszaküldi.

A következő bájtot már viszont SPI-on írja. Ugyanazt írja SPI-on, mint amit kapott UART-on.

Az SPI configure állapot megfelelően működik.

7.2.2. SPI write állapot tesztelése

Teszteléshez az SPI Flash memóriában egy teljes page-ot azaz oldalt írtam. Ez 256 bájtnyi adat, a legtöbb, amit egy kommunikációval lehet írni az SPI Flash-be [9].

Használt utasítássor az utasítás fájlból:

4400010404000504

4106

43020000004163636F7264696E6720746F20616C6C206B6E6F776E206C617773206F6
6206176696174696F6E2C207468657265206973206E6F207761792061206265652073686F
756C642062652061626C6520746F20666C792E204974732077696E67732061726520746F6
F20736D616C6C20746F206765742069747320666174206C6974746C6520626F6479206F6
666207468652067726F756E642E20546865206265652C206F6620636F757273652C20666C
69657320616E797761792062656361757365206265657320646F6E277420636172652077686
1742068756D616E73207468696E6B20697320696D706F737369626C652E2059656C6C6F
772C20626C61636B0

exit

Az első két utasítás azért kell, hogy beállítsuk a SPI kommunikáció hosszát és az Flash-nek elküldjük a "write enable" utasítást. Maga az adat, a harmadik utasításban van. És a "Bee Movie" filmnek a forgatókönyvének első 256 bájtja.

Teszt eredménye:

Log fájl tartalma:

Bytes written: 8

Bytes in queue: 1

Received: 44

Bytes written: 2

Bytes in queue: 1

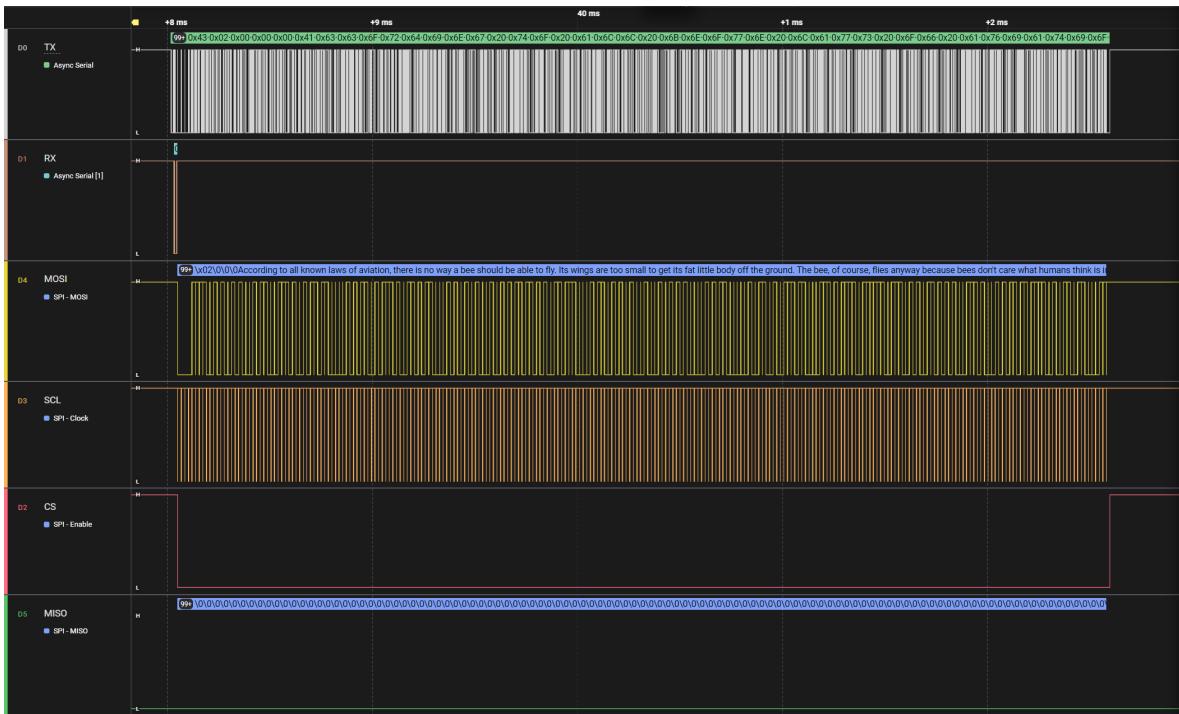
Received: 41

Bytes written: 262

Bytes in queue: 1

Received: 43

Rögzítés:



7.4. ábra. SPI write állapot tesztelése rögzítés

A rögzítésen látszik, hogy az programozó valóban az összes érkező bájtot helyesen elküldi a Flash-nek SPI-on keresztül.

Az SPI write állapot megfelelően működik.

7.2.3. SPI read állapot tesztelése

Teszteléshez az SPI Flash memóriában egy teljes page-et, azaz oldalt olvastam ki. Ugyan azt, amit beleírtam az SPI write tesztelés közben.

Használt utasítássor az utasítás fájlban:

4400010404000504

42030000000

Itt is az első utasítás azért kell, hogy beállítsuk a SPI kommunikáció hosszát.

Teszt eredménye:

Log fájl tartalma:

Bytes written: 8

Bytes in queue: 1

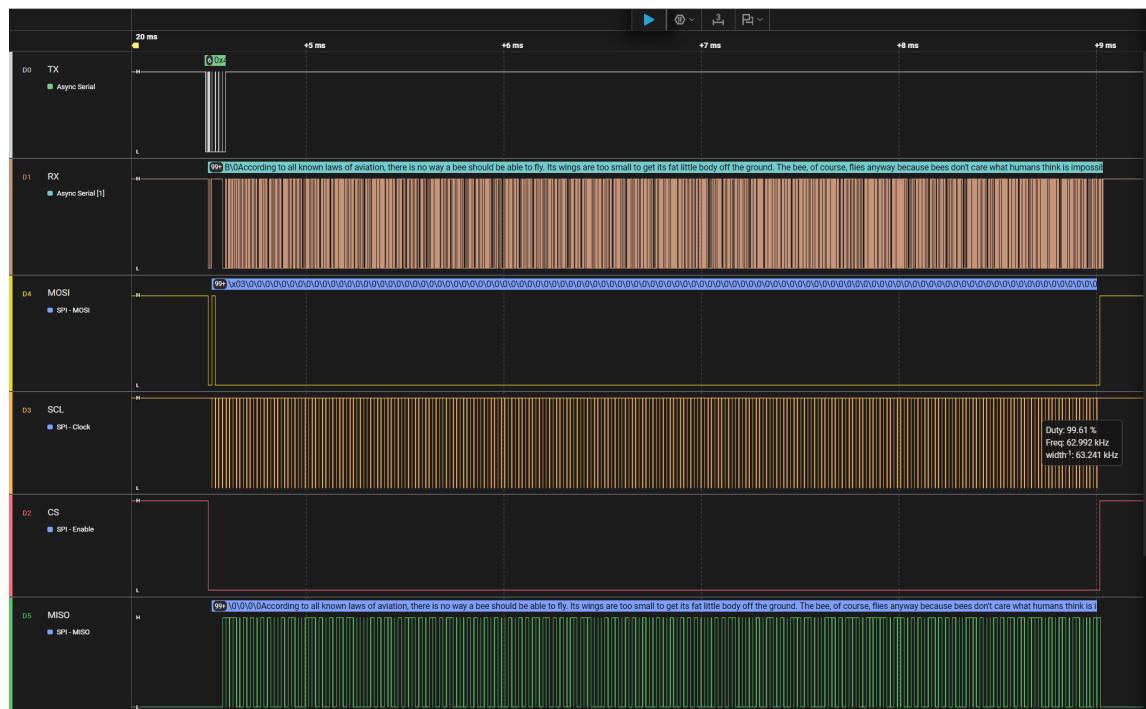
Received: 44

Bytes written: 6

Bytes in queue: 257

Received: 41 63 63 6F 72 (...többi adat bájt...) 36 B0

Rögzítés:



7.5. ábra. SPI read állapot tesztelése rögzítés

A rögzítésen látszik, hogy az programozó valóban az összes beírt bájtot helyesen visszaolvassa és elküldi UART-on a PC-nek.

Az SPI read állapot megfelelően működik.

7.2.4. I2C write és I2C read állapot tesztelése

A két állapotot együtt teszteltem, egy utasítás sorral. Az EEPROM-ba egy memória címre beleírtam az Test angol szó ASCII értékét, majd ugyan azt a címet olvastam ki.

Használt utasítássor az utasítás fájlban:

4400010404000804

46A00000546573740

wait(11ms)

45A00000A10

exit

- Az első utasítás azért kell, hogy beállítsuk a I2C kommunikáció hosszát.
- A második utasításban a 46 hexadecimális bájt az I2C write állapotot jelöli, az 0000 címre írunk.
- A harmadik utasítás egy váró utasítás, ez az EEPROM karakterisztikája miatt szükséges [8].
- A negyedik utasításban a 45 hexadecimális bájt az I2C read állapotot jelöli, az 0000 címről olvasunk.

Teszt eredménye:

Log fájl tartalma:

Bytes written: 8

Bytes in queue: 1

Received: 44

Bytes written: 9

Bytes in queue: 1

Received: 46

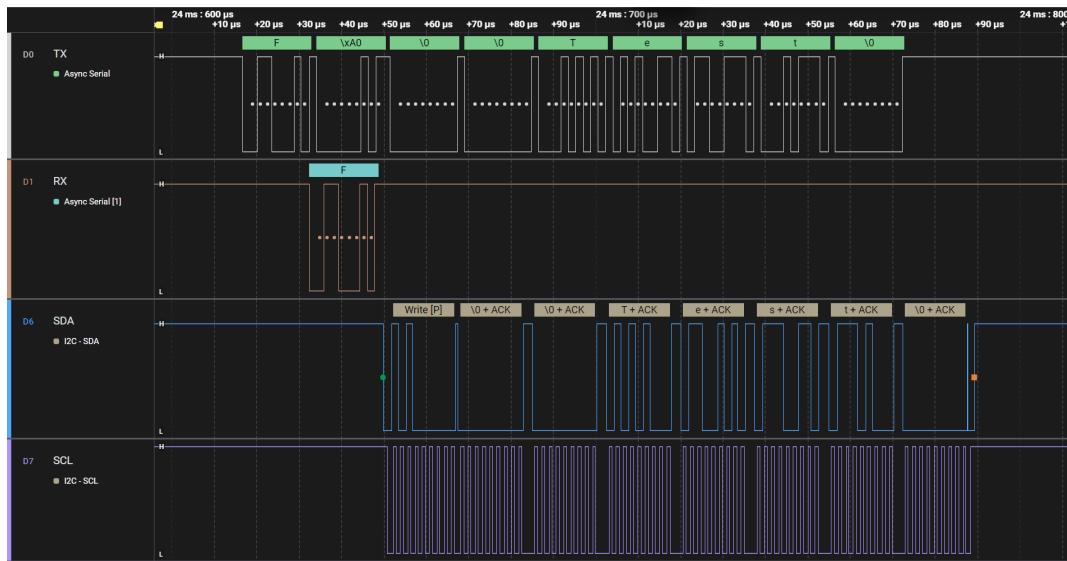
Waiting for 11 ms

Bytes written: 6

Bytes in queue: 5

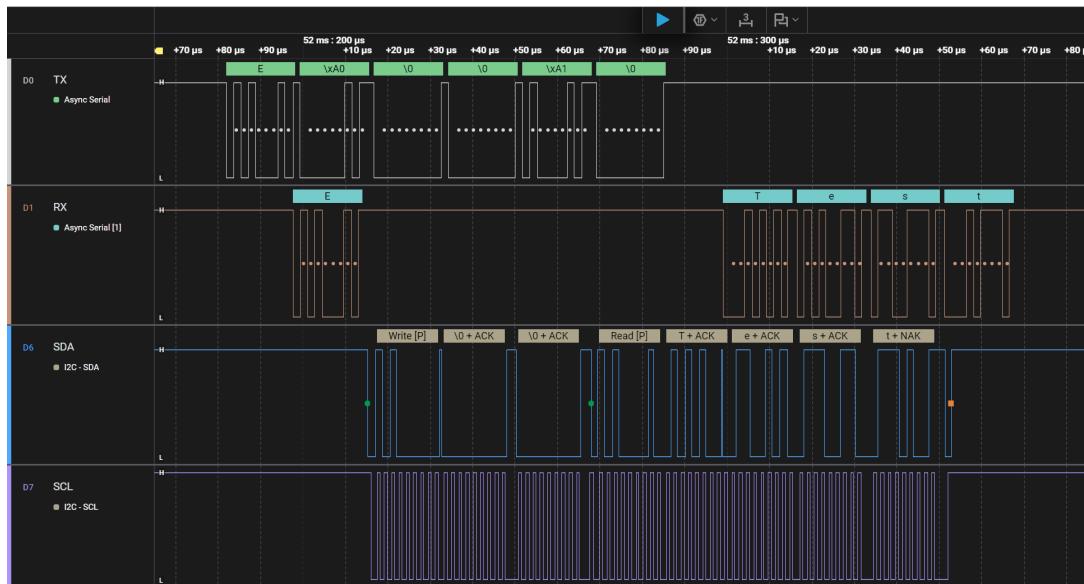
Received: 45 54 65 73 74

Rögzítések:



7.6. ábra. I2C write állapot tesztelése rögzítés

Itt látható a helyesen elküldött I2C írás.



7.7. ábra. I2C read állapot tesztelése rögzítés

Itt látható a helyes I2C olvasás. A programozó visszaolvasta és visszaküldte az előzőleg beírt adatokat.

Az I2C read és az I2C write állapot megfelelően működik.

7.2.5. Get settings állapot tesztelése

A Diplomamunka során sok rögzítést mutattam be, ahol látszik, hogy különböző hosszúságú adatkereteket küldött a programozó. A programozót mindenkor a Get settings állapottal kellett beállítanom.

Például a 4.5-as ábra a 23. oldalon és a 7.5-as ábra a 64. oldalon. Itt látható két külön hosszúságú SPI adatkeret.

A 4.7-as. ábra a 25. oldalon és a 7.6-as ábra a 66. oldalon. Itt látható két külön hosszúságú I2C adatkeret.

A Get settings állapot megfelelően működik.

8. fejezet

Összegzés

8.1. Magyar összefoglaló

Diplomamunkám célja az volt, hogy egy olyan hardver-szoftver rendszert hozzak létre, amely képes SPI és I2C protokollon keresztül különböző memóriák programozására. A rendszer felépítése egy PC-s vezérlő alkalmazásból, egy UART kommunikáción alapuló interfésből, valamint egy FPGA-ban implementált digitális áramkörből áll. A kommunikációs és vezérlő szoftvert C++ nyelven írtam meg, amely képes parancsokat küldeni, adatokat fogadni és azokat naplózni is. A VHDL-ben megírt modulokat úgy terveztem, hogy azok rugalmasan konfigurálhatók legyenek, lehetőséggel a kommunikációs keretek hosszának állítására. A tervezett FPGA-s logika tartalmaz SPI és I2C mester egységeket, UART vevő- és adómodulokat, valamint egy fő állapotgépet, amely a beérkező parancsokat feldolgozza.

A hardveres megvalósítás részeként egy kétoldalas nyákkot terveztem a KiCad szoftverrel. A tervezés során igyekeztem törekedni a rugalmasságra, például jumper-ek segítségével különböző feszültségű memóriachipek támogatása is megoldhatóvá vált.

A rendszer teszteléséhez valós EEPROM chipeket használtam, és logikai analizátor segítségével ellenőriztem a kommunikáció helyességét.

Összességében a projekt során sikerült elérnem a kitűzött célokat: egy működőképes, használható EEPROM programozót hoztam létre, amely gyakorlati példát szol-

gáltat az FPGA-k alkalmazására beágyazott rendszerek fejlesztésében. A munka során jelentős tapasztalatra tettem szert mind a digitális áramkörtervezés, mind az alacsony szintű kommunikációs protokollok kezelése terén.

8.2. Jövőbeli tervezek

Bár az elkészült programozó már most is képes EEPROM és Flash memória egységek programozására SPI és I2C protokollokon keresztül, számos irányba lehetne tovább bővíteni a rendszer funkcionalitását.

Elsőként szeretnék egy grafikus felületet készíteni a jelenlegi C++-ban írt parancssoros vezérlőalkalmazás helyett. Ez felhasználóbarátabbá tenné a programozót, és lehetőséget biztosítana a memóriák tartalmának vizuális megjelenítésére, szerkesztésére, valamint jobb fájl alapú be és kimentésre is.

Ezen kívül célom lenne újabb protokollok hozzáadása. Ez tovább növelné a programozó eszköz kompatibilitását különféle memóriatípusokkal és más perifériákkal.

Az protokollok sebesség beállítása most a design része, nem lehet az FPGA konfigurálása után változtatni. Ez is beállíthatóvá szeretném tenni.

Hardveres oldalon egy a most használt TinyFPGA modul helyett egy komplexebb FPGA használatával a rendszer teljesítménye és bővíthetősége is jelentősen nőne.

Végül, de nem utolsósorban, szeretnék a projektből egy nyílt forráskódú, dokumentált, közösség által is használható eszközt létrehozni. Ez elősegítené, hogy más fejlesztők vagy diákok is tanulhassanak a rendszerből, és akár saját igényeikhez igazítva továbbfejlesszék azt. Az egész forráskód már fent van GitHub-on.

8.3. English summary

As part of my final year project at the University of Miskolc, I set out to design and implement a configurable EEPROM programmer based on an FPGA platform. My goal was to create a standalone embedded system that would not only deepen my knowledge of FPGA technology and the VHDL hardware description language but

also offered practical utility in programming various EEPROM and memory chips via industry-standard communication protocols such as SPI and I2C.

The project integrates both hardware and software development. On the hardware side, I designed a printed circuit board using KiCad, taking into account integrity, power distribution, and flexibility for different memory modules using configurable jumpers. For the FPGA, I wrote the design in VHDL, creating modular components that handle communication, protocol framing, data handling, and control logic. These include UART communication modules for interfacing with a PC, SPI and I2C master modules for memory programming, and internal logic for command parsing and response generation.

On the software side, I developed a command-line interface in C++ for Windows that communicates with the FPGA via a COM port using UART. This interface allows users to send commands, receive responses, and log data from the programmer. A key challenge in the development was ensuring the precise timing and synchronization of the asynchronous UART and synchronous SPI/I2C protocols. I addressed this through careful selection of clock frequencies and the implementation of robust state machines. Additionally, I conducted extensive testing using logic analyzers and real memory modules to verify the system's functionality and reliability.

Overall, the project has provided me with valuable experience in embedded system design, digital electronics, and cross-disciplinary engineering skills, bridging software and hardware. I believe the resulting system serves as a useful tool for memory programming tasks and demonstrates the versatile power of FPGA-based solutions.

9. fejezet

Melléklet

- A diplomaunka teljes forráskódja

Irodalomjegyzék

- [1] Nandland: what is a testbench
- [2] Dr. Steve Arar: What Is VHDL? Getting Started with Hardware Description Language for Digital Circuit Design
- [3] S. Trimberger, Field-Programmable Gate Array Technology, Springer, 1994.
- [4] Texas Instruments: KeyStone Architecture Universal Asynchronous Receiver/Transmitter (UART)
- [5] Texas Instruments: A Basic Guide to I2C
- [6] Texas Instruments: KeyStone Architecture Serial Peripheral Interface (SPI)
- [7] A MachXO2 FPGA család adatlapja
- [8] A AT24C256 EEPROM adatlapja
- [9] A W25Q64FV FLASH adatlapja