ÓBUDAI EGYETEM

Kandó Kálmán Villamosmérnöki Kar
Mikroelektronikai és Technológiai Intézet

SZAKDOLGOZAT

OE-KVK

2021.

Hallgató neve: **Horicsányi László Gergely**

Hallgató törzskönyvi száma: **T006471/FI12904/K**

ÓBUDAI EGYETEM

Kandó Kálmán Villamosmérnöki Kar
Mikroelektronikai és Technológiai Intézet

HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a szakdolgozat saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozatban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja, a titkosításra vonatkozó esetleges megkötések mellett.

Budapest, 2021. 12. 13.

Horicsányi László Gergely

hallgató aláírása

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Óbudai Egyetem
Kandó Kálmán Villamosmérnöki Kar
Mikroeletronikai és Technológia Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: Horicsányi László Gergely
Szakdolgozat száma: SZD21060813441917
Törzskönyvi száma: T006471/FI12904/K **Neptun kódja:** [redacted]

Szak: villamosmérnöki (magyar nyelven) (Budapest)
Specializáció: Mikroelektronika és technológia specializáció
Szenzor és minőség modul

A dolgozat címe: Demonstrációs számítógép
A dolgozat címe angolul: Demonstration computer
A feladat részletezése: Tervezzen meg és készítsen el egy áramköri panelt tápegységgel és csatlakozókkal, amely egy egyszerű számítógép legfontosabb egységeit tartalmazza és amellyel a processzor és a kiegészítő egységek, buszrendszer működése mérőpontok és logikai analizátor, oszcilloszkóp segítségével bemutatható.

Intézményi konzulens neve: Horváth Márk

A kiadott téma elévülési határideje: 2023. június 30.
Beadási határidő: 2021. 12. 15.

A szakdolgozat: Nem titkos.

ÓBUDAI EGYETEM Kandó Kálmán Villamosmérnöki Kar Mikroelektronikai és Technológiai Intézet P.H. 1.

Kiadva: Budapest, 2021. 11. 12.

.....
Intézetigazgató

A dolgozatot beadásra alkalmasnak találom:

.....
belső konzulens

.....
külső konzulens

Tartalom

Abstract

The goal of this project was to build a device that can be used to explain the most important parts of modern day computers. Instead of using a commercial CPU, the machine was built with simple 74 series integrated circuits, the most complex one being an Arithmetic Logic Unit. This method needed substantially more components, so instead of designing one large printed circuit board, I created five different modulest hat can be connected with a backplane. Not only was this less expensive, it made tracking down problems easier, because I could test each module separately.

There were a few faults in the design, and not all of them could be fixed before the end of the semester, but the most important part worked: the computer could read an instruction from its program memory, execute it, and then read the next instruction.

Demonstrációs Számítógép

1. Bevezető

A szakdolgozatom célja egy olyan bemutató eszköz készítése volt, amivel elmagyarázható a számítógépek működésének alapjai. Mivel a gép célja az oktatás, nem pedig a számítási kapacitás, nem egy kész processzort, hanem sokkal egyszerűbb alapelemeket használtam, így az ember a saját szemével láthatja a processzor részeit. Már az Önálló Projekt I. és II. tárgyak során is egy ilyen gépen dolgoztam, az ott szerzett tapasztalatok alapján terveztem a mostani verziót.

Mivel a modern számítógépek meglehetősen összetettek, főleg a kora 80-as években készült gépeket vettem alapul. Ezen belül is leginkább a Zilog Z80-as mikroprocesszort és a Commodore 64 számítógép működését tanulmányoztam. Ezek alapján válogattam össze azokat az elemeket és funkciókat, amikkel a demonstrációs eszköznek rendelkeznie kell:

- saját memória, ami a programot és az adatokat tartalmazza
- egy programszámláló, ami követi, hogy épp melyik memóriacímnél tart a program
- egy dekódoló, ami képes az instrukciókat utasításokká alakítani
- regiszterek, amik kapcsolatban állnak a memóriával
- aritmetikai-logikai egység (más néven ALU), ami kapcsolatban áll a regiszterekkel
- digitális ki- és bemenetek, amikre külső eszközöket lehet csatlakoztatni

Volt egy ötletem, amit az Önálló Projekt tárgy során megpróbáltam megvalósítani, de végül arra jutottam, hogy nem éri meg ezzel is tovább komplikálni a gépet. A teljesség kedvéért viszont a szakdolgozatom elején elmagyarázom, ugyanis ez egy elég fontos része a számítógépek működésének.

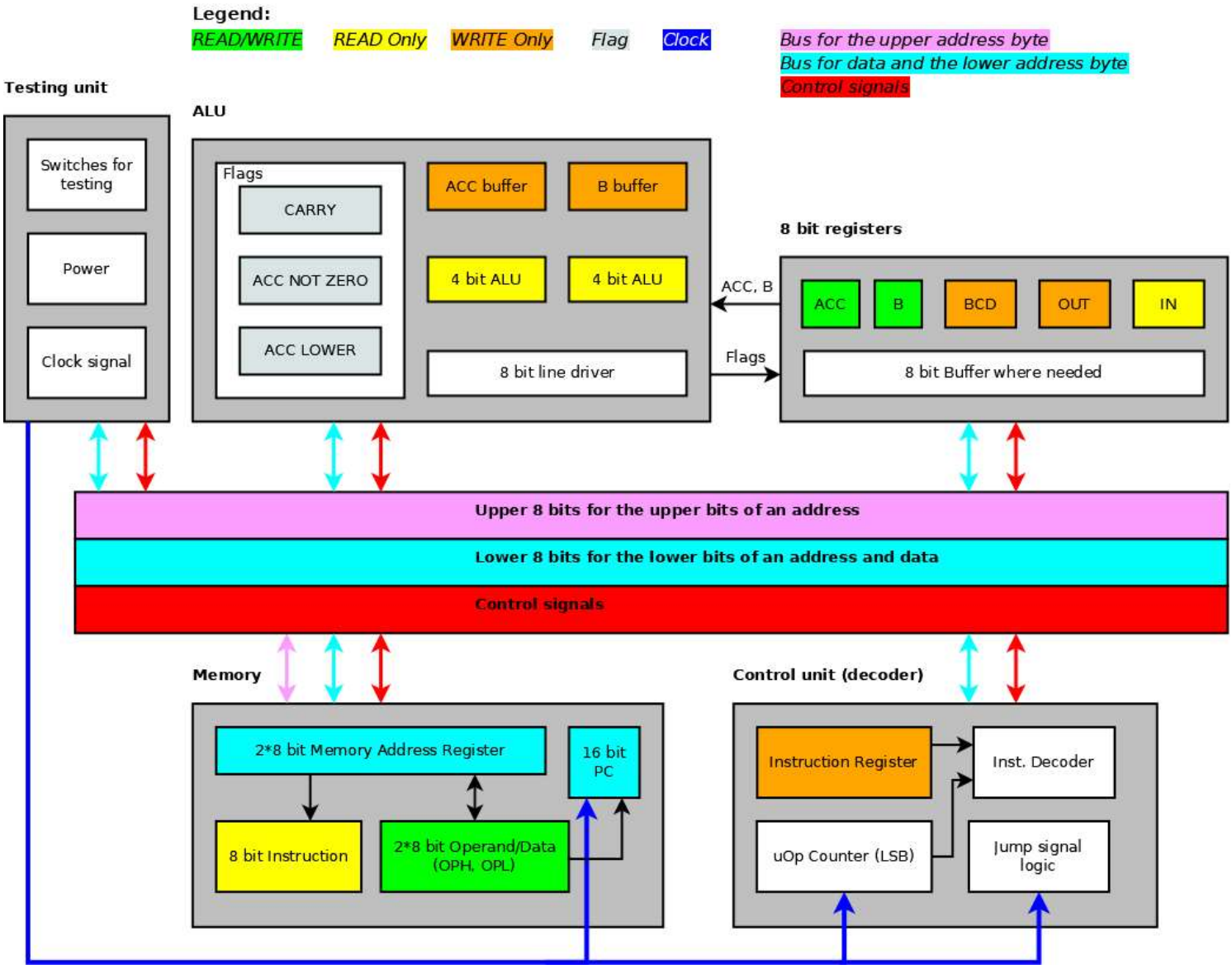
A korai számítógépek a futtatni kívánt programot valamilyen külső forrásból töltötték be a belső memóriájukba, általában floppy lemezről, vagy magnószalagról, de ha valaki elég türelmes volt, akkor akár kézzel is begépelhette magának a programot. Erre azért volt szükség, mert a belső memóriának használt RAM illékony, a benne tárolt adatok a gép kikapcsolásakor elvesznek. Cserébe sokkal gyorsabban lehet írni és olvasni, mint egy magnószalagot, vagy floppy lemezt. Ezt a műveletet úgy akartam bemutatni, hogy a gépre tettem volna egy mikrokontrollert, egy EEPROM-ot a futtatni kívánt programmal, a gép programmemóriájának pedig párhuzamos SRAM-ot használtam volna, ami egy illékony, de cserébe gyors memória fajta. Bekapcsoláskor a kontroller az EEPROM-ból olvasná be a programot és onnan másolná be a gép programmemóriájába. Ebben az esetben az EEPROM lett volna a floppy, a mikrokontroller pedig a floppy olvasó. Végül úgy döntöttem, hogy ez a rész nem éri meg az extra bonyolultságot, mert ez egy viszonylag könnyen megérthető folyamat. Ráadásul elég furán mutatott volna, ha a demonstrációs számítógépen van még egy mikrokontroller is, ami nagyságrendekkel nagyobb teljesítménnyel rendelkezik, mint maga a gép. Ehelyett egyszerűen párhuzamos EEPROM-okat használtam programmemóriának. Ezeknek ugyan milliszekundumos nagyságrendű az írási sebességük, de ez nem fog gondot okozni, hiszen a gép legfeljebb csak pár Hertzes órajellel fog futni. Sokkal többet ér az, hogy az EEPROM-okba írt program nem vész el minden kikapcsoláskor.

Összesen öt nyomtatott áramkört készítettem, amikből az egyik főleg a tesztelésért felelős, a maradék négy alkotja a számítógépet. Ez a felbontás két szempontból is előnyös: egyrészt olcsóbb volt több kisebb nyomtatott áramkört rendelni, mint egy nagyot, másrészt, ha tesztelés során valahol tervezési hibát találok, akkor újratervezés után elég csak a hibás áramkör, vagy áramkörök helyett újat rendelni.

A hardver mellett szoftverre is szükség volt. Először is tervezni kellett egy saját assembly nyelvet, amiben a demonstrációs számítógéphez lehetett programokat írni. Ezen kívül írnom kellett két programot: az egyiknek az a feladata, hogy lefordítsa az assemblyben írt kódot instrukciókra, amit aztán a program memóriaként szolgáló EEPROM-okba lehet írni. A másik a dekódoló egység feltöltéséhez kellett, mert sokkal gyorsabb, megbízhatóbb, és flexibilisebb ezt a feladatot egy programra bízni, mint kézzel begépelni.

Remélem ez az eszköz segíteni fog másoknak is megérteni a számítógépek működését.

2. Blokkvázlat

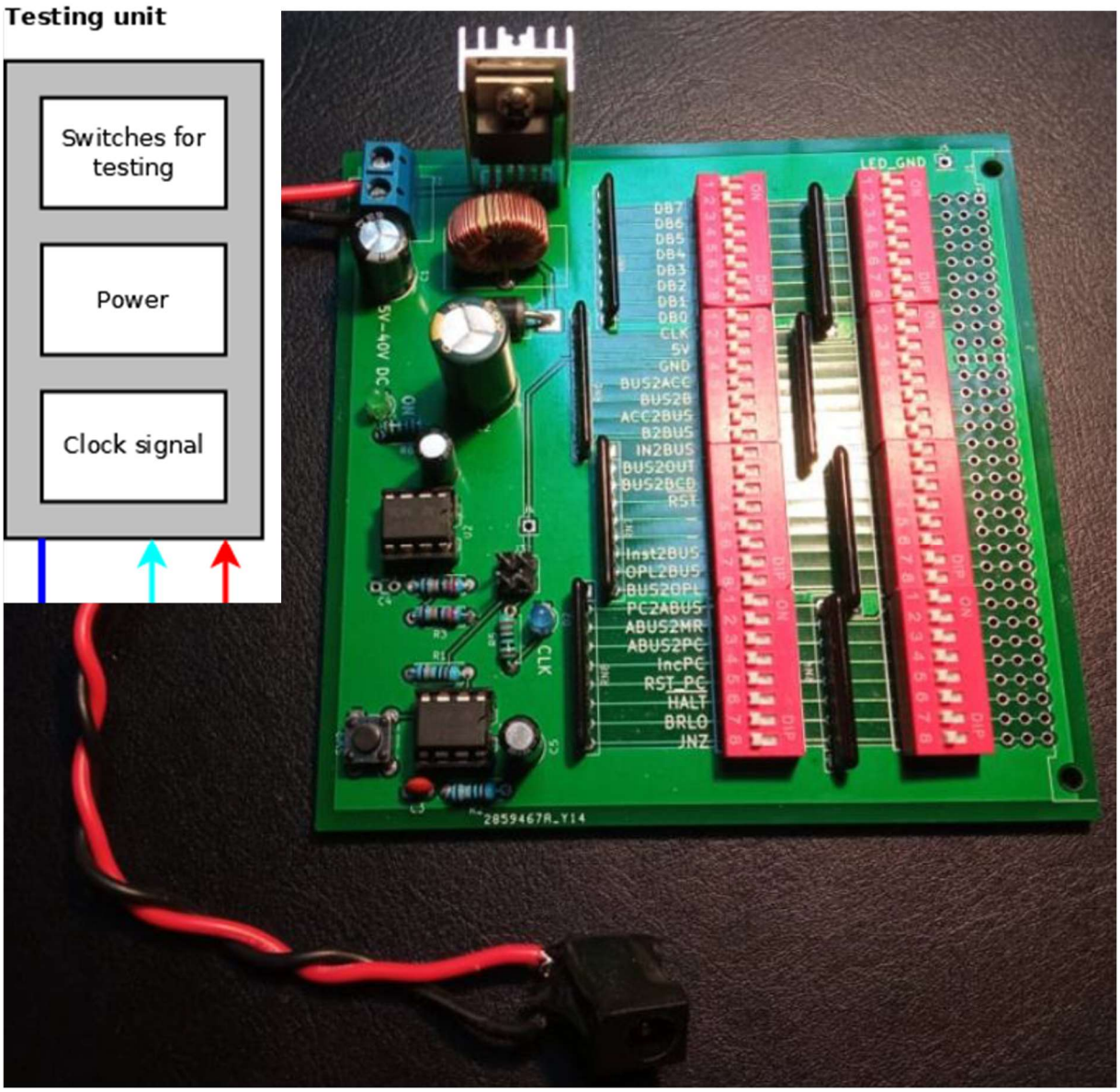


1.: A gép blokkvázlata

A gép blokkvázlatán jól látszanak a fő egységek: a tesztelő modul (*Testing unit*), az Aritmetikai Logika Egység (*ALU*), a 8 bites regiszterek (*8 bit registers*), a memória (*Memory*), és a vezérlő egység, más néven dekóder (*Control unit* vagy *decoder*), valamint az őket összekötő buszrendszer. A továbbiakban ezeket az egységeket fogom részletezni.

3. Az áramkörök

3.1. Tesztelő modul



2.: A tesztelő modul

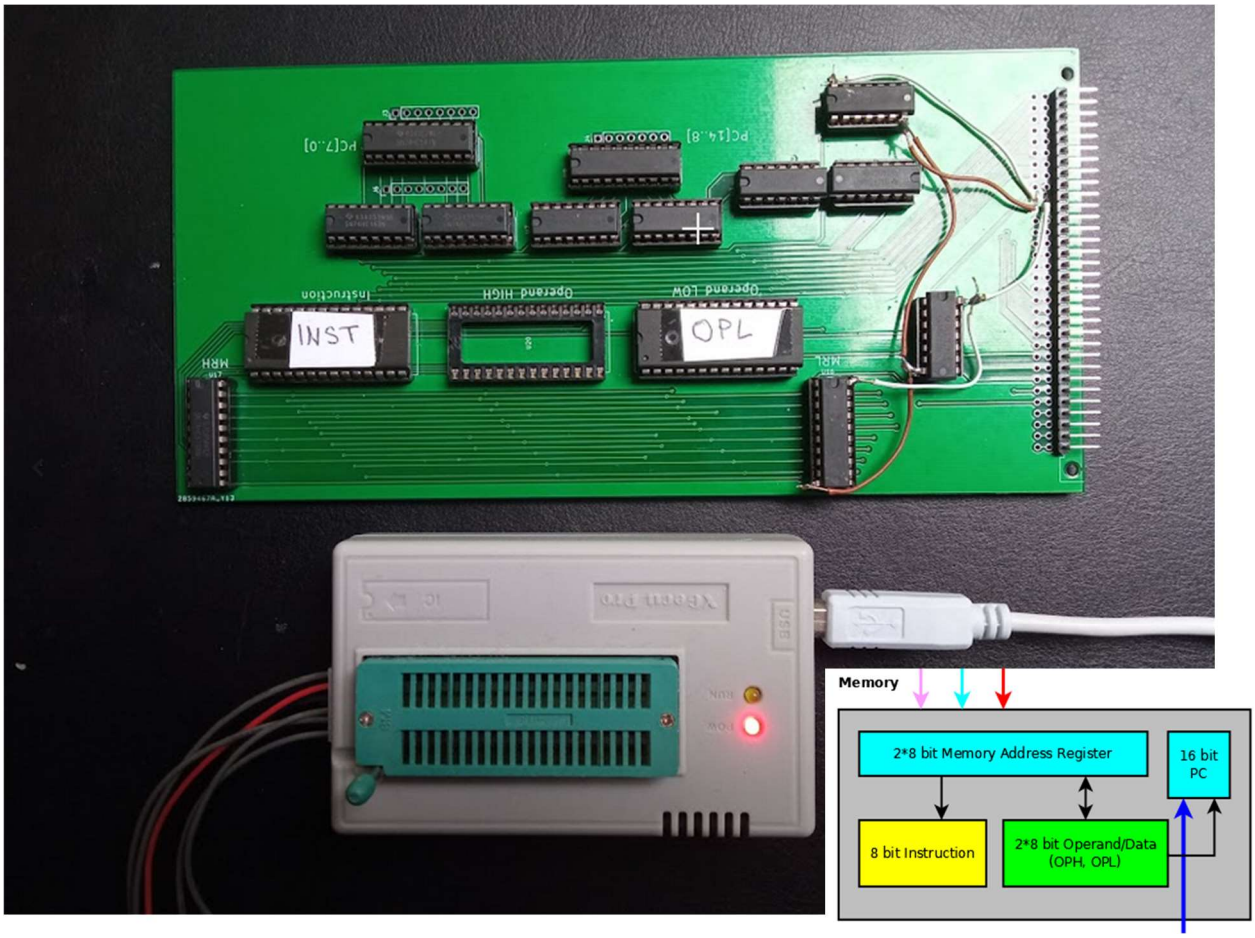
Elsőnek a legegyszerűbb áramkört szeretném bemutatni. Az Önálló Projektek során tapasztaltam, hogy mennyire hasznos lenne, ha a gépben futó jeleket kapcsolókkal tudnám állítani. Néhány speciális jelet leszámítva az összes jel egy 32 csatornás buszon fut, erre vannak rácsatlakoztatva a nyákok. Ezzel az áramkörrel a buszon futó jeleket lehet kapcsolgatni, de ez felelős az órajel és a megfelelő feszültség előállításáért is.

A jelek beállítására egy nagyon egyszerű rendszert dolgoztam ki: minden jelhez két kapcsoló tartozik, az első kapcsoló egy ellenálláshálókból készült feszültségosztó segítségével magas és alacsony szint között állítja a jel értékét, a második kapcsoló pedig ezt a jelet köti rá a megfelelő sávra a buszon. A feszültségosztónak van egy további előnye is: ha véletlenül egy olyan jelet akarnék magas értékre állítani, amit valamelyik integrált áramkör éppen alacsonyra húz, akkor nem közvetlenül rövidzárlaton át, hanem egy 470 Ohmos ellenálláson keresztül folyna bele áram, és annyit a gépen lévő összes integrált áramkör képes elnyelni. Ha pedig egy magas jelre próbálnék egy alacsony jelet kiadni, akkor is egy nagy ellenálláson keresztül folyna az áram.

Az órajel előállítására két lehetőség van, mindkettő egy 555-ös időzítőn alapul. Az egyik astabil módban működik, és egy nagyjából 2 Hz-es (elvileg 0,46s periódusidejű) jelet ad. A másik monostabil módba lett bekötve, és egy nyomógombot pergésmentesít, így lehetőség van manuálisan léptetni a programot. Ennek kicsit túl alacsonyra állítottam a periódusidejét, és így is előfordulhat, hogy egy lassú gombnyomás során két órajelimpulzus keletkezik, de még mindig jobb, mintha csak egy egyszerű nyomógombot használtam volna. A két órajel-forrás között jumperek segítségével lehet választani. A kiválasztott órajel kimenete egy kék LED-re is rá van kötve, így szemmel lehet követni az órajel szintjét.

Az 5V-os tápfeszültséget egy LM2576T-5.0 kapcsolóüzemű táp szolgáltatja. Ez egy fixen 5V-os tápegység, ami akár 3A-t is képes kiadni. Itt elő is került az első hiba, ugyanis a bemenő oldal szűrőkondenzátorát rossz helyre tettem a kapcsolási rajzon, és ez megzavarta az IC engedélyező lábát. Erről a szakdolgozat későbbi *Tesztelés, hibák és javítások* című fejezetében bővebben is írok.

3.2. Memória



3.: A memória modul, és az EEPROM-ok írására használt programozó. A vezetékek egy tervezési hiba kijavítására lettek beforrasztva.

Itt található a programot és adatokat tartalmazó memória, a memória cím regiszter, és a programszámláló. Ezek mellett vannak még olyan alkatrészek, amik a működést segítik, főleg néhány D-latch és logikai kapu.

Egy rendes számítógéppel ellentétben itt nem illékony memóriát, hanem EEPROM-ot találunk, egészen pontosan AT28C64 típusú párhuzamosan írható és olvasható memóriákat. Manapság inkább a sokkal kisebb, sorosan (többnyire I2C-n keresztül) írható EEPROM-okat lehet kapni, hiszen azok olcsóbbak, és kevesebb helyet foglalnak. A gyártási dátum alapján némelyik párhuzamos EEPROM öregebb, mint én. Az EEPROM-ok nagy előnye, hogy feszültség nélkül is megtartják a beleírt adatot, azaz a programot, viszont cserébe egy írás művelete nagyságrendekkel lassabb, többnyire milliszekundumokat vesz igénybe – a mérnöki világ tele van kompromisszumokkal. Ebben az esetben ez nem jelent nagy gondot, hiszen a gép órajele alig 2 Hz, és még a pergésmentesített nyomógombbal se lehetne akkora frekvenciájú órajelet kiadni, ami problémát okozna.

Összesen három darab EEPROM-ra volt szükség, az egyik az instrukciót tartalmazza, a másik kettőbe pedig egy 16 bites adatot lehet beírni. Ez az adat lehet egy instrukció operandusa, például az „LDIA, x” parancs egy „x” értéket ír be az ACC regiszterbe. Ez a memóriában úgy néz ki, hogy az instrukciókat tartalmazó EEPROM-ba („INST”) az „LDIA” parancs kódja kerül, és ugyanazon a címen bekerül az operandus magas és alacsony byteját tartalmazó EEPROM-okba („OPH” és „OPL”) az „x” érték. Természetesen nem minden instrukcióhoz tartozik operandus, sőt, azokat a memóriacímeket, amik a program vége után jönnek, fel lehet használni sima adattárolásra.

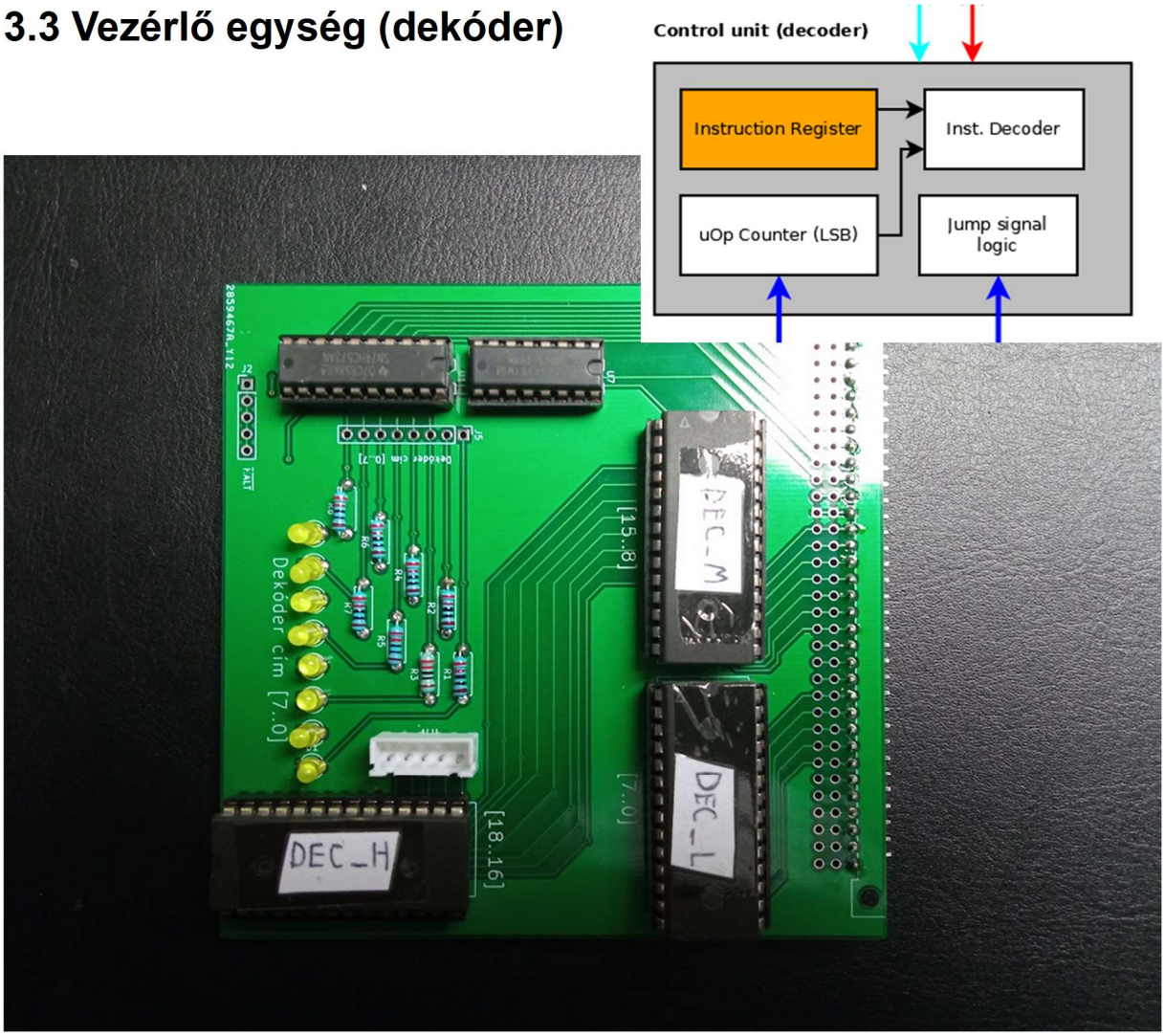
Mindhárom EEPROM saját DIP foglalattal rendelkezik, így viszonylag könnyen el lehet őket távolítani. A kivett memóriák programozásához egy TL866 II Plus programozót használok. Az általam írt compiler program három fájlt hoz létre, az első az znstrukciókat, a második az operandus magas byteját, a harmadik pedig az operandus alacsony byteját tartalmazza. Ezeket a programozó segítségével be lehet írni a megfelelő EEPROM-okba, és utána vissza lehet őket illeszteni a foglalatukba. Ahogy már korábban írtam, az igazi számítógépeknél ez másképp működik, de demonstrációs célokra ez a megoldás is megfelel.

A memória címzéséért a memória regiszterek („Memory Register”) felelősek. Ez két darab 8 bites D-latch, a bemeneteik az adatbuszra és a programszámlálóra, a kimeneteik pedig az EEPROM-ok címző bemeneteire van kötve. Kétféle forrásból lehet bele adatot írni: vagy a programszámláló értékét, vagy pedig egy JUMP utasítás keretében az operandus EEPROM-okban tárolt értéket írjuk a memória regiszterekbe.

A programszámláló ugyanolyan fontos a programok futtatásához, mint maga a memória. Ez tartja számon, hogy a program épp melyik instrukciónál tart. Ha pedig a program különböző részei között kell ugrani – például egy szubrutin hívás során –, akkor azt a programszámláló értékének változtatásával valósítjuk meg. Ebben a gépben a programszámláló négy darab 4 bites számláló, amiket úgy kötöttem be, hogy egy darab 16 bites számlálóként működjenek. Sajnos nem találtam olyan chipeket, amik egyszerre minden kívánatos tulajdonsággal rendelkeztek volna. A beépítésre került 74LS163 számlálókon nincsenek engedélyező lábak, kimeneteik mindig 0 vagy 1 értékben álltak. Mivel a programszámláló egy része az adatbuszon keresztül íródik be, szükség volt még két darab 8 bites bufferre, hogy le tudjam választani a számlálókat a buszról. Ezek a bufferek már tri-state kimenetekkel rendelkeznek, és az engedélyező lábuk segítségével tudom a programszámláló értékét a buszra juttatni.

Ezek mellett már csak néhány inverter és logikai kapu van ezen az áramkörön. A blokkvázlaton látszik, hogy mennyi vezérlő jelre van szükség. Sajnos az integrált áramkörök bemenetei nem egyformák, például a kimenetet engedélyező láb egyes alkatrészeken ponált, másokon negált jelet vár. Ezt úgy oldottam meg, hogy ahol szükséges volt invertereket és logikai kapukat alkalmaztam, hogy a nyákokba bekötendő jelek mindig ponáltak legyenek. Ha egy tömeggyártásra szánt terméket terveznék, akkor természetesen ehhez próbálnék igazodni, hogy meg lehessen spórolni az inverterek árát, viszont itt sokkal fontosabbnak tartottam az átláthatóságot.

3.3 Vezérlő egység (dekóder)



4.: A vezérlő egység a dekódolásért felelős három darab EEPROM-mal.

A vezérlő egység (más néven dekódoló egység vagy dekóder) végzi az instrukciók dekódolását, és ez adja ki a gép összes többi egységét vezérlő jeleket. A végrehajtandó instrukció a memóriából a buszon keresztül egy külön regiszterbe (*Instruction Register, IR*) kerül, így az instrukció végrehajtása közben felszabadul az adatbusz.

Mivel sok jel van, a dekódoló egységnek 3 darab EEPROM-ra van szüksége. Ezek lényegében egy aszinkron logikai hálózatként működnek, aminek a bemenete az elérni kívánt adat címe, a kimenete pedig az adott címen tárolt adat. Bármilyen logikai hálózatot logikai kapuk helyett EEPROM-okkal is megvalósíthatjuk. Ehhez annyit kell tennünk, hogy megszerkesztjük a hálózat igazságtábláját, a lehetséges bemeneti állapotokat címként kezeljük, és a hozzájuk szánt kimeneti állapotot pedig adatként beírjuk arra a címre. Ez különösen hasznos sok bemenettel rendelkező hálózatok esetén, amiket olcsóbb és egyszerűbb így megvalósítani.

A dekódolás menete a következő: minden assembly utasítás 4 lépésből áll. Tudjuk, hogy melyik utasítást hajtjuk végre, hogy hányadik lépésnél járunk, és azt is, hogy az adott lépésben mit kell a gép egyes egységeinek csinálnia, azaz hogy melyik vezérlő jel legyen 1, és melyik jel legyen 0. Minden egyes utasításhoz rendelünk egy számot, és ebből a számból, valamint abból, hogy hányadik lépésnél járunk, összeállítunk egy címet, és az ezen a címen tárolt adat tartalmazza a vezérlő jelek azon állapotát, amivel a gép a kívánt lépést hajtja végre. Ezeket az elemi lépéseket vagy állapotokat mikro-utasításnak („*micro-operation*” vagy röviden csak „*μop*”) nevezik. A legegyszerűbb dolog a lépés-szám nyomon követésére az, ha egy bináris számlálót használunk, amit az órajel léptet (ez a számláló a *uOp Counter*).

Vegyünk például egy összeadó műveletet. Ennek az utasításnak az azonosítószáma 0b10, így ez kerül majd az *Instruction Register*-be.

Utasítás	**Lépés**	**Cím**	Mikro-operáció feladata
0b10	**0b00**	**0b1000**	ACC-ba beírjuk az egyik operandust a memóriából
0b10	**0b01**	**0b1001**	B-be beírjuk a másik operandust a memóriából
0b10	**0b10**	**0b1010**	ACC-ba beírjuk az összeget az összegző áramkörből
0b10	**0b11**	**0b1011**	Már kész vagyunk, nem csinálunk semmit

A gép által végrehajtható instrukciókat egy táblázatba gyűjtöttem, és mindegyiket lebontottam négy lépésre. Minden egyes lépésekhez odaírtam, hogy melyik vezérlő jeleknek kell 1-es állapotba lenniük a kívánt működés eléréséhez, illetve még néhány adatot, például az instrukció nevét, mit csinál, vagy hogy kell-e neki valamilyen operandus. A dekódoló EEPROM-ok felprogramozásában ez a táblázat segített.

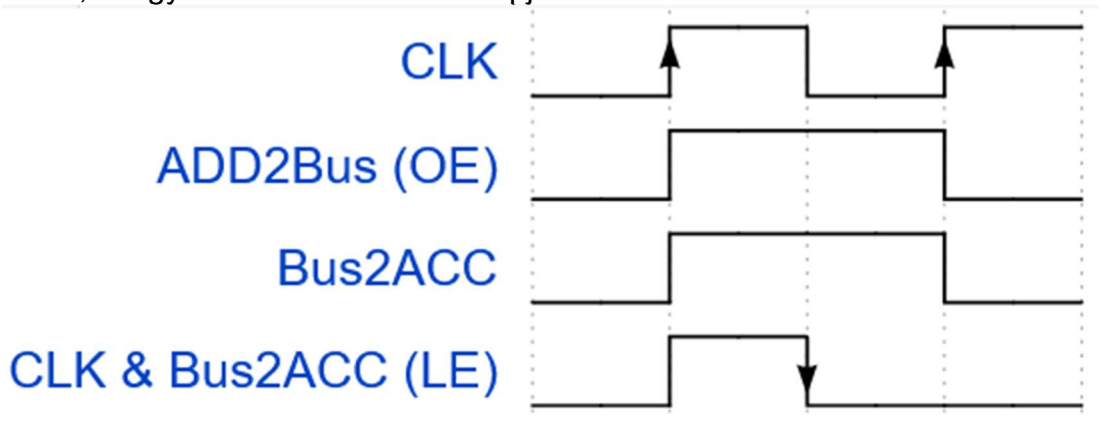
Az éppen végrehajtandó instrukció értékét (és egyben a dekóder EEPROM-ok címét) LED-ek segítségével lehet követni.

Látható, hogy az utolsó művelet alatt egy teljes órajelen át nem csinál semmit a gép. A modern számítógépeknél meg van oldva, hogy ne legyen kihasználatlan órajel, egyes modern mikrokontrollereknél (például Atmel AVR mikrokontroller családja) pedig minden egyes órajelre végrehajtódik egy utasítás, így ott fel se merül ez a probléma. Ennek a demonstrációs számítógépnek vannak olyan műveletei, amik mind a négy lépést kihasználják, és követhetőbb a kód, ha tudjuk, hogy minden egyes instrukció 4 órajel alatt hajtódik végre. Nem beszélve arról, hogy a hardvert is egyszerűbb ilyenre tervezni.

Egy lehetséges megoldás a kihasználatlan ciklusok megoldására egy új mikro-operáció beiktatása lenne, ami annyit csinálna, hogy a lépés-számláló értékét nullára állítaná. Ez sajnos nem lenne túl hasznos, mivel néhány speciális kivétellel az összes utasítás három vagy négy mikro-operációból áll, és semmit nem nyernénk azzal, ha a három lépéses utasítások végére beraknánk ezt negyedik lépésnek.

Ahhoz, hogy egy másolás során biztosan bekerüljön egy adott regiszterbe az adat, különböző ütemben kell vezérelni az *Output Enable* és a *Latch Enable* lábakat. A regisztereket és a buffereket is 74HC573 integrált áramkörök alkotják., Ez egy transzparens D-latch, azaz amíg a *Latch Enable* lába magas értéken van, a kimenetek követik a bemeneteket. Mondhatjuk úgy is, hogy az adat a *Latch Enable* láb lefutó élére íródik be.

Ha az Output Enable és a Latch Enable lábak egyszerre váltanának jelszintet, az egy logikai hazárdjelenséget eredményezne. Ezt elkerülendő a Latch Enable lábak vezérléséhez a vezérlő jelnek, valamint gép órajelének ÉS kapcsolatát kell venni, és így az alábbi működést kapjuk:

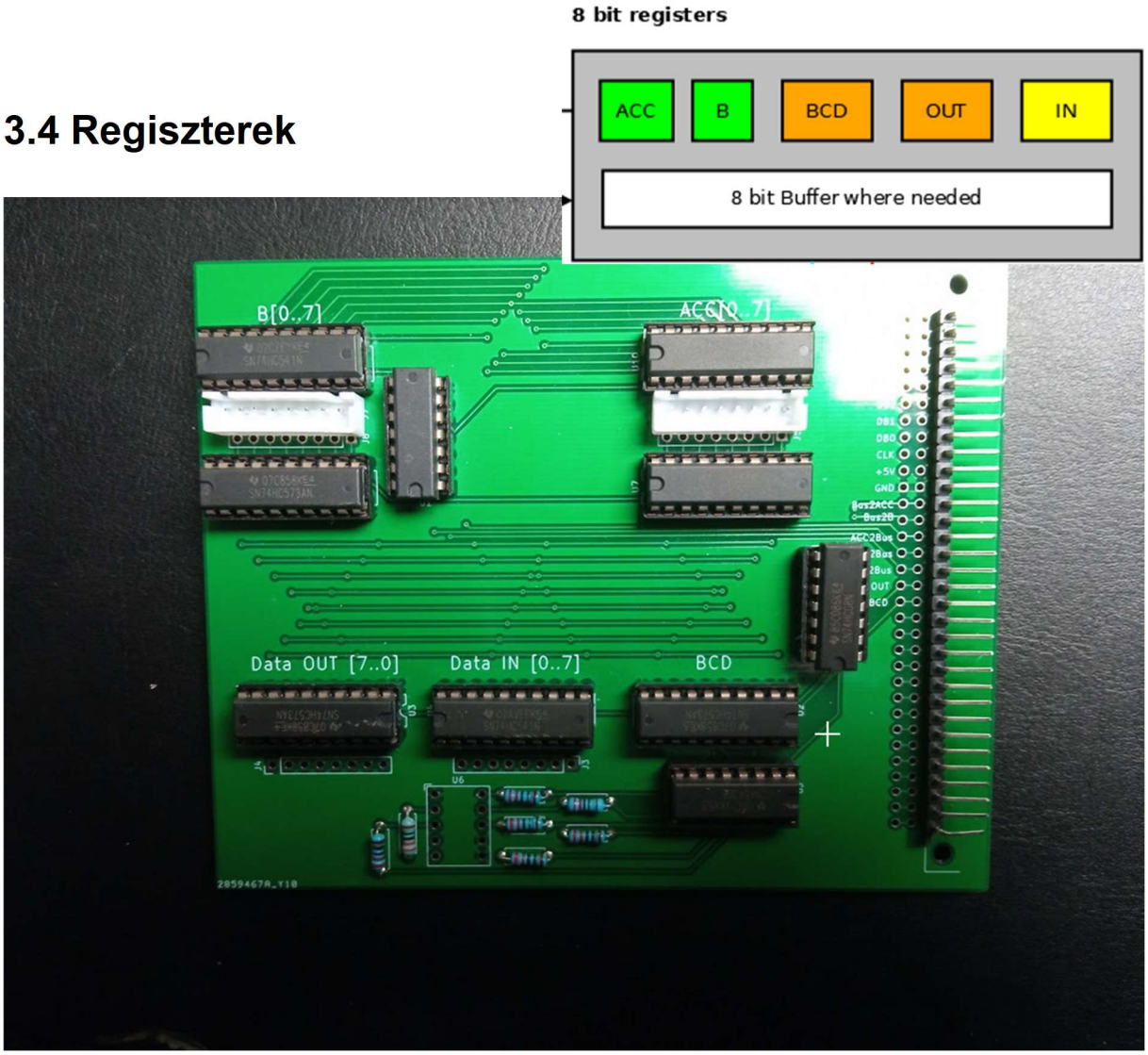


5.: Példa egy összeadás eredményének eltárolására. Ehhez egy Output Enable és egy Latch Enable lábat kell megfelelő ütemben vezérelni.

A CLK órajel felfutó éle vezérli az instrukció lépéseit számoló bináris számlálót, így minden felfutó élre más cím kerül a dekódolást végző EEPROM-ba, ezért a vezérlő jelek is az órajel felfutó éleire változnak. Az ADD2Bus jel a bináris összeadó kimenetét engedi rá a buszra, a Bus2ACC jel és az órajel ÉS kapcsolata pedig az ACC regiszter Latch Enable lábára van kötve, ami a buszról való olvasást vezérli. Ezzel a megoldással bőven van ideje a buszra adott értékeknek stabilizálódni, és az írás is biztosan azt az értéket fogja a regiszterbe juttatni, amit a buszra adtunk.

Az általam használt 74HC573 OE lába *active low* vezérlésű, de ha a dekódoló EEPROM-ba is így kerülne be az értéke, akkor nem működne az órajel ÉS kapcsolatával. Emellett a dekódoló programozását is kisebb eséllyel rontom el, ha minden jelet active high-ként kezelhetek. Emiatt a néhány active low jelet hardveresen, inverterekkel állítom a megfelelő logikai szintre.

3.4 Regiszterek



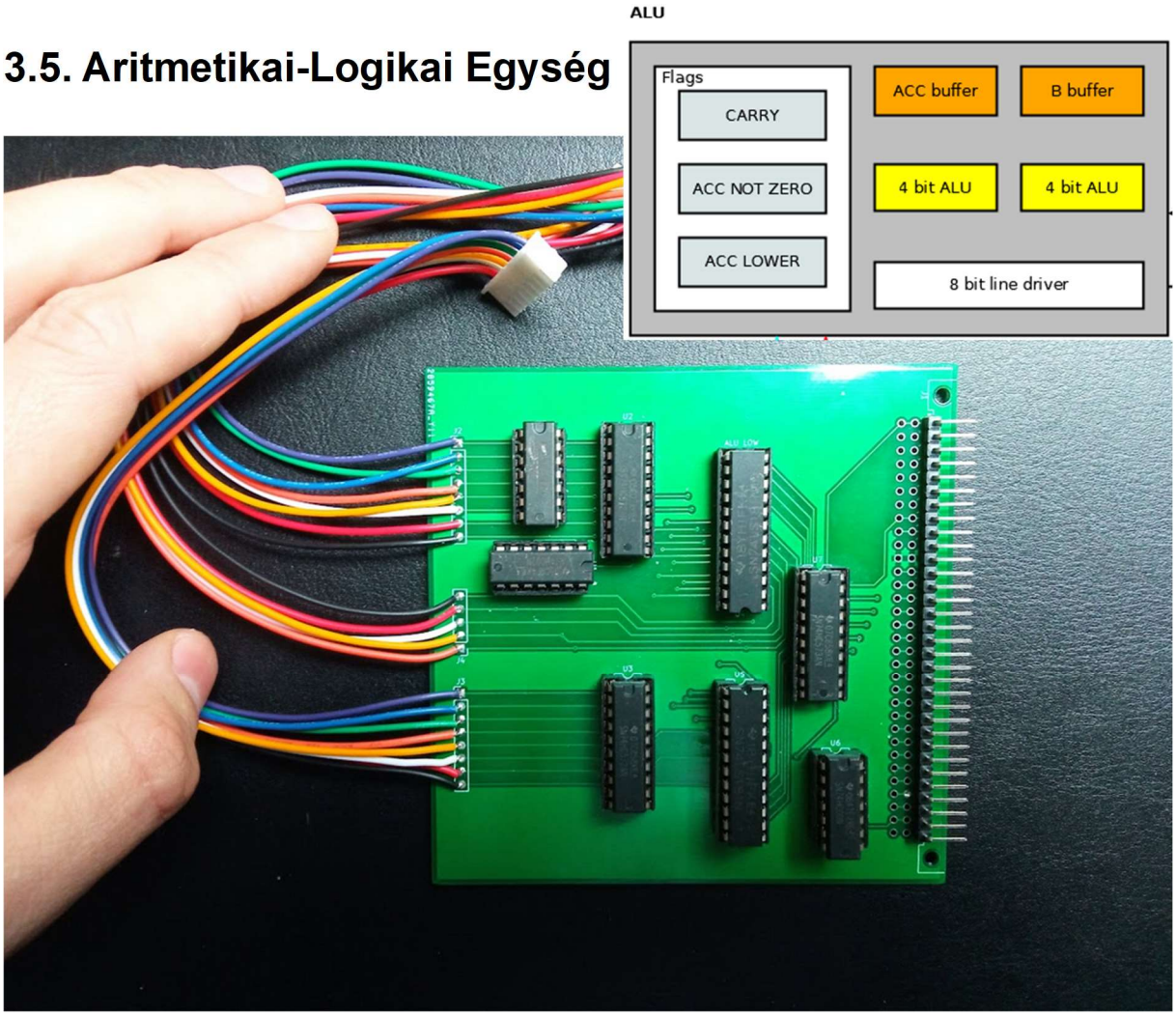
6.: A regisztereket tartalmazó modul

Ezen az áramkörön találhatók a különböző regiszterek. Adatkezelés szempontjából a két legfontosabb az *ACC* (a Zilog Z80-as processzor mintájára, az angol „*Accumulator*” szó rövidítése) és a *B* regiszterek. Ez a két regiszter tartalmazza azokat az adatokat, amikkel az aritmetikai és a logikai műveletek dolgoznak, a műveletek eredménye pedig mindig az *ACC* regiszterbe kerül.

A másik három regiszter arra szolgál, hogy a gép kommunikálni tudjon a külvilággal. A „BCD” regiszter egy 8 bites regiszter, aminek az értéke egy olyan dekóderre van kötve, ami az ott tárolt értéket egy hét szegmenses kijelzőn jeleníti meg. Sajnos nincs beépítve kijelző, mert rendelés nem olyan kijelzőt kaptam, mint amilyet rendeltem, és itthon nem találtam olyan kijelzőt, aminek a mérete és a lábkiosztása is megfelelet. Emellett van még két másik regiszter is: az „IN” regiszter egy bemenet, ami nyolc darab lábbal rendelkezik, és az ezekre kötött feszültségértéket be tudja olvasni. Az „OUT” regiszter ennek az ellenpárja, a beleírt értéket nyolc lábon jeleníti meg.

Ezek segítségével külső perifériákat is készíthetünk a géphez, de akár létező kommunikációs protokollokat is meg lehet valósítani. Például a Serial-Peripherial Interface (SPI) protokoll elvileg kompatibilis lenne ezzel a géppel, bár a lassú órajel miatt nem lenne valami nagy az adatátviteli sebesség.

3.5. Aritmetikai-Logikai Egység



7.: Az Aritmetikai-Logikai Egység. Ehhez a modulhoz külön csatlakozókat kellett beforrasztani, hogy az összes adatot át lehessen vinni.

Aritmetikai-Logikai Egység (ALU) nélkül a számítógép csak adatokat mozgatna az egyes alkatrészek között, ami csak mérsékelten hasznos. Egy ilyen eszközzel viszont képes a rendelkezésére álló adatokkal különböző műveleteket végezni. Egy ALU lényegében két részből áll: egy sor kombinációs hálózatból, amik különböző aritmetikai és logikai műveleteket végeznek, és egy multiplexerből. Az ALU bemenetére érkező adatot mindegyik kombinációs hálózat megkapja, viszont a multiplexer határozza meg, hogy melyik művelet eredménye jelenik meg a kimeneten.

Egy ilyen eszköz megtervezése nem bonyolultabb, mint a gép többi része, viszont meglehetősen időigényes, és a kész nyomtatott áramkör is nagy és drága lenne. Szerencsére ilyen áramköröket integrált kivitelben is lehet kapni. Ehhez a géphez két darab 4 bites 74LS181-es integrált ALU-t használtam. Ezek kaszkádosíthatók, így az egyik ALU az ACC és a B regiszterben található byteok alsó négy bitjével dolgozik, a másik pedig a két regiszter felső négy bitjét, valamint az alsó biteket kezelő ALU-ból érkező Carry bitet kezeli. Összesen 16 aritmetikai és 16 logikai művelet közül választhatunk, ám ebben a műveletek nagy része nem kimondottan hasznos.

Function Table

Mode Select Inputs				Active LOW Operands & F_n Outputs		Active HIGH Operands & F_n Outputs	
				Logic	Arithmetic (Note 2)	Logic	Arithmetic (Note 2)
S3	S2	S1	S0	(M = H)	(M = L) (C_n = L)	(M = H)	(M = L) (C_n = H)
L	L	L	L	$\overline{A}$	A minus 1	$\overline{A}$	A
L	L	L	H	$\overline{AB}$	AB minus 1	$\overline{A} + \overline{B}$	A + B
L	L	H	L	$\overline{A} + \overline{B}$	$A\overline{B}$ minus 1	$\overline{A}$ B	A + $\overline{B}$
L	L	H	H	Logic 1	minus 1	Logic 0	minus 1
L	H	L	L	$\overline{A} + \overline{B}$	A plus (A + $\overline{B}$)	$\overline{AB}$	A plus $A\overline{B}$
L	H	L	H	$\overline{B}$	AB plus (A + $\overline{B}$)	$\overline{B}$	(A + B) plus $A\overline{B}$
L	H	H	L	$\overline{A} \oplus \overline{B}$	A minus B minus 1	A ⊕ B	A minus B minus 1
L	H	H	H	A + $\overline{B}$	A + $\overline{B}$	$A\overline{B}$	AB minus 1
H	L	L	L	$\overline{A}$ B	A plus (A + B)	$\overline{A}$ + B	A plus AB
H	L	L	H	A ⊕ B	A plus B	$\overline{A} \oplus \overline{B}$	A plus B
H	L	H	L	B	$A\overline{B}$ plus (A + B)	B	(A + $\overline{B}$) plus AB
H	L	H	H	A + B	A + B	AB	AB minus 1
H	H	L	L	Logic 0	A plus A (Note 1)	Logic 1	A plus A (Note 1)
H	H	L	H	$A\overline{B}$	AB plus A	A + $\overline{B}$	(A + B) plus A
H	H	H	L	AB	$A\overline{B}$ minus A	A + B	(A + $\overline{B}$) plus A
H	H	H	H	A	A	A	A minus 1

Note 1: Each bit is shifted to the next most significant position.

Note 2: Arithmetic operations expressed in 2s complement notation.

8.: igazságtábla a 74LS181 adatlapjából

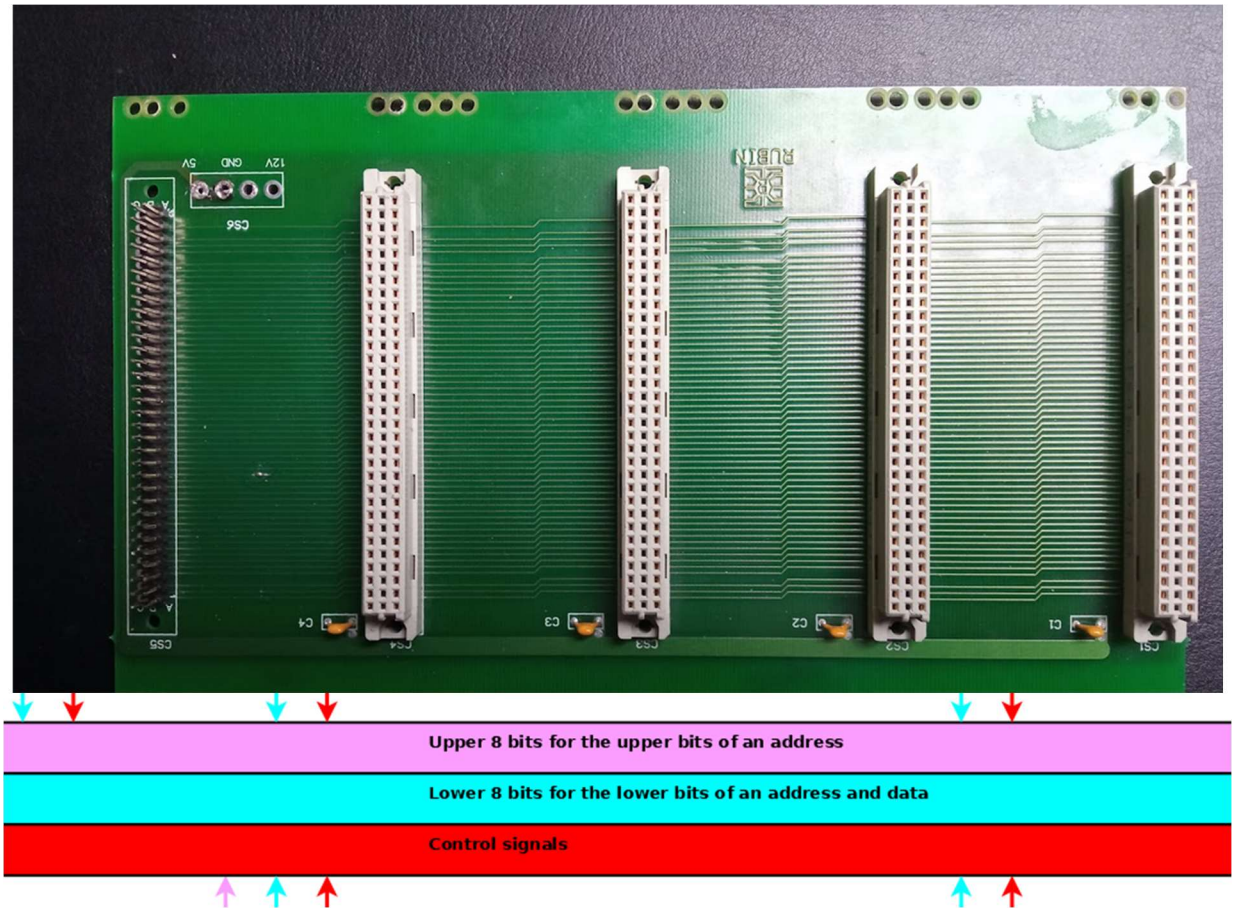
Például az *„A plus (A + B)”* összetett művelet helyett sokkal praktikusabb egy külön *„A plus B*” aritmetikai- és egy *„(A + B)*” logikai művelet. Az is határt szabott, hogy a sok vezérlő jel mellett még három darab dekódoló EEPROM is kevés volt ahhoz, hogy az ALU-n mind a négy címző bemenetet használjam. Végül az alábbi műveleteket lehet a géppel elvégezni:

Aritmetikai	Logikai
ACC + B	ACC **AND** B
ACC - B - 1	ACC **NAND** B
	ACC **OR** B
	ACC **XOR** B
	NOT ACC

Az ACC és a B regiszterek értékei nem a szokásos buszon, hanem külön vezetékeken keresztül kerülnek ide – egy igazi számítógépnél is a regiszterek egyik legfontosabb tulajdonsága, hogy az ALU könnyen eléri a bennük tárolt adatot. Mindkét regiszterhez tartozik egy-egy D-latch, ebbe másolódik be a regiszterek értéke, és innen jut be az ALU-ba. Erre azért volt szükség, mert ha közvetlenül a regiszterek értéke lenne bekötve az ALU-ba, akkor amint megpróbálnám az ALU kimenetét az ACC regiszterbe írni, azzal az ALU egyik bemenetét is átírnám, aminek hatására változhat az ALU kimenete, és egy instabil állapot alakulhat ki. Ezen kívül van egy buffer az ALU és a busz között, hogy biztonságosan le lehessen választani az ALU kimeneteit a buszról.

Mivel két darab négy bites aritmetikai logikai egység van a rendszerben, az eredmény egy 8 bites érték és egy Carry bit. A rendes számítógépek rendelkeznek egy állapot regiszterrel is, ami többek között a gép aritmetikai-logikai egységének állapotáról tartalmaz információt. Gyakran találhatunk egy úgynevezett „zérus” bitet, ami azt jelenti, hogy egy művelet elvégzése után a célregiszter – az a regiszter, amibe a művelet eredménye íródik – értéke nulla. Ebben a gépben nincs külön állapot regiszter, viszont mindkét ALU rendelkezik saját zérus bittel. Ennek a két bitnek a logikai kapcsolatát felhasználva elő tudtam állítani egy zérus bitet, ami a teljes 8 bites számra érvényes, viszont nincs olyan utasítás, ami ennek a bitnek az értékét kérdezné le. Egyedül a feltételes ugrásnál van szerepe, ugyanis a gép rendelkezik egy „Jump if Not Zero” utasítással, ami egy hagyományos JUMP és a zérus bit logikai kapcsolatából áll.

3.6 Alaplap



9.: Az alaplapként szolgáló nyomtatott áramkör

Az egyes modulok összekötésére a konzulensem javaslatára egy régi számítógépből származó nyomtatott áramkört használtam. Az Önálló Projekt során megpróbáltam egyszerű 2.54mm-es header csatlakozókat használni, ám ezek nagyon kontaktosnak bizonyultak. Ez az áramkör elég nagy csatlakozókkal rendelkezik, és van a végén egy tüskesor is, ami megkönnyíti a jelek mérését, de akár egy logikai analizátort is köthetünk rá.

4. Szoftver

A gép életre keltéséhez két szoftverre is szükség volt. Az egyik, az egyszerűbb, a dekóder feltöltésére szolgált. A másik lényegében egy assembler, ami az assembly nyelvben írt kódot fordította le. Mindkét programot C nyelvben írtam, a Code::Blocks fejlesztői környezet segítségével.

A legelső lépés a szükséges instrukciók összegyűjtése volt. Ezeket egy táblázatba rendeztem, és minden egyes instrukció mellé odaírtam, hogy melyik lépésben milyen vezérlő jeleknek kell aktívnak lenniük. Ezek voltak a mikro-utasítások.

Name	Operands	Description	Steps (always 4 clock cycles)
NOP	-	No operation	PC2ABus | ABus2MR
0			Inst2Bus | Bus2IR
			IncPC
			0
HALT	-	Block the uOp Counter's clock pulse	PC2ABus | ABus2MR
1			Inst2Bus | Bus2IR
			IncPC
			HALT
ADD	-	Add B to ACC, and store the result in ACC	PC2ABus | ABus2MR
2			Inst2Bus | Bus2IR
			S3 | S0 | IncPC | Bus2ACC
			0
SUB	-	Subtract B from ACC	PC2ABus | ABus2MR
3			Inst2Bus | Bus2IR
			S2 | S1 | IncPC | Bus2ACC
			0

10.: Részlet az instrukciókat tartalmazó táblázatból

Ahogy a jobb szélső oszlopban látható, az egyes vezérlő jelek nevei között a programozásban használatos logikai VAGY kapcsolat szerepel. Ez megkönnyítette a dekóder program megírását, hiszen csak át kellett másolnom ezeket a sorokat a program megfelelő részeibe.

4.1. A dekóder feltöltéséért felelős program

Ahhoz, hogy három darab EEPROM-ból használható dekódereket készítsünk, először meg kell határozni, hogy melyik kimeneti lábhoz (azaz melyik bithez) melyik vezérlő jelet kötjük. Megtehettem volna, hogy kézzel írok minden egyes vezérlőjel mellé egy sorszámot, de ez azzal járna, hogy ha később új jeleket akarok hozzáadni a géphez, vagy egy meglévőt akarok eltávolítani, akkor kézzel kéne újraírnom a számokat. Ez nem csak időigényes, de könnyen elrontható, ami később jelentős hibákat eredményezne a programban.

Ezt a problémát a C programnyelv enumeráció („*enumeration*”, „*enum*”) típusával oldottam meg. Ez ránézésre hasonlít egy egyszerű listára. Abban tér el, hogy változók helyett konstans címkékkel töltjük fel, és az enumeráció minden egyes címkéhez hozzárendel egy értéket, ami nullától indul és eggyel növekszik. Ez nagyon hasznos például állapotgépekkel dolgozó beágyazott rendszereknél, ahol elég az egyes állapotok neveit felsorolni, de a *switch–case* utasítások megvalósítását is megkönnyíti. Én arra használtam, hogy minden vezérlőjelhez hozzárendeljek egy számot, ami azt jelzi, hogy hányadik bitnek felel meg a jel. Például a három EEPROM-ból a legelső (azaz a legalacsonyabb helyiértékű biteket tartalmazó) EEPROM nulladik bitje az ABus2PC vezérlő jelre van kötve (ez a buszon található értéket másolja a programszámlálóba). Mivel ez lesz az enumeráció legelső tagja, a nullás értéket kapja.

```
20      enum Signal {
21
22          //First EEPROM (0..7)
23          enABus2PC,
24          enIncPC,
25          enRST_PC,
26          enABus2MR,
27          enPC2ABus,
28          enBus2OPL,
29          enOPL2Bus,
30          enInst2Bus,
31
```

11.: Részlet a programban használt enumerációból

Az enumerációban csak sorszámok vannak, ezért ezeknek a logikai VAGY kapcsolata mást eredményezne, mint amire a lista alapján szükségem lenne. A VAGY kapcsolatokhoz nem sorszámokra, hanem olyan változókra van szükségem, amelyekben egyetlen 1-es bit van a megfelelő helyiérték biten. Hogy az enumerációkat meg tudjam különböztetni a helyiértékeket tartalmazó változóktól, mindegyik címke kapott egy „en” előtagot.

```
56      //First EEPROM
57      unsigned long int ABus2PC = 1 << enABus2PC;
58      unsigned long int IncPC = 1 << enIncPC;
59      unsigned long int RST_PC = 1 << enRST_PC;
60      unsigned long int ABus2MR = 1 << enABus2MR;
61      unsigned long int PC2ABus = 1 << enPC2ABus;
62      unsigned long int Bus2OPL = 1 << enBus2OPL;
63      unsigned long int OPL2Bus = 1 << enOPL2Bus;
64      unsigned long int Inst2Bus = 1 << enInst2Bus;
```

12.: A szükséges változók létrehozása

A VAGY kapcsolatoknál használt változók mind *unsigned long int* típusúak. Ezek 4 byte-os változók, amikben elfér a három EEPROM-hoz tartozó 3 byte-nyi adat. Az „*1 << en___;*” kódrész feladata, hogy egy 1-es bitet toljon el balra annyi helyiértékkel, amennyi az adott enumeráció értéke. Ez azért jó, mert ha fejlesztés során változtatni akarnék a vezérlőjelek sorrendjén (például két jel helyét megcserélném), akkor azt elég az enumerációban elvégezni.

Vegyük például az IncPC jelet. Ez a jel a programszámláló értékét növeli eggyel (*Increment Program Counter*), és a legkisebb helyiértékeket tartalmazó számláló második bitjénél helyezkedik el. Így az enumerációban másodikként szerepel egy enIncPC érték, és mivel az enumeráció a listákhoz hasonlóan nullától indexál, az enIncPC értéke 1 lesz. Ezután az *IncPC = 1 << enIncPC;* művelet egy 1-es értéket eggyel balra tol, így IncPC bináris értéke 0b10. Az utána következő változó értéke 0b100, az azt követőé 0b1000, és így tovább. Ezeknek a logikai VAGY kapcsolata már tényleg azt eredményezi, amit szeretnénk: a kívánt vezérlő jelek értékét állítja 1-re.

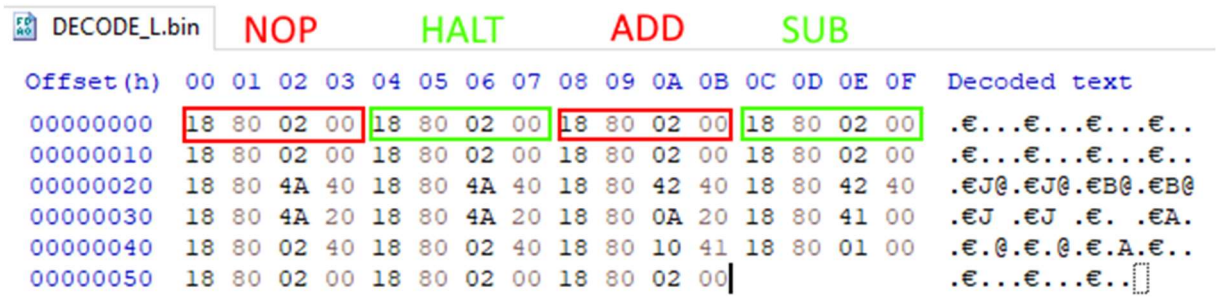
Ezután csak egy táblázatra van szükség, ami az instrukciókat mikro-utasításokra bontva tartalmazza. Mivel minden utasítás négy lépésből áll, az EEPROM-ok címzése elég egyszerűvé válik: egy cím alsó két bitje azt mutatja, hogy az adott instrukció hányadik lépésnél jár, a többi bit pedig az instrukció azonosítója.

```
88    unsigned long int Instruction[] = {
89
90    //NOP
91    PC2ABus | ABus2MR,
92    Inst2Bus | Bus2IR,
93    IncPC,
94    0,
95
96    //HALT
97    PC2ABus | ABus2MR,
98    Inst2Bus | Bus2IR,
99    IncPC,
100   HALT,
101
102   //ADD
103   PC2ABus | ABus2MR,
104   Inst2Bus | Bus2IR,
105   S3 | S0 | IncPC  | Bus2ACC,
106   0,
107
108   //SUB
109   PC2ABus | ABus2MR,
110   Inst2Bus | Bus2IR,
111   S2 | S1 | IncPC  | Bus2ACC,
112   0,
113
```

13.: Az első négy instrukció logikai kapcsolatokkal kódolva

A program mindhárom dekódoló EEPROM-nak egy külön fájlt hoz létre, amiket egy EEPROM-programozó segítségével lehet a chipekbe feltölteni.

Ellenőrzésképp egy bináris fájlszerkesztővel (HxD Hex Editor) megnyitottam a legalacsonyabb értékeket tartalmazó EEPROM-nak készült fájlt. Minden instrukció a „PC2ABus | ABus2MR” és az „Inst2Bus | Bus2IR” mikro-utasításokkal kezdődik. Az első lépés a programszámláló értékét a memória regiszterbe írja, hogy a memória címe a végrehajtandó utasításra mutasson. A második lépés eljuttatja a végrehajtandó instrukció azonosítóját a dekódolóba. Ezeknek a VAGY kapcsolatoknak a decimális értéke 24 és 8320. Ha csak az alsó byte-ot nézzük, akkor ebből decimálisan 24 és 128 marad, amiket hexadecimálisra váltva 0x18-at és 0x80-at kapunk. Ha a program jól működött, akkor a generált fájlban is ezt a két értéket kell látnunk négy byteonként megismételve:



14.: Az alacsony biteket tartalmazó dekóder EEPROM tartalma hexadecimálisan

A program nem lett elsőre tökéletes. Néhány appróbb hiba mellett a legnagyobb gondot az jelentette, hogy az egyik helyen egy byte helyett kettő került a fájlba, amitől az egész címzés elcsúszott. A debugger segítségével sikerült kideríteni, hogy ez az extra byte akkor jelenik meg, amikor decimális 10 (0xA) értéket próbálok írni. Minden egyes 0x0A elé megjelent egy 0x0D is.

Ez a Windows operációs rendszer miatt történt. Ha megnézünk egy ASCII táblázatot, láthatjuk, hogy a 0x0A karakter a soremelést („*Line Feed*”, programozásban gyakran ’\n’) jelenti. Azonban a Windows operációs rendszerben az új sort két karakterrel, egy „kocsi vissza” („*Carriage Return*”, ’\r’) és egy soremelés kombinációjával jelölik. A „kocsi vissza” ASCII értéke 0x0D, ez volt a látszólag semmiből megjelenő byte. A megoldás az volt, hogy az írni kívánt fájlt bináris fájlként kellett megnyitni. Ehhez nem volt elég a fájl kiterjesztését „.bin”-re állítani, hanem a programon belül is változtatni kellett az egyik paraméteren.

4.2. Assembler

A korábban írtak alapján látszik, hogy a dekóder számára a bejövő instrukciók csak címek, egészen pontosan kettes számrendszerben tárolt számok. Elméletben írhatunk úgy is programot, hogy a végrehajtani kívánt instrukciókat jelölő számokat, azaz az instrukció azonosítóját írjuk egymás után egy fájlba, amit aztán feltöltünk az instrukciókat tartalmazó EEPROM-ba. A gond az, hogy egy ilyen gépi kódban írt „program” egyáltalán nem lenne átlátható, minden egyes szám mellé írni kéne egy kommentet, ami elmagyarázná, hogy az adott szám éppen melyik instrukciónak felel meg.

Ezt a problémát oldja meg az Assembly nyelv, ami az utasítások azonosítóit emberek számára értelmezhető, rövid parancsokkal helyettesíti. Ezek többnyire olyan rövidítések, amik a parancs funkciójára utalnak, hogy könnyebb legyen megjegyezni őket. Mivel nekem csak az Atmel 8 bites mikrokontrollereivel volt gyakorlati Assembly tapasztalatom, az ott használt instrukcióneveket vettem alapul. Például a JMP parancs a „Jump” azaz „Ugrás” instrukció rövidítése.

Az assembler programhoz nagyészt ugyanazokat a fájlkezelő megoldásokat alkalmaztam, mint a dekóderhez. A fő különbség, hogy itt nem csak írni kell fájlokat, hanem olvasni is. Maga a program egy konzolos applikáció, az indításkor megjelenő konzol csak a fordítani kívánt fájl nevét várja el bemenetnek. Ha a fordítás sikeres, akkor három bináris fájlt kapunk: egyet az instrukciókat tartalmazó EEPROM-ba, és kettőt az operandusokat tartalmazó EEPROM-okba kell majd feltölteni.

Egy egyszerű kód így néz ki:

```
nop
in
ldib 8
add
out
jmp 0
```

Ez a kód a következőt csinálja:

1. vár egy órajel cikluson át
2. beolvassa az IN csatlakozó lábaira kötött értéket az ACC regiszterbe
3. a B regiszterbe betölti a nyolcas számot (0b1000)
4. a B regiszterben tárolt számot hozzáadja az ACC regiszterhez
5. kiírja az ACC regiszterben tárolt új eredményt az OUT csatlakozóra
6. visszaugrik a nullás memóriacímre, azaz a kód elejére, és előröl kezdi a programot

Ezt a kódot egy szöveges fájlban (.txt) kell tárolni. Ebből az assembler szavanként olvassa be a fordítani kívánt kódot, és minden szót külön „*string*”-ként kezel, mert ezzel a változótípussal könnyű dolgozni. A program egy nagy *if – else if* elágazás-rendszer alapján kikeresi, hogy a beolvasott sor melyik instrukciót tartalmazza, és ezalapján milyen értékeket kell írni az EEPROM-okba. Először egy *switch-case* struktúrát próbáltam használni, de végül ezt a megoldást átláthatóbbnak találtam.

```
96      if(!strcmp(instr , "nop")){  //strcmp == 0, if strings are equal
97          INST = 0;
98          OPH = 0;
99          OPL = 0;
100     }
101     else if(!strcmp(instr , "halt")){
102         INST = 1;
103         OPH = 0;
104         OPL = 0;
105     }
106     else if(!strcmp(instr , "add")){
107         INST = 2;
108         OPH = 0;
109         OPL = 0;
110     }
```

15.: Részlet az assembler programból. Az INST változó az Instrukció EEPROM-ba szánt értéket, OPH és OPL az Operandus EEPROM-ba szánt értéket tartalmazza.

Az olyan instrukciókat, amik operandust is tartalmaznak, kicsit másként kell beolvasni. Az instrukció után magát az operandust is be kell olvasni. A program az operandust mindig egy decimális számként kezeli, és egy két byte széles változóban tárolja. Ennek a változónak a felső byte-ja kerül az operandus felső byte-ját tartalmazó EEPROM-ba, az alsó byte-ja pedig az alsó byte-ért felelős EEPROM-ba. Az olyan instrukcióknál, ahol nincs szükség operandusra, az operandus helyére mindig nullák íródnak, ahogy az az előző ábrán látszik.

```
136     else if(!strcmp(instr , "lda")){
137         INST = 8;
138         fscanf(fp_i, "%d", &operand);  //get the operand
139         OPH = operand >> 8;      //get the high byte of the operand
140         OPL = operand & 0x00FF; //get the low byte of the operand
141     }
```

16.: Egy operandust tartalmazó instrukció kezelése

Az eredmény három darab bináris fájl, amiket a három darab memória EEPROM-ba kell feltölteni. Ha a program nem tartalmaz olyan JUMP utasítást, ami egy byte-nál nagyobb címre mutat, akkor az operandus magas byte-ját tartalmazó EEPROM üres lesz, így elég csak az alacsony byte-ot, és az instrukciókat tartalmazó EEPROM-okat berakni a gépbe.

5. Tesztelés, hibák és javítások

Egy ilyen bonyolultabb rendszernél nem valószínű, hogy sikerül mindent elsőre hibátlanul megtervezni. Mivel a rendszert különálló nyákokra bontottam, először a különálló részeknek a működését ellenőriztem le. Az Önálló Projekt II. során az egyik nyák bekapcsolásakor sikerült tönkretennem a bemeneti LM7805-ös feszültségszabályzót, így most valamivel óvatosabb voltam. Minden tesztet egy labortáp segítségével végeztem, a LED-eket is tartalmazó modulokon 50 mA-es, a többin 30 mA-es áramkorláttal. Ennek elégnek kell lennie ahhoz, hogy működtesse őket, de egy esetleges zárlat vagy hiba esetén még nem valószínű, hogy tönkremenne valamelyik alkatrész. A legrosszabb hibaállapot az lehet, amikor az adatbuszra több alkatrész is egyszerre próbál jelet küldeni, és az egyik alacsony, a másik magas értéket állítana be. A tesztelő modul ellenállásai ebben az esetben is korlátozzák az áramot, de ha mégse lenne elég, ebben a verzióban minden integrált áramkör saját DIP foglalatot kapott, hogy viszonylag könnyen lehessen őket cserélni.

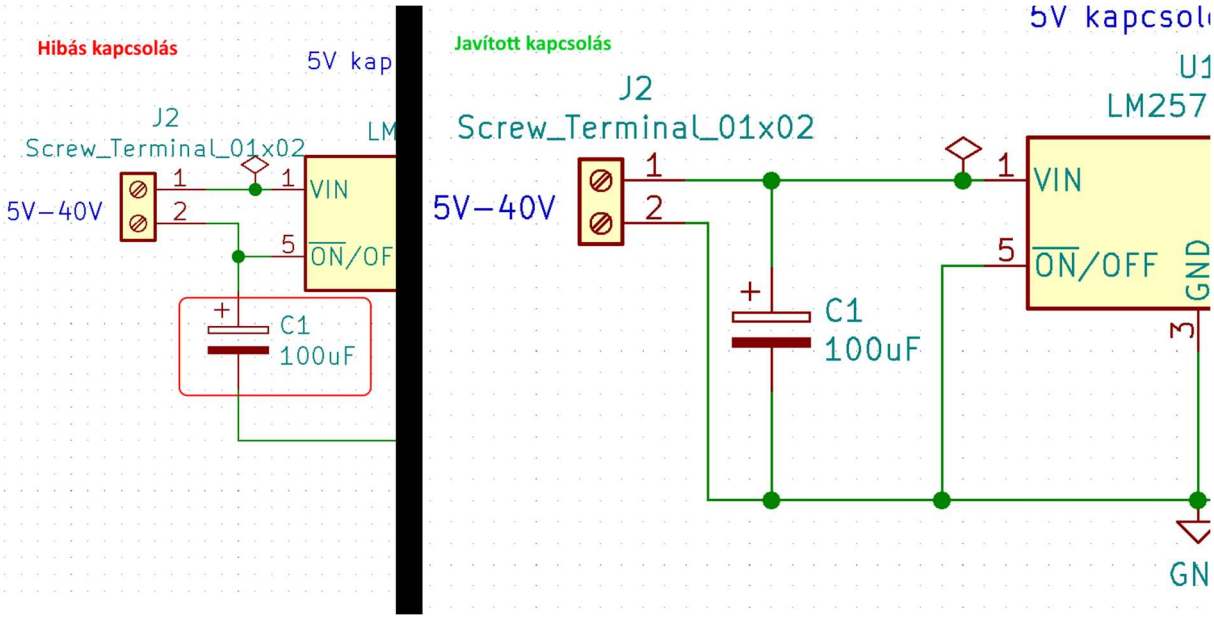
5.1. Tesztelő modul

Először a tesztelő modult vizsgáltam. Ez a legegyszerűbb áramkör, ráadásul ez adja a tápfeszültséget és az órajelet a többi modulnak is, illetve a kapcsolók segítségével a többi nyák ellenőrzését is jelentősen megkönnyíti. Bekapcsoláskor egy elég szokatlan működést tapasztaltam a kimenő oldalon:

17.: A tesztelő modul kimenő feszültsége az első bekapcsolás során

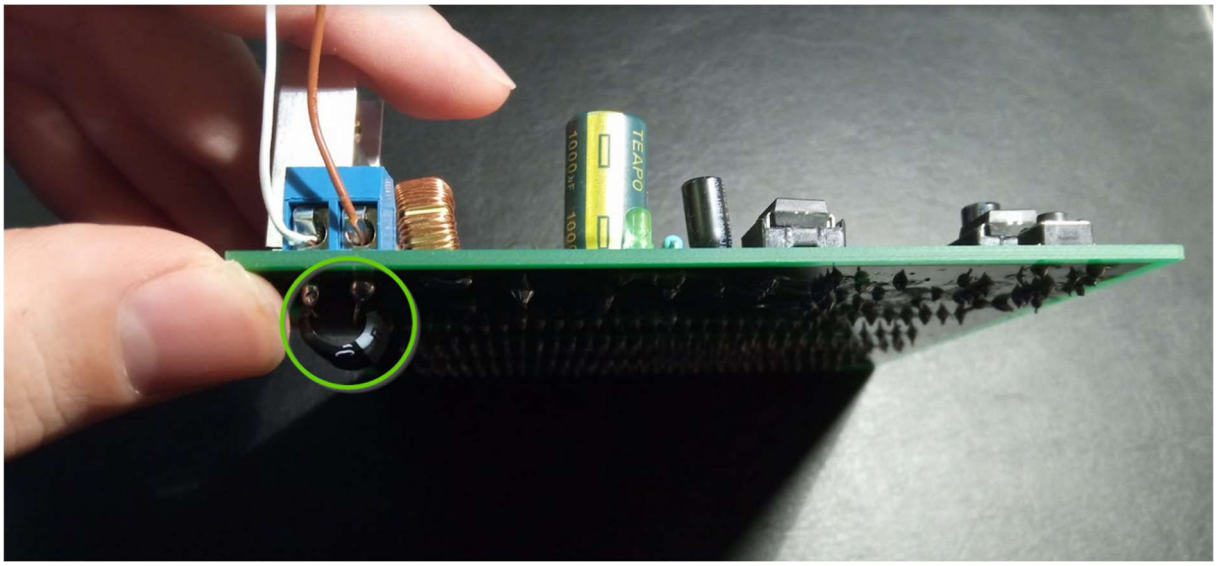
Az Önálló Projekt során egy egyszerű LM7805-ös tápegységet használtam, ami egy népszerű lineáris feszültségszabályzó. Hátránya, hogy minél nagyobb a bemenő és a kimenő feszültsége között a különbség, annál nagyobb teljesítmény disszipálódik el rajta. A számítógép minden integrált áramköre 5 V-ról működik, és úgy terveztem, hogy egy 12V-os laptop töltőről is meg lehessen táplálni, ami már egy jelentős feszültség különbséghez vezetne. Ha az összes LED egyszerre világít, akkor azok önmagukban több, mint 200 mA-t vesznek fel, ami nagyjából 1,5 W teljesítményt disszipálna a lineáris feszültségszabályzón.

A hatékonyság nevében ebben a projektben egy kapcsolóüzemű tápegységet használtam, konkrétan egy *LM2576T-5*-öt. Ennek a kimenő feszültsége fixen 5V, ráadásul beépített védelemmel rendelkezik rövidzár és túlmelegedés ellen. Amikor először bekapcsoltam a modult, a kimeneten csak tüskéket láttam, amik szabályosan 5,6V-os amplítúdóval és 127Hz-es frekvenciával jelentek meg. A jelalak és a frekvencia is annyira stabil volt, hogy azt hittem, a túláramvédelem léphetett működésbe, mivel a szakmai gyakorlatom során volt dolgom egy olyan integrált áramkörrel, ami ehhez hasonló módon reagált egy túláramra vagy rövidzárlatra. Ebből kifolyólag hosszú időn át tanulmányoztam az LM2576T adatlapját, mire rájöttem, hogy a hiba a kapcsolási rajzomban van.



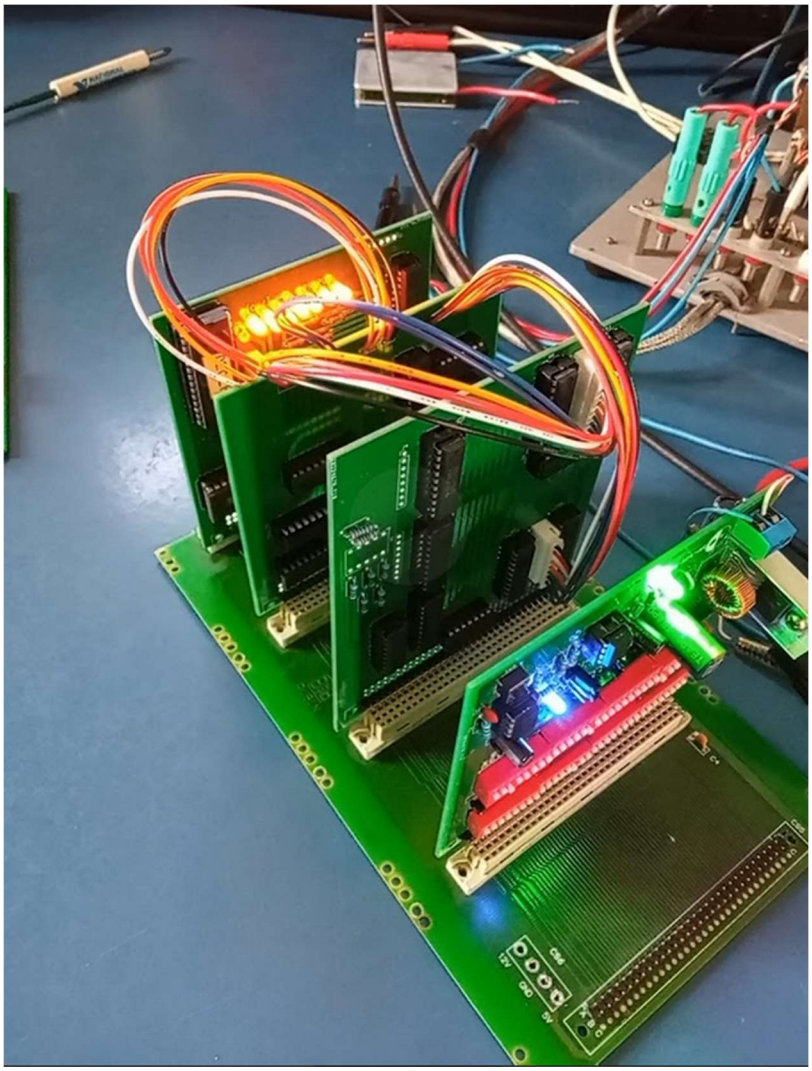
18.: A tesztelő modul kapcsolási rajza javítás előtt és után

A tápegység bemenetére kellett rakni egy 100 uF-os kondenzátort, de úgy tűnik, ezt utólag vettem észre, és elég hanyag módon helyeztem el. A bemenő sorkapocs két lába közé kellett volna bekötni, de sikerült az egyik földvezeték helyére raknom. Ezzel gyakorlatilag egyenáram szempontjából nem is volt bekötve a föld, és ez okozta a szokatlan viselkedést. Szerencsére ezt a problémát egy kis forrasztással könnyen meg lehetett oldani, és így már szépen működött az áramkör.



19.: A bemenő kondenzátor fektetve nem foglalt sok helyet

Ezután külön-külön bekötöttem a tesztelő modul mellé a többi nyákot is. A memória modult leszámítva egyikkel se volt gond. Végül a négy működő modult egyszerre is beraktam a megfelelő csatlakozókba, hogy megnézzem, mennyire működnek egyszerre. Ekkor még nem volt kész a fordító program, így nem tudtam programot futtatni a számítógépen, ehelyett a tesztelő modul segítségével állítottam be parancsokat az adatbuszra, majd ellenőriztem, hogy a dekóder helyesen dekódolja-e őket. Ehhez egy multimétert használtam, amivel megmértem a dekódolón az egyes jelek szintjeit. A NOP és az ADD utasításokat néztem végig, és ezeknek minden mikroutasításuk helyesen dekódolódott, és a dekódolandó mikroutasítás címe az órajellel szinkronban nőtt, pont úgy, ahogy kell.



20.: Az összeszerelt számítógép a memória modul nélkül. A közel látható kék LED az órajel szintjét jelzi, míg a távolabbi sárga LED-ek a dekóder címét mutatják.

5.2. Memória modul

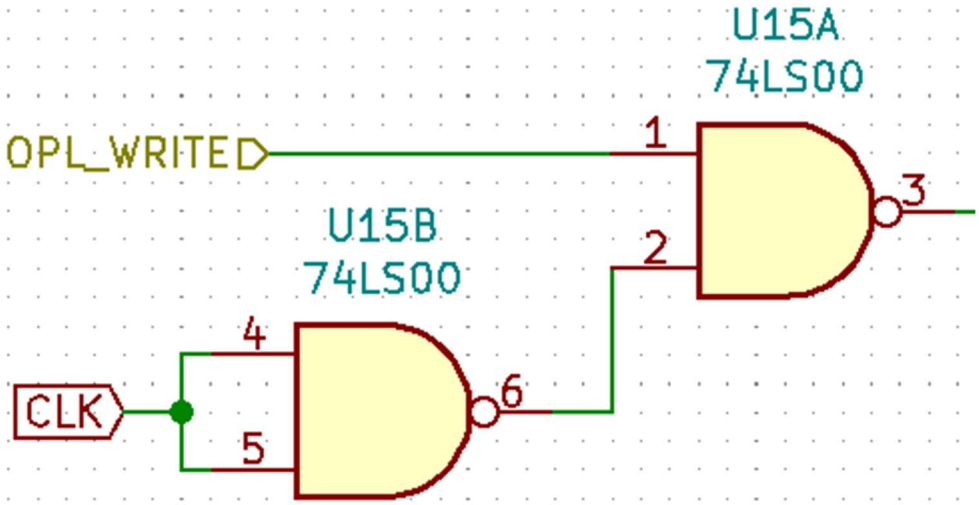
Amikor a memóriát próbáltam tesztelni, a tápegység áramkorlátba került, és nem világított az 5V-os feszültség meglétét jelző LED. Első ránézésre úgy tűnt, hogy egy szimpla zárlattal van dolgom, úgyhogy először a forrasztásokat ellenőriztem mind a csatlakozón, mind a DIP foglalatokon. Ezután megpróbáltam egy másik csatlakozóba bekötni, de az se segített. Itt már tervezési hibára gyanakodtam, úgyhogy elkezdtem átnézni a kapcsolást, valamint a felhasznált alkatrészek adatlapjait.

Az Önálló Projekt tárgyon szerzett tapasztalatom alapján attól tartottam, hogy itt is egyszerre több integrált áramkör próbált adatot írni a buszra, de az adatlapok és a kapcsolási rajzok alapján minden vezérlő jelet megfelelően kötöttem be.

Itt bizonyultak igazán hasznosnak a DIP foglalatok. Először is készítettem egy pár fényképet, hogy később egyszerűbb legyen az összeszerelés, majd egy pár laposfejű csavarhúzóval minden alkatrészt kivettem a foglalatából. Így üresen is rákötöttem az áramkört és a tesztelő modult az alaplapra, és alkatrészek nélkül nem került áramkorlátba a tápegység. Ezek szerint nem a forrasztással volt a gond.

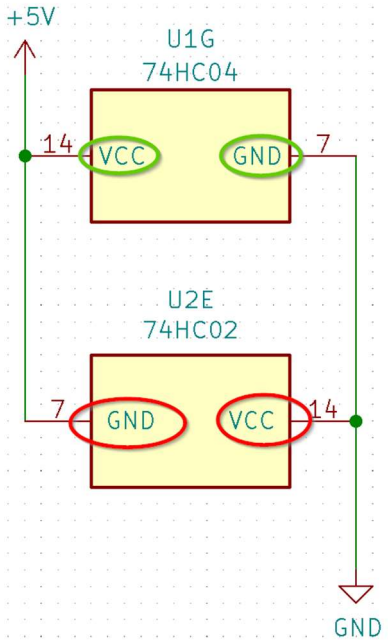
Ezután elkezdtem visszahelyezni az integrált áramköröket, és minden egyes berakott alkatrész után 20 mA-es áramkorlát mellett kapcsoltam feszültséget a számítógépre, és figyeltem, hogy mikor jelenik meg a hiba. Így sikerült megtalálnom, hogy a probléma néhány logikai integrált áramkörnél, illetve az alacsony memóriacímet tartalmazó regiszternél van. Először azt hittem, hogy elrontottam a logikai hálózatot, de szerencsére túl jól tanították a Moore-algebrát ahhoz, hogy ilyen hibát vétsek. Ehelyett ismét figyelmetlenségből rontottam el valamit.

Egy kapcsolási rajzon a logikai áramköröket a legtöbb integrált áramkörtől eltérő módon szokás jelölni. Hagyományos esetben egy téglatestet rajzolnak, rajta az egyes lábakkal, a lábak számozásával, és jó esetben valami címkével, ami megmondja, hogy melyik lábnak mi a funkciója. A logikai kapukból többnyire négyet, vagy többet szoktak egy tokba integrálni, viszont sokkal praktikusabb őket külön-külön szimbólumokként kezelni. Emiatt ezeket az alkatrészeket több részre bontják, minden egyes kapu külön szimbólumot kap megfelelő lábkiosztással, illetve van még egy szimbólum az áramforrásért felelős lábaknak.

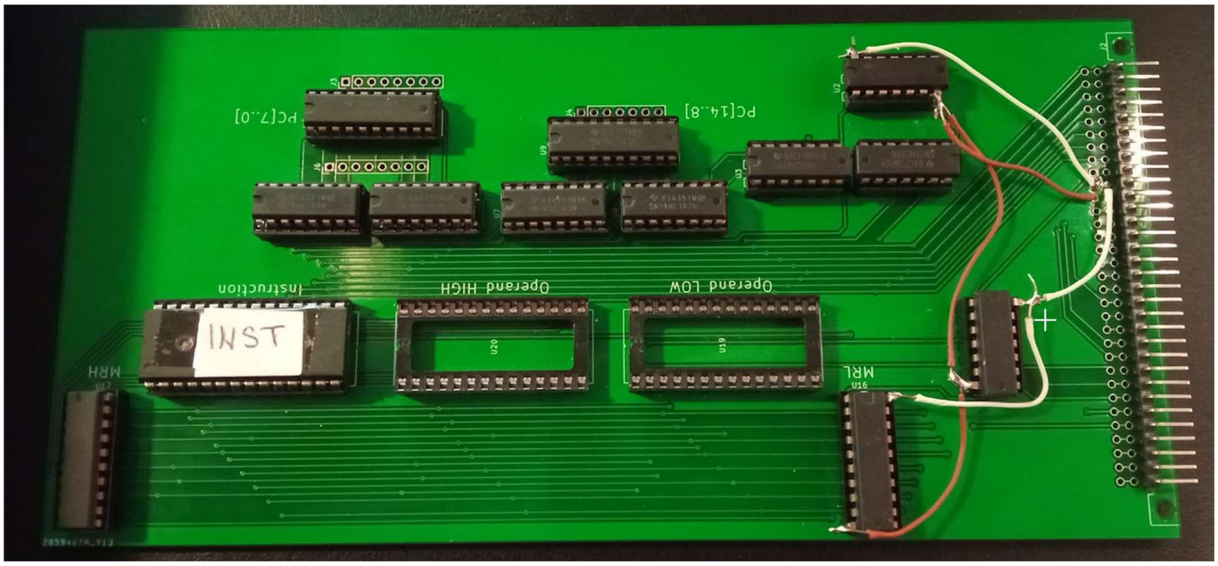


21.: Egy 74LS00 integrált áramkör négy NAND kapuból áll, amiből itt kettő látszik. Jelölésüket az IC azonosítója (U15), és egy betű (A, B) kombinációja adja.

Az okozta a szokatlan áramfelvételt, hogy néhány alkatrész szimbólumainál fordítva voltak az 5V-ot és a földet jelölő lábak, és nem vettem észre, amikor bekötöttem őket. A kapcsolási rajzon ezt elég egyszerű kijavítani, de a kész nyákon már forrasztani kellett. Az egyes integrált áramkörök GND és VCC lábait meghajlítottam, úgy, hogy kilógjanak a DIP foglalat fölött, majd ezeket a lábakat vezetékeken keresztül tápláltam meg.



22.: A 74HC04 és a 74HC02 szimbólumain ellentétes oldalon van a föld és a tápfeszültség



23.: Némi forrasztással ezt is meg lehetett oldani. A földet és a tápfeszültséget külön drótokkal vezettem az integrált áramkörök lábaihoz.

Így már nem került azonnal áramkorlátba a tápegység, úgyhogy a többi modult is bekötöttem. Egy NOP utasítás dekódolásának legelső lépésénél jelentősen megnőtt az áramfelvétel, de nem indultak újra a számlálók, és a többi lépésnél nem volt ilyen gond. Azokon a tesztpontokon, amik a memória regiszter bemenetéhez vezetnek, csak 3V-körüli értékeket mértem. Ezek össze vannak kötve az adatbusszal és az operandus alacsony byte-ját tartalmazó EEPROM-mal, úgyhogy kivettem az EEPROM-ot, de az nem segített.

A következő alkalommal megpróbáltam először csak a memória modult, a tesztelőt, és a dekódert bekötni. Így nem volt probléma az áramfelvétellel, és a feszültségszintek is rendben voltak. Ezután bekötöttem a regiszter modult, ami továbbra is jól működött, úgyhogy az ALU-t is csatlakoztattam. Kiderült, hogy ez okozta a gondot. Működés szempontjából az ALU a legkevésbé kritikus, hiszen, ha működik a gépen lévő egyszerű tesztprogram, akkor is kéne annyit látani, hogy egymás után más-más instrukciókat hajt végre a gép, és azt is, hogy adat kerül a regiszterekbe.

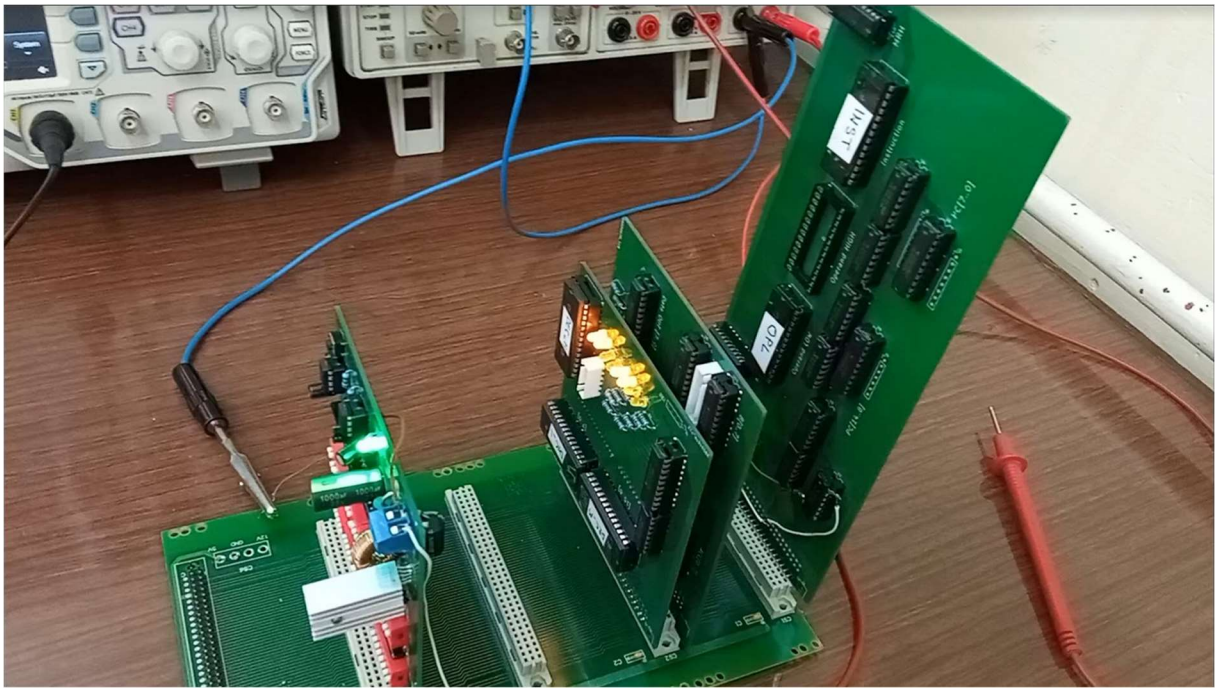
Ez nem egyelőre valósult meg. A gép ALU nélkül is csak egy NOP utasítást hajtott végre újra és újra. Először úgy tűnt, hogy a programszámlálóval, vagy az azt vezérlő logikai kapukkal lehet a gond, de végül kiderült, hogy a memória cím regiszter nem működött rendesen. A cím alacsony byte-ját tartalmazó regisztert sikerült a legutóbbi mérés során fordítva berakni a helyére, és ez tönkretette.

Szerencsére az ALU-ban is van egy pár ugyanilyen típusú integrált áramkör, és mivel az ALU-t úgyse használtam, egyszerűen levettem róla a szükséges alkatrészt, és áttettem a memória regiszterre. A megfelelő áramellátást ismét vezetékek beforrasztásával oldottam meg.

Emellett kijavítottam egy másik jelentős problémát: a dekóder EEPROM-jait korábban úgy programoztam fel, hogy a nem használt címeken csupa nulla legyen a vezérlő jelek értéke, nehogy olyan állapotba kerüljön a gép, ami kárt okozhat. Viszont a mérések során rájöttem, hogy ha a dekódoló címregisztere egy nem használt címre mutat (azaz egy olyan számra, amihez nem tartozik instrukció), akkor onnan nem fog tudni kilépni, hiszen, ha minden vezérlő jel nulla, akkor nem is fog olyan jel érkezni, ami frissítené a címregisztert. Ráadásul ezt a tesztelő modul kapcsolóival se lehet megoldani, ugyanis amikor a dekódoló egység is be van kötve, akkor az EEPROM-okból érkező vezérlő jelek megzavarják a tesztelő modullal beállított jeleket. Ezt úgy oldottam meg, hogy a dekódoló EEPROM-ok kihasználatlan címeit feltöltöttem NOP utasításokkal. Nem számít, hogy milyen címen van a dekódoló címregisztere a számítógép bekapcsolásakor, a gép be fogja tudni olvasni a következő instrukciót. Ehhez elég volt az EEPROM-ok tartalmát generáló programot bővíteni.

A következő tesztelés során először csak a tesztelő modult, a dekódert és a memóriát kötöttem be. Miután csatlakoztattam a gépet a labortáphoz és bekapcsoltam a tápegységet, a több hónapos munka után először futott egy program a számítógépen. A dekóder LED-jein látszott, hogy nem csak ugyanazt az instrukciót hajtja végre újra és újra, hanem egymás után más-más parancsok kerülnek be. Ráadásul ez nem csak valami véletlenszerű működés volt, mivel többször is újraindítottam a gépet, és minden alkalommal ugyanazok az instrukciók hajtódtak végre ugyanabban a sorrendben.

Ezek szerint működik a projekt legnehezebb része. Még egy csomó további tesztet és javítást lehetne végezni, és jó lenne az ALU-t is működésre bírni, hiszen anélkül nem valami hasznos egy számítógép, de ezekre már nem maradt időm a szakdolgozat leadási határideje miatt. A félév vége felé egyre jobban féltem attól, hogy nem sikerül időben megtalálnom a problémát, és a vizsgabizottság előtt kell megindokolnom, hogy miért nem működik a gépem. Így viszont sokkal magabiztosabban várom a záróvizsgát.



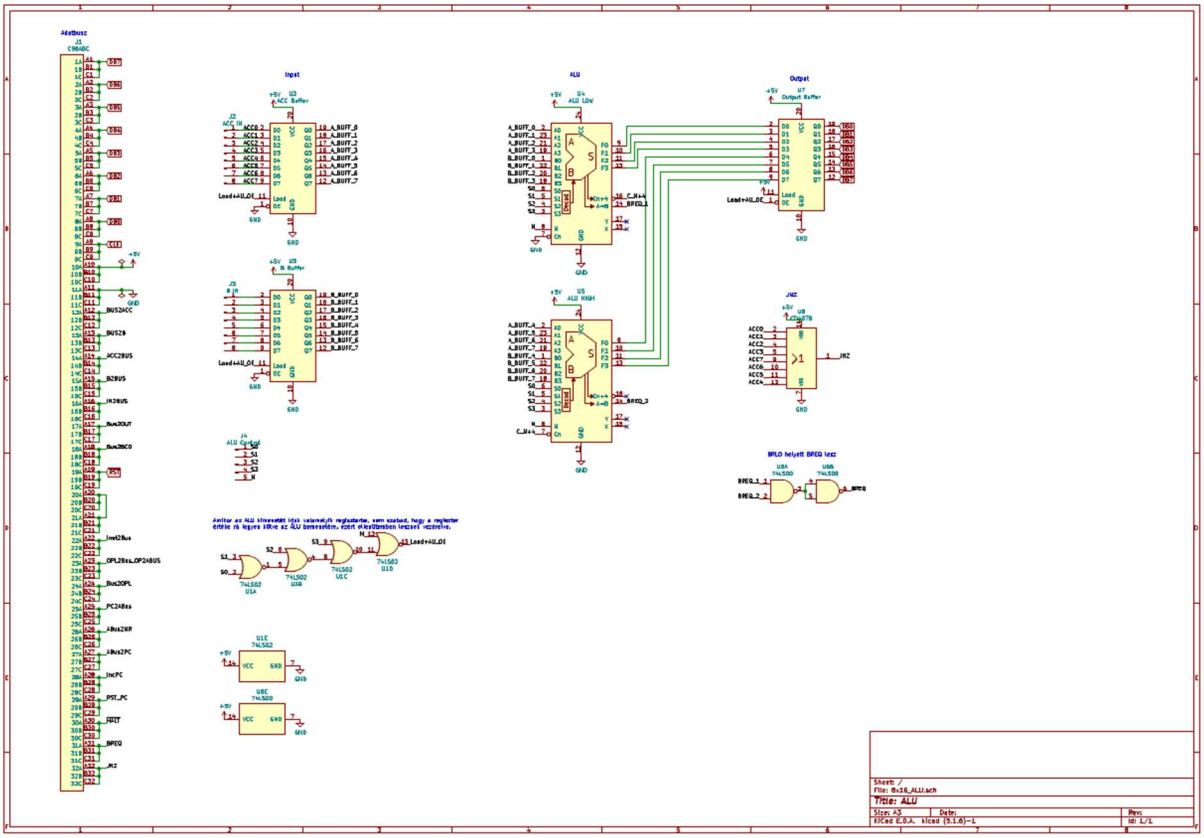
24.: A számítógép az első sikeres teszt közben. Az ALU kivételével minden be van kötve.

6. Mellékletek

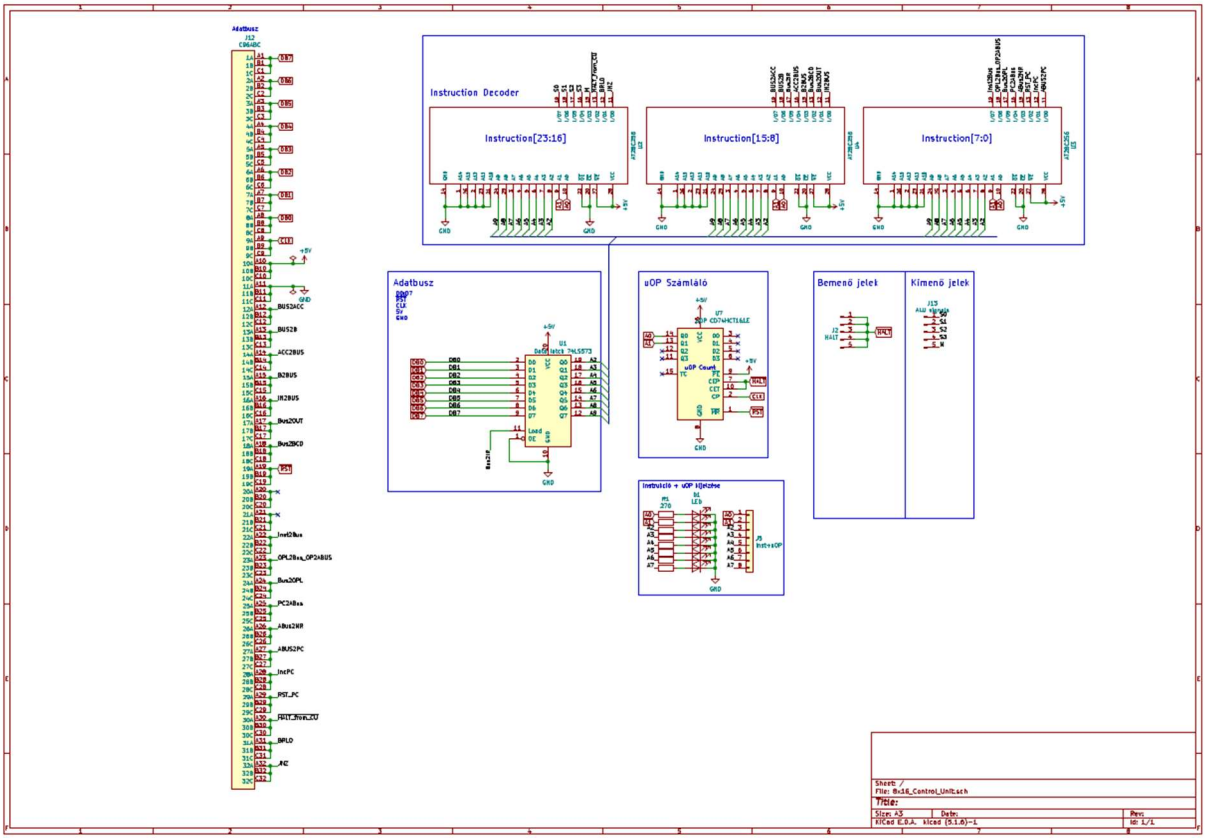
6.1 Kapcsolási rajzok

A mellékelt kapcsolási rajzokon már kijavítottam a korábban talált hibákat. Ezeket a dokumentumokat nem lehetett teljes felbontásban hozzáadni ehhez a szakdolgozathoz, ezért külön fájlokban is mellékelem őket.

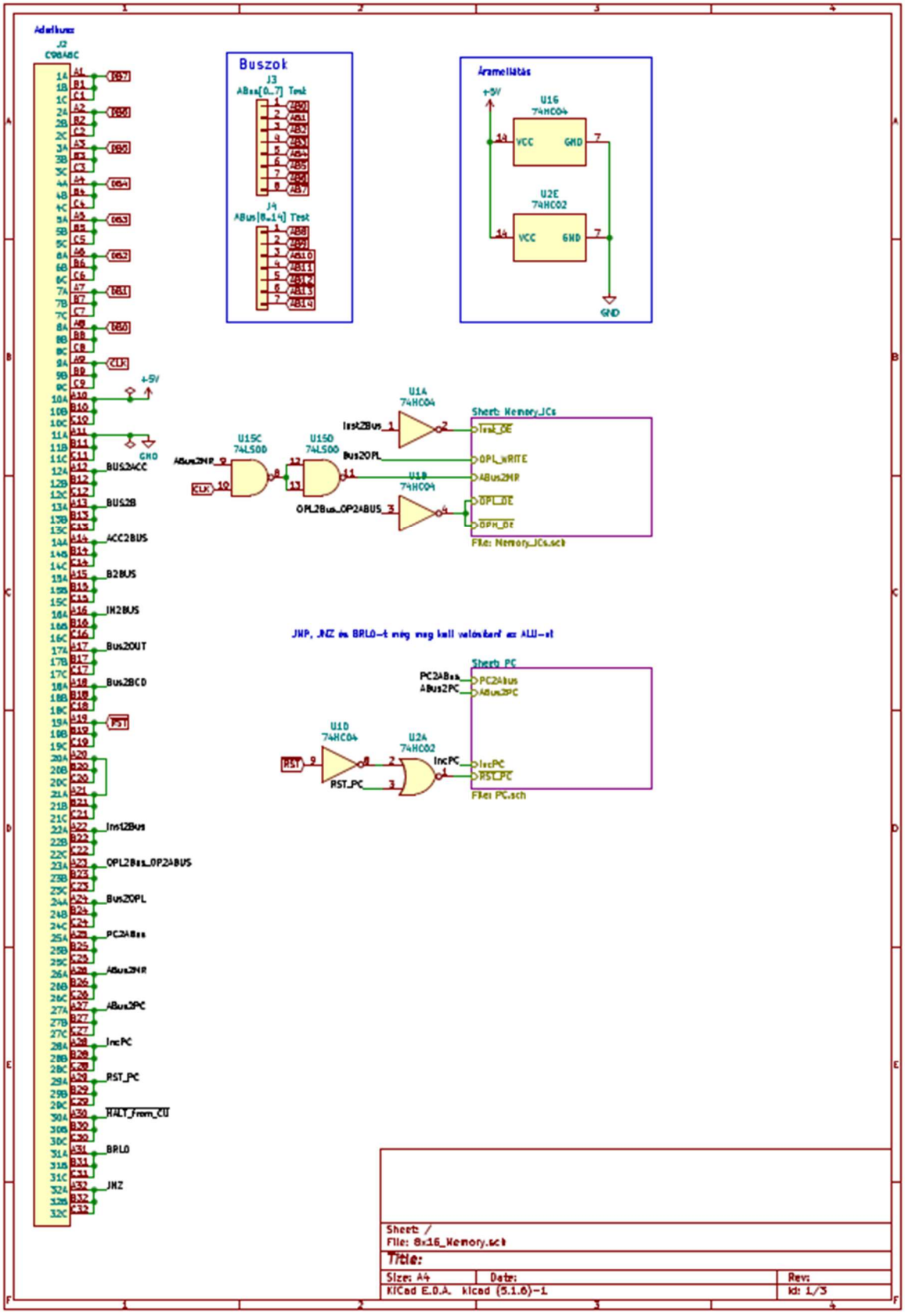
ALU kapcsolási rajza

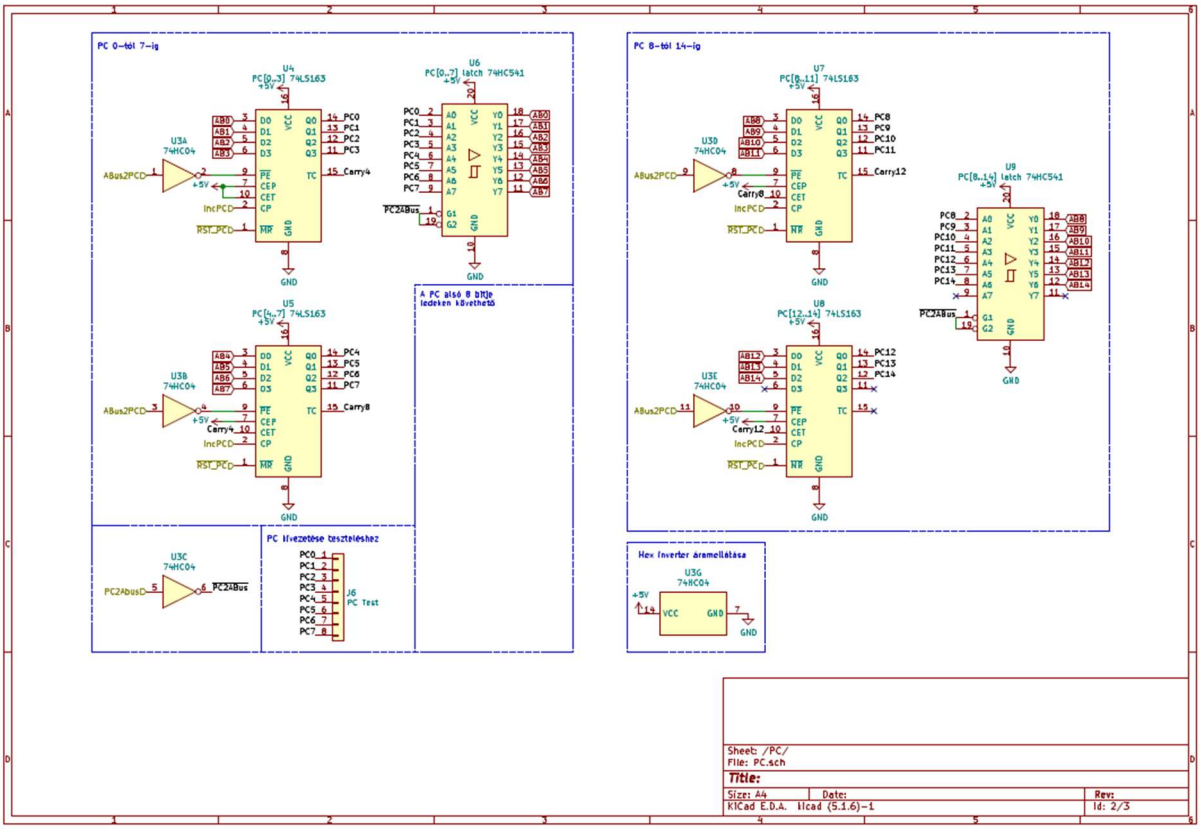


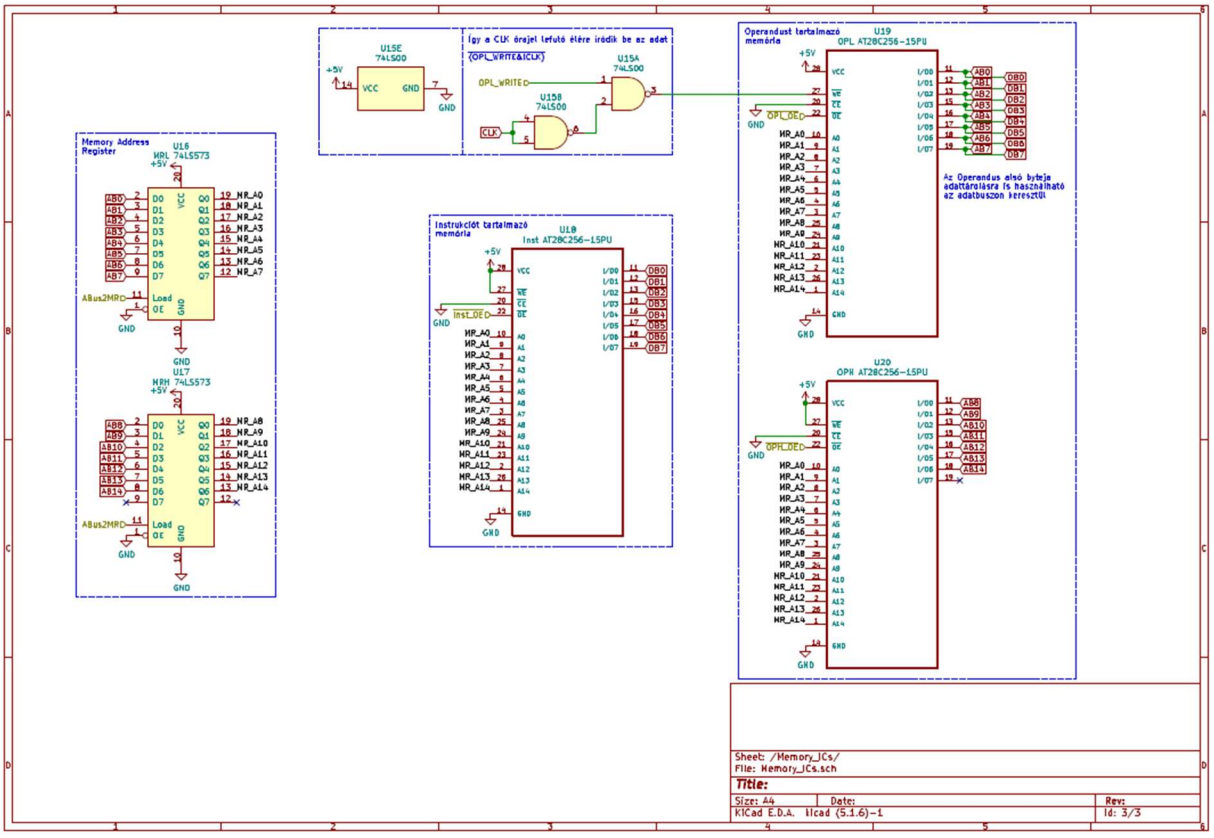
Vezérlő egység / dekódoló kapcsolási rajza



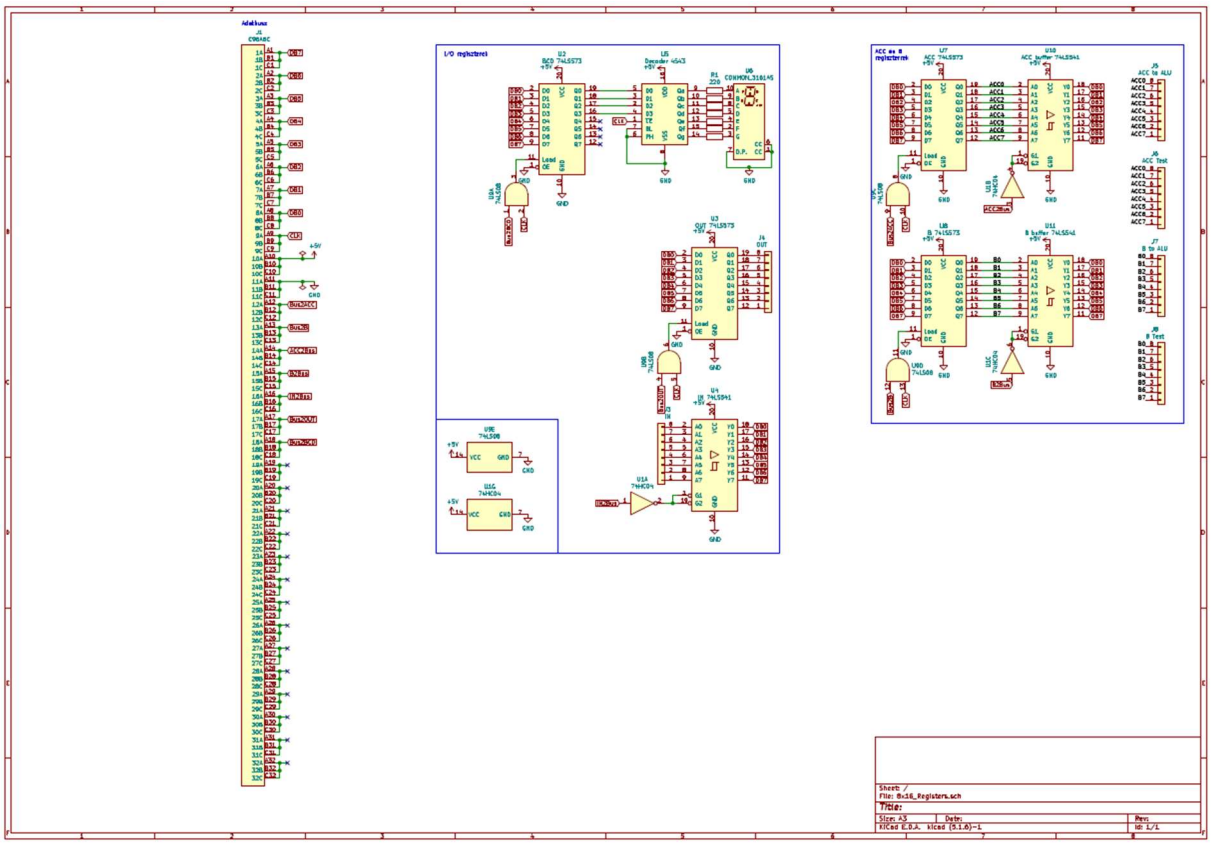
Memória kapcsolási rajza (39. - 40. oldal)



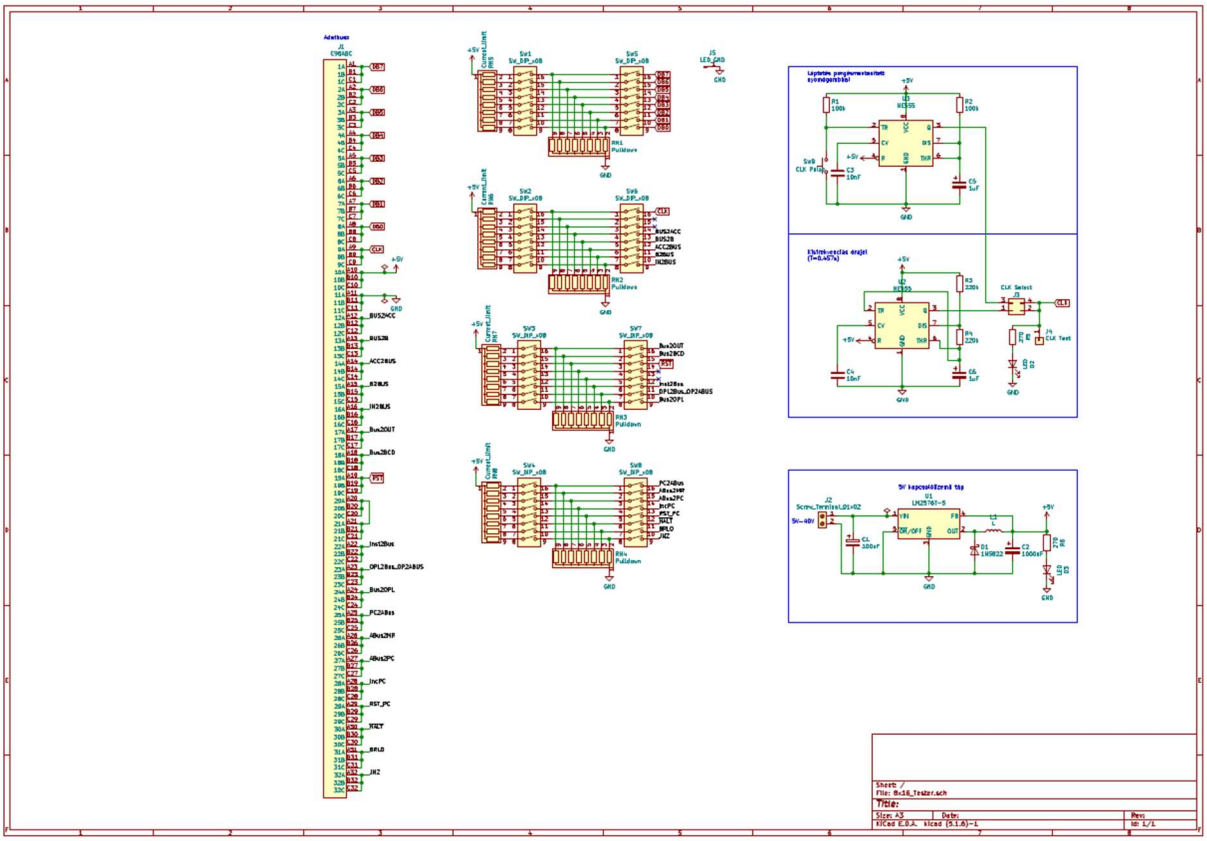

PC 0-tól 7-ig
PC 8-tól 14-ig
A PC alsó 8 bitje kiolvasható
PC kivezetése teszteléshez
Hex inverter áramellátása
Sheet: /PC/
File: PC.sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. kicad (5.1.6)-1
Id: 2/3


Így a CLK órajel lefutó élére íródik be az adat
Operandust tartalmazó memória
Az Operandus alsó byteja adattárolásra is használható az adatbuszon keresztül
Instrukciót tartalmazó memória
Memory Address Register
Sheet: /Memory_ICs/
File: Memory_ICs.sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. kicad (5.1.6)-1
Id: 3/3

Regiszterek kapcsolási rajza



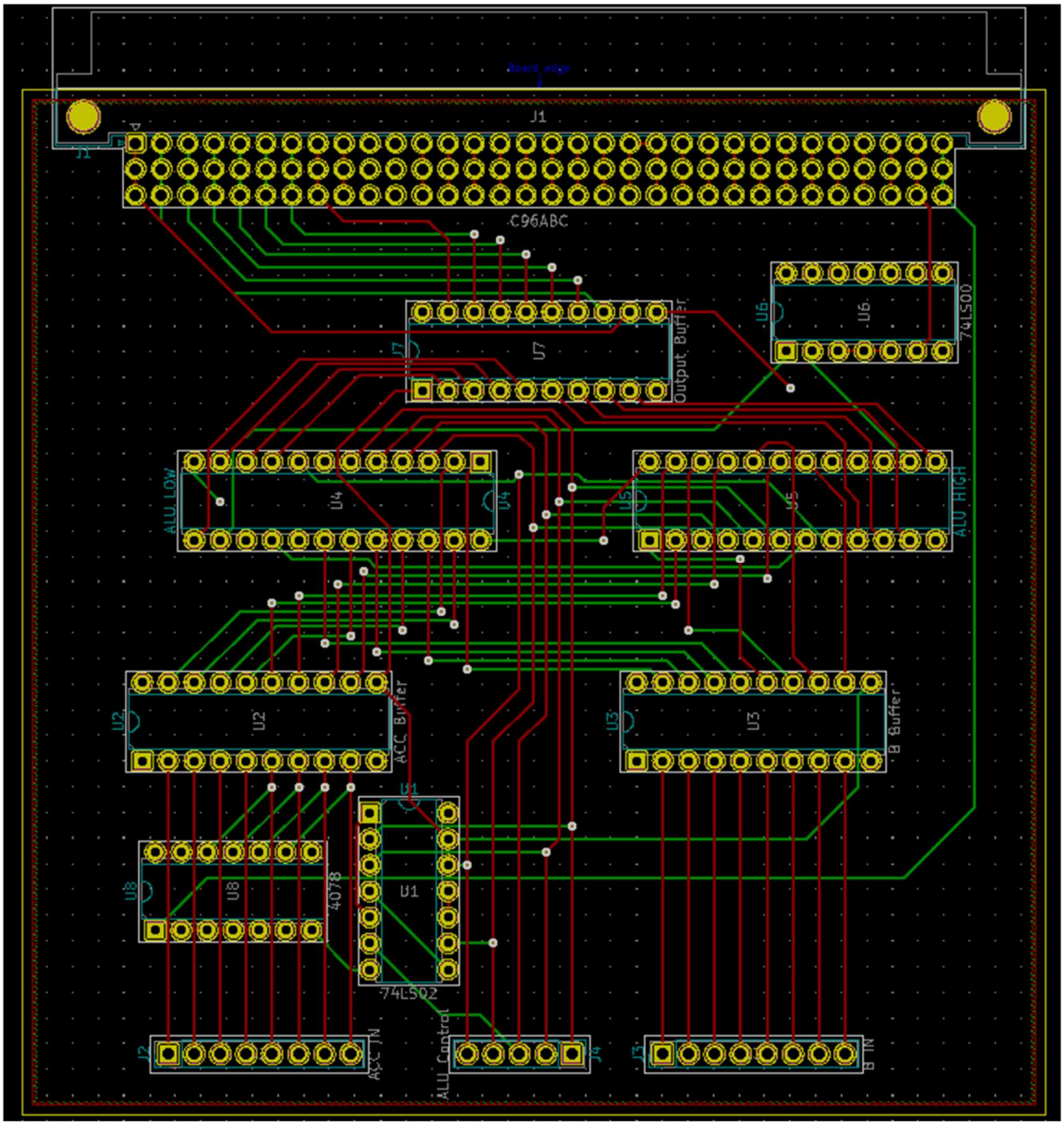
Tesztelő modul kapcsolási rajza



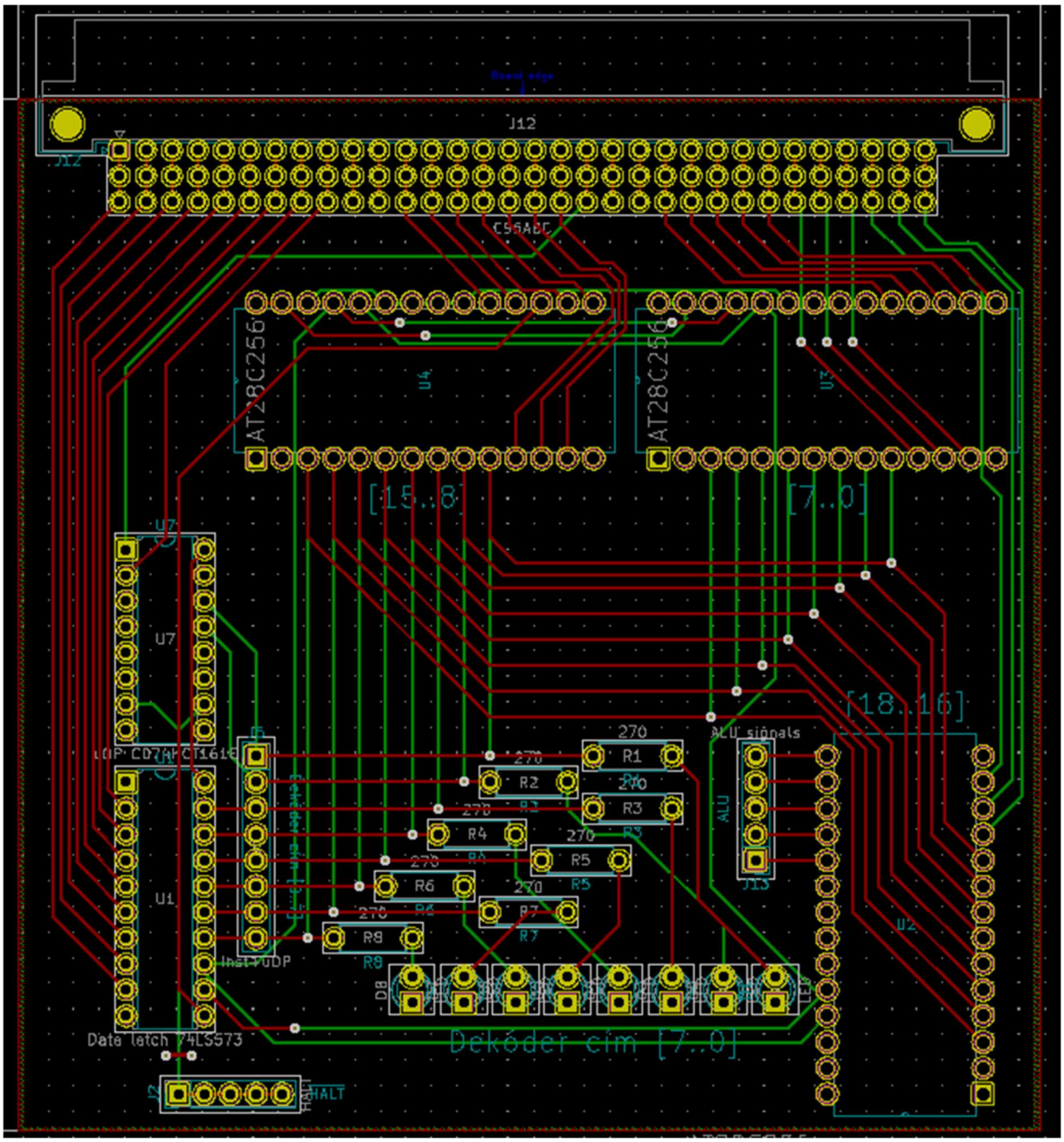
6.2 Nyomtatott áramkörök

Itt minden áramköri lap olyan állapotban van, ahogy a gyártótól rendeltem őket. Az egyes rétegek kitöltéseit elrejtettem, hogy az alsó rétegen is látszódjon a huzalozás.

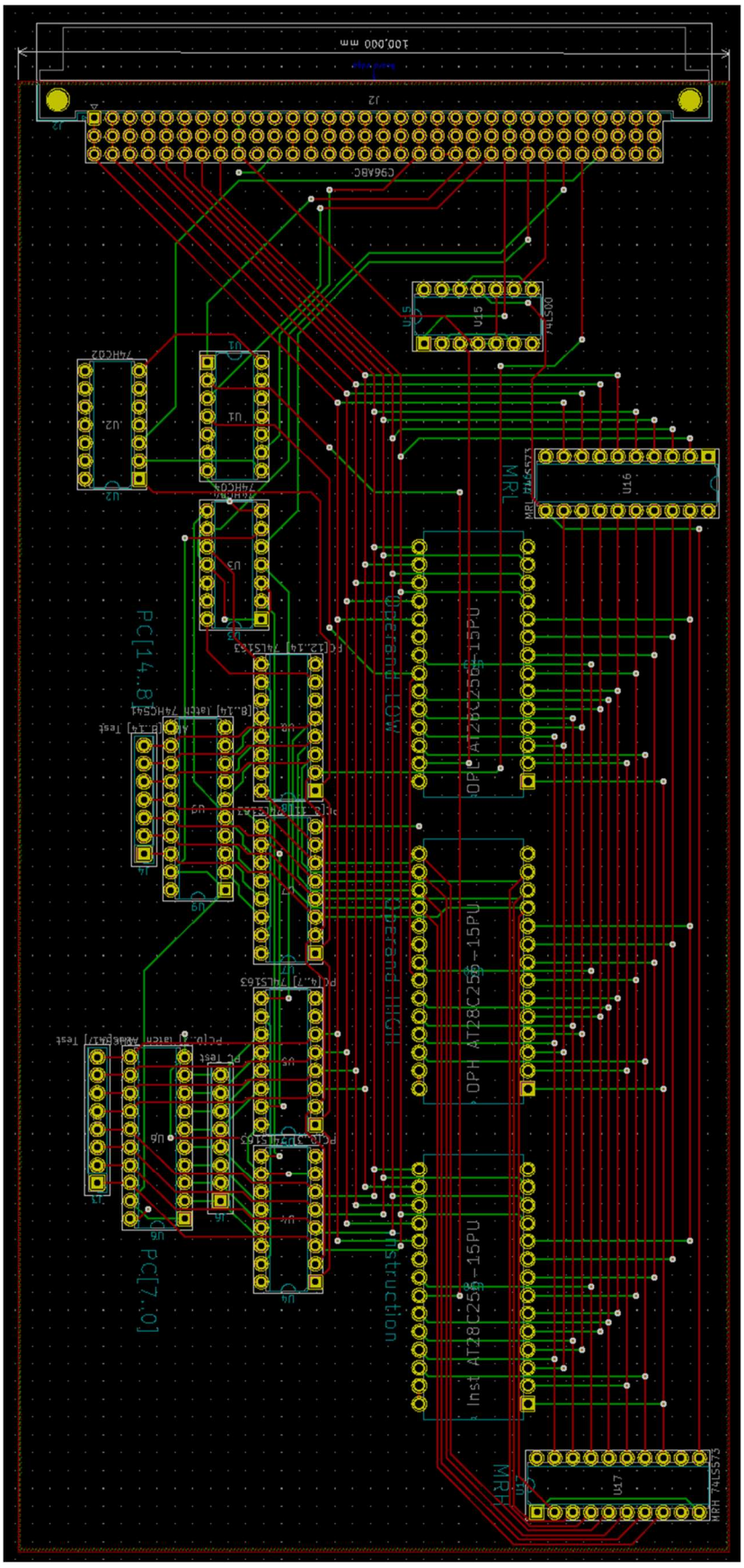
ALU



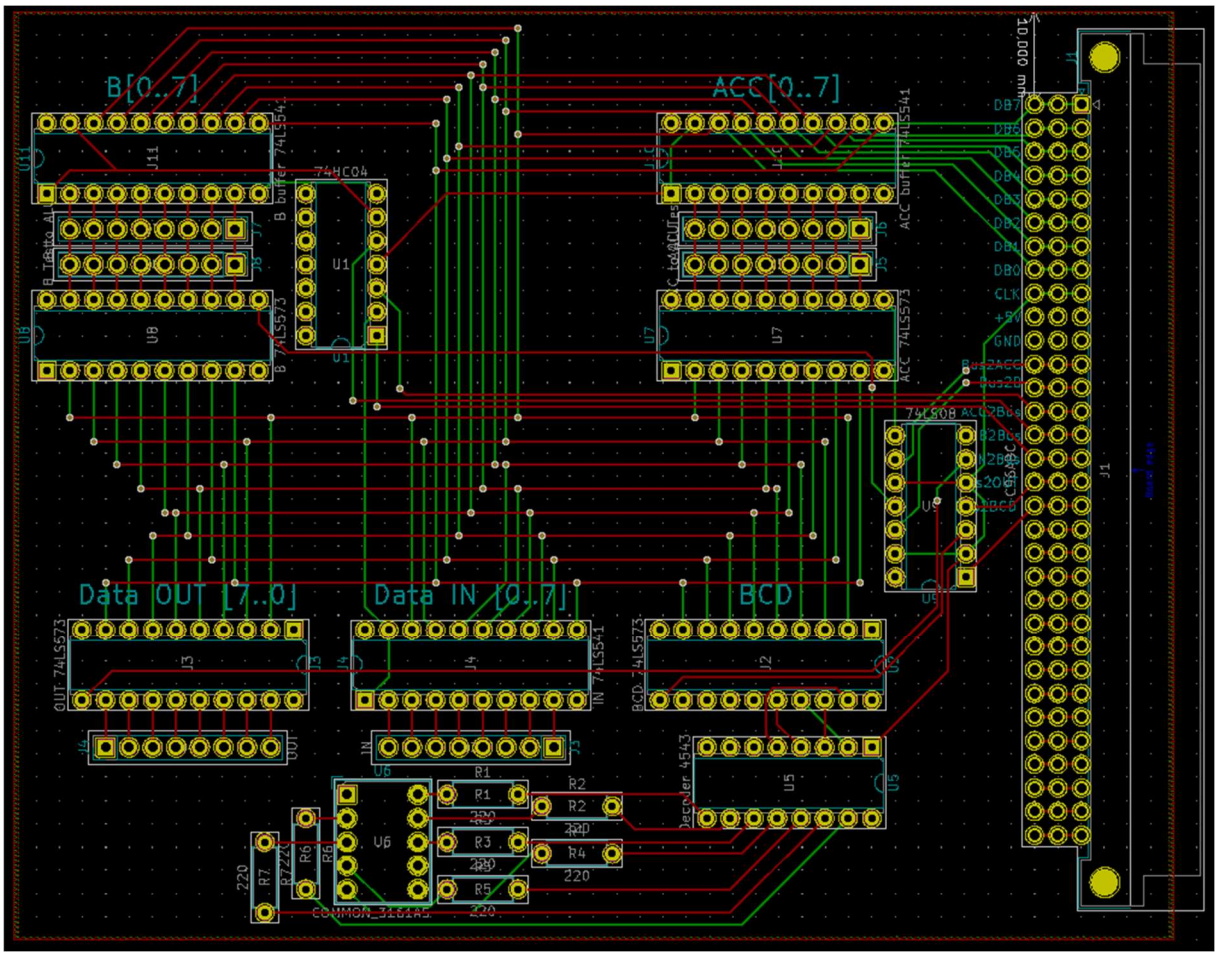
Vezérlő egység / dekóder



Memória


100,000 mm
J2
C964BC
U15
74LS00
U16
74LS573
MRL
U1
74HC02
U2
74HC04
U3
U5
PC[14..8]
Instruction LOW
OPL AT28C256-15PU
OPH AT28C256-15PU
Inst AT28C256-15PU
PC[7..0]
Instruction High
MRH
U17
74LS573

Regiszterek



Tesztelő modul