

Szegedi Tudományegyetem
Informatikai Intézet

**Az emulátor fejlesztési folyamat bemutatása a
Nintendo Game Boy konzolon**

Diplomamunka

Készítette:
Krizzák Tibor
informatika szakos
hallgató

Témavezető:
Dr. Tanács Attila
egyetemi adjunktus

Szeged
2018

Tartalomjegyzék

Feladatkiírás	4
Tartalmi összefoglaló	5
Bevezetés	6
1. Az emulátorok és a Nintendo Game Boy	7
1.1. Az emulátorokról	7
1.1.1. Az emulátor fogalma	7
1.1.2. Az emulátorok típusai	7
1.1.3. Az emulátorok jövője	8
1.2. Nintendo Game Boy	8
1.2.1. Története, jelentősége	9
1.2.2. Hardver specifikáció	9
1.2.3. Boot ROM	11
2. A fejlesztési folyamat	13
2.1. A fő ciklus	13
2.2. Alkalmazott eszközök	14
2.2.1. A Rust programozási nyelv	15
2.2.2. A minifb könyvtár	15
2.2.3. Fejlesztői környezet	16
2.2.4. <i>Debugger</i>	17
2.2.5. Memóriatérkép	18
2.3. A feladat specifikációja	19
3. A processzor és a memória implementációja	21
3.1. CPU	22
3.1.1. Regiszterkészlet	22
3.1.2. Ciklusok, frekvenciák	23
3.1.3. Betöltés - dekódolás - végrehajtás	24
3.2. Interrupt kezelés	25
3.3. Utasításkészlet	26
3.4. RAM	26
3.5. Időzítők	26
4. Függelék	27
4.1. A Nintendo Game Boy hivatalos architektúrája	27
4.2. A processzor opkód táblái	28

Nyilatkozat	30
Köszönnetnyilvánítás	31
Irodalomjegyzék	32

Feladatkiírás

A Nintendo Game Boy egy 1989-ban bemutatott, 8 bites kézi videojáték-konzol. A konzolban egy Zilog Z80 (az Intel 8080 utódja) processzor működik, kiegészítve néhány specifikus utasítással. A Game Boy emulátor fejlesztésének bemutatása során a CPU utasításait, a GPU renderelésének működését, a memóriakezelést, és a megszakítás-vezérlést kell implementálni. Ahhoz, hogy ezek megfelelően működjenek, a CPU frekenciájára, illetve a képfrissítési gyorsaságra is tekintettel kell lenni.

A játékkonzol emulátorok fejlesztése során a szűk, ezzel foglalkozó fejlesztői réteg kialakított egy egyértelmű, jól követhető fejlesztési folyamatot. A dolgozatban ezen keresztül kerüljenek bemutatásra a Nintendo Game Boy emulátor fejlesztési fázisai.

A tesztelés a más Zilog Z80 emulátor fejlesztők által készített teszt ROM-okon történjen.

Tartalmi összefoglaló

– A téma megnevezése:

Egy emulátor fejlesztési fázisainak bemutatása a Nintendo Game Boy hardveren keresztül, Rust nyelven implementálva.

– A megadott feladat megfogalmazása:

A feladat egy Nintendo Game Boy emulátor implementálása, és fejlesztési fázisainak bemutatása. A bemutatás során a CPU utasításait, a GPU renderelésének működését, a memóriakezelést, és a megszakítás-vezérlést kell érinteni, illetve az egyéb kisebb, de a működéshez elengedhetetlen megoldások is megemlítésre kerülnek. Ahhoz, hogy ezek megfelelően működjenek, a CPU frekvenciájára, illetve a képfrissítési gyorsaságra is tekintettel kell lenni.

– A megoldási mód:

Az emulátor fejlesztő közösség által összegyűjtött, – *reverse-engineered* – információkra, illetve a processzor gyártója által kiadott technikai dokumentációra hagyatkozva felépítettem és implementáltam a CPU struktúráját, utasításkészletét, majd a többi modult, részegységet. Meghatároztam a modulok közti kommunikációt, időzítéseket, adatfolyamatot. A videójáték-, illetve teszt ROM-ok byte-jait sorra beolvasva az emulátor meghatározza a megfelelő műveletet, meghatározott időközönként renderel, illetve kezeli a megszakításokat.

– Alkalmazott eszközök, módszerek:

Az emulátor Linux rendszeren, Rust nyelven került implementálásra, a `rustc` fordító, illetve a `cargo` package manager segítségével. A rendereléshez a `minifb` libraryt használtam, ami egy nagyon egyszerű framebuffer használatát teszi lehetővé. A fejlesztésre került egy debugger eszköz, illetve egy memóriatérkép eszköz is, ami nagyban megkönnyítette a hibakeresést.

– Elért eredmények:

Az implementált emulátor képes futtatni Memory Banking nélküli videójáték ROM-okat, az inputra az elvárásoknak megfelelően reagálva. A processzor műveletek és a renderelés az eredeti konzollal megegyező eredményt adnak. A közösségi Game Boy teszt ROM-ok szinte mindegyikét sikerrel végrehajtja.

– Kulcsszavak:

Nintendo, Game Boy, emulátor, fejlesztés, Rust

Bevezetés

A számítástechnikában az emuláció fogalma nem új keletű. Különböző területeken, különféle problémák megoldására használnak emulátorokat, ugyancsak különböző okokból. A nyomtatóktól kezdve, a DOS-kártyákon keresztül, a többmagos rendszertervezésen át egészen a videojáték konzolokig terjed a paletta - nem túlzás azt állítani, hogy az emulátorok ott vannak a minden napjainkban.

Ezen diplomamunka a videojáték konzolok emulátorainak fejlesztésére fókuszál. Többféle cél állhat a háttérben, ha valaki ilyen emulátor fejlesztésére adja a fejét: a régi hőskorbeli konzolok digitális megőrzése vagy életre keltése, későbbi szoftverfejlesztés az emulált hardveren, esetleg hobbiként. Az utóbbi évek tendenciája azt mutatja, hogy ez utóbbi ok egyre gyakoribb - az emulátor fejlesztői közössége napról napra nagyobb és aktívabb, szokások és kisebb fejlesztői folklór alakult ki az emulátor készítését illetően - a dolgozat ennek bemutatására helyezi a hangsúlyt.

Az emulátor fejlesztés szemléltetése Nintendo Game Boy kézi videojáték konzolon keresztül fog történni, amely a maga idejében egy igazán sikeres konzol volt, és tulajdonképpen kultusz épült köré. A 8 bites architektúrájából adódóan kevéssé bonyolult felépítéssel rendelkezik, népszerűségéből adódóan jól dokumentált, így az emulálásának implementációjához nincs szükség túl sok *reverse-engineering* gyakorlatra.

A dolgozat első néhány fejezetében az emulátorokról, a Nintendo Game Boy hardveréről, specifikációjáról, illetve a későbbi fejlesztés workflow-járól fog szó esni. Ezekben a fejezetekben van megfogalmazva, illetve leírva az, hogy pontosan mi az az emulátor, milyen hardver emulációjáról van szó, és hogy az emuláció teljes implementálásáig milyen ponton keresztül vezet az út. A következő nagyobb logikai egység az implementáció. Ennek részeként először bemutatásra kerülnek az alkalmazott eszközök, technológiák, majd az emulátor pontos és elvárt specifikációjának leírását az igazi implementációs szakasz követi.

A processzor modellezése a regiszterek, flagek, és egyéb jellemzők megtervezésével kezdődik, majd következő lépésként az utasításkészlet megvalósításával folytatódik. A CPU-hoz szorosan kapcsolódó memória ez után kerül tárgyalásra. A memória ismertetése után az időzítők, majd a PPU felépítése és működése szerepel. Az implementáció ezen pontján a Boot ROM már futtathatóvá válik, erről is esik majd néhány szó. A fejlesztési részt a joypad jellemzői és megoldásai zárják.

A dolgozat zárásként bemutatásra kerül az emulátor használata, illetve a fejlesztésből adódó dependenciák, majd végül a teszt ROM-ok jellemzői, futtatásuk, és a futtatási eredményeik.

1. fejezet

Az emulátorok és a Nintendo Game Boy

1.1. Az emulátorokról

Az utóbbi évtizedekben végbement – és jelenleg is tartó – technikai fejlődés következményeként rendkívül gyors a technológiai elavulás. Ennek következményeképp az eszközök életciklusa megrövidül, értékük rohamosan csökken. Gyakran előfordul azonban, hogy szükség van a régi *legacy* rendszerekre, vagy elengedhetetlen a visszafelé kompatibilitás, esetleg szeretnénk az adott hardvert a számítástechnikai jelentősége miatt valamilyen formában megőrizni, használhatóvá tenni. Az emulátorok ezekre a problémákra igyekszenek megoldást kínálni – persze rendkívül sok egyéb felhasználási terület mellett.

1.1.1. Az emulátor fogalma

Definíció szerint olyan hardvert vagy szoftvert nevezünk **emulátornak**, amely lehetővé teszi, hogy egy számítástechnikai rendszer (szokás ezt *host*-nak nevezni) úgy viselkedjen, mint egy másik számítástechnikai rendszer (ez pedig a *guest*). Jellemzően az emulátor a *host* rendszer számára teszi lehetővé olyan szoftver futtatását vagy periféria használatát, amely a *guest* rendszerhez lett kifejlesztve. Röviden megfogalmazva az emulátor egy olyan hardver vagy szoftver, ami egy másik eszközöt vagy programot emulál, imitál.

1.1.2. Az emulátorok típusai

Az emulátorok többsége csak a hardver architektúrát emulálja – ha operációs rendszer vagy egyéb szoftver is szükséges az emuláláshoz, akkor azt is biztosítani kell. Ebben az esetben az operációs rendszert és a szoftvert *interpretálni* (értelmezni) fogja az emulátor. A gépi kód *interpreteren* kívül azonban az emulátornak tartalmaznia kell a *guest* hardver minden lehetséges jellemzőjét, és viselkedését is virtuálisan: ha például egy adott memóriahezre való írás befolyásolja azt, hogy mi jelenik meg a képernyőn, úgy azt is emulálni kell. Habár lehetne az emulációt extrém részletességgel, atomi szinten végezni – például az áramkör adott részei által kibocsátott pontos feszültségingadozás emulálásával, stb. –, ez egyáltalán nem gyakori, az emulátorok általában megállnak a dokumentált hardver specifikáció, és digitális logika szimulációjának szintjén.

Némely hardver hatékony emulálásához extrém pontosság szükséges: az óraciklusokat, nem dokumentált jellemzőket, kiszámíthatatlan analóg elemeket, és *bugokat* mind-mind

implementálni kell. A klasszikus otthoni számítógépek esetében (például a Commodore 64) ez hatványozottan igaz, mert az ezekre a hardverekre írt szoftverek gyakran kihasználtak alacsony szintű programozási trükköket, melyeket főként a videójáték programozók és a *demoscene*¹ fedeztek fel.

Ezzel szemben léteznek azonban olyan platformok, amelyek alig használják a közvetlen hardver elérést, jó példa erre a PlayStation Vita. Ezekben az esetekben elég egy kompatibilitási réteget megvalósítani, amely a *guest* rendszer rendszerhívásait fordítja le a *host* rendszer hívásaira.

1.1.3. Az emulátorok jövője

A videójáték-konzol emulátorok világa, illetve az emulátor fejlesztő közösség helyzete igen érdekes. Az egyik oldalról megvizsgálva azt tapasztalhatjuk, hogy egyre nagyobb népszerűségnek örvendő területről van szó. Ami a másik oldalt illeti – a helyzet nagyon homályos. Újabb és újabb konzolok jelenniekn meg, egyre rövidebb életciklussal és egyre bonyolultabb architektúrával. Jól mutatja ezt a PlayStation 3 példája: 12 éve, 2006-ban jelent meg, és tökéletes emulátor még nem készült hozzá. A közösség nem tudja tartani a tempót a bonyolultság, és a rövid életciklusokból adódó szoros határidők miatt.

Sokak szerint viszont a jövőben nemhogy nehezebb, hanem inkább könnyebb lesz az emuláció: véleményük szerint a hardver emulációja nem lesz könnyű, viszont az utóbbi években nagyon sokat javult a szoftverek minősége és tisztasága egyaránt. A játékfejlesztők rá vannak kényszerítve az API-k (*Application Programming Interface* – alkalmasprogramozási interfész) használatára a hardver *bugjainak* kihasználása és a trükközés helyett, és ez lehetőséget adhat az API-kon alapuló emuláció elterjedése felé.

Fontos megemlíteni egy 2010-ben induló közösségi projektet, a RetroArch-ot, amely a videójáték konzol emulátorok számára biztosít egy prezentációs réteget, ún. *frontend*-et, amely egybefogja, és használhatóvá, futtathatóvá teszi a vele kompatibilis emulátorokat. Ez a megoldás nagyban megkönyíti a felhasználók életét, hiszen több tucatnyi rendszer emulátorát érhetik el egyetlen felületen keresztül, és a fejlesztők számára is jelent egy enyhe szabványosítási törekést.

Ahogy a fenti két vélemény, és a RetroArch példája is mutatja, sokan sokféleképpen vélekednek az emulátorfejlesztés jövőjéről, nem beszélve a frissen induló közösségi projektekről – szinte biztosan kijelenthető, hogy ez a terület nem fog egyhamar megszűnni.

1.2. Nintendo Game Boy

Egy emulátor fejlesztési folyamatának bemutatására a Nintendo Game Boy tökéletes példa több szempontból is. Elsősorban széleskörűen ismert, ebből adódóan az emulátor fejlesztői közösség által is jól dokumentált, a hardver szinte az utolsó részletig vissza lett fejtve. Ezekből a dokumentációk tehát jó kiindulási alapot nyújtanak. Az is fontos szempont, hogy a hardver a 8 bites érából származik, ami szinte garantálja az egyszerűbb architektúrát (ez persze relatív), így a könnyebb implementálhatóságot. Szintén megemlítendő, hogy a fejlesztői közösség által készített teszt ROM-ök nagyban segítik a

¹ A *demoscene* nemzetközi underground számítástechnikai szubkultúra, amelynek célja különböző számítógépes digitális művészeti alkotások (*demók*) készítése.

hibakeresést, verifikációt.

1.2.1. Története, jelentősége



1.1. ábra. A *Nintendo Game Boy* logója

A Game Boy egy Nintendo által gyártott hordozható videojáték konzol, amit a nagyközönség számára 1989-ben mutattak be. Ez volt a gyártó első 8 bites kézi konzolja, amihez a játékokat cserélhető kazetta formájában (angolul *cartridge*) lehetett megvásárolni.

Az okos marketingnek, és a jó Nintendo *brand*-nek köszönhetően a Game Boy kora legsikeresebb kézi konzolja lett, annak ellenére, hogy a versenytársaihoz (Atari Lynx, Sega Game Gear) mérten elavult technológiát használt. Ez egyben azt is jelentette, hogy a Game Boy-ban használt alkatrészek olcsóbbak, ismertebbek és kiforrottabbak voltak, mint a riválisoké. A tervezők alapgondolata az volt, hogy régebbi technológiát használnak fel innovatív módon. A konzol sikerét az olcsósága, az akkumulátor időtartama, és a platformon elérhető rengeteg játék mennyisége és minősége koronázta meg. Az 1997-ig értékesített 60 millió példányszám a Game Boy-t a gyártó egyik legsikeresebb termékévé tette. A készülék jellegzetes logója a 1.1-es ábrán látható.

1.2.2. Hardver specifikáció

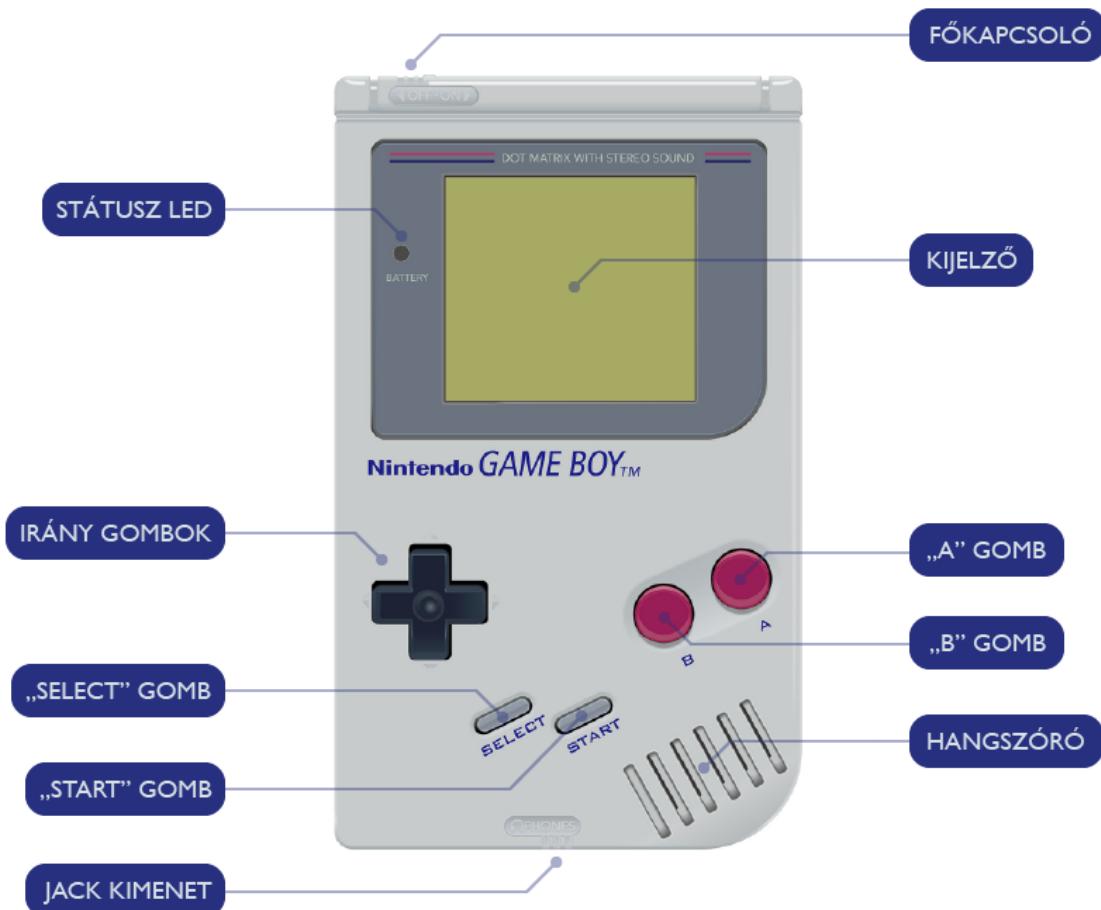
A hardver specifikációját két logikai egységre lehet osztani: a Game Boy hardverére és a *cartridge* hardverre. Ugyan ezek együtt alkotnak egészet, hiszen egyik sem használható a másik nélkül, ám technikailag két különálló egységről beszélhetünk.

Game Boy

A konzol külseje, és kezelőszervei a 1.2-es ábrán figyelhetők meg, az általa tartalmazott hardver elemek pedig a következők[2]:

- **CPU:** a 8 bites Zilog Z80-as CISC-processzor architektúrán alapuló – annak utasításkészletén enyhén módosított változata – Sharp LR35902.
- **RAM:** 8 kB beépített S-RAM
- **VRAM** (video memória): 8 kB beépített
- **ROM:** 256 Byte (Boot ROM-nak fenntartva)

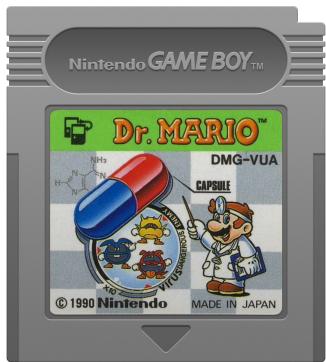
- **Hang**: 2 négyszögjel generátor, 1 programozható 32 mintás 4 bites PCM hullám, 1 fehér zaj, és egy audio bemenet a kazettából. (A külső kazetta bemenetet soha egy piacra dobott játék sem használta.) A jack kimeneten keresztül sztereó hangot ad.
- **Kijelző**: 166×144 pixel felbontású LCD kijelző, mérete átlósan 66 mm.
- **Színpaletta**: 4 árnyalat 2 biten tárolva – világos zöldtől a sötét zöldig.
- **Tápellátás**: 4 db AA elem, amely megközelítőleg 14-35 óra játékidőt biztosít.



1.2. ábra. A Game Boy és részei

Cartridge

A konzolhoz tartozó játék kazettákat a konzol hátuljába kellett címkkével kifelé fordítva becsúsztatni. Ezek a kazetták jól felismerhetők voltak a jellegzetes (nagyrészt) szürke színükről, illetve az elejükre ragasztott, az adott játékot ábrázoló címkéjükön, ami a 1.3-es ábrán is megfigyelhető. A Game Boy-hoz több típusú kazetta volt forgalomban, melyet az indokolt, hogy némely játék nagyobb erőforrást igényelt a futásához. A konzolnak köztudottan kicsi volt a memória mérete, így a játékfejlesztőknek különféle trükköket kellett bevetniük ahhoz, hogy a játékaikat futásra bírják. Erre a problémára a *Memory Bank Controller* alkalmazása volt a megoldás, ennek használatával a fejlesztők számára nagyobb ROM, illetve *MBC* verziótól függően nagyobb RAM volt elérhető. Az *MBC* részleteiről és típusairól az egyik későbbi fejezetben lesz szó.



1.3. ábra. A *Dr. Mario* játék kazettája

A kazetták többségében volt egy CR2025-ös típusú gombelem is, ami az elmentett játékállások tárolásából adódó erőforrás-ellátásért felelt. Az elem viszont nem tartott örökké – így mikor hosszú idő után ugyan, de lemerült, az összes mentett állás elveszett.

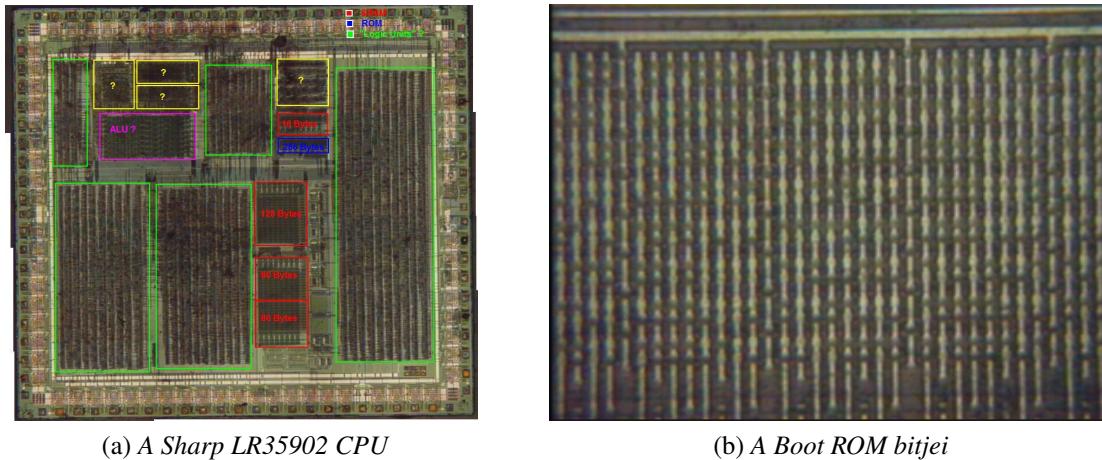
További érdekesség még a *cartridge*-ekkel kapcsolatban a Nintendo által kifejlesztett korabeli (de meglepően hatékony) másolásvédelmi és *homebrew*²-fejlesztőket kizáró mechanizmus. Annak érdekében hogy a kazettákat ne lehessen lemásolni és így terjeszteni, illetve a gyártóval kapcsolatba nem lépő hobbi fejlesztők ne tudjanak játékot kiadni a platformra, a Nintendo szokatlan, de hatékony megoldást választott. Amikor a felhasználó behelyezi a kazettát a Game Boy-ba, és bekapcsolja azt, akkor a Boot

ROM lefutását követően a játék csak akkor indul el, ha a játék kódjában megtalálhatók a Nintendo logót alkotó byte-ok. Ha ez hiányzik, akkor a játék nem fog futni. A trükk az, hogy ugyan a törvény nem tiltja, hogy játékokat fejlesszenek a hobbi fejlesztők a konzolra, viszont a Nintendo logó felhasználását szerzői jogi törvények védik. Így, ha a hobbi fejlesztő terjesztené a játékát, akkor be kell ágyaznia a Nintendo logót a kódba, amivel viszont szerzői jogi szabályokat sért. Ezzel a Nintendo elérte, hogy a konzolra kiadott játékok minősége magas legyen, hiszen minden játék kiadásáról végső soron ők döntötték.

1.2.3. Boot ROM

A Nintendo Game Boy Boot ROM-ja végzi a hardver elindulását követő inicializáló folyamatokat, illetve az előző részben említett másolásvédelmi eljárást. A ROM pontos tartalma egészen 2003-ig ismeretlen volt – ekkor viszont egy *Neviksti* nevű felhasználó publikálta a *cherryroms.com* fórumára a program teljes változatát. *Neviksti* azt is leírta, hogy hogyan sikerült visszafejtenie a kódöt: a Game Boy processzor chip tetejének leszedése után mikroszkóppal megvizsgálta az áramkört, majd amint megtalálta a Boot ROM

² A *homebrew* fogalmat azokra a videojátékokra vagy egyéb szoftverekre használjuk, amelyeket a fogyasztói réteg készít el a zárt forrású hardverekre, platformokra – tehát nincsenek kapcsolatban a célhardver gyártójával.



(a) A Sharp LR35902 CPU

(b) A Boot ROM bitjei

1.4. ábra. A CPU és a Boot ROM

lehetséges helyét (256 kB ROM van elkülönítve erre a CPU-ban), lefényképezte azt, majd bitről bitre haladva rekonstruálta a bináris állományt. A teljes processzor madártávlati nézete a 1.4-es ábrának (a) részén látható, a belenagyított (b) ábrán pedig az áramkör Boot ROM-ot tároló része szerepel, amin szemmel is jól láthatók a programot alkotó bitek.

2. fejezet

A fejlesztési folyamat

Ahogy már az előző fejezetekben is említésre került, az emulátor fejlesztői szubkultúrában többé-kevésbé kialakult egyfajta irányelv, amit érdemes követni az emulátor fejlesztésénél. Természetesen olyan leírást nem lehet készíteni ami bármilyen konzol emulátorának fejlesztésére használható – a hardverek különbözősége és a speciális megoldások nem teszik ezt lehetővé. Azt viszont meg lehet tenni, hogy egy általános tervezési mintát meghatározunk, és a tervezésnél - implementálásnál ezt követjük.

Az első teendő mindenkorban a lehető legtöbb tudásanyag összeszedése ilyen-olyan forrásokból: internetről, régi szaklapokból, esetleg magát a hardvert tanulmányozva. Nagy segítséget jelenthet például ha már valaki belekezdett ugyanazon hardver emulátorának fejlesztésébe, hiszen fontos információkkal szolgálhat. Egyes hardver emulátorok köré közösségek is összegyűlnek: így van ez a Game Boy esetén is. Ez a közösség egy honlapon gyűjtötte egybe az elérhető összes – eddig fellelt – információt a konzolról. A legfontosabb dokumentum azonban minden emulátor fejlesztése kapcsán a processzor dokumentációja, hiszen – ahogy majd látni a későbbiekben erre ki is térek – ezt fogjuk először implementálni. Mielőtt az implementációs szakaszba lépnénk, célszerű átgondolni az emulátor leendő struktúráját, működését, illetve az alkalmazott eszközöket. Továbbá az elvárt működést, az *inputot* és az *outputot* is át kell gondolni a tényleges fejlesztési munkálatok előtt.

TODO TODO TODO TODO TODO TODO TODO TODO TODO

A fenti ... ábrán látható módon fog alakulni az emulátor felépítése, architektúrája. Ahogy az jól megfigyelhető, a CPU áll a középpontban, ez tartalmazza a fő ciklust is. A többi modul ehhez csatlakozva, de külön álló egységek ként képzelhető el. Ennek megfelelően a fejlesztést a processzor megvalósításával kell kezdeni, majd a különálló modulok implementálásával folytatni. Ezen részegységek fejlesztésének időrendi sorrendje többnyire szabadon megválasztható, viszont célszerű a CPU - MMU - IRQ - PPU sorrendet követni. A 2.1-es alfejezetben lesz szó az emulátor "magjáról", a fő ciklusról, amely a processzor (és így a többi modul) alapját képezi.

2.1. A fő ciklus

A fő ciklus a Game Boy utasítás-végrehajtását emulálja, aminek egy leegyszerűsített modellje bármilyen Neumann-elvű számítógép processzorára illeszkedni fog. Ezt a fő ciklust elterjedtebb nevén **betöltő-dekódoló-végrehajtó** ciklusnak is nevezik. Lépései a

következők[1]:

1. A soron következő utasítás betöltése a memóriából az utasításregiszterbe.
2. Az utasításszámláló (másnéven *Program Counter*, vagy PC) beállítása a következő utasítás címére.
3. A beolvasott utasítás típusának meghatározása.
4. Ha az utasítás memóriabeli szót használ, a szó helyének meghatározása.
5. Ha szükséges, a szó beolvasása a CPU egy regiszterébe.
6. Az utasítás végrehajtása.
7. Vissza az 1. pontra.

A fenti szerkezet valamilyen módon minden emulátorban megtalálható, ez a felépítés alapja. A ciklus addig ismétlődik, amíg egy HALT, vagy egyéb kilépést/megállást szolgáló utasítás nem érkezik végrehajtásra. Természetesen a megszakításkezelő valamelyest bele-szól a ciklus működésébe, de erről majd egy későbbi fejezetben lesz szó.

A Game Boy emulátorban a fenti szerkezet egy egyszerűbb változata működik, ami váz-latszerűen így néz ki:

```
loop { // endless loop
    let next_byte = fetch_byte();
    let instruction = decode_instruction(next_byte);
    execute(instruction);
}
```

A fenti függvényeket, és azok működését a későbbiekben fogom részletezni.

A fő ciklus megtervezése tipikusan a CPU alap struktúrájának (regiszterek, RAM, stb.) implementálása után következik. Ezek után jöhet csak a legtöbb emulátor leghosszabb és legrepetitívebb része: a CPU műveleteinek implementálása.

2.2. Alkalmazott eszközök

A fejlesztéshez alkalmazott eszközök meghatározása fontos tényező, hiszen nagyban meg-könnyíthetjük vagy megnehezíthetjük a saját munkánkat. Először is célszerű egy programozási nyelvet választani, lehetőség szerint olyat, amihez léteznek olyan *library*-k, amelyekkel megvalósítható a program. Emellett az is lényeges, hogy a programozási nyelv gyors binárist generáljon – természetesen megvalósíthatjuk az emuláltot *Javascript* nyelven is, csak észrevehetően lassabb lesz, mint mondjuk a C++-os variánsa.

A programozási nyelv mellett a *debug*-olást nagyban megkönnyíti egy *disassemblers*, vagy optimális esetben egy másik emulátorhoz készített *debugger*. A ROM fájlokhoz szükséges lehet még egy *hex editor*¹, hogy pontosan lássuk azt, hogy milyen bájtokkal dolgozunk.

Ahhoz hogy lássuk, hogy a memóriában milyen adatok szerepelnek, célszerű egy memóriatérkép eszközt készíteni a fejlesztés során.

¹ A *hex editor* egy olyan szoftver, amely segítségével megtekinteni és módosítani lehet egy bináris adatfájlt. A "hex" előtag a hexadecimális rövidítésből ered: a bináris fájl bájtjait 16-os számrendszerben mutatja a program.

2.2.1. A Rust programozási nyelv

Az emulátor fejlesztéséhez a Rust programozási nyelvet választottam, több okból. Egyrészt ez előtt egy kisebb emulátor projekten dolgoztam a nyelvvel, és már akkor megtetszett az egyszerűsége, a környezete, a nyelv köré alakult közösségi. Másrészt a nyelvet az ehhez hasonló performancia-orientált feladatokra terveztek.

A **Rust** a fejlesztők weboldala szerint egy 2006 óta fejlesztett, rendszerfejlesztésre készített nyelv, amely villám gyorsan dolgozik, megelőzi a szegmentációs hibákat, és garantáltan gátolja a versenyhelyzetek kialakulását. Erősen típusos nyelv, szintaktikailag a C++-hoz hasonlít, viszont hozzá képest biztonságosabb memóriakezelést biztosít a sebesség megtartásával. A Rust világában tehát nincsen null pointer, lógó pointer, és versenyhelyzet sem. A fejlesztését és tervezését a Mozilla kutatói részlege kezdte el, majd idővel közösségi projektté alakult. Jelenleg 1.24.1-es jelzésű az aktuális verzió.

Fontos még megemlíteni, hogy a *Stack Overflow* weboldalon megrendezett éves fejlesztői kérdőív kitöltések alapján 2016-ban, 2017-ben és 2018-ban is a Rust nyerte a "leginkább kedvelt programozási nyelv" kategóriát. Egyéb érdekesség, hogy jól megfigyelhető, hogy az emulátor fejlesztő közösség túlnyomó többsége vagy C++-ban, vagy Rust-ban fejleszt – ez a nyelv kényelmességének, eleganciájának és sokoldalúságának is köszönhető.

Maga a nyelv szépsége azonban még minden – a nyelv mellett a **Cargo** eszköz egy fontos szempont. A Cargo nyilván tartja és rezolválja a Rust projektekben összeszedett függőségeket, illetve *buildeli* a projektet. Két *metadata* fájlban tárolja a projekttel kapcsolatos információkat, melyek alapján beszerzi és buildeli a projekt függőségeit. Ezt követően meghívja és futtatja a `rustc` fordítót a megfelelő paraméterekkel. A Cargo a külső *libraryket*, illetve függőségeket a *crates.io* közösségi központi repozitóriumból szerzi be.

2.2.2. A **minifb** könyvtár

Mivel grafikus programról beszélünk, ezért az ablakkezelés és az emulátor vizuális *outputja* fontos tényező. Ehhez – ha lehetséges – minél egyszerűbb és gyorsabb külső könyvtárat kell használnunk, ha szeretnénk megkönnyíteni és felgyorsítani a munkafolyamatunkat. A **minifb** *crate* ezt teszi lehetővé, hiszen ez egy platformfüggetlen, Rust-ban írt *library*, amivel az operációs rendszer által kínált natív ablakokat lehet megnyitni, és feltölteni egy 32 bites *bufferrel*. Támogatja a billentyűzet és egér eseménykezelést, és nemely operációs rendszer esetén (Windows, macOS) a menürendszeret.

A használata nagyon egyszerű:

```
window : Window::new("RUST BOY",           // name
                     160,                  // width
                     144,                  // height
                     WindowOptions {        // other options
                         resize: false,
                         scale: Scale::X2,
                         ..WindowOptions::default()
                     }).unwrap()
```

Amint látható, négy kötelezően megadandó paramétere van a `Window` struktúra konstruktörának, melyek rendre:

- `name`: az ablak címsorában szereplő szöveg,
- `width`: az ablak szélessége pixelben,
- `height`: az ablak magassága pixelben,
- `WindowOptions`: az egyéb ablakbeállításokat tartalmazó struktúra.

A negyedik paraméternél kiválaszthatjuk, hogy a `default` ablakbeállításokat szeretnénk-e – amennyiben igen, `WindowOptions :: default ()`-ot kell megadni. Ha saját beállításokat kívánunk megadni ebben a `WindowOptions` struktúrában, rendre ezek közül választhatunk:

- `borderless`: ezzel megadható, hogy az ablaknak legyen-e kerete vagy sem,
- `title`: ezzel megadható, hogy az ablaknak legyen-e címe vagy sem
- `resize`: ezzel megadható, hogy az ablak átméretezhető legyen-e vagy sem
- `scale`: ezzel a struktúrával megadható, hogy az ablak mekkora nagyítással jelenjen meg, választható opciók: `X1, X2, X4, X8, X16, X32`.

A konstruktur meghívását követően az ablak tartalmát (és a `framebuffert`) a következőképpen frissíthetjük:

```
window.update_with_buffer(&framebuffer).unwrap();
```

ahol a `&framebuffer` egy `&[u32]` típusú, `u32` számokat tároló, `width * height` méretű tömbre mutató referencia. A tömbben lévő számok tárolják el az adott pixel színét az ablakban: hexadecimálisan megadva az első két karaktert figyelmen kívül hagyjuk, majd az utána következő 6 karakter adja a szín hexadecimális megfelelőjét:

FF FF FF FF

A fentiek alapján látszik, hogy a második `FF` tag a piros (R), a harmadik `FF` tag a zöld (B), a negyedik `FF` tag pedig a kék (B) színért felel. Külön-külön tehát az RGB kódokat, még együtt a hexadecimális színkódot kapjuk.

2.2.3. Fejlesztői környezet

A fejlesztést *elementary OS*² rendszeren végeztem. Az emulátor fejlesztés sajátosságai miatt feleslegesnek éreztem egy IDE³ használatát, hiszen ha a programkód szyntaxa megfelelő, onnantól kezdve a hibakeresést az IDE-k által kínált eszközök sem tudják megkönyíteni, ahhoz saját *debugger* kell írni. Ilyen fejlesztői környezet használata helyett

² Az *elementary OS* egy *Ubuntu* alapú Linux disztribúció.

³ Az integrált fejlesztői környezet (angolul IDE, azaz Integrated Development Environment) a neve a számítógép-programozást jelentősen megkönyítő, részben automatizáló programoknak.

tehát a klasszikusnak mondható szövegszerkesztő (Atom, Rust *linterrel*⁴) és terminál párost használtam, a `rustc` fordító *warningjaira* és *errorjaira* hagyatkozva.

A `rustc` fordító *targetjeként* a `stable-x86_64-unknown-linux-gnu` beállítást használtam (alapbeállítás), ami a "hagyományos" 64 bites Linux disztribúcióra optimalizált fordítási paraméterezés. A fordítást, futtatást és a külső függőségek (*libraryk*) beszerzését a Cargo eszközzel valósítottam meg.

2.2.4. Debugger

0x34 : INC (HL)	A 0x00	B 0x00
0x21 : LD HL,nn 0xFF 0xE6	C 0x00	D 0x00
0x34 : INC (HL)	E 0x10	F 0x00
0x21 : LD HL,nn 0xFF 0xE5	H 0xFF	L 0xE6
0x34 : INC (HL)	SP 0xFFFF	
0x21 : LD HL,nn 0xFF 0xE2	PC 0x354	
0x20 : JR NZ	FLAG 0b0000	
0x05 : DEC B		
0x2C : INC L		
0x28 : JR Z, 0x1		
0xA7 : AND A,A		
0x7E : LD A, (HL)		
0x20 : JR NZ, - 0x9		
0x05 : DEC B		
0x2C : INC L		
0x28 : JR Z, 0x1		
0xA7 : AND A,A		
0x7E : LD A, (HL)		
0x06 : LD B, 0x2		
0x21 : LD HL,nn 0xFF 0xA6		
0x20 : JR NZ, 0x1B		
0xFE : CP A, 0xF		
0xEF : SBC A, 0xE		

2.1. ábra. Az emulátorhoz fejlesztett debugger

Fontos eszköz volt a fejlesztés során a *debugger*, amelyet az emulátorral párhuzamosan fejlesztettem. Nagyon hasznos, hogy pontosan végig lehet követni az emulátor működését, és az egyes processzorműveletek után beállt állapotokat, hiszen ez nagyban megkönnyíti a hibakeresést. A 2.1-es ábrán láthatjuk az eszköz működés közben: bal oldalon találhatóak a már elvégzett műveletek, a jobb oldal pedig a regiszterek állapotát mutatja.

Az elvégzett műveletek listájában legfelül a legutóbb végrehajtott művelet szerepel, a végén pedig a legrégebbi. A program az utolsó 50 állapotot tudja eltárolni, melyek közül az éppen kijelölt, aktív elemet piros kiemelés jelzi. Az egyes listaelemek az alábbi módon épülnek fel:

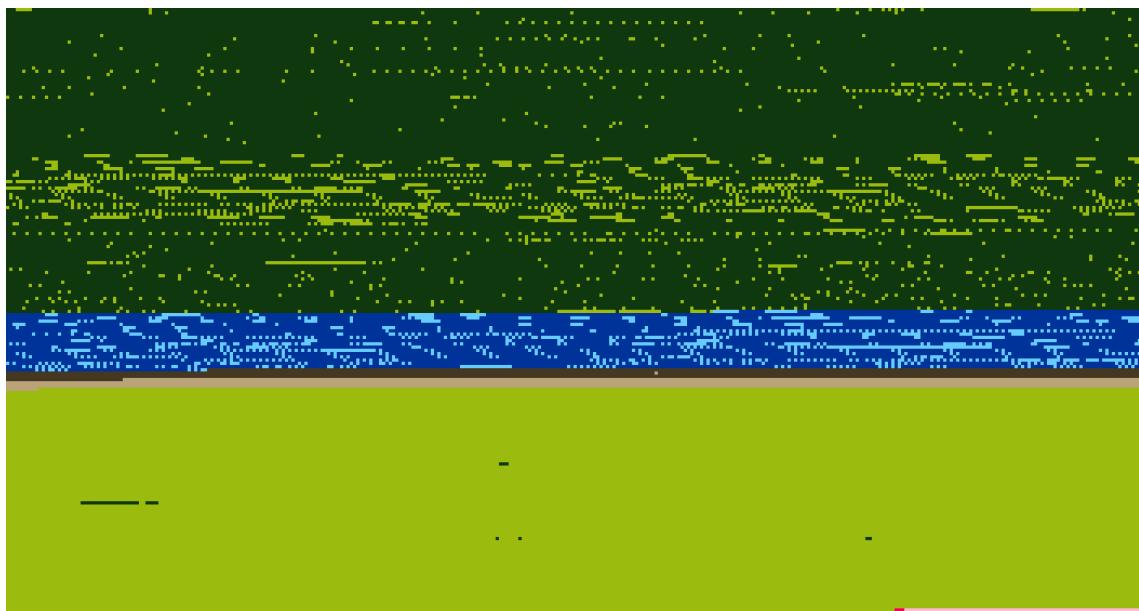
0x21 : LD HL,nn 0xFF 0xE6

⁴Olyan eszközöket nevezünk *linternek*, amelyek a forráskódot analizálva programozási hibákat, *bugokat*, stílusbeli hibákat, vagy gyanús felépítéseket jeleznek a felhasználónak.

A `0x21` jelzi az aktuális művelet *opkódját*⁵, mellette szerepel a művelet *mnemonikja*⁶, jelen esetben az `LD HL, nn`. A harmadik tag a művelet által beolvasott, és (operandus-ként) felhasznált bájtokat tartalmazza, itt: `0xFF 0xE6`. A példában (és a *debuggerben* is) be van színezve az utasítás – ennek egyszerű oka van: az utasításokat kategóriákra bontottam, majd külön színeket rendeltem hozzájuk, így már ránézésre is meg lehet mondani, hogy milyen típusú műveletről van szó. A mellékletként csatolt opkód táblázatban lévő színek megegyeznek a *debuggerben* látható színekkel.

A *debugger* jobb oldalában foglal helyet a regiszterek nézete, itt található meg rendre az összes regiszter, a *Stack Pointer*, és a *Program Counter* értéke, valamint az *F Flag* regiszter értéke binárisan – hogy látható legyen az összes általa tartalmazott flag állapota. Ezen értékek annak függvényében változnak (és mutatják az aktuális értékeket), hogy épp melyik művelet van kijelölve.

2.2.5. Memóriatérkép



2.2. ábra. Az emulátorhoz fejlesztett memóriatérkép eszköz

A *debugger* mellett a másik sokat használt eszköz a memóriatérkép. Ebben az ablakban megjelenik a Game Boy memóriájának összes bájtja, egy-egy pixel által reprezentálva. Az adott pixel világos, ha a bájt nulla, egyébként pedig sötétebb árnyalatú. A 2.2-es ábrán megfigyelhető, hogy több féle szín is megjelenik – ezek jelölik az egyes fontosabb, elkülönölt részeket a memóriában. A színeket is bevonva a reprezentációba a *debuggerhez* hasonlóan ennél az eszköznél is ránézésre leolvashatók adatok. Ahhoz, hogy pontosan megtudjuk egy adott bájt értékét és pozícióját a memóriában, rá kell kattintani, és a

⁵ Operációkód, azaz műveleti kód, vagy műveleti jelkód, utasításkészletek leírásában műveleti jelrész.
A CPU által beolvasott bináris szám, amit végrehajtható utasítás kódjaként értelmez.

⁶ A mnemonik az informatikában általában hosszabb elnevezésű művelet(sor) elnevezésére használatos rövidítés, amelyet az egész kifejezés helyettesítésére alkalmaznak, pl.: ADD, SUB.

terminál ablakban kiírásra kerülnek a szükséges információk. A kiírt adatok a következő formában jelennek meg a terminál ablakban:

```
BYTE : 0x46 0b01000110 – POSITION : 0x7984
```

Értelemszerűen a BYTE után szereplő két szám az egérrel kijelölt bájt értékét mutatja, míg a POSITION után szerepel a bájt helye a memóriában.

A Game Boy architektúrájában gyakori, hogy egyes regiszterek a memóriában kapnak helyet – erről a későbbiekben szó lesz –, és a memóriatérkép megoldással könnyedén meg lehet figyelni ezek értékeit, esetleg változásait. Emellett a dedikált és külön színnel kiemelt memória részeiken látszik, hogy fel van-e töltve, vagy teljesen üres – egy *sprite* renderelési *bug* kijavítását nagy mértékben megkönnyítette az, hogy látszott a memóriatérképen a *spriteok* hiánya.

2.3. A feladat specifikációja

Az emulátornak a feladatkiírásban meghatározott feltételeket kell teljesítenie, azaz:

- a CPU utasításokat és működését,
- a PPU renderelésének működését,
- a memóriakezelést,
- a megszakításvezérlést.

Ahhoz, hogy ezeket a feltételeket teljesíteni tudja, szükséges az *input* és *output* adatok (működés) pontos meghatározása.

Az input elvárt formai és tartalmi követelményei:

Az emulátor *inputjaként* a Game Boy DMG⁷ videójáték konzolhoz írt videójátékok ROM-jait adhatjuk meg, illetve a visszafejtett Boot ROM-ot. Videójátékok esetén az emulátor csak a MBC (*Memory Bank Controller*) nélküli ROM-okat képes futtatni. Előfordulhatnak olyan nem ismert, videójáték programozók által kihasznált *bugok*, amelyek gátolják a ROM tökéletes futtatását. Szükséges hogy a ROM tartalmazza a *headerjében* a Nintendo logó bájtjait (a Boot ROM-ban lévő *checksum* kiszámlolja ezt), mert ellenkező esetben a játék nem fog elindulni.

Inputnak tekinthetők még az emulált *joypaden* történő gombnyomások is, melyek hatással vannak az emulált szoftver működésére. A felhasználó egyszerre több gombot is lenyomhat – ennek emulációja megfelel az eredeti hardverével.

Az output elvárt formai és tartalmi követelményei:

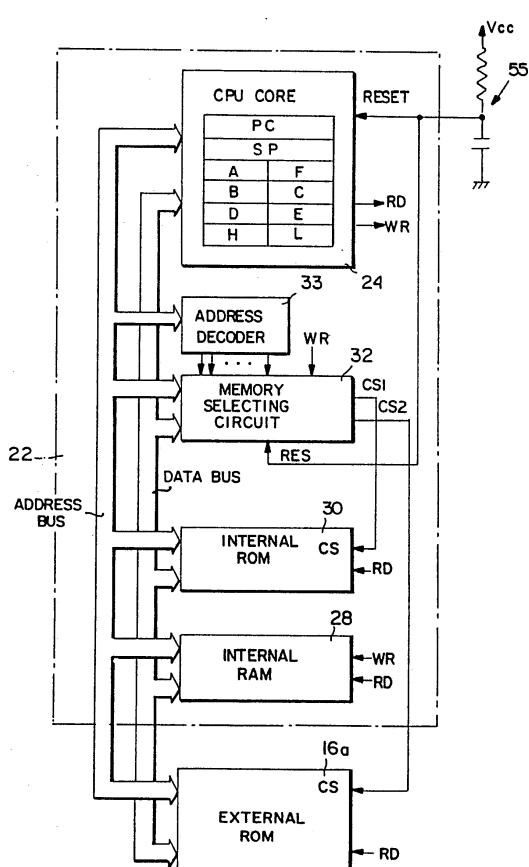
Az emulátor több féle *outputot* is előállít. A legfontosabb az emulált kijelzőre renderelt kép, amelynek meg kell egyeznie az eredeti konzol által kirajzolt képpel. A korhűség érdekében célszerű a 4 féle árnyalatot a konzol folyadékkristályos kijelzőjének jellegzetes zöld színeivel megegyező színekkel megjeleníteni.

⁷ Az eredeti, klasszikus 1989-ben kiadott Game Boy kódneve DMG.

Egyéb *outputnak* tekinthetjük a *debugger*, és a memóriatérkép által adott információkat is, hiszen az emulátor aktuális állapotáról a dnak visszajelzéseket. Természetesen ezek elhanyagolhatóak, de a hibakeresést – és így a fejlesztést – segítik.

3. fejezet

A processzor és a memória implementációja



3.1. ábra. A Game Boy szabadalmában ábrázolt processzor felépítés

függelék 4.1-es ábráját, amely szintén a Nintendo által benyújtott szabadalom része. Ez az ábra később nagy segítségünkre lesz a modulok felépítésének, és a rendszer struktúrájának implementálása során, hiszen elég részletesen mutatja be az egyes elemek közti kapcsolatokat, kommunikációt.

A korábban írtaknak megfelelően az implementációt a processzorral és a memóriával kezdjük – a megfelelő tudásanyag birtokában a processzor felépítésének modellezése az első feladat. Fontos már a korai tervezési folyamatotól kezdve szem előtt tartani a belső felépítés modelljét, és ennek tudatában úgy alakítani az implementálást, hogy egyrészt az eredeti hardverrel nagyrészt megegyező módon működjön (és épüljön fel), másrészt a további modulok, egységek és funkciók jól illeszkedjenek ehhez a fő komponenshez. Ehhez nyújt nagy segítséget a Nintendo által bejegyzett Game Boy szabadalmban ábrázolt felépítés, amit a 3.1-es ábrán figyelhetünk meg. Jól látszik a processzor és a memória kapcsolata, illetve ez előbbi felépítésébe is nyújt némi betekintést, utóbbinak egyes részeit pedig már ezen az ábrán is láthatjuk, a későbbiekben ezt jobban meg is vizsgáljuk. Az előző fejezetben ismertetett betöltő-dekódoló-végrehajtó ciklust alkotó elemek közül is megfigyelhetjük az utolsó kettőt. A hardver teljes felépítéséről tágabb képet kaphatunk, ha megfigyeljük a 4. fejezetbeli

3.1. CPU

3.1.1. Regiszterkészlet

Ahogy az első fejezetben már említésre került, a Game Boy-ban egy *Zilog Z80* alapú, *Sharp LR35902* típusú 8 bites processzor dolgozik. A Z80-hoz képest annyi változás történt, hogy a regiszterek elrendezését a *Sharp* mérnökei az *Intel 8080*-as processzortól vették kölcsön, illetve több utasítást is kivettek, helyükre sajátokat téve.

A 3.1-es ábrán látható, hogy a processzor nyolc darab 8 bites, és két darab 16 bites regiszterrel rendelkezik. A 8 bitesek név szerint: A, F, B, C, D, E, H, L, míg a 16 bitesek a PC és az SP. A 8 bites regiszterek felsorolásának sorrendje nem véletlen: a CPU egyik igen hasznos funkciója az, hogy regisztereket tud összevonni, két 8 bitesből 16 biteseket varázsolva – így ezekben a nagyobb regiszterekben el lehet tárolni memóriacímeket, vagy egyéb 16 bites adatokat. Az összevont regiszterek rendre: AF, BC, DE, HL.

Az A regiszter hagyományosan az akkumulátorregiszter¹, míg az F tárolja a *flageket*.

7	6	5	4	3	2	1	0
Z	S	H	C	0	0	0	0

3.1. táblázat. Az F Flag regiszter bitjei

A 3.1-es táblázatról leolvashatók az F regiszter *flagjainak* helyzete. A rövidítések jelentései a következők:

- *Z*: *Zero Flag*, értéke akkor 1, ha valamilyen logikai vagy aritmetikai művelet eredménye 0.
- *S*: *Subtract Flag*, értéke akkor 1, ha valamilyen kivonás műveletet végzett a processzor.
- *H*: *Half Carry Flag*, értéke akkor 1, ha valamilyen logikai vagy aritmetikai művelet során az akkumulátorregiszter alsó 4 bitjén túlcordulás vagy alulcsordulás keletkezett.
- *C*: *Carry Flag*, értéke akkor 1, ha valamilyen logikai vagy aritmetikai művelet során az akkumulátorregiszteren túlcordulás vagy alulcsordulás keletkezett. Nem összekeverendő a *Half Carry Flaggel*.

Az alsó 4 bit használaton kívüli, minden 0. Ezeket a *flageket* a processzor utasításai vezérlők a művelet kimenetelétől függően. Az emulátor fejlesztése szempontjából ez egy kritikus pont – a *flagek* pontos emulációján nagyon sok múlik, elképesztő mennyiségű hibakeresést spórolhatunk meg magunknak azzal ha odafigyelünk az implementációkor. A két 16 bites regiszter szerepe megegyezik ehhez hasonló architektúrákban tapasztaltakhoz: a PC, azaz *Program Counter* azt mutatja, hogy a processzor hol tart az utasítási sorban (a memóriában), míg az SP, vagy *Stack Pointer* mutatja a hívási verem (*Call Stack*) aktuális címét, azaz hogy éppen hol helyezkedik el a memóriában a verem legfelső eleme.

¹ Olyan regiszter, amiben az aritmetikai-logikai egység által végzett műveletek operandusai, illetve az eredmény átmenetileg tárolódik.

Érdemes megjegyezni, hogy a verembe pakoláskor – azaz a PUSH művelet hívásakor – a verem új, legfelső elemét a memóriában lefelé haladva szúrjuk be, azaz a legkisebb memória című verembeli elem képzi a verem tetejét.

Ami az implementációt illeti, a 8 bites regisztereket 8 bites *Unsigned Integer*ként, azaz u8-ként, a 16 biteseket pedig u16-ként deklaráltam, hiszen negatív értéket nem fog felvenni egyik regiszter sem. A processzort modellező struktúra tehát ilyen módon fog kinézni:

```
pub struct CPU {
    pub A : u8,
    pub B : u8,
    pub C : u8,
    pub D : u8,
    pub E : u8,
    pub F : u8,
    pub H : u8,
    pub L : u8,
    pub SP : u16,
    pub PC : u16,
    pub RAM : [u8; 65536],
}
```

3.1.2. Ciklusok, frekvenciák

A Nintendo Game Boy processzorának frekvenciája 4.194304 MHz. Azonban az egész rendszer a memóriához kötött, tulajdonképpen a memória elérésének sebessége meghatározza az összes egység sebességét is. Jelen esetben a memória az architektúra szűk keresztmetszete, ugyanis ~1 MHz sebességgel működik. A hardver egyes komponenseinek sebessége az alábbiak szerint alakul:

CPU	4,194,304 Hz = ~4 MHz
RAM	1,048,576 Hz = ~1 MHz
PPU	4,194,304 Hz = ~4 MHz
VRAM	2,097,152 Hz = ~2 MHz

3.2. táblázat. A Game Boy fő elemeinek sebessége

A fenti 3.2-es táblázaton látható, hogy az alkotóelemek sebességei nem egységesek, viszont hiába gyors a processzor, ha a memória visszafogja a rendszert. Így praktikussági okokból kétféle ciklust különböztetünk meg:

- az *órajel ciklust*, ami a specifikációban írt 4,194,304 Hz (azaz 4 MHz)-nek felel meg,
- és az *gépi ciklust*, ami megegyezik a RAM frekvenciájával, így 1,048,576 Hz (azaz 1 MHz) lesz.

Ettől a ponttól kezdve az utasítások időzítését gépi ciklus szerint tekintjük, és a ciklus kifejezés alatt a gépi ciklust értjük. Az oka ennek az, hogy az 1 MHz-es gépi ciklus minden komponens sebességére visszavezethető, egyfajta közös nevezőnek tekinthető.

3.1.3. Betöltés - dekódolás - végrehajtás

A betöltés-dekódolás-végrehajtás ciklus függvényeit az Opcode struktúra tartalmazza. A betöltést a `fetch` függvény valósítja meg, amely visszatér a CPU RAM-ot reprezentáló tömbjének PC regiszter értékével megegyező indexű elemével (amely `u8` típusú). A dekódolás és a végrehajtás műveletek az `execute` függvényben történnek. A `fetch` által visszatérít bájt dekódolása úgy történik, hogy az Opcode struktúra rendelkezik egy `opc` és egy `cb_opc` tömbbel, a processzor műveletet azonosító bájtot átadjuk az `opc` tömb számára, mint tömbindex. Az `opc` és a `cb_opc` tömbök rendre 256 eleműek, és függvényekre mutató pointereket tartalmaznak, így a bájt tömbindexként való használatával meghívható az adott opkódhoz tartozó utasítást megvalósító függvény.

Amennyiben CB prefixű opkód az aktuális utasítás, úgy először az `opc[0xCB]` által mutatott függvény hívódik meg, amely meghívja a `fetch` függvényt, hogy az betöltsse a CB prefixű táblán értelmezett művelet opkódját. Ugyaninnen hívódik meg a konkrét műveletet megvalósító függvény a `cb_opc` függvény pointer tömb használatával.

A processzor műveleteit megvalósító függvények visszatérési értéke egy `u8` szám, amely azt mutatja, hogy a konkrét művelet teljes végrehajtása hány processzor ciklus alatt történik meg. Az Opcode struktúra `execute` függvénye ezt az értéket szintén visszatérési értékként fűzi tovább.

A processzor a fenti műveleteket hívja meg a ciklus minden egyes iterációjában, majd a műveletek által visszaadott műveleti ciklus értékeit összeadva szinkronizálja össze a művelet végrehajtást a *rendereléssel*. A 60 FPS² renderelési sebességet céloztam meg, amely az eredeti konzol képernyőfrissítési értékével is nagyságrendileg megegyezik. A *renderelés* ideje így a következő számítás alapján megkapható:

$$4194304/60 = 69905,$$

ahol a 4194304-es érték a processzor órajel ciklusa, a 60 jelzi az FPS értéket, a végeredmény pedig megmutatja, hogy mekkora összegig kell folytatni a processzor műveleteinek végrehajtását. A fő ciklus sémája kiegészítve a szinkronizált PPU *rendereléssel* tehát így alakul:

```
loop { // endless loop
    while cycle <= 69905 {
        cycle += opcode.execute();
    }
    ppu.render();
}
```

² Frames Per Second, azaz képkocka per másodperc – a *renderelés* frissítési gyakoriságát megadó mértékegység.

3.2. Interrupt kezelés

Az utasításkészlet részletes bemutatása előtt fontos szót ejteni a Game Boy *interrupt* kezeléséről – több utasítás kapcsán is elő fog jönni ez a téma.

Adott események bekövetkezése (ez hardverenként eltér) *interruptot*, vagy **megszakítást** vált ki, ezzel kényszeríti a CPU-t, hogy az éppen futó programot azonnal felfüggeszze, és egy speciális eljárást, a **megszakításkezelőt** végrehajtsa, amely a hibaellenőrzést és egyéb speciális teendőket elvégezve értesíti a vezérlőt, hogy a megszakítás befejeződött.[1]

A Nintendo Game Boy architektúrájában kétféle megszakítást különböztetünk meg: létezik szoftveres, és hardveres megszakítás is. Ami a szoftveres megszakításokat illeti, ezeket a hardver programozói használhatták, az egyes RST (a RESET rövidítése) műveletekkel lehet előre definiált memóriacímekre ugrani. A hardveres megszakítások témaköre már kicsit bonyolultabb.

Alapvetően 5 féle hardveres megszakítást különböztetünk meg:

- *V-Blank*: A képernyő frissítése során periodikusan előidézett megszakítás, a későbbiekben – a PPU-t taglaló fejezetben – részletesebben is szó lesz rólá.
- *LCD STAT*: Többféle esemény is előidézheti ezt a típusú megszakítást, az egyik leggyakoribb ezek közül az, amikor a hardver egy adott sor újratárolásánál tart az LCD kijelzőn.
- *Timer*: Akkor következik be ez a megszakítás, amikor a TIMA időzítő regiszter túlcordul. A későbbiekbén erre is kitérek.
- *Serial*: A hardveren található soros port működése közben következik be a *Serial* megszakítás, ha egy konkrét adatátvitel befejeződött. Ezen emulátor esetében ezt a megszakítást nem implementáltam.
- *Joypad*: Ez a megszakítás bármelyik hardveres gomb lenyomásakor aktiválódik.

A megszakításokhoz a CPU kapcsán három dolog köthető a fentieken kívül. Egy fő interrupt kapcsoló, az *Interrupt Master Enable Flag*, amivel le lehet tiltani, vagy éppen engedélyezni lehet a megszakításokat *en bloc*, egy ún. *Interrupt Enable* regiszter, ahol külön-külön lehet engedélyezni vagy letiltani az egyes megszakításokat, illetve egy *Interrupt Flag* regiszter, amiben a megszakítási sorban várakozó, még (a CPU által) teljesítetlen megszakítások szerepelnek.

0xFFFF	Interrupt Enable	0xFF0F	Interrupt Flag
4	<i>Joypad</i>	4	<i>Joypad</i>
3	<i>Serial</i>	3	<i>Serial</i>
2	<i>Timer</i>	2	<i>Timer</i>
1	<i>LCD STAT</i>	1	<i>LCD STAT</i>
0	<i>V-Blank</i>	0	<i>V-Blank</i>

3.3. táblázat. Az *Interrupt Enable* és az *Interrupt Flag* regiszterek megszakítások szerinti bit kiosztása

3.3. Utasításkészlet

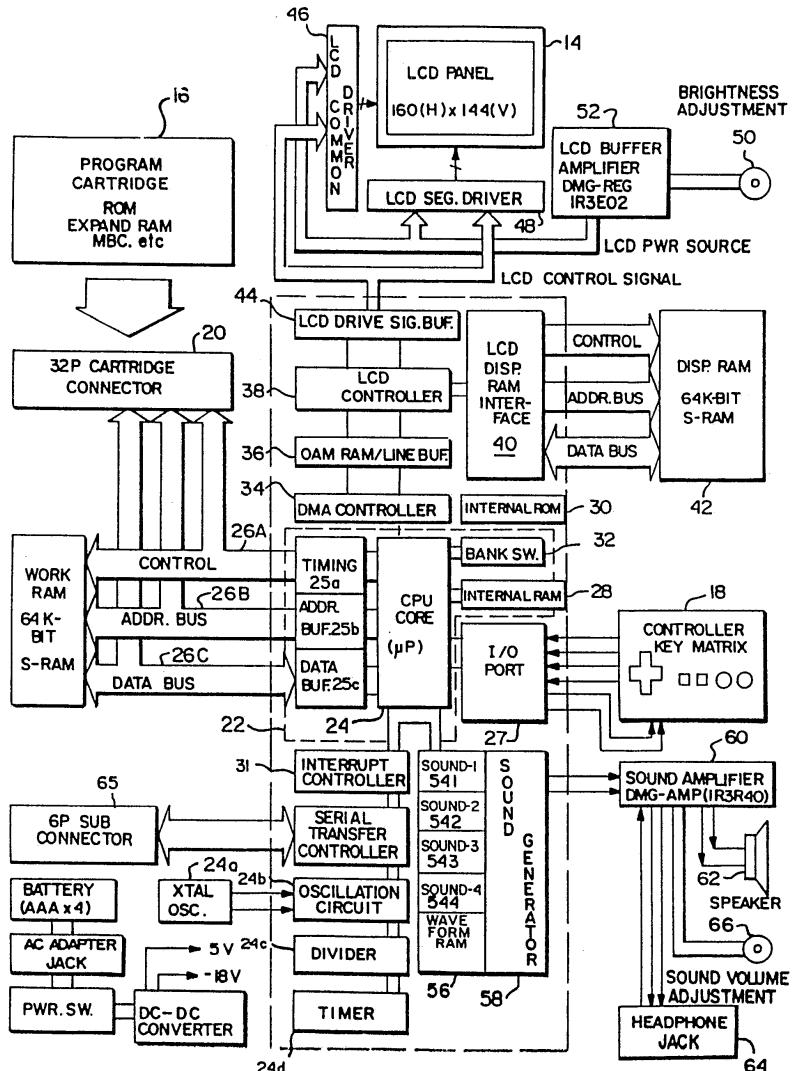
3.4. RAM

3.5. Időzítők

4. fejezet

Függelék

4.1. A Nintendo Game Boy hivatalos architektúrája



4.1. ábra. A szabadalomban szereplő Game Boy architektúra

4.2. A processzor opkód táblái

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF	
0x	LD BC, #16	LD (BC),A	INC BC	INC B	DEC B	LD B, #8	RCL A	UD (alt),SP	ADD HL,BC	LD A,(BC)	DEC BC	INC C	DEC C	LD C, #8	RRC A		
1x	1 - 4	3 - 12	1 - 8	1 - 8	1 - 4	1 - 4	2 - 8	0 - 00	1 - 4	3 - 20	1 - 8	1 - 4	2 - 8	0 - 00	1 - 4	0 - 00	
2x	STOP 0	UD DE, #16	LD (DE),A	INC DE	INC D	DEC D	LD D, #8	RCL A	JR FB	ADD HL,DE	LD A,(DE)	DEC DE	INC E	DEC E	UD E, #8	RRA	
3x	2 - 4	3 - 12	1 - 8	1 - 8	1 - 4	1 - 4	2 - 8	0 - 00	1 - 4	3 - 20	1 - 8	1 - 4	2 - 8	0 - 00	1 - 4	0 - 00	
4x	JR Z,F,B	LD HL, #16	LD (H+),A	INC HL	INC H	DEC H	LD H, #8	DAA	JR Z,F,B	ADD HL,HL	LD A,(H+)	DEC HL	INC L	DEC L	UD L, #8	CPL	
5x	2 - 12/8	3 - 12	1 - 8	1 - 8	1 - 4	1 - 4	2 - 8	0 - 00	1 - 4	2 - 12/8	1 - 8	1 - 4	2 - 8	0 - 00	1 - 4	- 11 -	
6x	JR NC,F,B	LD (H+),A	INC SP	INC (HL)	DEC (HL)	LD (HL),#8	SOF	JR G,F,B	ADD SP,HL	LD A,(H+)	DEC SP	INC A	DEC A	UD A, #8	GCF		
7x	2 - 12/8	3 - 12	1 - 8	1 - 8	1 - 4	1 - 4	2 - 12	- 0 - 0	1 - 4	2 - 12	1 - 8	1 - 4	2 - 12	- 0 - 0	1 - 4	- 0 - 0	
8x	LD B,B	LD B,C	LD B,D	LD B,E	LD B,F	LD B,L	LD B,(HL)	LD C,B	LD C,D	LD C,E	LD C,H	LD C,L	LD C,M	LD C,N	LD C,(HL)	LD CA	
9x	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4		
10x	LD D,B	LD D,C	LD D,D	LD D,E	LD D,F	LD D,L	LD D,(HL)	LD D,A	LD E,B	LD E,C	LD E,D	LD E,H	LD E,L	LD E,M	LD E,(HL)	LD EA	
11x	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4		
12x	LD H,B	LD H,C	LD H,D	LD H,E	LD H,F	LD H,L	LD H,(HL)	LD A,B	LD L,B	LD L,D	LD L,H	LD L,L	LD L,M	LD L,(HL)	LD LA		
13x	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4		
14x	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),F	HALT	LD (HL),A	LD A,B	LD A,C	LD A,D	LD A,H	LD A,L	LD A,M	LD A,(HL)	LD AA		
15x	1 - 8	1 - 8	1 - 8	1 - 8	1 - 8	1 - 8	1 - 8	1 - 8	1 - 8	1 - 8	1 - 8	1 - 8	1 - 8	1 - 8	1 - 8		
16x	ADD A,B	ADD A,C	ADD A,D	ADD A,E	ADD A,F	ADD A,L	ADD A,(HL)	ADD A,B	ADC A,B	ADC A,C	ADC A,D	ADC A,H	ADC A,L	ADC A,(HL)	ADD A,A		
17x	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4		
18x	Z, H, C	Z, H, C	Z, H, C	Z, H, C	Z, H, C	Z, H, C	Z, H, C	Z, H, C	Z, H, C	Z, H, C	Z, H, C	Z, H, C	Z, H, C	Z, H, C	Z, H, C		
19x	SUB B	SUB C	SUB D	SUB E	SUB F	SUB L	SUB (HL)	SUB A	SBC A,B	SBC A,C	SBC A,D	SBC A,H	SBC A,L	SBC A,(HL)	SBC AA		
20x	Z, 1, H	Z, 1, H	Z, 1, H	Z, 1, H	Z, 1, H	Z, 1, H	Z, 1, H	Z, 1, H	Z, 1, H	Z, 1, H	Z, 1, H	Z, 1, H	Z, 1, H	Z, 1, H	Z, 1, H		
21x	AND B	AND C	AND D	AND E	AND H	AND L	AND (HL)	XOR B	XOR C	XOR D	XOR R	XOR L	XOR R	XOR A	XOR A		
22x	A, X	A, X	A, X	A, X	A, X	A, X	A, X	A, X	A, X	A, X	A, X	A, X	A, X	A, X	A, X		
23x	Z, 0, 10	Z, 0, 10	Z, 0, 10	Z, 0, 10	Z, 0, 10	Z, 0, 10	Z, 0, 10	Z, 0, 00	Z, 0, 00	Z, 0, 00	Z, 0, 00	Z, 0, 00	Z, 0, 00	Z, 0, 00	Z, 0, 00		
24x	OR B	OR C	OR D	OR E	OR H	OR L	OR (HL)	OR A	CP B	CP C	CP D	CP E	CP F	CP G	CP (HL)	CP A	
25x	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4		
26x	Z, 0, 00	Z, 0, 00	Z, 0, 00	Z, 0, 00	Z, 0, 00	Z, 0, 00	Z, 0, 00	Z, 0, 00	Z, 0, 00	Z, 0, 00	Z, 0, 00	Z, 0, 00	Z, 0, 00	Z, 0, 00	Z, 0, 00		
27x	RET NZ	POP BC	JP NZ,A,16	JP a16	CALL NZ,A,16	PUSH BC	ADD A,d8	RST 00H	RET Z	JP Z,a16	PREFIX CB	CALL Z,a16	CALL a16	ADC A,0B	RST 08H	RST 18H	
28x	1 - 20/8	- - -	1 - 12	3 - 16/12	3 - 16	3 - 24/12	1 - 16	1 - 16	1 - 20/8	1 - 16	3 - 16/12	3 - 24/12	3 - 24/12	3 - 24/12	3 - 24/12	3 - 24/12	
29x	RET NC	POP DE	JP NC,F,B	CALL NC,A,16	PUSH DE	SUB 08	RST 10H	RET C	RET	JP C,a16	CALL C,a16	CALL a16	SBC A,B	SBC A,0B	RST 08H	RST 18H	
30x	1 - 20/8	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	
31x	UDN (a8),A	POP HL	LD (C),A	- - -	- - -	PUSH HL	AND C,B	RST 20H	ADD SP,F8	JP (HL)	UD (a16),A	- - -	- - -	- - -	RST 20H	RST 20H	
32x	2 - 12	1 - 12	2 - 8	- - -	- - -	1 - 16	2 - 8	1 - 16	2 - 16	1 - 4	3 - 16	2 - 8	2 - 8	1 - 16	RST 20H	RST 20H	
33x	UDN A,(a8)	POP AF	LD A,(C)	- - -	- - -	PUSH AF	OR D,B	RST 30H	LD HL,SP+8	LD SP,HL	UD A,(a16)	ELI	- - -	- - -	OP B	RST 30H	
34x	2 - 12	1 - 12	Z, N, H	C	- - -	1 - 4	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	2 - 8	1 - 16	

4.2. ábra. Az első 256 opkódot tartalmazó tábla

4.3. ábra. A második, CB prefixű 256 opkódot tartalmazó tábla

Nyilatkozat

Alulírott szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Tanszékén készítettem, diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy diplomamunkámat a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

Szeged, 2018. március 20.
aláírás

Köszönetnyilvánítás

Irodalomjegyzék

- [1] Andrew S. Tanenbaum, *Számítógép-architektúrák*, Panem Könyvkiadó Kft., Budapest, 2006.
- [2] Wikipedia, *Game Boy*, https://hu.wikipedia.org/wiki/Game_Boy.