

**Szegedi Tudományegyetem**  
**Informatikai Intézet**

**Nintendo Game Boy Zilog Z80 emulátor  
implementálása**

Diplomamunka

*Készítette:*  
**Krizsák Tibor**  
programtervező informatikus MSc  
szakos hallgató

*Témavezető:*  
**Dr. Tanács Attila**  
egyetemi adjunktus

Szeged  
2018

# Tartalomjegyzék

Feladatkiírás . . . . .	4
Tartalmi összefoglaló . . . . .	5
Bevezetés . . . . .	6
<b>1. Az emulátorok és a Nintendo Game Boy</b>	<b>7</b>
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
<b>2. A fejlesztési folyamat</b>	<b>13</b>
2.1. A fő ciklus . . . . .	14
2.2. Alkalmazott eszközök . . . . .	15
2.2.1. A Rust programozási nyelv . . . . .	15
2.2.2. A minifb könyvtár . . . . .	16
2.2.3. Fejlesztői környezet . . . . .	17
2.2.4. Debugger . . . . .	17
2.2.5. Memóriatérkép . . . . .	18
2.3. A feladat specifikációja . . . . .	19
<b>3. A processzor és a memória implementációja</b>	<b>21</b>
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
3.3.1. <i>Load</i> utasítások - LD . . . . .	26
3.3.2. Aritmetikai utasítások - ADD, ADC, SUB, SBC, INC, DEC . . . . .	27
3.3.3. Logikai utasítások - AND, XOR, OR, CP . . . . .	29
3.3.4. Verem utasítások - PUSH, POP . . . . .	30
3.3.5. Eljárás utasítások - JP, CALL, RET . . . . .	30
3.3.6. Bitmanipulációs utasítások - BIT, RES, SET, SWAP . . . . .	30

3.3.7. <i>Rotate</i> és <i>Shift</i> - RLC, RRC, RL, RR, SLA, SRA, SRL . . . . .	31
3.3.8. Egyéb, speciális utasítások . . . . .	32
3.4. Memória . . . . .	33
3.4.1. DMA . . . . .	34
3.4.2. Memory Bank Controller . . . . .	35
3.5. Időzítők . . . . .	36
3.5.1. TIMA . . . . .	36
3.5.2. DIV . . . . .	38
<b>4. A kijelző és a PPU implementációja</b> . . . . .	<b>39</b>
4.1. Az LCD kijelző . . . . .	39
4.1.1. Az LCD státuszok . . . . .	41
4.1.2. Az LCD interruptjai és flagje . . . . .	42
4.2. PPU - Pixel Feldolgozó Egység . . . . .	42
4.2.1. Alapvető tudnivalók . . . . .	42
4.2.2. Az LCD Control Register . . . . .	43
4.2.3. Tile rendering . . . . .	45
4.2.4. Sprite rendering . . . . .	49
<b>5. Joypad</b> . . . . .	<b>51</b>
5.1. Technikai háttér és a <i>Joypad Register</i> . . . . .	51
5.2. Implementáció . . . . .	53
5.2.1. Gombnyomás detektálás . . . . .	53
<b>6. Függelék</b> . . . . .	<b>56</b>
6.1. A Nintendo Game Boy hivatalos architektúrája . . . . .	56
6.2. A processzor opkód táblái . . . . .	57
Nyilatkozat . . . . .	59
Köszönnetnyilvánítás . . . . .	60
Irodalomjegyzék . . . . .	61

# 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 videojáték programozók és a *demoscene*<sup>1</sup> 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 videojáték-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 videojáték 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 törekvést.

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

<sup>1</sup> 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)

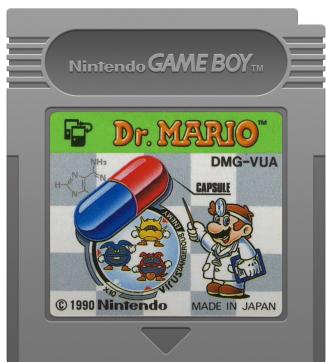
- **Hang**: 2 négyszögjel generátor, 1 programozható 32 mintás 4 bites PCM hullám, 1 fehér zaj, és egy audio bemenet a kazettából. (A külső kazetta bemenetet soha egy piacra dobott játék sem használta.) A jack kimeneten keresztül sztereó hangot ad.
- **Kijelző**:  $166 \times 144$  pixel felbontású LCD 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.

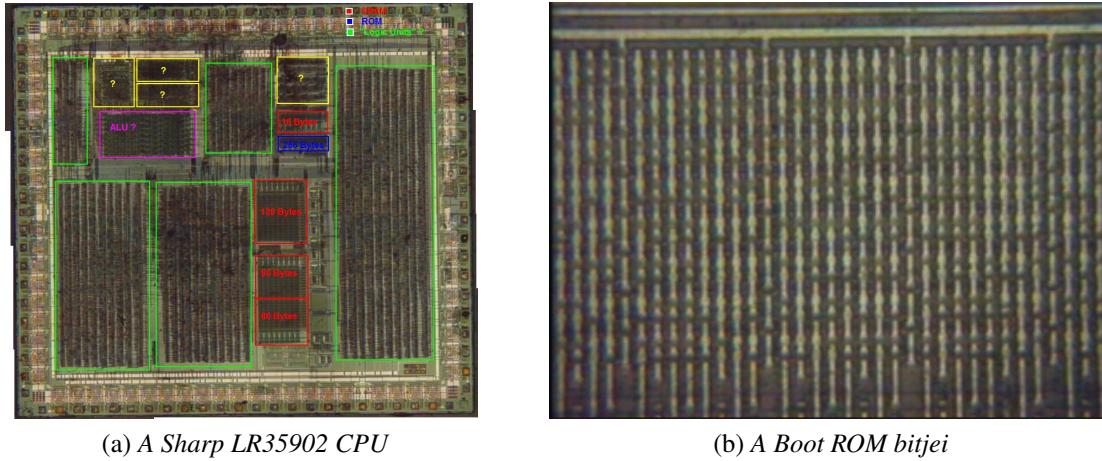
További érdekesség még a *cartridge*-ekkel kapcsolatban a Nintendo által kifejlesztett korabeli (de meglepően hatékony) másolásvédelmi és *homebrew*<sup>2</sup>-fejlesztőket kizáró mechanizmus. Annak érdekében hogy a kazettákat ne lehessen lemásolni és így terjeszteni, illetve a gyártóval kapcsolatba 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

<sup>2</sup> 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.



2.1. ábra. Az emulátor architektúrája

A 2.1-es 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 egységként 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 vázlatszerűen 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 későbbiekbén 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 megkönnyíthetjük 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*<sup>1</sup>, 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.

### 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 közösségi 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 *library*ket, illetve függőségeket a *crates.io* közösségi központi repozitóriumból szerzi be.

<sup>1</sup>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.

### 2.2.2. 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, Rustban 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 némely operációs rendszer esetén (Windows, macOS) a menürendsereket.

A használata nagyon egyszerű:

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

Amint látható, négy kötelezően megadandó paramétere van a `Window` struktúra konstruktornak, 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 konstruktor 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*<sup>2</sup> rendszeren végeztem. Az emulátor fejlesztés sajátosságai miatt feleslegesnek éreztem egy IDE<sup>3</sup> használatát, hiszen ha a programkód szintaxisa megfelelő, onnantól kezdve a hibakeresést az IDE-k által kínált eszközök sem tudják megkönnyíteni, ahhoz saját *debuggert* kell írni. Ilyen fejlesztői környezet használata helyett tehát a klasszikusnak mondható szövegszerkesztő (Atom, Rust *linterrel*<sup>4</sup>) 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

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 2.2-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`

A `0x21` jelzi az aktuális művelet *opkódját*<sup>5</sup>, mellette szerepel a művelet *mnemonikja*<sup>6</sup>, jelen esetben az `LD HL, nn`. A harmadik tag a művelet által beolvasható, és (operandusként) felhasznált bájtokat tartalmazza, itt: `0xFF 0xE6`. A példában (és a *debuggerben*

<sup>2</sup> Az *elementary OS* egy *Ubuntu* alapú Linux disztribúció.

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

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

<sup>5</sup> Operációkód, azaz műveleti kód, vagy műveleti jelkód, utasításkészletek leírásában műveleti jelrész. A CPU által beolvasható bináris szám, amit végrehajtható utasítás kódjaként értelmez.

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

<b>0x34 : INC (HL)</b>	A 0x00	B 0x00
0x21 : LD HL,nn 0xFF 0xE6	C 0x08	D 0x00
<b>0x34 : INC (HL)</b>	E 0x10	F 0x00
0x21 : LD HL,nn 0xFF 0xE5	H 0xFF	L 0xE6
<b>0x34 : INC (HL)</b>	SP 0xFFFF	
0x21 : LD HL,nn 0xFF 0xE2	PC 0x354	
0x20 : JR M2	FLAG 0b0000	
0x05 : DEC B		
0x2C : INC L		
0x28 : JR Z, 0x1		
0xA7 : AND A,A		
0x7E : LD A,(HL)		
0x20 : JR M2, - 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 M2, 0x1B		
0xFE : CP A, 0xF		
0xEF : AND A, 0xF		

2.2. ábra. Az emulátorhoz fejlesztett debugger

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

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 2.3-es ábrán megfigyelhető, hogy több féle szín is megjelenik – ezek jelölik az egyes fontosabb, elkülönölt részeket a memóriában. A színeket is bevonta a reprezentációba a *debuggerhez* hasonlóan ennél az eszközönél is ránézésre leolvashatók adatok. Ahhoz, hogy pontosan megtudjuk egy adott bájt értékét és pozícióját a memóriában, rá kell kattintani, és a 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.



2.3. ábra. Az emulátorhoz fejlesztett memóriatérkép eszköz

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 DMG<sup>7</sup> videojáték konzolhoz írt videojátékok ROM-jait adhatjuk meg, illetve a visszafejtett Boot ROM-ot. Videojátékok esetén az emulátor csak a MBC (*Memory Bank Controller*) nélküli ROM-okat képes futtatni. Előfordulhatnak olyan nem ismert, videojáték programozók által kihasznált *bugok*, amelyek gátolják a

<sup>7</sup> Az eredeti, klasszikus 1989-ben kiadott Game Boy kódneve DMG.

ROM tökéletes futtatását. Szükséges hogy a ROM tartalmazza a *headerjében* a Nintendo logó bájtjait (a Boot ROM-ban lévő *checksum* kiszámolja 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.

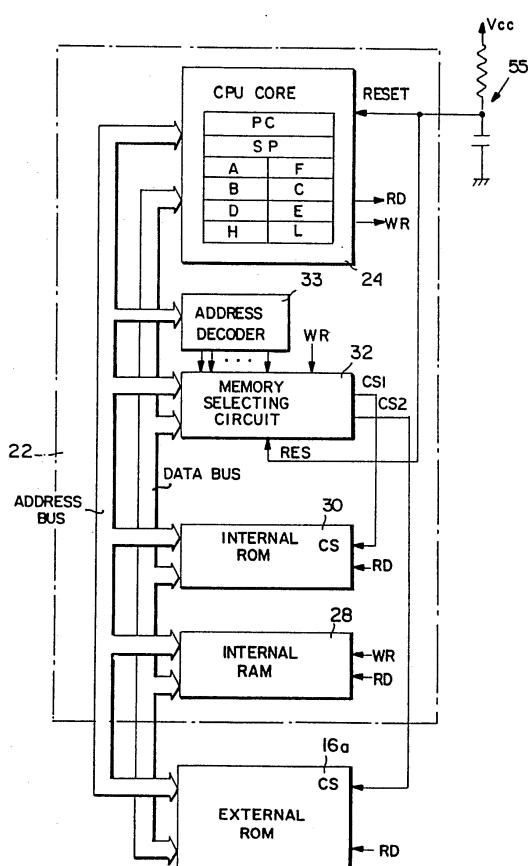
*Az output elvárt formai és tartalmi követelményei:*

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.

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 6.1-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 memoriával 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 akkumulátorregiszter<sup>1</sup>, 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 5.1-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 kimenetétől függően. Az emulátor fejlesztése szempontjából ez egy kritikus pont – a *flagek* pontos emulációján nagyon sok műlik, elképesztő mennyiségű hibakeresést spórolhatunk meg magunknak azzal ha odafigyelünk az implementációkor. Érdemes készíteni a CPU struktúrába *flag* beállító és lekérdező függvényeket is az utasítások implementálásához.

A két 16 bites regiszter szerepe megegyezik ehhez hasonló architektúrákban tapasztalatakhöz: a PC, azaz *Program Counter* azt mutatja, hogy a processzor hol tart az utasítási

<sup>1</sup> Olyan regiszter, amiben az aritmetikai-logikai egység által végzett műveletek operandusai, illetve az eredmény átmenetileg tárolódi.

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. 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 4.2-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 időszítését 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 opkóhoz 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 FPS<sup>2</sup> 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();
}
```

<sup>2</sup> *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 felfüggesse, 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 róla.
- *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.

<b>0xFFFF</b>	Interrupt Enable	Jump Location	<b>0xFF0F</b>	Interrupt Flag
4	<i>Joypad</i>	0x60	4	<i>Joypad</i>
3	<i>Serial</i>	0x58	3	<i>Serial</i>
2	<i>Timer</i>	0x50	2	<i>Timer</i>
1	<i>LCD STAT</i>	0x48	1	<i>LCD STAT</i>
0	<i>V-Blank</i>	0x40	0	<i>V-Blank</i>

3.3. táblázat. Az *Interrupt Enable* és az *Interrupt Flag* regiszterek megszakítások szerinti bit kiosztása, és a hozzájuk tartozó memóriacímek

Előfordulhat, hogy egyszerre több megszakítás érkezik. Ebben az esetben a prioritási sorrend az *Interrupt Flag* regiszterben elfoglalt helyek alapján alakul: a *V-Blank interrupt*

a legfontosabb, míg a *Joypad* megszakítás marad utoljára. Miután a prioritás alapján a megszakításvezérlő kiválasztotta, hogy melyik megszakítás következzen, a 3.4-as táblázaton szereplő adatoknak megfelelően az adott megszakításhoz tartozó memóriacímre ugrik a vezérlés.

Ami az implementálást illeti, a megszakítás vezérlő feladatait az `Interrupt` struktúra, és annak függvényei látják el. Az `IRQ` függvény segítségével tudnak az egyes modulok (PPU, időzítő, stb.) megszakítást kérni, melyet aztán a `handler` függvény dolgoz fel. A feldolgozás oly módon történik, hogy minden CPU művelet után a fő ciklusban meghívódik az `interrupt_checker` függvény, amely folyamatosan ellenőrzi, hogy érkezett-e új megszakítás, és hogy az engedélyezve van-e az *Interrupt Enable* regiszterben – ha igen, akkor a megszakítás azonosítóját paraméterként átadva meghívja a `handler` függvényt, amely a megszakításhoz tartozó memóriacímre ugrasztja a CPU-t.

### 3.3. Utasításkészlet

Ahogy azt már az előző alfejezetben ismertettem, az `Opcode` struktúrában implementált processzor műveletek visszatérési értéke egy `u8` típusú egész szám, amely a művelet végrehajtási idejével lesz egyenlő. Azt azonban még nem ismertettem, hogy hogyan néz ki egy utasítás sematikusan.

```
fn opcode_name_ff(&mut self, cpu : &mut CPU) -> u8 {  
    //...  
    4  
}
```

A fenti kódrészleten látható, hogy az egyes utasítások függvényeinek neve tartalmazza az utasítás nevét – *mnemonik* vagy bővebb alakban –, illetve az *opkódját*. A paraméterlistában szerepel a `&mut self` paraméter, amely a Rust nyelvben azt jelenti, hogy az adott függvény egy osztályfüggvény, és hogy módosíthatja az osztályt reprezentáló objektumot. A második paraméter (`&mut CPU`) pedig a fő CPU struktúrára mutató *mutable* referencia, azaz olyan `cpu` objektumot kapunk, amely módosítható – Rust nyelven *borrow*-oltuk, azaz kölcsön kaptuk az objektumot.

Sok esetben előfordul, hogy egy művelet egy vagy két operandussal rendelkezik, melyeket külön be kell tölteni a memoriából. Ezt a művelet függvényén belül tesszük meg a `fetch` metódus meghívásával.

#### 3.3.1. Load utasítások - LD

A *Load*, azaz betöltő műveletek legnagyobb része nagyon egyszerűen működik: adott két regiszter – a példánkban legyen ez most X és Y –, ekkor a művelet így alakul:

`LD X, Y`

Ez triviálisan az Y regiszter értékét tölti be az X regiszterbe - ennek megvalósítása tulajdonképpen egy egyszerű kifejezés. Fontos hozzáenni, hogy az Y helyén állhat egy bájt is. Egyik *Load* művelet sem befolyásolja a *flagek* állapotát.

A továbbiakban ismertetésre kerülnek a fenti alapesettől eltérő betöltő utasítások.

### Beltöltés memóriacímről, vagy memóriacímre

A *Load* műveletek másik típusa esetén az X vagy Y regiszter helyett állhat memóriacím is. A cím értéke átadható (HL) összevont regiszter formában: `LD A, (HL)`, vagy két külön betöltött bájt összevonásaként: `LD A, (0x2f44)`.

### A *Stack Pointer* betöltője

A *Stack Pointer* betöltése a HL regiszterpárba az előzőhekhez képest bonyolultabb művelet. Egyetlen paraméteréből várja azt a *signed*, tehát előjeles(!) értéket, amely a betöltendő memóriahely címének távolságát mutatja az SP regiszterhez képest. A `fet ch` függvényel betöltött bájtot először *castolni* kell `i8` típusúra, majd az előjel vizsgálatát követően betölteni az adatot a HL-be a megfelelő helyről. A művelet az alábbiak szerint néz ki:

`LDHL SP, 0x42`

### Egyéb speciális betöltők

Több speciális betöltő művelet is elérhető, ezeket a fentiekhez való hasonlóságuk miatt csak felsorolásszerűen fogom ismertetni:

- `LD A, (C)`: ezzel a művelettel a *High RAM* részre tudunk írni – a C regiszter értékéhez hozzáadódik még a 0xFF00 cím, ezzel megkapva a pontos címet. Fordított operandusokkal is működik.
- `LDH A, (0x42)`: az előzőhez hasonló művelet, annyi különbséggel, hogy itt a C regiszter helyett paraméterben megadható az 0xFF00-hez mért eltolás értéke. Fordított operandusokkal is működik.
- `LD A, (HL+)`: a HL értékének megfelelő memóriahely A regiszterbe töltése után inkrementálja a HL értékét eggyel. Fordított operandusokkal is működik.
- `LD A, (HL-)`: az előzővel megegyező módon működik, inkrementálás helyett dekrementálással.

### 3.3.2. Aritmetikai utasítások - ADD, ADC, SUB, SBC, INC, DEC

Az aritmetikai utasítások alatt ezen az architektúrán az összeadás, kivonás, inkrementálás és dekrementálás műveleteket értjük. Ezek lényegi működése persze triviális, viszont a műveletek hatása a *flag* regiszterre fontos tényező, így ezt célszerű részletesebben megvizsgálni. Külalakukban megegyeznek, példaként szerepeljen az összeadás szintaxisa:

`ADD A, B`

amely értelemszerűen az A regiszter értékét teszi egyenlővé az A és B regiszterek értékeinek összegével.

## Összeadás

Két típusú összeadást megvalósító művelet áll a programozók rendelkezésére: az [ADD](#), és az [ADC](#). Ezen utasítások esetében közös probléma a túlcsordulás – hogyan kezeljük ha az összeg nagyobb, mint 255, azaz nem fér bele az [u8](#) adattípusba? Szerencsére a Rust nyelv biztosít erre egy olyan megoldást, amely a hardver viselkedésével megegyezik. A két [u8](#) típusú operandust nem a hagyományos módon ([u8 + u8](#)) adjuk össze, hanem a `wrapping_add` függvény segítségével a következő módon:

```
let a : u8 = 5;
let b : u8 = 6;
println!("{} ", a.wrapping_add(b)); // Output: 11
```

A fenti függvény abban az esetben, ha az összeg nagyobb lenne, mint 255, körbe *wrappeli* az eredményt, azaz elvégez egy  $\text{mod } 256$  műveletet az összegen. Így a túlcsordulás problémáját sikerült kiküszöbölni, az összeadás bármely értékkal elvégezhető úgy, hogy biztosan nem lépünk ki az adattípus méretéből.

Ami a *flageket* illeti, az összeadások esetében mindegyikük érintett:

- Z: állítsuk be 1-re, ha az összeg értéke 0, egyébként legyen az új értéke 0,
- N: mivel nem kivonásról van szó, az értékét állítsuk 0-ra,
- H: ha *half-carry* történik, állítsuk 1-re, egyébként 0-ra,
- C: ha túlcsordulás történik, állítsuk 1-re, egyébként 0-ra.

A *half-carry* avagy fél-túlcsordulás detektálása a következő képpen történik:

```
if (a & 0xF) + (b & 0xF) > 0xF {
    cpu.set_flag("H");
} else {
    cpu.reset_flag("H");
}
```

Látható, hogy az elágazás logikai kifejezésének bal oldalán a két operandus értékének vesszük külön-külön az alsó 4 bitjét (`a & 0xF`, `b & 0xF`), majd ha azok összege nagyobb, mint 15 (binárisan 00001111), akkor fél-túlcsordulás következett be. Ekkor állítsuk a H *flaget* 1-re, egyébként pedig 0-ra.

A C-vel jelölt túlcsordulás ellenőrzése triviális.

A különbség az [ADD](#), és az [ADC](#) műveletek között az, hogy – mint az a *mnemonikból* is kikövetkeztethető – az [ADC](#) az egyszerű összeadás mellett az összeghez hozzáadja a C flag értékét is (*ADD with Carry*). A flag értékének hozzádását szintén a `wrapping_add` függvénnyel tesszük.

## Kivonás

A [SUB](#), és az [SBC](#) kivonás műveletek nagyon hasonlóak a fentebb részletezett összeadás műveletekhez, természetesen kivonásos formában. `wrapping_sub` függvény helyett `wrapping_sub` függvényt használunk, amely negatív különbség esetén a másik irányba *wrappeli* át a végeredményt, szintén a  $\text{mod } 256$  számítást alkalmazva.

A *flagek* az összeadás műveletekhez képest a következőképpen alakulnak:

- Z: megegyezik az összeadásnál ismertetettel,
- N: ebben az esetben kivonásról van szó, így az értéke legyen 1,
- H: megegyezik az összeadásnál ismertetettel,
- C: ha alulcsordulás történik, állítsuk 1-re, egyébként 0-ra.

A *half-carry flag* beállítása megegyezik ugyan az összeadásnál tapasztalattal, viszont a detektálása más módon történik:

```
if (a & 0x0F) < (b & 0x0F) {
    cpu.set_flag("H");
} else {
    cpu.reset_flag("H");
}
```

Az elágazás feltételes logikai kifejezésében láthatjuk, hogy a két operandus alsó 4 bitjeit hasonlítjuk össze egymással. Ha a kivonandó ilyen módon nagyobb mint a kisebbítendő, akkor fél-alulcsordulás történik – ekkor állítjuk 1-re a megfelelő bitet, egyébként pedig 0-ra.

Alulcsordulás akkor történik, ha a különbség kisebb, mint 0.

Az **SBC** művelet ebben az esetben is annyiban különbözik a hagyományos kivonástól, hogy a C *carry flaget* is kivonja a kisebbítendőből.

### Inkrementálás, dekrementálás

Az inkrementálást (**INC**) és a dekrementálást (**DEC**) megvalósító műveletek gyakorlatilag rendre megegyeznek az összeadás, illetve kivonás műveletekkel, annyi különbséggel, hogy a második operandust minden esetben 1-nek tekintjük.

#### 3.3.3. Logikai utasítások - **AND**, **XOR**, **OR**, **CP**

Az és (**AND**), kizáró vagy (**XOR**), vagy (**OR**), és összehasonlító (**CP**, *compare*) műveletek is legalább olyan alapvetőek, és fontosak mint az előzőekben tárgyalt aritmetikai műveletek. Közülük az első három nagyon egyszerű megvalósítani, hiszen a legtöbb programozási nyelv képes ezen műveletek elvégzésének reprezentálására az & (és), ^ (kizáró vagy), és a | (vagy) operátorok segítségével.

Az fent említett első három művelet implementálása tehát könnyű feladat, a *flagok* beállítása pedig szintén egyszerű. A **XOR** és **OR** flag kezelése megegyezik: ha a végeredmény 0, akkor a Z flag legyen 1 (egyébként 0), az összes többi flaget pedig állítsuk 0-ra. Az **AND** esetén is hasonló a helyzet, annyi különbséggel, hogy a H flaget minden esetben 1-ra kell állítani.

A **CP** művelet kicsit különbözik a többitől – maga a művelet összehasonlít két értéket, és ha az értékük megegyezik, a Z flaget 1-re állítja. Ennek működése tulajdonképpen megegyezik a kivonás műveletével, csak a különbséget nem tároljuk sehol, a *flagok* beállítása a fontos – ebben a tekintetben teljesen ekvivalens az összehasonlítás a kivonással.

### 3.3.4. Verem utasítások - **PUSH**, **POP**

Lehetőség van regiszterpárok eltárolására a veremben – ezt a funkciót a **PUSH** és a **POP** utasítások valósítják meg. A **PUSH** az AF, BC, DE, HL (konkrét műveletben szereplő) regiszterpárokat teszi bele a verembe (az SP által mutatott memóriacímre), a második regiszter taggal kezdve. A **POP** művelet pedig fordított sorrendben veszi ki az értékeket a veremből, majd állítja be velük a megfelelő regiszterek értékeit.

### 3.3.5. Eljárás utasítások - **JP**, **CALL**, **RET**

Ahhoz, hogy a programozók eljárásokat tudjanak írni a programok fejlesztése során, különféle utasításokra van szükségük. Ezek az utasítások a Game Boy architektúrájában az ugrás (**JP**), az eljáráshívás (**CALL**) és a visszatérés (**RET**).

A **JP** utasítás működése nagyon egyszerű: a CPU PC regiszterét állítja a paraméterben megadott memóriacímre, így a processzor ott fogja folytatni a futását. Több típusú **JP** művelet is szerepel a CPU utasításai között:

- **JP** **(HL)**: a HL regiszterpárban tárolt címre ugrik,
- **JP** **0x4a2a**: a paraméterben megadott 16 bites címre ugrik,
- **JP** **NZ 0x4a2a**: akkor ugrik a paraméterben megadott 16 bites címre, ha a **Z flag** értéke 0,
- **JP** **NC 0x4a2a**: akkor ugrik a paraméterben megadott 16 bites címre, ha a **C flag** értéke 0,
- **JP** **Z 0x4a2a**: akkor ugrik a paraméterben megadott 16 bites címre, ha a **Z flag** értéke 1,
- **JP** **C 0x4a2a**: akkor ugrik a paraméterben megadott 16 bites címre, ha a **C flag** értéke 1.

A **CALL** művelet annyiban hasonlít az előző **JP** művelethez, hogy szintén a paraméterben megadott címre fog ugrani a PC regiszter átállításával, előtte azonban az aktuális PC értékét beleteszi a verembe, elmentve azt. A **JP** művelethez hasonlóan szintén vannak feltételes eljárás hívás utasítások, melyek a **Z** és a **C flagek** aktuális állapota szerint működnek.

A **RET** utasítás az előző (**CALL**) utasítással kéz a kézben jár: kiveszi a veremből a két legfelső értéket – amelyek együtt egy memóriacímet alkotnak –, majd a PC regisztert erre az értékre állítja be, így tulajdonképpen az eljárás hívás végeztével a CPU visszatér arra a címre, ahol az eljárás hívása előtt tartott. Szintén vannak a **Z** és **C flagekhez** kötött **RET** utasítások, illetve létezik még egy **RETI** opkódú utasítás is, amely a visszatérés után engedélyezi az *Interrupt Enable* regiszterben az összes *interruptot*.

### 3.3.6. Bitmanipulációs utasítások - **BIT**, **RES**, **SET**, **SWAP**

Rendelkezésre állnak bitmanipulációs utasítások, amelyekkel regiszterek, vagy a memóriában lévő bájtok bitjeivel végezhetünk műveleteket: változtathatjuk (**RES**, **SET**, **SWAP**),

vagy lekérhetjük (**BIT**) őket. Ezek természetesen fontos műveletek, így a többi CPU utasításhoz hasonlóan törekedni kell a pontos emulációjukra.

A **RES** és **SET** esetében rendre 0 (*reset*) vagy 1 (*set*) értéket adhatunk az utasítás paramétereiben<sup>3</sup> megadott bitnek. Az *első paraméter* a módosítandó bit indexe az adott bájtban, a *második paraméter* pedig maga a módosítandó bitet tartalmazó regiszter vagy (**HL**) regiszterpárral megadott memóriacím. Ezek az utasítások nincsenek hatással az F *flag* regiszterre. Az implementációt tekintve egy egyszerű bitmaszkolással, majd *vagy*, ill. *és* műveletekkel lehet elérni egy-egy bit módosítását egy **u8** változó esetén.

A **BIT** műveettel lehet lekérni egy konkrét bit értékét – az utasítás az eredményét a Z *flag* segítségével tudjuk kiolvasni: amennyiben az eredmény 0 volt, a Z 1-gel lesz egyenlő, egyébként pedig 0-val. Ami a további *flageket* illeti, az N *flaget* mindig *reseteli*, a H-t pedig minden esetben beállítja az utasítás. Implementációban bitmaszkolással tudjuk megkapni egy adott bit értékét.

A **SWAP** utasítás esetében egy paraméter adott: a módosítani kívánt regiszter. Maga a művelet nagyon egyszerű: fel kell cserálni a regiszterrel vagy memóriacímmel megadott bájt felső és alsó 4 bitjét egymással, így tehát például a  $11110000_2$  bájtból a **SWAP** elvégzése után  $00001111_2$  lesz. Ha a végeredmény nullával egyenlő, akkor az utasítás beállítja a Z *flaget* 1-re, egyébként pedig nullára. A maradék három *flag* minden esetben *resetelésre* kerül.

### 3.3.7. Rotate és Shift - **RLC**, **RRC**, **RL**, **RR**, **SIA**, **SRA**, **SRL**

#### Rotate műveletek

A Game Boy processzorának architektúrájában több bitforgató utasítás is van: **RLC**, **RRC**, **RLCA**, **RRCA**, **RL**, **RR**, **RLA**, és **RRA**. Ezek többé-kevésbé ugyan olyan módon működnek. Vegyük példának a legegyszerűbb, balra forgató utasítást, az **RLC**-t. Egy  $11110000_2$  bájt esetén a balra forgatás eredménye a következő lesz:  $11100001_2$ . Ez az érték úgy született, hogy minden bitet egygyel balra tolunk, a 7. bit pedig *körbefordul* – belőle lesz a 0. bit. A forgató műveleteket ez a körbefordulás különbözteti meg az eltoló műveletektől, ott ilyenről nincs szó. Ha az eredeti érték 7. bitje 1-es értékű volt, akkor be kell állítani a C *flaget* is 1-re (egyébként 0-ra). Ami a többi *flaget* illeti, a Z-t akkor állítjuk 1-re ha a végeredmény 0 lett (egyébként 0-ra), a többi *flaget* pedig *reseteljük*. Az **RL** utasítás pontosan ugyanígy működik azzal a kivételel, hogy nem az eredeti bájt 7. bitjéből lesz az új bájt 0. bitje, hanem a C *flagból*.

Az **RRC** és az **RR** is a fentiek szerint alakul, csak nem balra forgat az utasítás, hanem jobbra – az eredeti bájt 0. bitjéből lesz az új bájt 7. bitje. Ebből adódóan az eredeti bájt 0. bitje szerint módosítjuk a C *flaget*.

Az **RLCA**, **RRCA**, **RLA**, és **RRA** utasítások esetén a különbség az előzőekhez képest csupán annyi, hogy mindegyikük esetében 0-ra állítjuk a Z *flaget*, valamint az összes utasítás az A regiszterrel dolgozik.

<sup>3</sup> Paraméter alatt jelen esetben nem külön beolvasott bájtokat tekintünk, hanem a konkrét művelethez (a processzor architektúrájának részeként) specifikált paramétereket.

### **Shift műveletek**

Mint ahogyan a fenti bekezdésben említésre került, a *shift* – azaz eltoló – utasítások esetén nincsen körbeforduló bit: az **SLA** (*Shift Left*) és **SRL** (*Shift Right*) esetén rendre balra, illetve jobbra tolódnak el a bitek, az újonnan bejövők pedig mindenkorban a 0 értéket veszik fel. A *flagek* beállításai is megegyeznek a forgató műveletekkel.

Az **SRA** utasítás kivételt képez, ugyanis ebben az esetben az eredeti bájt 7. bitje helyére nem 0, hanem az eredeti 7. bittel megegyező értékű bit érkezik.

### **3.3.8. Egyéb, speciális utasítások**

Azok az utasítások kerültek ebbe a csoportba, amelyek egytől egyik speciálisak, nem lehet őket nagyobb kategóriához sorolni.

#### **Üres műveletek - NOP, STOP, HALT**

A **NOP**, **STOP** és **HALT** műveletek hasonlítanak egymásra abban a tekintetben, hogy mindegyikük végrehajtása alatt a CPU tulajdonképpen nem csinál semmit. A **NOP** esetében egy előre eltervezett üres műveletről van szó, amelyet várakozások esetén használnak, a **STOP** utasítás addig leállítja a processzort és a kijelzőt amíg gombnyomás nem történik, a **HALT** pedig addig kikapsolva tartja a CPU-t amíg egy megszakítási kérés nem érkezik. Az utolsó két utasítás tipikusan olyan amit egy emulátor kapcsán nem szükséges implementálni, a hardver energiafogyasztásának optimalizálásában van szerepe.

#### **Carry flag műveletek - SCF, CCF**

Két, konkrétan a *flageket* állító utasítás az **SCF** és a **CCF**. Az **SCF**-fel be tudjuk állítani a *Carry flaget*, a **CCF**-fel pedig komplementálni tudjuk az értékét. Az N és H *flageket* mindenkorban *reseteli*.

#### **Komplementer, binárisan kódolt decimális műveletek - CPL, DAA**

Az A regisztert módosító utasítások közé sorolható a **CPL** és a **DAA** művelet.

A **CPL** utasítás – a nevéből is adódóan – az A regiszter fogja komplementálni ( minden bitjét az ellentétre állítja), és az N és H *flageket* beállítani. A Rust nyelvben a komplementálás egy egyszerű felkiáltójel operátorral érhető el, így az A regiszter komplementere az ! A lesz.

A **DAA** művelet az A regiszter értékét binárisan kódolt decimális formájúra hozza, amely azt jelenti, hogy sorra veszi a decimálisan ábrázolt A regiszter számjegyeit, majd azokat binárisan ábrázolja helyiértékük szerint csökkenő sorrendben. Ha a végeredmény 0 lesz, beállítja a Z *flaget* (egyébként *reseteli*), a H-t minden esetben *reseteli*, a C-t pedig a művelettől függően állítja be vagy *reseteli*.

#### **Restart műveletek - RST**

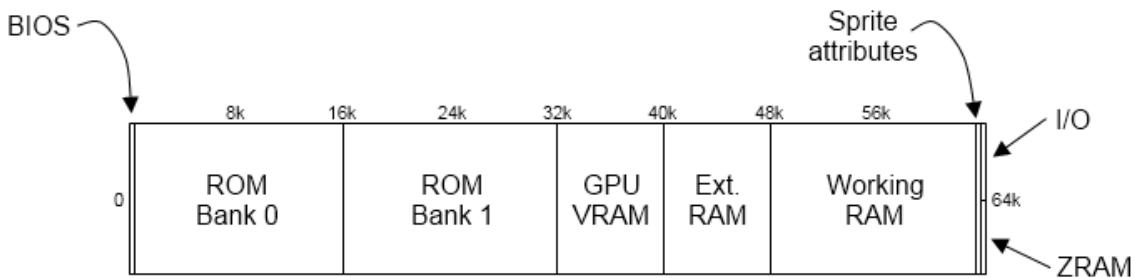
A **RST** műveletekkel a programozók szoftveres megszakításokat tudtak előidézni, melyek – **RST** utasítástól függően – az alábbi memóriacímekre állítják a PC regisztert az aktuális cím verembe helyezése után:

0x00, 0x08, 0x10, 0x18, 0x20, 0x28, 0x30, 0x38.

### Interrupt műveletek - EI, DI

A megszakítások letiltására és engedélyezésére a **DI** és **EI** utasítások valók, melyekkel szabályozni lehet a korábban már említett *Interrupt Master Enable Flaget*. Ez a flag egy, a CPU struktúrájában létező **bool** típusú változó értékét módosítja.

## 3.4. Memória



3.2. ábra. A memória térkép

A Game Boy memóriájának térképe a fenti, 3.2-es ábrán figyelhető meg. Jól látható, hogy több nagyobb részre van osztva, melyek működésükben igencsak elkülönülnek egymástól. Az emulátor implementációját nagyban megkönnyíti, hogy annak ellenére, hogy több modul is használja valamilyen módon a beépített memóriát, annak csak egy címtartománya van, amelyet a következő intervallumbeli bájtok alkotnak: 0x0000 - 0xFFFF. Az alaplapra forrasztott memória tehát 64 kB, amelynek rögtön az első felét a játék ROM-ja teszi ki.

A konzol elindítását követően a hardver első feladata a Boot ROM futtatása. Ez a szoftver a *bootolási* folyamat alatt a 0x0000 - 0x0100 területet foglalja el, amely a folyamat végeztevel kikerül a memóriából (*átmappelődik*), helyére pedig a betöltött játék első 256 bájtja kerül. Ez azt is jelenti, hogy az *átmappelés* után a Boot ROM bájtai többé nem elérhetők.

A játék ROM-jának 32 kB-ja után, a 8 kB méretű 0x8000 - 0x9FFF tartományon helyezkedik el a *Video Memória* (vagy *Grafikus Memória*, VRAM). Itt vannak eltárolva *tile*-ok (csempék), a játékokat felépítő grafikus építőelemek, illetve azok az adatok, melyek leírják, hogy melyik *tile* hol helyezkedik el a képernyőn. A csempék helyét leíró adatok segítségével tudja a Game Boy PPU-ja felépíteni a játék háttereként szolgáló grafikai elemeket. Ezekről bővebben a későbbi, PPU-t részletező fejezetben lesz szó.

A sorban következő 8 kB méretű blokk az 0x8000 - 0x9FFF intervallumon található, a neve pedig *External RAM* vagy *xRAM*. Ez a rész fizikailag a játék kazettáján helyezkedett el, többnyire a játék mentéseket tárolták itt.

A hagyományos értelemben vett RAM a 0xC000 - 0xDFFF területet foglalta el – a programozók ezen a részen tároltak el tetszőleges adatokat. *Work RAM*-nek is hívták emiatt. Ez a terület szorosan összefügg az 0xE000 - 0xFFFF memória résszel: a második, ún. *Shadow RAM* (Árnyék RAM) terület a *Work RAM* bitre pontos másolata. Érdekesség, hogy a Nintendo a hivatalos programozóknak szánt dokumentációjában megtiltja, hogy

a *Shadow RAM* területére írás történjen, ha ezt mégis megtesszük, akkor a *Work RAM* azonos bájtja is átíródik ugyanarra a bájtra, amit a *Shadow RAM*-ba írtunk. Szintén érdekes, hogy ha megnézzük, a két RAM területe nem egyezik meg, a *Shadow RAM* 128 bájttal "rövidebb", mint a *Work RAM*. Ennek az az oka, hogy a RAM legfelső 128 bájta a leggyorsabban írható és olvasható *High RAM* számára van fenntartva.

Az utolsó egység tehát a memóriában a *High RAM*, amely az 0xFE00 - 0xFFFF területet foglalja el. Itt található nagyon sok fontos adat, regiszter és *flag*. Az 0xFE00 - 0xFE9F területen az OAM, vagy *Sprite Attribute Table* található. A Game Boy képernyőjén megjelenő mozgó objektumok az itt eltárolt *sprite*-okból épülnek fel.

A kijelző, hanggenerátor, soros port, *joypad*, és az időzítők fontos attribútumait és működési elemeit tartalmazza az 0xFF00 - 0xFF7F szakasz. A diplomamunka további részében a megfelelő helyeken ezekről bővebben lesz szó.

Az 0xFF80 - 0xFFFF részt a *High RAM* programozók által saját célra használható (gyors elérésű) része foglalja el.

A 0xFFFF címen lévő bájt a korábban már említett *Interrupt Enable* regiszter.

### 3.4.1. DMA

A DMA vagy *Direct Memory Access* (Közvetlen Memóriaelérés) a Game Boy architektúrájának egy olyan eljárása, amely segítségével adatot lehet másolni a fentebb említett *Sprite Attribute Table*-be. Egy játék akkor tudja alkalmazni a DMA eljárást, ha az 0xFF46-es memóriacímre ír. Ahhoz, hogy ezt a működést emulálni tudjuk, készíteni kell egy csapdát a memória írást megvalósító függvénybe:

```
pub fn write_ram(&mut self, address : u16, value : u8) {
    // ...
    // if writing address is 0xFF46 -> DMA
    if address == 0xFF46 {
        self.dma(value);
    }
    // ...
}
```

A 0xFF46-os címre kérelmezett írás esetén tehát maga az írás történik meg, hanem a vezérlés a fenti csapdába kerül. A DMA-t aktiváló memóriacímre írandó adat adódik át a DMA függvénynek.

A *Sprite Attribute Table* a 0xFE00 - 0xFE9F memóriacímek között helyezkedik el, tehát 160 bájtot tárol. Ezt a 160 bájtot kell feltölteni az új adattal úgy, hogy a DMA függvény paraméterében átadott érték alapján kiszámoljuk a kezdőcímét, ahonnan a 160 darab bájt másolása fog megtörténni. Ennek a kezdőcímnek a kiszámolása könnyű feladat: el kell osztani 100-zal, vagy – ha gyorsabb véghajtást szeretnénk –, *shifteljük* el balra 8-cal. A DMA függvény tehát a következők szerint alakul:

```
pub fn dma(&mut self, value : u8) {
    // copyable data start address
    let addr : u16 = (value as u16) << 8;

    // start copying
    for i in 0..0xA0 {
        let n : u8 = self.RAM[(addr + i) as usize];
        self.write_ram(0xFE00 + i, n);
    }
}
```

Ezzel a szükséges bájtok másolása megtörtént.

### 3.4.2. Memory Bank Controller

Az alfejezet elején bemutatott memória térkép modellje állandó, hiszen olyan hardverről beszélünk, amely esetében nincs lehetőség a memória, vagy egyéb modul bővítésére. A tárgyalt modellben a játék ROM-ok tárolására 32 kB hely van fenntartva – ezzel pedig szöges ellentmondásban van az a tény, hogy több játék mérete meghaladta ezt a korlátot: a Pokémon játék például megközelítőleg 2 MB méretű. Említeni sem kell, hogy itt már nagyságrendbeli eltérésről van szó, tehát felmerül a kérdés, hogy *hogyan működnek a 32 kB-nál nagyobb méretű játékok a Game Boyon?*

Mielőtt azonban a feltett kérdésre adott választ taglalnám, fontos bemutatni, hogy néz ki a *Cartridge headerje* (fejléce). A ROM-ok ezen részében standardizált formában kapunk információkat az adott játék technikai, és általános jellemzőiről.

Memóriacím	Adat	Méret	Részletek
0x100 – 0x103	Belépési pont	4 Byte	Általában: NOP JP 0x150.
0x104 – 0x133	Nintendo logó	48 Byte	Másolásvédelmi célokra.
0x134 – 0x143	Cím	16 Byte	Nagybetűkkel, 0-kal közölve.
0x144 – 0x145	Kiadó	2 Byte	Az újabb játékoknál használatos.
0x146	SGB flag	1 Byte	SGB támogatás.
0x147	Cartridge típusa	1 Byte	MBC típusa és egyéb extrák.
0x148	ROM méret	1 Byte	Méret = 32 KB << [0x148]
0x149	RAM méret	1 Byte	A külső RAM mérete.
0x14A	Forgalmazás	1 Byte	Japán piac: 0, egyébként 1.
0x14B	Kiadó	1 Byte	A régebbi játékoknál használatos.
0x14C	ROM verzió	1 Byte	A játék verziószáma, általában 0.
0x14D	Fejléc összeg	1 Byte	A betöltés előtt leellenőrizendő.
0x14E – 0x14F	Globális összeg	2 Byte	Egyeszerű összegzés.
0x150h			A játék kódjának kezdete.

3.4. táblázat. A ROM-ok fejlécének adatai

A 3.4-es táblázatban szereplő adatokból látható, hogy a 0x147-es memóriacímen van eltárolva a *Cartridge* típusa, amely tulajdonképpen a Memory Bank Controller típusára utal. A fenti kérdésre a választ közvetetten ez az adat adja meg. A Game Boy megjelenését

követően felmerültek olyan igények, amelyek a nagyobb RAM-ot, illetve nagyobb elérhető ROM tárhelyet helyezték előtérbe. Ezekre a problémákra találták ki a Memory Bank Controller technológiát, amellyel az eredeti hardver módosítása nélkül lehetett bővíteni az előbb megnevezett tárhelyeket.

A bővítés egyszerű mechanizmuson alapul: a ROM adatait *Bank*-okra osztjuk fel, majd ezeket rakjuk be, vagy vesszük ki a ROM-ba. Ugyan ezzel a módszerrel lehetséges a RAM területének bővítése is.

A Game Boy 32KB méretű ROM-jának első 16 kB-ja le van fixálva: itt helyezkedik el a 0-s azonosítójú *bank*. A ROM maradék 16 kB-jába lehet betölteni a kívánt *bankot*. A *bankok* számát az határozza meg, hogy az MBC ROM módban, vagy RAM módban van-e. ROM módban egyáltalán nem elérhetők a RAM *bankok*, cserébe viszont 2 MB-nyi ROM-unk lehet, ez 128 darab 16 kB-os *bankot* jelent. Ami a RAM módot illeti, ebben az esetben 4 darab RAM, és 32 darab 16 kB-os ROM *bankunk* lehet.

A ROM és RAM *bankok* ki-, illetve bekapcsolását, illetve betöltését és kivételét úgy oldották meg, hogy előre meghatározott *magic number*-öket kell írni előre meghatározott memóriaterületekre – így váltak elérhetővé a kívánt *bankok*. Az MBC működésének részletesebb taglalására nem fog sor kerülni, mert az implementált emulátor ezt nem tartalmazza – az MBC nélküli alap rendszer is képes játékokat futtatni, így az idő szűke miatt a Memory Bank Controller implementálására már nem maradt idő.

## 3.5. Időzítők

A Game Boy hardver architektúrájának elengedhetetlen részét képezik az időzítők. Ahhoz hogy a rendszer egyes elemei között meglegyen a tökéletes együttműködés, szinkronizált végrehajtásra van szükség. Erre jelentenek megoldást a *timerek* (időzítők), amelyek segítségével meg lehet teremteni a modulok közti kohéziót.

Két típusú *timer* áll rendelkezésre az architektúrában: a **TIMA** (*Timer Counter*), illetve a **DIV** (*Divider Register*). Ezek alapjaikban ugyan megegyeznek, és így egymással összehasonlíthatók, viszont különböző feladatok ellátására szolgálnak, ebből kifolyólag pedig a működésük végső soron mégis eltérő – ezt a továbbiakban látni fogjuk. Ami az implementálást illeti, minden időzítő esetén a processzor minden műveletet követően leellenőrzi hogy szükséges-e léptetni az időzítőt. A részletesebb bemutatást a **TIMA**-val kezdjük, majd pedig a **DIV** fog következni.

### 3.5.1. TIMA

A *Timer Counterhez*, vagy **TIMA**-hoz három memóriacím, azaz három folyton változó adat kapcsolható. Ezeket a következő táblázatban gyűjtöttem össze:

TIMA	0xFF05
TMA	0xFF06
TMC	0xFF07

3.5. táblázat. A *TIMA* időzítőhöz köthető adatok memóriacímei

Ahogy a 3.5-ös táblázatból kiolvasható, a három érték maga a **TIMA**, a **TMA** és a **TMC**.

Az időzítő leegyszerűsített működése könnyen megérthető és implementálható. Ha elég idő telt el ahhoz hogy léptessük az időzítőt, akkor megtesszük ezt a léptetést. A léptetés során azonban figyelni kell arra, hogy túlcordulás történik-e: ha az időzítő új értéke nagyobb mint 255 (egy byte) akkor 2-es számú megszakítási kérést kell küldeni a processzornak, és be kell olvasni az időzítő új, bázis értékét – ez az az érték, ahonnan folytatódni fog a számlálás. Ha nincs túlcordulás, akkor a léptetésen kívül nincs további teendő.

Persze felmerülhet több kérdés is. *Honnan olvassuk ki az új bázis értékét? Honnan tudjuk hogy milyen gyakorisággal kell léptetni?* A fenti 3.5-ös táblázatban szereplő adatok fogják megadni ezekre a kérdésekre a válaszokat.

A TMA, vagy *Timer Modulo* az az érték, amelyet a fentebb taglalt időzítő eljárás során is használtunk: ebből olvassuk ki az időzítő túlcordulás utáni új bázis értékét, melyet abba a TIMA regiszterbe töltünk, amely az időzítő aktuális állapotát tartalmazza, és amelyet folyton inkrementálunk. Azzal, hogy nem egyszerűen minden 0-ról kezdjük újra a számozást, modulálni lehet a Game Boy időzítőjét.

A másik kérdésre a TMC, azaz a *Timer Control* fogja megadni a választ. Ez a byte tartalmazza az időzítő frekvenciáját, illetve segítségével ki-be lehet kapcsolni a *timer* működését. Ez utóbbi funkció 1 bittel (a 3.6-os ábrán **T**), míg az aktuális frekvencia sebessége 2 bittel van ábrázolva (a 3.6-os ábrán **F**) – így a TMC egy három bites regiszter. A bitek kiosztása a következők szerint alakul:

0	0	0	0	0	<b>T</b>	<b>F</b>	<b>F</b>
---	---	---	---	---	----------	----------	----------

3.6. táblázat. A *TMC* regiszter bitjeinek kiosztása

A fenti táblázat szerinti 0. és 1. bitek tárolják el tehát az időzítő frekvenciáját. Az egyes frekvenciák a következő táblázatban ábrázolt értékekkel vannak jelölve.

<b>FF</b>	<b>Frequency</b>
00	4096 Hz
01	262144 Hz
10	65536 Hz
11	16384 Hz

3.7. táblázat. A választható frekvencia opciók és az ábrázolási értékek

Tudjuk, hogy a CPU frekvenciája 4194304 Hz, így könnyen kiszámolható az, hogy hány processzor művelet után kell léptetni az időzítőt az egyes időzítő frekvenciák esetében – egyszerűen el kell osztani őket egymással. Ez az implementációt nagyban megkönnyíti, hiszen a fő ciklus minden iterációja egy órajelciklussal ekvivalens. Az órajelciklusokra átszámlolt frakvenciák tehát a fenti táblázat szerint rendre: 1024, 16, 64 és 256. Látható, hogy ezek igen kerek értékek.

A TIMA időzítő implementálása tehát minden összevetve így alakul:

```
// Is the clock enabled?
if CPU::get_bit(2, cpu.RAM[self.TMC as usize]) {
    // Is enough time passed to update the timer?
```

```

if cycle as u32 > self.timer_counter {
    // Update the current frequency.
    self.update_freq(cpu);
    // Check for the overflow.
    if cpu.RAM[self.TIMA as usize] >= 255 {
        // If an overflow occurs, load TMA to TIMA,
        let TMA = cpu.RAM[self.TMA as usize];
        cpu.write_ram(self.TIMA, TMA);
        // and send an interrupt request.
        self.interrupt.IRQ(cpu, 2);
    } else {
        // If no overflow, increment the TIMA.
        let TIMA = cpu.RAM[self.TIMA as usize];
        cpu.write_ram(self.TIMA, TIMA + 1);
    }
} else {
    // Sync the time with the CPU.
    self.timer_counter -= cycle as u32;
}
}

```

A fenti kódrészletet tartalmazó, Timer struktúra beli Update függvényt kell meghívni a fő ciklus minden iterációjában.

### 3.5.2. DIV

A DIV időzítő, ahogy már említésre került, úgy működésben, mint viselkedésben eltér a TIMA-tól. A DIV minden tekintetben sokkal egyszerűbb, mint a társa: nem lehet változtatni a frekvenciáját, és modulálni sem lehet.

A frekvencia fixen 16384 Hz, ami azt jelenti, hogy minden 256 CPU ciklus után kell inkrementálni az időzítő értékét. A moduláció hiánya pedig abból fakad, hogy túlcsordulás esetén nem egy másik memóriahezről töltünk értéket a DIV-be, hanem alapértelmezés szerint 0-ról kezdjük ismételten a számlálást. A TIMA-tól eltérően ennek az időzítőnek nincsen egyéb adata, egyedül a *counter* értéke adott.

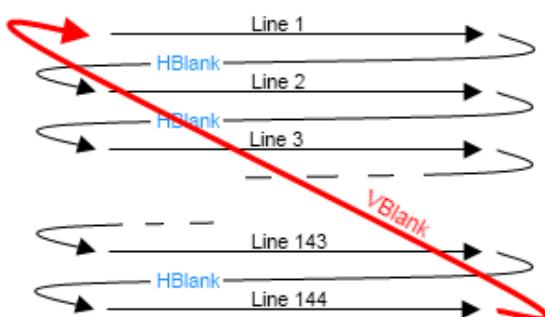
## 4. fejezet

# A kijelző és a PPU implementációja

A processzor és memória modulok után következik a Game Boy emulátor legnehezebb moduljának, a PPU-nak a fejlesztése. Ennek bonyolultsága abban rejlik, hogy rendkívül sok attribútummal, és egyéb jellemzőkkel, mini-mechanizmusokkal rendelkezik, amelyek között meg kell teremteni az együttműködést és kohéziót. Ezeket természetesen egyenként, pontosan kell megvalósítani annak érdekében, hogy az emuláció és ezzel együtt a megjelenítés is pontos legyen. Természetesen ez elvárható, hiszen ez azon kevés modulok közé tartozik, amelyek közvetlenül tartják a kapcsolatot a felhasználóval, így minden apró eltérés szembeötlő lehet.

A *Pixel Feldolgozó Egység (Pixel Processing Unit)* implementációja előtt azonban annak "keretrendszerét" kell megvalósítani, az LCD kijelző működését, állapotainak reprezentációját. Ezek nélkül a PPU nem tudna működni, hiszen a kijelző státusz állapotainak megfelelően történik a kirajzolás.

### 4.1. Az LCD kijelző



4.1. ábra. A letapogató és az üresjáratok

A Game Boy kijelzőjéről a hardver specifikáció ismertetése során már volt szó – ahogy ott is megemlítésre került, a szóban forgó LCD kijelző  $166 \times 144$  pixel felbontású, 4 árnyalat megjelenítésére képes. A képernyőre renderelt képpontokat a minifb könyvtárral fogjuk megjeleníteni, amelynek struktúráit, és alapvető funkcióit az 1.2.2.-es rész taglalja. Ahhoz azonban, hogy a renderelést megfelelő módon el tudjuk végezni, oda kell figyelni az LCD kijelző állapotaira, megszakításokra, egyéb jellemzőkre.

A konzol hardverének tervezése során a mérnökök több képernyőfrissítési módszert és megoldást a – már akkor korosnak számító – CRT technológiából vettek kölcsön. Mint az ismeretes, a katódsugárcsöves megjelenítők esetében a képernyőt sorról sorra pásztázta végig egy elektronnyaláb, majd az utolsó sor végeztével a műveletet újból a legelső sorral folytatta. Azonban – ahogyan a 4.1-es ábrán is megfigyelhető – a CRT-nek

több időre van szüksége ahhoz, hogy kirajzolja a megfelelő képpontokat a megfelelő helyre, nem csak végigszalad rajtuk. Idő kell tehát ahhoz, hogy az egyes sorok végéről a sugárnyaláb átvándoroljon a következő sor elejére, ahogy az is időbe kerül, hogy a nyaláb a képernyő utolsó pixeléről (jobb alsó sarok) visszamenjen a legelsőhöz (bal felső sarok). Természetesen ez utóbbi sokkal tövább fog tartani mint az előbbi – ezeket egyébként *horizontális üresjáratnak* (vagy *HBlank-nak*) illetve *vertikális üresjáratnak* (vagy *VBlank-nak*) nevezzük.

Ezek az üresjáratok nagyban meghatározzák a PPU emulációjának ritmusát, több dolgot is szűk intervallumon belül kell elvégezni. Az LCD kijelző által meghatározott intervallumokat, illetve pontos szinkronizációs időpontokat az alábbi táblázat tartalmazza.

Esemény	PPU mód azonosító	Igénybevett idő
Scanline (OAM elérése)	2	80 órajelciklus
Scanline (VRAM elérése)	3	172 órajelciklus
Horizontális üresjárat	0	204 órajelciklus
Egy sor kirajzolása		<b>456 órajelciklus</b>
Vertikális üresjárat	1	4560 órajelciklus
Képkocka (letapogatások és üresjáratok)		70224 órajelciklus

4.1. táblázat. A PPU időzítése

Ez alapján tehát rekonstruálni tudjuk az LCD kijelző és a PPU időzítéseinek működését, és azt, hogy egymással és a processzorral hogyan tudjuk szinkronban tartani őket. Ehhez szükség lesz egy, a CPU fő ciklusában minden alkalommal meghívott valamelyen aktualizáló függvényre, ahol ellenőrzés alatt lehet tartani a CPU és PPU együttműködését, szinkronizációját. Ez a függvény az `update_n_sync` lesz, melynek tartalma a következő:

```

self.lcd_status_update(cpu);
self.scanline_count += cycle as u16;

if self.scanline_count >= 456 {
    self.scanline_count = 0;

    if cpu.RAM[0xFF44] < 144 {
        self.draw_line(cpu);
    }

    if cpu.RAM[0xFF44] == 144 {
        self.interrupt.IRQ(cpu, 0);
    }

    cpu.RAM[0xFF44] += 1;

    if cpu.RAM[0xFF44] > 153 {
        cpu.RAM[0xFF44] = 0;
    }
}

```

A fenti programrészlet tehát menedzseli a renderelést: a megfelelő időben meghívja a megfelelő függvényeket, miközben frissíti a megfelelő változókat. Kicsit bővebben ez annyit tesz, hogy először is meghívja az `lcd_status_update` függvényt (később bemutatásra kerül), majd hozzáadja az utolsó CPU operáció műveletidejét egy számlálóhoz (`scanline_count`), amellyel számon tudjuk tartani, hogy mennyi művelet elvégzése szükséges a következő pixelsor kirendereléséig. Ha ez a számláló nagyobb egyenlő, mint 456, akkor további ellenőrzéseket végünk.

Először is lenullazzuk a számláló változót, majd megnézzük, hogy a `0xFF44` memóriacímen található érték elérte-e a 144-es értéket. Ha nem, akkor rajzoljuk ki a sort. Az előbb említett `0xFF44`-es címen található érték azt mutatja, hogy a PPU éppen melyik során jár a kirajzolásban, ezért hasonlítjuk össze 144-gyel: pontosan ennyi sora van a kijelzőnek.

Ha a `0xFF44`-es érték megegyezik 144-gyel, tehát az utolsó sor is ki lett rajzolva a képernyőre, 0-s azonosítójú megszakítási kérést kell küldeni a processzornak, ez lényegében a `VBlank` művelet elvégzését takarja. Ezeket az elágazásokat követően növelni kell a `0xFF44`-en található értéket (`scanline`, letapogató) eggyel, hiszen a következő sorra lépünk.

A következő, végső `if` elágazás igen érdekes: a `scanline` értékét vizsgálja, azt nézi meg, hogy nagyobb-e mint 153. *De miért pont 153?* Egész pontosan azért, mert a 5.1-es táblázatban ismertetettek szerint a `VBlank` művelet végrehajtási ideje pontosan 10-szerese az egy sor kirajzolásáénak. A képernyőn lévő sorok számát 0-tól indexeljük, így lesz a 144-ből 143, amihez ha hozzádunk 10-et, amíg a `VBlank` tart, megkapjuk a 153-as számot. Amint elérünk a 153. sort (vagyis befejeződött a `VBlank` művelet), kezdhetjük a következő képkocka renderelését a 0. sortól.

#### 4.1.1. Az LCD státuszok

A következőkben a fent említett `lcd_status_update` által elvégzett feladatokat, azaz az LCD kijelző státuszait és az azokra való reagálást fogom bemutatni.

Az kijelző aktuális állapotát a `0xFF41` memóriacímen található LCD Státusz Regiszter tárolja a 0. és 1. biteken – ebből következik hogy  $4 (2^2)$  ilyen állapot létezik:

- 00: *HBlank*
- 01: *VBlank*
- 10: *Sprite* adatok keresése a memóriában (OAM)
- 11: Adatok másolása a PPU számára a VRAM-ba

Mihelyst a `scanline` újra kezdi a képkocka kirajzolását az első sortól, a következő műveletek ciklusát hajtja végre újra és újra: először is 10-ra állítódik az állapot, majd 11-re, végül pedig 00-ra. Amikor elér a `VBlank` művelethez, a státusz 01 lesz egészen az üresjárat végéig. Ezt követően ismét az 10 állapotot veszi fel, és folytatódik a ciklus a fentiek szerint. Az egyes állapotokhoz tartozó végrehajtási időket tartalmazza a 5.1-es táblázat.

#### 4.1.2. Az LCD interruptjai és flagje

Abban a pillanatban, amikor az LCD kijelző állapota megváltozik, és a 00, 01 vagy 10 módba kerül (a fent ismertetetteknek megfelelően), interrupt hívás következhet be. Az LCD Státusz Regiszter 3., 4. és 5. bitjei felelnek az előbb említett 3 mód megszakításainak engedélyezéséért vagy letiltásáért (a 0xFFFF-en található *Interrupt Enable* regiszterhez hasonlóan), viszont ezeket nem az emulátor, hanem a játék fejlesztői állíthatták be az igényeknek megfelelően. A megszakítások LCD módokra való megfeleltetése a következők szerint alakul:

- 3. bit: 00-s LCD mód megszakításának engedélyezése, tiltása,
- 4. bit: 01-s LCD mód megszakításának engedélyezése, tiltása,
- 5. bit: 10-s LCD mód megszakításának engedélyezése, tiltása.

Fontos megjegyezni, hogy az LCD módok közül feltétlenük implementálni kell a kijelző kikapcsolásakor a 01-es módra való váltást, mert ha ezt nem tesszük meg, több játék is meg fog akadni a megszakítás elvégzésének hiánya miatt.

Ötletes megoldást tesz lehetővé az LCD Státusz Regiszter 2. és 6. bitje. A második bitet *Coincidence Flag*nek nevezzük, és akkor állítjuk 1-re, ha a *scanline* aktuális sor értéke (0xFF44) megegyezik az 0xFF45-ös memóriacímen található értékkel. A hatodik bit azt szabályozza, hogy ha a *Coincidence Flag* be van állítva, akkor legyen-e megszakítás kérés. Ennek a megoldásnak a segítségével tudtak a játékfejlesztők különféle speciális effekteket leprogramozni, hiszen a *Coincidence Flag* segítségével lehetett vizsgálni és interruptot kérni egy konkrét sor esetén.

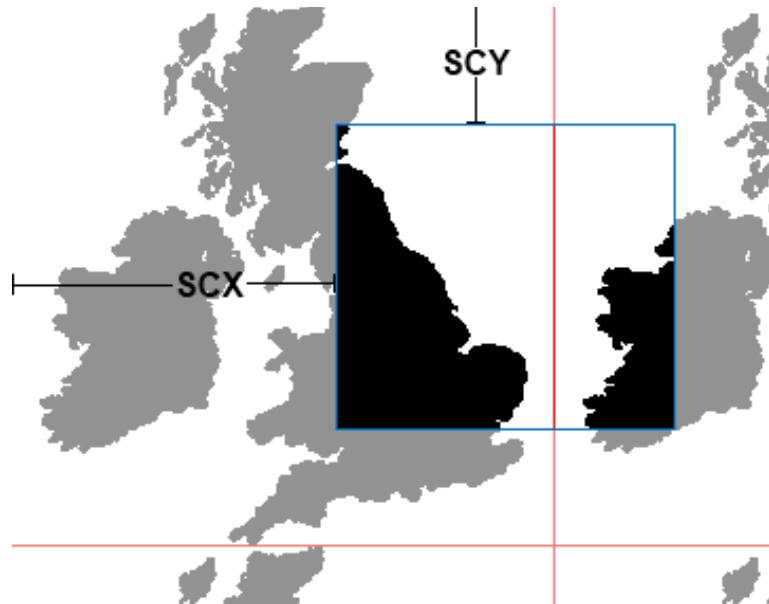
#### 4.2. PPU - Pixel Feldolgozó Egység

Az előző alfejezetben ismertetésre került a Game Boy kijelzőjének viselkedése, az ehhez felhasznált eszközök, módszerek, és a keretrendszer ami emögött működik. Az időzítés, megszakítások, módok és egyebek azonban önmagukban nem elegek ahhoz hogy az emulátor kirenderelhesse a képet a *framebuffer* ablakba, ehhez további vizsgálatok, és implementáció szükséges. A sorrend a következő: először is a megjelenítéshez szükséges alapvető ismeretekről lesz szó, majd a *tile*-ok, vagy csempék renderelésén keresztül fogom bemutatni azokat a módszereket amikre figyelni kell. A *sprite* renderelést tartalmazó rész a *tile* rendereléshez viszonyított különbségeket fogja tartalmazni.

##### 4.2.1. Alapvető tudnivalók

A Game Boy a grafikák megjelenítéséhez *tile*-okat és *sprite*-okat használ, ezek segítségével állítja össze a kívánt mintát a rendereléshez. Itt mindenki két definícióval is találkozunk, ezek a következőket jelentik: a *tile*-ok a játék háttérét alkotó, leginkább ismétlődő mintákat ábrázoló csempék, amelyek statikusak, a játéknak a kontextust adják meg, míg a *sprite*-ok a mozgó, folyton helyet változtató, villogó részek a képernyőn, illetve az irányítható karakterek. Mindkettejükkel elmondható, hogy általában 8×8 pixel méretűek, viszont

előfordulhatnak nagyobb,  $8 \times 16$  pixelesek is, melyek többnyire a játszható karaktereket ábrázolták (pl. Mario).



4.2. ábra. A képernyő scrollozásos megoldása – az SCX az X koordinátát, az SCY pedig az Y koordinátát tartalmazó regiszter

Az már többször is említésre került, hogy a hardver kijelzőjének felbontása  $160 \times 144$  pixel méretű, viszont virtuális tekintetben  $256 \times 256$  pixel a rajzolható terület mérete, amely ekvivalens  $32 \times 32$  darab  $8 \times 8$  méretű csempével. Ebből a nagyobb, 1:1 arányú méretből aztán úgy lesz a már sokat emlegetett  $160 \times 144$ , hogy egy pontosan ekkora méretű részt mozgatunk a  $256 \times 256$  méretű területen. A mozgatott "ablak" koordinátái külön el vannak tárolva egy regiszterben, ezt a játékfejlesztők szabadon használhatták. A fenti, 4.2-es ábra mutatja a fent ismertetett *scroll* mechanizmust.

Fontos még megemlíteni, hogy létezik egy harmadik típusú ábrázolási mód, ez pedig a *window* (ablak). Ennek segítségével a megjelenített pixelek a háttér *tile*-ok előtt, viszont a *sprite*-ok mögött foglalnak helyet. A programozók ezt a megjelenítési módot használhatták az alkalmazásaiak felhasználói felületeinek *HUD*<sup>1</sup>-jaként, melyekre tetszőleges ábrákat, szöveget helyezhetek el.

#### 4.2.2. Az LCD Control Register

Az előző alfejezetben említett LCD Státusz Regiszterhez hasonló, szintén az LCD kijelző állapotait és a megjelenítést segítő regiszter az *LCD Control Register*, amelynek memóriacíme a `0xFF40` és amely inkább a rendereléshez áll közelebb: számos, a PPU által aktívan használt paramétert tartalmaz. Ezek rendre a következők:

- **7. bit:** Ezzel a bittel lehet állítani, hogy az LCD kijelző ki-, vagy be legyen-e

<sup>1</sup> *Heads-Up Display*: más néven *status bar*, általában a játékos állapotának, illetve a játékban elért pontszámának közlésére szolgáló felület, az UI szerves része.

kapcsolva. Amennyiben nincs bekapcsolva (értéke 0), nem rajzolunk semmit sem. Ezt az előző, LCD kijelző funkciót implementáló kódrész kezeli le.

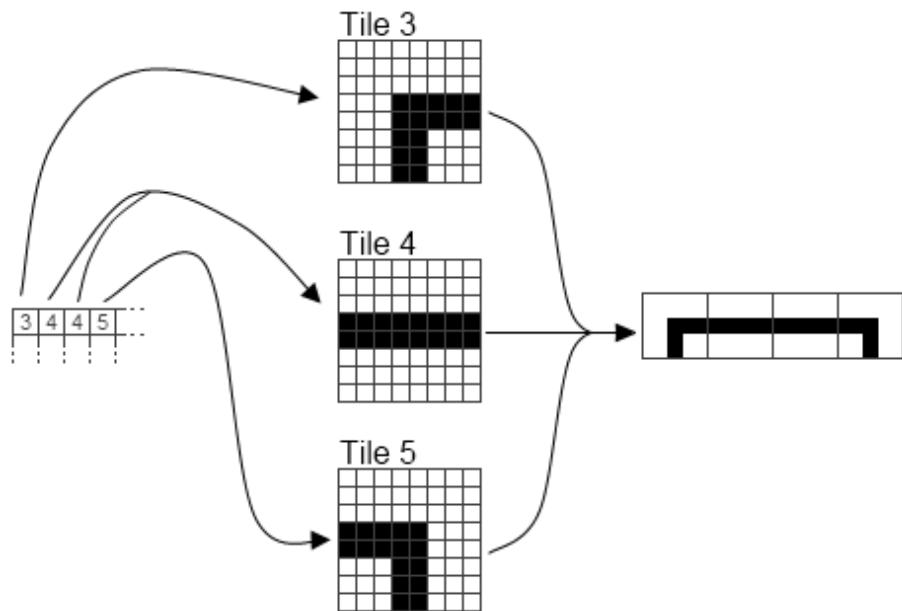
- **6. bit:** Ez a bit azt mondja meg, hogy a játék a memória mely részében keresse a *window* kirajzolásához szükséges *tile*-ok bájtjait. Ezek a bájtok az adott csempék azonosítót tárolják, minden csempéhez pontosan egy azonosító tartozik. Amennyiben az értéke 0, úgy az intervallum  $0x9800 - 0x9BFF$ , egyébként pedig  $0x9C00 - 0x9FFF$ .
- **5. bit:** Ez a bit szolgál a megjelenített *frame*-en belüli *window*-k ki-, illetve bekapsolására. 0 érték esetén nem rajzoljuk ki ezeket, egyébként igen.
- **4. bit:** Ezzel a bittel lehet állítani, hogy a háttérként, illetve *window*-ként kirajzolt *tile*-ok bájtjai merre találhatóak. Ezek a bájtok a konkrét csempéket tárolják. 0 érték esetén a *tile*-ok helye a memóriában  $0x8800 - 0x97FF$ , 1-es érték esetén pedig  $0x9C00 - 0x9FFF$ . Érdekesség, hogy amennyiben az 0-s érték által reprezentált módon vagyunk, úgy a 6-os bitnél említett azonosító bájtok előjeles bájtok, érték-készletük pedig ennek megfelelően a  $[-128, 128]$ -as intervallum.
- **3. bit:** Ez a bit azt mondja meg, hogy a játék a memória mely részében keresse a háttér kirajzolásához szükséges *tile*-ok bájtjait. Ezek a bájtok is az adott csempék azonosítót tárolják, a 6-os bithez hasonló módon. Ha az értéke 0, akkor az intervallum  $0x9800 - 0x9BFF$ , egyébként pedig  $0x9C00 - 0x9FFF$ .
- **2. bit:** Ezzel a bittel azt lehet megadni, hogy a kirajzolandó *sprite*-ok mérete a már említettek szerint  $8 \times 8$ -as legyen (0 értékű bit esetén), vagy pedig  $8 \times 16$ -os (1-es értékű bit esetén).
- **1. bit:** Ez a bit az 5-ös bithez hasonló módon azt tudja szabályozni, hogy kirajzoljuk-e a vonatkozó adatokat. A konkrét adatok itt természetesen nem a *window*-hoz fognak tartozni, hanem a *sprite*-okhoz: 0-s érték esetén ezek renderelését lehet letiltani, illetve 1-es érték esetén engedélyezni.
- **0. bit:** Ez a bit előzőhöz, és az 5-ös bithez hasonlóan a kirajzolandó adatokat korlátozza: ebben az esetben a hátteret alkotó *tile*-ok renderelését lehet letiltani 0-s értékkel, vagy engedélyezni 1-essel.

Most, hogy ismerjük a fenti, LCD Control Register által tárolt beállításokat, és el lehet kezdeni implementálni a hátteret alkotó *tile*-ok kirajzolását végző eljárás(okat). Előtte azonban készíthetünk egy, az adott sor renderelését menedzselő, egyszerű függvényt:

```
fn draw_line(&mut self, cpu : &mut CPU) {
    // Check if the tile rendering
    // is whether on or off
    if CPU::get_bit(0, cpu.RAM[0xFF40]) {
        self.draw_tile(cpu);
    }
}
```

```
// Check if the sprite rendering
// is whether on or off
if CPU::get_bit(1, cpu.RAM[0xFF40]) {
    self.draw_sprite(cpu);
}
}
```

#### 4.2.3. Tile rendering



4.3. ábra. A tile-ok segítségével összeállított háttér

Az előzőek alapján már kaptunk némi ismeretet a PPU és az LCD működéséről, a fontosabb regiszterekről és renderelési módokról, ideje tehát rátérni a konkrét *tile renderingre*. Az első fontos kérdés ami felmerülhet, hogy *az említésre került  $256 \times 256$  méretű virtuális területnek pontosan melyik  $160 \times 144$  nagyságú részletét jelenítsük meg a képernyőn?* A korábbi, 4.2-es ábrán, és az ahhoz tartozó leírásban már volt erre vonatkozó utalás: a *hasznos* területet a *ScrollX* és *ScrollY* regiszterek segítségével kaphatjuk meg. A rendre *0xFF42* és *0xFF43*-as címeken elérhető, X és Y koordinátákat tároló bajtok mutatják az ominózus  $160 \times 144$ -es renderelendő terület bal felső sarkának távolságát az origótól (a teljes,  $256 \times 256$  méretű terület bal felső sarka). Amennyiben *window* renderelésére is szükség van, annak megjelenítési helyét hasonló módon, két regiszter segítségével (*WindowX*: *0xFF4B*, *WindowY*: *0xFF4A*) tehetjük meg, viszont itt a látható,  $160 \times 144$ -es terület bal felső sarka a kiindulópont, origó.

#### A tile adatainak meghatározása, algoritmus

Most, hogy megvannak a regiszterek, amelyek segítségével kiszámítható, hogy a virtuális terület mely része lesz konkrétan látható, következhet a renderelés következő szakasza. Az LCD Control Registerből ki kell olvasni a fentebb említett, megfelelő biteket, és ezek

alapján be kell állítani a *tile* azonosítók, és a konkrét *tile* leírók memóriában elhelyezkedő kezdőcímét. A biteket megvizsgálva ez egyszerű elágazásokkal megtehető. Ezt követően meg kell határozni a virtuális, teljes felbontáson értelmezett pozíciót, és a *tile* elhelyezkedését (sorát) a  $32 \times 32$ -es csempetérképen a következő módon:

```
if !using_window {
    y_pos = self.scroll_y.wrapping_add(cpu.RAM[0xFF44]);
} else {
    y_pos = cpu.RAM[0xFF44] - self.window_y;
}

let tile_row : u16 = (y_pos as u16 / 8) * 32;
```

Ezek az adatok a *tile*-ok kirajzolásához szükségesek lesznek. A következő lépés a nagy, 160 iterációval járó ciklus – ez a szám nem véletlen, minden egyes kirenderelt sorra egy iteráció jut. Ennek a ciklusnak a lépéseinél pszeudokód segítségével, rövidebb formában fogjuk végigkövetni, a bonyolultabb részek implementációjára fókuszálva, a triviálisakat kihagyva. A ciklus pszeudokódja:

1.  $i=1-től 160-ig$  egyesével:
  - 1.1 Számoljuk ki az X koordinátát.
  - 1.2 Számoljuk ki a tile oszlopát és határozzuk meg a tile memóriacímét.
  - 1.3 A cím segítségével másoljuk ki a memóriából a tile azonosítóját.
  - 1.4 Az azonosító segítségével határozzuk meg a tile pontos helyét a memóriában.
  - 1.5 A kurrens sor számának segítségével határozzuk meg a tile Y pozícióját.
  - 1.6 Az Y pozíció és memória hely segítségével másoljuk ki a tile konkrét sorához tartozó két bájtot.
  - 1.7 Vegyük ki a két bájt által tárolt adatokból a kurrens pixel színének értékét.
  - 1.8 Határozzuk meg a színpalletta alapján hogy az adott szín pontosan milyen.
  - 1.9 Írjuk a framebuffer megfelelő helyére a megfelelő szín értékét, majd GOTO 1.

Az X koordinátát egyszerűen a ciklusváltozót a `scroll_x`-hez hozzáadva megkaphatjuk. Ennek segítségével pedig kiszámolható a tile oszlopa: egyszerűen el kell osztani 8-cal (mert egy tile mérete  $8 \times 8$ ). A memóriacím meghatározása úgy történik, hogy a kezdőcímhez hozzáadjuk a már fent kiszámolt `tile_row`-t és az imént kiszámolt tile oszlopát (`tile_col`).

A csempe pontos helyének meghatározása a következők szerint történik:

```
if unsigned_data {
    tile_location += tile_num_u as u16 * 16;
} else {
    tile_location += ((tile_num_i as i16 + 128)*16) as u16;
}
```

Itt `tile_num_u` a csempe azonosítója `unsigned` formában, a `tile_num_i` pedig `signed`-ként. A 16-tal való szorzás azért történik, mert egy *tile* 8 soros, és minden sorhoz tartozik két leíró bájt. Az elágazás `else` ága azt az esetet taglalja, mikor a kiolasott azonosító előjeles típusú. Ebben az esetben egy *offset* értéket (128) kell adni az azonosítóhoz, mielőtt felhasználnánk.

Az aktuális sorhoz tartozó két bájtot immár ki tudjuk olvasni a memóriából a következő módon:

```
let mut line : u8 = y_pos % 8;
line *= 2;

let d1 : u8 = cpu.RAM[(tile_location
    + line as u16) as usize];
let d2 : u8 = cpu.RAM[(tile_location
    + line as u16 + 1) as usize];
```

A `line` segítségével megkapjuk a  $32 \times 32$ -es csempetérkép-beli sorszámot (ezért a 8-cal való osztás), majd ezt kettővel felszorozzuk, mert – ahogy már említésre került – két bájt tartozik egy sorhoz. Ezt a változót hozzáadva az előbb kiszámított memóriacímhez megkapjuk az első bájt címét, ahhoz egyet adva pedig a másodikét.

A következő feladat a fentihez hasonló módon szintén egy modulo 8-as művelet – de ez esetben az X koordinátán, hogy megtudjuk hogy a csempe aktuális során belül melyik pixelt keressük. Itt viszont egy apró trükkre van szükségünk: a csempe sorát leíró két bájt fordítva tárolja a szín értékeit, így "inverzet" kell vennünk (7-ből 0 lesz, 6-ból 1, stb.). Amennyiben ezzel megvagyunk, már csak a leíró bájtok által meghatározott színt kell visszakeresni a palettának megfelelően, majd pedig ezt az értéket a `framebuffer` memóriába (VRAM) kell írni.

## A színek és a paletta

A fentiekben említettek szerint tehát egy *tile* minden sorához két bájtnyi leíró adat tartozik. Ezeknek valamilyen kombinációja szükséges ahhoz, hogy a sorban lévő mind a 8 pixel színét el lehessen tárolni. A Game Boy kijelzője 4 féle színt tud megjeleníteni, 4 féle állapotot pedig 2 bittel tudunk ábrázolni. minden pixelhez tehát két bit kell hogy tartozzon... és pont két bájtunk van! Az egyes pixelek színeit az alábbiak alapján tudjuk kiolvasni:

<b>A tile sorának pixelei</b>	0	1	2	3	4	5	6	7
<b>2. byte</b>	0	1	1	0	0	0	1	0
<b>1. byte</b>	1	1	1	0	1	0	0	1
<b>Színek</b>	01	11	11	00	01	00	10	01

4.2. táblázat. A színek értékeinek meghatározása

A fenti 4.2-es táblázat alsó sorában szereplő értékeket úgy lehet megkapni, hogy a felette lévő két sor azonos oszlopaiban szereplő értékeket egymás után összeolvassuk. Figyelni kell rá, hogy az első bit a második bájtból jön, nem az elsőből, tehát fordított sorrendet alkalmaztak a tervezés során a Nintendo mérnökei. Ezek a kiolvasott értékek decimális számmá alakíthatók, ezek adják meg azt, hogy a palettából melyik színt kell használni. Az alapértelmezett paletta (illetve annak szürkeárnyalatos analógiája):

- 00 : fehér
- 01 : világosszürke
- 10 : szürke
- 11 : fekete

A programozók azonban megadhattak, és definiálhattak saját palettát, melyet az 0xFF47-es címen lehetett elérni. A fenti, decimálissá alakított szín azonosítókkal lehetett elérni a definiált színeket: 0-s érték esetén a paletta 0. és 1. bitjén tárolt színt kell használni, 1-es érték esetén a 2., illetve 3. bitet, és így tovább. Ezeket a konverziókat és visszakereséseket az emulátorban a PPU struktúra `select_colors` függvénye végzi el. A megfelelő szín megkeresése után, annak értékét beírva a `framebuffer`-be, a ciklus végére jutottunk, sikerült kirajzolni egy pixelt.

Az idáig ismertetett megoldások és modulok implementálásával az emulátor már le tudja futtatni a Boot ROM-ot, amely futása közben felülről lefelé beúszik a Nintendo logó, az alábbi módon:



4.4. ábra. Beúszó Nintendo logó – a Boot ROM futása közben

#### 4.2.4. Sprite rendering



4.5. ábra. A *Megaman* játék sprite-jai

a Megaman játék által tartalmazott összes *sprite*-ot figyelhetjük itt meg. Szerencsére esetükben csak előjel nélküli, *unsigned* azonosítókról beszélhetünk, tehát az előjeles esetek lekezelésével nem kell foglalkozni. A fenti memória területen 40 darab *sprite* adata fér el, renderelés során ezeken kell végigmenni, és a megfelelőket az *attribútumaiknak* megfelelően kirajzolni. Ezek az attribúrumok a SAT-ban, vagyis a *Sprite Attribute Table*-ben találhatók, a 0xFE00 – 0xFE9F területen. Itt minden *sprite*-hoz pontosan 4 bájtnyi attribútum van eltárolva. Ezek a következők:

- **A *sprite Y koordinátája*:** a látható kijelzőn értelmezett Y koordináta mínusz 16 (a 8×16-os *sprite*-ok miatt).
- **A *sprite X koordinátája*:** a látható kijelzőn értelmezett X koordináta mínusz 8 (a 8×16-os *sprite*-ok miatt).
- **Sprite azonosító:** ezt az azonosítót használva tudjuk megtalálni a megfelelő adatot a 0x8000 – 0x8FFF területen.
- **Attribútumok:** egyéb, a *sprite*-ok kinézetét és renderelését szabályozó opciókat tároló regiszter.

A fentiek közül az első három bájt triviális, hiszen a *tile renderingnél* is voltak hasonló értékek, azonban az attribútum regiszter újdonság. Nézzük ennek a regiszternek a bitjeit:

- **7. bit:** más néven Prioritás bit, vagy *Priority Flag*, amennyiben ennek az értéke 1, akkor a *sprite* a háttér és/vagy *window* alá fog kerülni, és így nem lesz látható, azzal a kivétellel, ha az előbbiekk közül valamelyik színe fehér – ebben az esetben látható lesz. Ha a bit értéke 0, akkor minden a háttér és/vagy *window* fölött fog megjelenni a *sprite*.
- **6. bit:** ez a bit az Y flip bit, ha az értéke 1, akkor az aktuális *sprite* fejjel lefelé fog kirajzolódni (tükröződik az Y tengely mentén). Ezzel az attribútummal van

Ahhoz, hogy ne csak a Boot ROM, illetve az egyszerűbb, csak *tile renderinget* használó játékok fussanak megfelelően, további renderelési módszerek implementálására van szükség. A mozgó, villogó, interaktív elemek az előzőekben már említett ún. *sprite*-ok, melyek megjelenítésükben különböznek a *tile*-ktől. A *sprite*-ok adatai a RAM felosztását taglaló fejezetben már említettek szerint a 0x8000 – 0x8FFF területen vannak eltárolva, a 4.5-ös ábrán látható erre egy példa:

megvalósítva például a Mario játékban az, mikor rááugrunk egy teknősre, ami ettől felfordul a szó szoros és átvitt értelmében is.

- **5. bit:** ez a bit az X flip bit, ha az értéke 1, akkor az aktuális *sprite* X tengely menti tükröképe fog menjelenni a kijelzőn. Ezzel lehetséges a karakter *sprite*-ját minden a menetiránnyal megegyező irányban kirajzolni.
- **4. bit:** ezzel a bittel az aktuális *sprite* kirajzolásához szükséges palettát tudjuk megadni, 0 értéke esetén a 0xFF48 címen található bájt lesz a palette, egyébként pedig a 0xFF49.
- **3. - 0. bitek:** nincsenek használatban.

Az attribútumok ismertetésével minden információ a birtokunkban van ahhoz, hogy implementáljuk a *sprite renderinget*. A renderelő ciklus leegyszerűsített pszeudokódja a következő:

0. Nézzük meg a sprite méretet.
1. i=0-től 40-ig egyesével :
  - 1.1 Számoljuk ki az sprite attribútumainak címét az i segítségével.
  - 1.2 Mentsük le a sprite X és Y pozícióit tartalmazó regiszterek értékeit.
  - 1.3 Mentsük le a sprite azonosítóját és az attribútumait.
  - 1.4 Ellenőrizzük le, hogy az aktuális sprite rajta van-e az aktuális scanline-on. Ha nem: GOTO 1.
  - 1.5 Határozzuk meg a sprite jelenleg kirajzolandó sorát.
  - 1.6 Határozzuk meg a sprite jelenlegi sorának adatait tartalmazó byte-ok kezdőcímét, és mentsük le őket.
  - 1.7 Vegyük ki a két bájt által tárolt adatokból a kurrens pixel színének értékét.
  - 1.8 Határozzuk meg a színpalette alapján hogy az adott szín pontosan milyen.
  - 1.9 Írjuk a framebuffer megfelelő helyére a megfelelő szín értékét, majd GOTO 1.

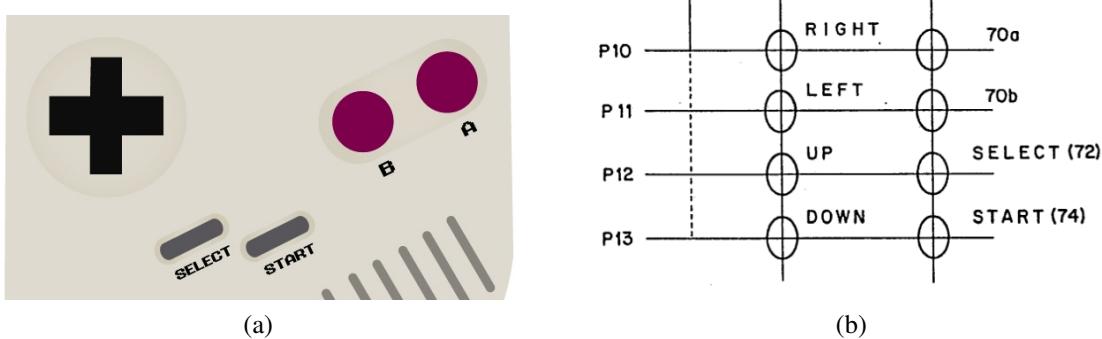
A fenti algoritmus nagy része a *tile rendering* ismertetése során adott megoldással összehasonlítva triviális, a maradék pedig az attribútumok leírásából kikövetkeztethető.

## 5. fejezet

# Joypad

A PPU, azaz a Game Boy renderelésért felelős grafikus alrendszere után itt az ideje a másik bemeneti modul, a *joypad* implementálásának. Ahogy a PPU esetében is, úgy itt is elmondható, hogy nagyon fontos részegységről van szó, hiszen míg a renderelés eredményét "csak" látja a kijelzőn a felhasználó, addig a *joypad* gombjait megnyomva tud érdemben *komunikálni*, reagálni egyes helyzetekre, interakcióba lépni a hardverrel, vagy az emulátorral, amely ennek hatására változtatja a működését. Ha rossz a gombnyomás élménye, többedszerre sem érzékeli a hardver vagy emulátor a gombnyomást, esetleg késik a reakció a gombnyomásra, az mind-mind csalódottságot, és idegességet ébreszt a felhasználóban, mely következtében végső soron abbahagyja a játékot. Nagyon fontos tehát nem csak az eredeti hardver viselkedésének emulálása, hanem az is, hogy az emulálás élménye se maradjon el a Game Boy-étől.

### 5.1. Technikai háttér és a *Joypad Register*



5.1. ábra. A *joypad* gombjai (a), és sematikus kapcsolási (logikai) rajza (b)

Ahogy a 5.1-es ábrán látható, a Game Boy konzolon 8 darab gomb található. Ezeket szokás két nagyobb csoportra is bontani. Az egyik csoport a bal oldalon látható ún. *D-pad*, amely látszólag egy nagy, plusz jel alakú gomb, az igazság azonban az, hogy 4 gombot olvaszt eggyé. A plusz jel négy ágának végein találhatóak meg az egyes irányokhoz rendelt gombok – innen jön a *D-pad* megnevezés is: *Direction pad*, azaz *irányító gomb*.

Ennek segítségével tudjuk a bal kéz hüvelykujjával irányítani a játékonként változó elemet – legyen az játszható karakter, Tetris-elem, stb.

A másik, akciógombok csoportját a maradék gombok alkotják: *A*, *B*, *Select* és *Start*. Az első kettővel általában a játékon belüli akciókat, képességeket, funkciókat tudjuk aktiválni, míg az utóbbi kettő a játék megszakítását, illetve a menükben való navigálást segíti.

Ami a fenti, 5.1-es ábra (b) részét illeti: ez a kép mutatja be a Game Boy gombainak kapcsolási rajzát. A karikák mutatják az egyes gombokat, mellettük a nevükkel, a vonalak pedig az őket összekötő vezetékeket jelölik. Első pillantásra bonyolultnak tűnhet, és felmerülhet bennünk, hogy miért nem két vonal (azaz vezeték) csatlakozik minden gombhoz – ezekre egyfajta kapcsolóként gondolva, amelyek nyomásra zárják az áramkört. Nos, a Nintendo japán mérnökei ott spóroltak és optimalizáltak ahol tudtak – ezt a gombok összekötésének megtervezésekor is szem előtt tartották. A hétköznapi számítógép billentyűzetekhez hasonló módon, egyfajta mátrixos elrendezést találtak ki, ezzel sok kábelt megspórolva. Ennek megfelelően van tehát két kábelcsoport, az egyik a *P10*, *P11*, *P12*, *P13*, amelyek rendre egy-egy gombhoz vannak kötve minden gombcsoportban, a másik pedig a *P14* és *P15*, amelyek azt jelölik ki, hogy melyik gombcsoportról van éppen szó. Ennek mintájára működik az *0xFF00* memóriacímen található *Joypad Register*, amely a mérnökök optimalizációjának hála egy bájtban képes eltárolni az összes gomb aktuális állapotát. Ennek a bájtnak a bitjei a következőket jelentik:

- **7. - 6. bitek:** nincsenek használatban.
- **5. bit:** akciógomb kiválasztó bit, a *P15*-ös vezetéknek felel meg. 0 érték esetén kerül kiválasztásra az *A*, *B*, *Select* és *Start* gombok csoportja.
- **4. bit:** iránygomb kiválasztó bit, a *P14*-es vezetéknek felel meg. 0 érték esetén kerül kiválasztásra a *D-pad* 4 darab iránygombja.
- **3. bit:** a 4. és 5. bitek értékétől függően a lefelé mutató iránygomb vagy a *Start* gomb állapotát mutatja. Akkor tekintjük lenyomott állapotúnak, ha az értéke 0. A *P13*-as vezetéknek felel meg.
- **2. bit:** a 4. és 5. bitek értékétől függően a felfelé mutató iránygomb vagy a *Select* gomb állapotát mutatja. Akkor tekintjük lenyomott állapotúnak, ha az értéke 0. A *P12*-es vezetéknek felel meg.
- **1. bit:** a 4. és 5. bitek értékétől függően a balra mutató iránygomb vagy a *B* gomb állapotát mutatja. Akkor tekintjük lenyomott állapotúnak, ha az értéke 0. A *P11*-es vezetéknek felel meg.
- **0. bit:** a 4. és 5. bitek értékétől függően a jobbra mutató iránygomb vagy az *A* gomb állapotát mutatja. Akkor tekintjük lenyomott állapotúnak, ha az értéke 0. A *P10*-es vezetéknek felel meg.

Érdekesség, hogy a fenti regiszter esetén minden esetben az jelenti a lenyomott állapotot, ha az adott bit értéke 0. Ebből az következik, hogy a megvalósítás során a regiszter esetében a *0xFF* érték lesz a *default*, ezt kell frissíteni, és a megfelelő biteket nullára állítani a megfelelő működés eléréséhez. A következő alfejezet fogja bemutatni, hogy mindez hogyan lehet implementálni.

## 5.2. Implementáció

```
1 LD A, $20
2 LD ($FF00), A
3 LD A, ($FF00)
4 LD A, ($FF00)
5 CPL
6 AND $0F
7 SWAP A
8 LD B, A
9 LD A, $10
10 LD ($FF00), A
11 LD A, ($FF00)
12 LD A, ($FF00)
13 LD A, ($FF00)
14 LD A, ($FF00)
15 LD A, ($FF00)
16 LD A, ($FF00)
17 CPL
18 AND $0F
19 OR B
20 LD B, A
21 LD A, ($FF8B)
22 XOR B
23 AND B
24 LD ($FF8C), A
25 LD A, B
26 LD ($FF8B), A
27 LD A, $30
28 LD ($FF00), A
29 RET
```

5.2. ábra.

A Ms. Pacman játék gombnyomás kezelő kódja

5.2-es ábrán látható kód mutatja a Ms. Pacman játék kódjának egy részletét, amely a gombok lenyomását figyeli. Először betölti az A regiszterbe a 0x20 (0b00100000) értéket (1. sor), amellyel kijelöli a P14-es csatlakozást, majd az A-t a *Joypad Registerbe* tölti (2. sor). A *Joypad Register* A regiszterbe történő kétszeri betöltéssel egyfajta várakozást valósít meg (3-4. sor). Ezt követően veszi az A regiszter komplementerét (5. sor), majd kinullázza az alsó 4 bitjét, felcseréli azt a felső 4 bittel, és elmenti a B regiszterbe (6-8. sor). A 9. sorban kijelöli a P15-ös vezetéket, majd a 10.-ben eltárolja azt a *Joypad Registerbe*. Ez után a 11.-től a 16. sorig várakozik, majd komplementálja A-t, veszi az alsó 4 bitjét, összevagyolja B-vel, és elmenti a B-be (17-20. sor). A következő részben beolvassa a régi *joypad* állapotot a memóriából, beállítja a jelenleg lenyomva tartott gombokat, elmenti az új állapotot reprezentáló bajtot, kiveszi a P14 és P15 kijelölését, *reseteli* az állapot regisztert, majd visszatér a szubrutin (21-29. sor).

A fenti magyarázat azért volt fontos, mert rávilágít egy olyan viselkedésre, amely felett nagy valószínűséggel elsiklik a legtöbb emulátor fejlesztő: ahhoz, hogy a lenyomott gomb kódját a regiszterbe írjuk, előbb ellenőrizni kell, hogy a regiszter értéke szerint (amit a játék állít be) a játék mely gombcsoporthoz kívánca. Ezen kód értelmezésével tehát sok-sok órányi *debuggolástól* kímélheti meg magát a fejlesztő – ez a többi területre is érvényes: érdemes, sőt, ajánlott a játékok kódját visszafejteni, és megnézni hogy mit hogyan csinálnak, mely memóriaterületekre írnak, hogyan kezelik a gomblenyomást, stb., hiszen végső soron az emulátor implementálásának "másik fele" a célhardverre írt játékok kódja: az egyik nem működik a másik nélkül.

Most, hogy már tisztában vagyunk a megfelelő regiszterrel, illetve azzal, hogy a játékok hogyan veszik hasznát a gomboknak, hogyan detektálják azok lenyomását, érdemes megtervezni az emulátor gomblenyomásának kezelését.

### 5.2.1. Gombnyomás detektálás

Ahhoz, hogy detektálni tudjuk egy gomb lenyomását, és egyúttal reagálni tudjunk rá, szükségünk van egy külső könyvtárra, amely megvalósítja ezt a fajta eseménykezelést. Az

emulátor fejlesztésének kezdetén, mikor a *framebuffert* megvalósító *library* kiválasztására került a sor, figyelembe vettet, hogy később szükség lesz majd a gombnyomások kezelésére is – így esett a választás a 2.2.2-es szakaszban taglalt *minifb* könyvtárra. Ez a könyvtár nem csak a renderelést könnyíti meg, hanem képes arra amire most szükségünk van: a gombnyomások regisztrálására.

A *minifb* *crate* Window struktúrája valósítja meg a renderelésnél is használt ablak megjelenítését, frissítését, és egyéb funkciókat, például a billentyű lenyomásának kezelését. A billentyűzettel kapcsolatos függvények, és azok magyarázatai a következők:

- **fn** `get_keys(&self)` : az aktuálisan lenyomott billentyűk azonosítóiit adja vissza egy `Option<Vec<Key>>` értékként, ezeket iterátorral tudjuk bejárni.
- **fn** `get_keys_pressed(&self, repeat: KeyRepeat)` : az előző függvényhez hasonló módon a lenyomott billentyűket adja vissza egy tömbben, azonban a `repeat` paraméterben meg lehet adni, hogy kiszűrje a folyamatosan nyomva tartott billentyűket.
- **fn** `is_key_down(&self, key: Key)` : a `key` paraméterben megadott billentyűazonosítóhoz tartozó billentyű lenyomását vizsgálja, egy `bool` értékkel tér vissza. A visszatérési érték akkor lesz igaz, ha a billentyű éppen le van nyomva.
- **fn** `is_key_pressed(&self, key: Key, repeat: KeyRepeat)` : az előző két függvény ötvözete: egy `bool` értékkel tér vissza, amelynek értéke 1, ha a paraméterben megadott azonosítóhoz tartozó billentyű le van nyomva, illetve szintén paraméterben megadható a folyamatosan lenyomott billentyűk kiszűrése is.

Ezek közül a függvények közül végül a legegyszerűbb működésű `is_key_down`-t választottam (annak ellenére, hogy az összes többivel megvalósítható a működés), amely segítségével az alábbi, 5.1-es táblázatban szereplő megfeleltetések szerint vizsgálom a lenyomott billentyűket:

<b>A Game Boy gombjai</b>	<i>Jobbra</i>	<i>Balra</i>	<i>Fel</i>	<i>Le</i>	<i>A</i>	<i>B</i>	<i>Select</i>	<i>Start</i>
<b>Megfeleltetett gombok</b>	<i>D</i>	<i>A</i>	<i>W</i>	<i>S</i>	<i>J</i>	<i>K</i>	<i>Space</i>	<i>Right Shift</i>
<b>Azonosítók</b>	0	1	2	3	4	5	6	7

5.1. táblázat. A gombok kiosztása a billentyűzeten

Így például a *jobbra* gombnak megfeleltetett *D* gomb lenyomását az alábbiak szerint vizsgálom:

```
if window.is_key_down(Key::D) {
    self.pressed_button(0, cpu);
} else {
    self.released_button(0);
}
```

A fenti kódrészlet a gomblenyomás kezelését megvalósító Joypad struktúrában szereplő `scan_window_button_pressed` függvény része. minden lehetséges billentyűre (8

darab) meg van írva a megfelelő `pressed` és `released` állapothoz tartozó függvényhívás. Ez a `scan_window_button_pressed` függvény minden CPU ciklus elején meghívódik a fő ciklusban.

A `released_button` egész egyszerűen veszi a felengedett billentyű azonosítóját, és a *Joypad Registerben* 1-re állítja az értékét.

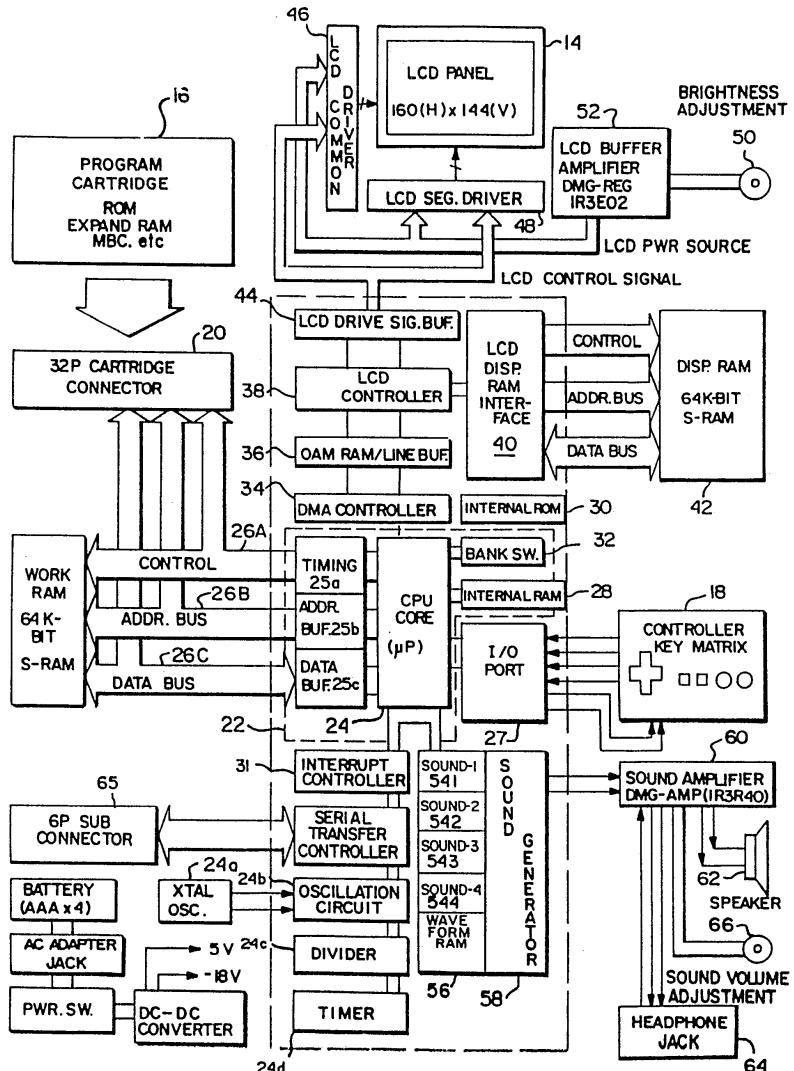
A `pressed_button` működése azonban kissé bonyolultabb. Ez a függvény először is megvizsgálja, hogy a paraméterben kapott gomb (billentyű) állapota változott-e a legutóbbihoz képest, majd ez alapján beállít egy `bool` típusú változót. Ezt követően megállapítja az azonosító alapján hogy a kapott gomb melyik csoportba tartozik, majd lementi az `0xFF00` aktuális állapotát egy változóba. A következő lépés az, hogy az adott gomb csoportba tartozó, kijelölt bit állapotát figyelembe véve megállapítja, hogy a játék kíváncsi-e az adott gomb lenyomására, és ha igen, (és az első lépés szerint állapotváltozás is történt), akkor megszakítási kérelmet küld a processzornak.

Az utolsó gomnyomással kapcsolatos függvény az `update_state`, amely a fejezet bevezetőjében lévő kód kapcsán ismertetett leírásnak megfelelően megnézi, hogy a játék melyik gombcsoportra kíváncsi, majd aszerint frissíti az `0xFF00` címen lévő *Joypad Register*t. Ezt a függvényt is a `scan_window_button_pressed`-hez hasonlóan meg kell hívni minden CPU ciklus elején, és visszatérési értékével frissíteni kell a *Joypad Register*t.

## 6. fejezet

### Függelék

#### 6.1. A Nintendo Game Boy hivatalos architektúrája



6.1. ábra. A szabadalomban szereplő Game Boy architektúra

## 6.2. A processzor opkód táblái

X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	XA	XB	XC	XD	XE	XF	
NOP	1 4	LD BC,d16	LD (BC),A	INC BC	INC B	DEC B	LD B,08	RLC A	LD (16),SP	AUD HL,BC	DEC BC	INC C	DEC C	LD C,d8	RCCA	
0x	.. .	3 12	1 8	1 8	1 4	2 8	1 4	0 0 0 C	3 20	1 8	1 8	1 4	2 8	1 4	0 0 0 C	
STOP	0	LD DE,d16	LD (DE),A	INC DE	INC D	DEC D	LD D,08	RRA	JR r8	AUD HL,DE	LD A,(DE)	DEC DE	INC E	DEC E	LD E,d8	RRA
1x	2 4	3 12	1 8	1 8	1 4	2 8	1 4	0 0 0 C	.. ..	1 8	1 8	1 4	2 8	1 4	0 0 0 C	
JR NZ,r8	2 12/8	LD HL,d16	LD (HL),A	INC HL	INC H	DEC H	LD H,08	DAA	JR Z,r8	ADD HL,HL	LD A,(HL)	DEC HL	INC L	DEC L	LD L,d8	CPL
2x	3 12	1 8	1 8	1 4	2 8	1 4	2 8	1 8	2 12/8	ADD HL,HL	LD A,(HL)	DEC HL	INC L	DEC L	LD L,d8	-1 1 -
JR NC,r8	2 12/8	LD SP,d16	LD (HL),A	INC SP	INC (HL)	DEC (HL)	LD (HL),08	SCF	JR C,r8	ADD HL,SP	LD A,(HL)	DEC SP	INC A	DEC A	LD A,d8	COP
3x	3 12	1 8	1 8	1 4	2 12	1 8	1 8	1 12	2 12/8	ADD HL,SP	LD A,(HL)	DEC SP	INC A	DEC A	LD A,d8	-0 0 0 C
LD B,B	1 4	LD B,C	LD B,D	LD B,E	LD B,F	LD B,G	LD B,A	LD B,C	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..
4x	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..
LD D,B	1 4	LD D,C	LD D,D	LD D,E	LD D,F	LD D,G	LD D,A	LD D,B	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..
5x	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..
LD H,B	1 4	LD H,C	LD H,D	LD H,E	LD H,F	LD H,G	LD H,A	LD H,B	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..
6x	1 4	1 4	1 4	1 4	1 4	1 4	1 4	1 4	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..
LD (HL)B	1 8	LD (HL)C	LD (HL)D	LD (HL)E	LD (HL)F	LD (HL)G	LD (HL)A	HALT	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..
7x	1 8	1 8	1 8	1 8	1 8	1 8	1 8	1 4	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..
ADD A,B	1 4	ADD A,C	ADD A,D	ADD A,E	ADD A,H	ADD A,L	ADD A,(HL)	ADD A,(HL)	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..
8x	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	Z 0 H C	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..
SUB B	1 4	SUB C	SUB D	SUB E	SUB H	SUB L	SUB (HL)	SUB A	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..
9x	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	Z 1 H C	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..
AND B	1 4	AND C	AND D	AND E	AND H	AND L	AND (HL)	AND A	XOR B	XOR C	XOR D	XOR E	XOR F	XOR G	XOR H	XOR A
Ax	Z 0 1 0	Z 0 1 0	Z 0 1 0	Z 0 1 0	Z 0 1 0	Z 0 1 0	Z 0 1 0	Z 0 1 0	Z 0 0 9	Z 0 0 9	Z 0 0 9	Z 0 0 9	Z 0 0 9	Z 0 0 9	Z 0 0 9	Z 0 0 9
OR B	1 4	OP C	OP D	OP E	OP (HL)	OP L	OP (HL)	OP A	CPB	CPD	CP E	CP G	CP L	CP P	CP A	CP (HL)
Bx	Z 0 0 9	Z 0 0 9	Z 0 0 9	Z 0 0 9	Z 0 0 9	Z 0 0 9	Z 0 0 9	Z 0 0 9	Z 0 0 9	Z 0 0 9	Z 0 0 9	Z 0 0 9	Z 0 0 9	Z 0 0 9	Z 0 0 9	Z 0 0 9
RET NZ	2 10/8	POP BC	JP NZ,a16	JP a16	PUSH BC	ADD A,d8	RST 00H	RET Z	JP Z,a16	PREFIX CB	CALL Z,a16	CALL a16	ADC A,d8	REST 08H	.. ..	.. ..
Cx	1 12	3 16/12	3 16	3 24/12	1 16	1 16	1 16	1 16	3 16/12	1 16	3 24/12	3 24/12	2 8	1 16	.. ..	.. ..
RET NC	1 12	JP NC,a16	CALL NC,a16	PUSH DE	SUB 08	RST 00H	RET C	RET C	JP C,a16	CALL C,a16	CALL C,a16	CALL C,a16	ADC A,d8	REST 18H	.. ..	.. ..
Dx	1 12	3 16/12	3 16	3 24/12	1 16	2 8	1 16	1 16	1 16	3 16/12	3 24/12	3 24/12	2 8	1 16	.. ..	.. ..
LDH (AB),A	2 12	POP HL	LD (C),A	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..
Ex	2 12	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..
LDH A,(AB)	2 12	POP AF	LD A,(C)	.. ..	.. ..	.. ..	.. ..	.. ..	PUSH AF	OR 30H	LD HL,SP+18	LD SP,HL	LD A,(A16)	EI	GP d8	REST 20H
Fx	2 12	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	1 16	2 8	1 8	1 16	1 4	1 16	.. ..	.. ..
		Z 1 H C	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	Z 0 0 9	.. ..	.. ..	.. ..	.. ..	.. ..	Z 1 H C	.. ..

6.2. ábra. Az első 256 opkódot tartalmazó tábla

6.3. 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. április 22.

.....

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](https://hu.wikipedia.org/wiki/Game_Boy).