



EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

PROGRAMOZÁSI NYELVEK ÉS FORDÍTÓPROGRAMOK

TANSZÉK

## NES játékkonzol emulációja Haskellben

*Témavezető:*

Poór Artúr  
egyetemi tanársegéd

*Szerző:*

Suhajda Tamás József  
programtervező informatikus BSc

*Budapest, 2020*

Az eredeti szakdolgozati / diplomamunka témabejelentő helye.

# Tartalomjegyzék

<b>1. Bevezetés</b>	<b>4</b>
<b>2. Felhasználói dokumentáció</b>	<b>6</b>
2.1. A feladat ismertetése . . . . .	6
2.2. Minimum rendszerkövetelmények . . . . .	7
2.3. Telepítés . . . . .	7
2.4. Indítás . . . . .	7
2.5. Kompatibilis kazetták . . . . .	7
2.6. Főmenü . . . . .	9
2.6.1. Kazetta kiválasztása . . . . .	9
2.6.2. Irányítási beállítások megtekintése . . . . .	10
2.7. Irányítás . . . . .	10
2.8. Játék alatt elérhető funkciók . . . . .	12
2.8.1. Mentés . . . . .	13
2.8.2. Betöltés . . . . .	14
2.8.3. A játék megállítása . . . . .	15
2.8.4. Visszatérés a főmenübe . . . . .	15
2.8.5. Képernyőképek . . . . .	16
<b>3. Fejlesztői dokumentáció I: A NES működésének ismertetése</b>	<b>17</b>
3.1. A NES felépítése . . . . .	17
3.2. Órajel-frekvenciák . . . . .	18
3.3. A központi feldolgozóegységhez kapcsolódó fogalmak . . . . .	18
3.3.1. Opciók . . . . .	18
3.3.2. Regiszterek . . . . .	20
3.3.3. Memórialap . . . . .	20

3.3.4. Hívási verem . . . . .	20
3.3.5. Megszakítás . . . . .	21
3.3.6. Opkód argumentum helye, fajtája . . . . .	21
3.3.7. Címzési módok . . . . .	22
3.3.8. Memóriatérkép . . . . .	23
3.3.9. Memóriatükrözés . . . . .	24
3.3.10. Státusz flagek . . . . .	24
3.3.11. Utasításkészlet . . . . .	25
3.4. Kazetták . . . . .	26
3.4.1. Az iNES fájlformátum . . . . .	26
3.5. A képfeldolgozó egység . . . . .	27
3.5.1. Színpalletta . . . . .	27
3.5.2. Paletta indexek . . . . .	27
3.5.3. Alakzattáblázat . . . . .	28
3.5.4. Rétegek . . . . .	29
3.5.5. Névtáblázatok . . . . .	30
3.5.6. Attribútumtáblázatok . . . . .	30
3.5.7. OAM (Object Attribute Memory) . . . . .	31
3.5.8. Regiszterek . . . . .	31
3.5.9. Memóriatérkép . . . . .	33
3.5.10. A háttér kirajzolásának egyszerű algoritmusa . . . . .	33
3.5.11. Sprite réteg kirajzolása . . . . .	35
<b>4. Fejlesztői dokumentáció II: Megvalósítás</b>	<b>37</b>
4.1. A feladat specifikációja . . . . .	37
4.2. Fejlesztői környezet . . . . .	37
4.3. Az emulációhoz kapcsolódó modulok . . . . .	39
4.4. A felhasználói felületnél használt technológiák . . . . .	39
4.5. A felhasználói felülethez kapcsolódó modulok . . . . .	40
4.6. Az emulációt magába záró monád . . . . .	41
4.7. Adatrepräsentáció . . . . .	42
4.8. Mentés és betöltés . . . . .	43
4.9. Fontos modulok részletes áttekintése . . . . .	43

4.9.1.	Nes.Cartridge.Memory . . . . .	44
4.9.2.	Nes.Cartridge.INES.Parser . . . . .	45
4.9.3.	Nes.Cartridge.Mappers . . . . .	45
4.9.4.	Nes.Emulation.Monad . . . . .	46
4.9.5.	Nes.CPU.Emulation . . . . .	46
4.9.6.	Nes.PPU.Emulation . . . . .	48
4.9.7.	Nes.Emulation.Controls . . . . .	48
4.9.8.	Nes.Emulation.MasterClock . . . . .	49
4.9.9.	Communication . . . . .	49
4.9.10.	Emulator.JoyControls . . . . .	49
4.9.11.	Emulator.Framerate . . . . .	50
4.9.12.	Emulator.CrtShader . . . . .	50
4.9.13.	Emulator.Logic . . . . .	51
4.9.14.	Emulator.Window . . . . .	51
4.10.	Egy processzorutasítás végrehajtása . . . . .	52
4.11.	A CPU és a PPU szinkronizációja . . . . .	53
4.12.	A grafikus felhasználói felület . . . . .	53
4.12.1.	Állapotok . . . . .	53
4.12.2.	Állapotátmenetek . . . . .	54
4.12.3.	Állapotmegjelenítés . . . . .	55
4.13.	Tesztelek . . . . .	56
4.13.1.	CPU tesztek . . . . .	56
4.13.2.	PPU tesztek . . . . .	57
4.13.3.	Eredmények . . . . .	57
4.13.4.	Tesztelt, megbízhatóan működő programok . . . . .	58
<b>5.</b>	<b>Összegzés</b>	<b>60</b>
<b>Ábrajegyzék</b>		<b>64</b>
<b>Táblázatjegyzék</b>		<b>65</b>

# 1. fejezet

## Bevezetés

A játékkonzol-emulátorok feladata, hogy egy kompatibilitási réteget képezzenek a modern x86 és ARM architektúrájú processzorok, valamint az elavult konzolokra megjelent játékok között, hogy azokat bárki zavartalanul élvezhesse a konzol birtoklása nélkül. Az emulációt végző programnak szoftveresen kell megvalósítania az eredeti konzol számítási egységei által nyújtott primitív utasításokat és a komponensek közötti kommunikációt, hogy a játékok az elvárt viselkedés szerint működjenek.

Az emulátorok világában a hardverközeli, elsősorban teljesítményre kihegyezett nyelvek használata (pl. C++) az elterjedt, mivel ezen a területen a program gyorsasága kulcsfontosságú. Ezeknél a nyelveknél az explicit memória kezelés és a vékony absztrakciós réteg megkönnyíti az optimalizációt, azonban ennek a kód átláthatósága látja a kárát. Ezzel szemben a funkcionális nyelvek erős kifejezőképessége és moduláris felépítést előnyben részesítő paradigmája az emulátorfejlesztésnél számos helyzetben könnyítik meg a programozó dolgát. A szakdolgozatom célja, hogy korszerű eszközök segítségével egy fejlesztőbarát, de mégis hatékony emulátort implementáljak funkcionális nyelven. A megfelelő teljesítménnyről a Haskell nyelv egyeduralkodó fordítója, a GHC gondoskodik, ami az egyik legfejlettebb fordítóprogram, ami funkcionális nyelvhez készült. Agresszív optimalizálási stratégiáján túl az is mellette szól, hogy a legfrissebb kiadása immár tartalmaz egy valós idejű alkalmazásokhoz szánt, alacsony késleltetésű szemétgyűjtőt.

A Nintendo Entertainment System (NES) generációjának legsikeresebb konzolja[16], több mint 61 millió darab kelt el belőle világszerte. Emiatt kezdettől fogva nagy volt az igény a NES emulátorokra és mára a hardver nem publikus

részei is fel lettek térképezve. Választásom azért erre a rendszerre esett, mert az internetről elérhető dokumentációk és leírások birtokában nincs szükség a konzol beszerzésére, a hardver viselkedésének felderítésére.

## 2. fejezet

# Felhasználói dokumentáció

### 2.1. A feladat ismertetése



2.1. ábra. Nintendo Entertainment System (1985)<sup>1</sup>

A program feladata a NES kazetták futtatása. A felhasználó grafikus felület segítségével kiválaszthatja a futtatni kívánt, iNES formátumú kazetta fájlt, majd annak betöltése után az emulátor belekezd a videókimenet előállításába.<sup>2</sup> Bemeneti eszközként billentyűzet vagy kontroller használható. A futtatás során lehetőség van az emulátor állapotának fájlként való mentésére és betöltésére. Az emulációt többféle módon is befolyásolhatja a felhasználó, amikbe beletartozik a játék szüneteltetése, valamint az utasításonként vagy képkockánként történő léptetés.

A programot azoknak ajánlom, akik egy letisztult kezelőfelülettel rendelkező, több platformon is elérhető NES emulátorral szeretnék játszani kedvenc játékaikat.

<sup>1</sup><https://upload.wikimedia.org/wikipedia/commons/8/82/Nintendo-Entertainment-System-NES-Console-FL.jpg>

<sup>2</sup>A hangfeldolgozást nem emulálja a program

## 2.2. Minimum rendszerkövetelmények

**Processzor:** Intel Core 2 Duo E8400 @ 3.0 GHz

**Memória:** 2 GB DDR2 @ 800 MHz

**Videókártya:** OpenGL 3.0 kompatibilis videókártya

**Tárhely:** 161 MB

**Operációs rendszer:** Linux, Windows

A program a fenti konfigurációt történő tesztelés során problémamentesen működött.

## 2.3. Telepítés

- Windows: A programot egy telepítőfájl segítségével telepíthetjük.

- Linux:

```
$ tar xvf pure-nes-1.0.tar.gz  
$ cd pure-nes-1.0  
$ ./configure  
$ make
```

## 2.4. Indítás

- Windows: Kattintsunk a program parancsikonjára a Start Menüben vagy az asztalon.

- Linux:

```
pure-nes-1.0$ stack exec pure-nes
```

## 2.5. Kompatibilis kazetták

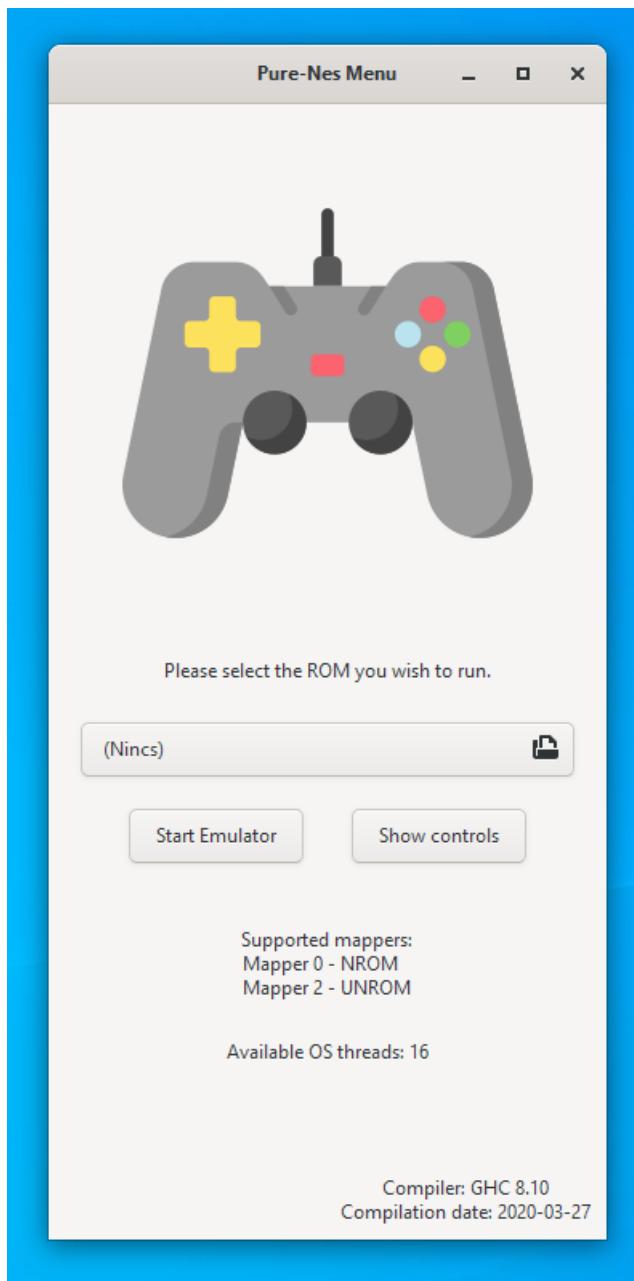
A NES megtervezésekor fontos cél volt, hogy a konzol életciklusa hosszú legyen, de ezt nem volt könnyű gazdaságos módon elérni. Azt a megoldás találták ki, hogy a

kazetta a játék mellett tartalmazzon speciális integrált áramköröket, amik igény szerint bővíthették a hardveres erőforrásokat. A kiegészítőegységek egy konkrét összeállítását *mapper*-nek nevezzük. A későbbi játékok előszeretettel használták ki ezt a lehetőséget. A legtöbb speciális áramkör a megnövelt háttértár kezeléséhez volt szükséges, de akadt olyan is, amelyik további hangcsatornákat adott hozzá a hangfeldolgozó egységhez. Az emulátorom jelenleg a 0, 2 és 3 azonosítójú mapper-eket használó kazettákat támogatja, ezáltal 471 játékot [1] képes elindítani<sup>3</sup>.

---

<sup>3</sup>Ritka esetekben előfordulhat, hogy egy elindítható kazetta nem működik megfelelően az emulált és a valódi hardver minimális eltérései miatt (lásd 4.13.3)

## 2.6. Főmenü



### 2.6.1. Kazetta kiválasztása

A fájlkiválasztó segítségével adjuk meg a futtatni kívánt kazettát vagy mentést, majd nyomjunk a *Start Emulator* gombra. A program hibaüzenettel jelzi, ha a kazetta nem kompatibilis vagy a fájl formátuma nem megfelelő.

### 2.6.2. Irányítási beállítások megtekintése

A főmenü *Show controls* gombjára kattintva megtekinthetjük a gombhozzárendeléseket.



### 2.7. Irányítás

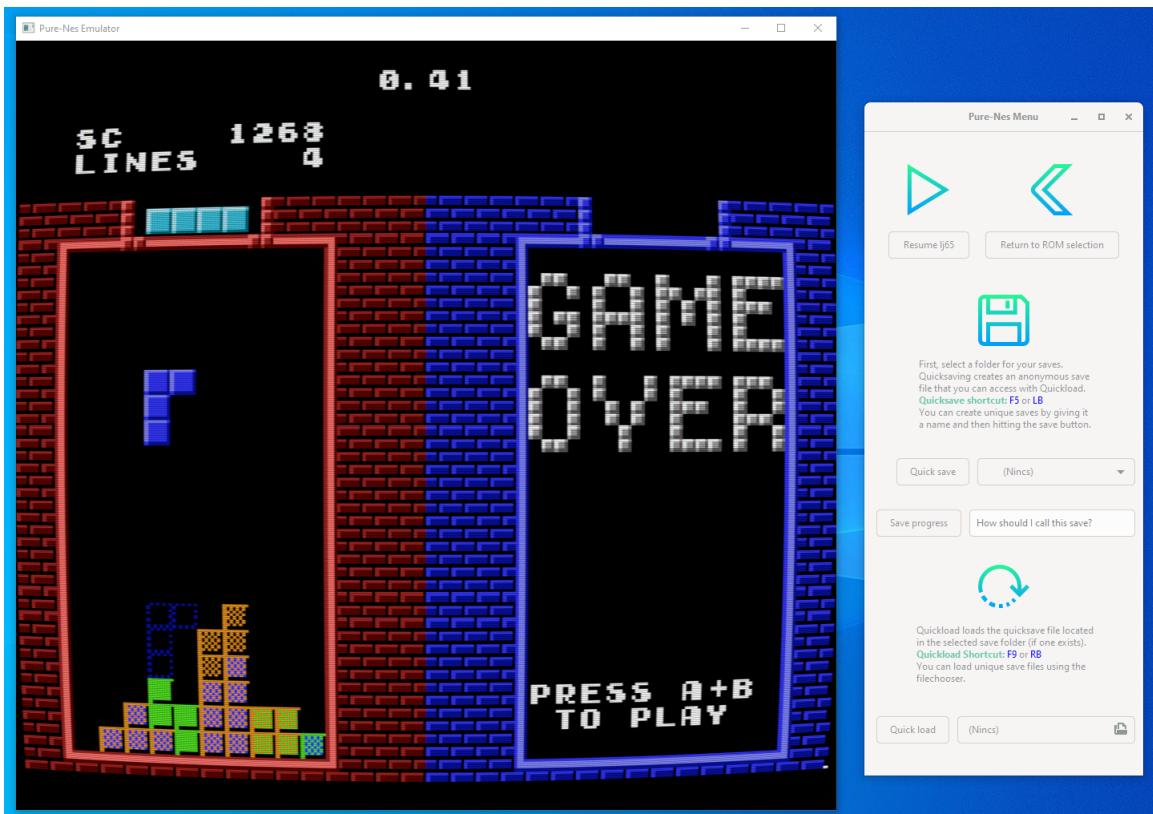
A 2.1 táblázatban látható módon vannak a fizikai gombok (a billentyűzeten vagy a kontrolleren) az emulált NES virtuális gombjainak megfeleltetve. Például, ha a játékon belül a *Select* gombot szeretnénk lenyomni, akkor a billentyűzeten a 3-as, vagy

a kontrolleren a 0. gombot kell lenyomnunk. A Joystick vezérlők gombkiosztása elszokott térni, ami azt jelenti, hogy a fizikailag ugyanott található gomboknak más az azonosítója. A felhasználó feladata, hogy szükség esetén egy másik programmal módosítsa eszközének kiosztását úgy, hogy az megegyezzen az emulátor szerint elvárta.

NES kontroller gomb	Billentyű	Kontroller
A	1	2. gomb
B	2	3. gomb
Select	3	0. gomb
Start	4	1. gomb
Up	Fel nyíl	DPad Fel
Down	Lefele nyíl	DPad Le
Left	Balra nyíl	DPad Bal
Right	Jobbra nyíl	DPad Jobb

2.1. táblázat. Játékok irányításához használt gombok

## 2.8. Játék alatt elérhető funkciók



2.2. ábra. A felhasználói felület játék közben<sup>4</sup>

Az 2.2 táblázatban felsorolt gombokkal lehet az emulációt irányítani és testre szabni. A szüneteltetés leállítja az emulált komponenseket és elérhetővé teszi a léptetési funkciókat. Képesek vagyunk a teljes rendszert egy processzorutasítással léptetni. minden léptetésnél láthatjuk a sztenderd kimeneten, hogy milyen utasítások fognak soron következni. Ha ennél gyorsabb ütemben szeretnénk léptetni az emulációt, akkor használjuk a képkockánti léptetést. A képre alapértelmezetten rákerül egy katódsugárcsöves megjelenítőket szimuláló effektus, de ezt a funkciót bármikor kikapcsolhatjuk.

<sup>4</sup>Link a képen szereplő LJ65 játékhoz: <https://github.com/Glavin001/node-nes/tree/master/roms/lj65>

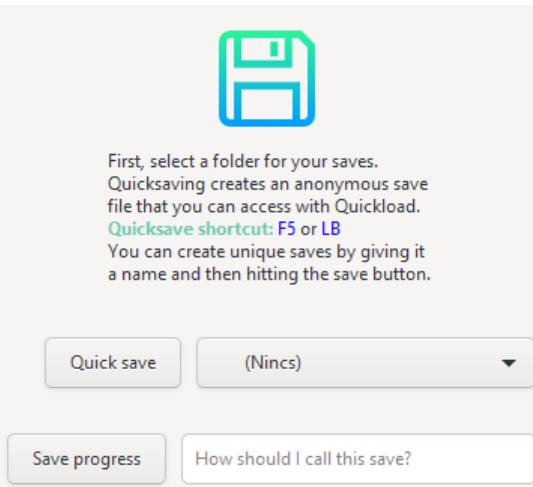
Emulátor funkció	Billentyű
Teljes képernyős nézet ki/be	R
Katódsugárcsöves képernyő-effektus ki/be	T
Szüneteltetés ki/be	Space
Egy processzorutasítás végrehajtása (szüneteltetés alatt)	E
Léptetés a következő képkockára (szüneteltetés alatt)	F

2.2. táblázat. Az emuláció vezérlése

### 2.8.1. Mentés

A felhasználónak ki kell jelölnie egy mappát, amit a program a mentések kezelésére használ. A program hibaüzenetet ad, ha enélkül próbálunk menteni vagy gyors betöltést végrehajtani.

A mentések a virtuális gép teljes állapota mellett a praktikusság érdekében a játék másolatát is tartalmazzák, tehát egy mentés betöltéséhez nem kell megőrizni a játék eredeti példányát.



### Gyorsmentés

A gyorsmentési funkcióval egy gombnyomásra elmenthetjük állásunkat. A mentés *quick.purenes* néven jön létre a mappában. Ha már létezik ilyen fájl, az felül lesz írva. Mentés létrehozása: **Quicksave** gomb vagy **F5** billentyű vagy a **4.** gomb a kontrolleren. A mentés sikereségét egy pipa vagy kereszt jelzi a kezelőfelületen

a mentés ikon mellett. Ha a kontroller rendelkezik rezgőmotorral, akkor a sikeres mentés rezgéssel is jelezve lesz.

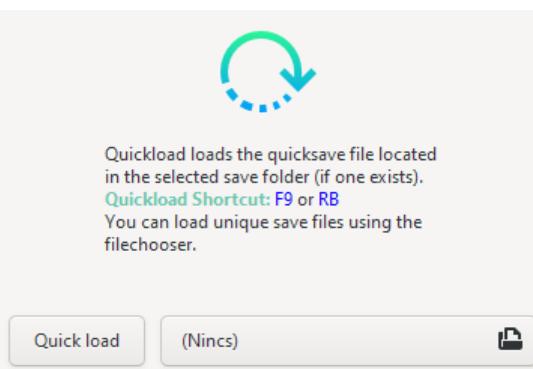
### Egyedi mentés létrehozása

Adhatunk nevet a mentéseknek, ehhez írjuk be a nevet a szövegmezőbe és nyom-junk a *Save* gombra. A mentés *{név}.purenes* néven jön létre a mappában. A *quick* nevet nem adhatjuk a mentésnek.

### 2.8.2. Betöltés

#### Gyorsbetöltés

A gyorsbetöltési funkció játszik közben érhető el, miután kiválasztottuk a mentéseket tároló mappát. A **Quickload** gombbal, vagy az **F9** billentyűvel, vagy a kontroller **6.** gombjával visszatölthetjük a korábban létrehozott gyorsmentést. A betöltés sikeresége a mentéshez hasonló módon jelenik meg a kezelőfelületen. Sikertelen betöltés abból adódhat, hogy a mappában nem található gyorsmentés, vagy a mentési fájl sérült. A sikeres betöltést itt is rezgés fogja követni.



#### Egyedi mentés betöltése

Ez a funkció arra szolgál, hogy a gyorsmentésen kívül más mentéseket is be tudjunk tölteni. A fájlkiválasztó segítségével válasszuk ki a betölteni kívánt mentést. **Figyelem:** Mindkét betöltési módnál az aktuális munkamenet elveszik. Ha ezt el szeretnénk kerülni, akkor mentsünk előtte.

### 2.8.3. A játék megállítása

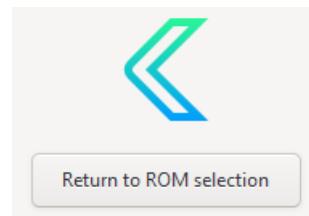
A játék futása bármikor felfüggeszthető a *Pause* gombbal, majd ezután folytat-ható a *Resume* gombbal.



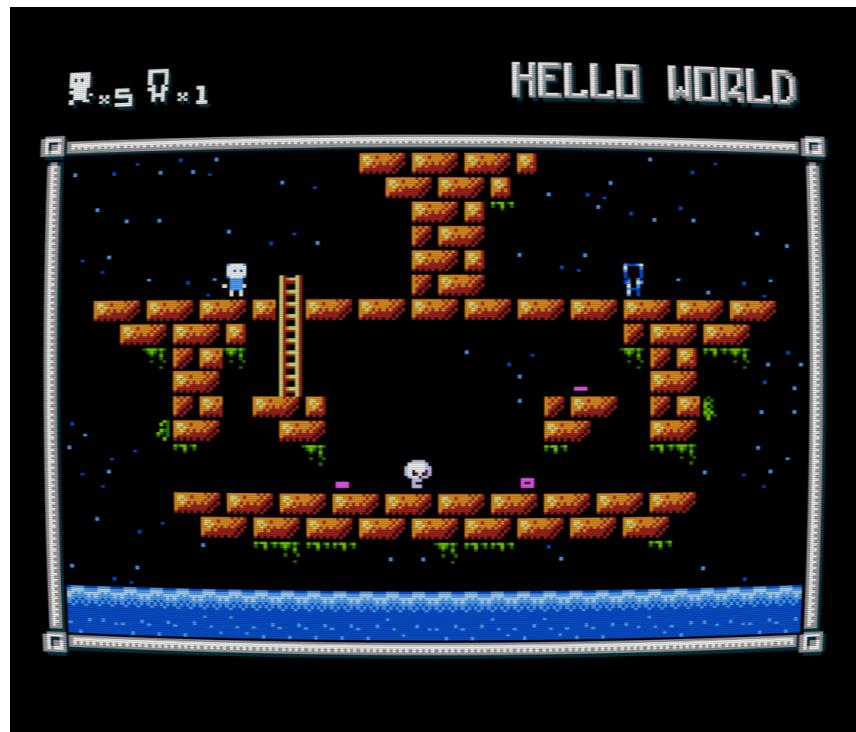
### 2.8.4. Visszatérés a főmenübe

A *Return To ROM selection* gomb bezárja a játékot és a felhasználói felületen visszanavigál minket a főmenübe.

**Figyelem:** Az állásunk elveszik, ha előtte nem mentünk.



### 2.8.5. Képernyőképek



2.3. ábra. Alter Ego. Linkért lásd: 4.13.4

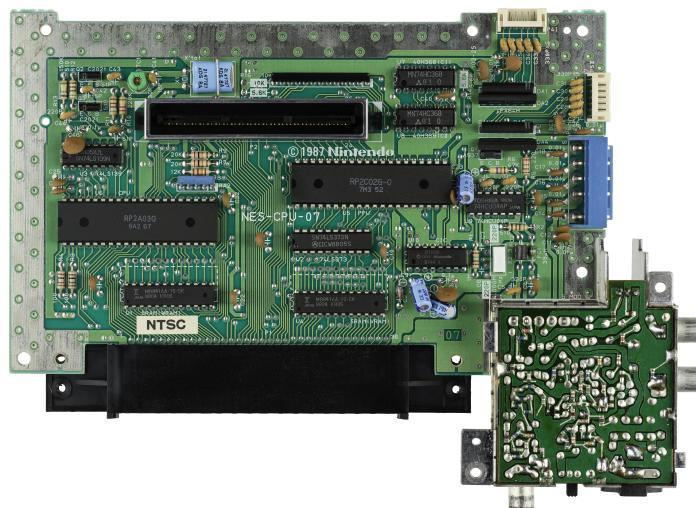


2.4. ábra. Lawn Mover. Linkért lásd: 4.13.4

### 3. fejezet

## Fejlesztői dokumentáció I: A NES működésének ismertetése

### 3.1. A NES felépítése



3.1. ábra. A NES alaplapja<sup>5</sup>

Ennek az ismertetőnek az a célja, hogy az olvasó magas szinten átlássa a NES legfontosabb egységeinek működését. A mélyebb megértéshez a forráskód tanulmányozását, valamint a *Nesdev*[5] oldalt ajánlom. A NES emulációjához az alábbi

---

<sup>5</sup><https://commons.wikimedia.org/wiki/File:Nintendo-NES-Mk1-Motherboard-Top.jpg>

komponenseket kellett megismernem, amik közül a CPU és a PPU működését fogom később részletesebben áttekinteni.

#### Ricoh RP2A03:

A hangchipet és központi feldolgozóegységet (CPU) tartalmazó integrált áramkör. Utóbbi nem más, mint az Apple II-ben és Commodore 64-ben használt 8-bites MOS Technology 6502.

#### Ricoh RP2C02:

A képfeldolgozó egység, rövid nevén PPU (Picture Processing Unit).

#### NROM[10], UNROM[11] és CNROM[12]:

Az emulátor által támogatott három kazettatípus.

#### Sztenderd NES kontroller[15]:

A konzol alapértelmezett beviteli eszköze.

## 3.2. Órajel-frekvenciák

A párhuzamosan működő komponenseket az órajelek hangolják össze. Az órajelfrekvencia határozza meg, hogy egy másodperc alatt hány atomi műveletet végez el egy komponens. A CPU és a PPU is rendelkezik saját órajelfrekvenciával, amit egy központi órajelből származtatnak [13].

- Központi órajel-frekvencia:  $f = \frac{236.25 \text{ MHz}}{11} \sim 21.477272 \text{ MHz}$
- CPU órajel-frekvencia:  $\frac{f}{12} \sim 1.789773 \text{ MHz}$
- PPU órajel-frekvencia:  $\frac{f}{4} \sim 5.369318 \text{ MHz}$

## 3.3. A központi feldolgozóegységhez kapcsolódó fogalmak

**Megjegyzés.** A hexadecimális értékeket \$ prefix-el jelölöm.

### 3.3.1. Opkód

Egy opkód [2, 3] a 6502 esetében csupán egyetlen bájt, amiből az utasításdeki doló egyértelműen meg tudja határozni a végrehajtandó utasítást és annak címzési

módját. Ezt a hozzárendelést az opkódmátrix írja le. Az emulációhoz emellett azt is tárolni kell az opkódmátrixban, hogy a végrehajtandó művelet hány CPU-órájel alatt fejeződik be, ugyanis csak ennek ismeretében tudjuk a CPU-t és a PPU-t precízen egymáshoz szinkronizálni.

	<b>x0</b>	<b>x1</b>	<b>x2</b>	<b>x3</b>	<b>x4</b>	<b>x5</b>	<b>x6</b>	<b>x7</b>	<b>x8</b>	<b>x9</b>	<b>xA</b>	<b>xB</b>	<b>xC</b>	<b>xD</b>	<b>xE</b>	<b>xF</b>
<b>0x</b>	BRK 7	ORA izx 6	KIL	SLO izx 8	NOP zp 3	ORA zp 3	ASL zp 5	SLO zp 5	PHP 3	ORA imm 2	ASL 2	ANC imm 2	NOP abs 4	ORA abs 4	ASL abs 6	SLO abs 6
<b>1x</b>	BPL rel 2*	ORA izy 5*	KIL	SLO izy 8	NOP zpx 4	ORA zpx 4	ASL zpx 6	SLO zpx 6	CLC 2	ORA aby 4*	NOP 2	SLO aby 7	NOP abx 4*	ORA abx 4*	ASL abx 7	SLO abx 7
<b>2x</b>	JSR abs 6	AND izx 6	KIL	RLA izx 8	BIT zp 3	AND zp 3	ROL zp 5	RLA zp 5	PLP 4	AND imm 2	ROL 2	ANC imm 2	BIT abs 4	AND abs 4	ROL abs 6	RLA abs 6
<b>3x</b>	BMI rel 2*	AND izy 5*	KIL	RLA izy 8	NOP zpx 4	AND zpx 4	ROL zpx 6	RLA zpx 6	SEC 2	AND aby 4*	NOP 2	RLA aby 7	NOP abx 4*	AND abx 4*	ROL abx 7	RLA abx 7
<b>4x</b>	RTI 6	EOR izx 6	KIL	SRE izx 8	NOP zp 3	EOR zp 3	LSR zp 5	SRE zp 5	PHA 3	EOR imm 2	LSR 2	ALR imm 2	JMP abs 3	EOR abs 4	LSR abs 6	SRE abs 6
<b>5x</b>	BVC rel 2*	EOR izy 5*	KIL	SRE izy 8	NOP zpx 4	EOR zpx 4	LSR zpx 6	SRE zpx 6	CLI 2	EOR aby 4*	NOP 2	SRE aby 7	NOP abx 4*	EOR abx 4*	LSR abx 7	SRE abx 7
<b>6x</b>	RTS 6	ADC izx 6	KIL	RRA izx 8	NOP zp 3	ADC zp 3	ROR zp 5	RRA zp 5	PLA 4	ADC imm 2	ROR 2	ARR imm 2	JMP ind 3	ADC abs 4	ROR abs 6	RRA abs 6
<b>7x</b>	BVS rel 2*	ADC izy 5*	KIL	RRA izy 8	NOP zpx 4	ADC zpx 4	ROR zpx 6	RRA zpx 6	SEI 2	ADC aby 4*	NOP 2	RRA aby 7	NOP abx 4*	ADC abx 4*	ROR abx 7	RRA abx 7
<b>8x</b>	NOP imm 2	STA izx 6	NOP imm 2	SAX izx 6	STY zp 3	STA zp 3	STX zp 3	SAX zp 3	DEY 2	NOP imm 2	TXA 2	XAA imm 2	STY abs 4	STA abs 4	STX abs 4	SAX abs 4
<b>9x</b>	BCC rel 2*	STA izy 6	KIL	AHX izy 6	STY zpx 4	STA zpx 4	STX zpy 4	SAX zpy 4	TYA 2	STA aby 5	TXS 2	TAS aby 5	SHY abx 5	STA abx 5	SHX aby 5	AHX aby 5
<b>Ax</b>	LDY imm 2	LDA izx 6	LDX imm 2	LAX izx 6	LDY zp 3	LDA zp 3	LDX zp 3	LAX zp 3	TAY 2	LDA imm 2	TAX 2	LAX imm 2	LDY abs 4	LDA abs 4	LDX abs 4	LAX abs 4
<b>Bx</b>	BCS rel 2*	LDA izy 5*	KIL	LAX izy 5*	LDY zpx 4	LDA zpx 4	LDX zpy 4	LAX zpy 4	CLV 2	LDA aby 4*	TSX 2	LAS aby 4*	LDY abx 4*	LDA abx 4*	LDX aby 4*	LAX aby 4*
<b>Cx</b>	CPY imm 2	CMP izx 6	NOP imm 2	DCP izx 8	CPY zp 3	CMP zp 3	DEC zp 5	DCP zp 5	INY 2	CMP imm 2	DEX 2	AXS imm 2	CPY abs 4	CMP abs 4	DEC abs 6	DCP abs 6
<b>Dx</b>	BNE rel 2*	CMP izy 5*	KIL	DCP izy 8	NOP zpx 4	CMP zpx 4	DEC zpx 6	DCP zpx 6	CLD 2	CMP aby 4*	NOP 2	DCP aby 7	NOP abx 4*	CMP abx 4*	DEC abx 7	DCP abx 7
<b>Ex</b>	CPX imm 2	SBC izx 6	NOP imm 2	ISC izx 8	CPX zp 3	SBC zp 3	INC zp 5	ISC zp 5	INX 2	SBC imm 2	NOP 2	SBC imm 2	CPX abs 4	SBC abs 4	INC abs 6	ISC abs 6
<b>Fx</b>	BEQ rel 2*	SBC izy 5*	KIL	ISC izy 8	NOP zpx 4	SBC zpx 4	INC zpx 6	ISC zpx 6	SED 2	SBC aby 4*	NOP 2	ISC aby 7	NOP abx 4*	SBC abx 4*	INC abx 7	ISC abx 7

3.2. ábra. A 6502 opkódmátrixa

Ha meg szeretnénk találni egy adott opkódhoz a hozzá tartozó információt, akkor szükségünk lesz az opkód hexadecimális alakjára, ami legfeljebb kétszámjegyű lehet. A nagyobb helyértékű számjegy a keresett cella sorát, a kisebbik pedig az oszlopát

írja le. Példaként a 3.2 ábrán láthatjuk, hogy a \$30 opkódhoz a BMI utasítás tartozik relatív címzéssel. Azok az opkódok csillaggal vannak jelölve, amiknek a futási ideje megnőhet az aktuális argumentumuktól függően. A szürkével jelölt opkódokhoz hivatalosan nincs utasítás rendelve. Ezeket a nem dokumentált „illegális” [14] opkódokat a processzor későbbi verzióinak hagyták fent. A tervezők nem tiltották meg azonban ezeknek a használatát, egyszerűen csak nem definiálták a viselkedésüket. Ennek ellenére több olyan illegális opkód is belekerült a dizájnba, ami később hasznosnak bizonyult. A játékfejlesztők próbálkozások útján felfedezték, hogy melyek azok az opkódok, amiknek a viselkedése determinisztikus és néhány speciális feladat esetén érdemes őket használni. Ritka ugyan, de van olyan játék, ami ezeket az opkódokat is használja, ezért ezeknek az opkódoknak az emulációját is megvalósítottam.

### 3.3.2. Regiszterek

A regiszter a processzor leggyorsabban elérhető memóriája. A gyártási költségek alacsonyan tartása végett csak 6 regiszter került a processzorba [2]. minden regiszter mellett zárójelben fel van tüntetve annak mérete bitekben megadva.

**A (8):** Akkumulátor, az aritmetikai műveletek eredményei ebbe kerülnek.

**X (8) és Y (8):** Index regiszterek, indirekt címzésnél használjuk őket. Ciklusok esetén a ciklusváltozót érdemes ezekben tárolnunk.

**S (8):** Verem mutató. A verem tetejének a kezdőcímtől vett eltolását tárolja.

**P (8):** Státusz regiszter, ami 7 darab flag bitet tárol.

**PC (16):** Programszámláló. A következő opkód memóriacímét tárolja. Méretéből következik, hogy a processzor teljes címtartománya 64 KiB nagyságú.

### 3.3.3. Memórialap

Az  $i$ . lap egy 256 bájtos egybefüggő memóriarész, ami a  $[i \cdot \$100, (i + 1) \cdot \$100)$  címtartományon helyezkedik el.

### 3.3.4. Hívási verem

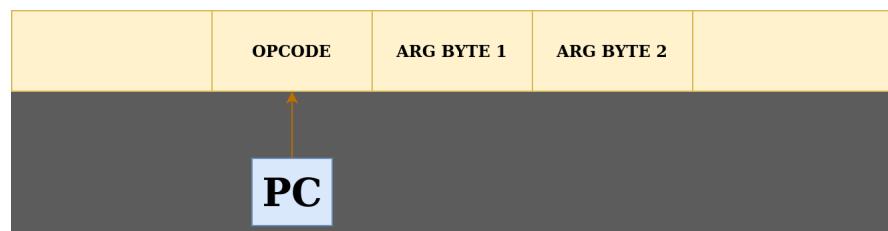
Az egymásba ágyazott eljárásokat a processzor hardveresen támogatja, amihez a vermet használja [2]. A verem az 1. lapon található, és a kisebb címek felé nő. Eljárás

hívásakor a verem tetejére kerül az aktuális programszámláló értéke, visszatéréskor pedig a veremről levett címre állítjuk be a programszámláló értékét [4].

### 3.3.5. Megszakítás

A komponensek kommunikációjának egyik módja a hardveres megszakítás. A 6502 chip egy darab maszkolható (*IRQ*) és egy nem maszkolható (*NMI*) megszakítással rendelkezik [2]. A megszakítási vektorokkal a program beállíthatja, hogy egy bizonyos megszakításra milyen szubrutinnal kíván reagálni. A vektorok 2 bájtos tárolók, amik a kezelő szubrutin címét tartalmazzák. Az NMI-hez tartozó vektor a \$FFFA és a \$FFFB címeken, míg az IRQ-hoz tarozó vektor az \$FFFE és a \$FFFF címeken található [7]. A 6502 „kicsi az elején” bájtsorrendű (little-endian) processzor, ezért a címeknek minden az alsó 8 bitjét tárolja az alacsonyabb címen, a felső 8 bitjét pedig a magasabb címen. A program dönthet úgy, hogy a maszkolható megszakítást figyelmen kívül hagyja (a kezelő szubrutin nem hívódik meg), ehhez az *IRQ Disable* flag-et be kell állítania a státusz regiszterben. A nem maszkolható megszakítás esetén erre nincsen lehetőség, a végrehajtás mindenkorban a kezelő szubrutinhoz ugrik.

### 3.3.6. Opkód argumentum helye, fajtája



3.3. ábra. Opkód argumentumainak helye

A 3.3 ábra szemlélteti, hogy az opkód argumentumok közvetlenül az opkód mögött, a  $PC+1$  és  $PC+2$  címeken helyezkedhetnek el a memóriában. Címzési módtól függően változhat az argumentumbájtok száma 0, 1 és 2 darab között. Ha az opkód rendelkezik argumentummal vagy argumentumokkal, akkor azok a következő fajtájúak lehetnek:

- 2 bájt, ami abszolút memóriacímet ábrázol kicsi-az-elején bájtsorrenddel
- 1 bájt, ami relatív eltolást ír le

- 1 bájt, ami a művelet közvetlen operandusa

### 3.3.7. Címzési módok

Egy opkód után azoknál a címzési módoknál áll argumentum, amiknél a művelet elvégzéséhez szükséges operandus nem regiszterben, hanem a memóriában van. A címzési módok [2, 6] azt határozzák meg, hogy az argumentumból hogyan kell kiszámolni az operandus effektív 16 bites memóriacímét. Az alábbiakban felsorolt címzési módoknál a zárójelben az opkódmátrixbeli név (amennyiben van) és az argumentumbájtok száma található.

**Akkumulátor mód (0):** nincs argumentum, az utasítás az **A** regiszter értékét módosítja.

**Azonnali mód (imm, 1):** az utasítás operandusa maga az argumentum. Jele: #  
Példa: az LDA #\$0 utasítás nullára állítja az **A** regisztert.

**Abszolút mód (abs, 2):** az argumentum az operandus effektív címe.

**0. lap mód (zp, 1):** A CPU kevés regiszterét azzal ellensúlyozták, hogy ennek a speciális módnak köszönhetően a nulladik lapot hatékonyabban lehet címezni, mint a többöt. Mivel a 0. lap mérete 256 bájt, ezért teljes cím helyett elég egyetlen bájt a címzéséhez. A kisebb paraméter gyorsabban beolvasható és egyúttal a kódmeretet is csökkenti.

**Indexelt 0. lap mód (zpx, zpy, 1):** Hasonlóan most is csak a 0. lapot tudjuk címezni, de az argumentumhoz hozzáadjuk valamelyik index regiszter értékét.  
Az operandus címének kiszámítása:  $(arg1 + index) \bmod 256$

**Indexelt abszolút mód (abx, aby, 2):** Az argumentumok egy teljes memóriacímét alkotnak, amihez hozzáadjuk a megadott index regiszter értékét.

**Implicit mód (0):** nincs szükség argumentumra, mert az utasítás regiszterekkel dolgozik.

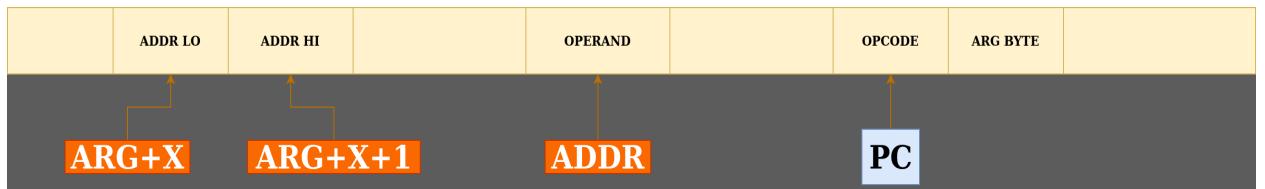
**Relatív mód (rel, 1):** Az elágazási utasítások használják ezt a címzési módot.  
Elágazásoknál, ha a feltétel teljesül, akkor az argumentummal el kell tolni a

programszámlálót. Az elágazási utasítások feltételeit lásd az *Utasításkészlet* szekcióban.

**Indirekt mód (ind, 2):** Erre a címzési módra a JMP utasításnál van szükség. A két argumentum bájt együtt egy teljes memóriacímet alkot, legyen ez  $m$ . Az operandus címét az  $m$  és  $m+1$  címeken találjuk, így tehát az operandus értékét a következő módon kapjuk meg:

$$\text{operand} := \text{read}( (\text{read}(m + 1) << 8) \mid \text{read}(m) )$$

**Indexelt indirekt mód (izx, 1):** Az X regisztert összeadjuk az argumentummal, így egy 0. lapon található címet kapunk. Erről a címről kell az operandus effektív címét kiolvasni.



3.4. ábra. Indexelt indirekt címzés

**Indirekt indexelt mód (izy, 1):** Az argumentum egy 0. lapon található címre mutat, amit ha összeadunk az Y regiszter értékével, akkor megkapjuk az operandus effektív címét.

Az abx, aby és izy indexelt címzési módok és bizonyos utasítások kombinációjánál előfordulhat, hogy a véleges operandus cím és az indexelés előtt álló cím különböző memórialapra esik. Ebben az esetben 1 órajelciklussal tovább tart az utasítás végrehajtása. Emellett az elágazási utasítások (relatív címzés) több ciklus alatt fejeződnek be, ha az ugrási feltétel teljesül. Ha az ugrás előtti PC értéke és az ugrási cél más lapra esik, akkor +2, ellenkező esetben csak +1 órajelciklussal kell számolni.

### 3.3.8. Memóriatérkép

A memóriatérkép [7] leírja a címtér felosztását a komponensek között. A memóriatérképből meg tudjuk állapítani, hogy egy adott címen található bájt kiolvasásához vagy írásához melyik komponens hardveres logikáját kell alkalmazni.

Tartomány	Eszköz
\$0000 – \$07FF	CPU RAM
\$0800 – \$1FFF	CPU RAM tükrözése
\$2000 – \$2007	PPU regiszterek
\$2008 – \$3FFF	PPU regiszterek tükrözése
\$4000 – \$4017	APU és IO regiszterek
\$4018 – \$401F	APU és IO regiszterek tükrözése
\$4020 – \$FFFF	Kazetta

### 3.3.9. Memóriatükrözés

Memóriatükrözésről beszélünk, amikor fizikailag ugyanazt a memóriaterületet több memóriacímről is el tudjuk érni [8]. Ha egy címtartomány  $x$  bájtonként tükrözve van, akkor minden olyan  $a$  és  $b$  memóriacím ugyanarra a bájtra mutat, ami a tartományba vagy a tükrözésébe esik és  $a \equiv b \pmod{x}$ . A CPU RAM 2 KiB-onként tükrözve van, amiből következik, hogy a \$0001, \$0801 \$1001, \$1801 címek ugyanarra a bájtra mutatnak. Egy címtartomány tükrözése lehet nagyobb mint az eredeti tartomány (például a CPU RAM esetében a tükrözési tartomány háromszor nagyobb).

### 3.3.10. Státusz flagek

0. bit: C = Carry

Összeadások, forgatások, eltolások során a leeső biteket jelzésére.

1. bit: Z = Zero

Aritmetikai művelet eredménye vagy mozgatott bájt 0.

2. bit: I = IRQ Disable

Az 1-re állításával maszkoljuk az IRQ-t.

3. bit: D = Decimal mode

A NES nem használja ezt a flag-et.

4. bit: B = BRK Command

Az 1-re állításával generálhatunk egy szoftveres megszakítást.

6. bit: V = Overflow

Aritmetikai műveleteknél a túlcordulás vagy alulcsordulás jelzésére. A BIT utasítás speciálisan használja.

7. bit: N = Negative

A manipulált bájt negatív, vagyis a legnagyobb helyiértékű bitje 1.

### 3.3.11. Utasításkészlet

A processzor hivatalos utasításkészlete [2, 3, 4]:

Aritmetikai és logikai egység (ALU)

**ADC:** Összeadás; **SBC:** Kivonás; **AND:** Logikai ÉS művelet; **ASL:** Bájt balra elcsúsztatása; **LSR:** Bájt jobbra elcsúsztatása; **ORA:** Logikai VAGY; **EOR:** Kizárasos VAGY; **INC, INX,INY:** növelés eggel; **DEC, DEX,** **DEY:** csökkentés eggel; **ROL, ROR:** bájt forgatása;

Összehasonlítás, tesztelés

**CMP, CPX, CPY:**

A megadott címen található bájt és rendre az A, X és Y regiszterekben található bájt között az = és a  $>=$  reláció kiértékelése

**BIT:** Az operandus bájt maszkolása ( $\&$ ) az A regiszter tartalmával, az eredmény szerint a Z,N,V flag-ek frissítése

Veremműveletek

**PHA:** Az akkumulátor regiszter értékének felrakása a veremre

**PHP:** A státusz regiszter értékének felrakása a veremre

**PHA:** Az akkumulátor regiszter új értékének levétele a veremről

**PHP:** A státusz regiszter új értékének levétele a veremről

Vezérlés

**JMP:** Vezérlés áthelyezése egy megadott memóriacímhez, avagy a programszámláló átállítása erre a címre

**JSR:** Szubrutin hívás (visszatérési cím elmentése + **JMP**)

**RTS:** Visszatérés szubrutinból (visszatérési cím kiolvasása + **JMP**)

**BRK:** Szoftveresen generált megszakítás

**RTI:** Visszatérés megszakításkezelőből

Flag manipuláció (a utasításnév harmadik betűje a változtatott flag-et jelöli)

**CLC, CLD, CLI, CLV, SED, SEC, SEI**

(C = Clear, S = Set)

Elágazások: vezérlés áthelyezése akkor, ha teljesül a feltétel az utasítás által vizsgált státusz flag-re

**BCC(C=0), BCS(C=1), BNE(Z=0), BEQ(Z=1), BPL(N=0),  
BMI(N=1), BVC(V=0), BVS(V=1)**

Bájtok mozgatása regiszterek és a memória között

**LDA, LDX, LDY, STA, STX, STY**

(L = Load, S = Store)

A név harmadik betűje a használt regisztert jelöli.

Bájtok mozgatása regiszterek között

**TAX, TAY, TSX, TXA, TXS, TYA**

A név második betűje a forrásregisztert, a harmadik a céltregisztert jelöli.

## 3.4. Kazetták

A kazettákon logikai szempontból kétfajta memória található [5]: PRG és CHR. PRG-ből lehet ROM és RAM is a kazettán. Ezekben a processzor által végrehajtandó utasítások, vagy a játéklogika kapott helyet. A CHR fajtáját tekintve ROM vagy RAM memória, amely a játék sprite-jait tárolja, amik 8\*8 pixeles kis képekből állnak. Ezeket a kis képeket ezentúl alakzatoknak fogom hívni.

### 3.4.1. Az iNES fájlformátum

Az iNES fájlformátumot egy korai NES emulátor vezette be a NES játékok bináris formában történő terjesztésére [5]. A fájl elején található 16 bájtos fejléc többek között a mapper konfiguráció azonosítóját, a képfeldolgozó névtábláinak tükrözési módját, valamint a PRG ROM, PRG RAM és a CHR méretét határozza meg [17]. A fejléc után ezek tartalma található.

## 3.5. A képfeldolgozó egység

### 3.5.1. Színpaletta

A színpaletta [9] hozzárendel egy \$0 és \$3F közötti azonosítóhoz egy színkódot. A NES csak a színpaletta színeit tudja megjeleníteni, így összesen 55 szín áll rendelkezésre (vegyük észre, hogy a fekete szín több azonosítóhoz is hozzá van rendelve). Egy pixel színét az fogja eldöntheti, hogy milyen azonosító tartozik hozzá. A színpaletta hozzárendeléseit futás közben nem változtathatjuk meg, mert azok a hardverbe vannak „égetve”.



3.5. ábra. A 2C02 színpalettája

### 3.5.2. Paletta indexek

A \$3F00 – \$3F1F címtartományon található paletta index egy memóriacím → azonosító hozzárendelést ír le [9], azonban ezt a program már változtathatja futás során, mert RAM típusú memóriáról van szó. A paletta index 4 bájtos csoportokra, úgynevezett palettákra van felosztva. A paletták bájtjai színpalettabeli azonosítókként szolgálnak. Emiatt közvetett módon ugyan, de ki tudjuk számolni egy pixel színét, ha tudjuk, hogy melyik palettát és azon belül hanyadik azonosítót kell használnunk. Jelentős limitáció, hogy a paletták 4. színe nem változtatható, mert ezek mindenig a \$3F00 címet (a háttérszínt) tükrözik.

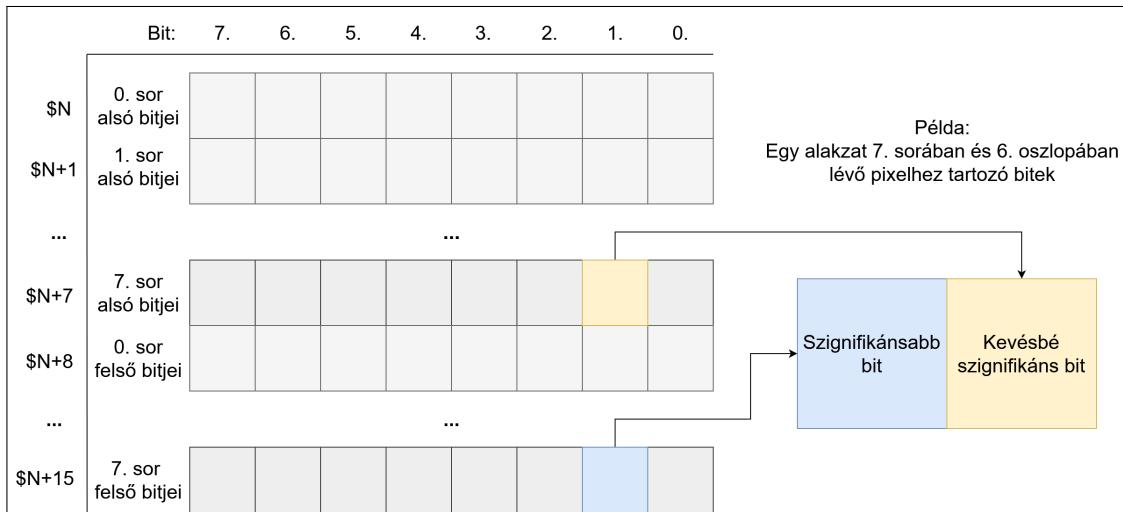
Tartomány	Paletta
\$3F00	Univerzális háttérszín
\$3F01 – \$3F03	0. háttér paletta
\$3F05 – \$3F07	1. háttér paletta
\$3F09 – \$3F0B	2. háttér paletta
\$3F0D – \$3F0F	3. háttér paletta
\$3F11 – \$3F13	0. sprite paletta
\$3F15 – \$3F17	1. sprite paletta
\$3F19 – \$3F1B	2. sprite paletta
\$3F1D – \$3F1F	3. sprite paletta

3.1. táblázat. A paletta indexek címei

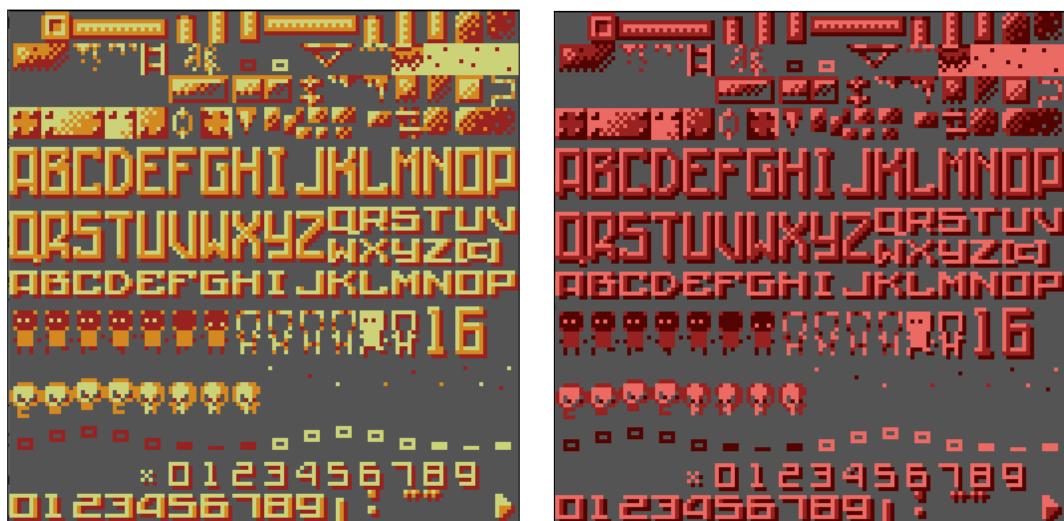
### 3.5.3. Alakzattáblázat

A CHR memóriában található két alakzattáblázat tárolja az alakzatokat egy speciális formátumban [9]. Ha az alakzatokat úgy reprezentálnánk, hogy az alakzat minden pixeléhez eltárolnánk egy színkódot, akkor pixelenként 6 bitre lenne szükségünk (mivel 55 színkóból választhatunk). Ehelyett pixelenként csak 2 bitet (egy szignifikáns és egy kevésbé szignifikáns bitet) tárolunk, amik együtt egy 0 és 3 közé eső sorszámot reprezentálnak. Ez a sorszám azt mondja meg nekünk, hogy valamelyik palettán belül hanyadik színpaletta-azonosítót használjuk a színereséshez. A paletta nincs előre meghatározva a CHR-ben, a dinamikusan módosítható attribútumtábla segítségével fogjuk később meghatározni, hogy a vizsgált pixelhez melyik paletta tartozik. Mivel a paletta index írható és olvasható memória is, így a program futási időben, dinamikusan változtathatja, hogy egy paletta milyen azonosítókat tartalmaz, ezáltal pár lépésben átszínezheti az összes alakzatot, ami az adott palettát használja. Egy alakzat  $8 \times 8$  pixeles, pixeljeinek bitjeit  $(8 \cdot 8 \cdot 2) \div 8 = 16$  egymást követő bajt tárolja a 3.6 ábrán szemléltetett módon. Először 8 bájton keresztül az alakzat sorainak kevésbé szignifikáns bitjei, majd ezután újabb 8 bájton át a szignifikánsabb bitek sorakoznak. Az oszlopok és bitek sorszámozása fel van cserélve, tehát az  $i$ . oszlophoz egy bájton belül a  $7 - i$ . bit tartozik. Egy alakzattáblázat  $16 \times 16$  darab alakzatot tartalmaz, ebből adódóan a táblázat teljes mérete  $16 \cdot 16 \cdot 16 = 4096$

bájt.



3.6. ábra. Egy alakzat reprezentálása



3.7. ábra. Az Alter Ego játék 0. alakzattáblázata különböző palettákkal kirajzolva

### 3.5.4. Rétegek

A képfeldolgozó két réteget képes kezelní hardveresen, ezek a háttér és a sprite rétegek [9]. Általában a képkocka azon részei tartoznak a háttérhez, amik ritkán változnak (például feliratok, számlálók, mozdulatlan képek) és a kirajzolásukhoz nem szükséges további transzformáció (eltolás, tükrözés). A háttérben az alakzatok 30 sorból és 32 oszloból álló négyzetrácsot alkotnak (ebből és az alakzatok méretéből ered a NES 256\*240 pixeles felbontása). A háttér rétegben egyesével nem

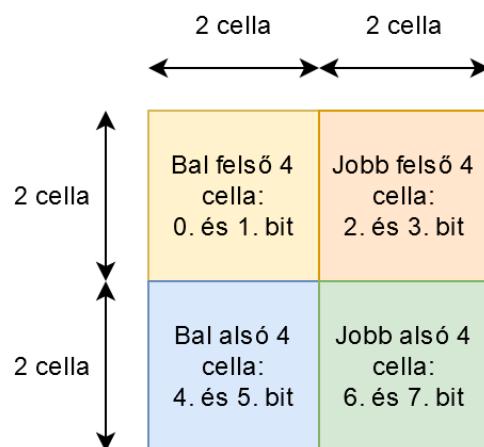
lehet alakzatokat eltolni, csak az egész háttér eltolása lehetséges. A sprite rétegen több lehetőség áll rendelkezésre, ugyanis itt egyenként, pixelpontosságú eltolással, valamint horizontális és/vagy vertikális tükrözéssel rajzolhatjuk ki az alakzatokat. A sprite rétegnél legfeljebb 64 alakzat szerepelhet egy képkockán (ez a megkötés a később ismertetett OAM méretéből adódik).

### 3.5.5. Névtáblázatok

A háttérréteg négyzetrácsának elrendezését a névtáblázatok tárolják [9]. Egy névtáblázat a háttér 960 darab alakzatcellájának mindegyikéhez eltárolja a cellába rajzolandó alakzat sorszámát. Ez a sorszám relatív, ugyanis minden az alaktáblázaton belül értendő, amit a CONTROLLER regiszterrel a program kiválaszt.

### 3.5.6. Attribútumtáblázatok

Minden névtáblázathoz tartozik egy 64 bájtos attribútumtáblázat [9], ami a névtáblázat 960 darab bájtja után következik. Az attribútumtáblázatban minden bájt egy  $4 \times 4$  alakzatcellából álló terület palettáit határozza meg. A bájtok 4 darab 2 bites részre vannak felosztva, ahol mindenek között 2 bites rész egy  $2 \times 2$  cellából álló terület palettájának sorszámát kódolja el (ugyan 8 paletta van, de ebből csak 4 háttérpaletta, így elég 2 bit). Ennek a reprezentációnak a következménye, hogy a háttér 4 cellából álló csoportjai mindenkorban egy palettán osztoznak. A 16 cellát lefedő bájt felosztása a 4 cellás területek között a 3.8 ábrán van szemléltetve.



3.8. ábra. Attribútumbájtok felosztása

### 3.5.7. OAM (Object Attribute Memory)

Az OAM [9] a sprite réteg elrendezését leíró 256 bájtos memória. 64 darab alakzatról tárol információt, amik a következők:

- X és Y koordináta
- A sprite réteg aktív alakzattáblázatán belüli sorszám
- Tükrözés
- Paletta sorszám
- Prioritás a háttérrel szemben

### 3.5.8. Regiszterek

A CPU és a PPU az alább látható 9 darab egy bájtos regiszter segítségével tud egymással kommunikálni. [9] A regisztereket a CPU a zárójelekben található címeken tudja elérni. Az olvasató regiszterek **R**, az írhatók **W** betűvel vannak megjelölve.

#### CONTROLLER(\$2000, W):

A kirajzolás vezérlésére szolgáló regiszter. Beállítható vele a következő képkockánál használandó táblázatok indexe.

- 0-1. bit:** Aktív névtáblázat indexe
- 2. bit:** VRAM cím inkrementálási mód
- 3. bit:** Aktív alakzattáblázat indexe a sprite rétegnél
- 4. bit:** Aktív alakzattáblázat indexe a háttér rétegnél
- 5. bit:** Sprite méret (8x8 vagy 8x16 pixel)
- 6. bit:** Az emuláció során nem használt bit
- 7. bit:** NMI generálása a PPU tétlen periódusának kezdetén

#### MASK(\$2001, W):

A rétegek egyenkénti ki- és bekapcsolása és speciális effektusok (például szürkeárnyalat) vezérelhetők vele.

#### STATUS(\$2002, R):

A kirajzolás alatt bekövetkező eseményeket jelzi a PPU a CPU-nak ezzel a regiszterrel. Ilyen esemény például a sprite-túlcordulás, ami akkor áll fent, ha több, mint a 8 alakzat kerülne egy sorra a sprite rétegben.

### OAMADDR(\$2003, W) és OAMDATA(\$2004, R/W):

A processzor ezen két regiszter segítségével képes új adatokkal feltölteni az OAM memóriát. Az alábbi kód részlet azt szemlélteti, hogy az *adatok* tömb tartalmát hogyan kell a CPU memóriájából az OAM-ba átmásolni egy megadott címtől kezdve. Az OAMDATA írása után az OAMADDR automatikusan inkrementálódik a másolás gyorsításának érdekében.

```
OAMADDR := 8 bites OAM cím
for i in 1..adatok.hossz
    OAMDATA := adatok[i]
```

### PPUSCROLL(\$2005, W):

Beállíthatjuk vele, hogy a hátteret hány pixellel szeretnénk arrébb csúsztatni (horizontális tükrözésnél vízszintesen, vertikális tükrözésnél függőlegesen).

### PPUADDR(\$2006, W) és PPUDATA(\$2007, R/W):

A névtáblák frissítésére szolgálnak. Hasonlóan kell őket használni, mint az OAMADDR és OAMDATA regisztereiket.

### OAMDMA(\$4014, W):

Az OAM memória frissítésének egy alternatív, gyorsabb módja a Direct Memory Access (DMA). Ekkor a processzorban található dedikált hardver másolja át az adatokat egyenesen a CPU RAM-ból az OAM memóriába. A másolás megkezdéséhez annak a memórialapnak a sorszámát kell beírni a regiszterbe, ahol az átmásolandó adatok találhatók.

### 3.5.9. Memóriatérkép

A PPU memóriatérképe [9]:

Tartomány	Paletta
\$0000 – \$0FFF	0. Alakzattáblázat
\$1000 – \$1FFF	1. Alakzattáblázat
\$2000 – \$23FF	0. Névtáblázat
\$2400 – \$27FF	1. Névtáblázat
\$2800 – \$2BFF	2. Névtáblázat
\$2C00 – \$2FFF	3. Névtáblázat
\$3000 – \$3EFF	A 0-3. névtáblázatok tükrözése
\$3F00 – \$3F1F	Paletta indexek
\$3F20 – \$3FFF	Paletta indexek tükrözése

3.2. táblázat. A képfeldolgozó memóriatérképe

### 3.5.10. A háttér kirajzolásának egyszerű algoritmusa

Ezen a ponton már minden részletet ismerünk ahhoz, hogy megérthessük a háttérkirajzolás logikáját. A következő pszeudokód azt szemlélteti, hogy a fent említett táblázatokat és a paletta indexet hogyan kell együtt használni a háttér kiszámolásához. Az egyszerűség kedvéért a háttéreltolást itt nem veszem figyelembe. Sajnos ez az algoritmus ebben a formában emulációra nem alkalmas, mert nehézzé teszi a processzor párhuzamos, megfelelő szinkronizációval történő futtatását.

```
// Egy bájt indexedik bitjének kiolvasása (0/1)
byte bit(byte bájt, int index)
begin
    return (bájt >> index) & 1
end

// Az alábbi függvényel olvassuk a PPU memóriáját (lásd PPU memóriatérkép)
byte olvas(cím)

type RGB_Kód = (byte, byte, byte)

// Palettaszámából és azonosítósorszámából a szín kikeresése (emlékeztető: $3F00 a
paletta index kezdőcíme)
```

```

RGB_Kód színKeresés( byte palettaSorszám, byte azonosítóSorszám)
begin
    byte színAzonosító := olvas($3F00 + palettaSorszám*4 + azonosítóSorszám)
    return színPaletta[színAzonosító]
end

// Az eredményül kapott pixeleket tároló kétdimenziós tömb
RGB_Kód pixelek[256][240]

procedure HáttérKirajzol
begin
    // A CONTROLLER regiszterrel a választott névtáblázat
    // és alakzattáblázat kezdőcímének meghatározása
    cím aktívNévtáblázat := $2000 + (CONTROLLER & 0b11) * $400
    cím aktívAlakzatTáblázat := bit(CONTROLLER, 4) * $1000

    // Végigiterálás a háttér celláin
    for cellaSor in 0..29
        for cellaOszlop in 0..31
            begin
                // A cellához tartozó névtáblabájt indexének kiszámolása
                byte NTB_Eltolás := cellaSor * 32 + cellaOszlop

                // A névtáblabájt kiolvasása
                byte NTB := olvas(aktívNévtáblázat + NTB_Eltolás)

                // A cellához tartozó attribútumbájt eltolása a névtábla
                // kezdőcíméhez viszonyítva.
                cím ATB_Eltolás :=
                    // Átlépjük a 960 névtáblabájtot
                    $3C0 +
                    // Átlépjük az előző cellasorok attribútumbájait
                    // Emlékeztető: egy sorban 32 alakzat van amik
                    // négyesével osztóznak a bájton
                    (cellaSor div 4) * 8 +
                    // Átlépjük a jelenlegi cellasorban az előző attribútumbájtokat
                    (cellaOszlop div 4)

                // Az attribútumbájt kiolvasása
                byte ATB := olvas(aktívNévtáblázat + ATB_Eltolás)

                // Az attribútumbájtnak a cellához tartozó 2 bites része lesz a palettaszám.
                // Ki kell számolni, hogy a cella melyik (bal felső, jobb felső, stb.)
                // kvadránsába esik a bájt által lefedett 4*4-es területnek, ugyanis így
                // kapjuk meg,
                // hogy a bájt melyik 2 bitjére van szükségünk

```

```

byte kvadráns := (cellaSor & 0b10)*2 + (cellaOszlop & 0b10)
byte palettaSorszám := (ATB >> kvadráns) & 0b11

// Az alakzat kezdőcíme az alakzattáblában
// Segítség: egy alakzat 16 bájtot foglal
cím alakzatCím := aktívAlakzatTáblázat + NTB*16

// Végigiterálás a cella 8*8 pixeles területén
for pixelSor in 0..7
begin
    cím alakzatSor := alakzatCím + pixelSor

    // A sor alsó bitjei
    byte alakzatLSB := olvas(alakzatSor)

    // A sor felső bitjei
    byte alakzatMSB := olvas(alakzatSor + 8)

    for pixelOszlop 0..7
    begin
        // A pixelhez tartozó 2 bittel a palettán belüli szín meghatározása
        byte azonosítóSorszám :=
            (bit(alakzatMSB, 7-pixelOszlop) << 1) | bit(alakzatLSB, 7-pixelOszlop)

        // A pixel X koordinátája a képernyón (0–255)
        byte X := cellaOszlop * 8 + pixelOszlop

        // A pixel Y koordinátája a képernyón (0–239)
        byte Y := cellaSor * 8 + pixelSor

        // Az RGB kód kikeresése és beállítása
        pixelek[X][Y] := színKeresés(palettaSorszám, azonosítóSorszám)
    end
end
end
end

```

### 3.5.11. Sprite réteg kirajzolása

A kirajzolás algoritmusában annyiban változik, hogy a névtáblázatok és attribútumtáblázatok helyett az OAM memóriát használva határozzuk meg az alakzatsorszámokat és palettaszámokat. A 240 sor mindenkorán a kirajzolás megkezdése előtt ki kell értékelni, hogy melyek azok az alakzatok, amik a következő soron láthatóak. Ezeknek az adatait egy pufferbe, u.n. másodlagos OAM-be kell helyezni (legfeljebb 8 OAM-bejegyzés fér bele). minden pixelnél megnézzük, hogy van-e olyan alakzat

a másodlagos OAM-ban, ami arra a pixelre esik. Ha több is van, akkor a kisebb indexű alakzat élvez nagyobb prioritást.

## 4. fejezet

# Fejlesztői dokumentáció II: Megvalósítás

### 4.1. A feladat specifikációja

A feladat a NES játékkonzol CPU-ját, PPU-ját és kazettáit valós időben emulálni képes szoftver implementálása, ahol a játékos a játék irányítása mellett képes a virtuális gép állapotának elmentésére és betöltésére, valamint az emuláció vezérlésére. A játékokat billentyűzettel vagy kontrollerrel lehet irányítani. A program felhasználói felülettel is rendelkezik, amivel szintén tudunk mentéseket kezelní és szüneteltethetjük az emulációt.

### 4.2. Fejlesztői környezet

A programot Haskell nyelven írtam és a GHC 8.10.1-es verziójával fordítottam. Szerkesztőprogramként Visual Studio Code-ot használtam. A Haskell-ben írt fügőségi könyvtárak telepítéséhez a Stack<sup>6</sup> eszközt használtam. Az SDL2 és a GTK könyvtárakat Linux-on a disztribúció csomagkezelőjével, Windows-on az MSYS2 konzol csomagkezelőjével telepítettem.

---

<sup>6</sup>Haskell Tool Stack: <https://docs.haskellstack.org/en/stable/README/>

## Csomagok telepítése

- Windows (MSYS2 konzollal):

```
pacman -S mingw-w64-x86_64-SDL2  
pacman -S mingw-w64-x86_64-gtk3
```

- Linux:

```
sudo apt-get install libgirepository1.0-dev  
sudo apt-get install libwebkit2gtk-4.0-dev  
sudo apt-get install libgtksourceview-3.0-dev  
sudo apt-get install libsdl2-dev
```

## Fordítás

- Windows: A fordításnál be kell állítani az alábbi környezeti változókat, hogy az MSYS2-vel telepített könyvtárakról tudjon a Stack. A lenti példában az alapértelmezett MSYS2 útvonalakat állítom be.

```
SET PATH=C:\msys64\mingw64\bin;C:\msys64\usr\bin;%PATH%  
SET PKG_CONFIG_PATH=C:\msys64\mingw64\lib\pkgconfig  
SET XDG_DATA_DIRS=C:\msys64\mingw64\share  
stack build
```

Ha az első fordításnál hibát kapunk a *gi-\** függőségek telepítésénél, akkor másoljuk a *C:\msys64\mingw64\bin\zlib1.dll* fájlt a Stack által telepített GHC *mingw/bin* mappájába. Ha már van ilyen fájl ott, akkor készítsünk róla másolatot, majd cseréljük le.<sup>7</sup>

- Linux:

```
stack build
```

## Futtatás

A programot a következő parancssal tudjuk elindítani:

```
stack exec pure-nes
```

---

<sup>7</sup>További információk: <https://github.com/haskell-gi/haskell-gi/wiki/Using-haskell-gi-in-Windows>

### 4.3. Az emulációhoz kapcsolódó modulok

Minden komponenshez (CPU, PPU, kazetták) tartozik egy *Memory* modul, ami definiálja a komponens emulációja során használt rekordokat. A *Serialization* modulok ezen rekordok mentését/betöltését tartalmazzák. A CPU és a PPU *Emulation* moduljai tartalmazzák a hozzájuk tartozó rekordokon végezhető műveleteket.

A *Monad* modul definiálja az emulációs monádot és az abban elérhető primitív műveleteket. A *MasterClock* modul összefogja a komponenseket és szinkronizálja azok emulációját. A *Controls* modul a NES sztenderd kontrollerének emulációját tartalmazza.

A *JoyControls* modul a fizikai kontrollerek kezelését végzi (újonnan csatlakoztatott kontroller és annak rezgőmotorjának inicializálása, gombnyomások megfeleltétele parancsoknak).

Az *AppResources* modulban található rekord az emulátor teljes állapotát tárolja (virtuális gép, shaderek, ablak, stb.). A *Window* modul az SDL2 multimédiakönyvtárat és az OpenGL-t használva megjeleníti az előállított képkockákat. A *CrtShader* modulban a CRT effektusért felelős OpenGL shaderek találhatók. A *Framerate* modul függvényeivel tudjuk szabályozni a képkockaszámot (fix vagy korlátozatlan számú képkocka/másodperc). A *Logic* modul fogadja az SDL-től érkező gombnyomás-eseményeket, feldolgozza a parancsokat és lépteti az emulációt.

A *Communication* modul definiálja azokat az eseményeket (*Event*), amiket az emulációs ablak a felhasználói felületnek küldhet (mentés/betöltés eredménye, megjelenítendő figyelmeztetés/hibaüzenet), illetve azokat a parancsokat (*Command*), amiket attól fogadhat (szüneteltetés, mentés/betöltés, leállítás, stb.).

### 4.4. A felhasználói felületnél használt technológiák

A GTK (GIMP Toolkit) egy nyílt forráskódú, kezelőfelületek tervezésére szolgáló szoftvercsomag. Az API-ja javarésztl imperatív stílusú, például a widget-ek létrehozását követően mellékhatásos függvényekkel tudjuk beállítani az attribútumokat és az eseménykezelőket. A Haskellben elérhető *gi-gtk-declarative*<sup>8</sup> könyvtár ezzel szemben egy deklaratív GTK API-t ad a kezünkbe, aminek segítségével röviden és tö-

---

<sup>8</sup><https://hackage.haskell.org/package/gi-gtk-declarative>

mören tudjuk összetett vezérlőelemek megjelenését leírni. A *gi-gtk-declarative-app-simple*<sup>9</sup> erre építve definiál felhasználói felületek vezérléséhez egy egyszerű architektúrát, ahol a felületet állapotgépnek tekintjük és az állapotátmeneteket események váltják ki. A felület létrehozásához elég a kezdőállapotot, az állapotmegjelenítő (*state* → *window*) függvényt és az állapotátmenet-függvényt definiálni.

## 4.5. A felhasználói felülethez kapcsolódó modulok

### A felhasználói felület moduljai

**State:** A felhasználói felület állapotait reprezentáló algebrai adattípus (*State*) definíciója található ebben a modulban. minden állapothoz tartoznak adatmezők, amik a megjelenítéshez és az állapotátmenetekhez szükséges információkat tárolják (lásd 4.4 ábra).

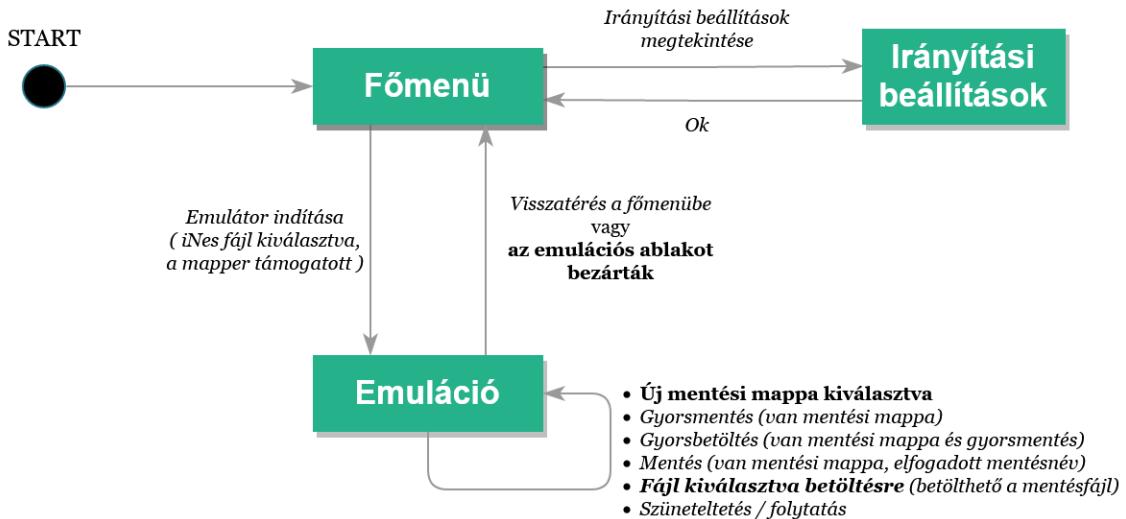
**InGame:** Az emulációs állapot megjelenítéséhez használt segédfüggvényt (*inGame*) tartalmazza, ami az állapot adatmezőiből összeállítja a deklaratív vezérlőleírást.

**Window:** Az *Üzenet*, *Irányítási beállítások* és *Főmenü* állapotok megjelenítéséhez használt segédfüggvényeket (*messageWidget*, *controlsWidget*, *startMenu*), valamint a segédfüggvények egyesítésével kapott függvényt (*visualize*) tartalmazza, ami minden állapotot meg tud jeleníteni.

**Logic (Main):** Az állapotátmenet-függvényt (*update*) tartalmazza, ami kezeli a felület vezérlőelemeitől vagy az emulációs ablaktól érkező eseményeket, azok hatására frissíti a jelenlegi állapotot, átmegy egy teljesen új állapotba vagy parancsot küld az emulációs ablaknak.

---

<sup>9</sup><https://hackage.haskell.org/package/gi-gtk-declarative-app-simple>



4.1. ábra. A felület állapotátmenetei

A 4.1 ábra a felület eseményeit és azoknak állapotváltoztató hatásait hivatott vizualizálni. A dőlt betűk részek gombnyomásokat, a kövér betűk eseményeket jelölnek. Ha egy eseménynél a zárójelben lévő feltétel nem teljesül, akkor egy negyedik, *Üzenet* nevű állapotba lépünk, ahol megjelenítjük a hibaüzenetet/figyelmeztetést. Innen az *Ok* gomb lenyomására visszatérünk az előző állapotba. A felület bármelyik állapotban bezárható, ekkor a teljes program terminál (az emulációs ablak is bezárul).

## 4.6. Az emulációt magába záró monád

A CPU, a PPU, valamint a teljes NES emulációjához érdemes egy monádot bevezetni, hogy ne kelljen mindenhol explicit paraméterként átadni a komponens rekordját. A ReaderT monádtranszformert miatt ez bármely ponton elérhető (olvasható) lesz. A *runEmulator* függvényteljesen lefuttathatjuk az emuláció mellékhatásait. Az *emulateSubcomponent* függvény célja, hogy a komponensek emulációját egyesítve lehetséges legyen a teljes rendszer emulációja.

```

import Control.Monad.Reader

newtype Emulator component value
  = Emulator (ReaderT component IO value)
  deriving
    (Functor, Applicative, Monad, MonadIO, MonadReader component)
  
```

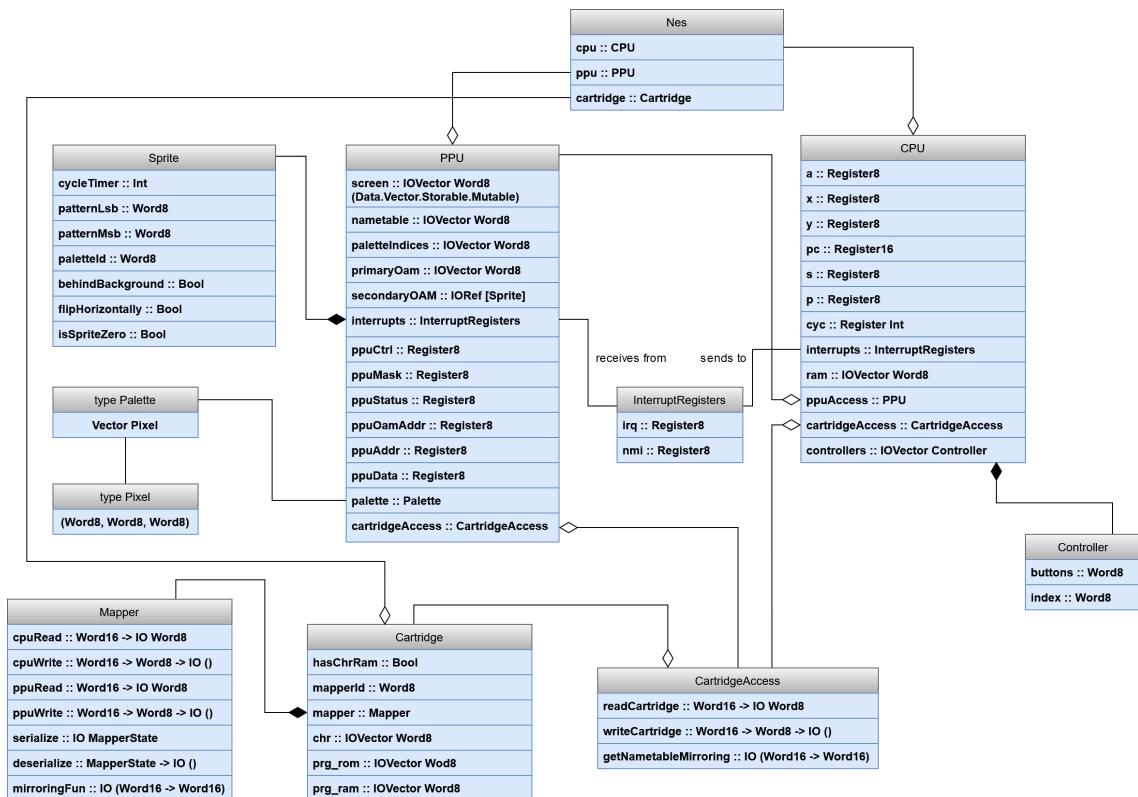
```

runEmulator :: c -> Emulator c v -> IO v
runEmulator component (Emulator effect)
= runReaderT effect component

emulateSubcomponent :: (c -> s) -> Emulator s v -> Emulator c v
emulateSubcomponent subcomponent (Emulator effect)
= Emulator (withReaderT subcomponent effect)
    
```

## 4.7. Adatreprézentáció

A teljesítmény fontos szempont volt, ezért a regisztereket változtatható és u.n. *unboxed* (közvetlenül az értéket tárolják és nem egy mutatót az értékre) referenciákkal ábrázoltam. Az egyes memóriarészleteket (CPU RAM, PRG, CHR, stb...) külön-külön, változtatható és unboxed vektorokkal reprezentálom.



4.2. ábra. Az emulációnál használt rekordok

Az emulátoromnál a megszakítások kezeléséhez segédregisztereket vezettem be, amik a valódi hardverben nincsenek jelen. A CPU és a PPU osztozik a megszakítási „regiszterekeken”, ezért a PPU megszakításokat tud küldeni a CPU-nak.

A kazetta memóriáját a *Cartridge* rekord tárolja. Ehhez közvetlenül nem tudunk hozzáférni, mert a mapper típusa szabja meg, hogy milyen logikájú a címfordítás. A CPU és a PPU a *CartridgeAccess* rekord függvényeit használva tudja olvasni és írni a kazettát, ami a háttérben a megfelelő típusú mapper olvasó és író eljárásait fogja meghívni. A hozzáférést minden két komponens személyre szabva kapja meg, így tehát a CPU esetében a *readCartridge* függvény a mapper *cpuRead* függvényének felel meg.

## 4.8. Mentés és betöltés

Ennek a funkciót a megvalósítása sokféleképp lehetséges. Én egy egyszerű megoldást választottam, ami abból állt, hogy azokhoz a rekordokhoz, amik változtatható referenciakat és vektorokat tartalmaznak, definiáltam egy változtathatatlan (*immutable*) rekordot. A tiszta rekordokhoz automatikusan generálni lehet serializáló műveleteket (erre a *cereal*<sup>10</sup> könyvtárat használtam), így tehát csak annyi dolgom volt, hogy rekordok közti konverziót implementáljam.

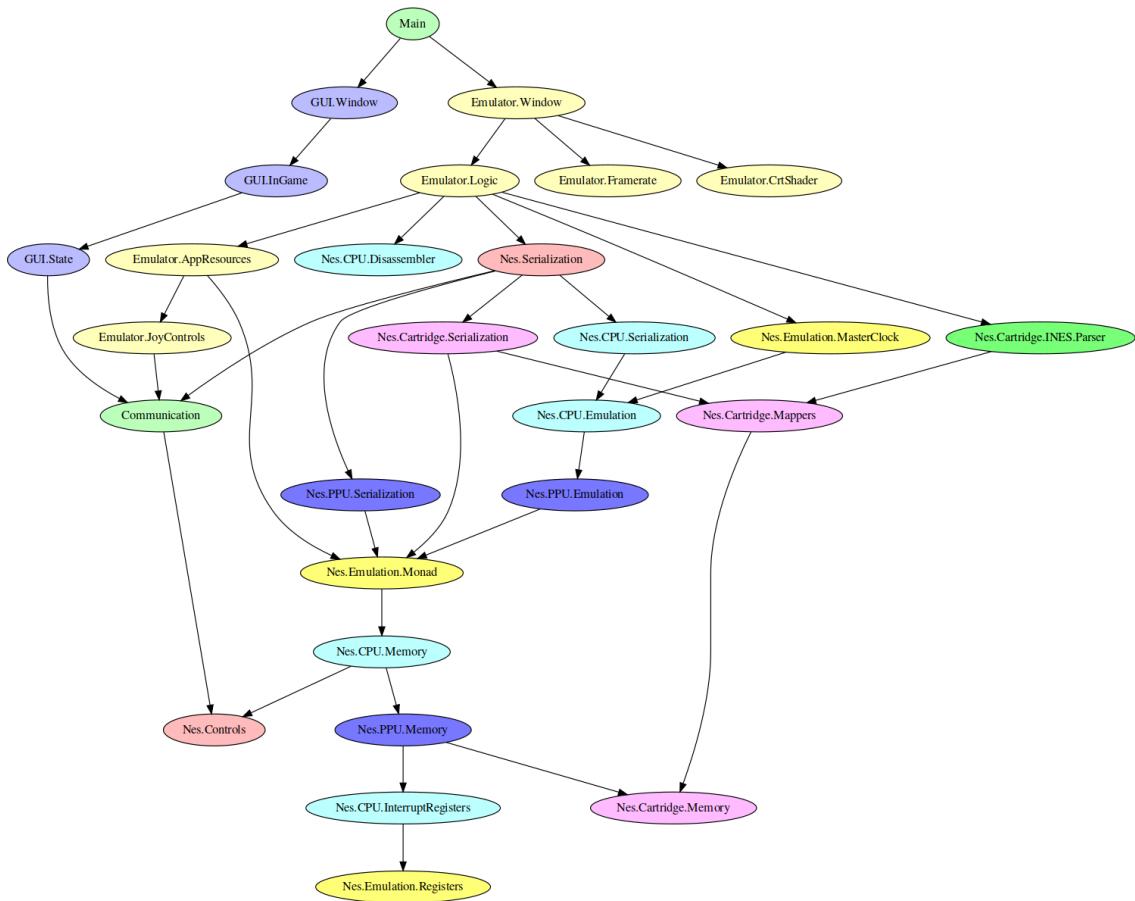
```
data SerializedPPU = SerializedPPU {  
    ...  
    secondaryOam      :: [Sprite],  
    ppuCtrl          :: Word8,  
    ppuMask          :: Word8,  
    ...  
} deriving (Generic, Serialize)
```

## 4.9. Fontos modulok részletes áttekintése

A program összes modulját és az azok közti függőségeket a 4.3 ábrán láthatjuk.

---

<sup>10</sup><https://hackage.haskell.org/package/cereal>



4.3. ábra. A program moduljainak függőségi gráfja

#### 4.9.1. Nes.Cartridge.Memory

A modul a Cartridge, Mapper és CartridgeAccess rekordok definícióját és a hozzájuk kapcsolódó függvényeket, valamint a névtáblázattükörözési függvényeket tartalmazza.

- **getCPUAccess :: Cartridge → CartridgeAccess**

Hozzáférés kérése a kazettához a CPU számára.

- **getPPUAccess :: Cartridge → CartridgeAccess**

Hozzáférés kérése a kazettához a PPU számára.

```
getPPUAccess :: Cartridge -> CartridgeAccess
getPPUAccess Cartridge{ mapper=Mapper{ppuRead, ppuWrite, mirroringFunction} }
= CartridgeAccess ppuRead ppuWrite mirroringFunction
```

### 4.9.2. Nes.Cartridge.INES.Parser

Műveletek a kazetta betöltéséhez az iNES fájlból.

- **iNESloader :: Get INES**

Az iNES fájlokhoz készített parser.

- **tryLoadingINES :: FilePath → IO INES**

Az iNES parser futtatása a megadott fájlon, hiba esetén kivétel dobása.

- **assembleCartridge :: INES → IO Cartridge**

Az iNES fájl tartalmából az emuláció során használt kazettareprezentáció előállítása.

- **loadCartridge :: FilePath → IO Cartridge**

Adott elérési útvonalról egy kazetta betöltése.

```
import Control.Monad (>=>)

loadCartridge = tryLoadingINES >=> assembleCartridge
```

### 4.9.3. Nes.Cartridge.Mappers

Ez a modul a mapper-ek implementációit tartalmazza. Ha új mapper-t szeretnénk hozzáadni az emulátorhoz, akkor elég pusztán itt definiálni számára egy konstruktort. A kazetta összeszerelésénél a mapper konstruktora megkapja a félkész kazettát és visszaadja az írásra/olvasásra szolgáló függvényeket tartalmazó rekordot, amik a megfelelő logikával fordítják át a címeket. A konstruktorok azért nem tiszta függvények, hogy tetszőleges belső állapotot létre lehessen hozni bennük. (a mapper-ek belső regisztereinek száma el szokott tért).

- **mappersById :: Map Word8 (Cartridge → IO Mapper)**

Azonosító → konstruktor hozzárendelés. Ha ehhez a Map-hez hozzáadunk egy új mapper konstruktort, akkor azt a mapper-t azonnal támogattnak érzékeli a betöltő logika.

- **nrom :: Cartridge → IO Mapper**

- **unrom** :: Cartridge → IO Mapper
- **cnrom** :: Cartridge → IO Mapper

#### 4.9.4. Nes.Emulation.Monad

Az emuláció során használt primitív műveletek.

- **powerUpNes** :: Cartridge → IO Nes  
„Behelyezzük” a kazettát és inicializáljuk a komponenseket.
- **useMemory** :: (comp → ref) → (ref → IO val) → Emulator comp val  
Kényelmi függvény arra, hogy a komponens rekordjának egy adott mezőjével műveletet végezzünk.

```
useMemory memory action = asks memory >= liftIO . action
```
- **readMemory** és **writeMemory** Komponens vektorral reprezentált memóriájának közvetlen olvasása és írása (a memóriatérkép nincs figyelembe véve).
- **readCartridgeWithAccessor** és **writeCartridgeWithAccessor** A kazettához a CPU-nak és a PPU-nak is saját hozzáférése van, amit ezekkel a függvényekkel tudnak használni.

```
readCartridgeWithAccessor ::  
(component -> CartridgeAccess) ->  
Word16 ->  
Emulator component Word8  
writeCartridgeWithAccessor ::  
(component -> CartridgeAccess) ->  
Word16 ->  
Word8 ->  
Emulator component ()
```

#### 4.9.5. Nes.CPU.Emulation

A 6502 teljes utasításkészletét tartalmazó modul. Néhány segédfüggvény, eljárás:

- **readReg** :: Prim a ⇒ (CPU → Register a) → Emulator CPU a  
A megadott regiszter olvasása. A regiszterek csak primitív, kicsomagolható típusokat tárolhatnak.

- **read :: Word16 → Emulator CPU Word8**

Olvasás a memóriából. A memóriatérkép és a cím alapján dönti el, hogy honnan (RAM, PPU regiszterek, stb...) kell olvasni.

- **write :: Word16 → Word8 → Emulator CPU ()**

Írás a memóriába. Hasonlóan itt is a memóriatérképet kell figyelembe venni.

- **push :: Word8 → Emulator CPU ()**

Bájt felrakása a stack-re.

- **pushAddress :: Word16 → Emulator CPU ()**

Cím felrakása a stack-re. Mivel a stack a kisebb címek felé nő, ezért először a szignifikánsabb bájtot kell felrakni, hogy tartsuk a kicsi-az-elején bájtsorrendet.

```
pushAddress addr = do
    let (high, low) = splitWord16 addr
    push high
    push low
```

- **nmi :: Emulator CPU ()**

NMI megszakítás érkezésekor meghívott eljárás. Elmenti a programszámlálót és a státusz-regisztert a stack-re és a megszakítási vektor által mutatott címen folytatja a végrehajtást.

- **processInterrupts :: Emulator CPU ()**

Az interrupt-ok nem érkeznek meg azonnal a processzorhoz, hanem időre van szükségük. Ennél a virtuális gépnél az interrupt regiszterek tárolják, hogy még hány utasítást kell végrehajtani a megszakításig. A processInterrupts eljárás csökkenti a számlálókat és ha valamelyik számláló eléri a nullát, akkor meghívja a megfelelő megszakítást feldolgozó eljárást.

- **oamDma :: Word8 → Emulator CPU ()**

Adott memórialapról másolás DMA-val a PPU OAM-ba.

#### 4.9.6. Nes.PPU.Emulation

- **cpuReadRegister** és **cpuWriteRegister**

A Nes.CPU.Emulation modul számára kiexportált függvények, amivel a CPU kommunikálhat a PPU-val.

- **getColor :: Word8 → Word8 → Emulator PPU Pixel**

Visszaadja, hogy adott palettán belül adott sorszámú azonosító milyen RGB kódot határoz meg.

- **getBackgroundColor :: Emulator PPU (Word8, Word8)**

Kinyeri a háttér csúsztatóregisztereiből a jelenlegi pixelhez tartozó háttérpaletta sorszámát és az azon belüli indexet.

- **overlaySpriteColor :: (Word8, Word8) → Emulator PPU Pixel**

Kombinálja az háttérszínt a sprite-rétegből származó színnel figyelembe véve a sprite-ok prioritását (háttér előtt/mögött). Az eredmény a pixel végső RGB kódja.

- **clock :: Emulator PPU ()**

A PPU állapotának léptetésére használt függvény. A PPU sorfolytonosan halad végig a képernyőn és állítja elő a pixeleket, amik vagy a háttérrétegből vagy a sprite-rétegből kerülnek ki. Emellett a háttérréteg következő celláinak adatait is előre betölti. A vertikális váltás (a CRT TV elektronágýúja a jobb alsó sarokból visszaáll a bal felső sarokba) ideje alatt tétlen (lásd az időzítési diagramot[19]).

#### 4.9.7. Nes.Emulation.Controls

- **press :: Button → Controller → Controller**

Ezzel a függvénnyel nyomhatunk le egy gombot a virtuális kontrolleren.

- **release :: Button → Controller → Controller**

Ezzel a függvénnyel engedhetünk fel egy gombot a virtuális kontrolleren.

- **read :: Controller → (Word8, Controller)**

A CPU ezzel a függvénnyel kérdezheti le a kontroller gombjainak állapotát. Az

i. meghívása a függvénynek visszaadja az *i*. gomb állapotát. (0 - felengedve, 1 - lenyomva) Az állapotváltoztató hatás a típusból egyértelműen látható.

- **write :: Word8 → Controller → Controller**

A CPU a kontroller írásával vissza tudja állítani a kontrollert az alaphelyzetbe (a 0. gombot fogja újra a *read* művelet lekérdezni).

#### 4.9.8. Nes.Emulation.MasterClock

- **syncCPUwithPPU :: Emulator CPU ()**

CPU utasítás végrehajtása, majd a megfelelő számú PPU órajel emulálása a szinkronizáció megőrzése érdekében.

- **emulateFrame :: Emulator Nes FrameBuffer**

Addig léptetjük a rendszert az előző függvényvel, amíg nem végzünk a képkockával.

- **resetNes :: Emulator Nes ()**

Visszaállítja alaphelyzetbe a komponenseket.

#### 4.9.9. Communication

##### 4.9.10. Emulator.JoyControls

- **init :: ButtonMappings → IO JoyControlState**

Létrehozza azt a rekordot, amiben a gombhozzárendeléseket és csatlakoztatott kontrollereket tartjuk nyilván.

- **manageButtonEvent**

Visszaadja, hogy a gombnyomáshoz milyen parancsok tartoznak (gyorsmentés, gyorsbetöltés vagy játékirányítás).

- **manageHatEvent**

A DPAD Fel/Le/Balra/Jobbra/Középen eseményeit figyeli és kiadja a megfelelő parancsokat a virtuális kontroller módosítására. Mivel a virtuális kontrollernél nincs Középen állapota a DPAD-nak, ezért arra is figyelni kell, hogy ezt az állapotot „*engedd fel az előző DPAD gombot*” parancsként kell továbbküldeni.

- **manageDeviceEvent**

Figyeli a kontrollercsatlakoztatásokat és az eltávolításokat. Az új kontrollereket inicializálja és eltárolja a rekordban, a kihúzottakat eltávolítja.

#### 4.9.11. Emulator.Framerate

- **uncapped :: MonadIO m ⇒ m Bool → m ()**

A megadott műveletet futtatja olyan gyorsan, amilyen gyorsan azt lehet addig, amíg a művelet igaz értékkal nem jelzi, hogy terminálni szeretne.

- **cappedAt :: MonadIO m ⇒ m Bool → Word32 → m ()**

A megadott műveletet futtatja a második paraméterben megadott frekvenciával (Hz), amíg a művelet igaz értékkal nem jelzi, hogy terminálni szeretne.

#### 4.9.12. Emulator.CrtShader

- **newShader :: ShaderType → ByteString → IO Shader**

Új GLSL shader lefordítása forráskódból. Inkompatibilis OpenGL verziónál előfordulhatnak fordítási hibák, ebben az esetben kivételt dob.

- **createProgramFrom :: [Shader → IO Program]**

Linkeli és validálja a shader-ekből készített programot, hiba esetén kivételt dob.

- **getCrtShaderProgram :: IO Program**

A modul exportált eljárása, ami létrehozza a CRT effektust megvalósító shader programot.

```
import Control.Monad ((=(<<), sequence)

getCrtShaderProgram =
    createProgramFrom =<< sequence
    [
        newShader VertexShader crtVertexShader,
        newShader FragmentShader crtFragmentShader
    ]
```

### 4.9.13. Emulator.Logic

- **translateSDLEvent :: SDL.EventPayload → Maybe Command**

Megadja egy beérkező eseményhez a hozzá tartozó parancsot.

- **pollCommands :: AppResources → IO [Command]**

Lekérdezi az SDL eseményeket és átfordítja őket belső parancsokra. A felhasználói felület által küldött parancsokat kiolvassa az *AppResources*-ban lévő csatornából (*TChan*). A végeredmény a két lista konkatenációja.

- **executeCommand :: AppResources → Command → Emulator Nes ()**

Az eljárás végrehajtja a paraméterül kapott parancsot.

- **advanceEmulation :: ViewFunction → AppResources → Emulator Nes Bool**

Végrehajtja az összes parancsot, lépteti az emulációt (ha az nincs szüneteltetve) és a paraméterül kapott megjelenítőfüggvényel frissíti a képernyőt.

```
type ViewFunction = AppResources -> IOVector Word8 -> Emulator Nes ()
```

### 4.9.14. Emulator.Window

- **acquireResources :: FilePath → CommResources → IO AppResources**

Inicializálja az SDL2 és OpenGL könyvtárakat, a létrehozott ablakot, renderert, kirajzolásnál használt textúrát, stb. visszaadja egy rekordban.

- **releaseResources :: AppResources → IO ()**

Felszabadítja a lefoglalt erőforrásokat és leállítja az SDL-t.

- **updateScreen :: ViewFunction**

A paraméterül kapott vektor a megjelenítendő kép nyers pixeladatait tartalmazza RGB24 formátumban. A vektor tartalmát feltölti a videómemóriában tárolt textúrába, majd ezután egy OpenGL (be van kapcsolva a CRT shader és sikerült lefordítani a shader programot) vagy SDL (ki van kapcsolva) hívással kirajzolja a textúrát a képernyőre.

- **runEmulatorWindow :: FilePath → CommResources → IO ()**

A modul exportált eljárása, amit a grafikus felület indít el egy új szálon. Az

emulátor a megadott elérési útvonalról tölti be a kazettát. A második paraméterben kapott csatornákkal tud kommunikálni az felhasználói felülettel.

## 4.10. Egy processzorutasítás végrehajtása

Ideje, hogy felhasználjuk a Nes.CPU.Emulation modulban implementált tömérdek mennyiségű utasítást. Az első lépés, hogy az utasítás opkódját kiolvassuk.

```
fetch :: Emulator CPU Opcode
fetch = readReg pc >>= read
```

Az opkódmátrix segítségével meg kell keresniük az opkódhöz tartozó utasítást és ciklusszámot. Az opkódmátrix eltárolásának egy egyszerű és meglepően hatékony módja a case elágazás.

```
data DecodedOpcode = DecodedOpcode {
    instruction :: Emulator CPU (),
    cycles :: Int
}

decodeOpcode :: Opcode -> DecodedOpcode
decodeOpcode opcode = case opcode of
    0x00 -> DecodedOpcode (implied >> brk) 7;
    ...

```

Az implementációmánál a dekódolt opkód nem külön, hanem az utasítással „egy-beégetve” tartalmazza a címzési módot (így könnyebb volt kezelní azt, hogy nem minden opkódhoz hozzá kellene adni a címzést). A címzési módokat megvalósító függvények arra számítanak, hogy a PC már az opkódot követő bajtra mutat, ezért a végrehajtás előtt meg kell növelni a PC értékét eggyel. A *cycle* eljárással megnöveljük az eltelt ciklusok számát a statikusan ismert értékkel, az *elapsedCycles* függvényel pedig megnézzük, hogy a végrehajtás után ténylegesen hány ciklus telt el. Így tehát a végső *runNextInstruction* eljárás:

```
import Data.Functor (<&>)

elapsedCycles :: Emulator CPU a -> Emulator CPU Int
elapsedCycles operation = do
    cyclesBefore <- readReg cyc
    operation
    readReg cyc <&> (\cyclesAfter -> cyclesAfter - cyclesBefore)
```

```
runInstruction :: DecodedOpcode -> Emulator CPU ()
runInstruction DecodedOpcode{instruction, cycles} = do
    modifyReg pc (+1)
    instruction
    cycle cycles

runNextInstruction :: Emulator CPU Int
runNextInstruction
= elapsedCycles (fetch <&> decodeOpcode >>= runInstruction)
```

## 4.11. A CPU és a PPU szinkronizációja

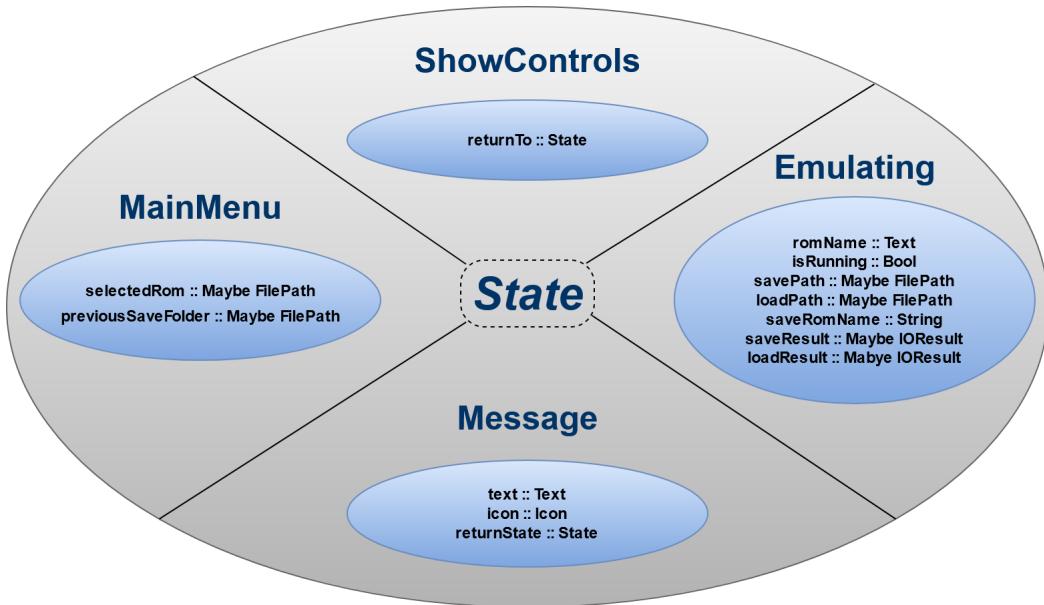
Az emuláció valójában szekvenciálisan történik, de arra oda kell figyelnünk, hogy egy processzorutasítás végrehajtása után megfelelő számú PPU léptetés történjen meg. Mivel a *runNextInstruction* visszaadja eredményül, hogy hány CPU órajel telt el elméletben az utasítás alatt, így már nem nehéz összehangolni a két komponenst annak tudatában, hogy a PPU órajel-frekvenciája háromszorosa a CPU órajel-frekvenciájának. A CPU-nak van hozzáférése a PPU-hoz, ezért ezt a CPU szintjén is meg tudjuk tenni.

```
syncCPUwithPPU :: Emulator CPU ()
syncCPUwithPPU = do
    clocks <- CPU.processInterrupts >> CPU.runNextInstruction
    directPPUAccess $ replicateM_ (clocks * 3) PPU.clock
```

## 4.12. A grafikus felhasználói felület

### 4.12.1. Állapotok

A felhasználói felület állapotait egy rekurzív algebrai adattípussal modelleztem. A 4.4 ábra mutatja, hogy az adattípusnak milyen konstruktőrök vannak és azok milyen mezőkkel rendelkeznek.



4.4. ábra. A felhasználói felület négy logikai állapota

A logikai állapotokon felül technikai okokból szükséges volt egy ötödik állapot bevezetése is annak a megvalósításához, hogy elhalványulásos átmenetekkel tudjunk váltani az állapotok között.

### 4.12.2. Állapotátmenetek

A mintaillesztési szintaxissal nagyon tömörén megfogalmazhatjuk azokat a feltételeket, amiknek teljesülniük kell egy állapotátmenet bekövetkezéséhez.

```

update :: CommResources -> State -> Event -> Transition State Event
update comms Emulating{savePath} SDLWindowClosed
= smoothTransition comms (MainMenu Nothing savePath)

```

Amikor a felhasználó bezárja az emulációs ablakot, akkor átmenettel a MainMenu állapotba lépünk (és megőrizzük a kiválasztott mentési mappát).

```

data MessageIcon = Info | Alert | Cross

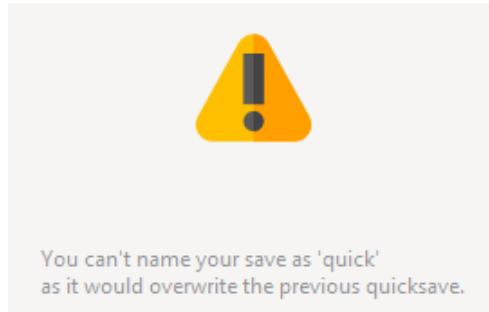
saveAsQuick = "You can't name your save as ..."

update comms e@Emulating{saveRomName = "quick"} SaveButtonPressed
= smoothTransition comms (Message saveAsQuick Alert e)

```

A második példa azt az eshetőséget kezeli, amikor a felhasználó *quick* néven szeretne egyedi mentést létrehozni. Ennek hatására az *Üzenet* állapotba lépünk át

ahol megjelenítünk egy figyelmeztető üzenetet (4.5 ábra). Arra is emlékeznünk kell, hogy melyik állapotból jöttünk, hogy később ugyanabba az állapotba térhessünk vissza.



4.5. ábra. A kiváltott figyelmeztetés<sup>11</sup>

#### 4.12.3. Állapotmegjelenítés

A szemléltetés kedvéért kiragadtam egy részt az *inGame* állapotmegjelenítő-függvényből, ahol azt láthatjuk, hogy a gomb feliratában a játék neve előtt a *Pause* vagy *Resume* szöveg lesz, az állapottól függően.

```
inGame :: State -> Widget Event
inGame Emulating{isRunning, romName} =
    container Box
        [#orientation := OrientationVertical, #valign := AlignCenter]
        [
            ...
            , BoxChild defaultBoxChildProperties $
                widget Button
                [
                    #label :=
                        ((if isRunning then "Pause" else "Resume") <=> romName)
                    , on #clicked TogglePause
                ]
            ...
        ]
    ]
```

A fenti függvény emellett még a következőkről is gondoskodni tud:

- Ha még nincs mentési mappa kiválasztva, akkor ne legyenek kattinthatók az érintett funkciók gombjai

---

<sup>11</sup>Link az ikonokra: <https://www.flaticon.com/packs/ui-interface-23>,

<https://www.flaticon.com>

- Ha már volt mentési mappa kiválasztva korábbi futtatásnál, akkor a mappa-választó URI-ját állítsuk be erre
- Ha már volt mentési próbálkozás, akkor ikonnal jelezzük a sikereségét, mutassuk az időpontját
- Ha már volt betöltési próbálkozás, akkor azt is hasonlóan jelenítsük meg
- Szüneteltetésnél és folytatásnál változzon a funkcióhoz tartozó gomb felirata és az ikon

## 4.13. Tesztelés

A fejlesztés a Test Driven Development (TDD) módszer szerint folyt. Mielőtt belekezdtem volna egy CPU utasítás vagy PPU eljárás implementálásába, a teszteket kibővítettem, hogy az új kódrészeket is lefedjék, ezáltal mindenkor tudtam, hogy jó irányba haladok-e.

Az emulátorok tesztelésére vannak kifejezetten erre a célra készített NES ROM-<sup>12</sup>, amik tökéletes aprólékosággal ellenőrzik, hogy minden utasítás az elvárt módon változtatja-e a hardver belső állapotát. Azért döntöttem ezek használata mellett, mert ilyen pontosságú tesztek írása 6502 Assembly-ben rengeteg időt vett volna el és a saját tesztek lefedettsége meg se közelítette volna a professzionális tesztek által nyújtott lefedettséget.

A tesztek vizuálisan és automatizáltan is futtathatóak. Vizuális futtatásnál a teszt ugyanúgy töltődik be, mint bármilyen más játék és a képernyőre írja ki a teszt eredményét. Az automatizált tesztelésnél egy speciális ciklusban fut az emuláció, ami kiértékeli a teszt terminálási feltételét minden léptetés után. A Nestest kivételével a tesztek az eredményt egy kóddal jelzik a \$6000-es címen és még hibaüzeneteket is írnak egy \$6004-től kezdődő nullával terminált sztring formájában.

### 4.13.1. CPU tesztek

#### Nestest

Részletesen végigteszteli az összes utasítást (beleértve néhány illegálisat is). Az utasítások címzési módját és sorrendjét is variálja. Összesen 8991 utasítást hajt

---

<sup>12</sup>[http://wiki.nesdev.com/w/index.php/Emulator\\_tests](http://wiki.nesdev.com/w/index.php/Emulator_tests)

végre. A teszthez tartozó naplóval ellenőrizhetjük, hogy a regiszterek értéke és az eltelt ciklusok száma egyezik-e az elvárttal. Mivel itt nem a ROM végzi az összehasonlításokat, hanem mi a napló segítségével, ezért a többi teszttel ellen-tétben akkor is használható, ha a CPU még nem tud helyesen összehasonlítani vagy a vezérlés még nem működik.

#### **Instr \_ test \_ v5**

16 tesztet tartalmaz, amik a címzési módokat, veremműveleteket, elágazásokat és megszakításokat tesztelik.

#### **Instr \_ Misc**

Az abszolút indexelt címzés és az elágazások szélsőséges eseteinek ellenőrzése.

### **4.13.2. PPU tesztek**

#### **PPU \_ vbl \_ nmi**

Ez a tesztcsomag a vertical blanking period státusz-flaget (be/kikapcsolása megfelelő időben történik), az NMI megszakítást (bekövetkezik-e, a PPU regiszterek befolyásolják-e) és a páros-páratlan képkockák 1 PPU órajelciklusnyi hosszkülönbségét ellenőrzi.

#### **PPU \_ sprite \_ hit és PPU \_ sprite \_ overflow**

12 tesztcsomag, amik a sprite réteggel kapcsolatos státusz flag-eket ellenőrzik (Sprite 0 Hit, Sprite Overflow).

### **4.13.3. Eredmények**

Emulátoroknál a 100%-os pontosság elérése nehéz feladat, mert az áramkörök felépítéséből természetes következményként adódó speciális viselkedéseket mesterségesen, összetett feltételek ellenőrzésével kell implementálni (például memóriabuszkonfliktus, NMI elnyomás, stb.). Az emulátorom időzítésekben és az előbb említett két esetben tér el minimálisan az eredeti hardvertől. Összességében a pontosság kielégítő, és a legtöbb játéknál nem okoz gondot az eltérés.

```
* OK Run all tests
OK Branch tests
OK Flag tests
OK Immediate tests
OK Implied tests
OK Stack tests
OK Accumulator tests
OK (Indirect,X) tests
OK Zeropage tests
OK Absolute tests
OK (Indirect),Y tests
OK Absolute,Y tests
OK Zeropage,X tests
OK Absolute,X tests

Up/Down: select test
Start: run test
Select: Invalid ops!
```

4.6. ábra. CPU ellenőrzése a Nestest ROM-mal

```
Sprite 0 Hit
  Basics:      OK (0.96s)
  Alignment:   OK (0.87s)
  Corners:    OK (0.51s)
  Flip:        OK (0.44s)
  Left clip:  OK (0.77s)
  Right edge: OK (0.59s)
  Screen bottom: OK (0.63s)
  Double height: OK (0.55s)
  Timing order:  OK (1.01s)
Sprite Overflow
  Basics:      OK (0.52s)
  Details:    OK (0.70s)
  Emulator:   OK (0.23s)

All 36 tests passed (3.03s)
```

4.7. ábra. Automatizáltan végrehajtott sprite-réteg tesztek

#### 4.13.4. Tesztelt, megbízhatóan működő programok

##### Homebrew játékok

- Game of Life: <https://www.romhacking.net/homebrew/48/>
- 2048: <https://www.romhacking.net/homebrew/65/>
- Alter Ego: <https://www.romhacking.net/homebrew/1/>
- Lawn Mover: <http://www.romhacking.net/homebrew/42/>

- Lan Master: <https://www.romhacking.net/homebrew/2/>
- Sudoku: <https://www.romhacking.net/homebrew/17/>
- Chase: <https://www.romhacking.net/homebrew/73/>
- Chicken Of The Farm: <https://www.romhacking.net/homebrew/109/>
- Falldown: <https://www.romhacking.net/homebrew/57/>
- Fighter F8000: [http://nesdev.com/fighter\\_f8000.zip](http://nesdev.com/fighter_f8000.zip)
- Magic Floor: <https://www.romhacking.net/homebrew/40/>
- Pegs: <https://www.romhacking.net/homebrew/22/>
- Pong 198x: <https://www.romhacking.net/homebrew/26/>
- NeSnake: <https://www.romhacking.net/homebrew/31/>
- NeSnake 2: <https://www.romhacking.net/homebrew/30/>
- MilionNESy: <https://www.romhacking.net/homebrew/36/>
- Sokoban: <http://nesdev.com/sokoban.zip>
- Concentration Room:  
[http://wiki.nesdev.com/w/index.php/Concentration\\_Room](http://wiki.nesdev.com/w/index.php/Concentration_Room)
- Eskimo Bob Demo:  
<https://spoony-bard-productions.itch.io/eskimo-bob>

## Techdemók

- Raster Demo: <http://nesdev.com/rstrdemo.zip>
- Motion <http://nesdev.com/animz.zip>
- Fullscreen Demo <http://nesdev.com/FullScreen.zip>
- Interlacing Demo <http://nesdev.com/interlac.zip>

## 5. fejezet

### Összegzés

A megszületett program sikeresen demonstrálja a Haskell nyelv alkalmasságát összetett, nagyfokú precizitást igénylő, de mégis hatékony alkalmazások készítésére. A magas szintű absztrakcióknak köszönhetően gyors fejlesztési ütemet tudtam tartani, így új funkciók implementálására és finomhangolására tudtam fordítani azt az időt, amit a hardverközeli nyelveknél a szegmentálási hibák kijavítása emészttet volna fel. Emellett elismerés jár a *gi-gtk-declarative* könyvtárnak, ami kísérleti stádiuma ellenére a gyakorlatban is jól alkalmazható eszközöknek bizonyult a felhasználó felület megalkotásához.

Az emulátor számos játékkal kompatibilis és az egyedi megjelenés, valamint a kényelmi funkciók miatt úgy gondolom, hogy jól használható eszköz bárki számára, aki meg szeretne ismerkedni a 80-as évek játékainak világával.

# Irodalomjegyzék

[1] NES ROM-ok listája.

<http://tuxnes.sourceforge.net/nesmapper.txt>

Dátum: 2020.04.27.

[2] Hivatalos 6502 adatlap.

[http://archive.6502.org/datasheets/rockwell\\_r650x\\_r651x.pdf](http://archive.6502.org/datasheets/rockwell_r650x_r651x.pdf)

Dátum: 2020.04.01.

[3] 6502 opkódok.

<http://www.oxyron.de/html/opcodes02.html>

Dátum: 2020.04.01.

[4] 6502 utasításkészlet leírása.

<http://obelisk.me.uk/6502/reference.html>

Dátum: 2020.04.01.

[5] *Nesdev*. NES referencia útmutató.

[http://wiki.nesdev.com/w/index.php/NES\\_reference\\_guide](http://wiki.nesdev.com/w/index.php/NES_reference_guide)

Dátum: 2020.04.01.

[6] *Nesdev*. NES CPU referencia útmutató.

<http://wiki.nesdev.com/w/index.php/CPU>

Dátum: 2020.04.01.

[7] *Nesdev*. CPU memóriatérkép.

[http://wiki.nesdev.com/w/index.php/CPU\\_memory\\_map](http://wiki.nesdev.com/w/index.php/CPU_memory_map)

Dátum: 2020.04.01.

[8] *Nesdev*. Memóriatükrözés.

[http://wiki.nesdev.com/w/index.php/Mirroring#Memory\\_Mirroring](http://wiki.nesdev.com/w/index.php/Mirroring#Memory_Mirroring)

Dátum: 2020.04.01.

[9] *Nesdev*. NES PPU referencia útmutató.

<http://wiki.nesdev.com/w/index.php/PPU>

Dátum: 2020.04.01.

[10] *Nesdev*. NROM leírás.

<http://wiki.nesdev.com/w/index.php/NROM>

Dátum: 2020.04.01.

[11] *Nesdev*. UNROM (UxROM) leírás.

<http://wiki.nesdev.com/w/index.php/UxROM>

Dátum: 2020.04.01.

[12] *Nesdev*. CNROM leírás.

[http://wiki.nesdev.com/w/index.php/INES\\_Mapper\\_003](http://wiki.nesdev.com/w/index.php/INES_Mapper_003)

Dátum: 2020.04.01.

[13] *Nesdev*. NES órajelek.

[http://wiki.nesdev.com/w/index.php/Cycle\\_reference\\_chart](http://wiki.nesdev.com/w/index.php/Cycle_reference_chart)

Dátum: 2020.04.01.

[14] *Nesdev*. NES illegális opkódok.

[http://wiki.nesdev.com/w/index.php/CPU\\_unofficial\\_opcodes](http://wiki.nesdev.com/w/index.php/CPU_unofficial_opcodes)

Dátum: 2020.04.01.

[15] *Nesdev*. NES sztenderd kontrollerének működése.

[http://wiki.nesdev.com/w/index.php/Standard\\_controller](http://wiki.nesdev.com/w/index.php/Standard_controller)

Dátum: 2020.04.01.

[16] Harmadik játékkonzol-generáció.

[https://en.wikipedia.org/wiki/List\\_of\\_home\\_video\\_game\\_consoles#Third-generation\\_\(1983%E2%80%932003\)](https://en.wikipedia.org/wiki/List_of_home_video_game_consoles#Third-generation_(1983%E2%80%932003))

Dátum: 2020.04.01.

[17] iNES fájlformátum.

<https://formats.kaitai.io/ines/index.html>

Dátum: 2020.04.01.

[18] Háttéreltolás.

[http://wiki.nesdev.com/w/index.php/PPU\\_scrolling](http://wiki.nesdev.com/w/index.php/PPU_scrolling)

Dátum: 2020.04.01.

[19] PPU időzítési diagram.

[http://wiki.nesdev.com/w/index.php/File:Ntsc\\_timing.png](http://wiki.nesdev.com/w/index.php/File:Ntsc_timing.png)

Dátum: 2020.04.01.

[20] *Timothy Lottes*. CRT effektust leíró GLSL shader.

<https://github.com/sdlpal/sdlpal/blob/master/shaders/crt.glsl>

Dátum: 2020.04.01.

# Ábrák jegyzéke

2.1.	Nintendo Entertainment System (1985) <sup>13</sup>	6
2.2.	A felhasználói felület játék közben <sup>14</sup>	12
2.3.	Alter Ego. Linkért lásd: 4.13.4	16
2.4.	Lawn Mover. Linkért lásd: 4.13.4	16
3.1.	A NES alaplapja <sup>15</sup>	17
3.2.	A 6502 opkód mátrixa	19
3.3.	Opkód argumentumainak helye	21
3.4.	Indexelt indirekt címzés	23
3.5.	A 2C02 színpalettája	27
3.6.	Egy alakzat reprezentálása	29
3.7.	Az Alter Ego játék 0. alakzattablázata különböző palettákkal kirajzolva	29
3.8.	Attribútumbájtok felosztása	30
4.1.	A felület állapotátmenetei	41
4.2.	Az emulációnál használt rekordok	42
4.3.	A program moduljainak függőségi gráfja	44
4.4.	A felhasználói felület négy logikai állapota	54
4.5.	A kiváltott figyelmeztetés <sup>16</sup>	55
4.6.	CPU ellenőrzése a Nestest ROM-mal	58
4.7.	Automatizáltan végrehajtott sprite-réteg tesztek	58

# Táblázatok jegyzéke

2.1.	Játékok irányításához használt gombok . . . . .	11
2.2.	Az emuláció vezérlése . . . . .	13
3.1.	A paletta indexek címei . . . . .	28
3.2.	A képfeldolgozó memóriatérképe . . . . .	33