



SAPIENTIA
ERDÉLYI MAGYAR
TUDOMÁNYEGYETEM

TUDOMÁNYOS DIÁKKÖRI DOLGOZAT

Egy CHIP-8 interpreter tervezése és megvalósítása

Szerző:

Hadnagy Lóránt, Sapientia EMTE, Marosvásárhelyi Kar, Informatika BSc. szak, I. évfolyam

Témavezető:

Dr. Antal Margit docens, Sapientia EMTE, Marosvásárhelyi Kar

2024

Abstract

Understanding low-level system operations is important in computer science education. The challenge lies in providing students with a hands-on approach to understanding computer architecture without demanding extensive prior knowledge of hardware.

This paper proposes the development of a CHIP-8 interpreter with the purpose of comprehending the overview of the system's functionalities. By working on this educational project, students gain insights into low-level system operations while simultaneously learning programming concepts and problem-solving skills.

This approach not only simplifies the learning process but also paves the way for further exploration and understanding in the field of emulation.

Keywords: *CHIP-8, interpreter, emulation, C++, virtualization*

Kivonat

Az informatika oktatásában fontos az alacsony szintű rendszerműveletek megértése. A kihívás abban rejlik, hogy a tanulóknak gyakorlatias megközelítést nyújtsunk a számítógép-architektúra megértéséhez anélkül, hogy hardverrel kapcsolatos átfogó előzetes ismereteket követelnénk meg.

Ez a dolgozat egy CHIP-8 interpreter fejlesztését javasolja, amelynek célja a rendszer funkcionalitásainak áttekintése. Ezen az oktatási projekten dolgozva a diákok betekintést nyerhetnek a rendszer alacsony szintű működésébe, miközben egyidejűleg programozási fogalmakat tanulnak és problémamegoldó készségeket fejlesztenek.

Ez a megközelítés nemcsak a tanulási folyamatot egyszerűsíti, hanem az emuláció területén való további felfedezés és megértés útját is megnyitja.

Kulcsszavak: *CHIP-8, interpreter, emuláció, C++, virtualizáció*

Tartalomjegyzék

Tartalomjegyzék.....	4
1. Bevezetés.....	5
2. CHIP-8 specifikációk.....	6
2.1 Memória.....	6
2.2 Regiszterek.....	8
2.3 Stack.....	9
2.4 Kijelző.....	10
2.5 Billentyűzet.....	11
2.6 Utasítások.....	12
2.7 Alternatív utasítások.....	15
2.8 Programok betöltése.....	19
2.9 Végrehajtási ciklus.....	19
3. Tervezés és implementáció.....	20
3.1 Felhasznált technológiák.....	20
3.2 Jellemzők.....	22
3.3 Opkód dekódolás.....	26
3.4 Egyéb részletek.....	33
3.5 Dekódoló módszerek összehasonlítása.....	34
4. Összehasonlítás más implementációkkal.....	36
5. Összegzés.....	38
Hivatkozások.....	39

1. Bevezetés

A CHIP-8 egy interpretált programozási nyelv, amelyet Joseph Weisbecker (1932-1990) fejlesztett ki az RCA 1802-es mikroprocesszorra [1, 4]. Kezdetben a COSMAC VIP és a Telmac 1800 8 bites mikroszámítógépeken használták az 1970-es évek közepén. A CHIP-8 programok egy virtuális gépen futnak, melynek célja az volt, hogy egyszerűsítse a videojátékok programozását a korai számítógépeken.

Az 1990-es évek elején a CHIP-8 nyelvet Andreas Gustafsson élesztette újjá, aki CHIP-48 [2, 3] néven készített egy CHIP-8 interpretert a HP48 grafikus számológéphez. A HP48-ra akkoriban nem igazán lehetett gyorsan és egyszerűen játékokat készíteni, erre biztosított egy megoldást a CHIP-8.

CHIP-8 interpreterk és programok a mai napig készülnek, egyszerűségük, valamint történelmi jelentőségük és népszerűségük miatt.

A dolgozat célja az olvasó végigvezetése egy CHIP-8 interpreter fejlesztésén, aminek során jobban megértheti az emulációs elveket, és az alacsony szintű programozási készségeit is fejlesztheti. A tanulmányi projekt keretében a CHIP-8 virtuális géphez egy olyan szoftver-emulátor készül, amely képes CHIP-8 programokat végrehajtani és megjeleníteni.

2. CHIP-8 specifikációk

Ez a fejezet részletesen bemutatja a CHIP-8 értelmező alapvető specifikációit, amelyek többnyire Joseph Weisbecker COSMAC VIP implementációját [4, 5, 6] követik.

2.1 Memória

0xE90 - 0xFFFF változók
0x200 - 0xE8F Szabad hely
0x000 - 0x1FF Interpreter

A CHIP-8 modern implementációja 4 KiB memóriát használ (lásd **2.1. táblázat**). Ebből a 0x000 és 0x1FF címek közötti régió az eredeti interpreter számára fenntartott terület.

A memóriában tárolásra kerülnek a hexadecimális számjegyek *sprite*¹-jai, amelyek 5 bájtos, azaz 5 x 8 pixeles képelemek. A programok hivatkozhatnak a *sprite*-ok adataira, és elméletileg az eredeti interpreter helyén, a 0x000 és 0x1FF címek közötti tartományban bárhol elhelyezkedhetnek, de gyakorlatilag ezeket a 0x050 és 0x09F címek között szokás tárolni. Ezeket a *sprite*-adatokat *fontset*-nek nevezzük (lásd **2.2. táblázat**). Léteznek nagyobb felbontású *fontset*-ek is, amelyet a fejlettebb variánsok használnak.

A programokat a 0x200 címtől kezdődően kell betölteni. (Az ETI 660 számítógépek esetében a programok a 0x600 címen kezdődnek). Fontos megjegyezni, hogy eredetileg az utolsó 352 bájt változóknak és képernyő frissítésnek van fenntartva, így a szabadon használható memóriaterület a 0x200 és 0xE8F címek között helyezkedik el.

2.1. táblázat.

CHIP-8

Memória térkép

¹ [https://hu.wikipedia.org/wiki/Sprite_\(informatika\)](https://hu.wikipedia.org/wiki/Sprite_(informatika))

Karakter	Bináris	Hex	Karakter	Bináris	Hex	Karakter	Bináris	Hex
0	11110000 10010000 10010000 10010000 11110000	0xF0 0x90 0x90 0x90 0xF0	6	11110000 10000000 11110000 10010000 11110000	0xF0 0x80 0xF0 0x90 0xF0	C	11110000 10000000 10000000 10000000 11110000	0xF0 0x80 0x80 0x80 0xF0
1	00100000 01100000 00100000 00100000 01110000	0x20 0x60 0x20 0x20 0x70	7	11110000 00010000 00100000 01000000 01000000	0xF0 0x10 0x20 0x40 0x40	D	11100000 10010000 10010000 10010000 11100000	0xE0 0x90 0x90 0x90 0xE0
2	11110000 00010000 11110000 10000000 11110000	0xF0 0x10 0xF0 0x80 0xF0	8	11110000 10010000 11110000 10010000 11110000	0xF0 0x90 0xF0 0x90 0xF0	E	11110000 10000000 11110000 10000000 11110000	0xF0 0x80 0xF0 0x80 0xF0
3	11110000 00010000 11110000 00010000 11110000	0xF0 0x10 0xF0 0x10 0xF0	9	11110000 10010000 11110000 00010000 11110000	0xF0 0x90 0xF0 0x10 0xF0	F	11110000 10000000 11110000 10000000 10000000	0xF0 0x80 0xF0 0x80 0x80
4	10010000 10010000 11110000 00010000 00010000	0x20 0x60 0x20 0x20 0x70	A	11110000 10010000 11110000 10010000 10010000	0xF0 0x90 0xF0 0x90 0x90			
5	11110000 10000000 11110000 00010000 11110000	0xF0 0x80 0xF0 0x10 0xF0	E	11100000 10010000 11100000 10010000 11100000	0xE0 0x90 0xE0 0x90 0xE0			

2.2. táblázat. Klasszikus fontset

2.2 Regiszterek

A CHIP-8 a következő regiszterekkel rendelkezik:

- **Adatregiszterek**

A CHIP-8 interpreter 16 általános célú adatregisztert használ, amelyek V0-tól VF-ig vannak megszámozva. Ezek 8 bites regiszterek, amelyek előjel nélküli egész számokat képesek eltárolni.

Ezek az adatregiszterek szolgálnak az interpreteren belüli adatmanipuláció elsődleges módjaként. Különböző utasítások segítségével a regiszterek értékekkel tölthetők be, és azokat összeadásnak, kivonásnak vagy egyéb műveleteknek vethetjük alá.

A 16 regiszter közül különleges a VF regiszter, amelyet nem ajánlatos tetszőleges módon használni a programoknak, mivel néhány utasítás kapcsolóként (*flag*) használja.

- **Index (I) regiszter**

Az **I regiszter**, egy 16 bites regiszter, amely a 8 bites adatregiszterekkel ellentétben képes címeket is eltárolni, ezért legtöbbször csak a legalsó 12 bit van használatban.

Ezzel a regiszterrel lehetséges a memóriamanipuláció és a grafikához szükséges sprite-ok kiválasztása.

- **Program counter (PC) regiszter**

A **PC regiszter**, amely 16 bites, a végrehajtandó utasítás címét tárolja, ezért csak a legalsó 12 bit van használatban.

- **Stack pointer (SP) regiszter**

Az **SP regiszter**, amely 8 bites, a következő üres *stack* szintre mutat. Ezzel a regiszterrel vannak nyilvántartva a szubrutin hívások címei.

- **Időzítő regiszterek**

A CHIP-8 két időzítő regiszterrel rendelkezik, a **DT (*delay timer*)** és az **ST (*sound timer*)** 8 bites regiszterekkel, amelyek az események időzítéséért és a hangért felelősek. Ezen regiszterek értékei 60 Hz-es frekvenciával egyesével csökkennek, amíg el nem érik a 0 értéket. A **DT** regisztert a programok események koordinálására használják.

Az **ST** regiszter biztosítja a CHIP-8 hangkimenetének irányítását. Amíg az **ST** értéke nem nulla, a rendszer hangjelzést ad ki. Ennek a hangnak a frekvenciája és hullámformája nincs meghatározva.

2.3 Stack

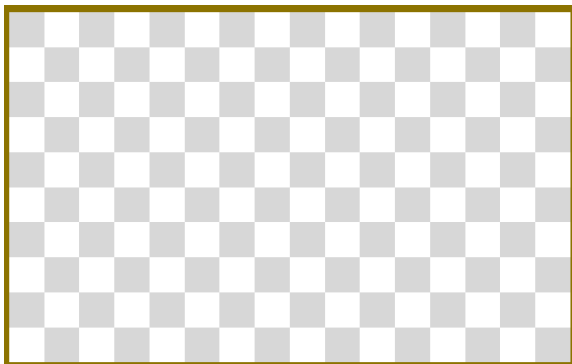
Az eredeti CHIP-8 specifikáció szerint a rendszer 12 szintes *stack*-el rendelkezik, de általában a modern implementációkban 16 szintűt szoktak használni. Egy másik változat a körkörös *stack*, ami megoldást jelenthet az olyan hibás programok futtatására, amelyek nem térnek vissza a szubrutinjaikból, és csak folyton új hívásokat végeznek.

A fejlesztés megkönnyítése érdekében, illetve más CHIP-8 variánsok jövőbeli támogatása érdekében a *stack*-et egy különálló tömbbel szokás megvalósítani, viszont hitelesebb megoldás lenne a *stack*-et a memóriába tenni. Ebben az esetben a *stack* magasabb címek felől nő az alacsonyabbak irányába.

2.4 Kijelző

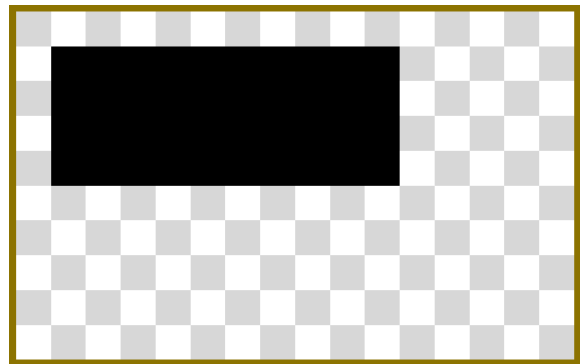
Az interpreter rendelkezik egy képernyőpufferrel is, amelyet grafika megjelenítésére használ. Mivel a kijelző 64 pixel széles és 32 pixel magas, így az interpreterben egy 64 x 32-es tömbbel van modellezve. Ezzel képes a rendszer *sprite*-okat megjeleníteni, amelyek maximum 15 bájtból állhatnak, azaz 8 x 15 pixelesek lehetnek. Minden pixelnek két állapota lehet: aktív vagy inaktív, így csak 2 színű lehet a képernyő. A rajzolás a *sprite* pixeljei és a képernyőpuffer pixeljeinek XOR-olásával történik, amelynek eredménye lesz a képernyőpuffer adott pixeljének új értéke. Ennek a folyamatnak a leegyszerűsített szemléltetése látható a 2.1., 2.2., 2.3. és a 2.4. ábrákon [7].

1. Üres képernyővel kezdünk.



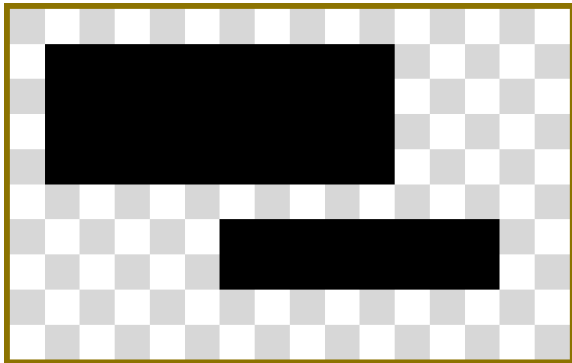
2.1. ábra. Egyszerűsített rajzolás, 1. lépés

2. Egy 10x4-es *sprite*-ot rajzolunk az (1,1) és (10,4) pontok által meghatározott mezőre.



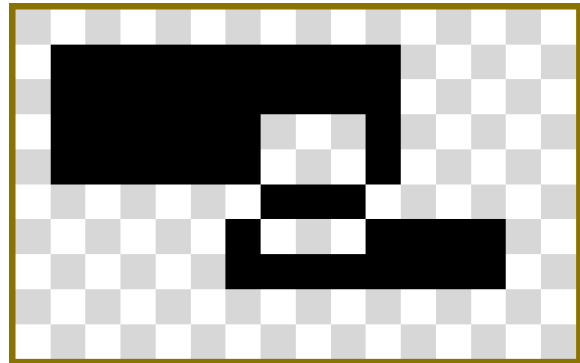
2.2. ábra. Egyszerűsített rajzolás, 2. lépés

3. Ezután rajzolunk egy 8x2-es *sprite*-ot a (6,6) és (13,7) pontok által meghatározott mezőre.



2.3. ábra. Egyszerűsített rajzolás, 3. lépés

4. Egy 3x4-es *sprite*-ot rajzolunk a (7,3) és (9,6) pontok által meghatározott mezőre



2.4. ábra. Egyszerűsített rajzolás, 4. lépés

2.5 Billentyűzet

Az eredeti mikroszámítógépek, amelyekre a CHIP-8-at tervezték, egy 4 soros, 4 oszlopos billentyűzettel rendelkeznek, (lásd **2.5. ábra**), amelyen a hexadecimális szimbólumok vannak feltüntetve.



2.5. ábra. COSMAC VIP billentyűzete

Az implementációban ez egy 16 elemű, logikai típusú tömb, amely a billentyűk állapotát tárolja.

2.6 Utasítások

Minden CHIP-8 utasítás 2 bájt hosszú és csökkenő bájtsorrendben² (*big endian*) van eltárolva. A memóriában minden utasítás első bájtjának páros címen kell elhelyezkednie. Ha a program *sprite*-adatokat tartalmaz, akkor azokat ki kell egészíteni, hogy az ezt követő utasítások megfelelően helyezkedjenek el a memóriában.

A korai CHIP-8 utasításkészletben csak 31 utasítás szerepelt [4, 5]. A 8XY1, 8XY2, 8XY4 és 8XY5 utasítások dokumentálva voltak, viszont kimaradtak a 8XY3, 8XY6, 8XY7 és 8XYE utasítások.

Ez amiatt történt, hogy a COSMAC VIP számítógépen a 8XYN utasítás az N értékből képzett egybájtos szubrutin végrehajtásával működik, így a kimaradt 4 utasítás is helyes, és gyakran használják őket az eredeti implementációval kompatibilis programokban [6]. Éppen ezért az alapvető CHIP-8 utasításkészlet részeként tekinthetünk rájuk.

A Weisbecker eredeti interpreterére épülő, alapvető CHIP-8 utasításkészletet a **2.3. táblázat** tartalmazza.

Megjegyzések a táblázathoz:

- NNN egy memóriacím;
- NN egy bájt;
- N egy nibble;
- X és Y az adott regiszterek számai;
- A CHIP-8-hoz több féle mnemonik³ is létezik. Ezek közül a Chipper [8] és az Octo [9] a legelterjedtebb, viszont sokan használják a sajátjukat is.

² <https://hu.wikipedia.org/wiki/B%C3%A1jtsorrend>

³ hosszabb elnevezésű művelet(sor) elnevezésére használatos rövidítés

Utasítás	Mnemonik		Leírás
	Chipper	Octo	
0NNN	SYS NNN	0x0N 0xNN	Végrehajtja a gépi nyelvi szubrutint az NNN címen
00E0	CLS	clear	Kiüríti a képernyőpuffert
00EE	RET	return	Visszatér a szubrutinból
1NNN	JP NNN	jump NNN	Az NNN címre ugrik
2NNN	CALL NNN	:call NNN	Végrehajtja a szubrutint az NNN címen
3XNN	SE VX, NN	if VX != NN then	Kihagyja a következő utasítást, ha VX egyenlő NN
4XNN	SNE VX, NN	if VX == NN then	Kihagyja a következő utasítást, ha VX nem egyenlő NN
5XY0	SE VX, VY	if VX != VY then	Kihagyja a következő utasítást, ha VX egyenlő VY
6XNN	LD VX, NN	VX := NN	Betölti az NN-t a VX-be
7XNN	ADD VX, NN	VX += NN	Hozzáadja az NN-t a VX-hez
8XY0	LD VX, VY	VX := VY	Betölti a VY értékét a VX-be
8XY1	OR VX, VY	VX = VY	Betölti a VX OR VY művelet eredményét VX-be
8XY2	AND VX, VY	VX &= VY	Betölti a VX AND VY művelet eredményét VX-be
8XY3	XOR VX, VY	VX ^= VY	Betölti a VX XOR VY művelet eredményét VX-be
8XY4	ADD VX, VY	VX += VY	Hozzáadja VY értékét VX-hez Túlcsordulás esetén VF 1 lesz, különben 0
8XY5	SUB VX, VY	VX -= VY	Kivonja VY értékét VX-ból Túlcsordulás esetén VF 1 lesz, különben 0

8XY6	SHR VX {, VY}	VX >= VY	Egy bittel jobbra tolja VY értékét, és betölti VX-be VF az eltolás előtti legalacsonyabb helyiértékű bit lesz
8XY7	SUBN VX, VY	VX -= VY	VX-et egyenlővé teszi VY mínusz VX-el Túlsordulás esetén VF 0 lesz, különben 1
8XYE	SHL VX {, VY}	VX <= VY	Egy bittel balra tolja VY értékét, és betölti VX-be VF az eltolás előtti legmagasabb helyiértékű bit lesz
9XY0	SNE VX, VY	if VX == VY then	Kihagyja a következő utasítást, ha VX nem egyenlő VY-al
ANNN	LD I, NNN	i := NNN	Betölti az NNN címet az I regiszterbe
BNNN	JP V0, NNN	jump0 NNN	Az NNN cím plusz V0 címre ugrik
CXNN	RND Vx, NN	VX := random NN	Egy véletlen számot maszkol NN-el, és betölti VX-be
DXYN	DRW VX, VY, N	sprite VX VY N	Rajzol egy sprite-ot a (VX, VY) által jelzett pozícióba, az I regiszterben tárolt címen kezdődő, N bájtnyi adattal Ha bármely aktív pixel inaktívvá válik, VF 1 lesz, különben 0
EX9E	SKP VX	if VX -key then	Kihagyja a következő utasítást, ha VX hexadecimális alakjának megfelelő billentyű le van nyomva
EXA1	SKNP VX	if VX key then	Kihagyja a következő utasítást, ha VX hexadecimális alakjának megfelelő billentyű nincs lenyomva
FX07	LD VX, DT	VX := delay	Betölti a DT aktuális értékét VX-be
FX0A	LD VX, K	VX := key	Megvárja egy billentyű lenyomását, és betölti az eredményt VX-be
FX15	LD DT, VX	delay := VX	Betölti VX-et a DT-be

FX18	LD ST, VX	buzzer := VX	Betölti VX-et az ST-be
FX1E	ADD I, VX	i += VX	Hozzáadja VX-et az I regiszterhez
FX29	LD F, VX	i := hex VX	Betölti VX hexadecimális alakjának megfelelő sprite-adat címét az I regiszterbe
FX33	LD B, VX	bcd VX	Betölti VX BCD-megfelelőjét az I, I + 1 és I + 2 címeken
FX55	LD [I], VX	save VX	Betölti a memóriába, az I regiszterben tárolt címtől kezdődően, V0-tól VX-ig a regiszterek értékét I regiszter értéke I + X + 1 lesz
FX65	LD VX, [I]	load VX	Feltölti a memóriából, az I regiszterben tárolt címtől kezdődően, V0-tól VX-ig a regisztereket I regiszter értéke I + X + 1 lesz

2.3. táblázat. Alapvető CHIP-8 utasításkészlet

2.7 Alternatív utasítások

A gyakorlatban több CHIP-8 variáns létezik, ezek különböző adaptációi vagy továbbfejlesztései az eredetinek, amelyekben számos változtatás és alternatív utasítás található.

Implementációmban néhány egyszerű, alternatív utasítást biztosítottam, amelyek az eredeti **COSMAC VIP**, illetve a **CHIP-48** [2, 3], **SUPER-CHIP 1.0** [10] (**SC10**) és **SUPER-CHIP 1.1** [11] (**SCHIP**) variánsokból származnak.

Gyakran a programok, hibásan, keverik ezeket az alapvető utasításkészlettel, ezért megszokottá vált, hogy ezeket is elérhetővé teszik, kompatibilitás céljából.

A felsorolásra kerülő alternatív utasítások (lásd **2.4. táblázat**) tetszőlegesen keverhetők a programomban.

- **COSMAC VIP**

Mint korábban említettem, a **COSMAC VIP** mikroszámítógépen a 8XYN utasítások sajátos szubrutin képzéssel működnek. Ennek az a következménye, hogy minden 8XYN utasítás kapcsolóként használja a VF regisztert. Emiatt a 8XY1, 8XY2 és 8XY3 utasítások le fogják nullázni a VF regiszter értékét.

- **CHIP-48**

Andreas Gustafsson **CHIP-48** interpreterje több szempontból is eltér az alapvető utasításkészlettől:

- a BNNN utasítást lecseréli a BXNN utasításra, amely az XNN cím plusz VX címre ugrik, az NNN cím plusz V0 helyett;
- a 8XY6 és 8XYE utasítások direkt a VX regisztert tolják el, és nem a VY regiszter értékét;
- az FX55 és FX65 utasítások az I regisztert értékét csak X-el növelik, nem $X + 1$ -el.

- **SUPER-CHIP 1.0**

A **SUPER-CHIP 1.0** alternatív utasításai megegyeznek a CHIP-48 alternatív utasításaival:

- a 8XY6 és 8XYE utasítások direkt a VX regisztert tolják el, és nem a VY regiszter értékét;
- az FX55 és FX65 utasítások az I regisztert értékét csak X-el növelik, nem $X + 1$ -el.

- **SUPER-CHIP 1.1**

A **SUPER-CHIP 1.1** 8XY6 és 8XYE alternatív utasításai megegyeznek a CHIP-48-éval, viszont az FX55 és FX65 alternatív utasításai egyáltalán nem növelik az I regiszter értékét:

- a 8XY6 és 8XYE utasítások direkt a VX regisztert tolják el, és nem a VY regiszter értékét;
- az FX55 és FX65 utasítások egyáltalán nem növelik az I regiszter értékét.

Alternatív utasítások táblázata:

Utasítás	Variáns	Mnemonik		Leírás
		Chipper	Octo	
8XY1	VIP	OR VX, VY	VX = VY	Betölti a VX OR VY művelet eredményét VX-be VF értéke 0 lesz
8XY2	VIP	AND VX, VY	VX &= VY	Betölti a VX AND VY művelet eredményét VX-be VF értéke 0 lesz
8XY3	VIP	XOR VX, VY	VX ^= VY	Betölti a VX XOR VY művelet eredményét VX-be VF értéke 0 lesz
BXNN	CHIP-48	JP VX, NNN	Jump0 NNN + VX	Az NNN cím plusz VX címre ugrik
8XY6	CHIP-48 SC10 SCHIP	SHR VX {, VY}	VX >>= VY	Egy bittel jobbra tolja VX értékét VF az eltolás előtti legalacsonyabb helyiértékű bit lesz

8X5E	CHIP-48 SC10 SCHIP	SHL VX {, VY}	VX <= VY	Egy bittel balra tolja VX értékét VF az eltolás előtti legmagasabb helyiértékű bit lesz
FX55	CHIP-48 SC10	LD [I], VX	save VX	Betölti a memóriába, az I regiszterben tárolt címtől kezdődően, V0-tól VX-ig a regiszterek értékét I regiszter értéke I + X lesz
FX65	CHIP-48 SC10	LD VX, [I]	load VX	Feltölti a memóriából, az I regiszterben tárolt címtől kezdődően, V0-tól VX-ig a regisztereket I regiszter értéke I + X lesz
FX55	SCHIP	LD [I], VX	save VX	Betölti a memóriába, az I regiszterben tárolt címtől kezdődően, V0-tól VX-ig a regiszterek értékét
FX65	SCHIP	LD VX, [I]	load VX	Feltölti a memóriából, az I regiszterben tárolt címtől kezdődően, V0-tól VX-ig a regisztereket

2.4. táblázat. *Alternatív utasítások*

A megadott alternatív utasítások **csak egy részét** képezik a variánsok utasításkészletének. A fennmaradó utasítások vagy szükségtelenek, vagy eltérő specifikációkra érvényesek.

A projektemnek nem célja a teljes körű CHIP-8 variánsok implementálása, hanem egy alap CHIP-8 rendszer egyszerű és rugalmas kialakítása, amelyet akár később tovább lehet fejleszteni.

2.8 Programok betöltése

Ahhoz, hogy utasításokat tudjunk végrehajtani, előbb kell egy program, ahonnan beolvashatjuk ezeket. Figyelni kell arra is, hogy nem minden program kompatibilis az alap CHIP-8 utasításkészlettel.

A <https://github.com/JohnEarnest/chip8Archive> linken elérhető egy kódtár, amely tartalmaz egy adatbázist; ez megpróbálja dokumentálni a CHIP-8 programok kompatibilitását. Ennek segítségével akár megoldható lehet a programok és azok futásához szükséges beállítások automatikus betöltése is.

2.9 Végrehajtási ciklus

A végrehajtási ciklus a 2 bájtos utasítások memóriából való kiolvasásával kezdődik, a **PC** által jelzett helyről. Ezt az utasítást majd dekódoljuk, így megkapjuk az utasítás opkódját⁴, és a hozzá tartozó adatokat. Ha vannak adatok, ezekkel az utasítás műveletet végez.

A végrehajtás gyorsasága gyakran változtatható, ez igaz az én implementációmra is. Rendkívül fontos megjegyezni, hogy az időzítők csökkenésének sebessége ettől független, és kötelezően 60 Hz. Ez azért fontos, mert a programok a **DT**-t használják az események időzítésére, amely ha nem az elvárt 60 Hz-es sebességgel csökken, akkor az hibás működéshez vezethet. Hasonló a probléma az **ST** esetében is, habár ez nem annyira kritikus a program működésének szempontjából.

⁴ <https://hu.wikipedia.org/wiki/Opk%C3%B3d>

3. Tervezés és implementáció

Az interpreter kivitelezése során több lehetséges úton is el lehet indulni. Törekedtem arra, hogy egy egyszerű és rugalmas implementációt biztosítsak, amely könnyedén bővíthető. A projekt GitHub kódtárja a következő linken érhető el:

<https://github.com/AE7TB99/MIC8-Interpreter>

Az elkövetkezőkben **MIC8** (Multi-Instance CHIP-8 Interpreter) néven fogok a saját implementációmra hivatkozni.

3.1 Felhasznált technológiák

A **MIC8** a következő harmadik féltől származó technológiákat használja fel:

- **Dear ImGui**

A **Dear ImGui**⁵ egy egyszerű, grafikus felhasználói felületet biztosító könyvtár a C++ nyelvhez. Optimalizált vertexpuffereket generál, amelyek könnyedén használhatóak az alkalmazásokban. Gyors, hordozható, független a rendereléstől, és önálló (nincsenek külső függőségek).

A **MIC8** felhasználói felülete a **Dear ImGui** segítségével lett megvalósítva.

- **OpenGL**

Az **OpenGL**⁶ az iparágban általánosan elfogadott 2D és 3D grafikus alkalmazásprogramozási felület⁷ (*API*), amely lehetővé teszi számos alkalmazás futtatását különböző számítógépes platformokon. Független az ablakkezelő rendszertől és az operációs rendszertől, valamint támogatja a hálózati átláthatóságot.

A **MIC8** a CHIP-8 képernyőpufferének megjelenítéséhez, míg a Dear ImGui a renderelőjeként használja az **OpenGL**-t.

⁵ <https://github.com/ocornut/imgui>

⁶ <https://www.khronos.org/opengl/>

⁷ https://hu.wikipedia.org/wiki/Alkalmaz%C3%A1sprogramoz%C3%A1si_fel%C3%BClet

- **GLFW**

A **GLFW**⁸ egy nyílt forráskódú könyvtár, amely több platformon használható az OpenGL, OpenGL ES és Vulkan asztali fejlesztéséhez. Egyszerű alkalmazásprogramozási felületet kínál ablakok, kontextusok és felületek létrehozásához, valamint bemeneti események fogadásához.

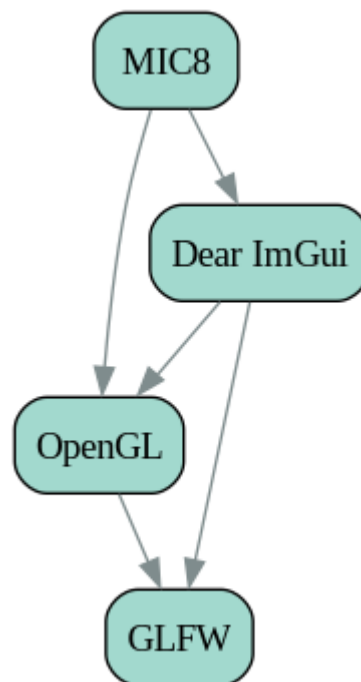
A **MIC8**-ban a **GLFW** a Dear ImGui platformjaként és az OpenGL kontextus létrehozására szolgál.

- **Emscripten**

Az **Emscripten**⁹ egy LLVM¹⁰-et használó, teljes fordító eszköztár a WebAssembly¹¹-hez, mely különös tekintettel van a sebességre, a méretre és a webes platformra.

A **MIC8**-ban az **Emscripten** felelős a C++ kód böngészőre való kompilálásáért, és az OpenGL WebGL-be való alakításáért.

A MIC8 függőségi gráfja a **3.1 ábrán** látható:



3.1. ábra. MIC8 függőségi gráf

⁸ <https://www.glfw.org/>

⁹ <https://emscripten.org/>

¹⁰ <https://llvm.org/>

¹¹ <https://webassembly.org/>

3.2 Jellemzők

Ebben az alfejezetben a MIC8 tulajdonságait ismertetem.

- **Cross-platform**

Az általam fejlesztett program kompatibilis több operációs rendszerrel és böngészővel is. A szoftver Windows, Linux és macOS platformokon egyaránt futtatható. Windows alatt a MSYS2¹² / MinGW¹³ használatával, míg böngészőben az Emscripten compiler segítségével érhető el.

- **Több instancia támogatása**

Egy különlegessége a MIC8-nak, hogy képes egyidejűleg több CHIP-8 instanciát futtatni, akár különböző utasításkészletekkel. Ez lehetővé teszi a felhasználónak, hogy gyorsan és könnyedén váltson több futó CHIP-8 program között.

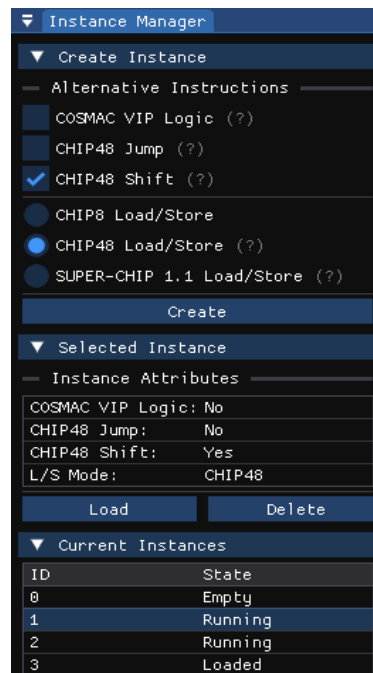
- **Felhasználói felület**

Próbáltam egy intuitív felhasználói felületet is létrehozni, amely ellátja a felhasználót néhány egyszerű és látványos eszközzel, amelyek megkönnyítik a rendszerrel való interakciót és ennek megértését.

¹² <https://www.msys2.org/>

¹³ <https://www.mingw-w64.org/>

A **3.2. ábrán** látható **instancia-menedzser ablak** teszi lehetővé a különböző instanciák létrehozását, törlését és kiválasztását, illetve kijelzi a felhasználó számára annak jelenlegi beállításait.



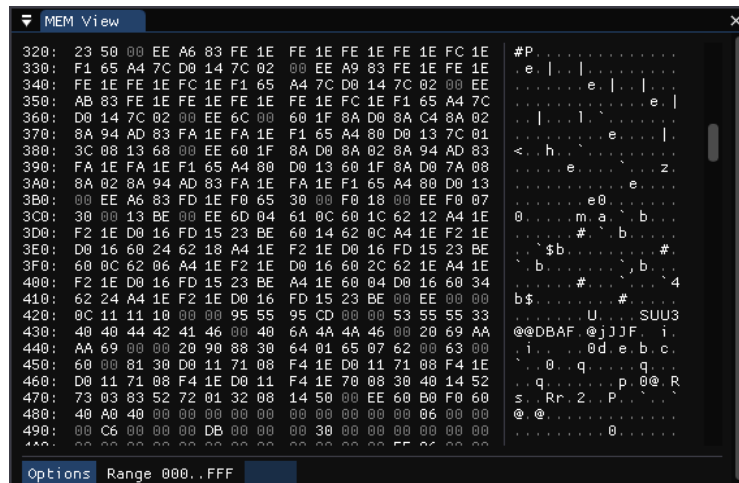
3.2. ábra. *Instancia-menedzser ablak*

A **képernyőpuffer ablakban** (lásd **3.3. ábra**) a a CHIP-8 által megjelenített grafika látható.



3.3. ábra. *Képernyőpuffer ablak*

A **memória nézet ablak** lehetővé teszi a memória megjelenítését és a PC követését (lásd **3.4. ábra**). Ezzel az eszközzel szemléltethető a CHIP-8 programok betöltése és végrehajtása.



3.4. ábra. *Memória nézet ablak*

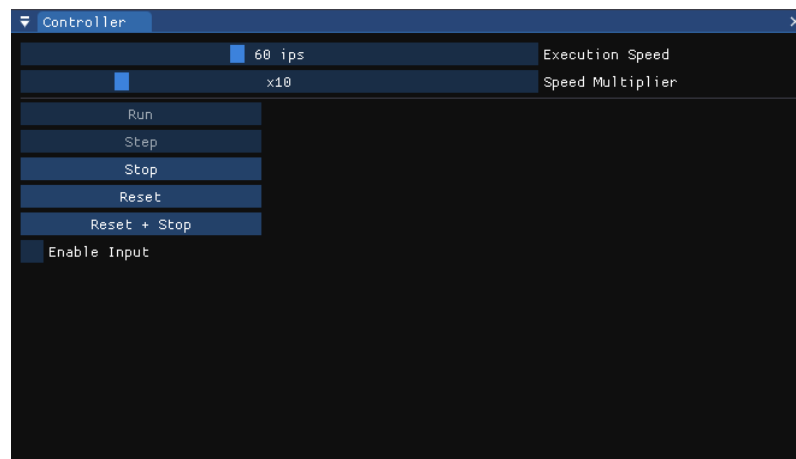
A **CPU nézet ablak** felelős a regiszterek és a stack megjelenítéséért (lásd **3.5. ábra**).

REG	VAL	LVL	ADDR
V0	0	0	0250
V1	20	1	02F2
V2	8	2	0306
V3	0	3	0306
V4	1	4	0000
V5	7	5	0000
V6	0	6	0000
V7	0	7	0000
V8	0	8	0000
V9	64	9	0000
VA	71	10	0000
VB	2	11	0000
VC	0	12	0000
VD	1	13	0000
VE	4	14	0000
VF	1	15	0000

PC: 2FE DT: 0
IR: 47C ST: 0
SP: 2

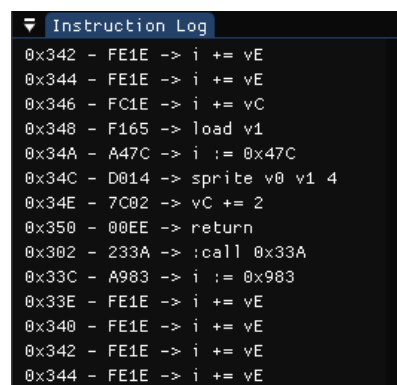
3.5. ábra. *CPU nézet ablak*

A **3.6. ábrán** látható **Controller ablakon** keresztül lehet az adott CHIP-8 instancia futtatási folyamatát és annak sebességét szabályozni, illetve engedélyezni a billentyűzet bemenetet.



3.6. ábra. *Controller ablak*

Az **utasítás napló ablak (3.7. ábra)** mutatja a végrehajtott utasítások naplóját, amely megkönnyíti a program futásának követését.



3.7. ábra. *Utasítás napló ablak*

Habár a projektem nem kínál *compiler*, *assembler*, *debugger* és hasonló komplex funkciókat, ezek nem szükségszerűek az általam kitűzött célhoz. A program még mindig fejlesztés alatt áll, később bővíthető a hiányzó funkciókkal.

3.3 Opkód dekódolás

Az opkód¹⁴ az utasításkészletek leírásában a műveleti jelrész. A CPU által beolvasott bináris szám, amit végrehajtható utasítás kódjaként értelmez.

Ötféle opkód család létezik:

1. Teljesen egyedi
2. 00E-vel kezdődők
3. 8XY-nal kezdődők
4. EX-szel kezdődők
5. FX-szel kezdődők

Az opkódok dekódolására több módszer is lehetséges [12, 13]. Ezekből a legegyszerűbbekről fogok beszélni, mint a **mintaillesztés**, a **switch utasítás** és a **metódustábla**, amelyeket bárki könnyűszerrel meg tud valósítani, illetve részletezem mindenik módszer előnyeit és hátrányait is.

Habár a CHIP-8 megvalósítható gyorsítótárral, JIT (*Just-In-Time*) vagy AOT (*Ahead-Of-Time*) technikákkal is, és rengeteg optimalizációs módszer áll rendelkezésre, ezek nem tartoznak a dolgozat tárgykörébe, mivel a téma bevezető szintű bemutatására törekszem, hogy kezdők számára is könnyen érthető és követhető legyen.

- **Mintaillesztés (Pattern matching)**

A **mintaillesztés** egy elképesztően elegáns módja az opkódok dekódolásának. Egyszerűen kifejezhető vele az összetett logika, amelyet a dekódolás igényel. Minden minta közvetlenül egy adott utasításnak felel meg, ami könnyebben érthető, olvasható és karbantartható kódot eredményez (lásd **3.1. kódrészlet**).

¹⁴ <https://hu.wikipedia.org/wiki/Opk%C3%B3d>

```

let opcode = self .mem.read_word( self .pc);
let expand = self .expand_opcode(opcode);
let nnn = expand.word(1..=3);
let nn = expand.byte(2..=3);
let x = expand[1];
let y = expand[2];
let n = expand[3];

const A: u8 = 0xA;
const B: u8 = 0xB;
const C: u8 = 0xC;
const D: u8 = 0xD;
const E: u8 = 0xE;
const F: u8 = 0xF;
let inc: bool = match expand {
    [0, 0, E, 0] => self.CLS(),
    [0, 0, E, E] => self.RET(),
    [1, _, _, _] => self.JP(nnn),
    [2, _, _, _] => self.CALL(nnn),
    [3, _, _, _] => self.SE_xnn(x, nn),
    [4, _, _, _] => self.SNE_xnn(x, nn),
    [5, _, _, _] => self.SE_xy(x, y),
    [6, _, _, _] => self.LD_xnn(x, nn),
    [7, _, _, _] => self.ADD_xnn(x, nn),
    [8, _, _, 0] => self.LD_xy(x, y),

    ...

    [F, _, 1, 8] => self.LD_stx(x),
    [F, _, 1, E] => self.ADD_ix(x),
    [F, _, 2, 9] => self.LD_fx(x),
    [F, _, 3, 3] => self.LD_bx(x),
    [F, _, 5, 5] => self.LD_ix(x),
    [F, _, 6, 5] => self.LD_xi(x),

    _ => { println ! ("Unimplemented opcode! {:04X} at {:04X}",
        opcode, self.pc); true }
};

```

3.1. kódrészlet. Dekódolás mintaillesztéssel Rust nyelvben

Sajnos egyelőre **nincs** lehetőség **mintaillesztést** használni a C++ nyelvben, habár próbálkozások vannak¹⁵, de ez a funkció még nem elérhető a standard könyvtárban.

¹⁵ <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1371r3.pdf>

- **Switch utasítás**

A **switch** utasítás a legegyszerűbb idiomatikus megoldás az opkódok dekódolására. Egyszerű és hatékony, viszont a csoportosítások dekódolásához egybeágyazott **switch** utasításokra van szükségünk, ami véleményem szerint nehezen átlátható és nehezen karbantartható. Szerencsére a CHIP-8 esetében ez a megoldás nem lesz túl hosszú, de minél több variánst vagy alternatív utasítást szeretnénk implementálni, annál bonyolultabbá válik (lásd **3.2. kódrészlet**).

```
...

switch (current_instruction >> 12) {
    case 0x0:
        switch (current_instruction) {
            case 0x00e0:
                op_00e0();
                break;
            case 0x00ee:
                op_00ee();
                break;
            [[unlikely]] default:
                op_null();
        }
        break;
    case 0x1:
        op_1nnn();
        break;
    case 0x2:
        op_2nnn();
        break;
    case 0x3:
        op_3xnn();
        break;
    case 0x4:
        op_4xnn();
        break;
    case 0x5:
        switch (current_instruction & 0xf) {
            ...

```

3.2. kódrészlet. Dekódolás switch utasítással

- **Metódustábla**

A **metódustáblák** egy jól strukturált és átlátható módot biztosítanak az opkódok dekódolására. A **metódustábla** minden egyes eleme egy függvénymutatót tartalmaz, amely a megfelelő utasítást végrehajtó metódusra mutat.

Ez a megoldás még két változatra bontható:

- **globális metódustábla**, egy nagy tábla, amely az összes utasítás opkódját tartalmazza;
- **hierarchikus metódustáblák** (lásd **3.3. kódrészlet**), amelyek az opkód részei alapján hívják meg az annak megfelelő táblát, majd legvégül az utasítást.

```
std::array<op_type, 0xF + 1> OP_ARR_MAIN = [] constexpr {
    auto OP_ARR_MAIN_ = decltype(OP_ARR_MAIN) {};
    OP_ARR_MAIN_.fill(&chip8::op_null);
    OP_ARR_MAIN_[0x0] = &chip8::op_arr_0;
    OP_ARR_MAIN_[0x1] = &chip8::op_1nnn;
    OP_ARR_MAIN_[0x2] = &chip8::op_2nnn;
    OP_ARR_MAIN_[0x3] = &chip8::op_3xnn;
    OP_ARR_MAIN_[0x4] = &chip8::op_4xnn;
    OP_ARR_MAIN_[0x5] = &chip8::op_5xy0;
    OP_ARR_MAIN_[0x6] = &chip8::op_6xnn;
    OP_ARR_MAIN_[0x7] = &chip8::op_7xnn;
    OP_ARR_MAIN_[0x8] = &chip8::op_arr_8;
    OP_ARR_MAIN_[0x9] = &chip8::op_9xy0;
    OP_ARR_MAIN_[0xA] = &chip8::op_Annn;
    OP_ARR_MAIN_[0xB] = &chip8::op_Bnnn;
    OP_ARR_MAIN_[0xC] = &chip8::op_Cxnn;
    OP_ARR_MAIN_[0xD] = &chip8::op_Dxyn;
    OP_ARR_MAIN_[0xE] = &chip8::op_arr_E;
    OP_ARR_MAIN_[0xF] = &chip8::op_arr_F;
    return OP_ARR_MAIN_;
} ();
```

3.3. kódrészlet. MIC8 fő-metódustábla

A **3.4. kódrészletben** látható a **fő-metódustáblában** található metódus, amely az FX opkódok csoportjának megfelelő maszkolással hívja meg a **3.5 . kódrészletben** látható **al-metódustábla** valamely függvényét.

```
void chip8::op_arr_0() { (this->*OP_ARR_F[instruction & 0x00FFu])(); }
```

3.4. kódrészlet. *Metódus, amely meghívja az FX csoportnak megfelelő al-metódustáblát*

```
std::array<op_type, 0x65 + 1> OP_ARR_F = [] constexpr {  
    auto OP_ARR_F_ = decltype(OP_ARR_F) {};  
    OP_ARR_F_.fill(&chip8::op_null);  
    OP_ARR_F_[0x07] = &chip8::op_Fx07;  
    OP_ARR_F_[0x0A] = &chip8::op_Fx0A;  
    OP_ARR_F_[0x15] = &chip8::op_Fx15;  
    OP_ARR_F_[0x18] = &chip8::op_Fx18;  
    OP_ARR_F_[0x1E] = &chip8::op_Fx1E;  
    OP_ARR_F_[0x29] = &chip8::op_Fx29;  
    OP_ARR_F_[0x33] = &chip8::op_Fx33;  
    OP_ARR_F_[0x55] = &chip8::op_Fx55;  
    OP_ARR_F_[0x65] = &chip8::op_Fx65;  
    return OP_ARR_F_;  
}();
```

3.5. kódrészlet. *FX csoport al-metódustáblája*

A **metódustáblák** rendkívül hasznosak a CHIP-8 esetében, mivel lehetővé teszik a metódusok kicserélését az alternatív utasítások metódusaira, akár futásidőben is. Viszont ha egy olyan rendszert implementálunk, ahol erre nincs szükség, vagy csak egy alapvető eredeti CHIP-8 interpretert szeretnénk elkészíteni, akkor ezek a **táblák** statikus konstansok is lehetnek (lásd **3.6. kódrészlet**).

```

static constexpr std::array<op_type, 0xF + 1> OP_ARR_MAIN = [] constexpr {
    auto OP_ARR_MAIN_ = decltype(OP_ARR_MAIN) {};
    OP_ARR_MAIN_.fill(&chip8::op_null);
    OP_ARR_MAIN_[0x0] = &chip8::op_arr_0;
    OP_ARR_MAIN_[0x1] = &chip8::op_1nnn;
    OP_ARR_MAIN_[0x2] = &chip8::op_2nnn;
    OP_ARR_MAIN_[0x3] = &chip8::op_3xnn;
    OP_ARR_MAIN_[0x4] = &chip8::op_4xnn;
    OP_ARR_MAIN_[0x5] = &chip8::op_5xy0;
    OP_ARR_MAIN_[0x6] = &chip8::op_6xnn;
    OP_ARR_MAIN_[0x7] = &chip8::op_7xnn;
    OP_ARR_MAIN_[0x8] = &chip8::op_arr_8;
    OP_ARR_MAIN_[0x9] = &chip8::op_9xy0;
    OP_ARR_MAIN_[0xA] = &chip8::op_Annn;
    OP_ARR_MAIN_[0xB] = &chip8::op_Bnnn;
    OP_ARR_MAIN_[0xC] = &chip8::op_Cxnn;
    OP_ARR_MAIN_[0xD] = &chip8::op_Dxyn;
    OP_ARR_MAIN_[0xE] = &chip8::op_arr_E;
    OP_ARR_MAIN_[0xF] = &chip8::op_arr_F;
    return OP_ARR_MAIN_;
} ();

```

3.6. kódrészlet. Statikus konstans fő-metódustábla

A **metódustáblák** a CHIP-8 rendszeren kívül még más sajátos előnyöket is nyújtanak:

- **kevesebb kód duplikáció:**

Különböző CPU modelleket, változatokat, mikroarchitektúrákat és ISA kiterjesztéseket lehet emulálni anélkül, hogy a kódot különböző switch utasításokban duplikálnánk, és anélkül, hogy bármit is ellenőriznénk azokban az utasításokban, esetekben vagy opkódokban, ahol a CPU modelltől függő változásokra lenne szükség, amivel lehetséges elkerülni az ellenőrzések teljesítmény veszteségét.

Egy Z80 emulátornál például a **metódustáblák** vezetnek a legjobb megoldáshoz, mivel ez lehetővé teszi szinte az összes utasításkód újrafelhasználását a 0 megszakítási mód implementálásakor.

- **optimalizálás:**

Gyorsíthatunk a rendszeren azzal, hogy elkülönítődik az utasítások végrehajtása a megszakításkezeléstől. Kétféle dekódolás jön létre. Az egyik az utasítások végrehajtása előtt minden emulációs ciklusban ellenőrzi a megszakításokat, míg a másik csak egyszerűen végrehajtja az utasításokat. Így lehetséges az, hogy csak azokban az órajel-ciklus-intervallumokban ellenőrizzük a megszakításokat, amikor a hardver ténylegesen generálja őket.

3.4 Egyéb részletek

A projekt alatt érdemes figyelni arra, hogy amit lehet, azt már fordítási időben végezzünk el. Ilyen például a fontset elhelyezése a memóriában (lásd **3.7. kódrészlet**).

```
std::array<std::uint8_t, MEM_SIZE> mem = [] consteval {
    auto mem_ = decltype(mem) {};
    std::array<std::uint8_t, FONTSET_SIZE> fontset{
        0xF0, 0x90, 0x90, 0x90, 0xF0,
        0x20, 0x60, 0x20, 0x20, 0x70,
        0xF0, 0x10, 0xF0, 0x80, 0xF0,
        0xF0, 0x10, 0xF0, 0x10, 0xF0,
        0x90, 0x90, 0xF0, 0x10, 0x10,
        0xF0, 0x80, 0xF0, 0x10, 0xF0,
        0xF0, 0x80, 0xF0, 0x90, 0xF0,
        0xF0, 0x10, 0x20, 0x40, 0x40,
        0xF0, 0x90, 0xF0, 0x90, 0xF0,
        0xF0, 0x90, 0xF0, 0x10, 0xF0,
        0xF0, 0x90, 0xF0, 0x90, 0x90,
        0xE0, 0x90, 0xE0, 0x90, 0xE0,
        0xF0, 0x80, 0x80, 0x80, 0xF0,
        0xE0, 0x90, 0x90, 0x90, 0xE0,
        0xF0, 0x80, 0xF0, 0x80, 0xF0,
        0xF0, 0x80, 0xF0, 0x80, 0x80
    };
    std::ranges::copy(fontset, mem_.begin() + FONTSET_ADDR);
    return mem_;
}();
```

3.7. kódrészlet. Fontset memóriába helyezése fordítási időben

Még egy érdekes optimalizáció amit el lehet végezni, a *template* paraméterek (lásd **3.8 kódrészlet**) által, kompilálás ideje alatt kiválasztható rendszer paraméterek és alternatív utasítások. A kívánt viselkedést lehet a dekódoló struktúrában, vagy akár az adott utasításon belül is szabályozni (lásd **3.9 kódrészlet**).

```
template<mode_options mode = mode_options::DEFAULT_MODE,
        behavior_options behavior =
            behavior_options::MEMORY_INCREMENT_BY_X |
            behavior_options::MEMORY_INCREMENT_BY_1,
        framebuffer_size_options framebuffer_size =
            framebuffer_size_options::DEFAULT_SIZE,
        memory_size_options memory_size =
            memory_size_options::DEFAULT_SIZE,
        stack_size_options stack_size = stack_size_options::DEFAULT_SIZE,
        font_options font = font_options::VIP>
class core { ... }
```

3.8 kódrészlet. core osztály template paraméterei

```

auto op_8xy1() -> void {
    const std::uint8_t x = (current_instruction & 0x0F00u) >> 8u;
    const std::uint8_t y = (current_instruction & 0x00F0u) >> 4u;

    registers[x] |= registers[y];

    if constexpr (static_cast<bool>(behavior &
        behavior_options::LOGIC_QUIRK)) registers[0xf] = 0;
}

```

3.9 kódrészlet. Alternatív viselkedés template paraméterek segítségével

Ez a technika csak akkor éri meg igazán, ha több alternatív viselkedést szeretnénk implementálni.

3.5 Dekódoló módszerek összehasonlítása

A korábbiakban bemutatottak alapján a metódustáblák alkalmazása számos szempontból előnyös: rugalmasak, könnyen bővíthetők, és egyszerűsítik a kód karbantartását. Ugyanakkor érdemes megvizsgálni, hogy teljesítmény szempontjából hogyan viszonyulnak a switch utasítással történő dekódoláshoz. Ennek megválaszolására összehasonlító mérést végeztem, amely rögzíti a két módszer futási eredményeit.

A mérés elvégzéséhez a **nanobench**¹⁶ platformfüggetlen microbenchmarking könyvtárat, míg a rendszer beállításához a **pyperf**¹⁷ modult használtam. A futtatott CHIP-8 program, a **1dcell** (Ian Douglas, 2014. okt. 26), amely egy egyszerű, egydimenziós sejtautomata szimulációját valósítja meg, ami megfelelő terhelést biztosít a CHIP-8 számára.

A mérések eredményei (lásd **3.1 táblázat**) szerint a switch utasítás használata 114,6%-kal gyorsabb a metódustábláknál. Ennek oka, hogy a kompilátor a switch utasítás esetén optimalizációval ugrótáblát hozhat létre (lásd **3.10 kódrészlet**), míg a metódustáblák esetén nincs optimalizációs lehetőség.

¹⁶ <https://nanobench.ankerl.com/>

¹⁷ <https://github.com/psf/pyperf>

Módszer	rel	ns/op	op/s	err%	ins/op	bra/op	miss%	total
Met. tábla	100,0%	6,19	161.622.929,54	0,6%	36,26	6,92	0,6%	0,62
Switch	214,6%	2,88	346.865.372,43	0,7%	30,77	5,75	0,8%	0,29

3.1 táblázat. *Nanobench eredmények*¹⁸. A táblázat a Metódus tábla és Switch módszerek méréseit mutatja: rel: relatív teljesítmény, az első módszerhez képest normalizálva, ns/op: egy művelet elvégzéséhez szükséges idő (ns-ban), op/s: műveletek másodpercenként, err%: a mérések hibaszázaléka, ins/op: átlagosan végrehajtott utasítások száma egy művelet során, bra/op: átlagos elágazások száma műveletenként, miss%: az elágazás-előrejelzés hibaaránya, total: a mérés teljes futási ideje.

```

movzx    ebx, word ptr [rdi + 6224]
mov     eax, ebx
shr     eax, 12
jmp     qword ptr [8*rax + .LJTI55_0]
.LBB55_1:

```

3.10 kódrészlet. Switch utasítás - assembly

A CHIP-8 közösségben az implementációk közti összehasonlítást utasítás képkockánként, **IPF** (*instructions per frame*), illetve millió utasítás másodpercenként, **MIPS** (*million instructions per second*) alapján szokták végezni, melyek 60 Hz-en (1 képkocka ideje $\approx 16667 \mu s$) belül maximális számú utasítás végrehajtásokból számolódnak ki. Az eredmények (lásd **3.2 táblázat**) várhatóak voltak, és igazolják az előző mérés eredményeit, miszerint a metódus táblák jóval lassúbbak a switch utasítás segítségével megvalósított dekódolásnál. Emellett meggyőződtem, hogy az egyszerű interpreter teljesítménye hasonló más átlagos implementációkhoz képest, viszont nem ér el az optimalizált, tradicionális (nem JIT vagy AOT) interpreterek szintjét, amelyek jelenlegi rekordja $\approx 8 \text{ MIPF} < 500 \text{ MIPS}$

Módszer	MIPS	IPF
Metódus tábla	160,98	2.683.028,10
Switch	353,28	5.888.020,00

3.2 táblázat. *Dekódoló módszerek MIPS és IPF táblázata*¹⁸

¹⁸ A mérések előtt a rendszert a `pyp perf system tune` parancs segítségével hangoltam.
Rendszer specifikációk: AMD Ryzen 7 6800HS (16) @ 4,7 GHz
Kompilátor: LLVM/Clang 19.1.0 | Használt kapcsolók: `-std=c++23 -O3 -flto=thin`

A template paraméterek használata alternatív viselkedés kiválasztásához ilyen kevés utasítás esetén nem indokolt, mivel csak bonyolítaná a programot, miközben a teljesítményjavulás elhanyagolható lenne. Ennek a módszernek a mérésével ezért nem foglalkoztam.

4. Összehasonlítás más implementációkkal

Az interneten számos interpreter, sőt teljes CHIP-8 integrált fejlesztői környezet (IDE) is található. A legtöbb projekt viszont megáll az egyszerű parancssoros alkalmazásnál és egy ehhez tartozó megjelenő képernyőnél. Ez szerintem teljesen rendben van, hiszen a CHIP-8 projektek elsődleges célja a fejlesztés ideje alatti tanulás és felfedezés.

A következőkben bemutatok egy pár, mások által készített CHIP-8 implementációt.

Steffen Schümann (Gulrak) - Cadmium - <https://github.com/gulrak/cadmium>

A **Cadmium** az egyik legfejlettebb CHIP-8 interpreter, amely több variánst is támogat. A Cadmium egy mérföldkő a CHIP-8 terén, hiszen valószínűleg ez az első interpreter, amely képes a CHIP-8-at az eredeti COSMAC VIP-en implementált interpreter viselkedésével és időzítésével futtatni. Habár ebben a módban nem képes az interpreter gépi nyelvi szubrutinok futtatására, sokkal kisebb az erőforrás igénye, mint egy tényleges COSMAC VIP emulátornak.

John Earnest - Octo / C-Octo - <https://github.com/JohnEarnest/Octo> - <https://github.com/JohnEarnest/c-octo>

Az **Octo** egy modern IDE a CHIP-8 fejlesztéséhez, amely a böngészőben fut. Több magas szintű eszközzel rendelkezik, mint például *assembler*, *debugger* és *sprite*-szerkesztő. Az **Octo** a legelterjedtebb módja a CHIP-8 programok írásának, több variánst is támogat, és egyfajta standardként helyezkedik el a CHIP-8 programok terén. Az **Octo** kizárólag a böngészőben fut, ezért létrejött egy külön projekt, a **C-Octo**, amely egy natív programot kínál.

Les Harris - Dorito - <https://github.com/lesharris/dorito>

A **Dorito** egy cross platform modern IDE, az Octo-hoz hasonló, viszont natívan fut, és kínál 1-2 eszközt, ami nincs jelen Octo-ban, habár ez visszafelé is igaz. Mint az Octo, ez is több variánst támogat és kiváló fejlesztési lehetőségeket kínál.

Tim Franssen (Timendus) - Silicon8 - <https://github.com/Timendus/silicon8>

A **Silicon8** egy Go-ban írt nagyon hangulatos interpreter, hiszen egy kis TV-ben látható a futó program. Habár több variánst is kínál, nem képes a különböző beállítások között váltani, csak előre definiált beállításokat képes használni.

A **4.1. táblázat** a MIC8 implementáció és a fent bemutatott implementációk összehasonlítását összegezi.

Név	Platform				Dekódolás módja	Több instancia támogatása	Több variáns támogatása	Változtatható beállítások
	Linux	macOS	Windows	Web				
MIC8	✓	✓	✓	✓	Metódustábla alapú	✓	✗	✓
Cadmium	✓	✓	✓	✓	Metódustábla alapú	✗	✓	✓
Octo	✓	✓	✓	✓	Switch alapú	✗	✓	✓
Dorito	✓	✓	✓	✗	Switch alapú	✗	✓	✓
Silicon8	✓	✓	✓	✓	Switch alapú	✗	✓	✗

4.1. táblázat. *CHIP-8 implementációk összehasonlítása*

5. Összegzés

Összefoglalva, egy CHIP-8 interpreter fejlesztése értékes betekintést nyújt az érdeklődők számára az emuláció és az alacsony szintű programozás területébe. A CHIP-8 specifikációinak bemutatásával felhívtam a figyelmet erre az egyszerű, de történelmileg is jelentős rendszerre, amely kiindulópontként szolgálhat azok számára, akik érdeklődnek az emuláció iránt, ám korábban nem foglalkoztak vele.

A projekt során lehetőség nyílik különböző programozási technikák, akár a kompiláció idejű programozás és a metaprogramozás megismerésére is, habár ezek alkalmazása nem feltétlenül szükséges. A végzett mérések igazolták, hogy a switch utasítás lényegesen gyorsabb, mint a metódustáblák használata, noha a metódustáblák rugalmasabbak és átláthatóbbak, ami bizonyos esetekben előnyösebbé teszi őket, különösen, ha nincs szükség extra teljesítményre.

A CHIP-8 vizsgálata során értékes tapasztalatokat szereztem a CPU emuláció, az opkód-dekódolás és a minimalista grafikus rendszerek működéséről. Ez a projekt gyakorlati oktatási forrást jelenthet azoknak a diákoknak, programozóknak és emuláció iránt érdeklődőknek, akik mélyíteni szeretnék ismereteiket ezen a téren, és fejleszteni kívánják programozási készségeiket a fejlesztés során felmerülő problémák megoldásával. Az interpreter bővíthetősége és továbbfejlesztése lehetőséget biztosít további CHIP-8 variánsok implementálására, és akár oktatási célokra is alkalmazható.

A MIC8 GitHub linkje: <https://github.com/AE7TB99/MIC8-Interpreter>

Hivatkozások

- [1] Wikipédia szerkesztők, „CHIP-8,” *Wikipédia, A szabad enciklopédia*. [Online].
Elérhető: <https://en.wikipedia.org/wiki/CHIP-8>. [Hozzáférés dátuma: 2024. ápr. 20.].
- [2] A. Gustafsson, „CHIP48 - A CHIP-8 interpreter for the HP48SX (version 2.20),”
USENET: comp.sys.handhelds, 1990. szept. 12. Elérhető:
<https://groups.google.com/g/comp.sys.handhelds/c/zv7XZKIDS34>. [Hozzáférés
dátuma: 2024. ápr. 20.].
- [3] A. Gustafsson, „Source code to CHIP-48 version 2.25,” USENET:
comp.sys.handhelds, 1990. okt. 6. Elérhető:
<https://groups.google.com/g/comp.sys.handhelds/c/1hEF9Da2RJk>. [Hozzáférés
dátuma: 2024. ápr. 20.].
- [4] RCA Solid State Division, *RCA COSMAC VIP CDP18S711 Instruction Manual*, RCA
Corporation, Somerville, N.J. 08876, USA, 1978. febr. Elérhető:
https://storage.googleapis.com/wzukusers/user-34724694/documents/130d458901764787969269f48aeeee2a/VIP_Manual.pdf. [Hozzáférés dátuma: 2024. ápr. 20.].
- [5] T. Laudereau, R. Simpson, and T. Swan, *VIPER*, Aresco, vol. 1, no. 1, 1978. jún.
Elérhető: <https://github.com/mattmikolay/viper/blob/master/volume1/issue1.pdf>.
[Hozzáférés dátuma: 2024. ápr. 20.].
- [6] T. Laudereau, R. Simpson, and T. Swan, *VIPER*, Aresco, vol. 1, no. 2, 1978. aug.
Elérhető: <https://github.com/mattmikolay/viper/blob/master/volume1/issue2.pdf>.
[Hozzáférés dátuma: 2024. ápr. 20.].
- [7] A. Morlan, „Building a CHIP-8 Emulator [C++],” *Austinmorlan.com*, 2019. aug. 28.
[Online]. Elérhető:
https://austinmorlan.com/posts/chip8_emulator/#64x32-monochrome-display-memory.
[Hozzáférés dátuma: 2024. ápr. 20.].

- [8] C. Egeberg, „Chipper V2.11 Documentation,” USENET: comp.sys.hp48, 1991. szept. 2. Elérhető:
<https://groups.google.com/g/comp.sys.hp48/c/e7In51mOgHY/m/8tR3ZKeX9FUJ>.
[Hozzáférés dátuma: 2024. ápr. 20.].
- [9] J. Earnest, *Octo*. [Szoftver]. GitHub, 2015. Elérhető:
<https://github.com/JohnEarnest/Octo/blob/gh-pages/docs/Manual.md>. [Hozzáférés dátuma: 2024. ápr. 20.].
- [10] E. Bryntse, „SUPER-CHIP: A new CHIP interpreter (the RIGHT one),” USENET: comp.sys.handhelds, 1991. máj. 16. Elérhető:
<https://groups.google.com/g/comp.sys.handhelds/c/fPUzuAkDdVs/m/k76waUy2ECkJ>.
[Hozzáférés dátuma: 2024. ápr. 20.].
- [11] E. Bryntse, „SUPER-CHIP v1.1 (now with scrolling),” USENET: comp.sys.handhelds, 1991. máj. 24. Elérhető:
https://groups.google.com/g/comp.sys.handhelds/c/sDY9zFb6KUo/m/JcYBK2_yerMJ.
[Hozzáférés dátuma: 2024. ápr. 20.].
- [12] V. Moya del Barrio, *Study of the techniques for emulation programming*, Doktori disszertáció, Politècnica de Catalunya Egyetem, 2001. jún. Elérhető:
<http://www.codeslinger.co.uk/files/emu.pdf> [Hozzáférés dátuma: 2024. ápr. 20.].
- [13] D. Mihocka, „NO EXECUTE!,” *emulators.com*, 2008. szept. 3. [Online]. Elérhető:
http://www.emulators.com/docs/nx25_nostradamus.htm [Hozzáférés dátuma: 2024. ápr. 20.].