Krystian Skibiński

Emulator maszyny wirtualnej CHIP-8
CHIP-8 virtual machine emulator

Publiczna wersja pracy inżynierskiej nie posiada ona żadnych danych uniwersytetu.

Wrocław, 2019

Spis treści

Streszczenie

Celem pracy jest przygotowanie interpretera maszyny wirtualnej CHIP-8, za pomocą którego możliwe będzie emulowanie programów napisanych dla tego komputera. Projekt zostanie zrealizowany przy użyciu języka skryptowego - Python wraz z wykorzystaniem biblioteki Tkinter oraz dokumentacją i testami.

Abstract

PyChip8 is a CHIP-8 computer interpreter written in Python, which allows you to run programs prepared for this machine directly on your computer.

1 Wstęp

W dobie błyskawicznego rozwoju internetu i komputerów, maszyny potrzebne do przetwarzania dużych porcji informacji są coraz bardziej skomplikowane. Z jednej strony, ważne są jak najmniejsze rozmiary tych urządzeń, z drugiej ceni się ich wydajność. To prowadzi do tworzenia coraz bardziej skomplikowanych podzespołów i systemów je napędzających. Przeciętna instrukcja komputera z lat 80. zajmowała około 150 stron[1], a jej zawartość opisywała budowę i działanie urządzenia, włącznie z zaprezentowaniem wszystkich instrukcji wbudowanego języka programowania, wyświetlaniem i manipulacją grafiki, czy dźwięku. Dzisiejsze instrukcje dla samych procesorów zawierają 5000 stron[2], a jest to jeden z elementów jednostki centralnej, poza tym pozostaje jeszcze system. Z tego powodu współczesne tworzenie oprogramowania opiera się na jeszcze większej abstrakcji, zestawach bibliotek i narzędzi. Nie mniej znajomość działania procesora, czy pamięci komputera, nadal stanowi istotną kwestie, dla twórców oprogramowania i inżynierów. Pod warstwą obecnie dostarczanych, modułów i pakietów, nadal są przeprowadzane typowe dla wczesnych komputerów operacje. Stanowią one dobre źródło do tego, aby zagłębić się w budowę współczesnych architektur.

Projekt ten powstał w celach hobbistycznych, jako wynik fascynacji 8-bitowymi komputerami i chęci zbudowania własnej jednostki. Jego założeniem jest przybliżenie użytkownikowi działania prostej architektury wczesnych elektronicznych maszyn cyfrowych. Dlatego został on upubliczniony jako *wolne oprogramowanie* w serwisie *github* [3] na *licencji MIT*[4]. W jego ramach było przygotowanie emulatora maszyny wirtualnej CHIP-8. Opracowany program działa jako aplikacja okienkowa, która odtwarza napisane w tym języku pliki binarne, wraz z emulacją grafiki, dźwięku i obsługą klawiatury.

W rozdziale drugim została poruszona kwestia komputerów 8-bitowych, ich historia, w tym również omawianej maszyny wirtualnej CHIP-8. Następnie w rozdziale trzecim przedstawiono zagadnienia teoretyczne związane z pracą. Na początku omówiono maszynę wirtualną, związane z nią zagadnienia, takie jak emulacja, czy wirtualizacją, przedstawiono różnicę między tymi pojęciami, następnie opisano zagadnienia potrzebne do opracowania własnego emulatora, takie jak przesunięcia bitowe, stosy, czy rejestry. Rozdział czwarty został poświęcony specyfikacji omawianego CHIP-8, jest to abstrakcyjna instrukcja, którą musiał spełniać komputer, na którym chciano zaimplementować ten system. W tym rozdziale poruszono budowę jego pamięci, procesora, wspieranych urządzeń wejścia, wyjścia. W rozdziale piątym przedstawiono specyfikację projektu, to jakie problemy niesie za sobą budowa emulatora, jego architekturę, strukturę klas, a także wymagania funkcyjne i niefunkcyjne, wraz z schematami użycia stworzonego programu. Następnie w rozdziale szóstym zawarto implementacje projektu, opisano proces budowy poszczególnych elementów maszyny wirtualnej, a także przedstawiono implementacje niektórych z instrukcji procesora. Rozdział siódmy poświęcony był napisanym testom. Pokrótce zaprezentowana została ich struktura i

[1]Instrukcja dla programistów najpopularniejszego komputera lat 80 - Commodore 64 zawiera 491 strony [2].

[2]Instrukcja dla projektantów oprogramowania, procesora Intel o architekturze i32 zawiera 4670 stron [6].

[3]Adres do repozytorium: github.com/krskibin/pyChip8

[4]Więcej o licencji *MIT* na stronie: https://opensource.org/licenses/MIT

omówiona najważniejsze techniki odpowiedzialne za ich poprawne przygotowanie. Rozdział ósmy, to z kolei, podsumowanie pracy, a także przedstawienie kolejnych etapów jej rozwoju.

2 Komputery 8-bitowe

2.1 Architektura 8-bitowa

To architektura komputera, w której używa się 8-bitowych rejestrów, jednostek arytmetyczno-logicznych i komórek pamięci [19]. Ta ilość danych była jednak niewystarczająca przy adresowaniu pamięci, dlatego komputery te posiadały najczęściej 16-bitową *szynę pamięci* [19], która pozwala na indeksowanie komórek do *64kB* [5]. Wówczas rozszerzano, również wielkość instrukcji i niektórych rejestrów do wartości 16-bitowych.

2.2 Historia

Rozwój komputerów rozpoczął się na dobre po drugiej wojnie światowej. W 1947 roku został wynaleziony tranzystor bipolarny, który od 1955 roku zaczął zastępować duże lampy elektronowe, znacząco zmniejszając rozmiary komputerów. Odtąd tranzystory mogły posiadać tysiące bramek logicznych przy relatywnie małym zużyciu miejsca. Następnym wielkim odkryciem, było wprowadzenie układów scalonych. Jack Kilby opracował pierwszy działający czip. Pół roku później Robert Noyce [9], zaprezentował swój układ, w którym rozwiązał wiele problemów jego konkurenta, największą zmianą było zastosowanie krzemu, zamiast germanu. Rozpoczęły się wyścigi firm technologicznych, aby stworzyć jak mniej zasobożerne jednostki. W 1968 roku Noyce wraz z Gordon E. Moore'em (późniejszym twórcą prawa Moore'a) otwierają firmę *Intel*, zaledwie trzy lata później wypuszczają na rynek pierwszy komercyjny [19] mikroprocesor *Intel 4004*, który z uwagi na małą moc obliczeniową trafia głównie do kalkulatorów [9], jednak jego twórcy pozostawiają innym możliwość jego przeprogramowania. Dało możliwość pierwszym hobbistom elektroniki na wykorzystanie tych czipów we własnym zakresie. Dopiero drugi produkt *Intela* przyniósł rewolucję. Zrezygnowano z architektury 4-bitowej na rzecz 8-bitowej, która w tamtych czasach była jedynie opracowana teoretycznie. *Intel 8008* ugruntował pozycje firmy i przyniósł ogromne zyski. Pierwotnie procesor był zaprojektowany dla firmy *CTC*, która chciała ich użyć do swoich terminali. *Datapoint 2200* [19] miał służyć do odczytywania danych z jednostek typu mainframe, czyli ogromnych stacji służących do krytycznych obliczeń finansowych lub statystycznych. Dzięki wykorzystaniu charakterystyki dostarczonej przez swoich partnerów, firma z Santa Clara opracowała technologię, która mogła napędzić pierwsze komputery osobiste.

Zanim jeszcze zaczęto sprzedawać pierwsze produkty *Intela*, Joseph Weisbecker w 1971 roku pracował przy projekcie procesora 8-bitowego [5], który został przerwany po premierze *i4004*. Jednostka sprzedawała się na tyle dobrze, że firma w której pracował Weisbecker, RCA zgodziła na kontynuowanie prac nad własnym czipem. Niedługo potem, zaprezentowano pierwszy procesor tej firmy, stworzony w technologii *CMOS* (ang. *Complementary Metal-Oxide Semiconductor*), złożony z dwóch układów *COSMAC 1801U* i *1801R* [5], rok później połączono je w jeden chip o nazwie *COSMAC 1802* . W między czasie (1975 rok), Weisbecker wydał zestaw edukacyjny pod nazwą RCA Microtut, zestaw do samodzielnego złożenia (ang. *Do-It-Yourself Computer*), któ-

[5] Pamięć była maksymalnie adresowana od 0 do 2^{16} czyli 65536 bitów.

ry miał uczyć podstaw budowy komputera i programowania. Niedługo później *RCA* wydało konsole *RCA Studio II*, powstałą na wniosek Weisbeckera. Opierał się na wcześniej wspominanym czipie *COSMAC 1802*. Komputer nie odniósł większego sukcesu, prawdopodobnie z uwagi na brak kontrolerów, zamiast, których użyto klawiatury heksadecymalnej [4]. Sprzęt ten posłużył córce Josepha, Joyce Weisbecker, która nauczyła się programować dzięki tej konsoli. W historii zapisała się jako pierwsza kobieta, która tworzyła gry komercyjne [5]. W 1976 roku czasopismo *Popular Electronics* wydało opracowanych przez Weisbeckera schemat budowy komputera *COSMAC ELF*, będącą blisko powiązaną z *Microtut*jednostką zaprojektowaną dla hobbistów, którzy bez większych przeszkód mogli złożyć go w swoim domu. W późniejszej serii magazyn dodał do urządzenia prostą kartę graficzną opartą o tę z konsol *Studio II* [5]. W tym samym roku *RCA*, oddało do sprzedaży następny komputer, *COSMAC VIP*, który w głównej mierze opierał się na *COSMAC ELF*. Posiadał on wbudowany język programowania zaprojektowany przez Weisbeckera - CHIP-8 [4].

3 Emulacja

3.1 Wirtualizacja, a emulacja

Pomimo swojej złożoności systemy komputerowe nadal ewoluują, jest to możliwe dzięki dobrze zaprojektowanym interfejsom, które oddzielają poziomy abstrakcji, przez co niepotrzebna jest szczegółowa znajomość implementacji niższego poziomu, ułatwia to w niezależny sposób rozwój warstwy sprzętowej jak i oprogramowania. Niestety ma to swoje ograniczenia, aplikacje, które są dystrybuowane jako skomplikowane pliki binarne zależą od konkretnego interfejsu systemu operacyjnego [7]. Istnieją dwie główne metody, które zapewniają sposób na obejście takich ograniczeń. Pierwsza z nich to wirtualizacja. Jej zadaniem jest ominięcie warstwy sprzętowej lub systemowej i odpowiednie jej zmapowanie na inny system lub architekturę sprzętową. Sam koncept wirtualizacji nie musi wyłącznie odnosić się do poszczególnych podzespołów lub podsystemów, ale do całych maszyn. Wówczas programiści muszą dostarczyć oprogramowanie, które wspiera całą architekturę urządzenia[7, 10], tworząc w ten sposób maszynę wirtualną. Dzięki temu omijają oni rzeczywistą kompatybilność urządzeń, jaki i ograniczenia zasobów sprzętowych. Innym podejściem na wyabstrahowanie jednego systemu w drugim jest emulacja, która w swoich założeniach również opiera się na maszynie wirtualnej, ale zamiast dostarczać wirtualne interfejsy jak w przypadku wirtualizacji, symuluje działanie całej architektury urządzenia. Główna różnica polega na tym, że w pierwszym przypadku zestaw instrukcji procesora (*ISA*) gospodarza [7] jest taki sam, lub w większości podobny do tego w maszynie wirtualnej. Niepasujące do zestawu hosta kody procesora są przechwytywane i wykonywane przez oprogramowanie [17]. Z kolei podczas emulacji implementuje się maszynę wirtualną na komputerze gospodarza, którego zestaw instrukcji jest inny od tego z bazowego sprzętu (gościa). W tym przypadku żadna z instrukcji nie jest bezpośrednio przetwarzana przez procesor gospodarza, tylko interpretowana, za pomocą specjalnego programu (interpretera) [7]. Wyodrębnia on każdy ciąg bitów, następnie odkodowuje i emuluje ich wykonanie. Oprogramowanie hosta, które wykonuje tę interpretacje, naśladując zestaw instrukcji maszyny wirtualnej i jej wirtualną konfigurację sprzętową, jest nazywane emulatorem [17]. Niestety cały ten proces jest relatywnie wolny. W latach 70, twierdzono nawet, że emulacja nie jest powiązana z tworzeniem maszyny wirtualnej, ponieważ proces ten nie jest wystarczająco wydajny, aby mógł on dostatecznie symulować działanie danego urządzenia [10]:

> "The second characteristic of a virtual machine monitor is efficiency. It demands that a statistically dominant subset of the virtual processor's instructions be executed directly by the real processor, with no software intervention by the VMM. This statement rules out traditional emulators and complete software interpreters (simulators) from the virtual machine umbrella."

Jednak z początkiem lat 90. coraz popularniejsze stawały się komputery oparte na architekturze 32-bitowej, które z łatwością mogły emulować 8-bitowe systemy z końca lat 70.

3.2 Bity i bajty

Pomimo swojej złożoności, istotą komputera jest przetwarzanie informacji. To elektryczna maszyna, która może rozpoznać tylko dwa stany napięciowe; brak lub bardzo niskie napięcie i wysokie napięcie, generując przy tym sygnał ciągły (analogowy), który zamieniany jest na sygnał cyfrowy, czyli skwantowany. Przyjęto, że sygnał ten będzie reprezentowany w formie zero-jedynkowej, jeden gdy napięcie jest wysokie i zero gdy jest niskie. Taki system przedstawiania cyfr nazywany jest dwójkowym lub binarnym. Już jedna liczba binarna może przedstawić, w którym z dwóch możliwych stanów jest dany układ, dlatego jest ona uznawana za najmniejszą cząstkę informacji możliwą do przetworzenia przez komputer, dzięki czemu zyskała miano bitu. Inną z podstawowych funkcji komputera jest możliwość przechowywania informacji. Dawniej zdecydowano, że najmniejszą ilością przechowywanych bitów w jednym adresie komórki pamięci będzie bajt, który z początku nie miał określonej liczby [9], dopiero później ustalono, że będzie to 8 bitów. Natomiast, bity o określonej długości na których operacje wykonuje procesor nazwano *słowem*. Jeśli słowo tworzą 32 bity, to taki procesor określany jest jako 32-bitowy. Niestety, przy dużej ilości danych generowanych przez komputery ich przedstawianie za pomocą systemu dwójkowego nie jest czytelne dla człowieka. Ze względu na chęć zachowania większej zwięzłości, w literaturze informatycznej, zdecydowano na zapisywanie ich za pomocą systemu szesnastkowego [9].

Użycie systemu szesnastkowego ułatwia identyfikowanie poszczególnych komórek pamięci czy wartości rejestrów.
Wszystkie instrukcje przetwarzane przez procesor CHIP-8, również są zapisane w takiej formie. Dzięki temu osoby piszące oprogramowanie na tę jednostkę zamiast używać *słów* o długości 16 bitów, mogą z powodzeniem zapisywać je w formie czteroznakowej. Dodatkowo w instrukcji [14] zastosowano podział na mniejsze jednostki informacji niż bajty.

Do reprezentacji 4-bitowej (jeden znak w systemie szesnastkowym) użyto *półbajta* (ang. *nibble*, *nybble*), z czego bity 7-4 nazywane są *młodszymi*, a bity 0-3 *starszymi* [4]. Aby łatwiej było identyfikować różne rejestry w jednej instrukcji, wprowadzono oznaczenia x, dla młodszych bitów pierwszego bajtu kodu procesora i y dla starszych bitów drugiego. Z kolei wartość 12-bitowa jest określana jako adres, ponieważ za jej pomocą identyfikuje się komórki pamięci w komputerze.

3.3 Operatory binarne

Operowanie na informacji przez procesor nie sprowadza się jedynie do prostych działań matematycznych. Jednostka obliczeniowa wykonuje też operacje binarne, które nie tylko służą programistom do manipulacji zmiennymi, ale również wykorzystywane są przez sam procesor do odpowiedniego pozyskiwania i interpretowania instrukcji. Wśród operatorów bitowych dostępnych w większości współczesnych języków programowania można wyróżnić, *jednoargumentowe*, do których należy negacja, reprezentowana najczęściej za pomocą tyldy ("∼"), lub te wieloargumentowe w skład których wchodzą: koniunkcja ("&"), alternatywa ("|"), alternatywa wykluczająca ("^"), przesunięcia bitowe ('»' lub '«').

3.4 Asembler i deasembler

W związku z tym, że każdy zestaw kodów operacyjnych dla procesorów różni się w zależności od ich architektury i producenta, powstała cała grupa niskopoziomowych języków programowania nazywana *językami asemblera*, w których każda instrukcja jest mnemonikiem[6] kodu procesora. Proces zamiany zrozumiałego dla człowieka kodu na maszynowy jest dokonywany za pomocą specjalnego programu, który nazywany jest *asemblerem*. Instrukcja *COSMAC VIP* [18] nie posiada składni dla języka assemblera, jednak w *Cowgod's Chip-8 Technical Reference* zamieszczone zostały mnemoniczne nazwy poszczególnych instrukcji, które posłużą do nazwania odpowiadających im funkcji w projekcie emulatora. Dobrą praktyką jest tworzenie programów nazywanych deasemblerami, które przetwarzają kod pliku binarnego na ten w postaci mnemonicznej, zrozumiałej dla człowieka. Ich podstawowym celem jest dokładniejsze zrozumienie działania emulowanego urządzenia. Tego typu programy często stanowią podstawę dla pełnoprawnego emulatora.

3.5 Rejestry i stosy

Rejestry są to pojemniki na dane o niewielkich rozmiarach służące do przetrzymywania krótkotrwałego wyniku operacji, adresów lokacji w pamięci operacyjnej itd. Dostęp do nich, w porównaniu między innymi do pamięci RAM, jest błyskawiczny [1]. Procesory najczęściej posiadają kilka rodzajów rejestrów, różniące się one zastosowaniem. W projekcie zostały użyte, te ogólnego przeznaczenia, służące głównie do przetrzymywania wyników operacji arytmetyczno-logicznych na liczbach naturalnych i rejestry specjalne w skład których wchodzą rejestry sterujące (sterujące zachowaniem procesora) i rejestry stanu (przechowujące informacje np. o wystąpieniu pewnego rodzaju zdarzenia) [1].

Innym z ważnych elementów budowy procesora jest stos, którego zadaniem jest zapis tymczasowych danych, takich jak argumenty funkcji, zmienne lokalne, adresy powrotów z funkcji itp. Zasada jego działania opiera się na znanej z algorytmiki strukturze danych o tej samej nazwie. Jednostka obliczeniowa za pomocą rejestru nazywanego wskaźnikiem stosu (ang. *stack pointer*) [1] steruje jego rozmiarem i przy wykonaniu odpowiedniej instrukcji dodaje lub usuwa wskazywaną wartość inkrementując lub dekrementując przy tym jego wskaźnik. Zasada ta jest określona jako *LIFO*(ang. *First In Last Out*) [3]. Ostatni dodany element do stosu jest, tym, który jako pierwszy zostanie z niego usunięty.

3.6 Procesor

Procesor jest jednostką w maszynie odpowiedzialną za wykonywanie podstawowych operacji. Natomiast rejestry można przyrównać do zmiennych w językach programo-

[6]Posiada ona uproszczoną słowną reprezentacje czynności procesora

wania. CHIP-8 posiada ich kilka (V0 - VF), można je potraktować jak zmienne w języku C++:

```
unsigned char V0, V1, V2, V3, V4 ... VF;
```

Procesory posiadają także program counter (PC), który można potraktować jako znany z języków, wskaźnik:

```
unsigned char *pc;
```

To jaki ciąg liczb odpowiada danej instrukcji zapisane jest między innymi w dokumentacji technicznej do CHIP-8 [4]. Dla przykładu, gdy program counter wskazuje na 0x8FC0, to wykonywana jest instrukcja LOAD(VF, VC). Wartość rejestru VF przeniesiona zostaje do rejestru VC, odpowiada to asemblerowej operacji MOV, lub w *języku C*: VF = VC. Gdyby *pointer couter* wskazywał na wartość 0x8FC4, to została by wykonana instrukcja ADD, która w *języku asemblera* , jej odpowiednik w *C* to VF = VF + VC.

Kolejną ważną kwestią jest czas wykonywania instrukcji. Realizacja każdego kodu procesora w CHIP-8 mierzona jest za pomocą tak zwanych cykli. We współczesnych jednostkach obliczeniowych zależy ona od ilości wykonywanych instrukcji, natomiast w starszych takich jak *COSMAC-1802* czas ten był z góry określony przez producenta urządzenia *Cowgod*.

4 Specyfikacja CHIP-8

CHIP-8 jest to interpretowany język programowania oryginalnie zaprojektowany dla komputerów DIY późnych lat 70 i wczesnych 80. Jego główną ideą było uczynienie procesu tworzenia gier łatwiejszym, a także danie możliwości przenoszenia kodu na różne maszyny. W tym celu opracowane zostało specjalne środowisko uruchomieniowe nazywane maszyną wirtualną. Jego schemat jest ogólnodostępny i dobrze udokumentowany. Język ten nie odniósł większego sukcesu jako stricte komputerowy. Stał on się przede wszystkim głównym oprogramowaniem dla mniejszych urządzeń w latach 80. i 90. szczególnie dla kalkulatorów. Najbardziej znanym modelem jest *TI-83*, wydany przez *Texas Instruments*, do zaawansowanych obliczeń naukowych [4].

4.1 Pamięć

Język CHIP-8 jest kompatybilny z pamięcią *RAM* do 4KB (4086B), której lokalizacja zaczyna się od adresu 0x000 (0) i kończy na 0xFFF (4096). Pierwsze 512 (0x000 - 0x1FF) bajtów odnoszą się do interpretera i nie są używane przez żadne inne programy, które w większości wczytywane są do pamięci od następnego bajtu (0x200), choć zdarzają się również takie, które wczytywane są dopiero przy 1536 (0x600) bajcie.

4.2 Rejestry

CHIP-8 posiada 8-bitowe rejestry ogólnego przeznaczenia. Ich nazwy zaczynają się od *V*, a następnie jest im przypisana kolejna cyfra heksadecymalna, co daje szesnaście rejestrów (V0-VF), z tym, że rejestr VF jest używany jako flaga dla niektórych instrukcji, dlatego nie powinien być wykorzystywany. Przygotowano także 16-bitowy rejestr nazywany *I* służący głównie do przechowywania adresów pamięci. Warto wspomnieć o dwóch 8-bitowych rejestrach specjalnego przeznaczenia są one wykorzystywane do trzymania stanu dla opóźniacza (ang. delay) i czasomierza. Są one automatycznie obniżane co 60Hz. Istnieją jeszcze tak zwane *pseduo-rejestry* do których nie ma dostępu z wykonywanych programów. Jeden z nich *program counter* (16-bitowy), trzyma adres obecnie wykonywanej komórki w pamięci. Kolejny, *stack pointer* (8-bitowy) jest używany do wskazywania najwyższego poziomu kopca. Natomiast kopiec to tablica szesnastu elementów po 16-bitów, w której trzymane są adresy. Powinny one zostać zwrócone po zakończeniu danego podprogramu (eng. *subroutine*). CHIP-8 pozwala na działanie do szesnastu zagnieżdżonych podprogramów.

4.3 Klawiatura

Układ klawiszy obsługiwanych przez maszynę wirtualną CHIP-8, odpowiada temu wykorzystanemu w komputerze *COSMAC VIP*. Posiada on klawiaturę heksadecymalną, której schemat znajduje się poniżej:

Tabela 1: Układ klawiszy CHIP-8

1	2	3	C
4	5	6	D
7	8	9	E
A	0	B	F

4.4 Ekran

W oryginalnej implementacji języka CHIP-8 został użyty monochromatyczny ekran o rozmiarze 64x32-piksele. Każdy piksel identyfikowany jest za pomocą współrzędnych x, y, których numeracja zaczyna się w lewym górnym rogu ekranu.

Tabela 2: Format ekranu

(0, 0)	(63, 0)
(0, 31)	(63, 31)

Obraz rysowany jest poprzez tak zwane duszki (ang. *sprite*), czyli grupie bajtów odpowiadających binarnej reprezentacji obrazu. Mogą one mieć wielkość do 15 bajtów, co przekłada się na maksymalny rozmiar 8x15 pikseli.

Dodatkowo programy wczytane do pamięci urządzenia mogą odnosić się do duszków reprezentujących notacje szesnastkową, czyli znaków od *0* do *F*. Ich dane w postaci heksadecymalnej powinny znajdować się w obszarze pamięci interpretera (adresy od 0x000 do 0x1FF), po włączeniu urządzenia.

4.5 Dźwięk i zegary

W swojej pierwotnej implementacji komputer posiada dwa zegary, jeden służy do opóźnień (ang. *delay timer*), a drugi do sterowania dźwiękiem. Oba zegary aktywne są tylko wtedy, gdy ich rejestry mają wartość większą od zera. Zadaniem pierwszego z nich jest jedynie odejmowanie jedynki z wartości trzymanej w rejestrze DT przy częstotliwości 60Hz. Drugi natomiast robi dokładnie to samo, również odejmując jeden od wartości w rejestrze ST przy dokładnie tej samej częstotliwości. Różnica polega na tym, że w chwili dekrementacji wartości trzymanej w rejestrze, zegar ten uruchamia zamontowany w komputerze brzęczyk. Ma on tylko jeden ton, a jego zakres nie jest nigdzie podany i zależy wyłącznie od autora interpretera.

4.6 Instrukcje procesora

Język CHIP-8 posiada 36 różnych instrukcji odnoszących się do działań matematycznych, obsługi grafiki i sterowania wykonywaniem programu. Każda z nich składa się z dwóch bajtów, z czego ten najbardziej znaczący musi być przechowywany w parzyście zaadresowanej komórce pamięci *RAM*. Bajty ładowanych duszów muszą być dopełnione tak, aby nie złamać tej zasady. Lista wszystkich dostępnych kodów procesora wraz z ich opisami jest dostępna w instrukcji komputera *COSMAC VIP* [14] lub w *Cowgod's Chip-8 Technical Reference* [4].

5 Założenia projektowe

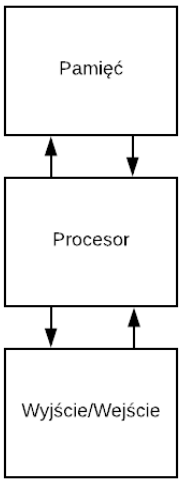
W tym rozdziale opisane zostały problemy na jakie można natknąć się w trakcie jego implementacji, wraz z wyszczególnieniem dostępnych rozwiązań. Następnie scharakteryzowana została wybrana architektura aplikacji i szczegółowo opisano jej schemat wraz z pakietami wykorzystanymi do jego implementacji. Omówiono również środowisko programistyczne w jakim będzie działać emulator.

5.1 Opis problemu i dostępnych rozwiązań

Podstawowym problemem przy projektowaniu emulatora jest implementacja modułu procesora. To on odpowiada za obliczenia, przetwarzanie pamięci *RAM* oraz zarządzanie rejestrami. Jego działanie jest ściśle określone poprzez zestaw instrukcji, które nieodpowiednio przetłumaczone na język programowania, zaburzają działanie całego programu, generując przy tym trudne do odnalezienia błędy. W tym przypadku, sprawdza się dobrze zaprojektowana struktura kodu, oparta na zasadach *KISS* (*ang. Keep It Simple*) i *DRY* (*Don't Repeat Yourself*), dbających o zachowanie klarowności kodu projektu. Pomagają w tym równie, testy jednostkowe, które poza tym mogą sprawdzać, czy dany fragment programu zachowuje się tak, jak zaplanował to programista. Nie chronią one jednak przed błędami wynikającymi ze nieścisłości w napisanej specyfikacji technicznej, czy też jej niewłaściwym zrozumieniem. Dlatego, aby dobrze zapoznać się z instrukcjami CHIP-8 warto dopisać moduł zajmujący się deasembleracją kodu bitowego wczytanego z programu działającego na tę platformę, będzie on również pomocny przy ewentualnym odnajdywania błędów emulatora. Inną dobrą praktyką jest implementacja własnego debuggera, który na bierząco wskazywałby na wykonywaną instrukcję, a także wyświetlałby bieżący stan rejestrów. Emulator musi posiadać moduł odpowiedzialny za wyświetlanie grafiki, dźwięki i obsługę urządzeń wejścia. Istnieje wiele dostępnych pakietów napisanych w języku Python, takich jak Tkinter, PyQT oraz wxWidgets nadających się do implementacji tych funkcji. Dobrym rozwiązaniem jest także wykorzystanie bibliotek odpowiedzialnych za grafikę, takich jak pyglet, bezpośrednio odnoszących się do OpenGL lub skierowanych do tworzenia gier: pygame, arcade. Istotnym problemem jest również interakcja z użytkownikiem i dostarczenie mu interfejsu do wczytywania *obrazów ROM*.

5.2 Architektura systemu

Architektura aplikacji została przemyślana tak, aby każdy z modułów odpowiadał za inną funkcje, oddzielając poszczególne warstwy abstrakcji. Komponent odpowiedzialny za pamięć, przechowuje wczytane dane, które następnie są przetwarzane przez moduł symulujący jednostkę obliczeniową i wyświetlane użytkownikowi w warstwie *wyjścia-/wejścia*. Zewnętrzne interakcje są również przekazywane przez warstwę prezentacji do opracowania przez procesor, a przy chęci zmiany wczytanego przez emulator programu, żądanie to trafia do procesora, który następnie usuwa dane trzymane w module pamięci. Koncepcji tej odpowiada *architektura warstwowa* [16].



Rysunek 1: Uproszczony diagram architektury systemu

5.3 Struktura klas

Struktura klas została oparta na *agregacji*, czyli relacji, która charakteryzuje się, tym, że jeden obiekt jest właścicielem kolejnego, może być on również współdzielony. Klasy *Processor* i *Disassembler* posiadają atrybut o nazwie *memory*, który przechowuje instancje klasy *Memory*. Podobnie jest z strukturą obiektu *Screen*, gdzie pole *proc*, trzyma instancje *Processor*, która dodatkowo jest współdzielona z klasą *Window*. Z kolei relacja między tymi klasami to *kompozycja*. Charakteryzuje się ona tym, że obiekt zagregowany ma taki sam czas życia jak jego właściciel, dodatkowo jego istnienie poza nim nie ma sensu [20].

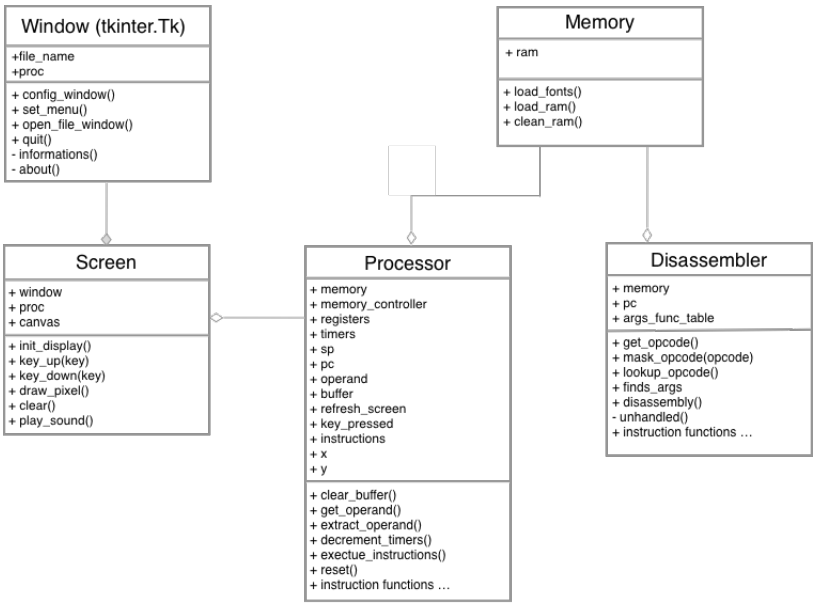
Dla uproszczenia na diagramie poniżej nie ma wypisanych metod związanych z obsługą instrukcji procesora, ze względu na ich ilość. Są one oznaczone jedynie jako *instruction functions*. Na schemacie nie ma również graficznego odniesienia do biblioteki *tkinter*. Relacja. dziedziczenia pomiędzy nią, a obiektem *Window* została zasygnalizowana w nawiasach obok nazwy klasy.

5.4 Wykorzystane biblioteki

W celu napisania projektu wykorzystano następuje biblioteki:

- *Tkinter*, jest to zorientowany obiektowo pakiet napisany jako warstwa dla języka Tcl/Tk [12]. Dostarcza on niezależne od systemu operacyjnego narzędzia do tworzenia aplikacji okienkowych, włącznie z obsługą multimediów i urządzeń wejścia, takich jak klawiatura, czy mysz komputerowa. Moduł ten od wersji języka Python 3.0 zawiera się *Standardowej Bibliotece Pythona* [7] i nie wymaga dodatkowej instalacji.

[7] Zestaw pakietów i modułów dostarczanych wraz z instalatorem języka Python, ich wykaz dostępny jest pod adresem: https://docs.python.org/3/library/



Rysunek 2: Uproszczony diagram schematu klas

- *Pytest* - to zewnętrzne (nie należy do Standardowej Biblioteki języka Python) narzędzie do testowania aplikacji. Jego dokładny opis i zastosowanie zostało zaprezentowane w rozdziale poświęconym testowaniu projektu.
- *Pipenv/virtualenv* - zarówno *pipenv*, jaki i *virtualenv* to pakiety służące do tworzenia odizolowanych środowisk programistycznych, dzięki którym możliwe jest instalowanie dowolnych pakietów języka Python bez integracji w pliki systemowe komputera. Pozwala to na korzystanie z różnych wersji paczek zależnie od wymogów projektu w jednym środowisku operacyjnym. *Pipenv* ponadto dostarcza narzędzi, które automatycznie pobierają wymagane paczki, tworzy dla nich wirtualne środowisko i dba o spójność wersji, zapisując plik w którym trzymane są wszystkie zależności, nawet te pośrednie [15].
- *Argparse* - moduł odpowiedzialny za interfejs linii poleceń. Zapewnia obsługę argumentów i opcji podanych przy wywołaniu programu. Dodatkowo automatycznie generuje sekcje *help* i wiadomości o błędach [11].

5.5 Analiza wymagań

Projekt informatyczny powinien mieć jasno zdefiniowane wymagania ustalające jego podstawowe funkcje. W rezultacie ma to zapobiec implementacji niepotrzebnych fragmentów kodu i wyszczególnić zestaw cech, które powinien posiadać produkt, aby uznać go za gotowy. Taka analiza zapewnia skupienie się na jasno określonych celach.

5.5.1 Wymagania funkcyjne

Zadaniem emulatora jest wczytywanie dostarczonych przez użytkownika plików, *tylko do odczytu*[8], które będą wczytane do pamięci maszyny wirtualnej. Następnie dane te będą przetwarzane na zaimplementowane instrukcje procesora. W rezultacie, użytkownikowi wyświetli się okno programu, które będzie generować emulowaną grę wraz z możliwością interakcji z nią, dzięki klawiaturze. Ponadto aplikacje będzie odtwarzał dźwięk, a także da użytkownikowi możliwość bezbłędnego przerwania programu lub zresetowania aktualnego stanu maszyny wirtualnej wraz z możliwością zmiany odtwarzanego pliku binarnego. Z dodatkowych funkcji, możliwa będzie także zmiana rozdzielczości, a co za tym idzie rozmiaru okienka wyświetlanego okna i zmiana czasu cyklu procesora, tak aby przyśpieszyć odtwarzany program. Opcje te będą dostępne w menu umieszczonym pod górnym paskiem okienka, a poszczególne z nich będzie również będzie można zmienić uruchamiając program z linii poleceń, dodając do komendy odpowiednie argumenty.

5.5.2 Wymagania niefunkcyjne

Emulator będzie ograniczał się do uruchamiania plików jedynie przeznaczonych dla maszyny wirtualnej CHIP-8. Jego wydajność będzie stosunkowo niewielka, ze względu na wybrany język programowania, a także ograniczenia biblioteki *tkinter*. Dodatkowo sama emulacja wiąże się z wykorzystywaniem dużych zasobów pamięciowych (wczytywane pliki są w całości przetrzymywane w pamięci komputera), a także wydajnościowych, gdyż pętla takiego programu, to wykonanie szeregu instrukcji i interakcji, jednak czas ten będzie nadal o wiele szybszy niż podane w specyfikacji taktowanie procesora [4].

5.5.3 Scenariusze użycia

Scenariusze użycia (ang. *use cases*), to przedstawienie z opisami poszczególnych etapów działania aplikacji w krokach. Mają one na celu uproszczenie wymagań funkcyjnych poprzez ich uporządkowanie. Są również dobrym sposobem do zobrazowania zagrożeń jakie czeka dany system i zaplanowaniem ich kontrolowania. W tym przypadku, zostaną przedstawione wyłącznie scenariusze, gdzie będą istniały ich alternatywne ścieżki. Takie jak zmiana prędkości emulatora, czy rozdzielczości nie mają w założeniu generować alternatywach scenariuszy, dlatego w tym podrozdziale zostaną pominięte.

UC-01: Uruchomienie emulatora z linii komend bez podania argumentów
Aktorzy: Użytkownik
Cel: Uruchomienie aplikacji okienkowej z działającą emulacją wybranego programu, który reaguje na wejście z klawiatury. Dostęp do menu górnego. Po prawidłowym wyłączeniu aplikacja nie zgłasza błędu.
Główny scenariusz:

1. Aplikacja prezentuje ciemne tło. Z okienkiem informującym o braku wczytanego pliku binarnego.

[8] z ang. read-only memory, ROM, pamięć, której można jedynie odczytywać, w przypadku emulatorów są pliki binarne (mogą mieć również rozwinięcie .c8, .c8h, ch8) lub obrazy gier.

2. Użytkownik akceptuje informacje, klikając *OK*, okienko zostaje zamknięte.
3. Z paska górnego zostaje wybrany *File > Open*.
4. Użytkownik wybiera odpowiedni plik binarny (*, .c8, .ch8).
5. Program wczytuje plik i wyświetla grafikę.
6. Użytkownik zakańcza działanie programu poprzez kliknięcie krzyżyka lub opcje w menu.

Alternatywny scenariusz:

3a Użytkownik wybiera niepoprawny plik.

1. Aplikacja wyświetla błąd ″Cannot read file.".
2. Użytkownik wraca do punktu 2.

UC-02: Uruchomienie emulatora z podaniem nazwy pliku jako argument
Aktorzy: Użytkownik
Cel: Uruchomienie aplikacji okienkowej z działającą emulacją wybranego programu, który reaguje na wejście z klawiatury. Dostęp do menu górnego. Po prawidłowym wyłączeniu aplikacja nie zgłasza błędu.
Główny scenariusz:

1. Użytkownik podaje argument *-f NAZWAPLIKU*.
2. Program wczytuje plik i wyświetla grafikę.
3. Użytkownik zakańcza działanie programu poprzez kliknięcie krzyżyka lub opcje w menu.

Alternatywny scenariusz:

1a Użytkownik wybiera niepoprawny plik.

1. Aplikacja wyświetla błąd ″Cannot read file.".
2. Użytkownik wraca do punktu 1.

UC-03: Po uruchomieniu programu bez wybrania emulatora użytkownik resetuje emulacje
Aktorzy: Użytkownik
Cel: Obsługa błędu związanego z zresetowaniem programu bez wczytania pliku binarnego. Użytkownik następnie wybiera plik binarny i uruchamia emulacje
Główny scenariusz:

1. Aplikacja prezentuje ciemne tło. Z okienkiem informującym o braku wczytanego pliku binarnego.
2. Użytkownik akceptuje informacje, klikając *OK*, okienko zostaje zamknięte.
3. Użytkownik wybiera z górnego menu *File > Reset*

4. Pojawia się okno informujące o braku pliku binarnego do wczytania.

5. Użytkownik postępuje zgodnie z UC-01 lub UC-02

Alternatywny scenariusz:

5a Użytkownik wybiera z górnego menu *File* > *Reset*

1. Aplikacja wyświetla błąd ″Cannot read file.".

2. Użytkownik wraca do punktu 4.

UC-04: Uruchomienie programu z wpisaniem argumentu *-d –disassembler*
Aktorzy: Użytkownik
Cel: Wypisanie na ekran konsoli kodów procesora w postaci mnemonicznej.
Główny scenariusz:

1. Użytkownik podaje argument *-d* lub *–disassembler* przy komendzie do uruchomienia programu wraz z *-f* lub *–file*

2. W konsoli zostają wypisane kody procesora w postaci mnemonicznej.

Alternatywny scenariusz:

1a Użytkownik nie podaje argumentu *-f* lub *–file*

1. Aplikacja wyświetla w konsoli text ″No input file".

2. Użytkownik wraca do punktu 1.

5.6 Środowisko programistyczne

Do uruchomienia emulatora wymagana jest instalacja języka programowania Python w wersji 3.7.0 lub nowszej. Można go pobrać z strony
https://www.python.org. Następnie w za pomocą terminala systemu UNIX lub aplikacji *cmd* w systemie Windows. należy przejść do katalogu projektu, a następnie wpisać komendę *python pychip* i potwierdzić wykonanie polecenia za pomocą klawisza enter. Niektóre aplikacje napisane w języku Python często wymagają modułów lub pakietów, które nie są dostarczone w *Standardowej Bibliotece Pythona*. W przypadku tego projektu, wszystkie wymagane pakiety do jego uruchomienia należą do biblioteki standardowej, jednak w celu włączenia testów należy zainstalować moduł *pytest*. Najprościej można to zrobić, w zależności od systemu operacyjnego, poprzez jeden ze wspomnianych programów emulujących terminal. Należy użyć komendy: *pip install pytest*, a następnie uruchomić testy za pomocą komendy *pytest*. W przypadku, gdyby nastąpił konflikt z wersjami pakietów, można wykorzystać wirtualne środowisko lub skorzystać z narzędzia *pipenv*. Dokładana instrukcja znajduje się w pliku *README.md* w katalogu projektu [9].

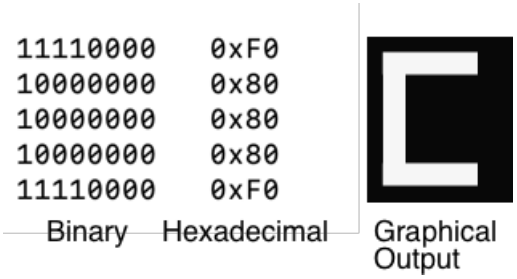
[9] Adres do repozytorium: https://github.com/krskibin/pyChip8

6 Implementacja

W niniejszym rozdziale opisano szczegółowo implementację komponentów składających się na budowę emulatora: pamięć, procesor, wybranych instrukcji maszyny wirtualnej, deasembler, urządzenia wejścia i wyjścia, a także moduł obsługujący argumenty wiersza poleceń i główną pętlę programu.

6.1 Moduł pamięci

Pamięć maszyny wirtualnej została zaimplementowana jako oddzielna klasa, której zadaniem jest stworzenie *tablicy bitowej* (ang.*bytearray*) mającej rozmiar odpowiadający wielkości pamięci *RAM* ze specyfikacji CHIP-8 (4096B) [4]. Następnie od zerowego indeksu wczytywany jest do niej zestaw czcionek. Są one zapisane w tablicy jako cyfry szesnastkowe.



Rysunek 3: Konwersja litery "C" z liczby binarnej do szesnastkowej i postaci graficznej

Kolejną metodą klasy *Memory* jest ta, odpowiedzialna za wczytanie pliku binarnego z programem. Działa ona podobnie do poprzednio przedstawionej funkcji, odpowiedzialnej za wczytywanie czcionek, z tą różnicą, że wykorzystana została wbudowana w język Python funkcja *open()* z argumentem *'rb'* (*read binary*), odpowiedzialnym za wczytywanie pliku w postaci binarnej. Tym razem dane są wczytywane do pamięci *RAM* od adresu z indeksem 512, czyli przestrzeni specjalnie przeznaczonej na zewnętrzny program [4].

6.2 Deasembler

Deasembler to dodatkowy moduł zaimplementowany w celu lepszego zapoznania się z instrukcjami procesora. Podobnie jak kod odpowiedzialny za emulowanie procesora. Wczytuje on instancje klasy odpowiedzialnej za zarządzanie pamięcią, a następnie za pomocą metody *get_ opcode*, przekazuje wartość do zmiennej *opcode*. Następnie wydobywane są argumenty instrukcji, czyli bajty, który nie służą do rozpoznania wykonywanej operacji. Wskazują one jedynie odpowiednie numery rejestrów lub bajty które zostaną do nich dodane, a następnie przekazuje je do atrybutu *args*. Kolejnym krokiem jest zamaskowanie argumentów w zmiennej trzymającej kod instrukcji, tak, aby posiadała ona informacje odpowiadające jedynie za identyfikacje kodu procesora. Otrzymane dane służą jako klucz w słowniku, jego wartością jest natomiast funkcja zwracająca podstać mnemoniczną danego *słowa*. Na końcu wszystkie otrzymane informacje są odpowiednio formatowane jako zmienna typu *string*. Cała procedura powta-

rzana jest w pętli od pierwszego adresu w pamięci do ostatniego, a licznik programu za każdą iteracją jest inkrementowany o 2.

```
MacBook-Pro-Krystian:pychip krystian$ python3 pychip -f roms/PONG -d
Loading roms
0x200:  6a02    LD      Va      0x2
0x202:  6b0c    LD      Vb      0xc
0x204:  6c3f    LD      Vc      0x3f
0x206:  6d0c    LD      Vd      0xc
0x208:  a2ea    LD      I,      2ea
0x20a:  dab6    DRW     V6, Vb6  6
0x20c:  dcd6    DRW     V6, Vd6  6
0x20e:  6e00    LD      Ve      0x0
0x210:  22d4    CALL            2d4
0x212:  6603    LD      V6      0x3
0x214:  6802    LD      V8      0x2
0x216:  6060    LD      V0      0x60
0x218:  f015    LD      DT, V0
0x21a:  f007    LD      V0, DT
0x21c:  3000    SE      V0      0x0
0x21e:  121a    JP              21a
0x220:  c717    RND     V7      0x17
0x222:  7708    ADD     V7      0x8
0x224:  69ff    LD      V9      0xff
0x226:  a2f0    LD      I,      2f0
0x228:  d671    DRW     V1, V71  1
0x22a:  a2ea    LD      I,      2ea
```

Rysunek 4: Efekt działania modułu deasemblera

6.3 Moduł procesora

Klasa *Processor* przyjmuje instancje pamięci jako argument, następnie przy inicjalizacji tworzony jest słownik trzymający stan rejestrów *V*, *S* i *I*, a także *delay* i *sound*. Następnie tworzone są zmienne, wskazujące na komórkę w pamięci *RAM*, czyli *program counter* i wskaźnik na stos *stack pointer*. Kolejnymi polami są: lista odzwierciedlająca bufor ekranu i zmienna typu *true/false* trzymająca informacje, czy zawartość ekranu była edytowana. Jest ona używana do sprawdzenia, czy moduł odpowiedzialny za wyświetlanie obrazu potrzebuje wygenerować na nowo grafikę. Ostatnimi atrybutami są tworzone przy inicjalizacji: *pressed_key* posiadający ostatniego kliknięty klawisza i słownik *instructions*.

Działanie klasy procesora oparte jest o wczytywanie obecnie wskazanej przez *program counter* i następnej komórki do specjalnej zmiennej nazwanej *operand*. Spowodowane jest to tym, że kody procesora mają postać szesnastobitową, a pamięć jest ośmiobitowa. Z tego względu wartość *operand* musi zostać przesunięta o 8 bitów w prawo, przez co do tej zmiennej dołączane jest osiem zer, następnie wykorzystuje operacje alternatywy, dzięki temu wartość z obu komórek włączana jest do zmiennej *operand*.

Wszystkie instrukcje procesora są zaimplementowane jako funkcje. W Pythonie sama funkcja bez jej wywołania jest traktowana jako obiekt, dlatego można go wykorzystać jako wartość w słowniku, którego kluczami będą liczby od 0x0 do 0xF, odpowiadają one zakresowi starszych półbajtów pierwszych bitów instrukcji przedstawionych w *Cowgod's CHIP-8 Technical Reference* [4].

```
1 self.instructions = {
2     0x1: self.jp_addr,
```

```
    0x2: self.call_addr,
    0x3: self.se_vx_byte,
    0x4: self.sn_vx_byte,
    0x5: self.se_vx_vy,
    0x6: self.ld_vx_byte,
    0x7: self.add_vx_byte,
    0x9: self.sne_vx_vy,
    0xa: self.ld_i_addr,
    0xc: self.rnd_vx_byte,
    0xd: self.drw_vx_vy_n,
    0x0: self.sys_ops,
    0x8: self.logical_ops,
    0xf: self.rest_ops,
    0xE: self.press_ops
}
```

Listing 1: Budowa słownika *instructions*

Po dostarczeniu wartości pamięci do zmiennej *operand* wybierany jest z niej starszy półbajt pierwszego bajta, który staje się kluczem do wcześniej wspomnianego słownika *instructions*. Na jego podstawie wybierana zostaje instrukcja, która będzie wykonywana. W razie gdyby taki klucz nie istniał, wykonywana jest pusta funkcja lambda, która wyświetli informacje o błędzie. Zanim to jednak się stanie, wartość licznika programu podnoszona[10] jest o 2.

```
def get_operand(self):
    return self.memory[self.pc] << 8 | self.memory[self.pc + 1]

def extract_opcode(self):
    return (self.operand & 0xf000) >> 12

def execute_instruction(self):
    self.operand = self.get_operand()
    opcode = self.extract_opcode()
    self.pc += 2
    self.instructions.get(opcode, lambda x: ...)()
```

Listing 2: Framgent kodu, służący do wywoływania instrukcji procesora.

Po wykonaniu kodu operacyjnego zostaje zmieniana jeszcze wartość zegarów emulowanego urządzenia.

6.3.1 Instrukcje

Istnieją przypadki, gdzie pod wspomnianym wcześniej identyfikatorem można wyróżnić kilka kodów procesora, wówczas są one weryfikowane na podstawie pozostałych bitów wewnątrz jednej funkcji za pomocą instrukcji warunkowej *if*. Takimi przypadkami są klucze w słowniku o wartościach: *0x0*, *0x8*, *0xE*, *0xF*.

Implementacja poszczególnych z ich wygląda następująco:

- Wywołaj adres z stosu (*call_ addr*):

[10] Wartość licznika programu inkrementowana jest za każdym razem o 2, tak aby pominąć wykorzystane już komórki, które pełnią role młodszych bajtów w zmiennej *operand*

Do rejestru *S*, dodawana jest wartość *licznika programu*, a *wskaźnik stosu* jest inkrementowany o jeden. Następnie za pomocą operacji binarnej *AND*, z wartości *operand* wybieranych jest 12 ostatnich bitów. Zostają one przypisane do *licznika programu*.

```
self.registers['s'].append(self.pc)
self.sp += 1
self.pc = self.operand & 0x0fff
```

Listing 3: Framgent kodu, odpowiedzialny za instrukcje *call_addr*

- Pomiń następną instrukcje jeżeli *Vx*==*Vy* (*sne_vx_vy*):

Dzięki operacji binarnej *AND* z pierwszego bajta *słowa* wyciągane są młodsze bity. W celu usunięcia drugiego bajta wykonywane jest przesunięcie bitowe o 8. Uzyskana zmienna posłuży do identyfikacji rejestru *Vx*. Następnie w podobny sposób ze *słowa* wyciągana jest wartość do identyfikacji rejestru *Vy*. Jeżeli wartości z rejestrów *Vx* i Vy są sobie równe, to *licznik programu* inkrementowany jest o 2.

```
x = (self.operand & 0x0f00) >> 8
y = (self.operand & 0x00f0) >> 4
if self.registers['v'][x] == self.registers['v'][y]:
    self.pc += 2
```

Listing 4: Implementacja instrukcji *sne_vx_vy*

- Dodaj *Vx* do *Vy* (*add_vx_vy*):

Wartość z rejestru *Vy* jest dodawana do rejestru *Vx*. Jeżeli po dodaniu *Vx* przekracza 256 bitów, ustawiana jest flaga w rejestrze *Vf*. Sprawdzona zmienna za pomocą instrucji *AND* zostaje zredukowana do wartości mniejszej niż 256. W przeciwnym przypadku, flaga *Vf* zostaje wyzerowana.

```
self.registers['v'][x] += self.registers['v'][y]
if self.registers['v'][x] > 0xFF:
    self.registers['v'][0xF] = 1
    self.registers['v'][x] &= 0xFF
else:
  self.registers['v'][0xF] = 0
```

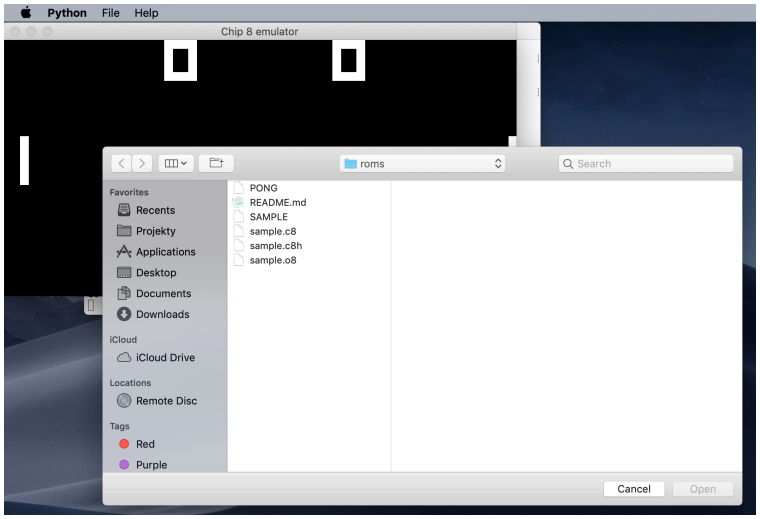
Listing 5: Implementacja instrukcji *add_vx_vy*

6.4 Moduł wejścia/wyjścia

Podstawą tego modułu jest klasa *Window*, dziedzicząca z*tkinter.Tk*. Odpowiada ona za podstawę interfejsu użytkownika wyświetlającego emulacje, jego konfiguracje, a także pasek menu przy górnej belce. Przy jej inicjalizacji zostaje ustalony rozmiar okna, a także zablokowana możliwość ręcznego zmieniania jego wielkości. Następnie zostaje ustalona kolejność komend na pasku narzędzi. Pierwszą opcją jest *file*, w której zawiera funkcja *open*. Odpowiedzialną za wywołanie okna wyboru pliku. Jest to standardowy *dialog* z pakiety *tkinter.filedialog* [12].

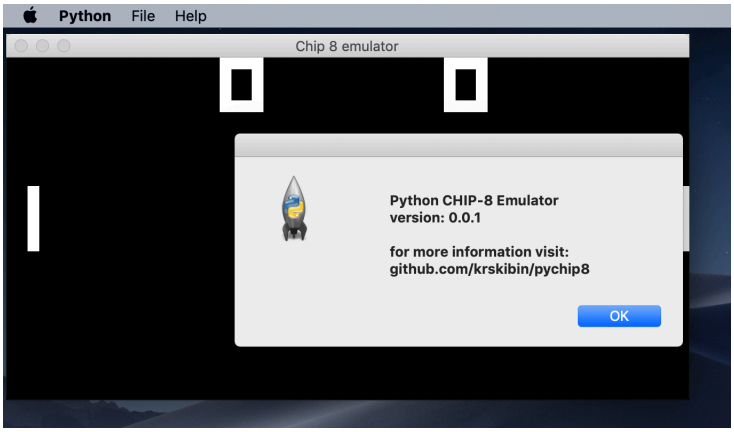
Rysunek 5: Dialog do wybierania plików (system *GNU/Linux*)

W opisywanym projekcie potrzebne były różne konfiguracje dla każdej ze wspieranych platform. Dla przykładu system windows do przedstawienia ścieżek używa znaku *lewego ukośnika*, gdy w platformach *uniksowych* jest to zwykły ukośnik. Dodatkowo *dialog* w systemie*macOS* wyświetlający pliki nie posiada możliwości wyboru formatu, więc w tym przypadku, trzeba było pozostawić opcję wyboru tylko wszystkich plików.



Rysunek 6: Dialog do wybierania plików (system *macOS*)

Kolejną opcją jest *reset*, gdzie przywracane są wszystkie rejestry do wartości początkowych, a emulacja rozpoczyna się na nowo. Następną sekcją jest *help*, która zawiera opcje *about*, w których są zawarte informacje o wersji programu z linkiem do *githuba* projektu i *changelog*, gdzie opisano zmiany w wersja. We wszystkich tych przypadkach skorzystano z modułu *tkinter.messagebox* [12], który wyświetla okienka informacyjne. System *macOS* posiada pasek menu, który zmienia się zależnie od aplikacji przez co, nie ma pojawia się pasek narzędzi w okienku aplikacji. Ma on dodatkową sekcje, która ma taką samą nazwę jak ta aplikacji. Z tego względu dla tej platformy dodano funkcję, która zmienia wyświetlane w nim informacje na te z sekcji *about*.



Rysunek 7: *Messagebox* z informacjami (system macOS)

Następnym zaimplementowanym fragmentem kodu jest klasa *Screen*. Odpowiada ona za sterownie wcześniej opisaną klasę *Window*. To przy jej inicjalizacji podpinane są zdarzenia dla kliknięcia klawisza. Są to funkcje:*key_ down* i *key_ up*. Ich zadaniem jest mapowanie klawiszy z klawiatury użytkownika na heksadecymalną, którą oryginalnie obsługiwały komputery dla których był napisany CHIP-8 [14].

```
self.keymap = {
    '1': 1, '2': 2, '3': 3, '4': 12, 'q': 4, 'w': 5,
    'e': 6, 'r': 13, 'a': 7, 's': 8, 'd': 9, 'f': 14,
    'z': 10, 'x': 0, 'c': 11, 'v': 15,
}
```

Listing 6: Słownik z mapowaniem klawiszy (wartości w nawiasach, to rzeczywiste klawisze)

Do tej klasy podpięto również obiekt *tkinter.Canvas* [12], który dostarcza interfejs pozwalający na rysowanie kształtów. Wykorzystuje ją metoda *draw_pixel*, której zadaniem jest iteracja po *buferze* z modułu procesora i gdy napotka wartość *1*, to w tym miejscu, na odpowiednie przeskalowanym kanwasie rysuje kwadrat. Dla oszczędzenia pamięci, przed wywałniem tej metody, wszystkie obiekty z kanwasu są usuwane. Ostatnią metodą w tym module jest ta odpowiedzialna za czyszczenie ekranu emulatora. Jest ona interfejsem do metody zaimplementowanej w *tkinter* i przyjmuje ona specjalną stałą *tkinter.ALL* zawierającą referencje do każdego obiektu kanwasu.

6.5 Główny moduł programu

Jest to moduł odpowiedzialny za ciągłe działanie emulatora. Tworzone są w nim instancje zaimplementowanych klas, a także przekazane wymagane argumenty. Następnie ustalana jest rozdzielczość ekranu, obecność dźwięku i szybkość działania procesora. Kolejnym krokiem jest rozpoczęcie *głównej pętli aplikacji*, która z każdą iteracją wykonuje kolejną instrukcje procesora, a także definiuje jego prędkość taktowania.

Rysunek 8: Działający emulator, gra *TETRIS* (autor Fran Dachille, rok 1991)

W module znajduje się również implementacja interfejsu linii poleceń. Jest to obiekt z pakietu *argparse*, który za pomocą metody *add_ argument* [11], definiuje opcje, przyjmowane przez program przy wywołaniu, a także ich opis, typ i nazwy zmiennych do których zostaną przepisane one w kodzie aplikacji.

```
$../pychip: python pychip -h
usage: [-h] [-f ROMS] [-d] [-r RES] [-t DLY]

Chip8 emulator

optional arguments:
  -h, --help            show this help message and exit
  -f ROMS, --file ROMS  use to provide roms file
  -d, --disassembler    run disassembler for given roms file
  -r RES, --res RES     set CHIP-8 screen resolution
  -t DLY, --time DLY    set emulation delay time
```

Listing 7: Wynik uruchomienia programu z opcją *-h*

7 Testy

Projekt był tworzony zgodnie z techniką *programowania sterowanego testami (ang. test driven development)* [21]. W związku z tym, przed napisaniem jakiejkolwiek fragmentu kodu najpierw opracowywany był dla niego test. To podejście pomaga utrzymać większą jakość kodu, a także pozwala na szybsze wykrycie, czy dana funkcja zwraca pożądaną wartość lub odpowiednio modyfikuje zmienne. W takim projekcie jak emulator, gdzie najmniejsze odchylenie w generowanych przez program wartościach może sprawić, że emulowany program będzie działał niezgodnie z oczekiwaniami, testy przydają się jako narzędzie wspierające debugowanie kodu.

Twórcy *Pythona* w bibliotece standardowej zapewnili pakiet *unittest* odpowiedzialny za testy, jednak w projekcie został użyty *pytest* ze względu na większą ilość narzędzi przydatnych przy pisaniu testów dla dużej ilości danych wejściowych.

Ilość interakcji w programie jest mała, a ich każdorazowe wystąpienie bezpośrednio wpływa na stan programu, dlatego uznano, że testy integracyjne i akceptacyjne [21] nie są potrzebne, a do przetestowania aplikacji wystarczą testy jednostkowe.

Strukturę testu jednostkowego definiuje zasada *Arrange-Act-Assert* [21]. Na samym początku wszystkie dane są przygotowywane (arrange) i odizolowywane. Ważne, aby zewnętrzne zależności, takie jak inne klasy, czy obsługa strumieni wejścia i wyjścia zastąpione były atrapami (ang. *mock*). Jest to główna różnica między podejściem jednostkowym, a modułowym, gdzie tego typu zależności są dopuszczalne. Następnym krokiem jest wykonanie działania na testowanym fragmencie kodu (*act*) i sprawdzenie, czy jego wywołanie zwróciło porządny rezultat (*assert*). Przeważnie biblioteki testowe dostarczają odpowiednią funkcjonalność do porównywania danych oczekiwanych z tymi, które są przetwarzane przez sprawdzany fragment programu. Inaczej jest w przypadku modułu *pytest*, który dokonuje tego wykorzystując wbudowaną w język Python funkcję *assert* [8]. Jej działanie jest podobne do standardowej instrukcji *if* z tą różnicą, że niespełnienie warunku od razu traktowane jest jako błąd [8].

```
class TestProcessor:
  def test_reset(self):
    """
    Processor.reset() powinien ustawic pointer counter
    do domyslnej wartosci (0x200)
    """
    # arrange
    proc = Processor()
    proc.pc = 0x230

    # act
    proc.reset()

    # assert
    assert self.pc == 0x200
```

Listing 8: Przykład prostego testu przy użyciu biblioteki *pytest*

Kolejnym ważnym komponentem *pytestu* jest parametryzacja testów. Zapewnia ona dostarczenie różnych danych wejściowych i oczekiwanych wyników bez potrzeby two-

rzenia testów dla innych przypadków [8].

```
class TestDisasssembler:
    mask_params = ([0xFDEE, 0xF0EE], [0xE337, 0xE037],
                   [0x0ABC, 0x00BC], [0x8DCA, 0x800A],
                   [0xAACE, 0xA000])

    @pytest.mark.parametrize(("opcode", "expected"), mask_params)
    def test_opcode_mask(self, opcode, expected):
        dasm = Disassembler()
        assert dasm.mask_opcode(opcode) == expected
```

Listing 9: Przykład testu sparametryzowanego.

W obu listingach wewnątrz każdego testu jest tworzony nowy obiekt klasy, do której należy sprawdzana funkcja. Procedurę tę można zautomatyzować, tworząc tak zwane *fixture* [8]. Są to najczęściej funkcje, których zadaniem jest automatyczne wykonanie się zanim zależne od nich testy zostaną rozpoczęte.

```
@pytest.fixture(scope="function")
def dasm(request):
    return Disassembler()

@pytest.mark.usefixtures('dasm')
class TestDisassembler:
    # ...

    @pytest.mark.parametrize(("opcode", "expected"), mask_params)
    def test_opcode_mask(self, opcode, expected, dasm):
        assert dasm.mask_opcode(opcode) == expected
```

Listing 10: Przykład poprzedniego testu z użyciem *fixture*.

Jak już wcześniej zostało wspomniane, testy jednostkowe nie mogą korzystać z zewnętrznych zależności potrzebnego do utworzenia sprawdzanej klasy lub funkcji. Zamiast nich tworzymy obiekty atrapy. Imitują one jedynie rzeczywistą instancje klasy, jej atrybutu, jak również ich wywołania. W przypadku tego projektu wykorzystano moduł z biblioteki standardowej *unittest.mock*. Dostarcza ona klasę *MagicMock*, jej zadaniem jest tworzenie imitacji obiektów, wtedy, kiedy test wymaga dostępu do ich metod lub atrybutu [13]. Następnie za pomocą *patch* można imitować pola lub funkcje wybranej klasy *MOCK* i nadawać im zwracane wartości.

```
pytest.fixture(scope="function")
def proc(request):
    memory = mock.MagicMock()
    processor = Processor(screen)
    return processor

class TestProcessor:
    extract_params = ([0x0523, 0x0], [0x1fcd, 0x1], [0x2cde, 0x2])
    @pytest.mark.parametrize(("operand", "expected"), extract_params)
    def test_extract_opcode(self, operand, expected, proc):
    mod = f"{self.module_name}.operand"
    with mock.patch(mod, new_callable=mock.PropertyMock) as mock_op:
       mock_op.return_value = operand
       assert proc.extract_opcode() == expected
```

Listing 11: Przykład przykład testowania za pomocą atrap

Technika programowania sterowanego testami zapewniła, że od samego początku wdrażania projektu, jego kod był lepszej jakości i łatwiejszy do utrzymywania. Filozofia ta skupia dużą uwagę, na tym, aby każdy fragment programu nadawał się do przetestowania, co pozwalało na szybkie zweryfikowanie, w której części emulatora zwracał niewłaściwe rezultaty. Atrapy przyczyniły się do jeszcze większej izolacji poszczególnych modułów, co gwarantowało, że niepowodzenie danego testu zależało wyłącznie od sprawdzanej funkcji. Dzięki, parametryzacji, możliwe było przebadanie wielu danych wejściowych i porównanie, czy wszystkie zwracają porządny rezultat. To z kolei, pomagało znajdować wartości, których problematyczności nie uwzględniono w pierwszych iteracjach kodu programu.

8 Podsumowanie

W ramach niniejszej pracy powstał program pozwalający na wczytywanie gier z zewnętrznych plików, a następnie ich emulację, wyświetlanie w środowisku graficznym i interakcję. Na potrzebę użytkownika dowolna jest manipulacja parametrami aplikacji, tak, aby dostosować prędkość działania procesora lub skalę grafiki. Możliwe jest wybieranie innych plików do emulacji za pomocą zawartego w interfejsie graficznym paska menu, lub ich resetowanie.

Emulator uruchamia wszystkie programy opracowanych na maszynę wirtualną CHIP-8.

Projekt w całości spełnia przyjęte wymagania i został opracowany zgodnie z przeprowadzonymi założeniami projektowymi i specyfikacją maszyny wirtualnej [4].

8.1 Możliwe udoskonalenia

Dalszy rozwój emulatora powinien opierać się na zaimplementowaniu instrukcji dla *Super CHIP-48* [4], czyli ulepszonej wersji CHIP-8. Tu potrzebne byłoby również opracowanie ulepszonego mechanizmu wyświetlania grafiki i dodanie przewijania ekranu [4]. Dobrym pomysłem wydaje się zaimplementowanie *debuggera*, który podświetlałby aktualnie wykonywaną instrukcję w oknie i dawał możliwość ich wykonywania krok po kroku lub cofania, przy okazji wyświetlając w innym miejscu aktualny stan rejestrów. W kwestii zupełnie nowych funkcji można byłoby dodać moduł odpowiedzialny za kompilowanie specjalnie utworzonej odmiany *języka assemblera* do plików, które wczytywałby emulator. Umożliwiałoby to tworzenie własnoręcznie napisanych gier pod tę platformę.

Literatura

[1] G. Coldwind. *Zrozumieć programowanie*. Wydawnictwo Naukowe PWN, 2015.

[2] Commodore Buisness Machines, Inc. and Howards W. Sams and Co., Inc. *COMMODORE 64 PROGRAMMER'S REFERENCE GUIDE*, 1982.

[3] Th. H. Cormen, Ch. E. Leiserson, and R. L. Rivest. *Wprowadzenie do Algorytmów*. Massachusetts Institute of Technology, 2002.

[4] Internet: Cowgod. http://devernay.free.fr/hacks/chip8/c8tech10.htm, 1997. Cowgod's Chip-8 Technical Reference.

[5] Internet: Edwards, B. https://www.fastcompany.com/90147592/rediscovering-historys-lost-first-female-video-game-designer, 2017. Rediscovering History's Lost First Female Video Game Designer.

[6] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2016.

[7] E. James. The architecture of virtual machines. *IEEE Computer Society*, 2005.

[8] Internet: Krekel, H. https://docs.pytest.org/en/latest/contents.html. Pytest documentation.

[9] Ch. Petzold. *Code: The Hidden Language*. Microsoft Press, 2002.

[10] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Association for Computing Machinery*, 1974.

[11] Internet: Python Software Foundation. https://docs.python.org/3/library/argparse.html, 2001-2017. Argparse — Parser for command-line options, arguments and subcommands.

[12] Internet: Python Software Foundation. https://docs.python.org/3/library/tk.html, 2019. Graphical User Interfaces with Tk.

[13] Internet: Python Software Foundation. https://docs.python.org/3/library/unittest.mock.html, 2019 last update. Python3 unittest.mock documentation.

[14] RCA Corporation. *RCA COSMAC VIP CDP18S711 Instruction Manual*, 1978.

[15] Internet: Reitz, K. https://pipenv.readthedocs.io/en/latest/, 2017. Pipenv: Python Dev Workflow for Humans.

[16] Mark Richards. *Software Architecture Patterns*. O'Reilly Media, Inc., 2015.

[17] D. S. H. Rosenthal. Emulation & virtualization as preservation strategies, 2015.

[18] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited, 2002.

[19] Patrick H. Stakem. *4- and 8-bit Microprocessors Architecture and History*. PRRB Publishing, 2013.

[20] Internet: Trybulec, K. https://www.p-programowanie.pl/uml/diagramy-klas-uml/, 2017. Diagramy klas UML.

[21] D. Woźniak. *TDD. Techniki programowania sterowanego testami.* Helion, 2018.