

POLITECHNIKA KOSZALIŃSKA



WYDZIAŁ ELEKTRONIKI I INFORMATYKI

INFORMATYKA
Technologie Internetowe i Mobilne

Patryk Banaś
U-15568

Emulator konsoli Nintendo Entertainment System
Nintendo Entertainment System Emulator

Praca inżynierska wykonana pod kierunkiem
dr inż. Robert Świta

Spis treści

1. Wprowadzenie.....	3
1.1 Emulator NES.....	3
1.2 Cel	3
2. Podstawy techniczne emulacji konsoli NES	5
2.1 CPU	5
2.1.1 2A03	5
2.1.2 Rejestry.....	5
2.1.3 Tryby adresowania	7
2.1.4 Instrukcje	8
2.1.5 Przerwania.....	9
2.1.6 Mapa pamięci i urządzenia we/wy zmapowane w pamięci	9
2.2 PPU	11
2.2.1 2C02	11
2.2.2 Rejestry.....	12
2.2.3 OAM	13
2.2.4 Mapa pamięci PPU	13
2.2.5 Tabele wzorów i Palety kolorów	15
2.2.6 Tabele nazw / Tabele atrybutów	17
2.2.7 Mirroring	18
2.3 Kartridże.....	18
2.3.1 Mapper'y pamięci	20
2.3.2 Format pliku iNES	20
2.4 Kontroler	21
3. Dostępne emulatory konsoli NES.....	22
4. Narzędzia i metodologia.....	23
4.1 Język programowania	23
4.2 Grafička i GUI.....	23
4.3 Architektura.....	23
5. Implementacja.....	25
5.1 Emulacja CPU	25
5.2 Emulacja PPU	28
5.3 Import i obsługa kartridżów w formacie iNES	32

6. Prezentacja działania aplikacji	35
6.1 PPU Viewer.....	35
6.2 Weryfikacja działania.....	36
6.2.1 Gry.....	36
6.2.2 ROM'y testowe.....	37
7. Wnioski.....	39
8. Bibliografia	41
9. Spis rysunków	42
10. Spis tabel	43

1. Wprowadzenie

1.1 Emulator NES

Emulatory są programami komputerowymi, które zostały zaprojektowane w celu naśladowania funkcjonalności innego systemu komputerowego lub urządzenia. Osiągają to poprzez interpretację instrukcji i danych oryginalnego systemu i wykonywanie ich na systemie hosta. Emulatory mogą być wykorzystywane do różnych celów, w tym do tworzenia oprogramowania, testowania systemu i gier. [1]

Jednym z najpopularniejszych rodzajów emulatorów jest emulator NES. Nintendo Entertainment System (*NES*) była popularną konsolą do gier video, która została wydana w latach 80. Była znana ze swoich innowacyjnych gier i prostego, ale skutecznego projektu sprzętu. Popularność NES doprowadziła do powstania wielu emulatorów, które mogą uruchamiać gry NES na nowoczesnych komputerach i urządzeniach mobilnych. [4]

Tworzenie emulatora NES wymaga głębokiego zrozumienia architektury sprzętowej NES, a także oprogramowania, które na nim działa. NES jest oparty na 8-bitowym mikroprocesorze i ma wiele wyspecjalizowanych układów, które obsługują grafikę, dźwięk i dane wejściowe. Emulator musi być w stanie dokładnie emulować wszystkie te komponenty, aby poprawnie uruchamiać gry NES.

Jednym z kluczowych wyzwań podczas opracowywania emulatora NES jest osiągnięcie dokładnej emulacji sprzętu graficznego i dźwiękowego. NES wykorzystuje unikalny system generowania grafiki i dźwięku, który różni się od nowoczesnych systemów. Grafika jest generowana przy użyciu techniki zwanej „renderowaniem opartym na kafelkach” (*z ang. tile-based rendering*), w której małe kafelki o wymiarach 8×8 pikseli są łączone w celu utworzenia większych obrazów. Dźwięk jest generowany przy użyciu wyspecjalizowanych układów scalonych, które wytwarzają charakterystyczny 8-bitowy dźwięk, unikalny dla NES.

Oprócz dokładnej emulacji sprzętu emulator NES musi również być w stanie dokładnie emulować oprogramowanie działające na NES. Gry NES zostały zaprogramowane przy użyciu asemblera, który jest językiem programowania niskiego poziomu, specyficzny dla architektury NES.

Emulatory stały się ważnym narzędziem do zachowywania historii gier video, ponieważ pozwalają graczom doświadczyć klasycznych gier na nowoczesnym sprzęcie. Rozwój nowych emulatorów również przyczyną się do ciągłej ewolucji branży gier. Umożliwia to programistom eksperymentowanie z nowymi pomysłami i technologiami w bezpiecznym i kontrolowanym środowisku. [1]

1.2 Cel

Celem tej pracy jest stworzenie emulatora konsoli NES, który ma na celu jak najdokładniejsze odwzorowanie zachowania oryginalnego sprzętu. Aby to osiągnąć, niniejsza praca skupi się na emulacji procesora MOS 6502 i procesora obrazu (*PPU*), które są głównymi komponentami NES. Dodatkowo emulator będzie obsługiwał format pliku iNES do importu kartridżów oraz będzie zawierał usługę co najmniej jednego typu Mapper'a, czyli komponentu

sprzętowego rozszerzającego zakres jednoczesnego adresowania danych przez PPU. Ostateczna implementacja emulatora zostanie wykonana przy użyciu języka C#.

Wybór języka C# jako języka implementacji emulatora NES wynika z faktu, że umożliwia on uzyskanie przejrzystego i ustrukturyzowanego kodu, co ułatwia zrozumienie i utrzymanie architektury emulatora. C# jest również popularnym i powszechnie używanym językiem programowania, co oznacza, że istnieje duża społeczność programistów, którzy go znają i mogą zapewnić wsparcie i zasoby. Ponadto język C# jest wieloplatformowy, co oznacza, że można go uruchomić na wielu systemach operacyjnych, takich jak Windows i MacOS, co czyni go dobrym wyborem dla projektu, który wspiera inne implementacje GUI. Ta funkcja pomoże zapewnić, że projekt może być używany przez szerokie grono użytkowników, niezależnie od ich systemu operacyjnego. [2]

Skupiono się na procesorze MOS 6502 i jednostce przetwarzania obrazu (*PPU*) jako głównych komponentach emulatora, ponieważ są to podstawowe komponenty kontrolujące zachowanie konsoli NES. Dzięki dokładnej emulacji tych komponentów emulator będzie w stanie jak najdokładniej odtworzyć zachowanie oryginalnego sprzętu.

Obsługa formatu pliku iNES do importowania kartridżów jest niezbędna, aby emulator mógł uruchomić gry, ponieważ jest to powszechnie używany format dla NES ROM. Obsługa co najmniej jednego typu Mapper'a jest również ważna, ponieważ jest to komponent sprzętowy, który umożliwia PPU dostęp do większego zakresu pamięci, co jest niezbędne w niektórych grach, które zużywają więcej pamięci niż standardowa przestrzeń adresowa NES. [3]

2. Podstawy techniczne emulacji konsoli NES

2.1 CPU

2.1.1 2A03

Jednostka centralna NES, znana również jako 2A03, to mikroprocesor używany w konsoli do gier wideo Nintendo Entertainment System (*NES*). Jest on oparty na mikroprocesorze MOS Technology 6502, który był popularnym wyborem dla komputerów domowych i konsoli do gier w latach 80. 2A03 jest taktowany zegarem 1,79 MHz i ma 8-bitową magistralę danych. Posiada łącznie 40 pinów. 2A03 ma wbudowaną jednostkę przetwarzania dźwięku (*ang. audio processing unit, APU*), która służy do generowania dźwięku w grach NES. [5]

2.1.2 Rejestry

Procesor 2A03 używany w konsoli NES zawiera kilka rejestrów, które służą do przechowywania danych i sterowania pracą procesora. Rejestry te obejmują [6]:

- **Rejestr licznika programu** (*ang. program counter, PC*): Ten 16-bitowy rejestr przechowuje adres pamięci następnej instrukcji, która zostanie wykonana przez CPU. Licznik programu jest zwiększany o liczbę bajtów w instrukcji po wykonaniu każdej instrukcji, umożliwiając procesorowi sekwencyjne poruszanie się po programie. Licznik programu można modyfikować za pomocą instrukcji skoku (*ang. jump*), rozgałęzienia (*ang. branch*) i podprogramu (*ang. subroutine*), umożliwiając procesorowi przeskakiwanie do różnych miejsc w pamięci, wykonując inny kod.
- **Wskaźnik stosu** (*ang. Stack Pointer, SP*): Ten 8-bitowy rejestr wskazuje szczyt stosu, obszar pamięci używany do przechowywania danych tymczasowych podczas wykonywania programu. Wskaźnik stosu jest zmniejszany o jeden, gdy dane są umieszczane na stosie i zwiększany o jeden, gdy dane są usuwane ze stosu. Stos służy do przechowywania adresów zwrotnych, gdy podprogram jest wywoływany oraz do zapisywania danych tymczasowych, gdy proces jest przerwany.
- **Akumulator** (*ang. Accumulator, A*): Ten 8-bitowy rejestr służy do przechowywania wyników operacji arytmetycznych i logicznych wykonywanych przez CPU. Akumulator jest również używany jako jeden z operandów w większości instrukcji, umożliwiając procesorowi wykonywanie na nim operacji i przechowywanie wyników. Może być używany na przykład do dodawania, odejmowania, operacji bitowych i wielu innych operacji.
- **Rejestr indeksu X (X)**: Ten 8-bitowy rejestr służy do przechowywania wartości, która może być użyta jako przesunięcie dla operacji pamięciowych. Może być używany na przykład jako indeks w pętli lub do uzyskania dostępu do elementów w tablicy w pamięci.
- **Rejestr indeksu Y (Y)**: Ten 8-bitowy rejestr jest podobny do rejestrów X, służy również do przechowywania wartości, która może być wykorzystywana jako przesunięcie dla operacji pamięciowych. Może być używany na przykład jako drugi indeks w tablicy 2D lub w połączeniu z rejestrzem X w celu uzyskania dostępu do danych w większej tablicy.
- **Rejestr stanu procesora** (*ang. Processor Status, P*): Ten 8-bitowy rejestr zawiera kilka indywidualnych flag wskazujących stan procesora i wyniki niektórych operacji. Flagi te obejmują: [7]

- **Flaga przeniesienia** (*ang. Carry Flag, C*): Wskazuje, czy ostatnia operacja arytmetyczna zakończyła się przeniesieniem (*ang. carry out*) lub zapożyczeniem (*ang. borrow in*). Można jej użyć na przykład do sprawdzenia, czy liczba jest zbyt duża, aby zmieściła się w rejestrze 8-bitowym lub do wykonania arytmetyki wielobajtowej.
- **Flaga zera** (*ang. Zero Flag, Z*): Wskazuje, czy wynikiem ostatniej operacji było zero. Można go użyć na przykład do sprawdzenia, czy liczba jest równa zero lub do porównania dwóch liczb.
- **Flaga maski przerwania** (*ang Interrupt Mask, I*): Ta flaga służy do włączania lub wyłączania przerwań. Gdy flaga jest ustawiona, system nie będzie mógł wykonywać przerwań, chociaż przerwań niemaskowalnych nie można wyłączyć. Oznacza to, że jeśli flaga *Interrupt Mask* jest ustawiona, CPU nie będzie odpowiadać na żadne przerwania, z wyjątkiem przerwań niemaskowalnych. Może to być przydatne w sytuacjach, w których konieczne jest tymczasowe wyłączenie przerwań, aby uniemożliwić im zakłócania ważnych operacji.
- **Tryb dziesiętny** (*ang. Decimal Mode, D*): Wskazuje, czy procesor jest w trybie dziesiętnym z kodowaniem binarnym (BCD), czy w normalnym trybie binarnym. **W 2A03 i 2A07 ta funkcjonalność została usunięta, prawdopodobnie by uniknąć problemów licencyjnych** [8]. Ten tryb jest rzadko używany w grach NES i większość z nich pozostawia tę flagę w stanie 0, gry, które obsługiwają ten tryb prawdopodobnie były napisane dla nieoryginalnych konsol z innym procesorem niż 2A03/2A07. [9]
- **Polecenie przerwania** (*ang. Break Command, B*): Wskazuje, czy ostatnią wykonaną instrukcją była instrukcja BRK. BRK to niemaskowalna instrukcja przerwania, której można użyć na przykład do obsługi sytuacji awaryjnej lub do zaimplementowania funkcji debugowania.
- **Flaga przepełnienia** (*ang. Overflow Flag, V*): Wskazuje, czy ostatnia operacja spowodowała przepełnienie lub niedopełnienie. Można go użyć na przykład do sprawdzania, czy liczba nie jest zbyt duża, aby zmieścić się w rejestrze 8-bitowym, lub czy jest za mała.
- **Flaga ujemna** (*ang. Negative Flag, N*): Wskazuje, czy wynik ostatniej operacji był ujemny. Można go użyć na przykład do sprawdzenia, czy liczba jest ujemna lub do zaimplementowania reprezentacji liczb według wielkości znaku.



Rysunek 1. Binarna reprezentacja rejestru stanu procesora w postaci jednego bajta. Od lewej (tj. najbardziej znaczącego bitu): flaga ujemna (N), flaga przepełnienia (V), nieużywany bit (zawsze ustawiony na 1), polecenie przerwania (B), tryb dziesiętny (D), flaga maski przerwania (I), flaga zera (Z) i flaga przeniesienia (C).

Podsumowując, rejesty te są używane przez procesor do przechowywania danych i informacji kontrolnych podczas wykonywania programu. Licznik programu śledzi bieżącą instrukcję, wskaźnik stosu zarządza pamięcią stosu, akumulator przechowuje wyniki operacji arytmetycznych, rejestr X i Y służą jako przesunięcia operacji pamięciowych, a rejestr stanu procesora zawiera flagi wskazujące aktualny stan procesora.

Należy zauważyć, że procesor NES jest oparty na 6502, który jest procesorem 8-bitowym, co oznacza, że rozmiar rejestrów jest 8-bitowy, a adresowanie pamięci jest ograniczone do 64 KB, więc ma ograniczone możliwości w porównaniu z nowoczesnymi procesorami. Jest to jednak wystarczające, aby konsola mogła uruchamiać rozmaite gry.

2.1.3 Tryby adresowania

Tryby adresowania to sposoby, w jakie procesor może uzyskać dostęp do lokalizacji pamięci w celu wykonania na nich operacji. Różne tryby adresowania zapewniają różne poziomy elastyczności i kontroli nad dostępną lokalizacją pamięci. Na przykład 6502 ma 13 różnych trybów adresowania, z których każdy ma swoje unikalne cechy [10][11].

- **Implikowany** (*ang. Implicit*): Dane, na których operuje się, wynikają bezpośrednio z samej instrukcji. Ten tryb adresowania jest używany przez instrukcje, które działają na określonych rejestrach, takich jak rejestr stanu.
- **Akumulatora** (*ang. Accumulator*): Ten tryb adresowania jest używany przez instrukcje działające bezpośrednio na akumulatorze, rejestrze używanym do operacji arytmetycznych i logicznych.
- **Natychmiastowy** (*ang. Immediate*): Ten tryb adresowania umożliwia programiście określenie wartości literalnej, zwykle 8-bitowej, która znajduje się bezpośrednio po instrukcji. Ten tryb jest powszechnie używany do ładowania wartości do rejestrów.
- **Strony zerowej** (*ang. Zero Page*): Ten tryb adresowania pozwala na szybki i bezpośredni dostęp do pierwszej strony pamięci, czyli lokacji pamięci od adresu 0 do 255. Ten tryb jest przydatny do minimalizacji zużycia pamięci i zwiększenia szybkości.
- **Bezwzględny** (*ang. Absolute*): Ten tryb adresowania umożliwia programiście określenie bezwzględnego adresu pamięci, który jest wartością 16-bitową. Indeksowanej wersji tego trybu można użyć do przesunięcia adresu za pomocą rejestrów X lub Y.
- **Względny** (*ang. Relative*): Ten tryb adresowania jest używany specjalnie do instrukcji rozgałęziających (*ang. branching instructions*). Pozwala programiście określić podpisana 8-bitową wartość, która jest dodawana do licznika programu. Ten tryb jest przydatny do wykonywania względnych skoków w pamięci.
- **Pośredni** (*ang. Indirect*): Ten tryb adresowania umożliwia programiście określenie adresu pamięci, który jest następnie używany do dostępu do innej lokalizacji pamięci. Ten tryb jest przydatny do pośredniego dostępu do pamięci. Ten tryb ma błąd przy zawijaniu (*ang. wrapping bug*) [12].
- **Indeksowy pośredni** (*ang. Indexed Indirect*) / **Pośredni preindeksowany** (*ang. Indirect pre-indexed*) i **Pośredni indeksowany** (*ang. Indirect Indexed*) / **Pośredni postindeksowany** (*ang. Indirect post-indexed*): Oba te tryby adresowania obejmują użycie adresu pamięci, który jest najpierw modyfikowany przez rejestyre X lub Y, a następnie używany do uzyskania dostępu do innej lokalizacji pamięci. Różnica między nimi polega na tym, kiedy następuje modyfikacja rejestrów: w trybie indeksowanym pośrednim adres jest modyfikowany przed użyciem do uzyskania dostępu do pamięci, podczas gdy w trybie pośrednim indeksowanym adres jest najpierw używany do uzyskiwania dostępu do pamięci, a następnie modyfikowany.

Każdy tryb adresowania ma swoje zalety i wady i może być używany w różnych sytuacjach w zależności od potrzeb programu. Ważne jest, aby wybrać odpowiedni tryb adresowania dla zadania, ponieważ może to znacznie wpływać na wydajność i wydajność programu.

2.1.4 Instrukcje

Procesor 2A03 łącznie posiada 56 instrukcji [13], które można podzielić na kilka grup funkcjonalnych. Te grupy obejmują:

- Operacja ładowania (*ang. Load*) / zapisywania (*ang. Store*): Instrukcje te służą do ładowania rejestru z pamięci lub zapisywania zawartości rejestru w pamięci. Przykłady obejmują LDA, STA i TXS.
- Operacje transferu rejestru (*ang. Register Transfer*): Te instrukcje służą do kopiowania zawartości rejestru X lub Y do akumulatora, lub kopiowania zawartości akumulatora do rejestru X lub Y.
- Operacje na stosie (*ang. Stack*): Instrukcje te służą do składowania (*ang. push*), pobierania (*ang. pull*) ze stosu lub manipulowania wskaźnikiem stosu za pomocą rejestru X.
- Operacje logiczne: Te instrukcje służą do wykonywania operacji logicznych na akumulatorze i wartości przechowywanej w pamięci. Przykłady ADC, ROLA i EOR.
- Operacje arytmetyczne: Te instrukcje są używane do wykonywania operacji arytmetycznych na rejestrach i pamięci.
- Inkrementacje / Dekrementacje: Instrukcje te służą do zwiększania lub zmniejszania rejestrów X, Y lub wartości przechowywanej w pamięci.
- Przesunięcia (*ang. Shifts*): Instrukcje te służą do przesuwania bitów akumulatora lub komórki pamięci o jeden bit w lewo, lub prawo.
- Skoki / Wywołania (*ang. Calls*): Instrukcje te służą do przerywania sekwencyjnego wykonywania kodu i wznowianie od określonego adresu.
- Rozgałęzienia (*ang. Branches*): Instrukcje te służą do przerywania sekwencyjnego wykonywania kodu i wznowianie od określonego adresu, jeśli spełniony jest warunek. Warunek polega na sprawdzeniu określonego bitu w rejestrze stanu.
- Operacje na rejestrze stanu (*ang. Status Register*): Te instrukcje służą do ustawiania lub kasowania flagi w rejestrze stanu.
- Funkcje systemowe (*ang. System Functions*): Te instrukcje służą do wykonywania rzadko używanych funkcji.

Każda instrukcja ma unikalny kod operacji z zakresu od 0x00 do 0xFF. Podczas wykonywania instrukcji procesor pobiera i dekoduje kod operacji oraz uruchamia określoną instrukcję. Wykonanie każdej instrukcji wymaga określonej liczby cykli, co wynika z wewnętrznych elementów procesora wraz z jego magistralą i pamięcią. Na przykład w magistrali może znajdować się tylko jedna wartość w tym samym czasie. Niektóre instrukcje występują w wielu odmianach wykorzystujących różne tryby adresowania, co daje w sumie 151 prawidłowych kodów operacji (*ang. opcode*) z możliwych 256[14]. Instrukcje mają długość jednego, dwóch lub trzech bajtów, w zależności od trybu adresowania. Pierwszy bajt to kod operacji, a pozostałe bajty to operandy.

2.1.5 Przerwania

Przerwania to mechanizm do tymczasowego zatrzymania normalnego wykonywania instrukcji w procesorze. Przerwania te pozwalają procesorowi zająć się ważnymi zadaniami lub zdarzeniami, które mają miejsce poza głównym przepływem programu. 2A03 ma trzy różne typy przerwań: Reset, NMI (*przerwanie niemaskowalne*) i IRQ (*żądanie przerwania*). [15]

Gdy występuje przerwanie, aktualny stan programu, w tym licznik programu i rejestrystru stanu, są zapisywane na stosie. Następnie procesor przeskakuje do predefiniowanej lokalizacji w pamięci, zwanej wektorem przerwań, aby wykonać określoną procedurę obsługi przerwania. Po obsłużeniu przerwania procesor powraca do poprzedniego stanu, przywracając wartości ze stosu i kontynuując wykonywanie od miejsca, w którym zostało przerwane.

Przerwania resetowania są wyzwalane, gdy NES jest włączony lub po naciśnięciu przycisku resetowania. Przerwanie to ma najwyższy priorytet i powoduje, że procesor przeskakuje pod adres znajdujący się na 0xFFFFC i 0xFFFFD.

Przerwania NMI są wyzwalane przez PPU po zakończeniu rysowania ramki i wejściu w stan zwany V-Blank. Ten mały przedział czasu to jedyny moment, w którym procesor może wysyłać dane do pamięci PPU bez powodowania uszkodzenia grafiki (*np. artefakty*). Wektor przerwań dla NMI znajduje się w 0xFFFFA i 0xFFFFB.

Przerwanie IRQ są generowane przez Mapper'y pamięci lub instrukcje BRK. Te przerwania są ignorowane przez procesor, jeśli ustawiona jest flaga wyłączenia przerwania. Wektor przerwań dla IRQ znajduje się w 0xFFFFE i 0xFFFFF.

Należy zauważyć, że NES ma opóźnienie przerwania wynoszące 7 cykli, co oznacza, że rozpoczęcie wykonywania procedury obsługi przerwań po wyzwoleniu przerwania zajmuje 7 cykli procesora. [16]

2.1.6 Mapa pamięci i urządzenia we/wy zmapowane w pamięci

2A03 ma 16-bitową magistralę adresującą, co pozwala mu zaadresować do 64 KB pamięci. Jednak pamięć fizyczna w NES jest ograniczona do 2 KB, a część tej pamięci jest zarezerwowana dla określonych funkcji, takich jak dźwięk, komunikacja PPU i DMA. [17]

CPU uzyskuje dostęp do pamięci za pomocą magistrali, przy czym magistrala adresowa ustawia adres wymaganej lokalizacji, magistrala sterująca wskazuje, czy żądanie jest odczytem, czy zapisem, a magistrala danych odczytuje lub zapisuje bajt pod wybranym adresem. Pamięć jest podzielona na trzy części: ROM wewnętrz kartridża, pamięć RAM kartridża, pamięć RAM procesora, rejstry I/O i rozszerzenie pamięci ROM. *Tabela 1* przedstawia mapę pamięci procesora używaną przez NES, pokazującą układ pamięci.

Tabela 1. Mapa przestrzeni adresowej konsoli / CPU

Adresy	Opis	Sprzęt
0xFFFF	PRG-ROM Bank 1	Pamięć programu ROM kartridża (32 KB)
0xC000		
0xBFFF	PRG-ROM Bank 0	Pamięć RAM kartridża (8 KB)
0x8000		
0x7FFF	SRAM lub WRAM	Pamięć RAM kartridża (8 KB)
0x6000		
0x5FFF	<i>Expansion ROM</i>	
0x4020		
0x401F	Rejestry APU, sygnały z kontrolerów (<i>joypadów</i>), DMA	Rejestry odwzorowane w przestrzeni adresowej (<i>Memory mapped I/O, MMIO</i>)
0x4000		
0x3FFF	<i>Mirror adresów 0x2000 – 0x2007</i>	
0x2008		
0x2007	Rejestry PPU	Pamięć RAM procesora (2 KB)
0x2000		
0x1FFF	<i>Mirror adresów 0x0000 – 0x07FF</i>	
0x0800		
0x07FF	RAM	Pamięć RAM procesora (2 KB)
0x0200		
0x01FF	Stos (<i>Stack</i>)	Pamięć RAM procesora (2 KB)
0x0100		
0x00FF	Strona zerowa (<i>Zero Page</i>)	Pamięć RAM procesora (2 KB)
0x0000		

Pamięć jest podzielona na różne sekcje, takie jak strona zerowa (*0x0-0xFF*), która jest używana w niektórych trybach adresowania w celu oszczędzania pamięci i poprawy szybkości, stos (*0x100-0x1FF*) oraz zwykła pamięć RAM do przechowywania danych gry (*0x200-0x800*). Ponadto pamięć jest wielokrotnie dublowana (*ang. mirrored*), co oznacza, że dostęp do określonego adresu pamięci w jednej lokalizacji zwróci taką samą wartość, jak w dostępie do tego samego adresu w innej lokalizacji.

CPU i PPU komunikują się poprzez MMIO (*ang. memory mapped I/O*) [19], w tej metodzie rejestrów są odwzorowane w przestrzeni adresowej. Na rejestrach 0x2000 – 0x2007, a CPU wysyła dźwięk do APU przez rejestrów 0x4000 – 0x4020. Wejście kontrolera NES jest przechowywane pod adresem 0x4016 – 0x4017 i jest odczytywane przez rejestr przesuwny, który zajmuje 24 odczyty, aby pobrać wszystkie informacje z kontrolera.

Region pamięci 0x4020 – 0xFFFF składa się z:

Rozszerzona pamięć ROM odnosi się do pamięci, która nie jest częścią standardowej wbudowanej pamięci urządzenia, ale raczej dodatkową funkcją dodaną do urządzenia w celu rozszerzenia jego możliwości. W kontekście pamięci NES region pamięci zwany *Expansion ROM* jest obszarem pamięci pomiędzy adresami 0x4020 – 0x5FFF. Ten region może zawierać dodatkowy kod lub dane używane przez kartę NES w celu zapewnienia dodatkowych funkcjonalności wykraczających poza to, co zapewnia podstawowy system NES.

Pamięci RAM karty (znana również jako *Work RAM* lub *SRAM*). Jest to pamięć o dostępie swobodnym na samym kartę, która może być używana przez oprogramowanie NES do przechowywania danych. Ten typ pamięci jest ulotny, co oznacza, że jej zawartość jest tracona po wyłączeniu zasilania. Niektóre karty wykorzystywały do rozszerzenia dostępnej pamięci VRAM, a nawet do zapisywania stanu gry (*w tym przypadku karta dodatkowo posiadała baterię do ciągłego zasilania pamięci*) [18].

PRG-ROM jest podzielony na „górnego” i „dolnego” banki, aby reprezentować pojedynczą, ciągłą przestrzeń ROM w ograniczonej przestrzeni adresowej. Termin „bank” odnosi się do sekcji pamięci, którą można wybrać i uzyskać do niego dostęp niezależnie od innych sekcji. Pozwala to na użycie większych pamięci ROM w systemach z ograniczoną przestrzenią adresową.

Jednak PRG-ROM może mieć więcej niż dwa banki. Na przykład niektóre karty mogą mieć wiele banków pamięci ROM, z których każdy można wybrać i uzyskać do niego dostęp niezależnie. Pozwala to na przechowywanie większych gier na jednym karcie, a system ma dostęp tylko do określonego banku danych, którego potrzebuje w danym momencie. Liczba banków używanych przez kartę może być różna, a niektóre mogą mieć nawet 32 banki. Konkretna implementacja przełączania banków może się różnić w zależności od Mapper'a karty.

2.2 PPU

2.2.1 2C02

PPU (*Picture Processing Unit*) to wyspecjalizowany układ opracowany przez Nintendo, który współpracuje z procesorem jako koprocesor w konsoli NES. Jest odpowiedzialny za generowanie sygnałów wideo z danych graficznych przechowywanych w pamięci. Został opracowany przez Ricoh i jest znany jako Ricoh RP2C02 (*dla wersji NTSC*) lub RP2C07 (*dla wersji PAL*). PPU ma własną przestrzeń adresową, która zawiera 10 kilobajtów pamięci (*8 kilobajtów pamięci ROM lub RAM w karcie i 2 kilobajty pamięci RAM w konsoli*) do przechowywania kształtów tła i sprite'ów, paletę kolorów i OAM (*pamięć atrybutów obiektów*) do przechowywania atrybutów sprite'ów.[20]

PPU ma paletę 64 kolorów, z czego 25 kolorów jest w stanie wyświetlić na raz i może wyświetlić 64 sprite'ów jednocześnie. PPU działa z prędkością około trzech razy większą niż CPU, czyli około 5,37 MHz w systemie NTSC. Cała komunikacja pomiędzy CPU i PPU odbywa się poprzez MMIO zlokalizowane w 0x2000 – 0x2007. PPU działa poprzez renderowanie 240 widocznych linii wybierania/skanowania (*ang. scanlines*) 60 razy na sekundę. Grafika najpierw jest renderowana przez wpisy w tabeli nazw (*ang. Name Tables*), na tej linii może narysować

do 8 sprite'ów (jedna klatka może mieć maksymalnie 64 sprite'y), powtarzając ten proces 240 razy przed wejściem w stan V-Blank (vertical blank). [21]

2.2.2 Rejestry

Jednostka przetwarzania obrazu (*PPU*) NES jest kontrolowana przez zapis do dwóch rejestrów odwzorowanych w przestrzeni adresowej (*MMIO*), rejestru kontrolnego *PPU 1* (*0x2000*) i rejestru *PPU 2* (*0x2001*). Rejestry te są używane do kontrolowania różnych aspektów zachowania *PPU*, takich jak wyłączenie/włączenie NMI (*Non-Maskable Interrupt*), wybór rozmiaru duszka i zwiększenie adresu. Rejestr stanu *PPU* (*0x2002*) jest używany przez *PPU* do raportowania swojego stanu do *CPU* i jest tylko do odczytu. Inne rejesty *PPU* mapowane w pamięci obejmują rejestr adresów *OAM* (*0x2003*), rejestr danych *OAM* (*0x2004*), rejestr przewijania *PPU* (*0x2005*), rejestr adresów *PPU* (*0x2006*) i rejestr danych *PPU* (*0x2007*) oraz rejestr *OAM DMA* (*0x4014*).[22]

Szczegóły dotyczące bitów i funkcji każdego z tych rejestrów:

PPUCTRL (0x2000)

- Bit 0-1 (*NN*): wybór tablicy nazw, wybiera tablicę nazw dla tła.
- Bit 2 (*I*): tryb przyrostowy (*z ang. increment mode*), gdy ustawiony na 1, następny adres do odczytu/zapisu z pamięci *PPU* jest zwiększany w pionie o 32, w przeciwnym razie jest zwiększany w poziomie o 1.
- Bit 3 (*S*): wybór kafelków duszków, wybiera tabelę wzorów dla sprite'ów (*przesunięcie o 0KB lub 4KB*).
- Bit 4 (*B*): wybór kafelków tła, wybiera tabelę wzorów dla tła (*przesunięcie o 0KB lub 4KB*).
- Bit 5 (*H*): wysokość duszka, ustawia wysokość duszków na 8x8 lub 8x16 pikseli.
- Bit 6 (*P*): tryb master/slave, kierunek magistrali EXT (*0 – wejście, 1 – wyjście*).
- Bit 7 (*V*): włączenie NMI, po ustawieniu na 1 zostanie wyzwolony V-Blank NMI.

PPUMASK (0x2001)

- Bit 0 (*G*): tryb greyscale (*skala szarości/odcienie szarości*), po ustawieniu na 1, obraz będzie wyświetlany w odcieniach szarości.
- Bit 1 (*m*): włączenie renderowania tła na lewej kolumnie ekranu, gdy ustawiony na zero, 8 pikseli od lewej strony ekranu nie wyświetli tła. *
- Bit 2 (*M*): włączenie renderowania sprite'ów na lewej kolumnie ekranu, gdy ustawiony na zero, 8 pikseli od lewej strony ekranu nie wyświetli sprite'ów. *
- Bit 3 (*b*): włączenie tła, gdy jest ustawione na 0, ukrywa tło.
- Bit 4 (*s*): włączenie sprite'ów, gdy ustawiony na 0, ukrywa sprite'y.
- Bit 5-7 (*RGB*): podkreślenie kolorów (*z ang. „color emphasis”*), jeżeli ustawimy zielony bit, obraz stanie się bardziej zielony (*0b010*). Zgodnie z informacjami podanymi w [23], zielony będzie w 108.6% bardziej podkreślony (*podczas gdy wartość koloru niebieskiego spadnie do 88.2% i czerwonego do 79.4%*).

*Lewa krawędź ekranu ma specjalne przełączniki do sterowania jego wyglądem. Służą do wygładzania niespójności podczas przewijania.

PUSTATUS (0x2002)

- Bit 0-4: nieużywane.
- Bit 5 (**O**): przepełnienie duszków, gdy jest ustawione na 1, wskazuje, że na tej samej linii wybierania/skanowania (*ang. scanline*) znajduje się więcej niż 8 sprite'ów.
- Bit 6 (**S**): flaga „Sprite Zero Hit”, pierwszy piksel pierwszego sprite'a z pamięci OAM, zderzył się z nieprzezroczystym pikselem pola gry w ostatniej klatce.
- Bit 7 (**V**): flaga V-Blank, gdy jest ustawione na 1, wskazuje, że jest w stanie V-Blank.

OAMADDR (0x2003)

Określa adres w pamięci atrybutów obiektu (OAM) do odczytu lub zapisu.

OAMDATA (0x2004)

Odczytuje lub zapisuje dane z/do OAM.

PPUSCROLL (0x2005)

Określa przesunięcie przewijania w poziomie i w pionie tła.

PPUADDR (0x2006)

Określa adres w pamięci PPU do odczytu lub zapisu.

PPUDATA (0x2007)

Odczytuje lub zapisuje dane z/do pamięci PPU.

OAMDMA (0x4014)

Przesyła dane z pamięci procesora do pamięci OAM, używanej do ładowania sprite'ów.

2.2.3 OAM

Pamięć atrybutów obiektów (*OAM*) to pamięć wewnętrz PPU zawiera informacje o maksymalnie 64 sprite'ów, z których każdy zajmuje 4 bajty pamięci. Informacje przechowywane w OAM obejmują pozycję Y i X duszka, numer indeksu kafelka oraz atrybuty, takie jak odwracanie (*ang. flipping*). Informacje te są zwykle kopowane z pamięci procesora do OAM za pośrednictwem transferu bezpośredniego dostępu (*DMA*), który w praktyce jest szybszy niż gdyby używać rejestrów OAMADDR i OAMDATA. [24]

Pierwszy sprite jest znany jako „Sprite Zero” i jest powszechnie używany do podzielenia renderowanego obrazu na dwie części, z którego tylko jedna będzie przewijać tło. Gdy pierwszy duszek z tablicy zostanie narysowany na nieprzezroczystym tle, PPU ustawia flagę „Sprite Zero Hit”. [25]

2.2.4 Mapa pamięci PPU

PPU adresuje przestrzeń 16 KB, 0x0000 – 0x3FFF, która jest oddzielona od szyny adresowej procesora. Mapa pamięci PPU jest podzielona na trzy główne sekcje: tablica wzorów (*ang. Pattern Tables*), tablice nazw (*ang. Name Tables*). Dodatkowo istnieje jeszcze pamięć

atrybutów obiektów (*OAM*) i „*Secondary OAM*”, które nie są obecne na tej przestrzeni adresowej. [26]

Tabele wzorów służą do przechowywania danych kafelków tła i sprite'ów. Istnieją dwie tablice wzorów, zwanyimi, „*Pattern Table 0*” i „*Pattern Table 1*”. Każda znajduje się odpowiednio w zakresach 0x0000 – 0x0FFF i 0x1000 – 0x1FFF. Każda ma rozmiar 4 KB (*0x1000*). Tabele te są zwykle mapowane bezpośrednio na pamięć ROM lub RAM ze znakami.

Tablice nazw są używane do tworzenia tła poprzez indeksowanie do tabel wzorów jego elementów. Istnieją cztery tablice nazw znajdujące się w zakresach 0x2000 – 0x23FF, 0x2400 – 0x27FF, 0x2800 – 0x2BFF i 0x2C00 – 0x2FFF, z czego dwa ostatnie są kopiami (*ang. mirrored*) dwóch pierwszych. Każda tabela nazw zawiera 32×30 kafelków (*960 Bajtów*), co odpowiada pełnej klatce 256×240 pikseli. Dodatkowo każda tabela ma tabele atrybutów o wielkości 64 Bajtów, co łącznie daje 1 KB na jedną tablicę nazw.

Indeksy palet znajdują się w zakresie adresów 0x3F00 – 0x3F1F i mają rozmiar 32 Bajtów (*0x0020*). Ta sekcja pamięci przechowuje dane palet zarówno dla sprite'ów jak i kafelków tła. PPU jest w stanie wytworzyć ponad 50 różnych kolorów, ale tylko 8 aktywnych palet z 4 kolorami każda może być zdefiniowana jednocześnie.

Tabela 2. Mapa przestrzeni adresowej PPU

Adres	Opis	Sprzęt
0xFFFF	Mirror adresów 0x0000 – 0x3FFF (16 KB)	
0x4000		
0x3FFF	Mirror adresów 0x3F00 – 0x3F1F (32 B)	Mirror palet (32 B)
0x3F20		
0x3F1F	Paleta duszków (16 B)	
0x3F10		Pamięć palet SRAM (32 B)
0x3F0F	Paleta tła (16 B)	
0x3F00		
0x3EFF	Nieużywana przestrzeń adresowa	
0x3000		
0x2FFF	Mirror adresów 0x2000 – 0x23FF (1 KB) i 0x2400 – 0x27FF (1 KB)	Mirror tabeli nazw (2 KB)
0x2800		
0x27FF	Tabela atrybutów 1 (64 B)	
0x27C0		
0x27BF	Tabela nazw 1 (960 B)	
0x2400		Pamięć VRAM PPU (2 KB)
0x23FF	Tabela atrybutów 0 (64 B)	
0x23C0		
0x23BF	Tabela nazw 0 (960 B)	
0x2000		

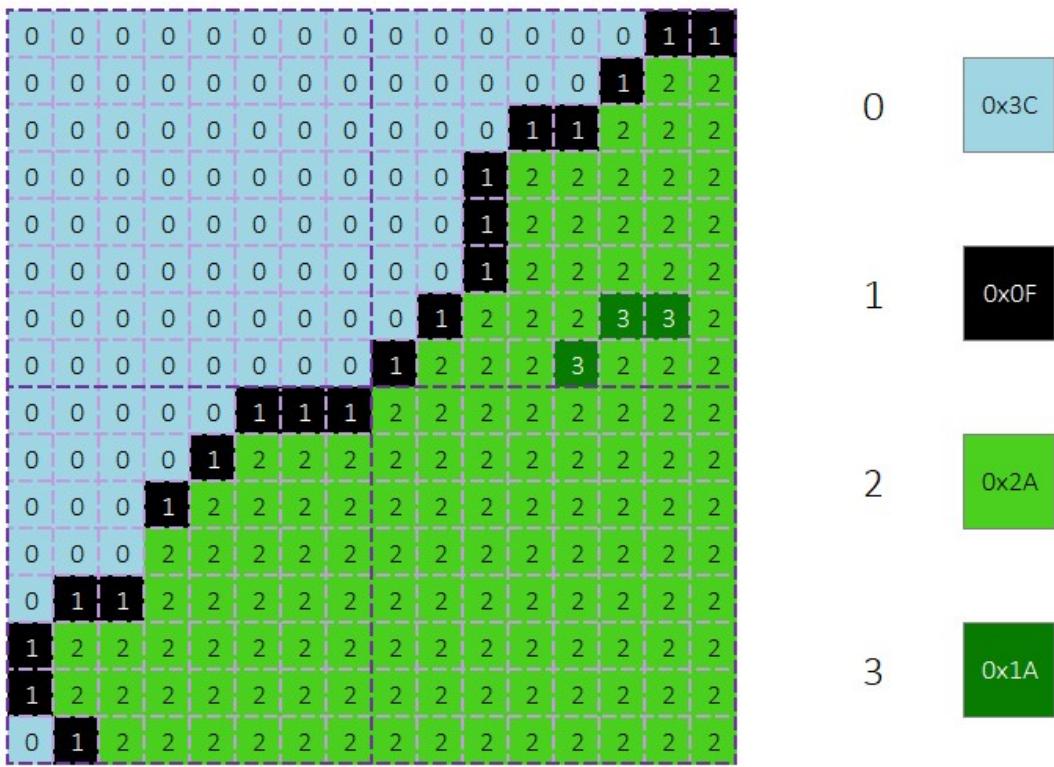
Tabela 3. Mapa przestrzeni adresowej PPU (c.d.)

Adres	Opis	Sprzęt
0x1FFF	Tabela wzorów 1 (4 KB)	
0x1000		
0x0FFF		Tabele wzorów (8 KB)
	Tabela wzorów 0 (4 KB)	Pamięć CHR ROM w kartridżu (8 KB)
0x0000		

2.2.5 Tabele wzorów i Palety kolorów

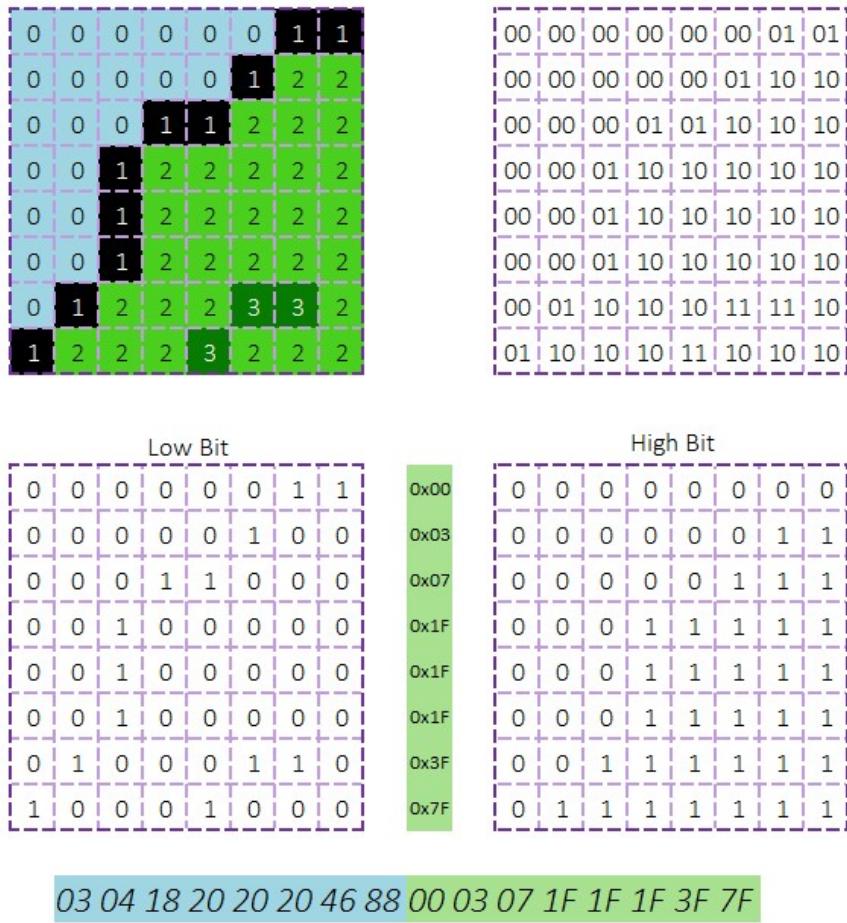
Obrazy sprite'ów, które są przechowywane w pamięci tabeli wzorów PPU, stanowią podstawę dla wszystkich grafik na NES. Te kafelki danych obrazu o wymiarach 8 na 8 pikseli składają się razem, tworząc rozbudowane tła, szybko poruszające się postacie i szeroką gamę efektów specjalnych. Dane dla kafelka sprite mają strukturę bardzo podobną do nowoczesnego obrazu komputerowego, takiego jak PNG. Oba formaty są reprezentowane przez dwuwymiarową siatkę liczb, przy czym każda wartość reprezentuje kolor dla danego piksela, ale tam, gdzie PNG może obsługiwać miliony kolorów, kafelek NES może obsługiwać jedynie 4. W praktyce kafelek w ogóle nie przechowuje danych o kolorach, liczba reprezentująca każdy piksel jest w rzeczywistości indeksem odnoszącym się do koloru w jednej z aktywnych palet systemu.

Palety mają specjalne miejsce w pamięci PPU i mogą być ustawione przez program gry, zapisując serię bajtów do tej pamięci. NES może przechowywać do 8 palet jednocześnie, 4 dla tła i 4 dla pierwszego planu. Każdy z tych palet zawiera 4 kolory. Pierwszy kolor dla każdej palety na indeksie zero tak naprawdę wcale nie jest kolorem, bez względu na to, jaką wartość ustawia programista, zawsze będzie interpretowany przez PPU jako kolor przezroczystości. Kiedy NES renderuje kafelek i napotka wartość 0 dla danego piksela, zinterpretuje ten piksel jako przezroczysty i po prostu nic nie renderuje.



Rysunek 2. Kompozycja bloku tła składającego się z 4 kafelków. Każdy kafelek ma siatkę pikseli z indeksami koloru aktywnej palety.

Łącząc to wszystko razem, sprite'y na NES są sformatowane jako siatka liczb, z których każda ma wartość od 0 do 3 i reprezentuje indeks koloru palety. Wartość 0 reprezentuje przezroczysty piksel sprite'a, podczas gdy wartości 1, 2 lub 3 reprezentują określony kolor w jednej z palet PPU. Ponieważ piksele są ograniczone do wartości od 0 do 3, oznacza to, że można je przechowywać przy użyciu tylko 2 bitów. Każdy kafelek 8×8 pikseli ma 64 piksele i wymaga użycia 128 bitów lub 16 bajtów pamięci. Ponieważ najmniejsza ilość pamięci, do której może odnosić się 6502, to bajt, programiści musieli wykazać się kreatywnością w sposobie przechowywania tych 2-bitowych obrazów kafelka na piksel.



Rysunek 3. Lewy górnny róg: Kafelek tła z siatką indeksów koloru palety, Prawy górnny róg: reprezentacja binarna wartości indeksów kafelka, Lewy dolny róg: niski bit wartości indeksów, Prawy dolny róg: wysoki bit wartości indeksów

Przyglądając się sposobowi przechowywania danych dla tych kafelków, skupimy się na prawym górnym kafelku kompozycji z *rysunku 2*. Jak wyjaśniono wcześniej, kafelek zawiera siatkę o wymiarach 8 na 8 pikseli, przy czym każdy piksel jest reprezentowany przez 2-bitową wartość indeksu koloru. Ponieważ NES nie może bezpośrednio pracować z liczbami dwubitowymi, zamiast tego, jak widać na *rysunku 3*, dzieli wartość dla każdego piksela na dwie oddzielne części, niski bit (*Low Bit*) i wysoki bit (*High Bit*). Aby zapisać obraz w pamięci tabeli wzorów, gra najpierw zapisuje niskie bity jako ciąg 8 bajtów, a zaraz potem zapisuje wysokie bity jako kolejny ciąg 8 bajtów. Łącząc to wszystko razem, otrzymujemy oczekiwane 16 bajtów danych dla obrazu kafelka, ale ułożonych w taki sposób, aby można było nimi łatwo manipulować przez procesor i poprawnie interpretować je przez PPU. [27]

2.2.6 Tabele nazw / Tabele atrybutów

PPU zawiera specjalny obszar pamięci do przechowywania tak zwanych tablic nazw (*ang. Name Tables*). Te tabele nazw to ogromne siatki bajtów, które odwołują się do kafelków w tabeli wzorów i służą do komponowania tła dla gry. Tabele nazw same w sobie są dość proste, są to po prostu siatki o wymiarach 32 na 30 kafelków, ponieważ każdy kafelek ma

dokładnie wymiary 8 na 8 pikseli, co oznacza, że każda tabela nazw reprezentuje obraz o wymiarach 256 na 240 pikseli, który wypełnia cały ekran. Każdy bajt odnosi się do kafelka w tablicy wzorów, a ponieważ istnieje maksymalnie 256 kafelków, oznacza to, że pojedynczy bajt może odnieść się do dowolnego z nich. [28]

Podstawowa implementacja gry przechowuje bajty całej tabeli wzorów w pamięci ROM, a następnie ładuje je jeden po drugim do pamięci tabeli nazw PPU. Po zakończeniu, gdy dane tła są załadowane, PPU może wyświetlić całą kompozycję na ekranie.

W przypadku gier produkcyjnych ponowne renderowanie całej tabeli nazw między każdą klatką zajmuje zbyt dużo czasu, więc większość gier wykorzystuje zaawansowane techniki przechowywania i przetwarzania danych tabeli nazw, dzięki czemu mogą aktualizować tła w czasie rzeczywistym bez spowalniania gry.

Na końcu każdej tablicy nazw znajduje się mały obszar pamięci zwany tablicą atrybutów (*ang. Attribute Table*). Jest to siatka bajtów, do której odwołuje się PPU w celu określenia, której palety użyć podczas renderowania kafelków obrazu w tabeli nazw. W porównaniu z tabelami nazw, tabele atrybutów są dość skomplikowane pod względem formatowania. [27]

2.2.7 Mirroring

Mirroring (ang. mirroring) to technika polegająca na powielaniu danych w celu uzyskania ich kopii lub odbicia. W kontekście konsoli NES, mirroring może odnosić się również do faktu, że ostatni bit adresu w pamięci VRAM jest nieużywany w celu uzyskania efektu powielenia obrazu na ekranie.

W przypadku konsoli NES, adresy w pamięci VRAM są 15-bitowe, a więc składają się z dwóch bajtów. Pierwszy bajt określa numer tabeli nazw, a drugi bajt określa konkretną komórkę w tabeli nazw. W technice mirroringu, ostatni bit adresu w drugim bajcie jest ignorowany i zawsze ustawiany na 0 lub 1 w zależności od wybranej techniki mirroringu.

W technice poziomego mirroringu, ostatni bit adresu w drugim bajcie zawsze jest ustawiony na 0 lub 1, co oznacza, że ta sama komórka w tabeli nazw jest używana zarówno dla górnej, jak i dolnej części ekranu. W technice pionowego mirroringu, ostatni bit adresu w drugim bajcie jest używany do wyboru jednej z dwóch tabel nazw, co oznacza, że ta sama komórka w tabeli nazw jest używana zarówno dla lewej, jak i prawej części ekranu.

W ten sposób, dzięki zastosowaniu techniki mirroringu, możliwe jest uzyskanie optymalizacji pamięci VRAM i umożliwienie wyświetlania tych samych obrazów na różnych obszarach ekranu.

2.3 Kartridż

Kartridż konsoli NES jest przede wszystkim odpowiedzialny za dostarczanie kodu programu gry i danych z obrazami sprite'ów. Kod programu jest przechowywany w układzie scalonym zwanym pamięcią programu ROM (*PRG-ROM*), a dane obrazu są przechowywane w pamięci znaków ROM (*CHR-ROM*) lub pamięci znaków RAM (*PRG-RAM*). Oba układy są połączone z CPU i PPU systemu za pośrednictwem magistrali, która istnieją na płytce drukowanej (*PCB*) dla samego kartridżu. Układy scalone ROM i RAM mają szereg pinów adresowych i pinów danych, które są połączone ścieżkami ze stykami na spodzie płytki. Kiedy

kartridż jest włożony do konsoli, te styki stykają się z pinami na złączu kartridża, powodując bezpośrednie połączenie układów scalonych do szyn adresowych i danych konsoli. Patrząc na układ pinów kartridża NES [POINT TO IMG], widać, że zdecydowana większość tych styków jest dedykowana do tego zadania. [29]

1	GND	31	RF VCC
2	A11	32	ϕ_2
3	A10	33	A12
4	A9	34	A13
5	A8	35	A14
6	A7	36	D7
7	A6	37	D6
8	A5	38	D5
9	A4	39	D4
10	A3	40	D3
11	A2	41	D2
12	A1	42	D1
13	A0	43	D0
14	R/W	44	ROM SEL
15	TRQ	45	SOUND 1
16	GND	46	SOUND 2
17	RÖ	47	WE
18	VRAM A10	48	VRAM CS
19	PPU A6	49	PPU A13
20	PPU A5	50	PPU A7
21	PPU A4	51	PPU A8
22	PPU A3	52	PPU A9
23	PPU A2	53	PPU A10
24	PPU A1	54	PPU A11
25	PPU A0	55	PPU A12
26	PPU D0	56	PPU A13
27	PPU D1	57	PPU D7
28	PPU D2	58	PPU D6
29	PPU D3	59	PPU D5
30	+5V	60	PPU D4

ϕ_2 – wyjście zegara procesora
 A0..A14 – magistrala adresowa procesora
 D0..D7 – dwukierunkowa magistrala danych procesora
 PPU A0..A13 – magistrala adresowa PPU
 PPU D0..D7 – dwukierunkowa magistrala danych PPU

Znak $\bar{}$ nad sygnałem oznacza AKTYWNY NISKI (0 V/masa)

Rysunek 4. Schemat przedstawiający rozmieszczenie pinów w kartridżu Famicom.

Rysunek 4 jest układem pinów dla kartridża *Famicom*, japońskiej wersji konsoli. Jest nieco prostszy niż układ pinów dla kartridża NES, ale jest funkcjonalnie równoważny i łatwiejszy do analizy.

Spośród 60 pinów, które są obecne na kartridżu *Famicom*, 4 z nich służą do dostarczania zasilania, a aż 45 z nich jest przeznaczonych do szyn adresowych i danych. Warto zauważyć, że NES nie zapewnia kartridża z pełnymi 16-bitowymi adresami CPU lub PPU. Ma to sens w przypadku pamięci programu ROM, ponieważ zajmuje ona tylko górną połowę pamięci procesora, a zatem wymaga tylko 15 bitów do pełnego zaadresowania.

Jeśli chodzi o pamięć znaków ROM, jest to nieco bardziej skomplikowane. Zgodnie z tym jak konsola była zaprojektowana, NES oczekuje, że pamięć ROM lub RAM znaków będzie mieścić maksymalnie 8 KB w dowolnym momencie – wystarczająca ilość danych do przechowywania dwóch pełnych stron z 256 kafelków 8×8 pikseli. Widzimy, że 8 KB pamięci można zaadresować przy użyciu tylko 13 bitów, ale zamiast 13 bitów, NES dostarcza 14-bitowy adres PPU do kartridża. Tak jest, ponieważ gry mogą czasami przechowywać własną pamięć RAM (*konkretnie VRAM*), która zawiera dodatkowe 8 KB przestrzeni adresowej, więc potrzebuje dodatkowego bitu.

Pozostałe piny służą do przekazywania sygnałów między kartridżem a NES. Na przykład:

ROM_SEL – gdy pin wyboru pamięci ROM jest ustawiony na stan niski, informuje to kartridż, że NES próbuje odczytać bajt informacji z pamięci programu ROM i kartridż powinien włączyć ten układ na wyjście.

IRQ – ten pin wysyła sygnał z kartridża do NES. IRQ oznacza żądanie przerwania w CPU. Kiedy procesor wykryje niskie napięcie na tym pinie, zatrzyma wykonywanie i przeskoczy do innej części kodu, aby wykonać tak zwaną procedurę obsługi przerwań. Niektóre układy mapujące pamięć, takie jak MMC3, mogą wykorzystać IRQ do wysyłania sygnału przerwania po wyrenderowaniu przez PPU określonej liczby linii skanowania. Pozwala to na zaawansowane techniki graficzne, takie jak zmiana palet lub innych danych związanych z PPU, w połowie klatki.

2.3.1 Mapper'y pamięci

Układy mapujące mają sporo zaawansowanych funkcji, jednak ich głównym celem jest wybieranie banków w układach ROM i RAM kartridża. Dzięki temu gry mogą przekroczyć ograniczenia 16-bitowej przestrzeni adresowej narzuconą przez architekturę systemu.

Gry często zawierają więcej kodu programu i danych graficznych, niż system może obsłużyć w danym momencie. Aby obejść ten problem, układ ROM i RAM można podzielić na szereg banków, które następnie można wymieniać w zależności od potrzeb programu gry. Specyfika może zmieniać się od mapper'a do mapper'a, ale generalnie sposób, w jaki program to robi, polega na zapisywaniu danych w przestrzeni programu ROM. Kiedy mapper widzi dane zapisywane pod określonym adresem programu ROM, zinterpretuje te dane jako komunikat kontrolny i zmieni swoje zachowanie, zmieniając banki ROM lub wykonując jakąś specjalną funkcję.

Kartridże często zawierają układy pamięci RAM ogólnego przeznaczenia, które pozwalają im zwiększyć pamięć RAM zapewnianą przez system. Układy mapujące generalnie są odpowiedzialne za koordynację między układami RAM i ROM na podstawie adresu podanego przez procesor. Najczęściej te układy będą zajmować mapę pamięci, zaczynając od adresu 0x6000 i mieć rozmiar od 2 do 8 KB, ale ogólnie rozmiar i lokalizacja pamięci może się znacznie różnić w zależności od mapper'a. [29]

2.3.2 Format pliku iNES

Format iNES to format pliku używany do przechowywania obrazów ROM gier i programów NES. Został wprowadzony w roku 1996 przez Marata Fayzullina i stał się standardem dla ROM'ów NES. [31]

Struktura pliku iNES składa się z 16-bajtowego nagłówka, po którym jest faktyczna zawartość ROM'u. Nagłówek zawiera informacje o „mapperze”, rozmiarze PRG-ROM, rozmiarze CHR-ROM, „mirror'u”, pamięci RAM podtrzymywanej przez baterie i innych szczegółach. Po nagłówku następują dane PRG i CHR, które zawierają kod i grafikę gry. [29]

Inne formaty NES ROM to UNIF (*Universal NES Image Format*), FDS (*Famicom Disk System*), NES 2.0 i NSF (*NES Sound Format*). UNIF jest podobny do iNES, ale obsługuje więcej mapper'ów i ma inną strukturę nagłówka. FDS jest używany w grach wydanych na

Famicom Disk System, który został wydany tylko w Japonii. NES 2.0 to zaktualizowana wersja iNES, która dodaje obsługę bardziej zaawansowanych mapperów i innych funkcji. NSF to format używany do plików muzycznych NES, które można odtwarzać na NES lub emulatorze.

2.4 Kontroler

Kontroler NES to urządzenie używane do grania w gry na konsoli. Ma osiem przycisków, które zgłaszają się jako bity na linii danych: *A*, *B*, *Select*, *Start*, *Góra*, *Dół*, *Lewo* i *Prawo*. Jednostka NES jest dostarczana z co najmniej jednym standardowym kontrolerem, którego można używać w obu portach kontrolera. Rejestr przesuwny kontrolera musi być ustawiony, aby ustawić stan przycisków, można to zrobić poprzez wysłanie bitu informacji na adres 0x4016. Następnie procesor odczytuje po kolejny bit, łącznie osiem razy przez adres 0x4017.

4021 IC jest używany jako 8-bitowy równoległy-szeregowy rejestr przesuwny (*Parallel In Serial Out, PISO*) w kontrolerze. [30]

3. Dostępne emulatory konsoli NES

Mesen 0.9.9

Mesen to bardzo dokładny emulator NES dla systemów Windows i Linux. Oferuje wiele funkcji, takich jak stany zapisu, filtry wideo, grę sieciową, przewijanie emulacji do tyłu, przetaktowywanie (ang. overclocking), kody (cheat codes), możliwość podmiany grafiki gry alternatywnymi rozwiązaniami w wysokiej rozdzielczości. [32]

Zagłębiając się bardziej, emulator charakteryzuje się kilkoma kluczowymi cechami:

Dokładność: Mesen dąży do zapewnienia dokładnej emulacji konsol NES i Famicom, co oznacza, że stara się jak najbardziej naśladować ich zachowanie. Obejmuje to emulację taktowania sprzętu, wyjścia audio i wideo, a także inne mniej znane przypadki i ograniczenia.

Kompatybilność: obsługuje szeroką gamę gier i urządzeń periferyjnych NES i Famicom, w tym popularne tytuły, takie jak Super Mario Bros., The Legend of Zelda i Mega Man. Obsługuje również wszystkie mapper'y licencjonowanych gier i całkiem sporą część tych nielicencjonowanych.

Narzędzia do debugowania: zawiera szereg narzędzi do debugowania, które mogą pomóc przy zidentyfikowaniu i naprawieniu problemów w kodzie emulacji lub samego kodu gry. Narzędzia te obejmują profiler kodu, podgląd pamięci i podgląd danych PPU.

Wydajność: został zaprojektowany tak, aby był szybki i wydajny, co oznacza, że może działać na szerokiej gamie sprzętu i systemów operacyjnych.

Open source: emulator jest open source, co oznacza, że można uzyskać dostęp do jego kodu źródłowego i zmodyfikować go do własnych potrzeb. Może to być szczególnie przydatne, jeśli tworzymy własny emulator i chcemy uczyć się z kodu tego emulatora lub włączyć niektóre jego funkcje do własnego projektu.

MetalNES

Na podstawie źródeł [33] i [34], MetalNES jest bardzo unikalnym emulatorem, który próbuje symulować zachowanie NES na poziomie tranzystorowym, co jest poziomem dokładności wykraczającym wykraczającym poza to, co większość emulatorów próbuje osiągnąć. Ten poziom dokładności jest zwykle osiągane dzięki zastosowaniu emulacji opartej na sprzecie za pomocą FPGA, a nie emulacji oprogramowania.

Należy zauważyć, że MetalNES jest obecnie na wczesnym etapie rozwoju i nie jest jeszcze zoptymalizowany pod kątem wydajności. W rezultacie jest obecnie bardzo powolny i nie nadaje się do grania w większości gier NES, ponieważ nie ma obsługi większości Mapper'ów, których wymaga większość gier NES. Obecnie kompiluje się jedynie na systemach macOS.

Chociaż MetalNES nie jest obecnie praktyczny dla większości użytkowników, może być cennym narzędziem dla programistów emulatorów i hobbystów, którzy są zainteresowani badaniem wewnętrznego działania NES na bardzo niskim poziomie. Fakt, że MetalNES jest open source, oznacza, że może to być również cennym źródłem zasobem dla programistów, którzy chcą uczyć się kodu i potencjalnie włączyć niektóre z jego funkcji do własnych projektów.

4. Narzędzia i metodologia

Chociaż dostępnych jest wiele dobrze znanych emulatorów NES, ten emulator różni się od nich tym, że koncentruje się na celach edukacyjnych. Kod emulatora ma strukturę przypominającą architekturę konsoli NES, zapewniając głębsze zrozumienie podstawowych komponentów i procesów. To podejście edukacyjne stawia na prostotę i przejrzystość, dzięki czemu jest doskonałym narzędziem do nauki dla początkujących twórców gier i twórców emulatorów.

W porównaniu z innymi emulatorami ten emulator może nie mieć niektórych zaawansowanych funkcji i złożoności, ale zapewnia podstawy zrozumienia, które są niezbędne do dalszego rozwoju. Koncentrując się na podstawowych komponentach konsoli NES, ten emulator stanowi skuteczny punkt wejścia dla tych, którzy chcą zgłębić się w świat emulacji NES.

4.1 Język programowania

Wspaniałą rzeczą w używaniu C# jest to, że jest to wieloplatformowy język programowania, co oznacza, że sama część emulacji może działać na wielu systemach operacyjnych. W przeciwieństwie do C++, C# jest uważany za język programowania zorientowany na komponenty, co umożliwia jasne przedstawienie fizycznej architektury konsoli.

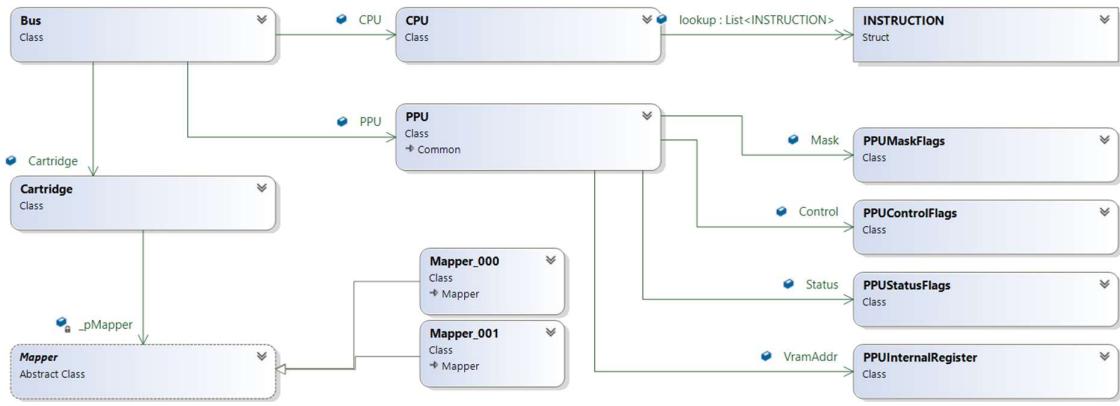
4.2 Grafi ka i GUI

OpenGL został użyty jako podstawowe API graficzne, była to jedyna dostępna wówczas opcja, ponieważ DirectX jest powiązany wyłącznie z systemem Windows, a OpenGL tak jak C# jest wieloplatformowy.

Jeśli chcemy zastąpić GUI utworzone przy użyciu Windows Forms i WinAPI innym GUI, które będzie kompatybilne dla innego systemu operacyjnego, bez konieczności wprowadzania ogromnych modyfikacji w podstawowej emulacji. To sprawia, że emulator jest bardzo wszechstronny i można go dostosować do różnych środowisk, co jest ogromną zaletą. To oddzielenie części emulacji od GUI ułatwia również konserwację i ulepszenie emulatora w przyszłości, ponieważ zmiany można wprowadzać w jednym bez wpływu na drugi.

4.3 Architektura

Diagram klas na *rysunku 5* reprezentuje uproszczoną architekturę konsoli NES. Program emuluje zachowanie NES, używając klas do symulacji jego różnych komponentów.



Rysunek 5. Diagram klas emulatora.

Klasa `Bus` reprezentuje główną magistralę danych w systemie NES, która służy do połączenia wszystkich komponentów systemu. Zawiera odniesienia do trzech innych klas: `CPU`, `PPU` i `Cartridge`.

Klasa `CPU` reprezentuje jednostkę centralną NES, która wykonuje instrukcje dla gry. Zawiera strukturę `INSTRUCTION`, która definiuje zestaw instrukcji dla procesora.

Klasa `PPU` reprezentuje jednostkę przetwarzania obrazu NES, która generuje grafikę dla gry. Zawiera kilka klas reprezentujących wewnętrzne rejestrów i flagi kontrolujące różne aspekty generowania grafiki.

Klasa `Cartridge` reprezentuje kartridż z grą, który jest wkładany do konsoli NES. Zawiera odniesienie do klasy `Mapper`, która jest odpowiedzialna za mapowanie danych gry do pamięci NES.

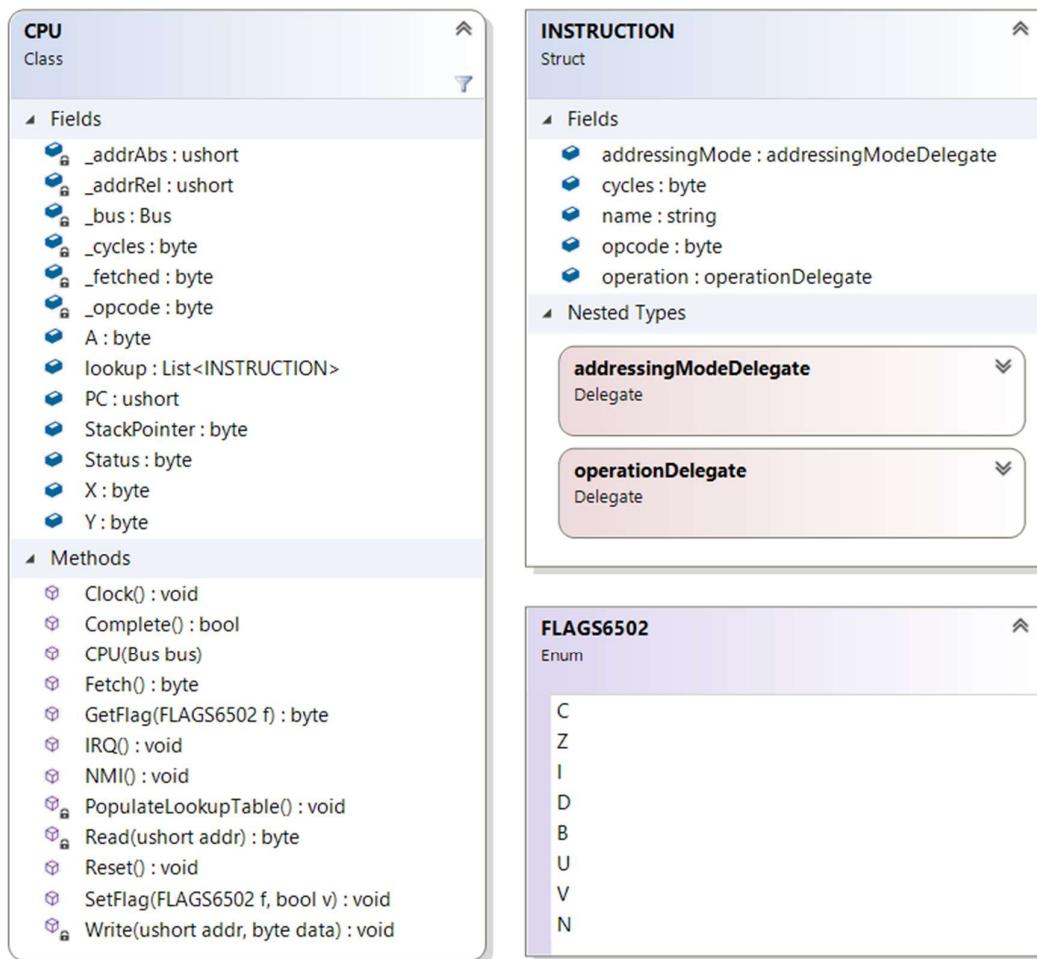
Klasa `Mapper` to klasa abstrakcyjna, która definiuje interfejs dla różnych klas mapper'ów, które mapują dane gry na pamięć NES. Klasy `Mapper_000` i `Mapper_001` to specyficzne implementacje mapper'ów, które są używane do mapowania danych gry w pamięci NES.

Dostarczone klasy przypominają architekturę konsoli, a każda klasa reprezentuje określony komponent systemu. Klasa `Bus` służy jako główny hub do łączenia ze sobą wszystkich komponentów, podczas gdy klasy `CPU` i `PPU` reprezentują dwie główne jednostki przetwarzające konsoli. Klasy `Cartridge` i `Mapper` reprezentują odpowiednio kartridż z grą i mapper, który mapuje dane gry na pamięć NES.

5. Implementacja

5.1 Emulacja CPU

Kod odpowiedzialny za emulacje procesora znajduje się w klasie `CPU` i jest podzielony na trzy klasy częściowe, oddzielając metody odpowiedzialne za tryby adresowania i operacje do osobnych plików. `AddressingModes` zawiera takie metody jak *IMP()*, *IMM()*, itd., a `Operations` zawiera funkcje takie jak *ADC()*, *SBC()*, *AND()*, itd. (na diagramie ukryto metody trybów adresowania i operacji).



Rysunek 6. Szczegółowy diagram klasy CPU i jego struktur.

W klasie `CPU` znajduje się lista przeszukiwania `lookup`, inaczej zwana „*lookup table*”, zawierająca wszystkie operacje, które może wykonywać procesor MOS 6502. Każda nowa operacja do wykonania jest wyszukiwana na tej liście. Wywołanie metody *Cycle()* reprezentuje wykonanie jednego cyklu procesora. Zmienna `cycle` służy do zliczania liczby cykli pozostałych w bieżącej instrukcji. Funkcja *Reset()* do resetowania CPU do stanu domyślnego. *Fetch()* służy do pobierania wartości operandu z pamięci.

Zostały również utworzone funkcje do ustawiania i pobierania wartość flag rejestru stanu procesora. Funkcja `IRQ` jest używana do przerwania działania procesora, a funkcja `NMI` jest używana do przerwania niemaskownego.

Jedną z najważniejszych części tego kodu jest funkcja *Clock()*.

```
public void Clock()
{
    if (_cycles == 0)
    {
        _opcode = Read(PC);

        PC++;

        _cycles = lookup[_opcode].cycles;

        Byte additional_cycle1 = lookup[_opcode].addressingMode();
        Byte additional_cycle2 = lookup[_opcode].operation();

        _cycles += (Byte)(additional_cycle1 & additional_cycle2);
    }
    _cycles--;
}
```

Rysunek 7. Kod źródłowy metody *Clock()* w klasie CPU.

Ta funkcja odpowiada za wykonywanie instrukcji w procesorze emulatora NES. Najpierw sprawdza, czy liczba cykli dla bieżącej instrukcji wynosi zero. Jeśli tak, odczytuje opcode następnej instrukcji z pamięci na adresie wskazanym przez rejestr PC (*licznik programu*). Po czym zwiększa PC o jeden i pobiera liczbę cykli wymaganych dla instrukcji z tabeli `lookup`. Wywoływane są funkcje *addressingMode()* i *operation()* w celu określenia adresu pamięci i operacji do wykonania. Te funkcje mogą zwracać dodatkowe cykle, które należy dodać do całkowitej liczby cykli. Na koniec każdego cyklu zmniejszana jest wartość `_cycle`.

Wykorzystanie tablicy przeglądowej do określenia trybu adresowania i operacji do wykonania jest skutecznym sposobem obsługi wielu różnych instrukcji, które może wykonywać jednostka centralna.

Ta tabela jest listą składającą się ze struktur `INSTRUCTION`, które zawierają informacje o instrukcji procesora, a mianowicie nazwę, kod operacji (*opcode*), operację, tryb adresowania i liczbę cykli.

```

struct INSTRUCTION
{
    public string name;
    public Byte opcode;
    public delegate Byte operationDelegate();
    public operationDelegate operation;
    public delegate Byte addressingModeDelegate();
    public addressingModeDelegate addressingMode;
    public Byte cycles;
}

```

Rysunek 8. Kod źródłowy struktury pojedynczej instrukcji w klasie CPU.

Pole `name` zawiera nazwę instrukcji i jest przydatne przy debugowaniu procesora. Pole `opcode` to bajt zawierający binarny kod operacji dla instrukcji. Pole `operation` jest wskaźnikiem funkcji, który wskazuje na implementację instrukcji. Pole `addressingMode` jest również wskaźnikiem funkcji, ale tym razem wskazującym na implementację trybu adresowania używanego przez instrukcję. Pole `cycles` przechowuje liczbę cykli wymaganych do wykonania instrukcji na procesorze 6502.

Struktura `INSTRUCTION` jest używana w tabeli `lookup` do odwzorowania kodu operacji instrukcji na odpowiednią implementację. Pozwala to procesorowi na szybkie wykonywanie instrukcji bez konieczności wykonywania dodatkowego dekodowania. Struktura zapewnia wygodny sposób hermetyzacji informacji o każdej instrukcji, dzięki czemu kod jest bardziej czytelny i łatwiejszy w utrzymaniu.

Operacje dodawania i odejmowania w 6502 są jednymi z najbardziej użytecznych, ale także problematycznych. Instrukcja dodawania dodaje dane z akumulatora i pbranej pamięci, łącznie z bitem przeniesienia (C), umożliwiając sekwencyjne dodawanie 8-bitowych słów. Problem pojawia się przy używaniu liczb ze znakiem, gdzie przekroczenie zakresu roboczego może spowodować przepełnienie, powodując utratę znaczenia tych liczb. Jednak dwie flagi w 6502 mogą pomóc to wykryć: flaga ujemna (N), która jest ustawiona, gdy wynik operacji ma ustawiony najbardziej znaczący bit, oraz flaga przepełnienia (V).

Tabela 3 określa, kiedy należy ustawić bit przepełnienia.

Tabela 4. Tabela prawdy określająca bit przepełnienia.

A	M	R	V	A^R	$\sim(A^M)$
0	0	0	0	0	1
0	0	1	1	1	1
0	1	0	0	0	0
0	1	1	0	1	0
1	0	0	0	1	0
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	0	1

Aby określić bit przepelnienia w operacji dodawania. Uwzględniamy wartość w akumulatorze (A), dodawane dane (M) oraz wynik (R). Analizując najbardziej znaczący bit każdego bajta i tworząc nową kolumnę dla bitu przepelnienia (V), określamy, kiedy należy go ustawić. Bit przepelnienia jest ustawiany tylko w dwóch sytuacjach, więc wprowadzamy formułę logiczną, która daje te wyniki. Bit przepelnienia jest ustawiany, gdy $(A \wedge R) \vee (\neg(A \wedge M))$ jest równe 1.

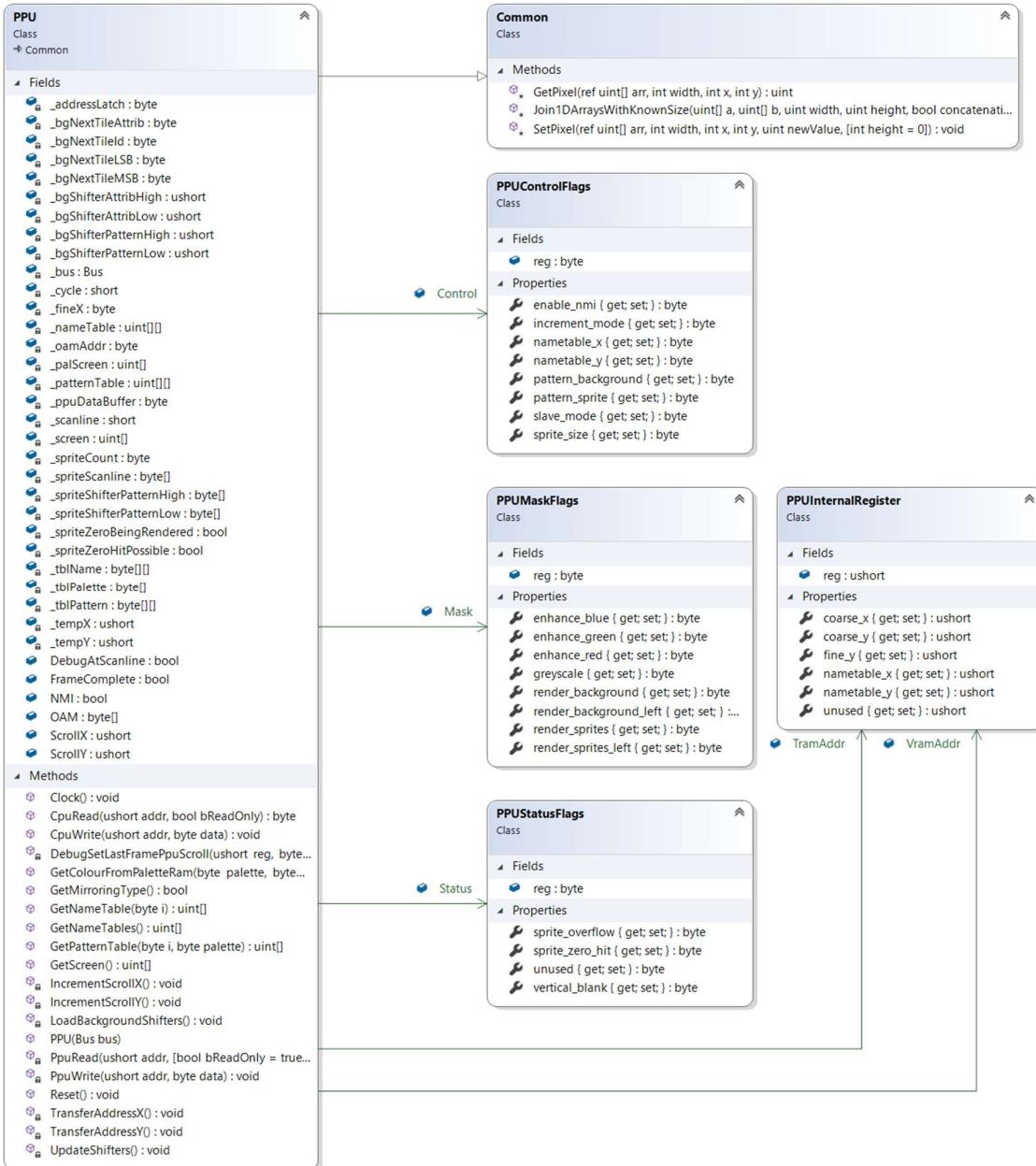
```
public Byte ADC()
{
    Fetch();
    _temp = (UInt16)((UInt16)_temp + (UInt16)_fetched + (UInt16)GetFlag(FLAGS6502.C));
    SetFlag(FLAGS6502.C, (_temp > 255));
    SetFlag(FLAGS6502.Z, (_temp & 0x00FF) == 0);
    SetFlag(FLAGS6502.V, ((~(UInt16)_temp) & (UInt16)_fetched) & ((UInt16)_temp & 0x0080) != 0);
    SetFlag(FLAGS6502.N, (_temp & 0x80) != 0);
    A = (Byte)(_temp & 0x00FF);
    return 1;
}
```

Rysunek 9. Kod źródłowy metody `ADC()` w klasie `CPU`, implementującej instrukcję `ADC`.

W kodzie instrukcja `ADC` najpierw pobiera dane, a następnie wykonywane jest dodawanie. Aby wykonać dodawanie w domenie 16-bitowej, wszystkie zmienne są rzutowane na 16 bitów. Pozwala to na łatwe sprawdzenie, czy potrzebny bit przeniesienia, co byłoby wskazane przez ustawiony bit w starszym bajcie 16-bitowego słowa. Flaga przeniesienia jest następnie odpowiednio ustawiana. Flaga zera jest ustawiana, gdyby wynik jest równy zero (`_temp` jest 16-bitowy, dlatego maskujemy młodszy bajt). Aby obliczyć flagę przepelnienia, używamy wcześniej wspomnianej formuły logicznej $(A \wedge R) \vee (\neg(A \wedge M))$ i patrzymy na najbardziej znaczący bit młodszego bajta. Flaga ujemna jest ustawiana na podstawie najbardziej znaczącego bitu wyniku. Wynik następnie jest zapisywany w akumulatorze. Warto zauważyć, że instrukcja `ADC` wymaga dodatkowego cyklu do wykonania, dlatego zwracamy 1 na końcu.

5.2 Emulacja PPU

Kod reprezentujący PPU składa się łącznie z sześciu klas: `PPU`, `Common` (która zawiera funkcje, takie jak `SetPixel`, `GetPixel`), `PPUControlFlags`, `PPUMaskFlags`, `PPUStatusFlags` i `PPUInternalRegister`.

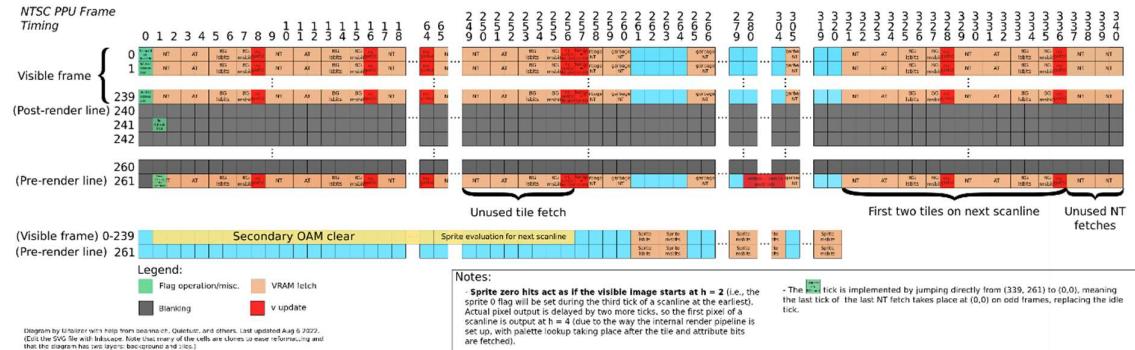


Rysunek 10. Szczegółowy diagram klasy PPU i innych klas związanych z nią.

Klasa `PPU` jest podstawową klasą emulacji jednostki przetwarzania obrazu i zawiera różne pola i metody służące emulacji 2C02. Ma pola takie jak `FrameComplete` i `DebugAtScanline`, które wskazują, czy ramka została ukończona i czy debugger powinien zatrzymać się na określonej linii skanowania. Posiada różne rejestrze do zarządzania wewnętrznym stanem PPU, takie jak `VramAddr` i `TramAddr` do przechowywania odpowiednio aktywny adres „wskaźnika” do tablicy nazw, aby wyodrębnić informacje o kafelku tła i tymczasowych adresu (*które mają być „przenoszone” do „wskaźnika” w różnych momentach*), a także `Control`, `Mask` i `Status` do zarządzania różnymi flagami związanymi z PPU.

Klasa `PPU` ma również tablicę OAM do przechowywania informacji o 64 sprite'ach w konsoli NES, a także inne pola do zarządzania stanem wewnętrznym, takie jak `NMI`, `cycle` i `scanline` do śledzenia bieżącej linii skanowania (*linia wybierania / scanline*) i cyklu.

Tablice do przechowywania informacji o renderowaniu tła i pierwszego planu NES, są w `tblName` do przechowywania tablicy nazw pamięci VRAM, `tblPalette` do przechowywania danych palet kolorów pamięci SRAM. A informacje o wszystkich dostępnych kolorach NES są ustawiane przy inicjalizacji klasy `PPU` w `palScreen`.



Rysunek 11. Schemat synchronizacji NTSC układu 2C02 (PPU). (źródło: [21])

Aby dokładnie emulować sygnał wideo NTSC, metoda *Clock()*, która odpowiada za przesunięcie PPU o jeden cykl zegara, musi być zsynchronizowana z taktowaniem sygnału wideo pokazanym na *rysunku 11*. Ta metoda obsługuje stan PPU, który polega na pobieraniu informacji o tle i sprite'ach oraz składaniu ich w piksele, które są na końcu wyświetlane.

```
public void Clock()
{
    if (_scanline >= -1 && _scanline < 240)
    {
        // ----- BACKGROUND RENDERING -----
        if (_scanline == 0 && _cycle == 0)...
        if (_scanline == -1 && _cycle == 1)...
        if ((_cycle >= 2 && _cycle < 258) || (_cycle >= 321 && _cycle < 338))...
        if (_cycle == 256)
        {
            IncrementScrollY();
        }
        if (_cycle == 257)
        {
            LoadBackgroundShifters();
            TransferAddressX();
        }
        if (_cycle == 338 || _cycle == 340)
        {
            _bgNextTileId = PpuRead((UInt16)(0x2000 | (VramAddr.reg & 0xFFFF)));
        }
        if (_scanline == -1 && _cycle >= 280 && _cycle < 305)
        {
            TransferAddressY();
        }
        // ----- FOREGROUND RENDERING -----
        if (_cycle == 257 && _scanline >= 0)...
        if (_cycle == 340)...
    }
    if (_scanline == 240)...
    if (_scanline >= 241 && _scanline < 261)...
    // --- Background
    Byte bg_pixel = 0x00; // bg Pixel
    Byte bg_palette = 0x00; // bg Palette
    if (Mask.render_background != 0)...
    // --- Foreground
    Byte fg_pixel = 0x00;
    Byte fg_palette = 0x00;
    Byte fg_priority = 0x00;
    if (Mask.render_sprites != 0)...
    Byte pixel = 0x00; // FINAL Pixel
    Byte palette = 0x00; // FINAL Palette
    if (bg_pixel == 0 && fg_pixel == 0)... else if (bg_pixel == 0 && fg_pixel > 0)... else if (bg_pixel > 0 && fg_pixel == 0)... else if (bg_pixel > 0 && fg_pixel > 0)...
    SetPixel(ref _screen, 256, (Int16)(_cycle - 0x0001), _scanline, GetColourFromPaletteRam(palette, pixel), 240);
    _cycle++;
    if (_cycle >= 341)...
}
```

Rysunek 12. Kod źródłowy metody *Clock()* w klasie PPU.

Metoda rozpoczyna od sprawdzenia, czy bieżąca linia skanowania jest widoczna, czy nie (*tj. między liniami skanowania 0 a 239*). Jeśli tak, kontynuuje renderowanie tła i pierwszego planu dla bieżącej linii skanowania.

Widoczny obszar ekranu jest renderowany przez sekwencję powtarzających się zdarzeń, które występują co dwa cykle zegara. Zdarzenia te obejmują:

- ładowanie bieżącego wzoru kafelka tła i atrybutów do rejestru przesuwnego,
- pobranie następnego identyfikatora kafelka tła,
- odczytanie poprawnego bajta atrybutu dla określonego adresu,
- pobranie następnego bitu (*LSB*) kafelka tła,
- i następnego bitu (*MSB*) kafelka tła

Na koniec zwiększa wskaźnik kafelka tła do następnego kafelka poziomo w pamięci tabeli nazw.

Po wyrenderowaniu widocznego obszaru ekranu zwiększcza jest bieżąca linia skanowania w dół, a następnie powtarza tę samą sekwencję zdarzeń dla następnej linii skanowania, aż do osiągnięcia końca ramki.

Następnie rozpoczyna się proces renderowania pikseli i rysowania wynikowego obrazu.

W przypadku piksela tła sprawdza, czy renderowanie tła jest włączone. Jeśli jest włączone, to wyodrębnia odpowiednie informacje o pikselach i paletach z rejestrów przesuwnych `bgShifter`, `bgShifterAttrib` i odpowiednio ustawia zmienne `bg_pixel` i `bg_palette`. Jeśli renderowanie w tle jest wyłączone, zmienne `bg_pixel` i `bg_palette` są ustawione na 0.

Następnie renderuje piksel pierwszego planu. Sprawdza, czy renderowanie sprite'ów jest włączone, i jeśli jest, to sprawdza, czy bieżący piksel jest zakryty sprite'm. Jeśli sprite zakrywa piksel, zmienne `fg_pixel` i `fg_palette` są ustawiane na odpowiednie wartości, a zmienne `fg_priority` jest ustawiane w celu określenia, czy sprite powinien być rysowany przed, czy za tłem.

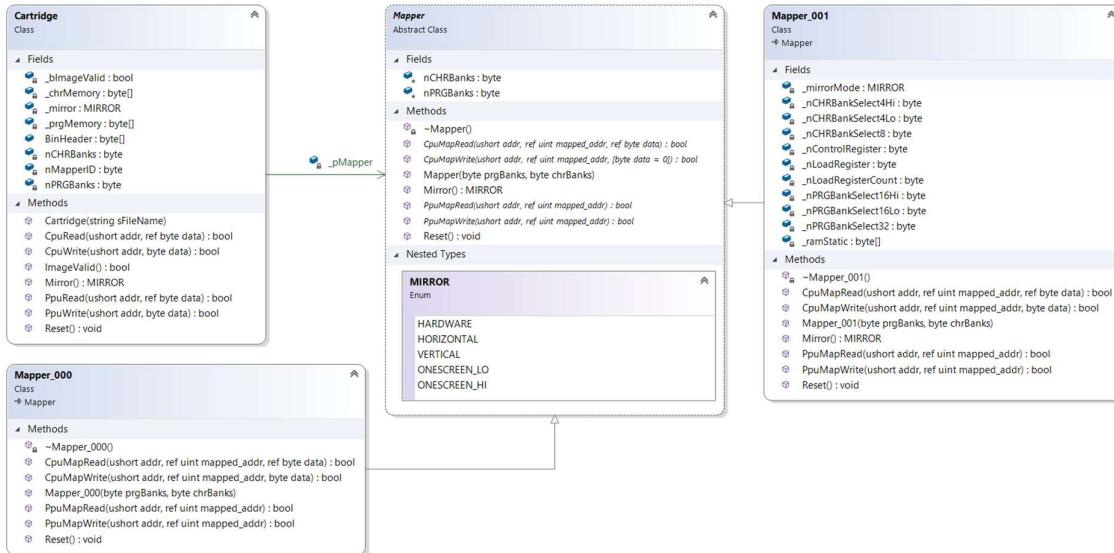
Jeśli żaden sprite nie zakrywa piksela lub renderowanie sprite'ów jest wyłączone, zmienne `fg_pixel` i `fg_palette` są ustawione na 0.

Ostatnim krokiem jest określenie koloru, który ma być użyty dla bieżącego piksela. Jeśli piksel tła jest rysowany przed sprite'em lub jeśli nie ma sprite'u, kolor jest określany zmienną `bg_palette`. Jeśli sprite jest rysowany przed pikselem tła, kolor jest określany przez zmienną `fg_palette`.

Zmienna piksel jest następnie ustawiana na odpowiedni kolor na podstawie priorytet tła i sprite'a, a tablica `screen` jest aktualizowana o nowy piksel za pomocą metody *SetPixel*.

Na koniec metoda przechodzi do następnego cyklu i linii skanowania. Jeśli bieżąca linia skanowania jest ostatnią w ramce, pole `FrameComplete` przyjmuje wartość `true`, co oznacza, że narysowano pełną klatkę.

5.3 Import i obsługa kartridżów w formacie iNES



Rysunek 13. Szczegółowy diagram klasy Cartridge i innych klas związanych z nią

Klasa `Cartridge` reprezentuje kartridż z grą. Klasa zawiera informacje o grze jak i o samym kartridżu, takie jak liczba banków pamięci Program ROM, Character ROM, identyfikator mapper'a i tryb mirror. Klasa posiada również konstruktor, który wczytuje pliki gry i ładuje go do pamięci.

Klasa `Mapper` jest klasą abstrakcyjną używaną jako klasa bazowa dla innych klas mapujących. Powodem używania klasy abstrakcyjnej jest dostarczenie szablonu dla innych klas, które z niej dziedziczą. Klasa abstrakcyjna może zawierać metody i pola, które muszą implementować klasy dziedziczące. Jest to przydatne, ponieważ umożliwia zdefiniowanie wspólnego interfejsu dla grupy powiązanych klas.

Klasy mapper `Mapper_000` i `Mapper_001` zapewniają różne funkcje mapowania dla różnych kartridżów. Wybór tych klas jest oparty na identyfikatorze mapper'a odczytanym z pliku gry. Odpowiedni mapper jest ładowany na podstawie identyfikatora mapper'a. Dzięki temu emulator może obsługiwać różne typy gier, które wykorzystują różne techniki mapowania.

```

public Cartridge(String sFileName)
{
    _bImageValid = false;

    FileStream fs = new FileStream(sFileName, FileMode.Open);

    fs.Read(name, 0, 4);
    if (name[0] != 'N' && name[1] != 'E' && name[2] != 'S' && name[3] != 0x1A)
        throw new Exception("Nieznany format pliku.");

    fs.Read(prg_rom_chunks, 0, 1); fs.Read(chr_rom_chunks, 0, 1); fs.Read(flag_6, 0, 1); fs.Read(flag_7, 0, 1);
    if((flag_7[0] & 0x0C) == 0x08)
        throw new Exception("Nieobsługiwany format pliku. Wykryto format NES 2.0");

    fs.Read(prg_ram_size, 0, 1); fs.Read(tv_system1, 0, 1); fs.Read(tv_system2, 0, 1); fs.Read(unused, 0, 5);

    // Jeżeli "trainer" instnieje pomijamy go
    UInt16 skip = 0;
    if ((flag_6[0] & 0x04) == 0x04){...}

    // Ustalenie ID Mapper'a
    nMapperID = (Byte)((flag_7[0] >> 4) << 4) | (flag_6[0] >> 4));
    _mirror = ((flag_6[0] & 0x01) == 0x01) ? MIRROR.VERTICAL : MIRROR.HORIZONTAL;

    // File Format
    Byte nFileType = 1;
    if ((flag_7[0] & 0x0C) == 0x08) nFileType = 2;

    // iNES Format
    if (nFileType == 1){...}

    // NES 2.0 format
    if (nFileType == 2)
    {
    }

    // Załadowanie odpowiedniego Mapper'a
    switch (nMapperID){...}

    _bImageValid = true;
    fs.Close();
}

```

Rysunek 14. Kod źródłowy konstruktora klasy Cartridge.

Klasa `Cartridge` w emulatorze NES obsługuje importowanie i obsługę plików iNES. Kiedy konstruktor jest wywoływany, pobiera ścieżkę pliku ze zmiennej `sFileName`, która zawiera plik o formacie iNES do zainportowania.

Następnie kod odczytuje nagłówek pliku iNES w celu określenia formatu pliku i wyodrębnia odpowiednie informacje, takie jak liczba banków PRG i CHR oraz identyfikator mapper'a. Jeśli plik jest w formacie iNES, banki są ładowane do pamięci. Jeśli plik jest w formacie NES 2.0, klasa `Cartridge` go nie obsługuje w obecnej implementacji.

Po załadowaniu gry do pamięci odpowiedni Mapper jest wybierany na podstawie identyfikatora mapper'a wyodrębnionego z nagłówka iNES. Mapper jest wówczas odpowiedzialny za obsługę mapowania pamięci między procesorem/PPU a kartą grą.

```

public bool CpuRead(UInt16 addr, ref Byte data)
{
    UInt32 mapped_addr = 0;
    if (_pMapper.CpuMapRead(addr, ref mapped_addr, ref data))
    {
        if (mapped_addr == 0xFFFFFFFF)
        {
            return true;
        }
        else
        {
            // Mapper zwrócił offset do banku pamięci w kartridżu
            data = _prgMemory[(Int32)mapped_addr];
        }
        return true;
    }
    else
    {
        return false;
    }
}

```

Rysunek 15. Kod źródłowy metody *CpuRead* w klasie *Cartridge*.

Klasa `Cartridge` zapewnia również metodę *CpuRead*, która umożliwia procesorowi odczytywanie danych z kartridża. Ta metoda mapuje adres z procesora na adres pamięci w kartridżu przy pomocy wybranego Mapper'a, a następnie odczytuje dane z tego adresu pamięci. Podobna komunikacja występuje pomiędzy PPU, przy użyciu metod *PpuRead/PpuWrite*.

6. Prezentacja działania aplikacji

Na *rysunku 16* przedstawiono emulator, na którym jest uruchomione demo programu napisanego dla konsoli NES. Demo jest dołączone do kodu źródłowego i znajduje się również w skompilowanej wersji emulatora.



Rysunek 16. Zrzut ekranu przedstawiający uruchomiony emulator z ROM'em "demo.nes".

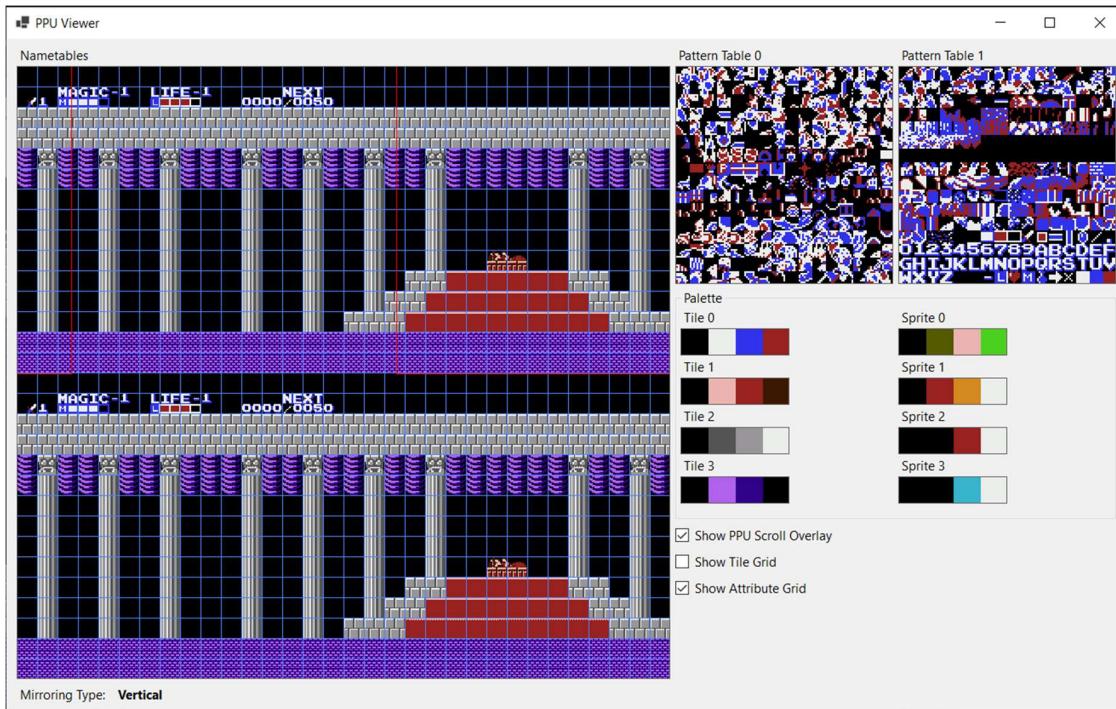
6.1 PPU Viewer

Jest widok danych PPU w czasie rzeczywistym. Zapewnia to wizualną reprezentację zawartości tabel nazw, tabel wzorów i palet kolorów, umożliwiając zobaczenie i zrozumienie wewnętrznego działania emulatora.

Po lewej widzimy wizualną reprezentację tabele nazw, które fizycznie znajdujących się w pamięci RAM należącej do PPU. Pod tabelami nazw mamy informacje o rodzaju mirror'u (kierunku odbijania/zawijania się tabel nazw w pamięci).

W prawym górnym rogu widzimy wizualną reprezentację dwóch tabel wzorów, które fizycznie znajdują się na kartridżu, a konkretnie na układzie zwanym Character ROM.

Pod tabelami wzorów mamy informacje, o paletach które pobierane są z pamięci SRAM znajdującym się wewnętrz układow PPU.



Rysunek 17. Zrzut ekranu przedstawiający okno emulatora dedykowane do podglądu danych PPU.

6.2 Weryfikacja działania

Aby upewnić się, że emulator działa poprawnie. Należy przetestować emulator przy użyciu różnych metod, jak uruchomienie ROM'ów testowych, czy sprawdzenia gier, z którymi emulator powinien być kompatybilny.

6.2.1 Gry

Emulator został przetestowany z różnymi grami w systemie NTSC, wydanych na kartridżach z układami mapowania pamięci znymi jako *Mapper 0* i *Mapper 1*.

W tabeli 4 jest przedstawiona lista gier, które zostały przetestowane:

Tabela 5. Lista przetestowanych gier.

Tytuł gry	System	Mapper
Super Mario Bros.	NTSC	0
Donkey Kong	NTSC	0
Bomberman	NTSC	0
Excitebike	NTSC	0
Ice Climber	NTSC	0
Ice Hockey	NTSC	0
Zelda II: The Adventure of Link	NTSC	1
Bionic Commando	NTSC	1

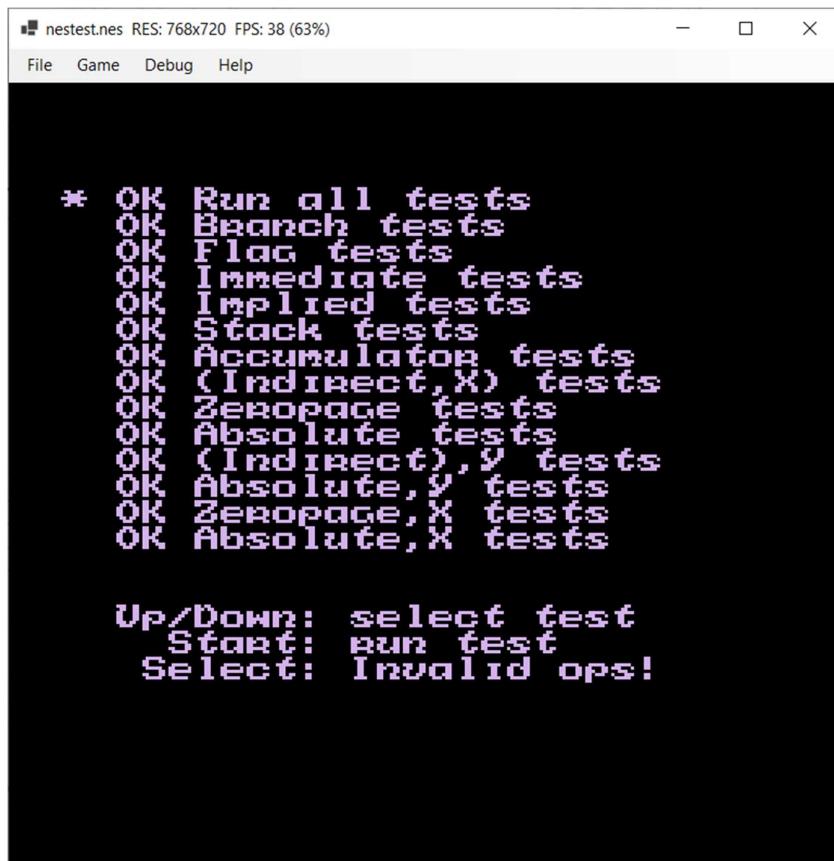
Wszystkie gry uruchamiają się bez żadnych problemów. Nie napotkano żadnych błędów wizualnych, a same gry działają tak jak na oryginalnej konsoli. Do weryfikacji tego gry zostały uruchomione także na emulatorze *Mesen*, w celu porównania.

Pełną listę gier, które powinny być obsługiwane przez ten emulator można znaleźć pod tym adresem [35].

6.2.2 ROM'y testowe

nestest – autorstwa *kevtris*, to ROM który, dość dokładnie testuje działanie procesora. Jest o jeden z najlepszych testów na początek, do sprawdzenia działania procesora w emulatorze.

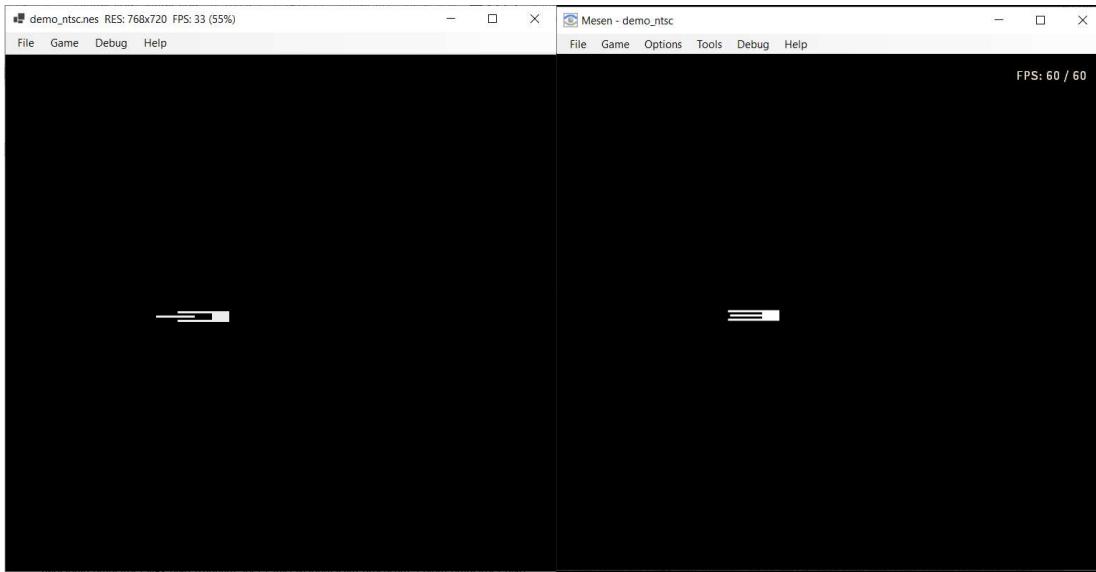
Emulator przeszedł wszystkie testy oferowane przez ten ROM pozytywnie, sygnalizując `OK` przy każdym teście (*rysunek 18*).



Rysunek 18. Rezultat działania ROM'u nestest.

nmi_sync – autorstwa *blargg*, weryfikuje synchronizację NMI, tworząc określony wzór na ekranie. Rysuje linie poprzez ustawianie bitu 0 na adresie 0x2001 (*PPUMASK*), aby włączyć tryb greyscale. Czas zapisu określa pozycję na ekranie, więc wszelkie problemy z synchronizacją spowodują przesunięcie lewej strony linii. Linie odniesienia są pokazane powyżej i poniżej rysowanej linii, pokazując prawidłowe położenie lewej krawędzi.

W wersji NTSC (*która jest emulowana w tym emulatorze*) lewy piksel środkowej linii powinienny być ciemniejszy, co jest spowodowane jego miganiem. Na *rysunku 19*, po lewej stronie, jest pokazany wynik testu emulatora, a po prawej stronie, przykład poprawnego wyświetlanego obrazu na emulatorze *Mesen*. Testy w aktualnej implementacji wskazują na problemy synchronizacji z PPU w modelu obsługi NMI. Rozwiążanie tego problemu będzie wymagać dalszych badań.



Rysunek 19. Po lewej: wynik testu emulatora, Po prawej: wynik testu na emulatorze Mesen.

7. Wnioski

Ta praca na temat rozwoju emulatora Nintendo Entertainment System (NES) ma znaczącą wartość, jeśli chodzi o przyczynienie się do zrozumienia architektury komputerów, języków programowania i tworzenia oprogramowania.

Tworzenie emulatora NES było trudnym, lecz satysfakcjonującym procesem. Celem emulatora było dokładne odtworzenie zachowania oryginalnego sprzętu NES, a produkt końcowy z powodzeniem osiągnął ten cel. Emulator jest w stanie emulować procesor MOS 6502 i jednostkę przetwarzania obrazu (PPU), które są głównymi komponentami konsoli NES. Emulator obsługuje również format pliku iNES do importowania kartidge'ów i obsługę co najmniej jednego Mapper'a.

Podczas procesu tworzenia emulatora nauczyliśmy się, jak ważne jest dokładne emulowanie komponentów sprzętowych, takich jak CPU i PPU. Zdobyliśmy także wiedzę na temat działania mapper'ów i ich znaczeniu w poszerzaniu zakresu jednociennej adresacji danych przez PPU.

Implementacja emulatora została wykonana przy użyciu języka C#, który zapewnił przejrzystą i ustrukturyzowaną bazę kodu, która odzwierciedlała architekturę konsoli. Ułatwiło to zrozumienie i utrzymanie architektury emulatora. Praca także przyczyniła się do głębszego zrozumienia Windows Forms, WinAPI i OpenGL API, które zostały użyte do opracowania graficznego interfejsu użytkownika (GUI) emulatora.

Ponadto użycie tych technologii w implementacji emulatora znacznie podkreśla znaczenie programowania międzyplatformowego, które zapewnia w kolejnych iteracjach oprogramowanie, że może być używane przez szerokie grono użytkowników, niezależnie od ich systemu operacyjnego.

Emulator ma kilka funkcjonalność wynikających z założeń, w tym ładowanie zrzutów pamięci ROM z poziomu GUI i konsoli, obsługę formatu iNES, obsługę Mapper'a 0 i 1. Dodatkowo została zaimplementowana możliwość podglądu wizualnej reprezentacji danych PPU w trakcie emulacji oraz informacji o nagłówku iNES.

Pomimo pomyślnej implementacji, nadal istnieją pewne ograniczenie związane ze wstępnych założeniami. Emulator nie posiada emulacji dźwięku, co jest istotnym aspektem gier. Dodatkowo nie posiada opcji ustawienia innych skrótów klawiszowych czy nawet obsługi kontrolerów takich jak Xbox One. Patrząc na inne dostępne emulatory, widzimy, że brakuje również takich funkcjonalności jak debugger'a dla instrukcji 6502, możliwości tworzenia zapisów stanu gry oraz obsługi większej liczby mapper'ów. Ograniczenia te można rozwiązać w przyszłych aktualizacjach emulatora i tu warto zaznaczyć, że przejrzysta architektura projektu może znacząco ułatwić ten proces.

Opracowanie emulatora dla tak przestarzałej konsoli, jak NES, jest złożonym i czasochłonnym procesem, który wymaga znaczących badań i wiedzy technicznej.

Po pierwsze, opracowanie emulatora wymaga dogłębniego zrozumienia architektury sprzętowej oryginalnej konsoli. Celem emulatorów jest jak najdokładniejsze odtworzenie zachowania

oryginalnego sprzętu. Aby to osiągnąć, należy dokładnie zbadać i zrozumieć komponenty sprzętowe, ich funkcje oraz wzajemne interakcje.

Po drugie, programowanie emulatora wymaga zaawansowanej wiedzy z zakresu informatyki, w tym niskopoziomowych języków programowania i architektury komputerów. Na przykład opracowanie emulatora NES wymaga doświadczenia w programowaniu na procesorze MOS 6502, 2C02 (PPU) i innych komponentów sprzętowych, a także znajomości struktur danych i algorytmów wydajnej emulacji.

Wszystkie te czynniki składają się na trudność i czasochłonność tworzenia emulatora. Znaczna większość czasu jest poświęcona badaniom, aby stworzyć dokładny i funkcjonalny emulator, który odwzorowuje zachowanie oryginalnej konsoli.

8. Bibliografia

- [1] <https://en.wikipedia.org/wiki/Emulator>
- [2] https://www.reddit.com/r/EmuDev/comments/vgy8tr/is_c_a_good_language_for_emulators/
- [3] https://en.wikipedia.org/wiki/Memory_management_controller
- [4] <https://history-computer.com/the-history-of-nintendo-entertainment-system-nes-emulation/>
- [5] <https://www.nesdev.org/wiki/2A03>
- [6] https://www.nesdev.org/wiki/CPU_registers
- [7] https://www.nesdev.org/wiki>Status_flags
- [8] <https://retrocomputing.stackexchange.com/questions/2592/what-is-the-relationship-between-the-ricoh-2a03-and-the-mos-6502>
- [9] <https://forums.nesdev.org/viewtopic.php?t=16770>
- [10] <https://www.nesdev.org/obelisk-6502-guide/addressing.html>
- [11] Ruszczyc, Jan (1987). Asembler 6502. Rozdział 5
- [12] <https://www.nesdev.org/obelisk-6502-guide/reference.html#JMP>
- [13] <http://www.6502.org/tutorials/6502pcodes.html>
- [14] https://www.masswerk.at/6502/6502_instruction_set.html
- [15] <https://www.nesdev.org/wiki/User:Koitsu>
- [16] <http://www.6502.org/tutorials/interrupts.html#1.3>
- [17] <http://nesdev.parodius.com/nintech.txt>
- [18] <https://forums.nesdev.org/viewtopic.php?t=19313>
- [19] <https://pl.wikipedia.org/wiki/MMIO>
- [20] <https://www.nesdev.org/wiki/PPU>
- [21] https://www.nesdev.org/wiki/PPU_rendering
- [22] <https://www.nesdev.org/2C02%20technical%20reference.TXT>
- [23] <https://www.nesdev.org/NinTech.txt>
- [24] nestech.txt <http://nesdev.parodius.com/nodox200.zip>
- [25] <https://forums.nesdev.org/viewtopic.php?t=8832>
- [26] <http://nesdev.parodius.com/NESDoc.pdf>
- [27] https://austinmorlan.com/posts/nes_rendering_overview/
- [28] https://www.nesdev.org/wiki/PPU_nametables
- [29] https://d1.amobbs.com/bbs_upload782111/files_28/ourdev_551332.pdf
- [30] https://www.nesdev.org/wiki/Standard_controller
- [31] <https://www.nesdev.org/wiki/INES>
- [32] <https://www.mesen.ca/oldindex.php>
- [33] <https://github.com/iaddis/metalnes>
- [34] <https://hothardware.com/news/metalnes-ultimate-nes-emu-your-pc-isnt-fast-enough>
- [35] https://nescartdb.com/search/advanced?system_op=equal&system=NTSC&ines_op=less_than_eq&ines=1&page=13

9. Spis rysunków

RYSUNEK 1. BINARNA REPREZENTACJA REJESTRU STANU PROCESORA W POSTACI JEDNEGO BAJTA. OD LEWEJ (TJ. NAJBARDZIEJ ZNACZĄCEGO BITU): FLAGA UJEMNA (N), FLAGA PRZEPEŁNIENIA (V), NIEUŻYWANY BIT (ZAWSZE USTAWIONY NA 1), POLECENIE PRZERWANIA (B), TRYB DZIESIĘTNY (D), FLAGA MASKI PRZERWANIA (I), FLAGA ZERA (Z) I FLAGA PRZENIESIENIA (C).....	6
RYSUNEK 2. KOMPOZYCJA BLOKU TŁA SKŁADAJĄCEGO SIĘ Z 4 KAFELKÓW. KAŻDY KAFELEK MA SIATKĘ PIKSELI Z INDEKSAMI KOLORU AKTYWNEJ PALETY.....	16
RYSUNEK 3. LEwy GÓRNY RÓG: KAFELEK TŁA Z SIATKA INDEKSÓW KOLORU PALETY, PRAWY GÓRNY RÓG: REPREZENTACJA BINARNA WARTOŚCI INDEKSÓW KAFELKA, LEwy DOLNY RÓG: NISKI BIT WARTOŚCI INDEKSÓW, PRAWY DOLNY RÓG: WYSOKI BIT WARTOŚCI INDEKSÓW	17
RYSUNEK 4. SCHEMAT PRZEDSTAWIAJĄCY ROZMIESZCZENIE PINÓW W KARTRIDŻU FAMICOM.....	19
RYSUNEK 5. DIAGRAM KLAS EMULATORA.....	24
RYSUNEK 6. SZCZEGÓLOWY DIAGRAM KLASY CPU I JEGO STRUKTUR.....	25
RYSUNEK 7. KOD ŹRÓDŁOWY METODY <i>CLOCK()</i> W KLASIE CPU.....	26
RYSUNEK 8. KOD ŹRÓDŁOWY STRUKTURY POJEDYNCZEJ INSTRUKCJI W KLASIE CPU.....	27
RYSUNEK 9. KOD ŹRÓDŁOWY METODY <i>ADC()</i> W KLASIE CPU, IMPLEMENTUJĄCY INSTRUKCJĘ <i>ADC</i> ..	28
RYSUNEK 10. SZCZEGÓLOWY DIAGRAM KLASY PPU I INNYCH KLAS ZWIĄZANYCH Z NIĄ.....	29
RYSUNEK 11. SCHEMAT SYNCHRONIZACJI NTSC UKŁADU 2C02 (PPU). (ŹRÓDŁO: [21]).....	30
RYSUNEK 12. KOD ŹRÓDŁOWY METODY <i>CLOCK()</i> W KLASIE PPU.....	30
RYSUNEK 13. SZCZEGÓLOWY DIAGRAM KLASY CARTRIDGE I INNYCH KLAS ZWIĄZANYCH Z NIĄ.....	32
RYSUNEK 14. KOD ŹRÓDŁOWY KONSTRUKTORA KLASY CARTRIDGE.....	33
RYSUNEK 15. KOD ŹRÓDŁOWY METODY <i>CpuRead</i> W KLASIE CARTRIDGE	34
RYSUNEK 16. ZRZUT EKRANU PRZEDSTAWIAJĄCY URUCHOMIONY EMULATOR Z ROM'EM "DEMO.NES" ..	35
RYSUNEK 17. ZRZUT EKRANU PRZEDSTAWIAJĄCY OKNO EMULATORA DEDYKOWANE DO PODGLĄDU DANYCH PPU.....	36
RYSUNEK 18. REZULTAT DZIAŁANIA ROM'U NESTEST.....	37
RYSUNEK 19. PO LEWEJ: WYNIK TESTU EMULATORA, PO PRAWEJ: WYNIK TESTU NA EMULATORZE MESEN.....	38

10. Spis tabel

TABELA 1. MAPA PRZESTRZENI ADRESOWEJ KONSOLI / CPU	10
TABELA 2. MAPA PRZESTRZENI ADRESOWEJ PPU.....	14
TABELA 3. TABELA PRAWDY OKREŚLAJĄCA BIT PRZEPEŁNIENIA.....	27
TABELA 4. LISTA PRZETESTOWANYCH GIER.	36