Emulator Nintendo Entertainment System

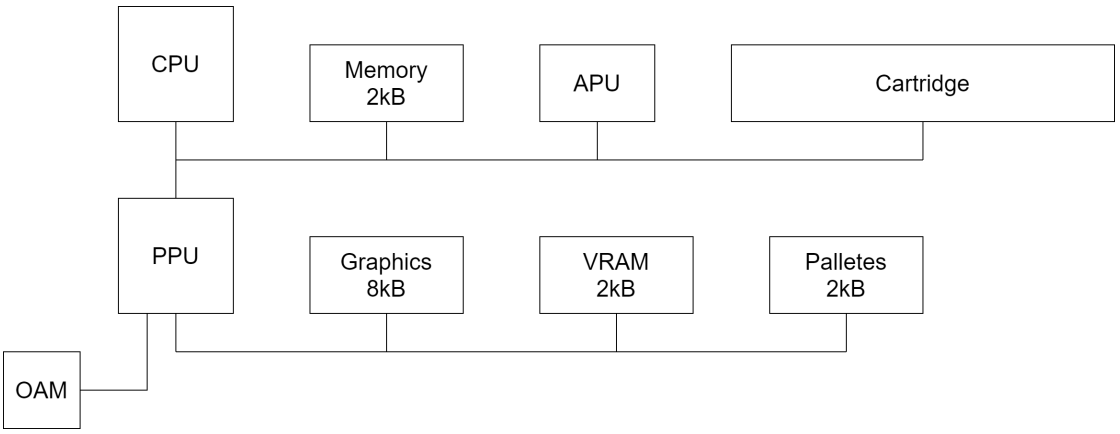
Michał Wójtowicz, 252770

14 czerwca 2021

1 Architektura Nintendo Entertainment System

NES posiada dwie główne magistrale. Do pierwszej podłączony jest procesor CPU, procesor APU, procesor PPU, pamięć wewnętrzna (RAM) i pamięć kartridża. Do drugiej podłączony jest procesor PPU, pamięć zawierająca grafikę, pamięć wewnętrzna (VRAM) i pamięć zawierająca kolory. Warta uwagi jest też podłączona bezpośrednio do procesora PPU pamięć atrbutów obiektów.

Poniżej został przedstawiony uproszczony schemat architektury NES.



Rysunek 1: Uproszczony schemat architektury NES

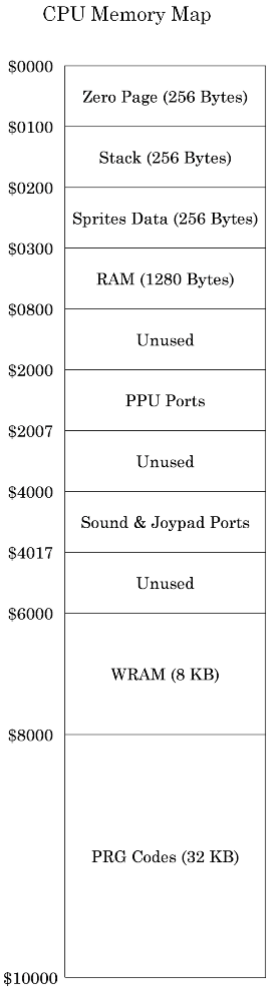
1.1 CPU

Mikroprocesor użyty w NES ma nazwę kodową 2A03. W znacznej części bazuje on na procesorze MOS 6502.
Różnica dotyczy trybu liczb dziesiętnych, który w 2A03 nie jest podłączony, mimo że wciąż się tam znajduje. Ponadto, 2A03 ma wbudowaną logikę obsługi dźwięku (APU).

Procesor działa z taktowaniem ok. 1.79 MHz. Zawiera 13 trybów adresowania i 56 instrukcji (kodów operacji). Dostępne są 8-bitowe rejestry: akumulator A, dwa rejestry indeksowe X oraz Y i rejestr statusowy P. Ponadto procesor używa dwubajtowego licznika programu PC i bajtowego wskaźnika stosu. Do dyspozycji procesora są 2kB pamięci RAM.

Flagi zawarte w rejestrze statusowym to przeniesienie C, zero Z, wyłączenie przerwań I, tryb dziesiętny D (nieużywany), komenda przerwy B, przepełnienie O i wartość ujemna N. Jeden bit jest wolny.

Zakres adresowania procesora to 64kB. Poza własną pamięcią RAM, procesor ma dostęp do rejestrów PPU i APU, urządzeń wejścia/wyjścia i pamięci kartridża. Reszta pamięci, do której nie zostały podłączone komponenty systemu, jest "odbiciem" istniejących połączeń.
Przykładowo, rejestr kontrolny PPU zajmuje 8 bitów pamięci rozpoczynającej się na pozycji $2000 - ostatni bit znajduje się na pozycji $2007. Chcąc odczytać konkretną pozycję z rejestru potrzebne są jedynie 3 ostatnie bity z adresu, co można osiągnąć maskując adres wartością 0x07. W taki sposób adresowanie na rejestr PPU powtórzone jest 1023 razy, na całym zakresie $2000 - $3FFF.



Rysunek 2: Adresowanie CPU

Każdy cykl zegarowy w procesorze jest cyklem odczytu lub zapisu.

Procesor obsługuje dwa tryby przerwań - przerwanie wywołane zapytaniem (Interrupt Request - IRQ) oraz przerwanie niemaskowalne (Non-Maskable Interrupt - NMI). IRQ wywoływane jest, gdy na wyjściu IRQ zostanie wykryty stan niski. NMI wywoływane jest, gdy stan wyjścia NMI przejdzie z wysokiego na niski.

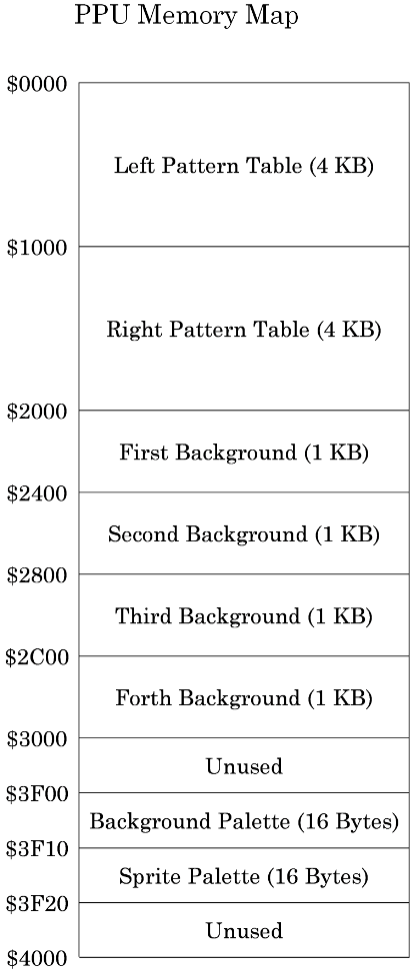
Interesującym błędem zawartym w CPU jest błąd z instrukcją skoku pośredniego. Mianowicie, podając adres $xxFF i próbując odczytać dwa bajty, spodziewanym przejściem byłoby $xxFF→$x(x+1)00. Jednak przeniesienie dwóch młodszych bajtów nie zostanie uwzględnione, przez co rzeczywiste przejściem będzie $xxFF→$xx00. Błąd ten został odkryty bardzo dawno, obejścia tego błędu były zaimplementowane w gry - emulując więc procesor należy pamiętać aby uwzględnić w implementacji ten błąd, jako że jest to działanie niepoprawne, lecz jednak spodziewane.

1.2 PPU

Picture Processing Unit, jednostka odpowiedzialna za generowanie obrazu. Tworzony obraz kompozytowy to 240 linii pikseli. Oryginalnie, w wersji PAL, używany był PPU z nazwą kodową 2C02.

Jednostka działa z taktowaniem 3 razy większym niż procesor CPU, co sprowadza się do ok. 5.37 MHz. Nie zawiera własnych trybów adresowania ani zdefiniowanych instrukcji. Posiada za to 3 8-bitowe rejestry określające pracę PPU: kontrolny, maskujący i statusowy. PPU ma też dostęp do pamięci OAM (256 bajtów) i własnej pamięci RAM - 2kB.

Zakres adresowania PPU to 16kB, $0000 - $3FFF. Jest on całkowicie niezależny od adresowania CPU. Umożliwia na dostęp do pamięci własnej PPU (w której znajdują się informacje o tłach), do pamięci OAM (w której znajdują się informacje o obiektach (sprites)), do palet kolorów i do tabeli wzorców (w której znajdują się kształty tworzące renderowane grafiki).

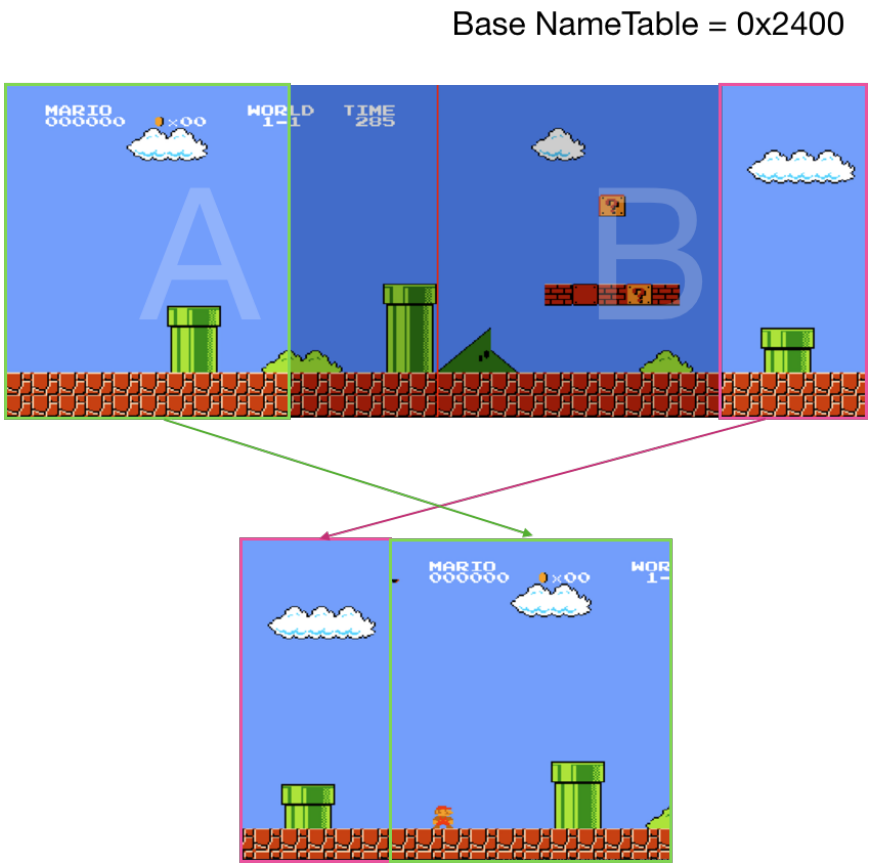


Rysunek 3: Adresowanie PPU

PPU był w stanie renderować tła oraz do 64 obiektów na raz. Rozmiar renderowanego obrazu to 256x240 pikseli. Obiekty miały rozmiary 8x8 pikseli lub 8x16 pikseli. Tło można było przewijać - zarówno w osi X, jak i Y - rząd pikseli "na raz"(tzw. "fine scrolling").

Zarówno tła jak i obiekty utworzone były z pól 8x8 pikseli, definiowanych przez tabele wzorców. W tej tabeli zdefiniowane były jedynie dwa bity koloru - dodatkowe dwa bity odczytywane były z pamięci OAM (Object Attribute Memory). W pamięci PPU załadowane były informacje o tłach, tzw. "nametables" - definiowały one pozycję pól budujących tło.

Nie trudno jednak obliczyć, że pamięci PPU było aż nadto - generowanych było $32 \times 30 = 960$ pól, których opis zawierał się w jednym bajcie pamięci PPU. A dostępnej pamięci było 2kB - pozwalało to wczytać kolejny element tła i być gotowym na jego wyświetlenie zanim do niego doszło. I właśnie ta cecha NESa umożliwiła "fine scrolling". Dzięki renderowaniu teł z pewnym przesunięciem osiągany był efekt płynnego przewijania.



Rysunek 4: Przewijanie teł

Wspomniana wcześniej pamięć OAM zawierała informacje o wszystkich 64 obiektach. Jego współrzędne X i Y, numer pola reprezentującego obiekt i flagi, opisujące dwa bity z koloru obiektu, pozycję obiektu w osi Z (przed tłem czy za tłem) i lustrzane odbicia wzorca obiektu.

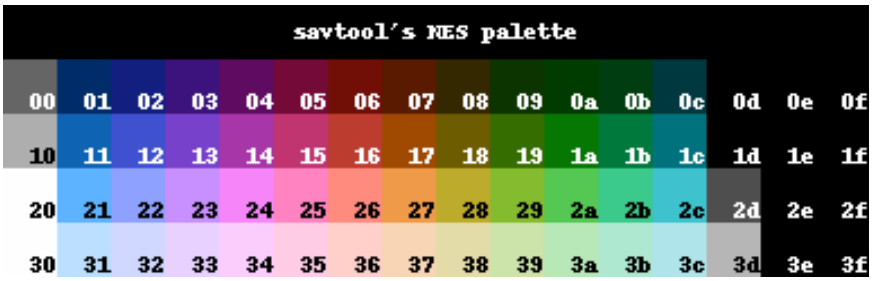
Pamięć OAM była nadpisywana bardzo często. Była też dość duża (256 bajty), przez co kopio-

wanie bajtów pojedynczo było bardzo nieefektywne. Pamięci OAM umożliwiono DMA - Direct Memory Access, dostęp bezpośredni do procesora. Przez to, ilekroć wystąpiła aktualizacja pamięci OAM (wpisanie $\$XX$ do rejestru OAM DMA), procesor był zatrzymywany, a cała strona \$XX00 – \$XXFF przepisywana była do pamięci OAM. Po zakończeniu transferu działanie procesora było wznawiane. W ten sposób aktualizacja OAM odbywała się ok. 4 razy szybciej niż w przypadku ręcznego kopiowania bajtów.

Renderowanie obrazu odbywało się w liniach, tzw. "scanlines". I mimo maksymalnie 64 obiektów na ekranie, w jednej takiej linii znajdować się ich mogło maksymalnie 8. W sytuacji zbyt wielu obiektów na jednej linii, ustawiana była flaga przepełnienia. Dlatego też, kiedy na ekranie dużo się dzieje, obiekty mogą migotać.

PPU było też odpowiedzialne za wykrywanie kolizji. Jeżeli pierwszy obiekt (przezroczysty) trafił na tło (nieprzezroczyste), dochodziło do kolizji - ustawiana była odpowiednia flaga. Podczas jednej klatki wystąpić mogła jedna kolizja.

NES nie obsługiwał kolorów w znanym nam dziś formacie RGB - konkretne kolory miały konkretne wartości i tylko korzystając z odpowiednich tabel można było odczytać kod odpowiedniego koloru.



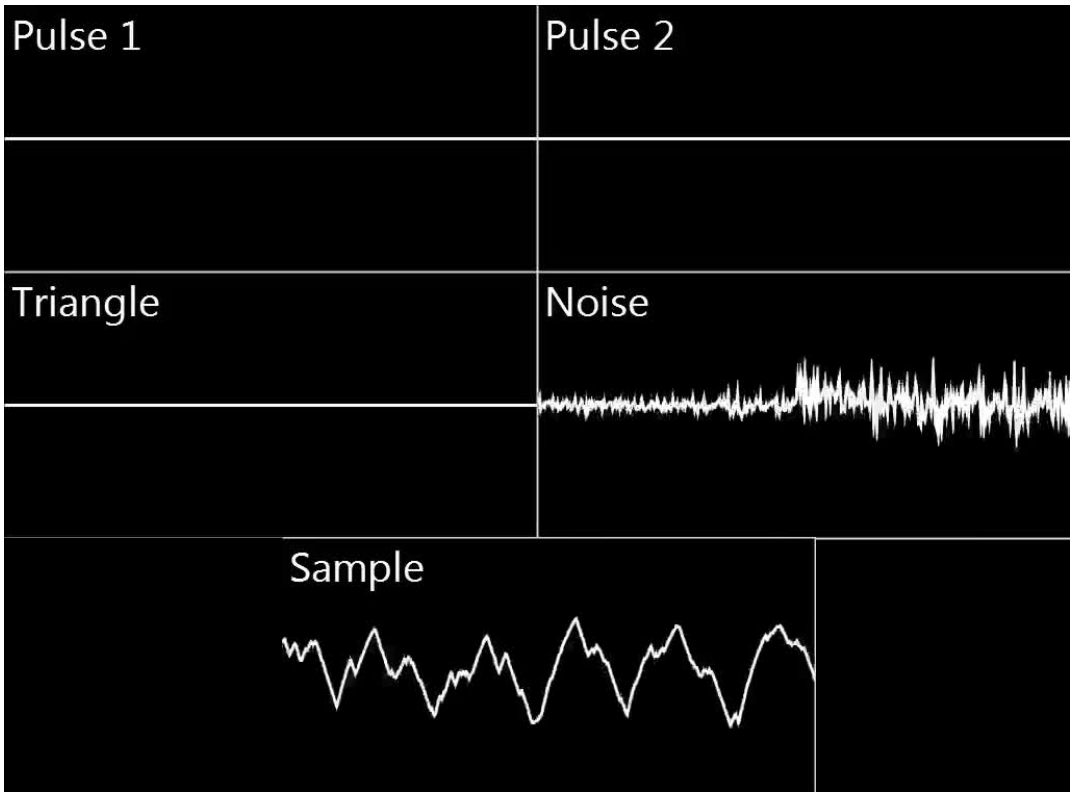
Rysunek 5: Paleta kolorów PPU

1.3 APU

Jednostka APU wspierała dwa kanały z falą pulsacyjną, kanał z falą trójkątną, kanał z szumem oraz kanał z modulacją delta. W celu odtwarzania dźwięku z danego kanału należało odpowiednio skonfigurować rejestr tego kanału.

Kanały z falą pulsacyjną wspierały określenie częstotliwości i długości trwania dźwięku, przemiatanie częstotliwości i obwiednie głośności. Kanał z szumem używał rejestru przesuwającego aby generować szum pseudolosowy.

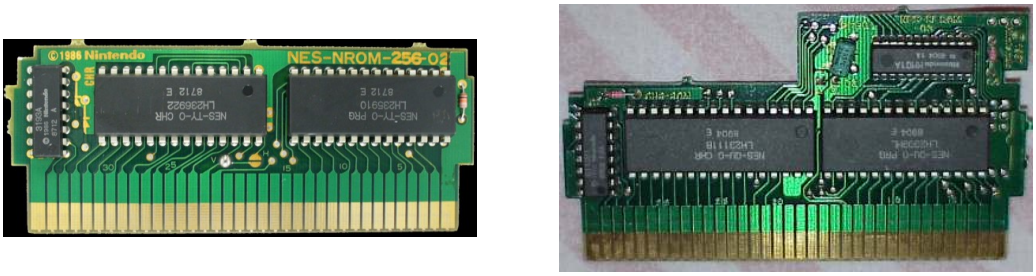
Kanał modulacji delta mógł odtwarzać dźwięk na podstawie wzorców (sampli) zapisanych w pamięci.



Rysunek 6: Przykładowo wypełnione kanały audio - dźwięk z pamięci z szumem

1.4 Mappery

Gry na platformę NES cierpiały z powodu dość rygorystycznych ograniczeń pamięciowych - 32kB dla pamięci programu i 8kB dla pamięci graficznej. O ile w początkowych czasach NESa było to wystarczająco dużo, z czasem ambicje programistów rosły. Obmyślono więc mapper - układ znajdujący się na kartridżu umożliwiający zmianę banków pamięci programu lub grafiki.



Rysunek 7: Porównanie dwóch mapperów

Mimo, że mappery były częścią kartridża, obrazy kartridży emulują jedynie zawartość pamięci kartridża - bez mapperów. Obsługa wielu mapperów, w celu obsługi wielu gier, leży więc po stronie emulatora.

1.5 Kontrolery

Kontrolery systemu NES zawierały 8 przycisków i opisywane były przez 8-bitowy rejestr. Odczyt z kontrolerów inicjowany był zapisem 1 do adresu $4016, a koniec odczytu zapisem 0 do tego samego adresu.

Odczyt polegał na odczytaniu rejestru kontrolera 8 razy. Odczytywany był jedynie najstarszy bit, a po odczycie rejestr był przesuwany o jedną pozycję w lewo - rejestr typu PISO.

Rysunek 8: Podstawowy kontroler systemu NES

2 Emulacja

Emulator został napisany w języku C++.

2.1 Emulacja CPU

Z uwagi na dużo bardziej dostępną dokumentację mikroprocesora 6502 i małe różnice między nim i układem 2A03, emulacja będzie bazować na modelu 6502.

W celu emulacji CPU zdefiniowane zostały wszystkie tryby adresowania procesora 6502:

```
// Addressing Modes
uint8_t IMM();      // Immediate Addressing
uint8_t ABS();      // Absolute Addressing
uint8_t ZP0();      // Zero Page Addressing
uint8_t ZPX();      // Indexed Zero Page Addressing
uint8_t ZPY();      // Indexed Zero Page Addressing
uint8_t ABX();      // Indexed Absolute Addressing
uint8_t ABY();      // Indexed Absolute Addressing
uint8_t IMP();      // Implied Addressing
uint8_t REL();      // Relative Addressing
uint8_t IZX();      // Indexed Indirect Addressing
uint8_t IZY();      // Indirect Indexed Addressing
uint8_t IND();      // Absolute Indirect
```

Rysunek 9: Lista trybów adresowania

```
// Indexed Absolute addressing:
// Absolute addressing with X register offset
uint8_t cpu6502::ABX()
{
    uint16_t lo = read(pc);
    pc++;
    uint16_t hi = read(pc);
    pc++;

    addr_abs = (hi << 8) | lo;

    addr_abs += x;

    // Check if the address changed to a different page,
    // if so notify system that additional clock cycle may be necessary
    if ((addr_abs & 0xFF00) != (hi << 8))
        return 1;
    else
        return 0;
}
```

Rysunek 10: Przykładowa implementacja trybu adresowania: adresowanie bezwzględne z przesunięciem

Podobnie kody operacji:

```
uint8_t ASL();      // Arithmetic Shift Left
uint8_t BCC();      // Branch on Carry Clear
uint8_t BCS();      // Branch on Carry Set
uint8_t BEQ();      // Branch on Equal
uint8_t BIT();      // Test Bits
uint8_t BMI();      // Branch on Minus
uint8_t BNE();      // Branch on Not Equal
uint8_t BPL();      // Branch on Plus
uint8_t BRK();      // Break
uint8_t BVC();      // Branch on Overflow Clear
uint8_t BVS();      // Branch on Overflow Set
uint8_t CLC();      // Clear Carry
uint8_t CLD();      // Clear Decimal
uint8_t CLI();      // Clear Interrupt
uint8_t CLV();      // Clear Overflow
uint8_t CMP();      // Compare Accumulator
uint8_t CPX();      // Compare X register
uint8_t CPY();      // Compare Y register
uint8_t DEC();      // Decrement Memory
uint8_t DEX();      // Decrement X
uint8_t DEY();      // Decrement Y
uint8_t EOR();      // Bitwise Exclusive OR
uint8_t INC();      // Increment Memory
uint8_t INX();      // Increment X
uint8_t INY();      // Increment Y
uint8_t JMP();      // Jump
uint8_t JSR();      // Jump to Subroutine
```

Rysunek 11: Fragment listy kodów operacji

```
// Add with Carry:
// Adds fetched value and Carry bit to Accumulator
// Carry is set if val > 0x00FF
// Zero is set if val == 0x0000
// Negative set if val >= 0x0080
// Overflow is set if value overflows ((A XOR Result) & NOT(A XOR Fetched))
uint8_t cpu6502::ADC()
{
    fetch();
    temp = (uint16_t)a + (uint16_t)fetched + (uint16_t)GetFlag(C);

    SetFlag(C, temp > 0x00FF);
    SetFlag(Z, (temp & 0x00FF) == 0);
    SetFlag(N, temp & 0x80);
    SetFlag(V, (((uint16_t)a ^ (uint16_t)temp) & ~((uint16_t)a ^ (uint16_t)fetched)) & 0x0080);

    a = temp & 0x00FF;
    return 1;
}
```

Rysunek 12: Przykładowa implementacja kodu operacji: dodanie z przeniesieniem

W celu dostępu do kodów operacji została utworzona tablica dwuwymiarowa, zawierająca tak samo rozłożone (MSD—LSD) zestawy instrukcji jak w procesorze 6502.

```
using a = cpu6502;
lookup =
{
    { "BRK", &a::BRK, &a::IMP, 7 },{ "ORA", &a::ORA, &a::IZX, 6 },{ "???", &a::XXX, &a::IMP, 2 },
    { "BPL", &a::BPL, &a::REL, 2 },{ "ORA", &a::ORA, &a::IZY, 5 },{ "???", &a::XXX, &a::IMP, 2 },
    { "JSR", &a::JSR, &a::ABS, 6 },{ "AND", &a::AND, &a::IZX, 6 },{ "???", &a::XXX, &a::IMP, 2 },
    { "BMI", &a::BMI, &a::REL, 2 },{ "AND", &a::AND, &a::IZY, 5 },{ "???", &a::XXX, &a::IMP, 2 },
    { "RTI", &a::RTI, &a::IMP, 6 },{ "EOR", &a::EOR, &a::IZX, 6 },{ "???", &a::XXX, &a::IMP, 2 },
    { "BVC", &a::BVC, &a::REL, 2 },{ "EOR", &a::EOR, &a::IZY, 5 },{ "???", &a::XXX, &a::IMP, 2 },
    { "RTS", &a::RTS, &a::IMP, 6 },{ "ADC", &a::ADC, &a::IZX, 6 },{ "???", &a::XXX, &a::IMP, 2 },
    { "BVS", &a::BVS, &a::REL, 2 },{ "ADC", &a::ADC, &a::IZY, 5 },{ "???", &a::XXX, &a::IMP, 2 },
    { "???", &a::NOP, &a::IMP, 2 },{ "STA", &a::STA, &a::IZX, 6 },{ "???", &a::NOP, &a::IMP, 2 },
```

Rysunek 13: Fragment tablicy zwierającej kody operacji

MSD \ LSD	0	1	2	3	4
0	BRK Implied 1 7	ORA (IND, X) 2 6			
1	BPL Relative 2 2**	ORA (IND), Y 2 5*			
2	JSR Absolute 3 6	AND (IND, X) 2 6			BIT ZP 2 3
3	BMI Relative 2 2**	AND (IND), Y 2 5*			
4	RTI Implied 1 6	EOR (IND, X) 2 6			
5	BVC Relative 2 2**	EOR (IND), Y 2 5*			
6	RTS Implied 1 6	ADC (IND, X) 2 6			

Rysunek 14: Fragment tablicy kodów operacji z dokumentacji procesora 6502

Należało również zaimplementować “inne” funkcje procesora, takie jak obsługa zegara (zliczanie cykli) czy przerwań:

```
// Performing a single clock cycle
void cpu6502::clock()
{
    if (cycles == 0)
    {
        // Get current Opcode
        opcode = read(pc);
        // Prepare Program Counter to read the next byte
        pc++;

        // Get cycle count for current Opcode
        cycles = lookup[opcode].cycles;

        // Check if Opcode requires additional cycles
        uint8_t additional_cycle1 = (this->*lookup[opcode].addrmode)();
        uint8_t additional_cycle2 = (this->*lookup[opcode].operate)();

        cycles += (additional_cycle1 & additional_cycle2);
    }

    cycles--;
}
```

Rysunek 15: Implementacja obsługi zegara

```
// Interrupt request
void cpu6502::irq()
{
    // IF Interrupts are allowed
    if (GetFlag(I) == 0)
    {
        // Push Program Counter to stack
        write(0x0100 + stck, (pc >> 8) & 0x00FF);
        stck--;
        write(0x0100 + stck, pc & 0x00FF);
        stck--;

        // Push Status register to stack
        SetFlag(B, 0);
        SetFlag(U, 1);
        SetFlag(I, 1);
        write(0x0100 + stck, status);
        stck--;

        // Get Program Counter address and set it
        addr_abs = 0xFFFE;
        uint16_t lo = read(addr_abs + 0);
        uint16_t hi = read(addr_abs + 1);
        pc = (hi << 8) | lo;

        // Execution time
        cycles = 7;
    }
}
```

Rysunek 16: Implementacja obsługi przerwań

W oryginalnym układzie procesora występował błąd: w przypadku pośredniego skoku $xxFF (tryb adresowania IND) wartość MSB pobierana była z adresu $xx00, a nie $x(x+1)00. Obejścia tego błędu są zakodowane w gry, przez co w celu poprawnego działania emuloator musi emulować również ten błąd.

```
// Absolute Indirect:
// Supplied address is a pointer, so it is necessary to extract data with read()
uint8_t cpu6502::IND()
{
    uint16_t ptr_lo = read(pc);
    pc++;
    uint16_t ptr_hi = read(pc);
    pc++;

    uint16_t ptr = (ptr_hi << 8) | ptr_lo;

    // Page boundary hardware bug
    if (ptr_lo == 0x00FF)
    {
        addr_abs = (read(ptr & 0xFF00) << 8) | read(ptr + 0);
    }
    else
    {
        addr_abs = (read(ptr + 1) << 8) | read(ptr + 0);
    }
    return 0;
}
```

Rysunek 17: Implementacja błędu adresowania pośredniego

2.2 Emulacja PPU

Emulacja bazuje na podstawowym PPU zawartym w wersji PAL konsoli NES - 2C02.

Implementacja wymagała zdefiniowania pamięci używanej przez PPU.

```
// Object Attribute Memory
uint8_t* ppuOAM     = (uint8_t*)OAM;

// Nametables
uint8_t     nameTable[2][1024];
// Pallete tables
uint8_t     palleteTable[32];
// Pattern tables
uint8_t     patternTable[2][4096];

// Palettes
olc::Pixel  palettes[0x40];
// "Sprite" - a screen, where pixels are drawn
olc::Sprite screen = olc::Sprite(256, 240);
```

Rysunek 18: Definiowanie pamięci używanej przez PPU

```
// Control register - controlling PPU opeartion
union PPUCTRL
{
    struct
    {
        uint8_t nametable_x         : 1;
        uint8_t nametable_y         : 1;
        uint8_t increment_mode      : 1;
        uint8_t sprite_pattern      : 1;
        uint8_t background_pattern  : 1;
        uint8_t sprite_size         : 1;
        uint8_t master_slave_mode   : 1;
        uint8_t enable_nmi          : 1;
    };
    uint8_t reg = 0x00;
};
// Mask register - controlling graphics rendering
union PPUMASK
{
    struct
    {
        uint8_t greyscale                   : 1;
        uint8_t render_background_leftmost  : 1;
        uint8_t render_sprites_leftmost     : 1;
        uint8_t render_background           : 1;
        uint8_t render_sprites              : 1;
        uint8_t enhance_red                 : 1;
        uint8_t enhance_green               : 1;
        uint8_t enhance_blue                : 1;
    };
    uint8_t reg = 0x00;
};
// Status register - reflecting state of PPU
union PPUSTATUS
{
    struct
    {
        uint8_t unused          : 5;
        uint8_t sprite_overflow : 1;
        uint8_t sprite_0_hit    : 1;
        uint8_t vertical_blank  : 1;
    };
    uint8_t reg = 0x00;
};
```

Rysunek 19: Definiowanie rejestrów wewnętrznych PPU

W celu implementacji płynnego przewijania skorzystano z rozwiązania sugerowanego przez Nesdev wiki - tzw. rejestr loopy.

```
// loopy register - named after the solution creator
// Smooth scrolling between nametables
union LOOPY
{
    struct
    {
        uint16_t coarse_x : 5;
        uint16_t coarse_y : 5;
        uint16_t nametable_x : 1;
        uint16_t nametable_y : 1;
        uint16_t fine_y : 3;
        uint16_t unused : 1;
    };
    uint16_t reg = 0x0000;
};
LOOPY   vram_addr;
LOOPY   tram_addr;

// Pixel offset - horizontal
uint8_t fine_x          = 0x00;

uint8_t address_latch   = 0x00;
uint8_t ppu_data_buffer = 0x00;
```

Rysunek 20: Implementacja płynnego przewijania pomiędzy nametables

Implementacja OAM.

```
struct ObjectAttribute
{
    uint8_t y;
    uint8_t id;
    uint8_t attribute;
    uint8_t x;
};

ObjectAttribute OAM[64];
uint8_t         oam_addr = 0x00;
```

Rysunek 21: Implementacja OAM

W nowej instancji PPU należało wypełnić palety kolorami. Wartości pobrane z wiki Nesdev, jako najwierniej odzwierciedlające oryginalną paletę kolorów.

```
PPU2C02::PPU2C02()
{
    // Loading the palettes representing the NES colours
    palettes[0x00] = olc::Pixel(84, 84, 84);
    palettes[0x01] = olc::Pixel(0, 30, 116);
    palettes[0x02] = olc::Pixel(8, 16, 144);
    palettes[0x03] = olc::Pixel(48, 0, 136);
    palettes[0x04] = olc::Pixel(68, 0, 100);
    palettes[0x05] = olc::Pixel(92, 0, 48);
    palettes[0x06] = olc::Pixel(84, 4, 0);
    palettes[0x07] = olc::Pixel(60, 24, 0);
    palettes[0x08] = olc::Pixel(32, 42, 0);
    palettes[0x09] = olc::Pixel(8, 58, 0);
```

Rysunek 22: Inicjalizacja PPU - ładowanie palet kolorów do pamięci

Należało udostępnić procesorowi CPU dostęp do rejestrów procesora PPU.

```cpp
// Get data from PPU registers
uint8_t PPU2C02::cpuRead(uint16_t addr, bool readOnly)
{
    uint8_t data = 0x00;

    if (readOnly)
    {
        switch (addr) {
        case 0x0000:    // Control
            data = control.reg;
            break;
        case 0x0001:    // Mask
            data = mask.reg;
            break;
        case 0x0002:    // Status
            data = status.reg;
            break;
        case 0x0003:    // OAM Address
            break;
        case 0x0004:    // OAM Data
            break;
        case 0x0005:    // Scroll
            break;
        case 0x0006:    // PPU Address
            break;
        case 0x0007:    // PPU Data
            break;
        }
    }
    else
    {
        switch (addr) {
        case 0x0000:    // Control
            break;
        case 0x0001:    // Mask
            break;
        case 0x0002:    // Status
            data = (status.reg & 0xE0) | (ppu_data_buffer & 0x1F);
            status.vertical_blank = 0;
            address_latch = 0;
            break;
        case 0x0003:    // OAM Address
            break;
        case 0x0004:    // OAM Data
            data = ppuOAM[oam_addr];
            break;
        case 0x0005:    // Scroll
            break;
        case 0x0006:    // PPU Address
            break;
        case 0x0007:    // PPU Data
            data = ppu_data_buffer;
            ppu_data_buffer = ppuRead(vram_addr.reg);

            if (vram_addr.reg >= 0x3F00)
                data = ppu_data_buffer;
            vram_addr.reg += (control.increment_mode ? 32 : 1);
            break;
        }
    }

    return data;
}
```

```cpp
// Write to the PPU registers
void PPU2C02::cpuWrite(uint16_t addr, uint8_t data)
{
    switch (addr) {
    case 0x0000:    // Control
        control.reg = data;
        tram_addr.nametable_x = control.nametable_x;
        tram_addr.nametable_y = control.nametable_y;
        break;
    case 0x0001:    // Mask
        mask.reg = data;
        break;
    case 0x0002:    // Status
        break;
    case 0x0003:    // OAM Address
        oam_addr = data;
        break;
    case 0x0004:    // OAM Data
        ppuOAM[oam_addr] = data;
        break;
    case 0x0005:    // Scroll
        if (address_latch == 0)
        {
            fine_x = data & 0x07;
            tram_addr.coarse_x = data >> 3;
            address_latch = 1;
        }
        else
        {
            tram_addr.fine_y = data & 0x07;
            tram_addr.coarse_y = data >> 3;
            address_latch = 0;
        }
        break;
    case 0x0006:    // PPU Address
        if (address_latch == 0)
        {
            tram_addr.reg = (uint16_t)((data & 0x3F) << 8) | (tram_addr.reg & 0x00FF);
            address_latch = 1;
        }
        else
        {
            tram_addr.reg = (tram_addr.reg & 0xFF00) | data;
            vram_addr = tram_addr;
            address_latch = 0;
        }
        break;
    case 0x0007:    // PPU Data
        ppuWrite(vram_addr.reg, data);
        vram_addr.reg += (control.increment_mode ? 32 : 1);
        break;
    }
}
```

Rysunek 23: Komunikacja z rejestrami PPU

Procesor PPU ma własny zapis/odczyt.

```cpp
// Read from PPU memory
uint8_t PPU2C02::ppuRead(uint16_t address, bool readOnly)
{
    uint8_t data    = 0x00;

    address &= 0x3FFF;

    if (cartridge->ppuRead(address, data)) {}

    else if (address >= 0x0000 && address <= 0x1FFF)
    {
        data = patternTable[(address & 0x1000) >> 12][address & 0x0FFF];
    }
    else if (address >= 0x2000 && address <= 0x3EFF)
    {
        address &= 0x0FFF;

        if (cartridge->mirror == Cartridge::MIRROR::VERTICAL)
        {
            // Vertical
            if (address >= 0x0000 && address <= 0x03FF)
                data = nameTable[0][address & 0x03FF];
            if (address >= 0x0400 && address <= 0x07FF)
                data = nameTable[1][address & 0x03FF];
            if (address >= 0x0800 && address <= 0x0BFF)
                data = nameTable[0][address & 0x03FF];
            if (address >= 0x0C00 && address <= 0x0FFF)
                data = nameTable[1][address & 0x03FF];
        }
        else if (cartridge->mirror == Cartridge::MIRROR::HORIZONTAL)
        {
            // Horizontal
            if (address >= 0x0000 && address <= 0x03FF)
                data = nameTable[0][address & 0x03FF];
            if (address >= 0x0400 && address <= 0x07FF)
                data = nameTable[0][address & 0x03FF];
            if (address >= 0x0800 && address <= 0x0BFF)
                data = nameTable[1][address & 0x03FF];
            if (address >= 0x0C00 && address <= 0x0FFF)
                data = nameTable[1][address & 0x03FF];
        }
    }
    else if (address >= 0x3F00 && address <= 0x3FFF)
    {
        address &= 0x001F;
        if (address == 0x0010) address = 0x0000;
        if (address == 0x0014) address = 0x0004;
        if (address == 0x0018) address = 0x0008;
        if (address == 0x001C) address = 0x000C;
        data = palleteTable[address] & (mask.greyscale ? 0x30 : 0x3F);
    }


    return data;
}
```

```cpp
// Write to PPU memory
void PPU2C02::ppuWrite(uint16_t addr, uint8_t data)
{
    addr &= 0x3FFF;

    if (cartridge->ppuWrite(addr, data)) {}

    else if (addr >= 0x0000 && addr <= 0x1FFF)
    {
        patternTable[(addr & 0x1000) >> 12][addr & 0x0FFF] = data;
    }
    else if (addr >= 0x2000 && addr <= 0x3EFF)
    {
        addr &= 0x0FFF;
        if (cartridge->mirror == Cartridge::MIRROR::VERTICAL)
        {
            // Vertical
            if (addr >= 0x0000 && addr <= 0x03FF)
                nameTable[0][addr & 0x03FF] = data;
            if (addr >= 0x0400 && addr <= 0x07FF)
                nameTable[1][addr & 0x03FF] = data;
            if (addr >= 0x0800 && addr <= 0x0BFF)
                nameTable[0][addr & 0x03FF] = data;
            if (addr >= 0x0C00 && addr <= 0x0FFF)
                nameTable[1][addr & 0x03FF] = data;
        }
        else if (cartridge->mirror == Cartridge::MIRROR::HORIZONTAL)
        {
            // Horizontal
            if (addr >= 0x0000 && addr <= 0x03FF)
                nameTable[0][addr & 0x03FF] = data;
            if (addr >= 0x0400 && addr <= 0x07FF)
                nameTable[0][addr & 0x03FF] = data;
            if (addr >= 0x0800 && addr <= 0x0BFF)
                nameTable[1][addr & 0x03FF] = data;
            if (addr >= 0x0C00 && addr <= 0x0FFF)
                nameTable[1][addr & 0x03FF] = data;
        }
    }
    else if (addr >= 0x3F00 && addr <= 0x3FFF)
    {
        addr &= 0x001F;
        if (addr == 0x0010) addr = 0x0000;
        if (addr == 0x0014) addr = 0x0004;
        if (addr == 0x0018) addr = 0x0008;
        if (addr == 0x001C) addr = 0x000C;
        palleteTable[addr] = data;
    }
}
```

Rysunek 24: Odczyt/zapis danych przez PPU

Różne funkcje pomocnicze również zostały zdefiniowane: `GetScreen`, `GetColour`, `reset`, `ConnectCartridge`.

```
void PPU2C02::reset()
{
    fine_x          = 0x00;
    address_latch   = 0x00;
    ppu_data_buffer = 0x00;
    scanline        = 0;
    cycle           = 0;
    nexttile_id     = 0x00;
    nexttile_attrib = 0x00;
    nexttile_lsb    = 0x00;
    nexttile_msb    = 0x00;
    pattern_lo      = 0x0000;
    pattern_hi      = 0x0000;
    attrib_lo       = 0x0000;
    attrib_hi       = 0x0000;
    status.reg      = 0x00;
    mask.reg        = 0x00;
    control.reg     = 0x00;
    vram_addr.reg   = 0x0000;
    tram_addr.reg   = 0x0000;
}
```

```
// Connect cartridge to the PPU
void PPU2C02::ConnectCartridge(const std::shared_ptr<Cartridge>& cartridge)
{
    this->cartridge = cartridge;
}
```

Rysunek 25: Niektóre funkcje pomocnicze PPU

Funkcja zegarowa PPU jest bardzo rozbudowana. Z tego powodu nie zostanie przedstawiona w całości, a jedynie fragmentami.

Proces zegarowy rozpoczyna się od przewijania tła. Pola tła, poziome i pionowe są inkrementowane i zachowane.

```
auto IncrementScrollX = [&]()
{
    if (mask.render_background || mask.render_sprites)
    {
        if (vram_addr.coarse_x == 31)
        {
            vram_addr.coarse_x = 0;
            vram_addr.nametable_x = ~vram_addr.nametable_x;
        }
        else
            vram_addr.coarse_x++;
    }
};

auto TransferAddressX = [&]()
{
    if (mask.render_background || mask.render_sprites)
    {
        vram_addr.nametable_x = tram_addr.nametable_x;
        vram_addr.coarse_x = tram_addr.coarse_x;
    }
};
```

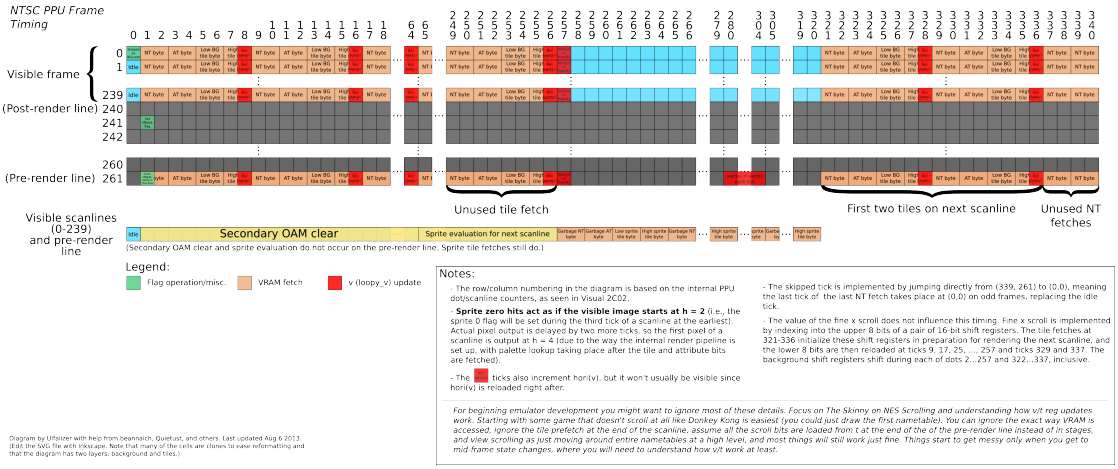
Rysunek 26: Inkrementacja i zachowanie poziomych pól tła

W kolejnym etapie ładowane są kolejne pola, ich wzory i atrybuty. Po tym przerzutniki tła, wzorców i atrybutów, są przesuwane.

```
auto LoadBackgroundShifters = [&]()
{
    pattern_lo  = (pattern_lo & 0xFF00) | nexttile_lsb;
    pattern_hi  = (pattern_hi & 0xFF00) | nexttile_msb;

    attrib_lo   = (attrib_lo & 0xFF00) | ((nexttile_attrib & 0b01) ? 0xFF : 0x00);
    attrib_hi   = (attrib_hi & 0xFF00) | ((nexttile_attrib & 0b10) ? 0xFF : 0x00);
};

auto UpdateShifters = [&]()
{
    if (mask.render_background)
    {
        pattern_lo <<= 1;
        pattern_hi <<= 1;
        attrib_lo <<= 1;
        attrib_hi <<= 1;
    }

    if (mask.render_sprites && cycle >= 1 && cycle < 258)
    {
        for (uint8_t i = 0; i < sprite_count; i++)
        {
            if (spriteScanline[i].x > 0)
            {
                spriteScanline[i].x--;
            }
            else
            {
                sprite_shifter_pattern_lo[i] <<= 1;
                sprite_shifter_pattern_hi[i] <<= 1;
            }
        }
    }
};
```

Rysunek 27: Operacje na przerzutnikach tła

Kolejne operacje emulują proces renderowania. Wykonywana czynność zależy od aktualnie renderowanej linii oraz cyklu zegarowego. Cały proces widać poniżej.



Rysunek 28: Renderowanie PPU

Przykładowo, dla niewidocznych linii (po linii 240):

```
if (scanline >= 241 && scanline < 261)
{
    if (scanline == 241 && cycle == 1)
    {
        status.vertical_blank = 1;
        if (control.enable_nmi)
            nmi = true;
    }
}
```

Rysunek 29: Przykładowe renderowanie zależne od renderowanej linii PPU

Kolejnym etapem jest sprawdzenie czy PPU ma wyświetlać tło i obiekty, a jeżeli tak, to odczytanie ich atrybutów.

```
uint8_t bg_pixel    = 0x00;
uint8_t bg_palette  = 0x00;

if (mask.render_background)
{
    uint16_t bit_mux = 0x8000 >> fine_x;

    uint8_t p0_pixel = (pattern_lo & bit_mux) > 0;
    uint8_t p1_pixel = (pattern_hi & bit_mux) > 0;
    bg_pixel = (p1_pixel << 1) | p0_pixel;

    uint8_t bg_pal0 = (attrib_lo & bit_mux) > 0;
    uint8_t bg_pal1 = (attrib_hi & bit_mux) > 0;
    bg_palette = (bg_pal1 << 1) | bg_pal0;
}
```

```
uint8_t fg_pixel = 0x00;
uint8_t fg_palette = 0x00;
uint8_t fg_priority = 0x00;

if (mask.render_sprites)
{
    sprite0HitRendered = false;

    for (uint8_t i = 0; i < sprite_count; i++)
    {
        if (spriteScanline[i].x == 0)
        {
            uint8_t fg_pixel_lo = (sprite_shifter_pattern_lo[i] & 0x80) > 0;
            uint8_t fg_pixel_hi = (sprite_shifter_pattern_hi[i] & 0x80) > 0;
            fg_pixel = (fg_pixel_hi << 1) | fg_pixel_lo;

            fg_palette = (spriteScanline[i].attribute & 0x03) + 0x04;
            fg_priority = (spriteScanline[i].attribute & 0x20) == 0;

            if (fg_pixel != 0)
                if (i == 0)
                    sprite0HitRendered = true;

                break;
            }
        }
}
```

Rysunek 30: Odczytanie atrybutów renderowanych grafik

Przedostatni etap renderowania, czyli uzyskanie ostatecznego piksela. Sprawdzenie czy w danym miejscu należy narysować nic, tło, obiekt, czy może obydwa - a w takim wypadku sprawdzenie czy możliwy jest “sprite 0 hit” - kolizja.

```
uint8_t pixel = 0x00;
uint8_t palette = 0x00;

if (bg_pixel == 0 && fg_pixel == 0)
{
    pixel = 0x00;
    palette = 0x00;
}
else if (bg_pixel == 0 && fg_pixel > 0)
{
    pixel = fg_pixel;
    palette = fg_palette;
}
else if (bg_pixel > 0 && fg_pixel == 0)
{
    pixel = bg_pixel;
    palette = bg_palette;
}
else if (bg_pixel > 0 && fg_pixel > 0)
{
    if (fg_priority)
    {
        pixel = fg_pixel;
        palette = fg_palette;
    }
    else
    {
        pixel = bg_pixel;
        palette = bg_palette;
    }

    if (sprite0HitPossible && sprite0HitRendered)
    {
        if (mask.render_background && mask.render_sprites)
        {
            if (~(mask.render_background_leftmost | mask.render_sprites_leftmost))
            {
                if (cycle >= 9 && cycle < 258)
                {
                    status.sprite_0_hit = 1;
                }
            }
            else
            {
                if (cycle >= 1 && cycle < 258)
                {
                    status.sprite_0_hit = 1;
                }
            }
        }
    }
}
```

Rysunek 31: Obliczenie finalnego piksela

I ostatnim krokiem jest wyrysowanie gotowego piksela. Po tym ilość cykli jest inkrementowana i sprawdzany jest warunek końcowy - określona liczba cykli. Flaga `frame_complete` używana jest w pętli do tworzenia klatki - ustawienie jej zatrzymuje generowanie klatki i umożliwia utworzenie następnej.

```
screen.SetPixel(cycle - 1, scanline, GetColour(palette, pixel));

cycle++;
if (cycle >= 341)
{
    cycle = 0;
    scanline++;
    if (scanline >= 261)
    {
        scanline = -1;
        frame_complete = true;
    }
}
```

Rysunek 32: Obliczenie finalnego piksela

2.3 Emulacja APU

Emulacja logiki dźwięku, fizycznie będącej częścią procesora 2A03. Tutaj została zaimplementowana osobno w celu poprawy czytelności kodu.

W celu implementacji dźwięku należało przygotować struktury reprezentujące sekwencję dźwiękową. Dla odpowiednich kanałów pulsacyjnych należało przygotować licznik długości, obwiednie głośności, przemiatanie częstotliwości oraz reprezentację samego kanału. Kanał szumu korzystał z tych samych struktur, wymagał jedynie dodatkowego rejestru przesuwającego.

```
// Freq sweeper
struct sweeper
{
    bool        enabled = false;
    bool        down    = false;
    bool        reload  = false;
    uint8_t     shift   = 0x00;
    uint8_t     timer   = 0x00;
    uint8_t     period  = 0x00;
    uint16_t    change  = 0x0000;
    bool        mute    = false;

    void track(uint16_t& target)
    {
        if (enabled)
        {
            change = target >> shift;
            mute = (target < 8) || (target > 0x07FF);
        }
    }

    bool clock(uint16_t& target, bool channel)
    {
        bool changed = false;
        if (timer == 0 && enabled && shift > 0 && !mute)
        {
            if (target >= 8 && change < 0x07FF)
            {
                if (down)
                    target -= change - channel;
                else
                    target += change;
                changed = true;
            }
        }

        if (timer == 0 || reload)
        {
            timer = period;
            reload = false;
        }
        else
            timer--;

        mute = (target < 8) || (target > 0x07FF);

        return changed;
    }
};
```

```
// Volume envelope
struct envelope
{
    void clock(bool bLoop)
    {
        if (!start)
        {
            if (divider_count == 0)
            {
                divider_count = volume;

                if (decay_count == 0)
                    if (bLoop)
                        decay_count = 15;
                else
                    decay_count--;
            }
            else
                divider_count--;
        }
        else
        {
            start = false;
            decay_count = 15;
            divider_count = volume;
        }
        if (disable)
            output = volume;
        else
            output = decay_count;
    }

    bool        start           = false;
    bool        disable         = false;
    uint16_t    divider_count   = 0x0000;
    uint16_t    volume          = 0x0000;
    uint16_t    output          = 0x0000;
    uint16_t    decay_count     = 0x0000;
};
```

Rysunek 33: Implementacja przemiatania częstotliwości oraz obwiedni dźwięku

```cpp
// Sequencer
struct sequencer
{
    uint32_t    new_sequence    = 0x00000000;
    uint32_t    sequence        = 0x00000000;
    uint16_t    timer           = 0x0000;
    uint16_t    reload          = 0x0000;
    uint8_t     output          = 0x00;

    uint8_t     clock(bool bEnable, std::function<void(uint32_t & s)> funcManip)
    {
        if (bEnable)
        {
            timer--;
            if (timer == 0xFFFF);
            {
                timer = reload;
                funcManip(sequence);
                output = sequence & 0x00000001;
            }
        }
        return output;
    }
};
```

```cpp
// Pulsewave signal
struct oscpulse
{
    double  frequency   = 0;
    double  dutycycle   = 0;
    double  amplitude   = 1;
    double  pi          = 3.14159;
    double  harmonics   = 20;

    double  sample(double t)
    {
        double  a = 0;
        double  b = 0;
        double  p = dutycycle * 2.0 * pi;

        auto approxsin = [](float t)
        {
            float j = t * 0.15915;
            j = j - (int)j;
            return 20.785 * j * (j - 0.5) * (j - 1.0f);
        };

        for (double n = 1; n < harmonics; n++)
        {
            double c = n * frequency * 2.0 * pi * t;
            a += -approxsin(c) / n;
            b += -approxsin(c - p * n) / n;
        }

        return (2.0 * amplitude / pi) * (a - b);
    }
};
```

Rysunek 34: Implementacja sekwencji dźwiękowej oraz kanału pulsacyjnego

W implementacji sygnału pulsacyjnego wykonano również aproksymację funkcji sinus - pozostawienie obliczenia funkcji sinus procesorowi było zbyt kosztowne obliczeniowo.

Zdefiniowano trzy kanały: dwa pulsacyjne i kanał szumu. Rezultat był wystarczająco dobry.

```cpp
sequencer       pulse1_seq;
oscpulse        pulse1_osc;
envelope        pulse1_env;
sweeper         pulse1_sweep;
lengthcounter   pulse1_lc;
bool            pulse1_enable   = false;
bool            pulse1_halt     = false;
double          pulse1_sample   = 0.0;
double          pulse1_output   = 0.0;

sequencer       pulse2_seq;
oscpulse        pulse2_osc;
envelope        pulse2_env;
sweeper         pulse2_sweep;
lengthcounter   pulse2_lc;
bool            pulse2_enable   = false;
bool            pulse2_halt     = false;
double          pulse2_sample   = 0.0;
double          pulse2_output   = 0.0;

sequencer       noise_seq;
envelope        noise_env;
lengthcounter   noise_lc;
bool            noise_enable    = false;
bool            noise_halt      = false;
double          noise_sample    = 0.0;
double          noise_output    = 0.0;
```

Rysunek 35: Zdefiniowanie kanałów audio

Utworzono obsługę zapisu z procesora. Odczyt nie był wymagany, APU nie wpływa na inne elementy systemu.

```
void APU2A03::write(uint16_t address, uint8_t data)
{
    switch (address)
    {
    case 0x4000:
        switch ((data & 0xC0) >> 6)
        {
        case 0x00:
            pulse1_seq.new_sequence = 0b01000000;
            pulse1_osc.dutycycle    = 0.125;
            break;
        case 0x01:
            pulse1_seq.new_sequence = 0b01100000;
            pulse1_osc.dutycycle    = 0.250;
            break;
        case 0x02:
            pulse1_seq.new_sequence = 0b01111000;
            pulse1_osc.dutycycle    = 0.500;
            break;
        case 0x03:
            pulse1_seq.new_sequence = 0b10011111;
            pulse1_osc.dutycycle    = 0.750;
            break;
        }
        pulse1_seq.sequence = pulse1_seq.new_sequence;
        pulse1_halt         = (data & 0x20);
        pulse1_env.volume   = (data & 0x0F);
        pulse1_env.disable  = (data & 0x10);
        break;
    case 0x4001:
        pulse1_sweep.enabled    = data & 0x80;
        pulse1_sweep.period     = (data & 0x70) >> 4;
        pulse1_sweep.down       = data & 0x08;
        pulse1_sweep.shift      = data & 0x07;
        pulse1_sweep.reload     = true;
        break;
    case 0x4002:
        pulse1_seq.reload = (pulse1_seq.reload & 0xFF00) | d
        break;
```

Rysunek 36: Fragment obsługi zapisu z procesora do APU

Funkcja zegarowa w APU ma dużo bardzo konkretnych wartości. Dźwięk jest zmieniany tylko w określonych interwałach. Obliczenie odpowiedniej liczby cykli umożliwia zachowanie rytmu.

```
void APU2A03::clock()
{
    bool    quarter = false;
    bool    half    = false;

    globalTime += (0.3333333333 / 1789773);

    if (clock_count % 6 == 0)
    {
        frame_clock_count++;

        if (frame_clock_count == 3729)
        {
            quarter = true;
        }
        if (frame_clock_count == 7457)
        {
            quarter = true;
            half        = true;
        }
        if (frame_clock_count == 11186)
        {
            quarter = true;
        }
        if (frame_clock_count == 14916)
        {
            quarter = true;
            half        = true;
            frame_clock_count   = 0;
        }
```

Rysunek 37: Flagi wyznaczające rytm

```
// Updates volume envelopes
if (quarter)
{
    pulse1_env.clock(pulse1_halt);
    pulse2_env.clock(pulse2_halt);
    noise_env.clock(noise_halt);
}
// Updates note length and frequency sweeper
if (half)
{
    pulse1_lc.clock(pulse1_enable, pulse1_halt);
    pulse2_lc.clock(pulse2_enable, pulse2_halt);
    noise_lc.clock(noise_enable, noise_halt);

    pulse1_sweep.clock(pulse1_seq.reload, 0);
    pulse2_sweep.clock(pulse2_seq.reload, 1);
}
```

Rysunek 38: Zmiany zależne od rytmu

W kolejnym etapie aktualizowane są rejestry fali pulsacyjnej i rejestr przesuwający szumu. Obliczany jest też ich dźwięk wyjściowy.

```
pulse1_seq.clock(pulse1_enable, [](uint32_t& s)
    {
        s = ((s & 0x0001) << 7) | ((s & 0x00FE) >> 1);
    });

pulse1_osc.frequency = 1789773.0 / (16.0 * (double)(pulse1_seq.reload + 1));
pulse1_osc.amplitude = (double)(pulse1_env.output - 1) / 16.0;
pulse1_sample = pulse1_osc.sample(globalTime);

if (pulse1_lc.counter > 0 && pulse1_seq.timer >= 8 && !pulse1_sweep.mute && pulse1_env.output > 2)
    pulse1_output += (pulse1_sample - pulse1_output) * 0.5;
else
    pulse1_output = 0;

pulse2_seq.clock(pulse2_enable, [](uint32_t& s)
    {
        s = ((s & 0x0001) << 7) | ((s & 0x00FE) >> 1);
    });

pulse2_osc.frequency = 1789773.0 / (16.0 * (double)(pulse2_seq.reload + 1));
pulse2_osc.amplitude = (double)(pulse2_env.output - 1) / 16.0;
pulse2_sample = pulse2_osc.sample(globalTime);

if (pulse2_lc.counter > 0 && pulse2_seq.timer >= 8 && !pulse2_sweep.mute && pulse2_env.output > 2)
    pulse2_output += (pulse2_sample - pulse2_output) * 0.5;
else
    pulse2_output = 0;


noise_seq.clock(noise_enable, [](uint32_t& s)
    {
        s = (((s & 0x0001) ^ ((s & 0x0002) >> 1)) << 14) | ((s & 0x7FFF) >> 1);
    });

if (noise_lc.counter > 0 && noise_seq.timer >= 8)
    noise_output = (double)noise_seq.output * ((double)(noise_env.output - 1) / 16.0);

if (!pulse1_enable) pulse1_output = 0;
if (!pulse2_enable) pulse2_output = 0;
if (!noise_enable)  noise_output = 0;
```

Rysunek 39: Aktualizacje dźwięku

Powyższe operacje dzieją się jedynie co 6 cykl zegarowy - 6 razy wolniej niż PPU. Przez resztę cykli działanie APU jest pomijane, z różnicą jedynie dla funkcji przemiatania, która zyskuje dokładności przez wywoływanie jej jak najczęściej.

```
pulse1_sweep.track(pulse1_seq.reload);
pulse2_sweep.track(pulse2_seq.reload);

clock_count++;
```

Rysunek 40: Funkcje wywoływane co cykl zegarowy

2.4 Emulacja głównej magistrali

Główna magistrala CPU, do której podłączone są wszystkie elementy systemu - niektóre z dostępem pośrednim, przez PPU. Dodatkowo przez magistralę przekazywany jest generowany dźwięk.

```
// CPU, PPU and APU
CPU6502 cpu;
PPU2C02 ppu;
APU2A03 apu;

// Pointer to "cartridge" - *.nes iNES file format ROM
std::shared_ptr<Cartridge> cartridge;

// CPU ram, with a total of 2kB!
uint8_t cpuRam[2048];

// Controllers - NES can use two controllers
uint8_t controller[2];

// Calculated audio, used by engine
double  audioOutput = 0.0;
```

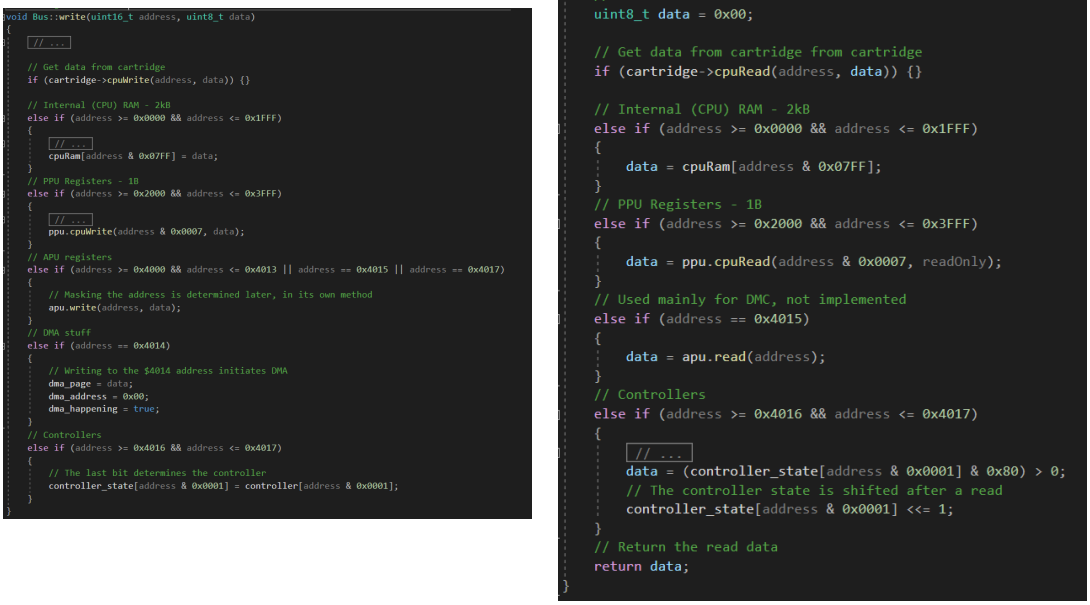
Rysunek 41: Komponenty wpięte do magistrali CPU

Ponadto, magistrala przechowuje dodatkowe informacje o dźwięku - w celu synchronizacji, stanie kontrolerów i DMA przeprowadzanym przez OAM.

```
// Audio constants
double  audioTimeSample = 0.0f;
double  audioTimeClock  = 0.0;
// ...and variables
double  audioTime       = 0.0;

// Controller state is 8 bit - 1 for each button
uint8_t controller_state[2];

// Direct Memory Access - transferring data from CPU memory into the OAM memory
uint8_t dma_page    = 0x00;
uint8_t dma_address = 0x00;
uint8_t dma_data    = 0x00;
// Flag to determine whether DMA transfer is happening
bool    dma_happening = false;
// Flag used to skip odd number of cycles,
// as CPU has to be on an even number of cycles
bool    dma_odd = true;

// Counting clock cycles to get CPU, PPU and APU to work synchronously
uint32_t sysClockCount = 0;
```

Rysunek 42: Komponenty wpięte do magistrali CPU

Zapis i odczyt przechodzący przez magistralę.



```
void Bus::write(uint16_t address, uint8_t data)
{
    // ...

    // Get data from cartridge
    if (cartridge->cpuWrite(address, data)) {}

    // Internal (CPU) RAM - 2kB
    else if (address >= 0x0000 && address <= 0x1FFF)
    {
        // ...
        cpuRam[address & 0x07FF] = data;
    }
    // PPU Registers - 1B
    else if (address >= 0x2000 && address <= 0x3FFF)
    {
        // ...
        ppu.cpuWrite(address & 0x0007, data);
    }
    // APU registers
    else if (address >= 0x4000 && address <= 0x4013 || address == 0x4015 || address == 0x4017)
    {
        // Masking the address is determined later, in its own method
        apu.write(address, data);
    }
    // DMA stuff
    else if (address == 0x4014)
    {
        // Writing to the $4014 address initiates DMA
        dma_page = data;
        dma_address = 0x00;
        dma_happening = true;
    }
    // Controllers
    else if (address >= 0x4016 && address <= 0x4017)
    {
        // The last bit determines the controller
        controller_state[address & 0x0001] = controller[address & 0x0001];
    }
}
```

```
// Reading from the bus
uint8_t Bus::read(uint16_t address, bool readOnly)
{
    // Variable to store the read data
    uint8_t data = 0x00;

    // Get data from cartridge from cartridge
    if (cartridge->cpuRead(address, data)) {}

    // Internal (CPU) RAM - 2kB
    else if (address >= 0x0000 && address <= 0x1FFF)
    {
        data = cpuRam[address & 0x07FF];
    }
    // PPU Registers - 1B
    else if (address >= 0x2000 && address <= 0x3FFF)
    {
        data = ppu.cpuRead(address & 0x0007, readOnly);
    }
    // Used mainly for DMC, not implemented
    else if (address == 0x4015)
    {
        data = apu.read(address);
    }
    // Controllers
    else if (address >= 0x4016 && address <= 0x4017)
    {
        // ...
        data = (controller_state[address & 0x0001] & 0x80) > 0;
        // The controller state is shifted after a read
        controller_state[address & 0x0001] <<= 1;
    }
    // Return the read data
    return data;
}
```

Rysunek 43: Zapis i odczyt, magistrala CPU

Funkcje dodatkowe, ułatwiające pracę z magistralą - ustawienie stałych dźwiękowych do synchronizacji audio, załadowanie kartridża i reset emulatora.

```
// Sets the audio constants
void Bus::SetSamplingFrequency(uint32_t sample_rate)
{
    // Time determined by sample
    audioTimeSample = 1.0 / (double)sample_rate;
    // Time determined by clock
    audioTimeClock = 1.0 / 5369318.0;
}

// Loads the cartridge
void Bus::insertCartridge(const std::shared_ptr<Cartridge>& cartridge)
{
    this->cartridge = cartridge;
    ppu.ConnectCartridge(cartridge);
}
```

```
// Resets the bus devices
void Bus::reset()
{
    cartridge->reset();
    cpu.reset();
    ppu.reset();

    dma_page        = 0x00;
    dma_address     = 0x00;
    dma_data        = 0x00;
    dma_odd         = true;
    dma_happening   = false;

    sysClockCount = 0;
}
```

Rysunek 44: Zapis i odczyt, magistrala CPU

Funkcja zegarowa magistrali - serce emulatora. Jest to główna funkcja, która wykonywana jest w pętli przez cały czas trwania emulacji.

```
// A single clock cycle on the bus
bool Bus::clock()
{
    // PPU is the fastest clock in the system! It's clocked every time
    ppu.clock();

    // APU is 6 times slower than PPU.
    // However, in this implementation, it updates frequency sweep every clock cycle
    apu.clock();

    // CPU is clocked 3 times slower than PPU - so clock it every third global clock
    if (sysClockCount % 3 == 0)
    {
        // Check if a DMA transfer is in progress
        if (dma_happening) { ... }
        // DMA is not in progress - CPU is not blocked, it clocks normally
        else
        {
            cpu.clock();
        }
    }
```

Rysunek 45: Główna funkcja zegarowa emulatora

Jednak, jeżeli wykonuje się transfer DMA, procesor jest zawieszony i jego funkcja nie wykonuje się.

```
// Check if a DMA transfer is in progress
if (dma_happening)
{
    // If so, wait one clock cycle - because they are on an odd count
    if (dma_odd)
    {
        // Check if they really are
        if (sysClockCount % 2 == 1)
        {
            // Yes - so the next cycle will be even
            dma_odd = false;
        }
    }
    else
    {
        // DMA consists of read and write
        // Reading happens off the CPU from the given $xx00-$xxFF range, byte at a time
        if (sysClockCount % 2 == 0)
        {
            // So - read 8 bytes from CPU RAM
            dma_data = read(dma_page << 8 | dma_address);
        }
        // Writing is to the PPU OAM memory, also byte at a time
        else
        {
            // So - write 8 bytes to the OAM memory of the PPU
            ppu.ppuOAM[dma_address] = dma_data;
            // And go to the next address
            dma_address++;

            // If value has overflown, DMA is finished
            if (dma_address == 0x00)
            {
                // Set the flags
                dma_happening   = false;
                // Odd flag to true, so it can be checked next time a DMA happens
                dma_odd         = true;
            }
        }
    }
}
```

Rysunek 46: Wywołanie funkcji DMA

Ostatnią rzeczą do wykonania jest synchronizacja audio i wywołanie przerwania niemaskowalnego, jeżeli wystąpiła taka potrzeba.

Synchronizacja audio polega na dodaniu czasu jednego cyklu zegarowego ($\frac{f_{APU}}{60}$) do sumy i sprawdzeniu czy zgadza się z częstotliwością próbkowania. Jeżeli tak, dźwięk jest gotowy do odtworzenia i przekazywany do silnika w funkcji głównej.

```
// Synchronize the audio
bool isAudioReady   = false;
// Add single clock operation time to the audio time
audioTime   += audioTimeClock;
// If the audio time exceeds the note time
if (audioTime >= audioTimeSample)
{
    // Subtract the note time from the audio time
    audioTime       -=  audioTimeSample;
    // And put the next note in
    audioOutput     =   apu.MixOutputs();
    // Audio is ready!
    isAudioReady    =   true;
}

// Check if non-masked interrupt is supposed to happen
if (ppu.nmi)
{
    ppu.nmi = false;
    // If so, execute it
    cpu.nmi();
}

// Add a clock count
sysClockCount++;

// Return true if audio is ready
return isAudioReady;
```

Rysunek 47: Synchronizacja audio i przerwanie niemaskowalne

2.5 Emulacja mapperów

Mappery używane w grach na NESa były bardzo różne. Ich działanie było jednak identyczne i polegało na tym samym - przekierowywaniu linii adresowych na różne banki pamięci. Metody i stałe były więc takie same w każdym mapperze.

```
    virtual bool cpuMapRead(uint16_t addr, uint32_t& mapped_addr)   = 0;
    virtual bool cpuMapWrite(uint16_t addr, uint32_t& mapped_addr)  = 0;
    virtual bool ppuMapRead(uint16_t addr, uint32_t& mapped_addr)   = 0;
    virtual bool ppuMapWrite(uint16_t addr, uint32_t& mapped_addr)  = 0;

    virtual void reset() = 0;


protected:
    uint8_t PRGSize = 0;
    uint8_t CHRSize = 0;
```

Rysunek 48: Metody i stałe mapperów

Zaimplementowany został jedynie mapper 0, znany też jako NROM. Dodanie większej liczby mapperów nie jest trudne, ale jednak czasochłonne - w celu szybszego ukończenia projektu zatrzymano się na jednym.

```
// Transform CPU address to PRG ROM offset
bool Mapper_000::cpuMapRead(uint16_t address, uint32_t& mapped_address)
{
    if (address >= 0x8000 && address <= 0xFFFF)
    {
        mapped_address = address & (PRGSize > 1 ? 0x7FFF : 0x3FFF);
        return true;
    }

    return false;
}

// Transform CPU address to PRG ROM offset
bool Mapper_000::cpuMapWrite(uint16_t address, uint32_t& mapped_address)
{
    if (address >= 0x8000 && address <= 0xFFFF)
    {
        mapped_address = address & (PRGSize > 1 ? 0x7FFF : 0x3FFF);
        return true;
    }

    return false;
}
```

```
// Transform PPU address to CHR ROM offset
bool Mapper_000::ppuMapRead(uint16_t address, uint32_t& mapped_address)
{
    if (address >= 0x0000 && address <= 0x1FFF)
    {
        mapped_address = address;
        return true;
    }

    return false;
}

// Transform PPU address to CHR ROM offset
bool Mapper_000::ppuMapWrite(uint16_t address, uint32_t& mapped_address)
{
    if (address >= 0x0000 && address <= 0x1FFF)
    {
        if (CHRSize == 0)
        {
            mapped_address = address;
            return true;
        }
    }
    return false;
}
```

Rysunek 49: Implementacja mappera NROM

2.6 Emulacja kartridży

Zawartość kartridży - pamięć programu i grafiki - jest wczytywana do wektorów. Flaga `valid` określa, czy kartridż został poprawnie odczytany. Określone są też zmienne dot. użytego mappera.

```
    // Which way to mirror nametables - or, to scroll
    enum MIRROR
    {
        HORIZONTAL,
        VERTICAL,
        ONESCREEN_LO,
        ONESCREEN_HI
    };

    MIRROR  mirror = HORIZONTAL;

private:

    // Flag to set when ROM was succesfuly read
    bool valid = false;

    // Memory containing program
    uint8_t PRGSize     = 0;
    std::vector<uint8_t> PRGMemory;
    // Memory containing graphics
    uint8_t CHRSize     = 0;
    std::vector<uint8_t> CHRMemory;

    // Loaded mapper
    uint8_t mapperID    = 0;
    std::shared_ptr<Mapper> mapper;
```

Rysunek 50: Zmienne kartridży

Użyte ROMy są w formacie iNES. Struktura nagłówka tego formatu została zawarta w strukturze.

```
// iNES Header structure
struct header
{
    char    name[4];
    uint8_t prg_rom_size;
    uint8_t chr_rom_size;
    uint8_t mapper1;
    uint8_t mapper2;
    uint8_t prg_ram_size;
    uint8_t tv_system1;
    uint8_t tv_system2;
    char    unused[5];
};
```

Rysunek 51: Struktura nagłówka formatu iNES

ROMy odczytywane były w standardowy sposób - z nagłówka odczytano informację o pliku i odczytano jego resztę w oparciu o nie.

```
// Create a new file stream
std::ifstream ifstream;
// Open the file
ifstream.open(fileName, std::ifstream::binary);
// Try to read the file
if (ifstream.is_open())
{
    // Read header - for iNES it's 16 bytes
    ifstream.read((char*)&cartHeader, sizeof(cartHeader));

    // If trainer is present (bit 2 set), move 512 bytes forward
    if (cartHeader.mapper1 & 0x04)
        ifstream.seekg(512, std::ios_base::cur);

    // Get Mapper ID
    mapperID    = ((cartHeader.mapper2 >> 4) << 4) | (cartHeader.mapper1 >> 4);

    // Get nametable mirroring
    mirror      = (cartHeader.mapper1 & 0x01) ? VERTICAL : HORIZONTAL;

    // Get size of PRG ROM
    PRGSize = cartHeader.prg_rom_size;
    PRGMemory.resize(PRGSize * 16384);

    // Read PRG memory from ROM
    ifstream.read((char*)PRGMemory.data(), PRGMemory.size());

    // Get size of CHR ROM
    CHRSize = cartHeader.chr_rom_size;
    if (CHRSize == 0)
        CHRMemory.resize(8192);
    else
        CHRMemory.resize(CHRSize * 8192);
    // Read CHR memory from ROM
    ifstream.read((char*)CHRMemory.data(), CHRMemory.size());

    // Load mapper
    switch (mapperID)
    {
    case 0:
        mapper = std::make_shared<Mapper_000>(PRGSize, CHRSize);
        break;
    }

    // Cartridge read correctly
    valid = true;
    ifstream.close();
}
```

Rysunek 52: Odczyt ROMu

Metody dostępu do pamięci kartridża - zarówno programowej (przez CPU), jak i graficznej (przez PPU).

```cpp
// Transform CPU address to PRG ROM offset
bool Mapper_000::cpuMapRead(uint16_t address, uint32_t& mapped_address)
{
    if (address >= 0x8000 && address <= 0xFFFF)
    {
        mapped_address = address & (PRGSize > 1 ? 0x7FFF : 0x3FFF);
        return true;
    }

    return false;
}

// Transform CPU address to PRG ROM offset
bool Mapper_000::cpuMapWrite(uint16_t address, uint32_t& mapped_address)
{
    if (address >= 0x8000 && address <= 0xFFFF)
    {
        mapped_address = address & (PRGSize > 1 ? 0x7FFF : 0x3FFF);
        return true;
    }

    return false;
}
```

```cpp
// Transform PPU address to CHR ROM offset
bool Mapper_000::ppuMapRead(uint16_t address, uint32_t& mapped_address)
{
    if (address >= 0x0000 && address <= 0x1FFF)
    {
        mapped_address = address;
        return true;
    }

    return false;
}

// Transform PPU address to CHR ROM offset
bool Mapper_000::ppuMapWrite(uint16_t address, uint32_t& mapped_address)
{
    if (address >= 0x0000 && address <= 0x1FFF)
    {
        if (CHRSize == 0)
        {
            mapped_address = address;
            return true;
        }
    }
    return false;
}
```

Rysunek 53: Odczyt i zapis do pamięci kartridża

Funkcje pomocnicze kartridża - “wyjęcie” kartridża i sprawdzenie czy załadowany kartridż jest poprawny.

```cpp
// Reset cartridge
void Cartridge::reset()
{
    // If mapper is loaded, reset it
    if (mapper != nullptr)
        mapper->reset();
}
```

```cpp
// Return whtether cartridge is valid
bool Cartridge::IsValid()
{
    return valid;
}
```

Rysunek 54: Funkcje pomocnicze kartridża

2.7 Emulator

Celem projektu było wykonanie emulatora - i udało się to osiągnąć. Te przetworzone dane nie umożliwiają jednak odczytu przez człowieka, a co dopiero - gry. Jako że celem projektu nie było wykonanie silnika, który mógłby takie dane przetworzyć, posłużyłem się gotowym rozwiązaniem - silnikiem olcPixelGameEngine. To podejście pozwoliło mi na skupieniu się na tworzeniu poprawnego emulatora.

Obiekt Emulator przyjmuje jeden argument - ciąg znakowy, który jest ścieżką do odtwarzanego ROMu.

```
Emulator(std::string &filename)
{
    this->filename   = filename;
    sAppName         = "Emulator systemu NES";
}
```

Rysunek 55: Metoda tworzenia Emulatora

Posiada on jedynie obiekt klasy Bus - będący całym emulowanym systemem - i kartridż, pochodzący z zewnątrz.

```
Bus     nes;

// To start/stop emulation
bool    running = false;

// Pointer to cartridge
std::shared_ptr<Cartridge> cartridge;
std::string filename;

// Used to tie together timing of NES and audio
static  Emulator* instance;
```

Rysunek 56: Zawartość klasy Emulatora

Emulator nadpisuje po klasie silnika olcPixelGameEngine dwie główne metody - jedną uruchamianą przy rozpoczynaniu Emulatora i jedną przy rysowaniu nowych klatek.

```cpp
// OLC engine - create method, start up emulator
bool OnUserCreate() override
{
    // Load the cartridge
    cartridge = std::make_shared<Cartridge>(filename);

    // If can't load cartridge, finish here
    if (!cartridge->IsValid())
        return false;

    // Insert cartridge into NES
    nes.insertCartridge(cartridge);

    // Reset NES - set starting values
    nes.reset();

    instance = this;

    // Set audio consts
    nes.SetSamplingFrequency(44100);
    olc::SOUND::InitialiseAudio(44100, 1, 8, 512);
    olc::SOUND::SetUserSynthFunction(GetSound);

    return true;
}
```

```cpp
// OLC engine - update method
bool OnUserUpdate(float time)
{
    // Get input
    EmulatorUpdate(time);
    return true;
}
```

Rysunek 57: Nadpisane funkcje silnika

W klasie `OnUserCreate` została użyta funkcja do synchronizacji dźwięku z obrazem. Funkcja odpowiedzialna jest za wywoływanie sygnału zegarowego w systemie aż do uzyskania dźwięku - przekazuje go wtedy dalej, a sama funkcja wraca do wykonywania się w pętli.

```cpp
// Synchonizing NES clock with audio
static float GetSound(int channel, float time, float timeStep)
{
    // Do cycles until audio sample is ready
    while (!instance->nes.clock());

    // Return the prepared audio sample
    return static_cast<float>(instance->nes.audioOutput);
}
```

Rysunek 58: Synchronizacja dźwięku z obrazem

Ostatnią częścią klasy emulatora jest odczyt z kontrolera - tu klawiatury - i aktualizacja obrazu. Poza standardowymi przyciskami z NESa, opisano przycisk spacji który rozpoczyna i zatrzymuje emulację oraz przycisk R, który restartuje emulację.

```
// Main method - update emu
bool EmulatorUpdate(float time)
{
    // Clear screen
    Clear(olc::DARK_BLUE);

    // Get controller input
    nes.controller[0] = 0x00;
    nes.controller[0] |= GetKey(olc::Key::X).bHeld ? 0x80 : 0x00;       // A
    nes.controller[0] |= GetKey(olc::Key::Z).bHeld ? 0x40 : 0x00;       // B
    nes.controller[0] |= GetKey(olc::Key::A).bHeld ? 0x20 : 0x00;       // Select
    nes.controller[0] |= GetKey(olc::Key::S).bHeld ? 0x10 : 0x00;       // Start
    nes.controller[0] |= GetKey(olc::Key::UP).bHeld ? 0x08 : 0x00;      // UP
    nes.controller[0] |= GetKey(olc::Key::DOWN).bHeld ? 0x04 : 0x00;    // DOWN
    nes.controller[0] |= GetKey(olc::Key::LEFT).bHeld ? 0x02 : 0x00;    // LEFT
    nes.controller[0] |= GetKey(olc::Key::RIGHT).bHeld ? 0x01 : 0x00;   // RIGHT

    // Start/stop
    if (GetKey(olc::Key::SPACE).bPressed) running = !running;
    // Reset
    if (GetKey(olc::Key::R).bPressed) nes.reset();

    // Draw rendered output
    DrawSprite(0, 0, &nes.ppu.GetScreen(), 2);
    return true;
}
```

Rysunek 59: Odczyt danych wejściowych

Funkcja główna emulatora. Odczytuje podaną ścieżkę do ROMu, rozpoczyna emulator i tworzy okno z obrazem.

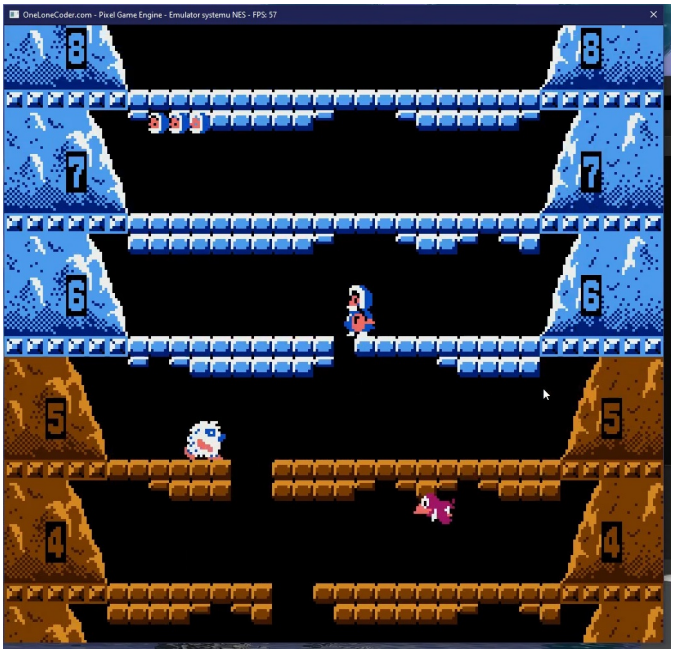
```
int main(int argc, char** argv)
{
    std::string filename = argv[1];
    Emulator emulator(filename);
    // Construct window (width, height, pixel_width, pixel_height, fullscreen, vsync)
    emulator.Construct(512, 480, 2, 2, 0, 1);
    // Start emulation
    emulator.Start();
    return 0;
}
```

Rysunek 60: Funkcja główna programu

3 Przykłady

W tej sekcji przedstawiono zrzuty ekranu podczas uruchamiania kilku przez emulator.

Rysunek 61: Uruchomianie gry Super Mario Bros. przez emulator



Rysunek 62: Uruchomianie gry Ice Climber przez emulator

Rysunek 63: Uruchomianie gry Donkey Kong przez emulator

Rysunek 64: Uruchomianie gry Soccer przez emulator

Rysunek 65: Uruchomianie gry Excitebike przez emulator

4 Podsumowanie

Emulator nie jest idealnym odwzorowaniem systemu NES. Cykle zegarowe nie są dokładne, emulator nie obsługuje wszystkich kanałów audio i liczba gier jest ograniczona przez zaimplementowanie tylko jednego typu mappera. Emulator pozwala jednak na przyjemną grę w obsługiwane tytuły, przy okazji dość wiernie odwzorowując logikę systemu.

Stworzenie emulatora systemu NES pozwoliło mi w bliskim stopniu zapoznać się ze sposobem działania komputera.
Potrzeba zaimplementowania całej logiki systemu - od kodów operacji procesora aż do wczytywania stanu kontrolerów do pamięci - pozwoliła mi lepiej zrozumieć logikę różnych operacji wykonywanych w systemie.
Zdefiniowanie rejestrów stanu - procesorów CPU, PPU, APU - pozwoliło mi dostrzec ich potencjał i ogromną rolę pełnioną przez nie w systemie.

Uważam emulację systemu NES za świetny sposób do nauki architektury systemów komputerowych. Mimo swojego wieku i oczywistych ograniczeń, schematy i sposoby radzenia sobie z problemami zachowane są w wielu mikroprocesorach do dziś.

Literatura

[1] NesDev Wiki, główne źródło
https://wiki.nesdev.com/

[2] MOSS 6502, Programing Manual
http://users.telenet.be/kim1-6502/6502/proman.html

[3] Dokumentacja procesora 6502
http://archive.6502.org/datasheets/rockwell_r650x_r651x.pdf

[4] Brad Taylor's 2A03 Technical Reference
http://nesdev.com/2A03%20technical%20reference.txt

[5] Brad Taylor's 2C02 Technical Reference
http://nesdev.com/2C02%20technical%20reference.TXT

[6] The skinny on NES scrolling - loopy
http://nesdev.com/loopyppu.zip

[7] javidx9 NES Emulator,
https://www.youtube.com/playlist?list=PLrOv9FMX8xJHqMvSGB_9G9nZZ_4IgteYf

[8] Errata nt. błędów i problemów z emulacją
http://wiki.nesdev.com/w/index.php/Errata

[9] Źródło obrazów kartridży
https://wowroms.com/en/roms/list/nintendo+entertainment+system