Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Bartosz Korczyński
Nr albumu: 371002

Tłumaczenie aplikacji przeznaczonych na procesor MOS 6502 na architekturę x86-64

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr. hab. inż. Marcina Peczarskiego
Instytut Informatyki

Warszawa, grudzień 2022

Streszczenie

Przedmiotem pracy jest opracowanie sposobu emulowania procesora MOS 6502 poprzez tłumaczenie skompilowanych programów na architekturę x86-64. W przeciwieństwie do klasycznej metody emulowania, czyli interpretacji rozkazów, rozwiązanie zaproponowane w niniejszej pracy magisterskiej jest bardziej wydajne. Praca jest podzielona na dwie części. Pierwsza (obejmująca rozdział 1. i 2) zawiera opis architektury procesora oraz krótkie wprowadzenie do tematu emulacji. W drugiej części (rozdziały 3. i 4.) znajduje się opis implementacji tłumacza powstałego w ramach pracy oraz porównanie działania z istniejącymi metodami emulacji. W rozdziale 4. zostały przedstawione wyniki pracy, czyli porównanie szybkości metody interpretowania oraz metody tłumaczenia i efekt przetłumaczenia gry przeznaczonej na prawdziwą konsolę do gier na platformę x86-64, potwierdzające postawioną w pracy tezę o wyższej wydajności translacji w porównaniu z klasycznymi emulatorami opartymi o interpretację rozkazów.

Słowa kluczowe

emulacja, MOS 6502, statyczna rekompilacja, x86-64

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

Software and its engineering → Software notations and tools → Compilers

Tytuł pracy w języku angielskim

Translation of applications for the MOS 6502 processor into the x86-64 architecture

Spis treści

Wprowadzenie

Kompilowanie to proces tłumaczenia kodu pewnego języka programowania na kod niższego poziomu. W przypadku kompilacji do kodu maszynowego programy kompilowane są na konkretną architekturę, co powoduje, że uruchomienie ich na innych systemach może nie być możliwe. Jednym z rozwiązań tego problemu jest ponowne skompilowanie programu, jednak wymaga to posiadania kodu źródłowego. Ten jednak nie zawsze jest dostępny — większość oprogramowania rozpowszechniana jest jedynie w formie plików wykonywalnych.

Problem braku kompatybilności może być rozwiązany przez emulację sprzętu na innych urządzeniach — głównie komputerach osobistych, ze względu na ich uniwersalność. Emulator to program, który potrafi symulować podzespoły innego systemu w taki sposób, by dla oprogramowania było nie do odróżnienia, czy jest uruchamiane na prawdziwym sprzęcie czy w emulowanym środowisku. Dla przykładu, dzięki emulatorowi konsoli Nintendo 64 możliwe jest granie w gry wydane na tę konsolę na swoim komputerze bez posiadania oryginalnego sprzętu. Emulacja nie jest jednak ograniczona wyłącznie do konsol do gier — jest to bowiem pojęcie odnoszące się do każdej sytuacji, w której symulowany jest inny system komputerowy. Jednak ze względu na szeroką bibliotekę gier wydanych na starsze konsole do gier, to właśnie emulacja konsol jest najbardziej popularna. Konsole bardzo rzadko oferują wsteczną kompatybilność z już wydanymi grami poprzednich generacji konsol, zatem emulacja z reguły stanowi jedyną możliwość uruchamiania starszych gier.

Emulacja całego urządzenia wymaga symulowania poszczególnych komponentów, takich jak procesor, układ graficzny czy układ dźwiękowy. Niniejsza praca magisterska skupia się na pierwszym elemencie. Istnieją dwie główne metody emulacji procesora: interpretacja rozkazów oraz translacja programów. Interpretery symulują instrukcje w trakcie wykonywania programu. Translacja polega na zamianie instrukcji architektury źródłowej na docelową, może być ona statyczna, gdy tłumaczenie odbywa się przed uruchomieniem programu, bądź dynamiczna, gdy kolejne instrukcje tłumaczone są w trakcie wykonywania programu. Interpretery są łatwiejsze w implementacji, lecz posiadają większy narzut wydajnościowy w porównaniu do przetłumaczonych programów.

Należy także zaznaczyć komercyjne zastosowanie emulacji. Firma Apple używa emulacji w swoim systemie operacyjnym jako jeden z kluczowych elementów. Program Rosetta służy do symulowania architektury PowerPC na architekturze Intel, a druga wersja Rosetta 2 jest używana do uruchamiania programów skompilowanych na platformę x86-64 na architekturze ARM. Aplikacja Rosetta jest wbudowana w system operacyjny macOS (dawniej OS X) i dla użytkownika końcowego jest w pełni transparentne, czy uruchamiana aplikacja została skompilowana natywnie, czy działa w emulatorze. Obie wersje systemu Rosetta używają bardziej wydajnej metody emulacji, czyli tłumaczenia programów, bowiem tutaj kluczowa jest szybkość działania i możliwie niski narzut wydajności.

Na przestrzeni lat powstało wiele procesorów o różnych architekturach. Jednym z najpopularniejszych układów w latach 70. i 80. poprzedniego wieku był MOS 6502. Ten 8-bitowy mikroprocesor został użyty we wczesnych komputerach osobistych, przemysłowych systemach

sterowania i przede wszystkim w konsolach do gier wideo. W sieci można znaleźć bardzo wiele emulatorów tego procesora, jednak są to interpretery instrukcji.

Celem niniejszej pracy magisterskiej jest napisanie tłumacza binarnych programów skompilowanych na procesor MOS 6502 na architekturę x86-64. Po otrzymaniu programu na wejściu tłumacz powinien utworzyć nowy plik wykonywalny, który będzie natywną aplikacją docelowej architektury, wykonującą oryginalny program. Takie rozwiązanie — jak zostanie wykazane w niniejszej pracy magisterskiej — jest wydajniejsze niż klasyczne emulatory oparte o interpretację rozkazów oraz prostsze w użyciu niż tradycyjny emulator (tłumaczenie generuje plik gotowy do uruchomienia, a użytkownik nie musi uczyć się obsługi samego emulatora).

Struktura dokumentu

Praca została podzielona na dwie części: pierwsza z nich (rozdziały 1. oraz 2.) zawiera ogólne informacje o procesorze MOS 6502 oraz o sposobach emulowania procesora. Rozdział 1. zaczyna się krótką historią procesora, a następnie omówiona jest architektura układu. Rozdział 2. przedstawia metody uruchamiania programów skompilowanych na inną architekturę: interpreter procesora, tłumaczenie programu oraz symulowanie oryginalnego układu scalonego. Pierwsza część pozwala czytelnikowi zrozumieć decyzje projektowe podjęte podczas implementowania tłumacza, przedstawione w kolejnych rozdziałach. Druga część (rozdziały 3. i 4.) opisuje wyniki tej pracy magisterskiej, czyli program tłumaczący i rezultaty jego działania. Składa się na nią rozdział 3. zawierający opis działania przygotowanego tłumacza i problemy napotkane podczas jego pisania. Rozdział 4. przedstawia wyniki tłumaczenia, w tym wydajność przetłumaczonych programów, porównanie z innymi metodami ich uruchamiania oraz użycie tłumacza w celu uruchomienia programów powstałych na konsolę Nintendo Entertainment System.

Rozdział 1

Opis procesora MOS 6502

1.1. Zarys historyczny

Ośmiobitowy procesor MOS 6502 został zaprojektowany i stworzony przez nieistniejącą już amerykańską firmę MOS Technology. Przedsiębiorstwo zostało założone w 1969 roku i specjalizowało się w wytwarzaniu mikroprocesorów. Największą popularność przyniosła tej firmie produkcja w latach 70. XX wieku czipów z rodziny 65xx, z których największe sukcesy odniosły 6510 oraz 6502, będący przedmiotem pracy magisterskiej.

Historia serii 65xx sięga roku 1975, gdy firmę Motorola opuściła grupa projektantów i dołączyła do MOS Technology ze swoim pomysłem stworzenia taniego i prostego mikroprocesora, a jednocześnie możliwie kompatybilnego z popularnym wówczas układem Motorola 6800. Efektem tej współpracy był MOS 6501 — projekt nieco podobny do 6800, ale dzięki kilku uproszczeniom 6501 był do 4 razy szybszy [12], a przy tym był oferowany na rynku w przystępnej cenie. Choć układ firmy MOS nie używał zestawu instrukcji wykorzystywanego w oryginalnym produkcie Motoroli, układ wyprowadzeń był kompatybilny z pierwowzorem, co było dużym ułatwieniem dla twórców komputerów w tamtych latach. To jednak spotkało się z szybką reakcją Motoroli, która wystąpiła z powództwem przeciwko MOS Technology, a w rezultacie — sprzedaż 6501 musiała zostać wstrzymana do czasu rozstrzygnięcia sporu na drodze sądowej. W jego miejsce od razu pojawił się mikroprocesor MOS 6502, który był bardzo podobny do poprzednika, ale miał inny układ wyprowadzeń, co pozwoliło na swobodne wprowadzenie procesora do sprzedaży.

Co interesujące, w podręczniku procesora znajduje się wiele porównań do Motoroli oraz opisane są różnice, tak aby projektanci sprzętu mogli łatwo przemigrować na układ firmy MOS Technology. Pomimo braku kompatybilności z istniejącymi czipami Motoroli, dzięki swojej niskiej cenie i prostocie procesor 6502 okazał się gigantycznym sukcesem i wraz z układem Zilog Z80 pozwolił na wprowadzenie na rynek wielu nowych, przystępnych komputerów osobistych.

Pierwszym komputerem, który używał nowego procesora, był Apple I wyprodukowany w 1976 roku, rok po premierze czipu. Rok później na rynku pojawiły się między innymi Commodore PET oraz Apple II. Pięć lat później do sprzedaży trafiła konsola Nintendo Entertainment System oraz nowa wersja komputera Commodore 64. Konsola ta wciąż używała tego samego mikroprocesora, a Commodore — zmodyfikowanej wersji MOS 6510 (procesor miał dodatkowy sześciobitowy port wejścia-wyjścia). Dzięki swojej popularności MOS 6502 jest — lekko zmienionej formie — produkowany do dzisiaj przez amerykańską firmę Western Design Center.

1.2. Architektura

Choć budowa wewnętrzna i architektura procesora nie są niezbędne do programowania ani do translacji skompilowanych programów, warto je omówić dla lepszego zrozumienia zagadnienia.

Procesor MOS 6502 operuje na ośmiobitowych słowach oraz szesnastobitowych adresach, co oznacza, że układ może adresować 64 KiB pamięci [2]. Mikroprocesor wyposażony jest w wewnętrzny zegar sterowany zewnętrznym generatorem drgań i w zależności od wersji może pracować z częstotliwością od 1 MHz do nawet 4 MHz [7]. Rozkazy zajmują od dwóch do siedmiu cykli procesora, zależnie od typu instrukcji, a głównie od liczby dostępów do pamięci potrzebnych do jej pobrania i wykonania. Procesor zawiera zaledwie trzy rejestry dostępne dla programisty: akumulator (rejestr A), rejestr X oraz rejestr Y, z których operacje arytmetyczne można wykonywać tylko na tym pierwszym, a pozostałe dwa rejestry służą jedynie do obsługi różnych trybów adresowania pamięci. Oprócz tego MOS 6502 ma niezbędne w każdym procesorze licznik instrukcji (PC: ang. *Program Counter*), wskaźnik stosu (SP: ang. *Stack Pointer*) oraz rejestr stanu procesora (SR: ang. *Status Register*) [1].

Akumulator jest na stałe połączony z jednostką arytmetyczno-logiczną (układem ALU) i jest jedynym rejestrem, na którym można wykonywać operacje dodawania, odejmowania oraz operacje bitowe. MOS 6502 nie ma sprzętowego wsparcia dla mnożenia i dzielenia. Rejestry `X` oraz `Y` są używane głównie w specjalnych trybach adresowania pamięci, ale dodatkowo można je wykorzystać jako tymczasowe komórki do przechowywania danych, ponieważ możliwe jest szybkie przesyłanie zawartości między rejestrami `A`, `X` oraz `Y`.

Licznik instrukcji jest jedynym rejestrem szesnastobitowym, a właściwie dwoma połączonymi rejestrami ośmiobitowymi, odpowiedzialnymi za górną i dolną połowę adresu, odpowiednio: PCH i PCL. Każda instrukcja odpowiednio zmienia rejestr PC, w przypadku skoków — na nowy adres, w przypadku pozostałych operacji — o tyle bajtów, ile zajmuje aktualnie wykonywany rozkaz.

Rejestr stanu procesora (SR) zawiera informacje widoczne w tabeli 1.1. Bit piąty nie jest używany, ale niektóre rozkazy procesora mogą zmienić jego wartość, dlatego przy emulowaniu istotne jest odtworzenie tych efektów, nawet jeśli większość aplikacji nie będzie na tym polegać.

Tabela 1.1: Rejestr stanu procesora

7	6	5	4	3	2	1	0
N	V	-	B	D	I	Z	C

- **N** — czy w ostatniej operacji arytmetycznej wynik był ujemny
- **V** — czy w ostatniej operacji arytmetycznej wystąpił nadmiar
- **B** — używany do odróżnienia instrukcji IRQ od BRK
- **D** — przełącza procesor w tryb operacji dziesiętnych (BCD)
- **I** — blokuje wywołanie procedury obsługi przerwań
- **Z** — czy w ostatniej operacji arytmetycznej wynik był zerowy
- **C** — czy w ostatniej operacji arytmetycznej wystąpiło przeniesienie

Niektóre wersje układów z rodziny 65xx wyposażone są w moduł operacji arytmetycznych w zapisie dziesiętnym kodowanym dwójkowo (kod BCD). Do przełączania się między trybami służą instrukcje `SED` oraz `CLD`, odpowiednio przełączające flagę `D` procesora. Tryb dziesiętny pozwala na łatwiejsze (dla programisty) wykonywanie operacji liczbowych w systemie dziesiętnym.

1.2.1. Zestaw instrukcji

Procesor MOS 6502 obsługuje 56 instrukcji. Niewielka liczba rozkazów oznacza, że programista może nauczyć się ich wszystkich na pamięć, co umożliwi mu szybkie pisanie programów. Pełna lista oficjalnych instrukcji wraz z opisem działania znajduje się w dodatku A. Sposób kodowania instrukcji przedstawiony jest w dodatku B.

1.2.2. Pamięć oraz stos

Procesor MOS 6502 ma szesnastobitową szynę adresową, co oznacza, że może adresować do 64 KiB pamięci. Twórcy komputerów korzystają z dowolności w zakresie sposobów podłączania pamięci do tej szyny. Niektóre adresy mogą na przykład prowadzić do pamięci tylko do odczytu, a inne być mapowane na urządzenia wejścia-wyjścia. W przypadku konsoli Nintendo Entertainment System mapowanie niektórych rejonów pamięci zależy także od aktualnie używanego kartrydża z grą.

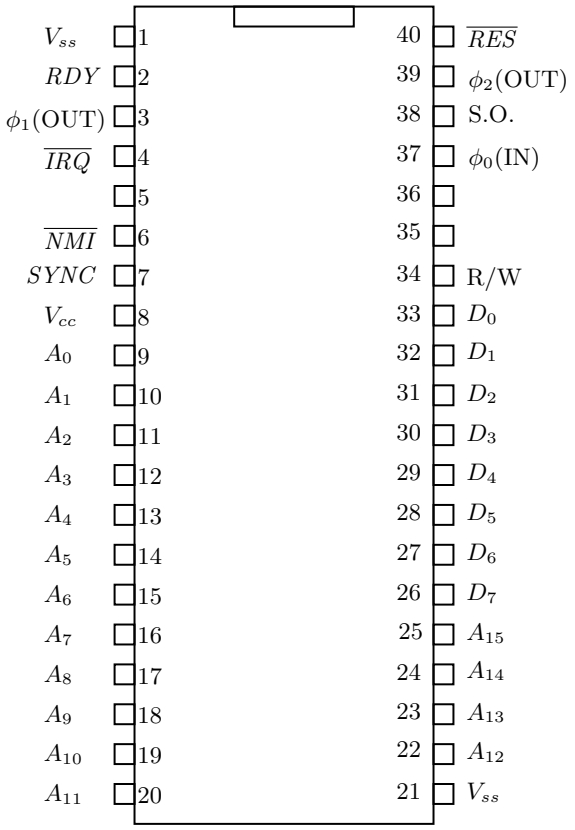
Stos znajduje się na *stronie pierwszej* (adresy 256–511), dlatego początkowe adresy powinny być zmapowane do pamięci o swobodnym dostępie (RAM). Natomiast pamięć programu (ROM) zwykle jest zmapowana na końcu obszaru adresowego, gdyż adresy procedur obsługi przerwań (podrozdział 1.2.4) są odczytywane z obszaru pamięci o adresach `0xFFFA`–`0xFFFF`. Ponadto, choć nie jest to konieczne, wskazane jest, aby *strona zerowa* (pierwsze 256 adresów pamięci) była do zapisu i odczytu, gdyż dostęp do niej jest szybszy niż do pozostałego obszaru pamięci. Jest to możliwe dzięki specjalnym trybom adresowania, które używają jedynie jednego bajtu na operand. Choć różnica wydaje się pozornie niewielka, każda operacja na stronie zerowej w porównaniu do innych adresów pozwala zaoszczędzić nawet do kilku cykli zegarowych, dając przy tym wymierne korzyści.

Wskaźnik stosu (`SP`) jest rejestrem ośmiobitowym, lecz stos zaczyna się od adresu 256, dlatego w przypadku operacji na stosie bit dziewiąty jest zawsze ustawiony. Rejestr `SP` wskazuje zawsze na dolne osiem bitów pierwszej wolnej komórki stosu. W przypadku operacji zapisania bajtu na stosie, najpierw ten bajt jest zapisywany w komórce o indeksie wskazywanym przez rejestr `SP`, a następnie wartość w rejestrze jest zmniejszana o jeden. Ponieważ szyna danych, jak i wszystkie rejestry są ośmiobitowe, również stos operuje w porcjach jednobajtowych. W celu zapisania dwubajtowego adresu (na przykład podczas wywoływania procedury), należy wykonać dwa oddzielne zapisy.

1.2.3. Wyprowadzenia MOS 6502

Rysunek 1.1 przedstawia wyprowadzenia procesora MOS 6502, jest ich 40, ale trzy z nich nie są używane. Widoczne jest 16 wyprowadzeń przeznaczonych do przesyłania adresu (A_0–A_{15}) oraz 8 wyprowadzeń przeznaczonych do przesyłania danych (D_0–D_7). Kierunek transmisji zależy od wyprowadzenia R/W. Wysoki stan oznacza odczyt z szyny danych, a niski — zapis danych. Wyprowadzenia te należy podłączyć odpowiednio do szyny adresowej oraz szyny danych. Na rysunku 1.2 przedstawiony jest przykładowy schemat połączenia mikroprocesora z układami zewnętrznymi.

Oprócz wyżej wspomnianych, procesor ma następujące wyprowadzenia [6]:



Rysunek 1.1: Układ wyprowadzeń MOS 6502, źródło [2] (strona 42, rysunek 1.15)

- ϕ_0(IN), ϕ_1/ϕ_2 (OUT) — procesor ma wbudowany oscylator, którego częstotliwość determinuje układ RC lub podłączony rezonator kwarcowy; procesor może być też taktowany zewnętrznym sygnałem o poziomach TTL; na podstawie sygnału zegara otrzymanego na wyprowadzeniu ϕ_0 oscylator generuje dwa sygnały zegarowe o nienakładających się przebiegach (podłączone do wyprowadzeń ϕ_1 oraz ϕ_2), sterujące pracą procesora i służące do synchronizowania z innymi układami,

- RDY — pozwala na wstrzymanie wykonywania instrukcji, może być użyty w celu obsługi zbyt wolnej pamięci lub do implementacji bezpośredniego dostępu do pamięci (ang. *Direct Memory Access*),

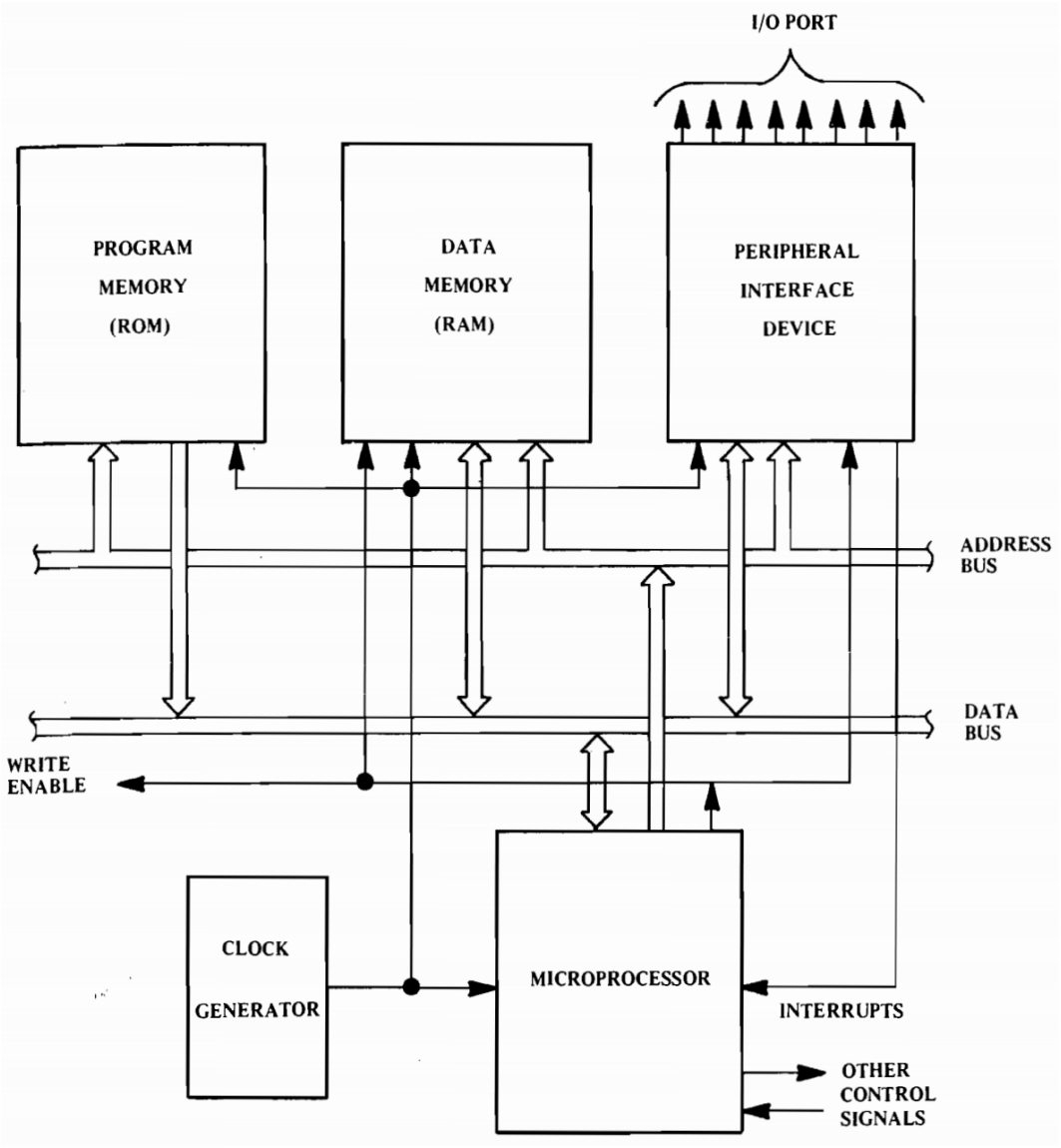
- $\overline{IRQ}$ — niski poziom wyprowadzenia rozpoczyna procedurę przerwania maskowalnego,

- $\overline{NMI}$ — opadające zbocze wyprowadzenia aktywuje procedurę przerwania niemaskowalnego,

- $S.O.$ — sygnał opadający ustawia flagę przepełnienia w procesorze,

- $SYNC$ — wyprowadzenie pozwalające stwierdzić, czy procesor jest w trakcie pobierania kolejnej instrukcji (ang. *opcode fetch*); odpowiednie sterowanie linią RDY na podstawie wyprowadzenia $SYNC$ umożliwia krokową pracę procesora,

- $\overline{RES}$ — zbocze narastające aktywuje procedurę zerowania jednostki centralnej.



Rysunek 1.2: Schemat komputera z procesorem MOS 6502. Widocznych jest między innymi kilka układów podłączonych do tej samej szyny danych i adresowej. Źródło: podręcznik MOS [2]

1.2.4. Przerwania i procedura startu

Jak każdy procesor układ MOS 6502 ma system przerwań. Ostatnie sześć bajtów przestrzeni adresowej (`0xFFFA`–`0xFFFF`) przechowuje wektor przerwań, czyli tablicę adresów procedur obsługi przerwań. W procesorze istnieją ich cztery rodzaje widoczne w tabeli 1.2. Status żądania przerwania (ang. *interrupt request*) jest sprawdzany po wykonaniu każdej instrukcji. W przypadku wykrycia przerwania, zamiast wykonania kolejnej instrukcji, następuje rozpoczęcie procedury obsługi przerwania. Procesor odkłada na stosie adres, do którego należy przywrócić kontrolę sterowania po zakończeniu obsługi przerwania (adres powrotu) oraz aktualną wartość rejestru stanu procesora. Zostaje ustawiona flaga blokowania przerwania (`I`). Następnie następuje odczytanie z wektora przerwań (tabela 1.2) szesnastobitowego adresu, do którego następuje skok.

Tabela 1.2: Przerwania procesora MOS 6502, źródło: opracowanie własne

Rodzaj	**Opis**	**Wektor przerwania**
BRK	programowe przerwanie, wywołane rozkazem `BRK`	`0xFFFE/0xFFFF`
IRQ	przerwanie aktywowane niskim poziomem wyprowadzenia IRQ	`0xFFFE/0xFFFF`
NMI	niemaskowalne przerwanie wywoływane zboczem opadającym linii NMI	`0xFFFA/0xFFFB`
RESET	przerwanie wywoływane zboczem narastającym wyprowadzenia RES	`0xFFFC/0xFFFD`

Sprzętowe przerwanie IRQ następuje po wykryciu niskiego stanu wprowadzenia IRQ. Jest to przerwanie maskowalne, czyli takie, którego odbiór można zablokować, w przypadku procesora MOS 6502 ustawiając flagę `I` stanu procesora. Ponadto po odebraniu dowolnego przerwania procesor również ustawia tę flagę, co oznacza, że domyślnie w trakcie obsługi przerwania IRQ inne takie przerwania są ignorowane. Inaczej jest w przypadku przerwania niemaskowalnego, którego nie można zablokować. Wykrycie opadającego zbocza wyprowadzenia NMI aktywuje obsługę tego przerwania (po skończeniu wykonywania aktualnej instrukcji), w szczególności przerwanie NMI może wystąpić w trakcie obsługi przerwania maskowalnego. Programowe przerwanie BRK działa analogicznie do przerwania maskowalnego IRQ, współdzielą one nawet procedurę obsługi. W przypadku przerwania programowego na stosie zostaje umieszczona wartość rejestru stanu procesora z ustawianą flagą `B`, w ten sposób programista może sprawdzić, które przerwanie nastąpiło (kod 1.1).

Kod 1.1: Przykładowy sposób rozróżnienia przerwania IRQ od przerwania BRK

```
1 PLA               ; odczytanie flag znajdujących się na stosie
2                   ;   do akumulatora
3 PHA               ; ponowne zapisanie tej wartości na stosie,
4                   ;   aby nie ,,zepsuć'' stosu
5 AND #%00010000    ; sprawdzenie flagi B
6 BNE procBrk       ; jeśli flaga ustawiona, skok do procBrk
7 ...               ; wystąpiło przerwanie IRQ
```

Szczególnym rodzajem przerwania jest zerowanie (ang. *reset*). Po wykryciu zbocza narastającego na wyprowadzeniu $\overline{RES}$ procesor rozpoczyna trwającą siedem cykli procedurę restartu. W trakcie zerowania licznik instrukcji jest ustawiany na adres przechowywany w pamięci o adresie `0xFFFC` (dolny bajt) oraz `0xFFFD` (górny bajt), natomiast wskaźnik stosu przyjmuje wartość `0xFD`. Nie jest to przypadkowa liczba — za procedurę restartu odpowiada ta sama część układu logicznego, która odpowiada za pozostałe przerwania. Jednak dodatkowo w trakcie jej trwania wyprowadzenie R/W jest ustawione na tryb odczytu, a przed przystąpieniem do obsługi przerwania, następuje wyzerowanie wskaźnika stosu. Cała procedura wygląda zatem następująco:

1. ustaw linię R/W na odczyt,
2. ustaw wskaźnik stosu na 0,

3. wykonaj standardową procedurę przerwania:

   3.1. zapisz na stosie górny bajt licznika instrukcji i zmniejsz wskaźnik stosu (jednak ze względu na linię R/W w rezultacie nastąpi odczyt tego bajtu ze stosu),

   3.2. zapisz na stosie dolny bajt licznika instrukcji i zmniejsz wskaźnik stosu (jak wyżej),

   3.3. zapisz na stosie stan flag procesora i zmniejsz wskaźnik stosu, (jak wyżej), w wyniku powyższych operacji wskaźnik stosu ma teraz wartość `0xFD`,

   3.4. odczytaj adres funkcji obsługi przerwania spod adresów `0xFFFC` (dolny bajt) i `0xFFFD` (górny bajt) i skocz do tego adresu.

Choć te „fałszywe zapisy" są zupełnie zbędne i procesor poświęca im niepotrzebnie kilka cykli, nie jest to problem, bowiem procedura restartu nie jest krytyczna pod względem czasu wykonania, a pozwala to zmniejszyć liczbę tranzystorów dzięki współdzieleniu części układu odpowiadającego za przerwania i zerowanie.

1.2.5. Adresowanie pamięci

Tryb adresowania pamięci to sposób określania operandów instrukcji. Procesor MOS 6502 obsługuje dwanaście trybów. Można je podzielić na dwie grupy: nieużywające i używające rejestrów indeksowych `X`/`Y`.

Uwaga notacyjna: w języku asemblera MOS 6502 zwykło się poprzedzać liczby w systemie szesnastkowym za pomocą znaku `$`. Ta konwencja została zachowana w tej pracy.

Adresowanie nieużywające rejestrów X/Y

- **Tryb natychmiastowy**

  Długość operandu: 1 bajt

  Operand bezpośrednio zawiera wartość używaną przez instrukcję. W języku asemblera MOS 6502 przyjęło się poprzedzać operand znakiem `#`.

  Przykład: `LDA #$10` — załaduj wartość `0x10` do rejestru `A`.

- **Tryb implikowany**

  Długość operandu: 0 bajtów

  Używany niejawnie, gdy operand wynika z instrukcji.

  Przykład: `INY` — zwiększenie wartości rejestru `Y` o jeden, wprost z tej instrukcji wynika, że inkrementacja następuje właśnie na tym rejestrze.

- **Tryb bezwzględny**

  Długość operandu: 2 bajty

  Operand jest 16-bitowym adresem, spod którego zostanie odczytana wartość dla instrukcji.

  Przykład: `LDA $4060` — załaduj do rejestru `A` wartość komórki pamięci o adresie `0x4060`.

- **Tryb bezwzględny (strony zerowej)**

  Długość operandu: 1 bajt

  Specjalny rodzaj adresowania bezwzględnego, różni się jedynie długością operandu, a co za tym idzie ograniczeniem do pierwszej strony pamięci. Chociaż jest to nadmiarowy tryb adresowania, autorzy procesora mieli dobry powód do jego dodania. Dzięki niemu jednostka centralna podczas dekodowania instrukcji musi odczytać tylko dwa, a nie trzy bajty (kod operacji i operand). Zaoszczędzony w ten sposób odczyt bajtu skraca czas wykonania instrukcji — oszczędza się kilka cykli procesora w porównaniu do poprzedniego trybu.

  Przykład: `LDA $40` — załaduj do rejestru `A` wartość komórki pamięci o adresie `0x40`.

- **Tryb względny**

  Długość operandu: 1 bajt

  Ten tryb adresowania używany jest jedynie w instrukcjach skoków warunkowych. Ograniczenie operandu do liczby jednobajtowej ze znakiem oznacza, że maksymalnie skok warunkowy może zmienić wskaźnik instrukcji na adres wcześniejszy o 128 bajtów lub przesunięty do przodu o 127 bajtów. Nie jest możliwe wykonanie dalszych skoków jedną instrukcją (bez stosowania wielokrotnego skakania).

  Dla czytelności w języku asemblera do skoków używa się etykiet, należy jednak pamiętać, że asembler zamienia je na względne adresy.

  Pseudokod:

```
PC <- PC + (Operand <= 127 ? Operand : (Operand - 256))
```

- **Tryb pośredni**

  Długość operandu: 2 bajty

  Ten tryb adresowania jest używany jedynie w dwóch instrukcjach: skoku do procedury (`JSR`) oraz skoku bezwarunkowego (`JMP`).

  Operand zawiera adres, spod którego zostanie odczytana 16-bitowa wartość.

  Zwyczajowo w asemblerze wartość operandu podaje się w nawiasach okrągłych.

  Przykład: `JMP ($4060)` — skok nastąpi do adresu, który w momencie wykonywania instrukcji znajduje się w pamięci pod adresem `0x4060`. Gdy w dwóch kolejnych komórkach pamięci o adresach `0x4060` oraz `0x4061` znajdują się liczby `2E` i `BC`, to skok nastąpi do adresu `0xBC2E` — cienkokońcówkowa (ang. *little-endian*) kolejność bajtów.

  Uwaga: W procesorze MOS 6502 występuje błąd związany z tym trybem adresowania: gdy dolny bajt adresu jest równy 255, wtedy adresy dwóch bajtów, które zostaną odczytane to nie `X` oraz następny `X + 1`, a `X` oraz `X - 255`. Procesor oczekuje bowiem, że obydwa bajty będą znajdowały się na tej samej stronie pamięci. Na przykład instrukcja `JMP ($40FF)` skoczy pod adres, którego dolny bajt znajdzie się w komórce `0x40FF`, a górny bajt w komórce `0x4000`. W niektórych późniejszych wersjach procesora z tej serii taki problem nie występuje. Odtworzenie błędów tego typu jest kluczowe dla poprawnego emulowania procesora MOS 6502.

  Pseudokod:

```
low_byte <- memory[Operand]
if (Operand & 0xFF) == 0xFF:
```

```
        high_byte <- memory[Operand & 0xFF00]
    else
        high_byte <- memory[Operand + 1]
    PC <- low_byte | high_byte << 8
```

Adresowanie indeksowane rejestrami X/Y

- **Tryb bezwzględny, indeksowany rejestrem X**

  Długość operandu: 2 bajty

  Rozszerzenie zwykłego bezwzględnego trybu adresowania. Operand zawiera adres bazowy, do którego dodawana jest wartość rejestru `X`. Ten tryb adresowania jest najczęściej używany w pętlach.

  Przykład: `LDA $2030, X` — do operandu `0x2030` dodawana jest wartość rejestru `X` i spod takiego adresu następuje odczytanie wartości.

  Pseudokod:

```
    value <- memory[(Operand + X) & 0xFFFF]
```

- **Tryb bezwzględny, indeksowany rejestrem Y**

  Długość operandu: 2 bajty

  Tryb adresowania analogiczny do bezwzględnego indeksowanego rejestrem X, ale zamiast rejestru X używany jest rejestr `Y`.

  Przykład: `LDA $2030, Y` — do operandu `0x2030` dodaje się wartość rejestru `Y` i spod takiego adresu następuje odczytanie wartości.

  Pseudokod:

```
    value <- memory[(Operand + Y) & 0xFFFF]
```

- **Tryb bezwzględny (strony zerowej), indeksowany rejestrem X**

  Długość operandu: 1 bajty

  Podobnie do nieindeksowanego bezwzględnego trybu adresowania strony zerowej, w tym przypadku operand, czyli bazowy adres, jest ograniczony do zerowej strony (adresy 0–255), jednak dodatkowo do operandu dodawana jest wartość rejestru `X`. Suma musi zawsze pozostawać na tej samej stronie, zatem w przypadku przekroczenia wartości `0xFF` górne dodatkowe bity są ignorowane.

  Przykład: `LDA $30, X` — do operandu `0x30` dodaje się wartość rejestru X i spod adresu `(0x30 + X) % 256` następuje odczytanie wartości.

  Pseudokod:

```
    value <- memory[(Operand + X) & 0xFF]
```

- **Tryb bezwzględny (strony zerowej), indeksowany rejestrem Y**

  Długość operandu: 1 bajty

  Odpowiednio jak w poprzednich trybach — operand jest ograniczony do adresu na pierwszej stronie, do niego dodaje się wartość rejestru `Y`.

  Przykład: `LDA $20, Y` — do operandu `0x20` dodaje się wartość rejestru `Y` i spod adresu `(0x20 + Y) % 256` następuje odczytanie wartości.

Pseudokod:

```
value <- memory[(Operand + Y) & 0xFF]
```

- **Tryb pośredni, indeksowany rejestrem X**

  Długość operandu: 1 bajt

  Operand jest adresem bazowym, ograniczonym do strony zerowej, do którego dodawana jest wartość rejestru `X`, spod tego adresu odczytywana jest wartość 16 bitowa (dwie komórki pamięci), która jest ostatecznym adresem, spod którego odczytuje się końcową wartość. Co istotne suma operandu oraz rejestru X jest ograniczona do 8 bitów, 9. bit jest ignorowany. Podobnie jest w przypadku odczytania dwóch kolejnych bajtów.

  Dzięki temu trybowi adresowania programista ma możliwość dynamicznego dostępu do dowolnego adresu w całej przestrzeni adresowej, mimo że żaden z rejestrów nie jest 16-bitowy.

  Przykład: `LDA ($70, X)`, niech wartość rejestru `X` wynosi `0xA0`, zatem adres pośredni należy odczytać spod komórek pamięci `0x70 + 0xA0` oraz `0x70 + 0xA0 + 1`. Ponieważ wynik wykracza poza stronę zerową, następuje zignorowanie dziewiątego bitu, zatem finalnie adres pośredni odczytuje się z komórek `0x11` oraz `0x12`, a następnie ponownie odczytuje się wartość spod tego adresu, czyli wartość operandu znajduje się w komórce pamięci `0x1211`.

  Pseudokod:

```
low_byte <- memory[(Operand + X) % 256]
high_byte <- memory[(Operand + X + 1) % 256]
indirect_address <- low_byte | (high_byte << 8)
value <- memory[indirect_address]
```

- **Tryb pośredni, indeksowany rejestrem Y**

  Długość operandu: 1 bajt

  Tryb adresowania analogiczny do poprzedniego, ale wartość rejestru `Y` jest dodawana dopiero po odczytaniu adresu pośredniego z komórek pamięci, na które wskazuje operand.

  Przykład: `LDA ($70), Y` — adres pośredni jest odczytywany spod komórek pamięci `0x70` oraz `0x70 + 1`, do tego adresu dodawana jest wartość rejestru `Y`, a następie odczytywana jest wartość spod adresu sumy.

  Pseudokod:

```
low_byte <- memory[Operand]
high_byte <- memory[(Operand + 1) % 256]
indirect_address <- low_byte | (high_byte << 8) + Y
value <- memory[indirect_address & 0xFFFF]
```

Rozdział 2

Przegląd metod uruchamiania programów skompilowanych na inną architekturę

2.1. Metody emulowania

W celu uruchomienia aplikacji skompilowanych na inną architekturę używa się emulatorów, czyli programów, które imitują oryginalny sprzęt i jego zachowanie. Jest to bardzo szerokie pojęcie i w zależności od rodzaju platformy źródłowej oraz docelowej wymaga różnego podejścia i ilości pracy. Emulować można różne elementy. W przypadku innej architektury niż architektura gospodarza należy co najmniej symulować oryginalny procesor, jednak emulowanie samej jednostki centralnej może nie być wystarczające. Przykładowo, w przypadku konsol do gier, aby cokolwiek zobaczyć, konieczne jest również emulowanie układu graficznego. W celu odegrania dźwięku trzeba emulować układ audio, a użycie klawiatury i myszki komputera gospodarza do sterowania, będzie wymagało emulowania kontrolera do gier.

Pierwsze emulatory datuje się na lata 60. ubiegłego wieku [4]. Wybrane konfiguracje komputerów IBM System/360 mogły uruchamiać starsze oprogramowanie bez przepisywania ich na nowy system. W trakcie dynamicznego rozwoju komputerów domowych w latach 70. i 80. również na tych platformach zaczęły pojawiać się próby emulacji innych systemów komputerowych. W roku 1979 na łamach czasopisma *Micro* [11] Dann McCreary zaprezentował swój emulator procesora Intel 8080 (wprowadzony na rynek rok przed premierą MOS 6502) przeznaczony dla komputera Commodore KIM-1, który używał jednostki MOS 6502. W artykule autor zaznacza, że ten emulator na każdą instrukcję Intela potrzebuje około stu operacji procesora gospodarza. Jest to istotne spowolnienie wykluczające wiele praktycznych zastosowań. W magazynie Compute! [10] z roku 1988 pada pytanie od czytelnika: „Czy Commodore 64 może emulować MS-DOS?", na które odpowiedź redaktorów była pozytywna, lecz zwrócili oni uwagę, że ze względu na wydajność jest to zupełnie niepraktyczne.

W latach 90. nastąpił jednak duży wzrost wydajności komputerów osobistych i to z tego okresu można znaleźć pierwsze informacje o praktycznych i faktycznie działających emulatorach na komputerach domowych. Autor emulatora iNES na swojej stronie pisze, że w roku wydania pierwszej wersji programu (1996) istniał tylko jeden inny program do emulacji konsoli NES [13][1]. Oznacza to, że faktycznie działający emulator konsoli firmy Nintendo powstał

[1] W Internecie można znaleźć informacje o emulatorze *Family Computer Emulator V0.35 for FM Towns*, którego początki datowane są na rok 1990, jednak należy mieć na uwadze brak możliwości zweryfikowania źródła tych informacji [23].

po około trzynastu latach od premiery urządzenia. Od mniej więcej tego okresu rozpoczął się gwałtowny rozwój takich aplikacji [5]. Emulator UltraHLE mógł uruchomić wybrane gry przeznaczone na konsolę Nintendo 64 (1996) już 3 lata po premierze urządzenia.

Dwie największe trudności w emulatorach to ograniczony dostęp do szczegółów na temat działania sprzętu oraz narzut wydajnościowy związany z symulacją innych układów. Przeważająca większość komputerów i konsol dostępnych na rynku to konstrukcje własnościowe i zamknięte, do których producenci udostępniają często tylko szczątkowe informacje na potrzeby programowania (choć czasem nawet dokumentacja nie jest ogólnodostępna, a przykazywana jedynie wybranym firmom). Z tego powodu inżynieria wsteczna jest podstawowym źródłem wiedzy na temat działania sprzętu na potrzeby emulatorów. Problemy wydajnościowe wynikają ze specyfiki emulacji — symulacja układów logicznych może być nawet o rzędy wielkości wolniejsza niż oryginalny sprzęt, dlatego konieczne jest znalezienie balansu między wiernością emulacji a kompatybilnością z oryginalnym oprogramowaniem. Każdy emulator jest tylko przybliżeniem prawdziwego sprzętu, lecz dobre przybliżenie pozwala w zadowalający sposób korzystać z tego rodzaju aplikacji.

Sposoby emulacji różnią się w zależności od symulowanego urządzenia — zupełnie inaczej wygląda emulowanie układu generującego grafikę czy karty dźwiękowej od odtwarzania zachowania procesora. Ta praca skupia się przede wszystkim na tym ostatnim. W większości przypadków potrzebne jest jedynie emulowanie wykonywania instrukcji przez procesor. Nowoczesne jednostki centralne są bardzo złożone wewnętrznie, potrafią jednocześnie wykonywać kilka rozkazów (superskalarność, wykonanie potokowe), mogą zmieniać kolejność wykonywania instrukcji (wykonywanie poza kolejnością), mają wysokowydajną pamięć podręczną, która przyspiesza działanie w przypadku typowego używania pamięci operacyjnej. W większości przypadków jednak nie ma potrzeby emulować tych aspektów procesora. Ważne są tylko te elementy układu, które mają wpływ na rezultat instrukcji. Wierne symulowanie oryginalnej szybkości sprzętu zależy od specyfiki emulowanego urządzenia. Konsole do gier są przykładem, gdzie odwzorowanie szybkości ma znaczenie. Jest oczywiste, że w przypadku zbyt wolnej emulacji granie w gry może być utrudnione lub niemożliwe, mniej jasny jest przypadek zbyt wydajnej emulacji. Dzisiejsze oprogramowanie (w tym gry) jest projektowane w taki sposób, aby nie zależeć od prędkości procesora, jednak starsze programy nie były pisane w taki sposób. Jeszcze w latach 90. ubiegłego wieku zdarzało się, że gra działała poprawnie tylko na jednym modelu procesora. Dlatego pisząc emulator, należy podjąć decyzję, czy trzeba limitować jego szybkość symulacji, czy też oczekiwane jest emulowanie tak szybko jak to możliwe. Zatem, o ile gra to wspiera, czasem emulacja może być „lepsza" niż oryginał dzięki rysowaniu w wyższej rozdzielczości i z większą liczbą klatek na sekundę. Istnieje kilka technik uruchamiania programów przeznaczonych na inne komputery niż komputer gospodarza. Najpopularniejszą metodą jest interpretacja instrukcji obcej architektury. To podejście jest bardzo elastyczne, ale może być dość wolne. Inną możliwością jest przetłumaczenie programu na architekturę maszyny gospodarza. W ostatnich latach pojawił się również pomysł symulacji procesora w całości jako układu logicznego. To rozwiązanie pozwala osiągnąć stuprocentową kompatybilność, ale wymaga o rzędy wielkości szybszego komputera niż oryginalny procesor.

2.2. Interpreter procesora

Prostą metodą emulacji jest napisanie interpretera kodu maszynowego. Ta metoda odpowiada sposobowi działania procesorów, czyli odczytywaniu oraz dekodowaniu kolejnych instrukcji programu i wykonywaniu każdej z nich. Typowa pętla interpretera przedstawiona jest w ko-

Kod 2.1: Pseudokod typowego interpretera kodu maszynowego

```
void run() {
    while (!halt) {
        step();
    }
}

void step() {
    switch (memory[PC++]) {
        case OPCODE_1:
            simulate_opcode_1();
            break;
        case OPCODE_2:
            simulate_opcode_2();
            break;
        ...
    }
}
```

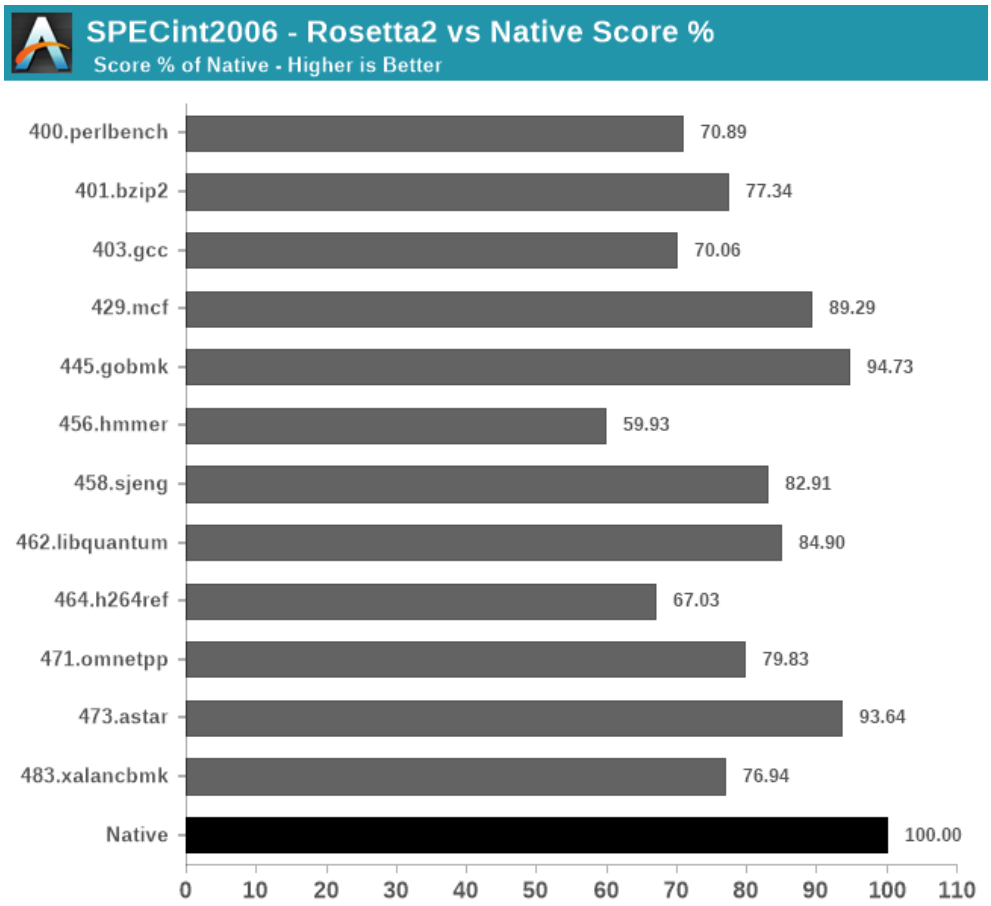
dzie 2.1. W każdym obrocie wykonywany jest jeden krok procesora, czyli symulacja kolejnego rozkazu. Interpretery nie są zbyt szybkie, ale są stosunkowo łatwe do napisania i nie mają wad pewnych bardziej zaawansowanych technik, na przykład programy samomodyfikujące się nie stanowią w tym przypadku problemu.

Warto zaznaczyć, że w przypadku emulacji niektórych urządzeń, takich jak układy graficzne, możliwa jest emulacja na wyższym poziomie abstrakcji. Gry na konsolę Nintendo 64 (1996) nie były już pisane w języku asemblera, tylko w C, a rysowanie na ekranie odbywało się poprzez specjalne, dostarczone funkcje (a nie jak dawniej przez bezpośrednie modyfikowanie obszaru pamięci mapowanego na ekran). Dzięki temu emulator UltraHLE zamiast symulować wewnętrzne zasady działania układu graficznego, przechwytuje wywołania funkcji graficznych i tłumaczy je na polecenia zrozumiałe przez docelową bibliotekę graficzną (Glide). To podejście pozwoliło już kilka lat po premierze konsoli grać w wybrane gry przy zadowalającej liczbie klatek na sekundę. Z drugiej strony biblioteka działających gier była dosyć skromna, gdyż to rozwiązanie tylko przybliżało działanie oryginalnego układu graficznego.

2.3. Translacja instrukcji

Translacja instrukcji polega na przetłumaczeniu instrukcji z programu źródłowego na zestaw instrukcji używany w docelowej architekturze. W zależności od stopnia podobieństwa między zestawami rozkazów, w niektórych przypadkach możliwe jest tłumaczenie instrukcji jeden do jednego, w innych przypadkach tłumaczenie może być bardziej skomplikowane. Tłumaczenie może odbywać się statycznie — bez konieczności wcześniejszego uruchomienia kodu (ang. *ahead of time*), bądź dynamicznie, czyli na bieżąco podczas wykonywania programu. Statyczna translacja czasem nie jest możliwa, ponieważ przed uruchomieniem kodu część instrukcji może nie zostać odkryta przez tłumacza. Na przykład pewne części pliku wykonywalnego mogą być zaszyfrowane i odszyfrowywane w czasie wykonywania (metoda często stosowana w zabezpieczeniach antypirackich oraz przeciwdziałających inżynierii wstecznej).

W dynamicznej translacji patrzy się na krótkie sekwencje kodu — zwykle rzędu pojedynczego bloku prostego[2], potem następuje tłumaczenie tego fragmentu i zapamiętanie wyniku. Kod jest tłumaczony tylko wtedy, gdy zostanie odkryty, a rozgałęzienia są wykonywane tak, aby wskazywały na już przetłumaczony i zapisany kod. Dynamiczne tłumaczenie w przeciwieństwie do tradycyjnej emulacji eliminuje wąskie gardło emulatora, czyli cykl pobierz instrukcję–zdekoduj–wykonaj w pętli (ang. *fetch–decode–execute*), za to powoduje narzut wydajności w czasie tłumaczenia. Jednak przetłumaczone sekwencje kodu są zwykle wykonywane wielokrotnie, co amortyzuje ten koszt. Kompilatory JIT (ang. *Just–In–Time*) mogą być postrzegane jako dynamiczne translatory z wirtualnego zestawu instrukcji (bajtkodu) na rzeczywiste rozkazy.



Rysunek 2.1: Porównanie wydajności aplikacji skompilowanych natywnie na procesory Apple ARM64 oraz ich odpowiedników skompilowanych na architekturę x86-64, jako procent wydajności wersji natywnej (więcej znaczy lepiej), źródło: [17]

Tłumaczenie instrukcji na natywne rozkazy ma zastosowanie przede wszystkim tam, gdzie znaczenie ma szybkość działania. Firma Apple zmieniała trzy razy architekturę swoich komputerów. Pierwsze przejście miało miejsce w 1994 roku z serii czipów Motorola 68000 na architekturę PowerPC, następnie w 2006 roku na procesory Intel i w 2020 roku kolejny raz na własne układy o architekturze ARM64. Każde takie przejście sprawia, że wszystkie aplikacje wymagają przekompilowania na nową architekturę. Ze względu na zależności od bibliotek

[2]Blok prosty — sekwencja kolejnych instrukcji z jednym wejściem i jednym wyjściem kontroli sterowania (wewnątrz bloku nie występują instrukcje skoków ani powrotów).

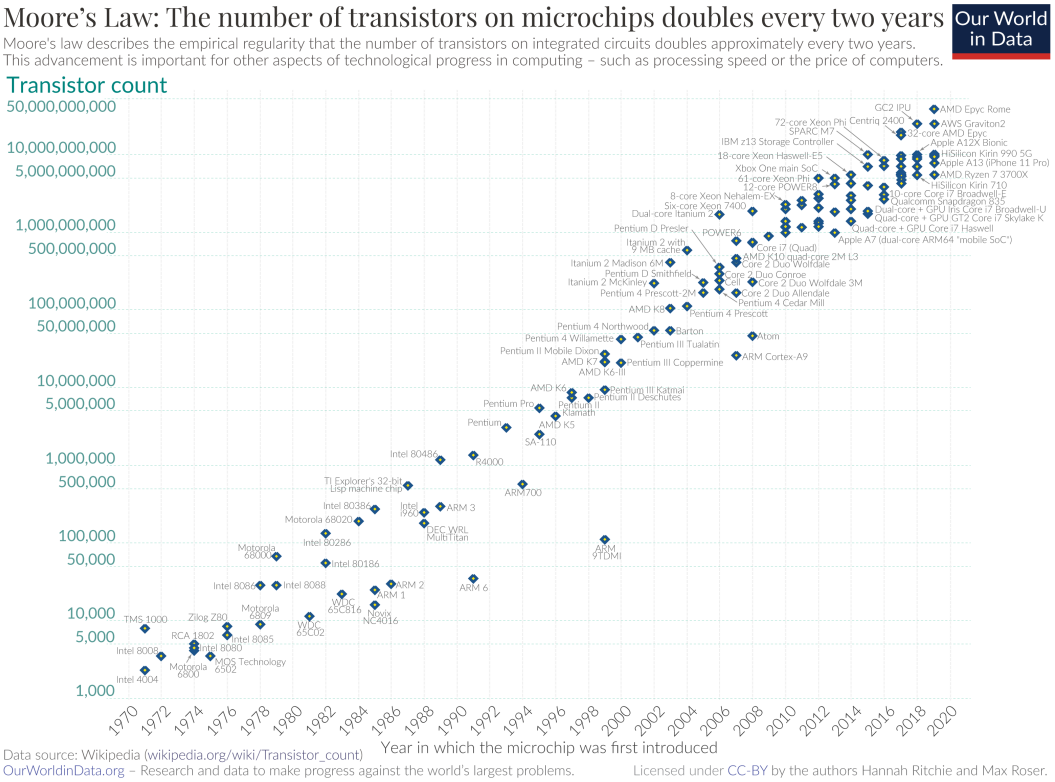
i kod specyficzny dla danej architektury proces ten może trwać długo. Dla użytkowników istotny jest dostęp do istniejącego oprogramowania, dlatego konieczne było zapewnienie alternatywnego rozwiązania dla uruchomienia programów, które jeszcze nie zostały przekompilowane. Przy każdym takim przejściu firma Apple wbudowała w system transparentną warstwę tłumaczącą programy z poprzedniej architektury na nową [16]. Dzięki niej przejście na nową architekturę nie było uciążliwe dla użytkowników, bowiem mogli oni korzystać ze wszystkich używanych wcześniej aplikacji. Tłumaczenie instrukcji jest efektywną metodą, choć natywnie skompilowane programy są i tak wydajniejsze. W przypadku tłumaczenia programów skompilowanych na architekturę x86-64 na procesory Apple narzut wydajności wynosi jedynie od 5 do 40 procent (rysunek 2.1). Dodatkowo przy każdej zmianie architektury nowe procesory były i tak wydajniejsze niż poprzednia generacja starszej architektury, dlatego użytkownik nie odczuwał żadnego spowolnienia związanego ze zmianą architektury. Metoda tłumaczenia instrukcji jest też czasem wykorzystywana w emulatorach konsol do gier, również z powodów wydajnościowych.

2.4. Symulowanie oryginalnego układu scalonego

Patrząc na procesor całościowo, w teorii najlepszym sposobem jest symulowanie oryginalnego układu scalonego jako układu bramek logicznych. Z oczywistych względów żaden producent nie udostępnia dokładnych schematów swoich produktów, przez co to rozwiązanie wydaje się bardzo trudne do zrealizowania. Bardzo trudne, lecz nie niemożliwe — w 2010 roku za pomocą metod inżynierii wstecznej grupie badaczy w składzie: Greg James, Barry Silverman i Brian Silverman udało się odtworzyć schemat procesora MOS 6502 [8]. Wykonali oni serię zdjęć wysokiej rozdzielczości poszczególnych warstw procesora i za pomocą przygotowanych skryptów odtworzyli wszystkie połączenia. Taki sposób okazał się możliwy dzięki prostej konstrukcji w połączeniu z faktem, że procesory w tamtych latach były projektowane ręcznie, bez pomocy zautomatyzowanych narzędzi [9]. Efektem ich prac jest dokładny schemat procesora, który już łatwo symulować. Takie rozwiązanie oznacza w stu procentach dokładny model procesora, choć należy pamiętać, że symulowanie zjawisk fizycznych również obarczone jest pewnym przyjętym przybliżeniem. Ich symulator nie uwzględnia analogowych zjawisk takich jak rezystancja, pojemność, prądy upływu, nie symuluje czasu propagacji, a tranzystory włączają się i wyłączają natychmiast. Jednak nie wpływa to na poprawność emulacji. Można bowiem założyć, że emulowany procesor działa poprawnie (celem nie jest weryfikacja poprawności, a emulacja istniejącego układu). Jest to najdokładniejsza metoda emulacji tego procesora. Po przeniesieniu wszystkich bramek do symulatora istotne są już tylko wirtualne wyprowadzenia. Wystarczy podłączyć wirtualny zegar, emulowaną pamięć RAM do wyprowadzeń szyny adresowej oraz danych.

Przy użyciu takiej symulacji autorom udało się uruchomić różne gry przeznaczone na konsolę Atari 2600. Niestety symulowanie zestawu bramek logicznych jest bardzo kosztowne obliczeniowo. To rozwiązanie może symulować procesor MOS z taktowaniem około 27 Hz, czyli kilkadziesiąt tysięcy razy wolniej niż oryginalne taktowanie jednostki centralnej, pomimo że procesor MOS składa się z zaledwie kilku tysięcy tranzystorów. Zoptymalizowana wersja symulatora na najszybszych komputerach domowych działa wielokrotnie szybciej — z taktowaniem ok. 33 kHz, to jednak wciąż tylko ułamek prędkości działania oryginału [19]. Z powodu szybkości działania symulator jest niepraktyczny do emulacji gier, za to stanowi nieocenione źródło wiedzy na temat tego procesora. Przykładowo analiza połączeń pozwala zrozumieć, jak dokładnie działają nieudokumentowane kody instrukcji. Zgodnie z prawem Moore'a co około 18–24 miesiące moc obliczeniowa komputerów podwaja się. Liczby tranzy-

storów we współczesnych procesorach są liczone w dziesiątkach i setkach miliardów (rys. 2.2), dlatego symulowanie nowszych i szybszych procesorów w ten sposób jest jeszcze trudniejsze i w praktyce tam, gdzie istotna jest wydajność — nierealne.



Rysunek 2.2: Wzrost liczby tranzystorów w procesorach w latach 1970–2020, źródło: [18]

Rozdział 3

Tłumaczenie instrukcji 6502 na architekturę x86-64

3.1. Mapowanie rejestrów

Przed rozpoczęciem tłumaczenia należy podjąć decyzję, jak zmapować rejestry procesora MOS na rejestry dostępne w architekturze x86-64. Na docelowej platformie rejestry są ogólnego przeznaczenia, więc nie ma większego znaczenia, jak zostaną przydzielone. Jedyną różnicą, która wpłynie nieznacznie na generowany kod, jest decyzja, czy użyć rejestrów zapisywanych przez wołającego (ang. *caller saved*), czy rejestrów zapisywanych przez wołanego (ang. *callee saved*). Używanie wyłącznie rejestrów zapisywanych przez wołanego skraca kod do wywoływania innych funkcji, ponieważ przed instrukcją `CALL` nie trzeba zapisywać wartości rejestrów na stosie. Z tego powodu w pracy używane jest mapowanie przedstawione w tabeli 3.1.

Tabela 3.1: Mapowanie rejestrów procesora MOS 6502 na rejestry architektury x86-64 w projekcie

MOS 6502	**x86-64**
A	R12B
X	R13B
Y	R14B

3.2. Mapowanie instrukcji

Pierwszym krokiem zamiany kodu maszynowego MOS na x86-64 jest określenie, jakie instrukcje docelowej architektury odpowiadają oryginalnym rozkazom procesora. W tabeli 3.2 znajdują się wszystkie instrukcje wraz z ich odpowiednikami oraz flagami, które są przez nie używane (modyfikowane lub odczytywane). Niektóre z nich tłumaczy się trywialnie bez żadnych problemów, na przykład operacje arytmetyczne — dodawanie z przeniesieniem oraz odejmowanie z pożyczką działają tak samo na obydwóch architekturach. Część z nich nie ma żadnych odpowiedników, na przykład rodzina procesorów AMD64 nie ma trybu operacji dziesiętnych, więc instrukcje do modyfikacji flagi `D` (czy tryb dziesiętny jest aktywny) oraz samo dodawanie i odejmowanie w tym trybie należy emulować. Nie jest również zaskoczeniem, że pewne operacje specyficzne dla danej architektury muszą być emulowane (między innymi rozkazy dotyczące przerwań). Niestety jest jeszcze jedna grupa instrukcji, które teoretycznie

wyglądają, jakby miały swoje odpowiedniki, ale możliwość ich użycia jest bardzo ograniczona. Przykładowo problematyczne są operacje na stosie. Stos na procesorze MOS 6502 znajduje się na pierwszej stronie pamięci (adresy 256–511). W przypadku wykroczenia poza ten zakres wskaźnik stosu po prostu odpowiednio się zawija. Nic nie stoi również na przeszkodzie, aby za pomocą instrukcji $\texttt{LDA}_{\texttt{MOS}}$ i bezwzględnego adresu w operandzie wczytać dowolną wartość ze stosu do akumulatora. Z tych względów zastąpienie instrukcji $\texttt{PHA}_{\texttt{MOS}}$ instrukcją $\texttt{PUSH}_{\texttt{x86-64}}$ jest właściwie niemożliwe. Ten element należy więc po prostu emulować, przeznaczając jeden z rejestrów ogólnego przeznaczenia na wskaźnik stosu i operować bezpośrednio na nim. Szczególnym przypadkiem jest para operacji $\texttt{JSR}_{\texttt{MOS}}$ i $\texttt{RTS}_{\texttt{MOS}}$, które służą kolejno do skoku do procedury oraz powrotu. Wyjście z wołanej funkcji działa podobnie do instrukcji $\texttt{RET}_{\texttt{x86-64}}$, czyli ze stosu zdejmowany jest adres powrotu i do niego następuje skok. Teoretycznie więc jest to sytuacja analogiczna do opisanej wyżej, jednak w odróżnieniu od typowych operacji na stosie, w tym przypadku bardzo rzadko zdarza się, aby programy odczytywały bądź modyfikowały adres powrotu. Choć jest to jak najbardziej możliwe, nie jest to typowy kod, dlatego w ustawieniach tłumacza będącego przedmiotem pracy, znajduje się opcja albo symulowania procedur, albo translacji odpowiednich instrukcji na $\texttt{CALL}_{\texttt{x86-64}}$ i $\texttt{RET}_{\texttt{x86-64}}$.

Tabela 3.2: Instrukcje MOS 6502 i opowiadające im instrukcje x86-64 wraz z informacją, które flagi są modyfikowane

Instrukcja MOS	**Instrukcja x86-64**	**Flagi MOS**	**Flagi x86-64**
ADC, SBC	ADC, SBC[c]	C, Z, N, V	OF, SF, ZF, CF
AND	AND	Z, N	OF (na 0), CF (na 0), SF, ZF
ASL/LSR	SHL/SHR	Z, C, N	OF, SF, ZF, CF
BCC	JNC	C	CF
BCS	JC	C	CF
BEQ	JNZ	Z	ZF
BIT	AND[a]	N, Z, V	
BMI	JS	N	SF
BNE	JZ	Z	ZF
BPL	JNS	N	SF
BRK	-	-	
BVC	JNO	V	OF
BVS	JO	V	OF
CLC	CLC	C	CF
CLD	-	D	
CLI	-	I	
CLV	-	V	
CMP, CPX, CPY	CMP/TEST	C, N, Z	CF, OF, SF, ZF
DEC, DEX, DEY	DEC	N, Z	OF, SF, ZF
INC, INX, INY	INC	N, Z	OF, SF, ZF
EOR	XOR	N, Z	OF (na 0), CF (na 0), SF, ZF
JMP	JMP	-	-
JSR	CALL[be]	-	-
LDA, LDX, LDY	MOV	N, Z	-
NOP	NOP	-	-
ORA	OR	N, Z	OF (na 0), CF (na 0), SF, ZF
PHA	PUSH[b]	-	-
PHP	PUSHF[bd]	-	-

PLA	POP[b]	N, Z	-
PLP	POPF[bd]	wszystkie	wszystkie
ROL/ROR	RCL/RCR	C, N, Z	CF, OF
RTI	-	-	
RTS	RET[e]	-	-
SEC	STC	C	CF
SED	-	D	
SEI	-	I	
STA, STX, STY.	MOV	-	-
TAX, TAY, TSX	MOV	N, Z	-
TXA, TYA	MOV	N, Z	-

[a] Instrukcja `BIT` zmienia w szczególny sposób flagi procesora (patrz dodatek A).

[b] Instrukcja oryginalna, jak i przetłumaczona korzystają ze stosu, jednak ze względu na różnice w szczegółach (maksymalny rozmiar stosu i zawijanie modulo 256, brak możliwości zmiany miejsca stosu) w praktyce nie jest możliwe przetłumaczenie oryginalnego stosu na natywny stos x86. Zamiast tego można po prostu używać instrukcji `MOV` i przeznaczyć jeden z rejestrów ogólnego przeznaczenia na emulowany wskaźnik stosu.

[c] Procesor MOS może dodawać i odejmować w systemie dziesiętnym, podobne instrukcje występują na architekturze x86, jednak zrezygnowano z nich w architekturze x86-64, zatem ten tryb należy emulować.

[d] Zarówno oryginalne instrukcje `PHP`/`PLP`, jak i odpowiedniki `PUSHF`/`POPF` służą do wkładania na stos oraz zdejmowania flag stanu procesora ze stosu, jednak ze względu na różnice w tych flagach w praktyce niemożliwe jest takie tłumaczenie, bowiem kolejne instrukcje mogą polegać na konkretnej wartości, która powinna znaleźć się na stosie.

[e] Tłumaczenie instrukcji `JSR` oraz `RTS` na `CALL` i `RET` działa tak długo, dopóki kod nie będzie zmieniał adresu powrotnego na stosie, co oznacza, iż większość programów powinna działać poprawnie przy takiej translacji, jednak dla pełnej zgodności nie jest możliwa zamiana `JSR` na `CALL`.

3.3. Mapowanie pamięci i trybów adresowania

Następnym krokiem jest określenie, w jaki sposób można zmienić odwołania do pamięci podczas translacji. Pod tym względem MOS 6502 jest bardzo prostym procesorem, adresy są 16-bitowe, nie zawiera żadnych mechanizmów pamięci wirtualnej, programista ma dostęp do całej przestrzeni adresowej, dlatego też najbardziej naturalnym mapowaniem wydaje się po prostu stworzenie tablicy bajtów rozmiaru 64 KiB i zamiana każdego dostępu do pamięci na dostęp do odpowiadającej komórki pamięci. Kod 3.1 przedstawia, jak może wyglądać przetłumaczenie dostępu do pamięci, używając bezwzględnego adresowania. Warto zaznaczyć, że ze względu na potrzebę generowania kodu relokowalnego, wszędzie używany jest względny adres.

MOS może jednak mieć wejście lub wyjście zmapowane w pamięci. Jest to bardzo wygodny sposób interakcji ze światem zewnętrznym, przykładowo, aby zmienić kolor tła na komputerze Commodore 64, należy zapisać wartość pod adresem `0xD021`. To miejsce w pamięci jest zmapowane na układ graficzny VIC-II, który może przechwycić taki zapis do pamięci i odpowiednio zmienić barwę. Dla procesora nie ma znaczenia, czy dany adres należy do pamięci RAM, ROM bądź jest przypisany do zewnętrznego urządzenia. Jednostka centralna nie musi być podłączona bezpośrednio do pamięci operacyjnej, zwykle w zależności od adresu, żądanie dostępu do pamięci trafia do różnych komponentów komputera. Z tego powodu pierwszy

Kod 3.1: Wczytanie zawartości komórki pamięci o adresie `0x20` do rejestru A

MOS 6502

```
1   LDA $20
```

x86-64

```
1   section .text
2     MOV R12B, [rel _memory + 0x20]
3   section .data
4     _memory: times 65536 db 0
```

pomysł prostego mapowania pamięci na tablicę nie jest wystarczający. W celu rozwiązania tego problemu wszystkie odczyty oraz zapisy do pamięci są delegowane do dodatkowej funkcji, która będzie odpowiednio emulowała oryginalne zachowanie i dopiero gdy dany adres faktycznie przynależy do pamięci operacyjnej — wtedy należy użyć tablicy.

3.4. Trudności i pułapki translacji

3.4.1. Dynamiczna zmiana operandów

Wydaje się, że najprostszym trybem adresowania do przetłumaczenia jest tryb natychmiastowy, instrukcja `LDX #$20` wczytuje liczbę `0x20` do rejestru `X`. W naiwny sposób można to przetłumaczyć jako `MOV R13B, 0x20`. Jednak nie zawsze taka konwersja będzie poprawna. Programy napisane na procesor MOS mają bowiem dostęp do całej pamięci, mogą odczytywać i zapisywać dane z dowolnego adresu, w szczególności możliwa jest zmiana kodu programu (w przypadku gdy kod nie znajduje się w pamięci tylko do odczytu). Kod 3.2 przedstawia przykładowy program, który dynamicznie w czasie wykonania zapisuje wartość `0x50` pod adres `0x5`, pod którym znajduje się operand instrukcji `LDX`. Jaką wartość powinien przyjąć rejestr `X` po wykonaniu tych trzech rozkazów? Mimo że oryginalnie była tam instrukcja `LDX #$20`, czyli wczytanie wartości `0x20` do rejestru, w wyniku modyfikacji piątej komórki pamięci, ta instrukcja zamieni się w `LDX #$50`, zatem rejestr `X` powinien zawierać liczbę `0x50`.

Kod 3.2: Program dynamicznie zmieniający operandy instrukcji

```
Adres    Pamięć       Mnemonik
$0000    A9 50        LDA #$50
$0002    85 05        STA $05
$0004    A2 20        LDX #$20
```

To oznacza, że w celu poprawnej konwersji, nawet ten najprostszy tryb adresowania powinien zostać zamieniony na odczyt operandu z pamięci. Należy jednak zauważyć, że nie jest to typowy wzorzec programowania (nawet w przypadku programów pisanych na procesor MOS), co więcej zwykle kod znajduje się w pamięci tylko do odczytu, wtedy ten problem nie występuje (poza dynamicznie generowanymi instrukcjami, patrz punkt 3.4.5). Z tego powodu tłumacz będący przedmiotem pracy nie wspiera dynamicznej zmiany operandów.

3.4.2. Mapowanie flag stanu procesora

Pomimo wielu lat różnicy, zarówno architektura x86-64, jak i MOS zawierają rejestr ze stanem procesora. Jest on wykorzystywany w instrukcjach skoków warunkowych. Co więcej, wiele

flag jest analogicznych w tych dwóch architekturach. W poprzednim podrozdziale celowo została pominięta kwestia emulacji rejestru flag, jednak jest to niezbędne do poprawnego uruchamiania oryginalnych programów.

Procesory z rodziny x86-64 mają następujące flagi (istotne z punku widzenia przedmiotu pracy):

- CF (Carry Flag) — informuje, czy w ostatniej operacji arytmetycznej nastąpiło przeniesienie (w przypadku dodawania) lub pożyczka (w przypadku odejmowania); w przypadku operacji przesunięcia bitowego-cyklicznego flaga przyjmuje wartość przesuniętego bitu,
- ZF (Zero Flag) — informuje, czy wynik ostatniej operacji wynosi 0,
- SF (Sign Flag) — informuje, czy wynik ostatniej operacji był ujemny,
- OF (Overflow Flag) — informuje, czy wynik ostatniej operacji ze znakiem zmieścił się w docelowym rejestrze.

Natomiast MOS 6502 ma następujące flagi arytmetyczne:

- C (Carry) — informuje, czy w ostatniej operacji arytmetycznej nastąpiło przeniesienie bądź pożyczka. Dodatkowo poza dodawaniem i odejmowaniem, ten rejestr jest również używany w instrukcjach przesunięcia bitowego-cyklicznego, nie jest on używany w przypadku instrukcji zwiększania-zmniejszania rejestru o jeden (`INC`, `DEC`, `INY`, `INX`),
- Z (Zero) — informuje, czy wynik ostatniej operacji wynosił zero,
- V (Overflow) — informuje, czy wynik ostatniego dodawania bądź odejmowania nie zmieścił się w bajcie,
- N (Negative) — informuje, czy wynik ostatniej operacji był ujemny.

Jak widać, istnieje bijekcja między tymi flagami. Jednakże zagłębiając się w szczegóły działania odpowiadających instrukcji, można dostrzec różnice w działaniu poszczególnych flag. Przykładowo alternatywa bitowa w przypadku procesora MOS modyfikuje flagi `Z` oraz `N`, natomiast odpowiadająca jej instrukcja architektury x86-64 oprócz odpowiadających bitów, zeruje również flagi przepełnienia i przeniesienia. To oznacza, że instrukcji $\texttt{ORA}_{\texttt{MOS}}$ nie można po prostu zamienić na instrukcję $\texttt{OR}_{\texttt{x86-64}}$, bowiem zmieniłoby to te dwie flagi. W ogólności nie jest możliwe tłumaczenie instrukcji jeden do jednego, na powyższym przykładzie widać, że podczas translacji konieczne są dodatkowe instrukcje, aby odtworzyć oryginalne zachowanie.

W przypadku flag procesora, dostępne są dwa rozwiązania:

1. użycie flag architektury x86-64 i tam, gdzie zachowanie jest inne — odpowiednio dostosowywać te flagi. Nie jest to jednak proste, nawet w pozornie łatwych przypadkach. Instrukcja $\texttt{CLV}_{\texttt{MOS}}$ zeruje flagę przepełnienia, nie istnieje jednak odpowiadająca jej instrukcja x86-64. Jedno z rozwiązań używa instrukcji $\texttt{PUSHF}_{\texttt{x86-64}}$ oraz $\texttt{POPF}_{\texttt{x86-64}}$, które wkładają i zdejmują rejestr `EFLAGS` ze stosu. Używając ich, można manipulować pojedynczymi bitami:

```
1       ; CLV
2       PUSHF
3       AND QWORD PTR [RSP], ~0x0800 ; zerowanie flagi OF (0x0800)
4       POPF
```

Należy najpierw zapisać aktualne flagi procesora na stosie, zmodyfikować jeden bit, a następnie przywrócić poprzedni stan. Niesie to za sobą poważne konsekwencje — dostęp do rejestru został zamieniony na dostęp do pamięci. Choć ze względu na stopień skomplikowania architektury x86-64 nie można jednoznacznie stwierdzić jak długo wykonuje się pojedyncza instrukcja, możliwe jest jednak zmierzenie pewnych parametrów. Na stronie internetowej [22] zostały przedstawione pomiary czasu wykonania różnych instrukcji[1], z której można odczytać, że wymienione wyżej instrukcje służące do dodawania i zdejmowania flag ze stosu dodają kilkukrotnie większe opóźnienie niż włożenie na stos wartości rejestru oraz kilkudziesięciokrotnie większe opóźnienie w porównaniu do przeniesienia wartości jednego rejestru do drugiego (por. tabela 3.3), ich przepustowość jest również dużo mniejsza. W tym podejściu jednak nieuniknione jest używanie tych rozkazów. Nie jest to też jedyny problem tego rozwiązania — jest ono bardzo podatne na błędy i pomyłki. Różne instrukcje w różny sposób modyfikują flagi, dlatego bardzo łatwo jest przypadkiem nadpisać ich stan.

2. Rezygnacja z mapowania flag, zamiast tego manualne symulowanie oryginalnego stanu procesora w jednym z rejestrów ogólnego przeznaczenia. Załóżmy, że ten stan będzie trzymany w rejestrze `R15B`, w takim samym formacie jak na MOS. W takim razie po każdej instrukcji, która zmienia flagi, trzeba zasymulować to działanie. Przykładowo, instrukcja wczytania wartości do rejestru zmienia odpowiednio flagi `Z` (czy wartość wynosi 0) oraz `N` (czy wartość jest ujemna). Zatem przykładowa przetłumaczona instrukcja `LDA 0` może wyglądać następująco:

```
1     ; LDA 0
2     MOV R12B, [rel _memory + 0] ; rejestr A jest zmapowany na R12B
3     TEST R12B, R12B
4
5     SETZ R8B              ; ustaw R8B na 1 jeśli R12B równe 0, 0 wpp.
6     SETS R9B              ; ustaw R9B na 1 jeśli R12B mniejsze od 0, 0
    ↪ wpp.
7
8     AND R15B, ~(0x80|0x2) ; zerowanie flagi N oraz Z
9
10    SHL R8B, 0x1          ; zamiana wartość logiczną w rejestrze R8B na
    ↪ odpowiadający jej drugi bit (flaga Z)
11    SHL R9B, 0x7          ; analogicznie rejestr R9B będzie
    ↪ przechowywał odpowiadający bit flagi N
12    OR R15B, R8B
13    OR R15B, R9B
```

To rozwiązanie generuje istotnie więcej instrukcji, jednak nie trzeba wtedy przejmować się efektami ubocznymi instrukcji i pozwala to uniknąć zbędnego dostępu do stosu.

[1]Pomiary przedstawione na stronie są wygenerowane za pomocą programu AIDA64. Dla różnych instrukcji sprawdzane są dwa wskaźniki: opóźnienie (ang. *latency*), czyli czas, po którym następna instrukcja tego samego typu, która zależy od poprzedniej może zacząć się wykonywać. Przepustowość (ang. *throughput*) to czas, po którym może zacząć wykonywać się następna niezależna instrukcja tego samego typu. Mierzenie polega na wielokrotnym wykonaniu tego samego rozkazu procesora, np. opóźnienie instrukcji `ADD` to ciąg `ADD rax, rax / ADD rax, rax / ADD rax, rax...`, badanie przepustowości tej instrukcji to test dodawania niezależnych rejestrów, np. `ADD rax, rax / ADD rbx, rbx / ADD rcx, rcx`.

Tabela 3.3: Porównanie opóźnienia i przepustowości instrukcji MOV, pary PUSH/POP oraz PUSHF/POPF dla procesora Intel(R) Core(TM) i9-10900K, źródło: [22]

Instrukcja	**Opóźnienie**	**Przepustowość**
MOV r64, r64	0.27 ns	0.09 ns
PUSH r64 + POP r64	1.17 ns	0.15 ns
PUSHF + POPF	6.58 ns	6.58 ns

Z dwóch wymienionych rozwiązań problemu, drugi pomysł ma więcej korzyści, jest to jednak powielanie pracy, która i tak zostaje wykonana przez procesor. Jest to szczególnie widoczne w przypadku najbardziej skomplikowanych instrukcji dodawania z przeniesieniem i odejmowania z pożyczką (`ADC`, `SBC`). Te dwie instrukcje wpływają na wszystkie wymienione wyżej flagi i działają w identyczny sposób zarówno na architekturze źródłowej MOS, jak i docelowej x86-64. Możliwe jest jeszcze trzecie rozwiązanie będące połączeniem pierwszego i drugiego. Pomysł polega na przeznaczeniu rejestru ogólnego przeznaczenia na symulowanie flag, ale tam gdzie to możliwe — należy próbować kopiować flagi x86-64. Zatem przetłumaczona instrukcja `ADC` najpierw kopiuje flagę przeniesienia do rejestru `EFLAGS`, następnie używa wbudowanego w x86-64 dodawania z przeniesieniem, a na końcu kopiuje flagi z powrotem do rejestru.

```
1    ; ADC #$01
2
3    CLC           ; wyzerowanie flagi przeniesienia CF
4    TEST R15B, 1 ; czy flaga C ustawiona
5    JZ noCarry
6    STC           ; ustawienie flagi CF jeśli flaga C była aktywna
7  noCarry:
8
9    ADC R12B, 1 ; dodawanie z przeniesieniem (R12B trzyma wartość A)
10                ;   skopiowanie flag do rejestru R15B
11   SETC CL      ; CL <- 1 jeśli było przeniesienie (flag C)
12   SETZ R8B     ; R8B <- 1 jeśli wynik równy 0 (flaga Z)
13   SETS R9B     ; R9B <- 1 jeśli wynik ujemny (flaga N)
14   SETO R10B    ; R10B <- 1 jeśli było przepełnienie (flaga O)
15
16   AND R15B, 0x3c ; wyzerowanie wszystkich flag C, Z, Z oraz O
17
18   SHL R8B, 0x1  ; odpowiednie przesunięcia bitowe,
19   SHL R9B, 0x7  ;   aby rejestry CL, R8B, R9B, R10B zamiast wartości
20   SHL R10B, 0x6 ;   logicznych przechowywały odpowiednie bity
21   OR R15B, CL   ; przeniesienie tych bitów do rejestru
22   OR R15B, R8B  ;   przeznaczonego na flagi
23   OR R15B, R9B
24   OR R15B, R10B
```

Taki sposób pozwala na zmniejszenie liczby linijek kodu, a jednocześnie unika problemów rozwiązania pierwszego. Zerowanie flagi przepełnienia (`CLV`) to teraz po prostu jedna koniunkcja bitowa `AND R15B, ~1`, z kolei zachowanie intelowskiej instrukcji `XOR` nie jest problemem — po prostu nie należy kopiować wyzerowanych flag przepełnienia i przeniesienia (bo na MOS tak się nie dzieje).

3.4.3. Odróżnienie danych od instrukcji

Na wejściu tłumacz MOS–x86-64 otrzymuje plik z programem, którego część to kod, a część — dane aplikacji. Struktura takich plików różni się w zależności od komputera, na który przeznaczona jest dana aplikacja, wiadomo jednak zawsze, od którego adresu zaczyna się wykonywanie programu. Problem jest jednak z określeniem, gdzie kończą się instrukcje, a zaczynają dane. Wszystkich kombinacji operandów i różnych trybów adresowania jest w sumie 151, co oznacza, że wybierając losowy bajt jest tylko około 40% szans, że na pewno nie będzie on kodował żadnego rozkazu (a to wyliczenie nie uwzględnia tego, że program może potencjalnie używać nieudokumentowanych rozkazów). Co prawda możliwe jest szacowanie prawdopodobieństwa, czy dany ciąg bajtów koduje zestaw instrukcji (jeśli kolejno w tym ciągu znajdują się bajty kodujące jakieś instrukcje), jednak szacowanie nie jest w tym przypadku wystarczające — nawet jedna nieprzetłumaczona instrukcja sprawi, że program w pewnym momencie może przestać prawidłowo działać. Mając jednak na uwadze ograniczenie procesora MOS do 64 kilobajtów pamięci (wynikające z 16-bitowych adresów), najlepszą możliwą metodą jest przetłumaczenie wszystkich bajtów, pomimo że część z nich w rezultacie okaże się danymi.

Nawet jeśli jest pewność, że dany bajt to instrukcja (na przykład podczas dekodowania adresu początkowego), program może skoczyć w środek takiej instrukcji, tak jak pokazano w kodzie 3.3. Z tego powodu istotne jest, aby wygenerować kod dla absolutnie każdego bajtu.

Kod 3.3: Ciąg bajtów `AD 0D 12 8D 18 4C 4C 01 00` może oznaczać dwa zupełnie różne zestawy instrukcji, w zależności od tego, czy dekodowanie zacznie się od pierwszego, czy drugiego bajtu

```
Adres     Pamięć          Mnemonik
$0000     AD 0D 12           LDA $120D
$0003     8D 18 4C           STA $4C18
$0006     4C 01 00           JMP $0001

$0001     0D 12 8D           ORA $8D12
$0004     18                 CLC
$0005     4C 4C 01           JMP $014C
$0008     00                 BRK
```

3.4.4. Tłumaczenie skoków

W procesorze MOS występują trzy rodzaje skoków:

- bezwzględne — adres, do którego przechodzi kontrola sterowania, jest zapisany w 16-bitowym operandzie (np. $\texttt{JMP}_{\texttt{MOS}}$, $\texttt{JSR}_{\texttt{MOS}}$),
- względne — operand zawiera przesunięcie względem aktualnego licznika programu, używany w przypadku instrukcji skoków warunkowych,
- pośrednie — operand zawiera adres, spod którego odczytany zostanie finalny adres, do którego następuje skok.

Skoki bezwzględne można przetłumaczyć bardzo łatwo. Choć zamienione instrukcje mają zupełnie inną długość niż oryginalne rozkazy, a poza tym generowany kod jest relokowalny,

to wystarczy każdą instrukcję opatrzyć odpowiednią etykietą, aby w trywialny sposób dokonywać skoku pod oryginalne miejsce. Skoki z przesunięciem niczym się nie różnią — podczas tłumaczenia takiej instrukcji dokładnie wiadomo, jaki w tym miejscu jest wskaźnik instrukcji programu, wystarczy dodać odpowiednie przesunięcie (obydwie liczby są znane podczas konwersji) i w analogiczny sposób skoczyć do etykiety.

Kod 3.4: Skok bezwzględny i względny przetłumaczony jako skok do odpowiedniej etykiety

MOS 6502

```
$0400: JMP $0600
...

$0600: BEQ 2   ; skok +2 względem
↪  0x601, czyli do 0x603
$0602: CLC

$6003: LDA #$1
```

x86-64

```
  JMP instr600
...
instr600:
  JZ instr603
instr602:
  CLC
instr603:
  MOV R12B, 1
```

Skoki pośrednie są bardziej skomplikowane, ponieważ docelowy adres, do którego należy przejść, jest znany dopiero podczas wykonywania programu (podczas kompilacji wiadomo tylko gdzie w pamięci będzie znajdował się adres, do którego nastąpi skok, jednak wartość tego adresu może się zmieniać w trakcie wykonywania). Z tego powodu nie da się tutaj wykorzystać sposobu z poetykietowaniem instrukcji (bowiem etykiety istnieją tylko na poziomie kodu asemblera). Skoki pośrednie są jednak niezbędnym elementem zestawu instrukcji, dlatego też należy znaleźć sposób na ich obsługę. W celu rozwiązania tego problemu, potrzebne jest mapowanie (dostępne podczas wykonywania programu) z oryginalnych adresów MOS na przetłumaczone adresy x86-64. Ze względu na zmienną długość kodów rozkazów x86-64 nie jest możliwe określenie wzoru takiej bijekcji. Proste i skuteczne rozwiązanie to stworzenie tablicy o rozmiarze `64 KiB * sizeof(void*)`, w której dla każdego adresu MOS znajdzie się wskaźnik na przetłumaczoną instrukcję. Takie rozwiązanie zostało zastosowane w projekcie i działa prawidłowo dla każdego pośredniego skoku.

3.4.5. Programy zmieniające swój kod

Zarówno procesor MOS 6502, jak i przedstawiciele rodziny x86-64 posiadają architekturę von Neumanna, czyli rozkazy jak i dane trzymane są w tej samej przestrzeni adresowej. Nic nie stoi na przeszkodzie, aby program modyfikował w locie swoje instrukcje. Z dzisiejszego punktu widzenia nie jest to pożądana cecha ze względów bezpieczeństwa. Nowsze procesory przeważnie mają możliwość oznaczenia stron pamięci jako albo z prawem do wykonania, albo z prawem do zapisu (ang. *write xor execute* — `W^X`), jednak możliwość dynamicznej generacji kodu jest jak najbardziej używana, chociażby w przypadku kompilacji bezpośrednio w trakcie wykonywania (ang. *just-in-time compilation*). W ramach tej pracy tłumaczenie programów odbywa się przed uruchomieniem (ang. *ahead-of-time*), jest to celowa decyzja, zatem samo-modyfikowalny kod stanowi poważną przeszkodę w tym projekcie. Jedyną możliwością obejścia tego problemu jest zawarcie w wygenerowanym programie samego konwertera, który musiałby wykrywać modyfikacje instrukcji i albo przełączałby się w tryb emulacji, albo dynamicznie konwertowałby zmienione rozkazy. Obydwa sposoby stoją jednak w sprzeczności z ideą pracy. Na szczęście obsługa dynamicznych instrukcji nie jest kluczowa do popraw-

nego działania większości programów. Ta technika była używana w rzadkich przypadkach, na przykład w zabezpieczeniach antypirackich.

3.4.6. Dokładność cykli procesora

Nie w każdym przypadku emulacji konieczne jest dokładne symulowanie prędkości oraz taktowania oryginalnego systemu. W przypadku nowych konsol gry nie są pisane z myślą o konkretnym procesorze, dlatego często nawet pożądane jest, aby emulacja działała tak szybko jak to możliwe, gdyż dzięki większej mocy urządzenia-gospodarza potencjalnie gra może działać płynniej niż na oryginalnym sprzęcie (więcej klatek na sekundę). Jednak w przypadku procesorów takich jak MOS taktowanie ma znaczenie, dlatego symulowanie oryginalnej prędkości jest istotnym problemem do rozwiązania. Procesor MOS 6502 może działać z prędkością od 1 MHz do 4 MHz, a poszczególne instrukcje zabierają więcej cykli niż odpowiednie rozkazy dzisiejszych procesorów. Każda instrukcja zajmuje konkretną liczbę cykli, zależną od jej typu i rodzaju adresowania (w zależności od stopnia złożoności i liczby odczytów bądź zapisów pamięci liczba cykli rośnie).

Translator może generować kod w trzech trybach:

- przetłumaczenie programu jako jeden długi ciąg instrukcji, bez sztucznego opóźniania wykonywania,
- przetłumaczenie programu jako jeden długi ciąg instrukcji, wraz z wywołaniami `nanosleep` w celu przybliżonego symulowania oryginalnej prędkości,
- przetłumaczenie programu w taki sposób, by wywołanie specjalnej funkcji `mosCycle()` wykonywało dokładnie jeden rozkaz z programu.

Pierwszy tryb jest pozostałością po początkowym etapie pracy nad tłumaczem, gdy testowana była poprawność pojedynczych instrukcji, a dokładność cyklowa nie była potrzebna. Nie oznacza to jednak, że nie ma dla niego zastosowania. Ciekawym eksperymentem jest konwersja programów bez sztucznego ograniczenia prędkości i porównanie czasu wykonania programu po przetłumaczeniu i w emulatorze (również bez ograniczania prędkości).

Drugi tryb powstał jako proste rozszerzenie istniejącego kodu. Znając taktowanie, które jest emulowane oraz liczbę cykli, które zajmuje dana instrukcja, podczas tłumaczenia, po każdym rozkazie można dodać wywołanie funkcji systemowej `nanosleep`. Takie rozwiązanie generuje kilka problemów, przede wszystkim nie uwzględnia ono czasu wykonywania rozkazu, zatem wprowadza większe opóźnienie niż należy. Poza tym usypianie wątku na kilka mikrosekund może być bardzo zawodne, wymaga to zbyt dużej dokładności. To rozwiązanie zostało początkowo zaimplementowane w konwerterze, ale sprawdzało się tylko w przypadku bardzo prostych programów napisanych na potrzeby projektu.

Ponadto takie podejście znacznie utrudnia synchronizację między poszczególnymi komponentami emulacji. Przykładowo, w konsoli Nintendo Entertainment System na każdy takt procesora przypadają trzy cykle układu graficznego (ang. *PPU - pixel processing unit*). Obydwa elementy są podłączone do tego samego zegara, zatem zawsze są zsynchronizowane.

W trzecim trybie generowana jest specjalna funkcja `mosCycle()`, której wywołanie wykonuje jeden kolejny rozkaz i zwracana jest liczba cykli, które upłynęły (bowiem jedna instrukcja zajmuje kilka taktów). W ten sposób odpowiedzialność za synchronizację i opóźnianie przenoszona jest poziom wyżej, co znacznie ułatwia pisanie kodu (kod 3.5).

Kod 3.5: Kod przedstawia sposób zsynchronizowanego wywołania jednego kroku procesora oraz układu graficznego

```
void nes_step() {
    int cycles = mosCycle();
    for (int i = 0; i < cycles * 3; ++i) {
        step_ppu();
    }
}
```

3.4.7. Przerwania

W podrozdziale 1.2.4 zostały opisane przerwania dostępne w procesorze. Poza programowym przerwaniem `BRK`, pozostałe z nich są wywoływane przez zewnętrzne sygnały, więc tłumacz musi zapewnić możliwość emulowanego wywołania takiego przerwania. W przypadku wystąpienia przerwania należy przystąpić do jego obsługi zaraz po zakończeniu wykonywania aktualnej instrukcji. To oznacza, że w wynikowym kodzie, po każdym przetłumaczonym rozkazie należy umieścić procedurę odpowiedzialną za sprawdzenie, czy wystąpiło przerwanie i w przypadku pozytywnej odpowiedzi zapisać na stosie adres powrotu, flagi procesora i zmienić licznik instrukcji na odpowiedni adres z wektora przerwań. Takie rozwiązanie dosyć dokładnie symuluje oryginał.

3.5. Optymalizowanie generowanego kodu

Optymalizowanie wygenerowanego kodu jest utrudnione z powodu możliwych skoków w dowolne miejsce programu. Z tego powodu nie jest możliwe robienie założeń o poprzednich instrukcjach. Kod 3.6 przedstawia przetłumaczony ciąg instrukcji $\texttt{INX}_{\texttt{MOS}}$ (zwiększenie rejestru `X`). Przetłumaczenie tych trzech instrukcji na jedną instrukcję `ADD R14B, 3` jest niepoprawne, ponieważ w dowolnym momencie program może zmienić wskaźnik instrukcji na adres `0x402`, w wyniku czego inkrementacja odbędzie się tylko jeden raz. Ponadto, ze względu na możliwe skoki pośrednie, nie jest możliwe (w trakcie statycznej translacji) stwierdzenie, pod które adresy będą dokonywane skoki. Jednym z rozwiązań tego problemu jest translacja wszystkich możliwych kombinacji optymalizowanego bloku, jak pokazano w kodzie 3.7. Taki sposób dodaje jednak znaczną ilość kodu oraz przede wszystkim wprowadza skoki, których wcześniej można było uniknąć. Z tego powodu tłumacz nie dokonuje optymalizacji pomiędzy kilkoma instrukcjami.

Kod 3.6: Przetłumaczone instrukcje inkrementacji rejestru `X` oraz nieprawidłowa optymalizacja

MOS 6502

```
1  $0400: INX
2  $0401: INX
3  $0402: INX
```

x86-64

```
1  instr0400:  INC R14B
2  instr0401:  INC R14B
3  instr0402:  INC R14B
```

nieprawidłowy kod x86-64

```
1  instr0400:  ADD R14B, 3
2  instr0401:  NOP
3  instr0402:  NOP
```

Tłumacz optymalizuje jednak generowany kod w ramach pojedynczych instrukcji. Kod zaprezentowany w przykładzie powyżej, został już zoptymalizowany pod względem symulo-

Kod 3.7: Alternatywny sposób tłumaczenia kodu 3.6

```
instr0400:
  ADD R14B, 3
  JMP instr0403

instr0401:
  ADD R14B, 2
  JMP instr0403

instr0402:
  INC R14B

instr403:
  ...
```

wanych flag. Niezoptymalizowana wersja instrukcji zwiększenia rejestru `X` o jeden jest przedstawiona w kodzie 3.8. Rozkaz ten modyfikuje odpowiednio flagi stanu procesora, zatem zgodnie z opisem w podrozdziale 3.4.2 należy to zachowanie symulować. Jednak nie zawsze jest to konieczne. Jeśli następna instrukcja również nadpisze te flagi, a w międzyczasie żadna inna instrukcja ich nie odczytała, wtedy możliwe jest całkowite usunięcie kodu odpowiedzialnego za symulowanie flag. Oznacza to, że ewentualna możliwość optymalizacji zależy od „przyszłych" instrukcji. Dla każdej instrukcji procesora MOS tłumacz zawiera listę odczytywanych i zapisywanych flag stanu procesora. Jeśli modyfikowane flagi i tak zostaną nadpisane, to nie są w ogóle symulowane. Ponownie pewnym utrudnieniem są tutaj skoki pośrednie. Sam skok nie odczytuje ani nie modyfikuje flag, jednak ze względu na brak możliwości stwierdzenia, jakie instrukcje będą dalej wykonywane (a co za tym idzie — jakie flagi będą odczytywane), należy założyć pesymistyczny przypadek i nie optymalizować symulowania flag, gdy wśród „przyszłych" instrukcji znajduje się skok.

Kod 3.8: Przetłumaczona instrukcja $\texttt{INX}_{\texttt{MOS}}$ bez żadnych optymalizacji

MOS 6502

```
$0400: INX
$0401: ...
```

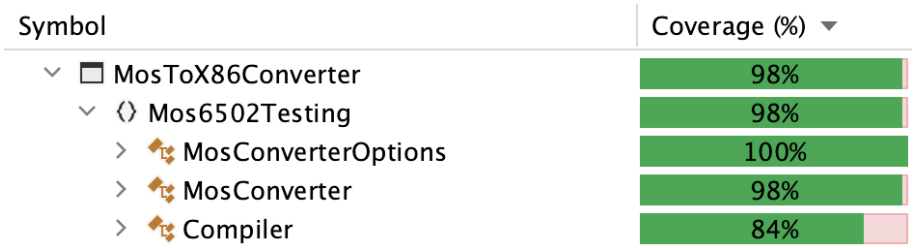
x86-64

```
instr0400:
  INC R14B
  SETZ R8B        ; R8B <- 1 jeśli wynik równy 0 (flaga Z)
  SETS R9B        ; R9B <- 1 jeśli wynik ujemny (flaga N)
  AND R15B, 0x7d  ; wyzerowanie flag Z i N
  SHL R8B, 0x1    ; odpowiednie przesunięcie bitowe, aby
  SHL R9B, 0x7    ;  rejestry R8B i R9B zamiast wartości
                  ;  logicznych, przechowywały
                  ;  odpowiadające bity
  OR R15B, R8B    ; przeniesienie tych bitów do rejestru
  OR R15B, R9B    ;  flag procesora
instr0401:
  ...
```

3.6. Weryfikacja i testowanie wyniku tłumaczenia

Generowanie natywnego kodu, czyli zagadnienie stanowiące trzon przedmiotu niniejszej pracy magisterskiej, jest mocno narażone na błędy, które w praktyce trudno zdiagnozować. Pomijając drobne pomyłki, przez które wynikowy program całkowicie nie da się uruchomić, zwykle wygenerowane instrukcje są poprawnymi rozkazami architektury x86-64, z prawidłowymi dostępami do pamięci, jednak mogą zawierać błędy związane z innym zachowaniem niż oryginał. Dodatkowo takie problemy trudno jest śledzić, gdyż często dają one o sobie znać dopiero w następnych instrukcjach. Przykładowo — błędnie symulowane flagi procesora nie wpływają na prawidłowe działanie samej instrukcji, dopiero później, gdy pojawi się instrukcja skoku, potencjalnie nastąpi skok w złe miejsce programu, co ponownie nie musi dać jednoznacznych objawów o wystąpieniu problemu.

Z tego powodu bardzo ważne było dokładne testowanie translatora. Projekt zawiera szereg testów jednostkowych oraz integracyjnych. Każda instrukcja zawiera odpowiadający jej test, w którym sprawdzane jest jej poprawne działanie, włączając w to flagi procesora. Dzięki temu wielokrotnie udało się wykryć na wczesnym etapie pomyłki w kodzie. Sumarycznie moduł tłumaczący jeden kod maszynowy na drugi ma ponad 90% linii pokrytych testami (rys. 3.1).



Rysunek 3.1: Pokrycie testami głównego projektu tłumacza

Oprócz tego, w sieci można znaleźć wiele zestawów testów przygotowanych specjalnie, aby pokrywały jak najwięcej przypadków. Projekt był testowany na dwóch takich zestawach: *6502 Functional Tests* [14] oraz *Nestest* [15]. Obydwa testy sprawdzają również emulację dynamicznie generowanego kodu. Z przyczyn opisanych w podrozdziale 3.4.1 ten konwerter tego nie wspiera, dlatego też kod *Nestest* został nieznacznie zmieniony (w całym teście dynamicznie generowane były jedynie dwie instrukcje, zostały one wpisane na stałe w pamięci RAM), natomiast *6502 Functional Test* ma możliwość wyłączenia sprawdzania dynamicznego kodu. Poza tymi dwoma ograniczeniami obydwa testy przechodzą w 100%. Użycie złożonych i kompleksowych testów pozwoliło wykryć kilka błędów w konwerterze na wczesnym etapie prac.

Ostatecznym „testem” jest translacja i uruchomienie prawdziwych programów napisanych na komputery z procesorem MOS 6502, rezultaty znajdują się w rozdziale 4.

Rozdział 4

Wyniki tłumaczenia

4.1. Powstałe elementy pracy

Translacja samych instrukcji nie jest wystarczająca do wytworzenia używalnego pliku wykonywalnego, a format plików źródłowych różni się w zależności od komputera, na który zostały przygotowane. Z uwagi na powyższe, tłumacz powstały w ramach pracy funkcjonuje przede wszystkim jako biblioteka. W oparciu o nią mogą zostać napisane konwertery plików konkretnych platform; jednym z nich jest narzędzie `NesConverter`, które przetwarza pliki z konsoli Nintendo Entertainment System na aplikacje architektury x86-64. Poza nim zostały stworzone następujące elementy:

- biblioteka tłumacząca MOS 6502 — x86-64

  Główny element niniejszej pracy, czyli moduł tłumaczący. Na wejściu biblioteka otrzymuje tablicę bajtów z kodem maszynowym programu przeznaczonego na procesor MOS 6502, opcje tłumaczenia (ich opis jest w podrozdziale 4.1.1). Na wyjściu zwracany jest przetłumaczony kod w asemblerze x86-64 (w formacie Intel). Asemblacja jest dokonywana za pomocą zewnętrznego narzędzia obsługującego typową składnię, takie jak `nasm` bądź `yasm`.

- asembler oraz deasembler MOS 6502

  Choć dostępnych jest wiele otwartoźródłowych programów służących do asemblacji oraz deasemblacji kodu na procesor 6502, z uwagi na przedmiot niniejszej pracy zasadne i proste było stworzenie takiego narzędzia od zera. Jest ono następnie używane w formie biblioteki we właściwym tłumaczu. Stworzony asembler obsługuje wszystkie oficjalne kody instrukcji oraz wspiera standardową składnię kodu maszynowego, w tym makra (także parametryzowane). Testy pokrywają ponad 80% kodu źródłowego, w tym przetestowane są wszystkie możliwe kombinacje instrukcji i trybów adresowania. Ponadto mogą one funkcjonować jako niezależne narzędzia uruchamiane z linii poleceń, co było szczególnie przydatne podczas testowania projektu.

- interpreter MOS 6502 w C#

  Głównym tematem pracy jest statyczne tłumaczenie, jednak w celu łatwiejszego testowania powstał również interpreter MOS 6502 w języku C#. W dalszej części pracy jest on porównywany z przetłumaczonymi programami. Należy zaznaczyć, że ze względu na narzut wydajności języków używających maszyny wirtualnej, oczywiste jest, że taki interpreter będzie wolniejszy niż interpreter napisany w języku kompilowanym do kodu maszynowego.

4.1.1. Opcje tłumacza

Tłumacz ma następujące opcje, które wpływają na generowany kod:

- tłumaczenie instrukcji $\texttt{JSR}_{\texttt{MOS}}$ na instrukcję $\texttt{CALL}_{\texttt{x86-64}}$ (`UseNativeCallAsJsr`)

  domyślnie instrukcja wywołania procedury jest symulowana za pomocą instrukcji skoku bezwarunkowego $\texttt{JMP}_{\texttt{x86-64}}$ pod adres procedury, a instrukcja powrotu jest tłumaczona na skok bezwarunkowy na adres przechowywany na symulowanym stosie; przełączenie tej opcji powoduje przetłumaczenie instrukcji $\texttt{JSR}_{\texttt{MOS}}$ oraz $\texttt{RTS}_{\texttt{MOS}}$ na odpowiednio $\texttt{CALL}_{\texttt{x86-64}}$ oraz $\texttt{RET}_{\texttt{x86-64}}$; ten sposób działa poprawnie, o ile program nie zacznie zmieniać adresów powrotu na symulowanym stosie, bowiem taka zmiana nie będzie miała odzwierciedlenia w prawdziwym stosie x86-64; gdy jednak stos nie jest dynamicznie zmieniany, tłumaczenie na odpowiedniki architektury x86-64 jest poprawne,

- tryb odpluskwiania oraz porównywania wyników (`DebugMode`)

  włączenie trybu odpluskwiania; powoduje dodanie wywołań funkcji o definicji `void(*)(struct mos_state_t state)` (struktura przedstawiona w kodzie 4.1) po każdej przetłumaczonej instrukcji; w ten sposób programista może przeanalizować działanie programu krok po kroku; ten przełącznik jest również używany w testach tłumacza do weryfikacji stanu procesora po każdej instrukcji,

- wykorzystywanie flag x86-64 w generowanym kodzie (`UseNativeFlags`)

  tak jak opisano w podrozdziale 3.4.2, flagi procesora mogą być przetłumaczone na dwa sposoby; domyślnie ta opcja jest włączona i tam gdzie to jest możliwe, generowany kod wykorzystuje flagi x86-64, wyłączenie tej opcji powoduje przeliczenie wartości rejestru flag po każdej instrukcji; ten przełącznik nie wpływa na poprawność generowanego kodu i w każdej sytuacji możliwe jest zarówno włączenie, jak i wyłączenie tej opcji,

- tryb symulacji instrukcja po instrukcji (`CycleMethod`)

  domyślnie tłumacz generuje kod jako jeden długi ciąg instrukcji, wykonywanych jedna po drugiej, w ten sposób wynikowy program jest wydajny, lecz niemożliwe jest synchronizowanie emulowanego procesora z innymi emulowanymi podzespołami (utrudnione jest także symulowanie przerwań); włączenie tej opcji powoduje wyemitowanie funkcji `struct mos_state_t mosCycle(struct mos_state_t state)`, która dla danego stanu podanego w argumencie wykonuje jedną instrukcję i zwraca zaktualizowany stan; to umożliwia precyzyjne synchronizowanie i opóźnianie wykonywania programu,

- sposób emulacji pamięci (`UseArrayAsMem`)

  gdy ta opcja jest włączona, na początku wykonywania programu następuje alokacja tablicy rozmiaru 64 KiB, pełniącej rolę symulowanej pamięci RAM; to rozwiązanie jest szybkie (dostęp do pamięci to po prostu dostęp do tablicy), jednak nie pozwala na symulację pamięci zmapowanej; w tym celu należy wyłączyć tę opcję i dostarczyć funkcje `unsigned char getValue(unsigned short address)` oraz `void setValue(unsigned short address, unsigned char value)`, które będą symulowały pamięć; te funkcje mogą odpowiednio symulować poszczególne regiony pamięci, na przykład traktując wybrany adres jako wejście kontrolera do gier.

Kod 4.1: Struktura `mos_state_t` reprezentująca stan emulowanego procesora; zajmuje ona osiem bajtów, zatem mieści się w jednym rejestrze x86-64, co ułatwia jej przekazywanie

```
struct __attribute__((__packed__)) mos_state_t
{
    unsigned char Cycles; // liczba cykli ostatniej instrukcji
    unsigned char IP_low; // dolny bajt adresu instrukcji
    unsigned char IP_high;// górny bajt wskaźnika instrukcji
    unsigned char SP;     // dolne osiem bitów rejestru stosu
    unsigned char Flags;  // flagi procesora MOS
    unsigned char Y;      // wartość rejestru Y
    unsigned char X;      // wartość rejestru X
    unsigned char A;      // wartość rejestru A
};
```

4.1.2. Wpływ opcji tłumaczenia na czas działania programów

Tabela 4.1 przedstawia czasy działania przetłumaczonych testowych programów w zależności od wybranych opcji translacji. Takie porównanie ma jedynie sens bez sztucznego opóźniania czasu wykonywania instrukcji. Jest to możliwe w przypadku tych programów, bowiem nie zależą one od innych emulowanych komponentów (na przykład układu graficznego). Wszystkie testy były uruchomione na komputerze wyposażonym w procesor Intel Core i5-4570, każdy został powtórzony 25 razy, a wyniki zostały uśrednione.

- `test NWD` — program liczący największy wspólny dzielnik liczb z zakresu `0x1`–`0x3FF` za pomocą metody odejmowania,
- `test dodawanie` — program dodający w pętli dwie szesnastobitowe liczby,
- `mnożenie macierzy` — program podnosi macierz rozmiaru 120 na 120 do kwadratu,
- `MOS 6502 Functional Tests` — zestaw testujący wszystkie instrukcje MOS 6502, przygotowany przez Klausa Dormanna [14].

Największy przyrost wydajności jest powodowany poprzez włączenie dostępu do pamięci jako tablicy bajtów, zatem tam, gdzie to możliwe, warto używać tego przełącznika (jednak, tak jak zostało wskazane w podrozdziale 3.3, nie zawsze jest to możliwe). Pozostałe dwie opcje dają mniejszy, choć widoczny zysk wydajnościowy, dodatkowo zależny od konkretnego programu (test NWD nie używa instrukcji wywołania procedury, dlatego opcja *JSR jako CALL* nie ma żadnego wpływu). W ogólności jednak uzyskane rezultaty są zgodne z oczekiwaniami, czyli bezpośrednie dostępy do pamięci są szybsze niż dostępy poprzez zewnętrzne funkcje (narzut związany w wywołaniem procedury), a natywne konstrukcje takie jak użycie instrukcji $\texttt{CALL}_{\texttt{x86-64}}$ i oryginalnych flag architektury x86-64 są wydajniejsze niż ich symulowanie.

4.1.3. Porównanie wydajności przetłumaczonych programów z innymi metodami

Znając optymalny zestaw opcji tłumaczenia, możliwe jest porównanie translacji z innymi metodami emulacji procesora. Rozwiązanie zostało porównane z interpreterem napisanym

Tabela 4.1: Porównanie wpływu opcji translacji na czas działania przetłumaczonych programów

OPCJE			**TEST**		
`JSR`$_{\texttt{MOS}}$ jako `CALL`$_{\texttt{x86-64}}$	Pamięć jako tablica bajtów	Optymalizacja flag	Test NWD	MOS 6502 Functional Tests	Test „Dodawanie”
Nie	Nie	Nie	1,60 s (± 5 ms)	94,35 ms (± 2 ms)	6,53 s (± 24 ms)
Nie	Nie	Tak	1.57 s (± 4 ms)	91,46 ms (± 1 ms)	6,44 s (± 25 ms)
Tak	Nie	Nie	1.60 s (± 4 ms)	91,80 ms (± 3 ms)	5,13 s (± 17 ms)
Tak	Nie	Tak	1,57 s (± 36 ms)	88,60 ms (± 2 ms)	5,19 s (± 195 ms)
Nie	Tak	Nie	0.50 s (± 4 ms)	29,99 ms (± 1 ms)	0,80 s (± 4 ms)
Nie	Tak	Tak	0.45 s (± 3 ms)	26,21 ms (± 1 ms)	0,76 s (± 4 ms)
Tak	Tak	Nie	0.50 s (± 4 ms)	29,00 ms (± 1 ms)	0,66 s (± 4 ms)
Tak	Tak	Tak	0.45 s (± 4 ms)	26,58 ms (± 1 ms)	0,64 s (± 4 ms)

w języku C# oraz interpreterem autorstwa Gianluca'i Ghettiniego opublikowanym na licencji MIT na platformie GitHub [20] (w testach skompilowany zarówno z, jak i bez optymalizacji, odpowiednio flagi kompilacji `-O3` oraz `-O0`). Ponadto trzy programy testowe przygotowane specjalnie w ramach tej pracy — „NWD”, „dodawanie” oraz „mnożenie macierzy” zostały także napisane w języku C, aby porównać narzut związany z translacją względem natywnie skompilowanych programów. Należy jednak zaznaczyć, że te programy zostały napisane w taki sposób, aby wykorzystywać tylko te instrukcje, które są dostępne także w procesorze MOS. Porównanie ma na celu sprawdzenie narzutu tłumaczenia, a nie tego jak wydajny może być analogiczny program napisany od zera i skompilowany na architekturę x86-64. Liczenie największego wspólnego dzielnika jest bardzo proste i szybkie za pomocą dzielenia modulo, jednak ta operacja arytmetyczna nie istnieje w zestawie instrukcji MOS, dlatego też algorytm w teście używa wolniejszej metody liczenia największego wspólnego dzielnika za pomocą odejmowania. Podobnie w przypadku mnożenia, które nie jest dostępne w procesorze MOS, zatem test symuluje to działanie dodawaniem. Co więcej, procesor MOS posiada tylko jeden rejestr, na którym mogą być wykonywane operacje arytmetyczne, zatem w praktyce po każdej operacji wartość rejestru jest zapisywana do pamięci, dlatego aby dostępy do pamięci nie były optymalizowane, zmienne w testowych programach w C mają modyfikator `volatile`. Ponadto procesor MOS operuje na ośmiobitowych porcjach danych, dlatego przepisany program również używa tylko jednobajtowych typów danych.

Tabela 4.2: Porównanie wydajności interpretacji, translacji oraz natywnie skompilowanych testów; pogrubiony wiersz przedstawia wyniki przedmiotu pracy

	Test NWD	Test „Dodawanie”	Mnożenie macierzy	MOS 6502 Functional Tests
Interpreter w C#	64,44 s	80,54 s	35,50 s	2,37 s
Interpreter w C++ (-O0)	28,70 s	39,74 s	16,74 s	1,16 s
Interpreter w C++ (-O3)	7,31 s	11,51 s	4,17 s	0,29 s
Przetłumaczony program	**0,45 s**	**0,64 s**	**0,42 s**	**0,027 s**
Natywna wersja x86-64 (-O0)	0,27 s	0,64 s	2,41 s	-
Natywna wersja x86-64 (-O3)	0,14 s	0,48 s	0,32 s	-

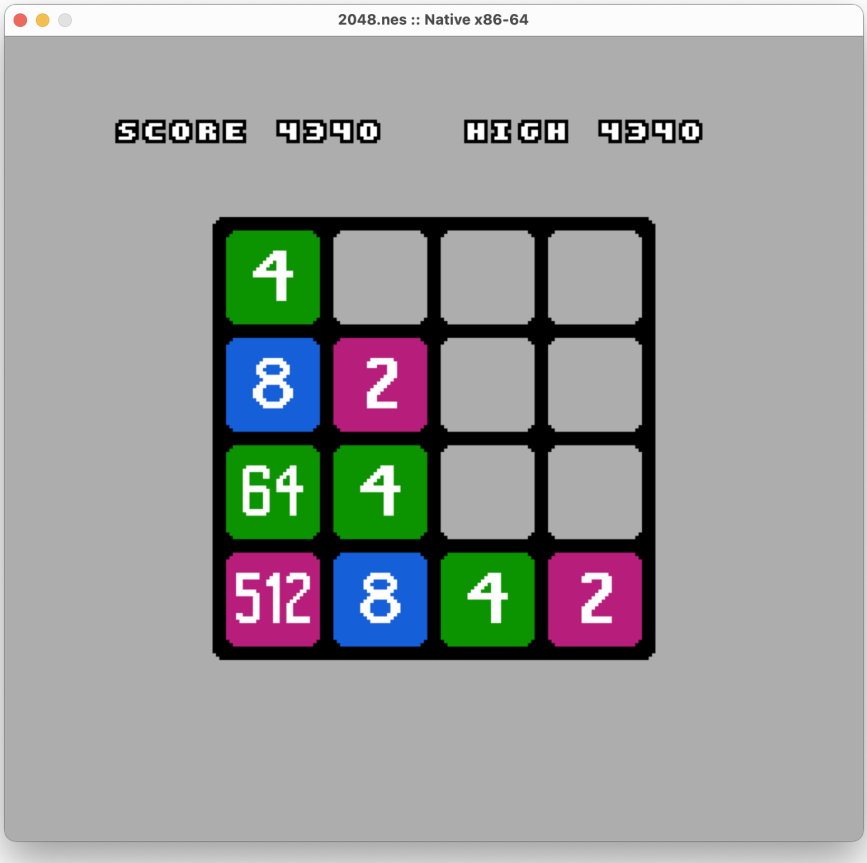
Wyniki porównania są pomyślne (tabela 4.2). Zgodnie z przewidywaniami interpreter napisany w języku C# jest znacznie wolniejszy od interpretera napisanego w C++, a najszybsze są testy przepisane w C. Najważniejszy jest jednak wynik przetłumaczonych programów; w przedstawionych testach są one nawet o rząd wielkości szybsze niż ich interpretowane odpowiedniki i tylko około od półtora do trzech razy wolniejsze od natywnie skompilowanych testów (w zależności od programu). To oznacza, że jeden z celów pracy, czyli sprawdzenie czy tłumaczone programy są istotnie wydajniejsze niż interpretowane, został spełniony. Tłumaczenie pozwala osiągnąć znacznie lepsze rezultaty niż metoda emulacji poprzez interpretowanie instrukcji.

4.2. Wykorzystanie translacji instrukcji do uruchamiania wybranych programów

Ostatecznym testem i celem pracy jest wykorzystanie tłumacza do konwersji prawdziwych programów. Jako platformę testową została wybrana konsola do gier Nintendo Entertainment System. Znajduje się w niej czip Ricoh 2A03, taktowany częstotliwością 1,79 MHz (w systemie PAL układ ma taktowanie 1,66 MHz), na który składa się procesor zgodny z MOS 6502 oraz układ dźwiękowy, który obsługuje łącznie pięć kanałów. Grafika generowana jest przez układ Ricoh 2C02 taktowany trzykrotną częstotliwością procesora. W roku wydania konsoli (1983) był to jeden z najbardziej zaawansowanych układów graficznych w komputerach domowych. W jednej chwili na ekranie może wyświetlać do 64 duszków (ang. *sprite*), generowana grafika ma rozdzielczość 256 na 240 pikseli, a w jednym czasie może być wyświetlone 25 kolorów (z 54 kolorowej palety). Standardowy kontroler ma osiem przycisków, a jednocześnie mogą być podłączone dwa kontrolery. Interakcja ze wszystkimi układami odbywa się poprzez pamięć zmapowaną. Poszczególne adresy odpowiadają za komunikację z układem graficznym, układem dźwiękowym oraz urządzeniami wejściowymi.

Poprawne emulowanie konsoli wymaga zadbania o odpowiednią synchronizację poszczególnych układów. Na każdy cykl procesora przypadają trzy cykle układu graficznego. Niedokładne symulowanie może skutkować nieprawidłową prędkością działania, błędami graficznymi, dźwiękowymi, zawieszeniem symulacji bądź brakiem jakiegokolwiek wyjścia wideo.

Emulacja konsoli wymaga co najmniej emulowania procesora, układu graficznego i urządzeń wejścia (układ dźwiękowy może zostać pominięty), natomiast praca ta skupia się jedynie na nowym sposobie symulowania tylko tego pierwszego. Dlatego w celu przetestowania tłumacza, został napisany prosty emulator tej konsoli, który symuluje wymienione wcześniej elementy. Do tłumaczenia została użyta opcja symulacji instrukcja po instrukcji, dzięki temu możliwa jest synchronizacja tej części emulatora, a sama integracja z pozostałymi komponentami była bardzo prosta. Konwersja została przetestowana na grze „2048” (rys. 4.1). Przetłumaczona gra działa dokładnie tak jak w innych emulatorach oraz na oryginalnym sprzęcie.



Rysunek 4.1: Przetłumaczony za pomocą narzędzia `NesConverter` klon gry „2048” napisany przez Valtteriego Heikkila, wydany na licencji MIT na konsolę Nintendo Entertainment System [21]; gra działa w pełni poprawnie

Rozdział 5

Podsumowanie

Tłumaczenie instrukcji jest jedną z dwóch najczęściej stosowanych metod uruchamiania programów przeznaczonych na inną architekturę niż architektura komputera gospodarza. Drugim rozwiązaniem jest użycie interpretera rozkazów procesora. Interpretacja zwykle jest łatwiejsza w implementacji, lecz nie tak wydajna jak translacja. W ramach niniejszej pracy magisterskiej została stworzona aplikacja tłumacząca programy przeznaczone na procesor MOS 6502 na architekturę x86-64. Tłumacz dokonuje deasemblacji, a następnie konwertuje instrukcje na rozkazy docelowej architektury. Po asemblacji i linkowaniu przetłumaczony program jest gotowy do uruchomienia bez potrzeby użycia dodatkowych pomocniczych programów.

Tłumaczenie aplikacji MOS 6502 na x86-64 zostało przetestowane zarówno na specjalnie przygotowanych programach testowych, jak i na prawdziwych aplikacjach przeznaczonych na konsolę do gier Nintendo Entertainment System. Efekty tłumaczenia są bardzo zadowalające — przetłumaczone programy działają tak jak na oryginalnym procesorze, a ponadto, zgodnie z oczekiwaniami, przedstawiona metoda jest bardziej wydajna niż klasyczne emulatory oparte na interpretacji rozkazów. Przetłumaczone programy są kilkukrotnie szybsze niż te emulowane za pomocą interpreterów kompilowanych do kodu maszynowego (a względem interpretera napisanego w języku C# — nawet kilkunastokrotnie szybsze). Co więcej, porównanie specjalnie przygotowanych w asemblerze programów dla procesora MOS 6502 oraz ich analogicznych wersji w języku C pokazało, że przetłumaczone programy są tylko od półtora do trzech razy wolniejsze od natywnie skompilowanych testów. Zatem cele postawione na początku pracy zostały osiągnięte.

W ramach pracy tłumacz został przetestowany na grze przeznaczonej na konsolę do gier wideo Nintendo Entertainment System. Nie jest to jedyne urządzenie z tym procesorem — jest on używany także w wielu innych komputerach. Użycie tłumacza do stworzenie emulatorów innych systemów jest bardzo łatwe i nie wymaga żadnych zmian w samym translatorze.

Tłumaczenie instrukcji może odbywać się statycznie (przed uruchomieniem programu), bądź dynamicznie (w trakcie wykonywania aplikacji). W tej pracy magisterskiej został zaimplementowany pierwszy sposób. Ta technika nie pozwala jednak na uruchamianie programów, które modyfikują swój kod w trakcie wykonywania. Jednym z możliwych kierunków rozwoju projektu jest dodanie dynamicznego tłumaczenia — w trakcie działania programu, gdy zostanie odkryty jeszcze nieprzetłumaczony kod.

Kolejnym etapem rozwoju projektu może być dodanie nowej docelowej architektury — architektury ARM, która zdobywa coraz większą popularność na komputerach osobistych. Ponadto, wykorzystując przedstawioną tu wiedzę, interesującym kierunkiem jest napisanie tłumaczy innych, nowszych procesorów, jednak jest to temat na następne prace badawcze.

Dodatek A

Lista rozkazów

Praca w wielu miejscach odnosi się do instrukcji procesora MOS 6502. Poniżej znajduje się lista wszystkich legalnych rozkazów wraz z opisem ich działania i pseudokodem dla lepszego zrozumienia działania.

W pseudokodzie używam nazewnictwa rejestrów widocznego w tabeli A.1. Zapis `L <- R` oznacza przypisanie wyliczonej wartości wyrażenia po prawej strony do lewej strony. Symbol `memory` odnosi się do wartości operandu wynikającej z rozkazu.

Tabela A.1: Oznaczenie rejestrów w pseudokodzie

Pseudokod	**Rejestr**
A	Akumulator
X	Rejestr X
Y	Rejestr Y
SP	Wskaźnik końca stosu
SR	Rejestr flag procesora
PC	Licznik rozkazów

Instrukcja ADC

Modyfikowane flagi: N, Z, V, C
Dodawanie z przeniesieniem. Procesor MOS może posiadać tryb operacji dziesiętnych, czyli tryb, w którym wysokie cztery bity oznaczają cyfrę dziesiątek, a niskie cztery bity cyfrę jedności. Wtedy bajt `0x28` reprezentuje liczbę dziesiętną `28` (a nie liczbę dziesiętną `40`).

```
1  tmp <- memory + A + (if C then 1 else 0)
2  Z <- (tmp % 255) == 0
3  if D:
4      if (A & 0xF) + (memory & 0xF) + (C ? 1 : 0) > 9:
5          tmp <- tmp + 6
6      N <- (tmp & 0x80) != 0
7      V <- ((A ^ memory) & 0x80) == 0 && ((A ^ tmp) & 0x80) != 0
8      if tmp > 0x99:
9          tmp <- tmp + 96
10     C <- tmp > 0x99
11 else:
12     N <- (tmp & 0x80) == 0x80
```

```
    V <- ((A ^ memory) & 0x80) == 0 && ((A ^ tmp) & 0x80) != 0
    C <- tmp > 255

A <- tmp % 255
```

Instrukcja AND

Modyfikowane flagi: N, Z
Koniunkcja bitowa komórki pamięci z akumulatorem. Wynik jest zapisywany do rejestru `A`.

```
A <- A & memory
```

Instrukcja ASL

Modyfikowane flagi: N, Z, C
Przesunięcie bitowe w lewo o jeden. Najmniej znaczący bit jest uzupełniany zerem.

```
memory <- memory << 1
```

Instrukcja BCC

Skok, gdy brak przeniesienia.

```
if (!C):
    PC <- operand
```

Instrukcja BCS

Skok, gdy wystąpiło przeniesienie.

```
if (C):
    PC <- operand
```

Instrukcja BEQ

Skok, gdy wynik zerowy.

```
if (Z):
    PC <- operand
```

Instrukcja BIT

Modyfikowane flagi: N, Z, V
Koniunkcja bitowa akumulatora z wartością. Jeśli wszystkie bity są rozłączne, flaga `Z` jest ustawiana, w przeciwnym wypadku - zerowana. Dodatkowo flaga `N` przyjmuje wartość najwyższego bitu, a flaga `V` - bitu szóstego.

```
Z <- A & value == 0
N <- value & 0x80
V <- value & 0x40
```

Instrukcja BMI

Skok, gdy wynik ujemny.

```
if (N):
    PC <- operand
```

Instrukcja BNE

Skok, gdy wynik niezerowy.

```
if (!Z):
    PC <- operand
```

Instrukcja BPL

Skok, gdy wynik dodatni.

```
if (!N):
    PC <- operand
```

Instrukcja BRK

Wywołanie przerwania programowego — na stos wkładany jest licznik instrukcji powiększony o jeden, następnie aktualne flagi procesora. Program skacze do procedury, której adres znajduje się w komórkach pamięci `0xFFFE` (dolny bajt) i `0xFFFF` (górny bajt).

```
memory[SP + 0x100] <- (IP + 1)_high
SP <- (SP - 1) & 0xFF
memory[SP + 0x100] <- (IP + 1)_low
SP <- (SP - 1) & 0xFF
memory[SP + 0x100] <- SR
SP <- (SP - 1) & 0xFF
PC <- memory[0xFFFE] | memory[0xFFFF] << 8
```

Instrukcja BVC

Skok, gdy brak przepełnienia.

```
if (!V):
    PC <- operand
```

Instrukcja BVS

Skok, gdy wystąpiło przepełnienie.

```
if (V):
    PC <- operand
```

Instrukcja CLC

Modyfikowane flagi: C
Zerowanie flagi przeniesienia.

```
C <- 0
```

Instrukcja CLD

Modyfikowane flagi: D
Zerowanie flagi trybu dziesiętnego (BCD).

```
D <- 0
```

Instrukcja CLI

Modyfikowane flagi: I
Zerowanie bitu zakazu przerwań.

```
I <- 0
```

Instrukcja CLV

Modyfikowane flagi: V
Zerowanie flagi przepełnienia.

```
V <- 0
```

Instrukcje CMP, CPX, CPY

Modyfikowane flagi: N, Z, C
Porównanie odpowiednio: akumulatora, rejestru `X` bądź rejestru `Y` oraz komórki pamięci. Odpowiednio zostają zaktualizowane flagi `N` (czy różnica wartości rejestru i pamięci potraktowana jako liczba ze znakiem jest ujemna), `Z` (czy rejestr jest równy wartości komórki pamięci) oraz `C` (czy różnica wartości rejestru i komórki pamięci potraktowana jako wynik bez znaku mieści się w zakresie jednego bajtu).

```
N <- (signed)(reg - mem) < 0
Z <- reg == mem
C <- (unsigned)(reg - mem) <= 255
```

Instrukcja DEC

Modyfikowane flagi: N, Z
Zmniejszenie o 1 bajtu pamięci.

```
memory <- memory - 1
```

Instrukcja DEX

Modyfikowane flagi: N, Z
Zmniejszenie o 1 wartości rejestru `X`.

```
X <- X - 1
```

Instrukcja DEY

Modyfikowane flagi: N, Z
Zmniejszenie o 1 wartości rejestru `Y`.

```
Y <- Y - 1
```

Instrukcja EOR

Alternatywa rozłączna bitowa komórki pamięci oraz akumulatora. Wynik zostaje zapisany do rejestru `A`.

```
A <- memory ^ A
```

Instrukcja INC

Modyfikowane flagi: N, Z
Zwiększenie o 1 bajtu pamięci.

```
memory <- memory + 1
```

Instrukcja INX

Modyfikowane flagi: N, Z
Zwiększenie o 1 wartości rejestru `X`.

```
X <- X + 1
```

Instrukcja INY

Modyfikowane flagi: N, Z
Zwiększenie o 1 wartości rejestru `Y`.

```
Y <- Y + 1
```

Instrukcja JMP

Skok bezwarunkowy.

```
PC <- operand
```

Instrukcja JSR

Skok do procedury, czyli skok bezwarunkowy z zapisaniem adresu powrotu na stosie. Zapisywana jest wartość licznika instrukcji pomniejszona o jeden.

```
memory[SP + 0x100] <- (IP - 1)_high
SP <- (SP - 1) & 0xFF
memory[SP + 0x100] <- (IP - 1)_low
SP <- (SP - 1) & 0xFF
PC <- operand
```

Instrukcja LDA

Modyfikowane flagi: N, Z
Załadowanie bajtu pamięci do akumulatora.

```
A <- memory
```

Instrukcja LDX

Modyfikowane flagi: N, Z
Załadowanie bajtu pamięci do rejestru `X`.

```
X <- memory
```

Instrukcja LDY

Modyfikowane flagi: N, Z
Załadowanie bajtu pamięci do rejestru `Y`.

```
Y <- memory
```

Instrukcja LSR

Modyfikowane flagi: N, Z, C
Przesunięcie bitowe w prawo o jeden. Najbardziej znaczący bit jest uzupełniany zerem.

```
memory <- memory >> 1
```

Instrukcja NOP

Instrukcja pusta.

Instrukcja ORA

Modyfikowane flagi: N, Z
Alternatywa bitowa komórki pamięci z akumulatorem. Wynik jest zapisywany do rejestru `A`.

```
A <- A | memory
```

Instrukcja PHA

Umieszczenie na stosie wartości akumulatora.

```
memory[SP + 0x100] <- A
SP <- (SP - 1) & 0xFF
```

Instrukcja PHP

Umieszczenie na stosie wartości rejestru flag procesora.

```
memory[SP + 0x100] <- SR
SP <- (SP - 1) & 0xFF
```

Instrukcja PLA

Modyfikowane flagi: N, Z
Zdjęcie ze stosu bajtu do akumulatora.

```
A <- memory[SP + 0x100]
SP <- (SP + 1) & 0xFF
```

Instrukcja PLP

Zdjęcie bajtu ze stosu do rejestru flag procesora. Flaga `B` oraz nieużywany bit piąty są zawsze ustawione niezależnie od wartości na stosie.

```
SR <- memory[SP + 0x100] | B | bit5
SP <- (SP + 1) & 0xFF
```

Instrukcja ROL

Modyfikowane flagi: N, Z, C
Przesunięcie cykliczne bitu w lewo. Najmniej znaczący bit zostaje ustawiony, w przypadku gdy wcześniej wystąpiło przeniesienie (flaga `C` była ustawiona).

```
memory <- (memory << 1) | (if C then 1 else 0)
```

Instrukcja ROR

Modyfikowane flagi: N, Z, C
Przesunięcie cykliczne bitu w prawo. Najbardziej znaczący bit zostaje ustawiony, w przypadku gdy wcześniej wystąpiło przeniesienie (flaga `C` była ustawiona).

```
memory <- (memory >> 1) | (if C then 0x80 else 0)
```

Instrukcja RTI

Powrót z procedury obsługi przerwania. W przeciwieństwie do rozkazu `RTS`, ta instrukcja przywraca również flagi, które zostały odłożone na stosie podczas przerwania. Flaga `B` oraz nieużywany bit piąty są zawsze ustawione, niezależnie od stanu zapisanego na stosie.

```
SR <- memory[SP + 0x100] | B | bit5
SP <- (SP + 1) & 0xFF
PClow <- memory[SP + 0x100]
SP <- (SP + 1) & 0xFF
PChigh <- memory[SP + 0x100]
SP <- (SP + 1) & 0xFF
```

Instrukcja RTS

Powrót z procedury.

```
PClow <- memory[SP + 0x100]
SP <- (SP + 1) & 0xFF
PChigh <- memory[SP + 0x100]
SP <- (SP + 1) & 0xFF
PC <- PC + 1
```

Instrukcja SBC

Modyfikowane flagi: N, Z, C, V
Odejmowanie z pożyczką.

```
tmp <- A - memory - (if C then 0 else 1)
Z <- (tmp % 255) == 0
if D:
    if (A & 0xF) - (C ? 0 : 1) < memory & 0xF:
        tmp -= 6
    N <- (tmp & 0x80) != 0
    V <- ((A ^ memory) & 0x80) == 0 && ((A ^ tmp) & 0x80) != 0
    if tmp > 0x99:
        tmp <- tmp - 96
    C <- tmp > 0x99
else:
    N <- (tmp & 0x80) == 0x80
    V <- ((A ^ memory) & 0x80) == 0 && ((A ^ tmp) & 0x80) != 0
    C <- tmp > 255

A <- tmp % 255
```

Instrukcja SEC

Modyfikowane flagi: C
Ustawienie flagi przeniesienia.

```
C <- true
```

Instrukcja SED

Modyfikowane flagi: D
Ustawienie flagi trybu dziesiętnego (BCD).

```
D <- true
```

Instrukcja SEI

Modyfikowane flagi: I
Ustawienie bitu zakazu przerwań.

```
I <- true
```

Instrukcje STA, STX, STY

Zapisanie akumulatora, rejestru `X` bądź rejestru `Y` w komórce pamięci.

```
memory <- A lub X lub Y
```

Instrukcja TAX

Modyfikowane flagi: N, Z
Przeniesienie wartości akumulatora do rejestru `X`.

```
X <- A
Z <- X == 0
N <- X < 0
```

Instrukcja TAY

Modyfikowane flagi: N, Z
Przeniesienie wartości akumulatora do rejestru `Y`.

```
Y <- A
Z <- Y == 0
N <- Y < 0
```

Instrukcja TSX

Modyfikowane flagi: N, Z
Przeniesienie wartości rejestru stosu do rejestru `X`.

```
X <- SP
Z <- X == 0
N <- X < 0
```

Instrukcja TXA

Modyfikowane flagi: N, Z
Przeniesienie wartości rejestru `X` do akumulatora.

```
A <- X
Z <- A == 0
N <- A < 0
```

Instrukcja TXS

Przeniesienie wartości rejestru `X` do rejestru stosu.

```
SP <- X
```

Instrukcja TYA

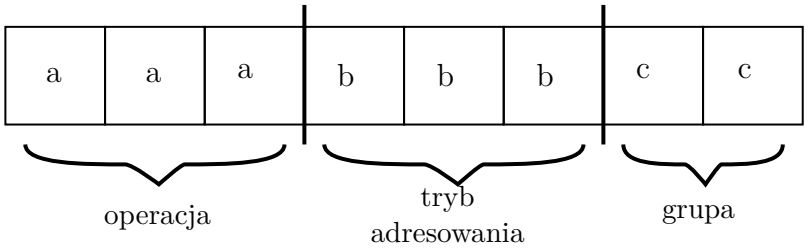
Modyfikowane flagi: N, Z
Przeniesienie wartości rejestru `Y` do akumulatora.

```
A <- Y
Z <- A == 0
N <- A < 0
```

Dodatek B

Kodowanie instrukcji

Rozkazy w języku maszynowym procesora MOS 6502 zajmują od jednego do trzech bajtów. Pierwszy z nich to zawsze kod instrukcji (ang. *opcode*), pozostałe jeden lub dwa bajty mogą zawierać operand zależny od operacji, może to być adres pamięci bądź wartość argumentu. Kod instrukcji jest zapisany w sposób widoczny na rysunku B.1. Choć nie jest to regułą, zwykle dwa najmniej ważne bity oznaczają do jakiej grupy należy rozkaz, trzy kolejny bity definiują tryb adresowania, a trzy najwyższe to kod operacji.



Rysunek B.1: Budowa kodu instrukcji

Podręcznik MOS [3] wyróżnia trzy grupy instrukcji, jednak instrukcje znajdujące się w danej grupie nie są ze sobą powiązane, podział jest przydatny w celu kodowania i dekodowania kodów instrukcji. Tabela B.1 przedstawia wszystkie możliwe kombinacje instrukcji i trybów adresowania. Dla każdej takiej pary można odczytać wartości bitów `aaa`, `bbb`, `cc`, które po złożeniu jak na rysunku B.1 dadzą kod instrukcji.

W tabeli znajduje się wiele pustych komórek (w tym cała grupa trzecia: `cc` $== 11_2$; dla tych kombinacji bitów nie istnieje oficjalna instrukcja). Jednak ze względu na sposób dekodowania rozkazów, próba wykonania niektórych takich kodów instrukcji kończy się sukcesem, choć efekty nie zawsze są przewidywalne. Przykładowo instrukcja o kodzie `0xA7` powoduje jednoczesne załadowanie wartości z pamięci do rejestrów `A` oraz `X`, czyli tak naprawdę jednoczesne wykonanie instrukcji `LDA` oraz `LDX`. Nie należy jednak polegać na takich instrukcjach, bowiem nie ma żadnej gwarancji ich poprawnego działania, co więcej czasem rozkaz może powodować niedeterministyczne efekty, na przykład operacja o kodzie `0x8B` zapisuje do akumulatora wynik koniunkcji bitowej operandu, rejestru `X` i rejestru `A`, jednak w zależności od zewnętrznych czynników przed koniunkcją następuje bitowa alternatywa rejestru `A` z pewną stałą.

Tabela B.1: Tabela wszystkich oficjalnych rozkazów procesora MOS 6502. Kolumna `cc` zawiera dwa dolne bity kodu instrukcji, kolumna `aaa` trzy najwyższe bity, a `bbb` — pozostałe trzy

bity cc	bity aaa	bity bbb 000_2	001_2	010_2	011_2	100_2	101_2	110_2	111_2
00_2	000_2	BRK impl		PHP impl		BPL rel		CLC impl	
	001_2	JSR abs	BIT zpg	PLP impl	BIT abs	BMI rel		SEC impl	
	010_2	RTI impl		PHA impl	JMP abs	BVC rel		CLI impl	
	011_2	RTS impl		PLA impl	JMP ind	BVS rel		SEI impl	
	100_2		STY zpg	DEY impl	STY abs	BCC rel	STY zpg,X	TYA impl	
	101_2	LDY #	LDY zpg	TAY impl	LDY abs	BCS rel	LDY zpg,X	CLV impl	LDY abs,X
	110_2	CPY #	CPY zpg	INY impl	CPY abs	BNE rel		CLD impl	
	111_2	CPX #	CPX zpg	INX impl	CPX abs	BEQ rel		SED impl	
01_2	000_2	ORA X,ind	ORA zpg	ORA #	ORA abs	ORA ind,Y	ORA zpg,X	ORA abs,Y	ORA abs,X
	001_2	AND X,ind	AND zpg	AND #	AND abs	AND ind,Y	AND zpg,X	AND abs,Y	AND abs,X
	010_2	EOR X,ind	EOR zpg	EOR #	EOR abs	EOR ind,Y	EOR zpg,X	EOR abs,Y	EOR abs,X
	011_2	ADC X,ind	ADC zpg	ADC #	ADC abs	ADC ind,Y	ADC zpg,X	ADC abs,Y	ADC abs,X
	100_2	STA X,ind	STA zpg		STA abs	STA ind,Y	STA zpg,X	STA abs,Y	STA abs,X
	101_2	LDA X,ind	LDA zpg	LDA #	LDA abs	LDA ind,Y	LDA zpg,X	LDA abs,Y	LDA abs,X
	110_2	CMP X,ind	CMP zpg	CMP #	CMP abs	CMP ind,Y	CMP zpg,X	CMP abs,Y	CMP abs,X
	111_2	SBC X,ind	SBC zpg	SBC #	SBC abs	SBC ind,Y	SBC zpg,X	SBC abs,Y	SBC abs,X
10_2	000_2		ASL zpg	ASL A	ASL abs		ASL zpg,X		ASL abs,X
	001_2		ROL zpg	ROL A	ROL abs		ROL zpg,X		ROL abs,X
	010_2		LSR zpg	LSR A	LSR abs		LSR zpg,X		LSR abs,X
	011_2		ROR zpg	ROR A	ROR abs		ROR zpg,X		ROR abs,X
	100_2		STX zpg	TXA impl	STX abs		STX zpg,Y	TXS impl	
	101_2	LDX #	LDX zpg	TAX impl	LDX abs		LDX zpg,Y	TSX impl	LDX abs,Y
	110_2		DEC zpg	DEX impl	DEC abs		DEC zpg,X		DEC abs,X
	111_2		INC zpg	NOP impl	INC abs		INC zpg,X		INC abs,X

Dodatek C

Opis załączników

Do pracy dołączony jest kod źródłowy tłumacza oraz dodatkowe biblioteki służące do uruchomienia testów oraz gier przeznaczonych na konsolę Nintendo Entertaiment System. Poniżej przedstawione są najważniejsze katalogi projektu:

- `Mos6502Assembler/` — narzędzie linii poleceń oraz biblioteka, która umożliwia asemblację kodu napisanego w języku asemblera dla procesora MOS 6502 do postaci pliku obiektowego,
- `Mos6502Disassembler/` — narzędzie linii poleceń oraz biblioteka, które odwrotnie przetwarza obiektowy kod na kod źródłowy w języku asemblera,
- `Mos6502Emulator/` — interpreter procesora MOS 6502, używany w celu porównania działania różnych metod emulacji,
- `MosToX86Converter/` — najważniejsza część kodu, czyli komponent odpowiedzialny za tłumaczenie skompilowanego kodu z postaci obiektowej MOS 6502 na architekturę x86-64,
- `MosToX86Converter.Test/` — testy sprawdzające poprawność oraz wydajność translacji,
- `NesConverter/` — narzędzie służące do konwersji plików ROM urządzenia Nintendo Entertainment System na program wykonywalny architektury x86-64; konwersja jest możliwa na systemach operacyjnych Linux oraz macOS,
- `NesConverter/runtime/` — środowisko uruchomieniowe przetłumaczonych gier przeznaczonych na konsolę NES.

Wymagane zależności

Tłumacz został napisany w języku C#, do uruchomienia wymagane jest środowisko .NET w wersji 6.0[1]. Do wygenerowania pliku wykonywalnego wymagany jest asembler `yasm`[2] oraz kompilator `clang`[3] (w celu skompilowania pomocniczej biblioteki). Interfejs graficzny narzędzia `NesConvert` został oparty o bibliotekę `SDL 2.0`[4].

[1] `https://dotnet.microsoft.com`
[2] `https://yasm.tortall.net`
[3] `https://clang.llvm.org`
[4] `https://www.libsdl.org`

Użycie tłumacza

Po zainstalowaniu wszystkich wymaganych zależności, testy uruchamia się poleceniem:

```
dotnet test
```

Narzędzie `NesConverter` wymaga podania dwóch argumentów linii poleceń: źródłowego pliku w formacie `iNES` oraz ścieżki do pliku wynikowego. Uruchamia się je następującym poleceniem:

```
dotnet run --project NesConverter/NesConverter.csproj [plik źródłowy]
↪  [plik docelowy]
```

Bibliografia

[1] J. Ruszczyc. *Asembler 6502*. SOETO, 1987.

[2] MOS Technology, Inc. *MCS6500 Microcomputer Family Hardware Manual*. 1976.

[3] MOS Technology, Inc. *MCS6500 Microcomputer Family Programming Manual*. 1976.

[4] V. M. del Barrio. *Study of the techniques for emulation programming*. 2001.

[5] J. Conley, E. Andros, P. Chinai, E. Lipkowitz i D. Perez. Use of a Game Over: Emulation and the Video Game Industry, A White Paper. *Northwestern Journal of Technology and Intellectual Property*, 2:1, 2004.

[6] Rockwell Automation, Inc. R650X and R651X microprocessors (CPU), 1987.

[7] Synertek, Inc. Synertek 1981–1982 Data Catalog, 1981.

[8] G. James, B. Silverman, B. Silverman. Visualizing a Classic CPU in Action: The 6502. In *ACM SIGGRAPH 2010 Talks*, SIGGRAPH '10. Association for Computing Machinery, 2010.

[9] G. James, B. Silverman, B. Silverman. Visualizing a Classic CPU in Action: The 6502 (SIGGRAPH 2010 slides). `http://www.visual6502.org/docs/6502_in_action_14_web.pdf`, 2010.

[10] L. Elko i in. Readers feedback. *Compute!*, 10(4):45–46, 1988.

[11] D. McCreary. 8080 Simulation with a 6502 / How to Speak 8080. *MICRO — The 6502 Journal*, (16):53–56, 1979.

[12] D. Fylstra. Son of Motorola (or, the $20 CPU Chip). *BYTE Magazine*, (3):56–62, 1975.

[13] M. Fayzullin. iNES — NES/Famicom emulator. `https://fms.komkon.org/iNES`, 2021.

[14] K. Dormann. 6502 65C02 Functional Tests. `https://github.com/Klaus2m5/6502_65C02_functional_tests`, 2013–2020.

[15] K. Horton. Nestest. `https://github.com/christopherpow/nes-test-roms/blob/master/other/nestest.txt`, 2009.

[16] Apple Inc. The 68LC040 Emulator. `https://developer.apple.com/library/archive/documentation/mac/PPCSoftware/PPCSoftware-13.html`, 1996.

[17] A. Frumusanu. The 2020 Mac Mini Unleashed: Putting Apple Silicon M1 To The Test. `https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested/6`, 2020.

[18] M. Roser i H. Ritchie. A logarithmic graph showing the timeline of how transistor counts in microchips are almost doubling every two years from 1970 to 2020; Moore's Law. https://ourworldindata.org/uploads/2020/11/Transistor-Count-over-time.png, 2020.

[19] M. Steil. Perfect 6502. https://github.com/mist64/perfect6502, 2020.

[20] G. Ghettini. MOS 6502 Emulator. https://github.com/gianlucag/mos6502, 2015.

[21] V. Heikkila. 2048. https://bitbucket.org/tsone/neskit/src/master/examples/2048/, 2014.

[22] InstLatX64. Opóźnienia instrukcji procesora Intel(R) Core(TM) i9-10900K CPU @ 3,70 GHz wygenerowane przez program AIDA64. http://users.atw.hu/instlatx64/GenuineIntel/GenuineIntel00A0671_RocketLake_InstLatX64.txt, 2020.

[23] First Famicom/NES emulator. https://www.zophar.net/forums/index.php?threads/first-famicom-nes-emulator.10169, 2009.