Emulator procesora Zilog Z80

Tomasz Kowalczyk

3 lutego 2019

Spis treści

Rozdział 1

Wstęp

Celem pracy jest wykonanie emulatora procesora Zilog Z80. Aplikacja umożliwia wczytanie programu w postaci kodu maszynowego, asemblację (proces tłumaczenia języka asemblera na kod maszynowy) i wykonanie. Dostępne są dwa tryby wykonania: ciągły i krokowy. W obu przypadkach emulator obrazuje stan rejestrów oraz umożliwia podgląd i zmianę zawartości pamięci programu. Aplikacja została zaimplementowana w języku Java.

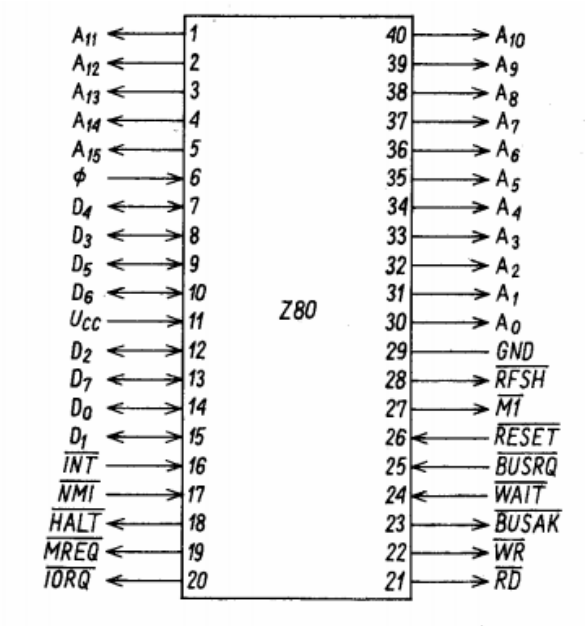
Procesor Zilog z80 był bardzo popularny na rynku mikroprocesorów[3]. Ukazał się w roku 1976, i szybko zdominował rynek 8-bitowych procesorów.

Jednym z powodów jego sukcesu, była prostota w sprzęganiu go z innymi urządzeniami, szczególnie z kontrolerami pamięci. Inną jego zaletą była lista rozkazów zgodna z popularnym w tamtym czasie procesorem, mianowicie Intelem 8086, co umożliwiało uruchamianie programów napisanych z pierwotnym przeznaczeniem dla Intela 8080 na Zilogu Z80[3].

Urządzenie to mimo zalet, miało również jedną dużą wadę. Jego wewnętrzna budowa była złożona jak na tamte czasy, wyjścia nie były ułożone w logiczny sposób (widoczne na rysunku 1.1), a lista rozkazów składała się z 158 pozycji, w tym 78 z nich zgodnych z Intel 8080A[4].

Samą aplikacje wykonano w języku Java 8 i biblioteki graficznej Java FX. Interfejs użytkownika został podzielony na 3 części:

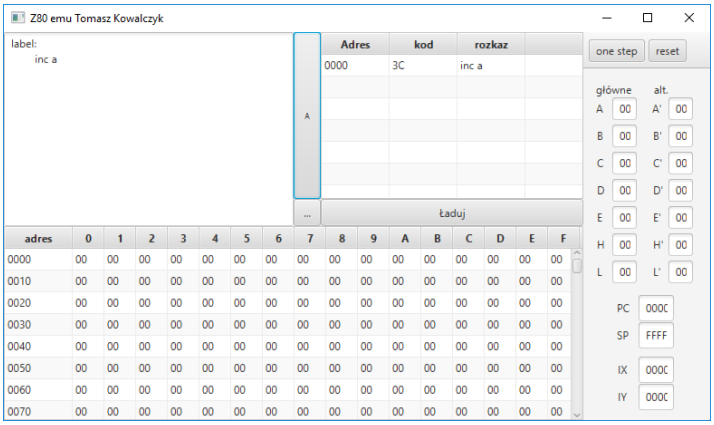
- widok kodu programu napisany w języku asembler, wraz z wynikowym kodem maszynowym
- widok pamięci w formie tabeli. Adres komórki uzyskuje się przez dodanie do siebie wartości z nazwy kolumny z wartością w nazwie wiersza. Edycje wykonuje się przez dwukrotne kliknięcie w komórkę tabeli, wpisaniu nowej wartości i zatwierdzeniu klawiszem Enter.



Rysunek 1.1: Wyprowadzenia mikroprocesora Z80 [3]

- widok stanu wewnętrznych rejestrów procesora, wraz z przyciskami debugującymi.

W aplikacji każda wyświetlona wartość podana jest w systemie heksadecymalnym, i także w takiej notacji można wprowadzać wartości (oprócz pola do edycji kodu asemblera, gdzie można używać innych notacji).



Rysunek 1.2: Widok główny emulatora

Rozdział 2

Zagadnienie emulacji

Emulator w kontekście informatyki, oznacza program który jest przystosowany do uruchomienia na specyficznym urządzeniu lub/i systemie, i pozwala na uruchomienie programów napisanych z przeznaczeniem dla innego urządzenia/systemu[5].

Inną ciekawą definicję emulatora podał Victor Moya del Barrio "Emulacja w informatyce oznacza emulowanie zachowania urządzenia lub oprogramowania za pomocą innego oprogramowania lub urządzenia" [1].

Emulacje CPU można przeprowadzić na 3 sposoby:[2]

- emulacja przez interpretowanie
- emulacja przez statyczną re-kompilacje
- emulacja przez dynamiczną re-kompilacje

Każda z tych metod wymaga oddzielnego omówienia.

2.1 Emulacja przez interpretowanie

Interpreter to najprostszy rodzaj emulatora. Odczytuje w pętli kod programu z wirtualnej pamięci. Odczytany bajt (lub bajty, rozkaz procesora może być wielobajtowy) zawiera informacje o rodzaju operacji jaką CPU powinno wykonać. Interpreter ma za zadanie odkodować informacje o operacji, a następnie ją wykonać. Między kolejnymi rozkazami powinien on zmienić wirtualne parametry (np inkrementacja licznika rozkazów), sprawdzić czy nie zostało wywołane przerwanie, obsłużyć urządzenia wejścia/wyjścia, liczniki, kartę graficzną, lub wykonać inne operacje zależne od emulowanego urządzenia. Przykładowa struktura interpretera została przedstawiona w kodzie 2.1.

```
int PC = 0;
while(warunekStopu()) {

  Rozkaz rozkaz = dekodujRozkaz(pobierzRozkaz());

  switch(rozkaz) {
    case ROZKAZ_1:
      rozkaz_1();
    case ROZKAZ_2:
      rozkaz_2();
    ...
  }

  obslugaPrzerwan();
  obslugaIO();
  inkrementacjaLicznikow();

  PC++;
}
```

Kod 2.1: Przykładowa struktura interpretera procesora

Emulacja przez interpretowanie jest najwolniejszą formą emulacji, ale także najłatwiejszą w debugowaniu. Pozwala na prześledzenie wykonania operacji, i podgląd wewnętrznych stanów urządzenia. Z tego powodu jest najczęściej wybierana w debuggerach procesorów, mikro-kontrolerów dla programistów.

2.2 Statyczna re-kompilacja

Statyczna re-kompilacja (ang. "Static binary translation") to proces konwertowania kodu maszynowego na inny kod maszynowy przeznaczony dla docelowej architektury. Plik wykonywalny tłumaczony jest raz, za jednym podejściem przez cały plik. Problemem tego rozwiązania są instrukcje skoków pośrednich czyli takich gdzie adres skoku przechowywany jest w rejestrze lub pamięci, i może on być uzyskany tylko podczas wykonywania programu. W takim przypadku niemożliwym jest przetłumaczenie wszystkich instrukcji pliku wykonywalnego[6].

2.3 Dynamiczna re-kompilacja

Dynamiczna re-kompilacja (ang. "Dynamic binary translator"), w odróżnieniu od translacji dynamicznej, tłumaczy kod blokami, podczas jego wykonywania. Re-kompilacja występuje "na żądanie" co jest wolniejsze od statycznej re-kompilacji, ale rozwiązuje problem związany z statycznym tłumaczeniem kodu wykonywanego za instrukcjami skoków pośrednich.

Raz przetłumaczony fragment kodu jest przechowywany w pamięci, na wypadek jego ponownego użycia, co pozwala zoptymalizować ten sposób emulacji[6].

Dynamicznej rekompilacji używa w dużym stopniu maszyna wirtualna javy. Wczesne wersje JVM (Java Virtual Machin) używały do swojego działania interpreterów, co okazało się mało wydajne. Dobrym sposobem na optymalizację maszyny wirtualnej okazało się dynamiczne tłumaczenie kodu maszynowego[7].

2.4 Różnica między symulacją a emulacją (i może virtualizacją???)

coś o wirtualizacji http://jpc.sourceforge.net/oldsite/Emulation.html

Zagadnienie emulacji często mylone jest z symulacją. Nie są to jednak jednoznaczne pojęcia.

W książe "Study of the techniques for emulation programming" Victor Moya del Barrio podaje taką to definicje emulatora "An emulator tries to duplicate the behaviour of a full computer using software programs in a different computer."[1].

Rozdział 3

Przegląd istniejących rozwiązań

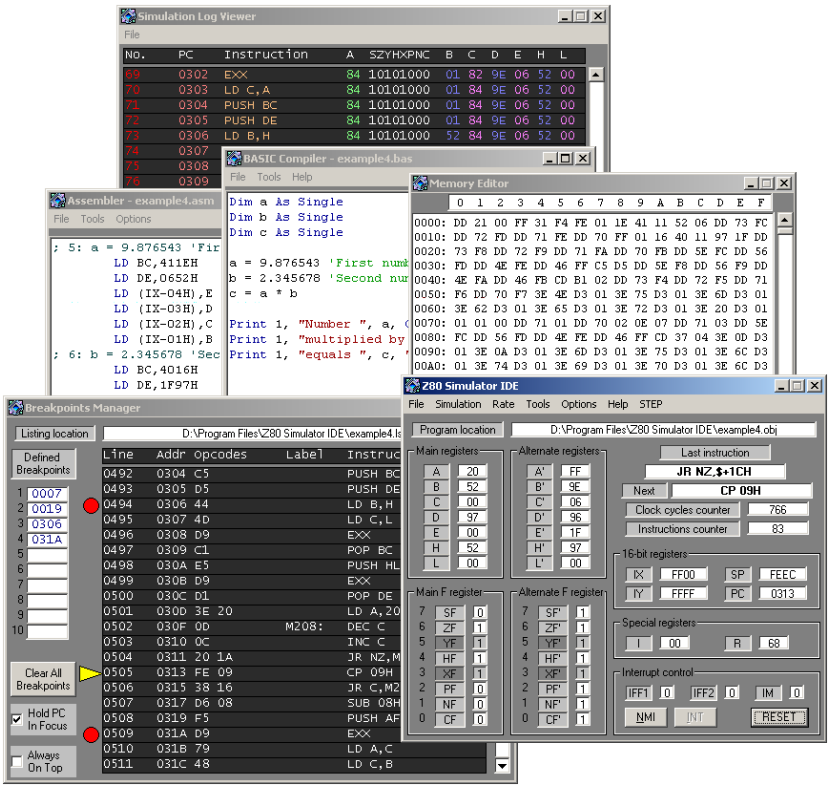
Zilog Z80 stał popularnym obiektem emulacji dzięki swojej popularności. Aplikacje te, pisano zarówno przez duże firmy, jak i hobbystów. Na jednej z usług hostingowych o nazwie „Github" która jest przeznaczona dla projektów programistycznych, można znaleźć około 200 repozytoriów z projektami emulującymi Z80, lub emulującymi urządzenia używające tego procesora[9].

Poniżej zaprezentowane są najciekawsze pozycje tych programów, które posiadają graficzny interfejs użytkownika, i pozwalają na wgląd w wewnętrzne stany procesora (czyli najbardziej przypominające zakresem swoich funkcji opisywany projekt). Przedstawiane są zarówno komercyjne rozwiązania, jak i te pisane przez amatorów.

3.1 Z80 SIMULATOR IDE

Dostępny pod adresem http://www.oshonsoft.com/z80.html płatny symulator posiadający najbardziej rozbudowany interfejs z wszystkich wymienionych pozycji. Pozwala on na prezentowanie wewnętrznych stanów procesora, manipulację przerwaniami, edytor pamięci umożliwiający działający również podczas symulacji, podgląd i manipulacja portami wejścia/wyjścia. Posiada również funkcje typowe dla debuggerów, możliwość wstrzymania działania programu w określonym miejscu, tryb pracy krokowej, interaktywny edytor i kompilator kodu asemblera[8]. Część funkcji została zaprezentowana na rysunku 3.1

Jedną z wad emulatora jest interfejs, który nie jest intuicyjny. Dla przykładu, w żadnym miejscu nie znajdziemy informacji, o tym w jakim formacie powinny być wprowadzane wartości liczbowe. Brak w programie systemu pomocy i opisów, co może odstraszyć początkującego użytkownika. Dodatkowo jest to rozwiązanie płatne i przeznaczone tylko dla platformy



Rysunek 3.1: Z80 SIMULATOR IDE [8]

MS Windows. Jest to narzędzie głównie dla specjalistów.

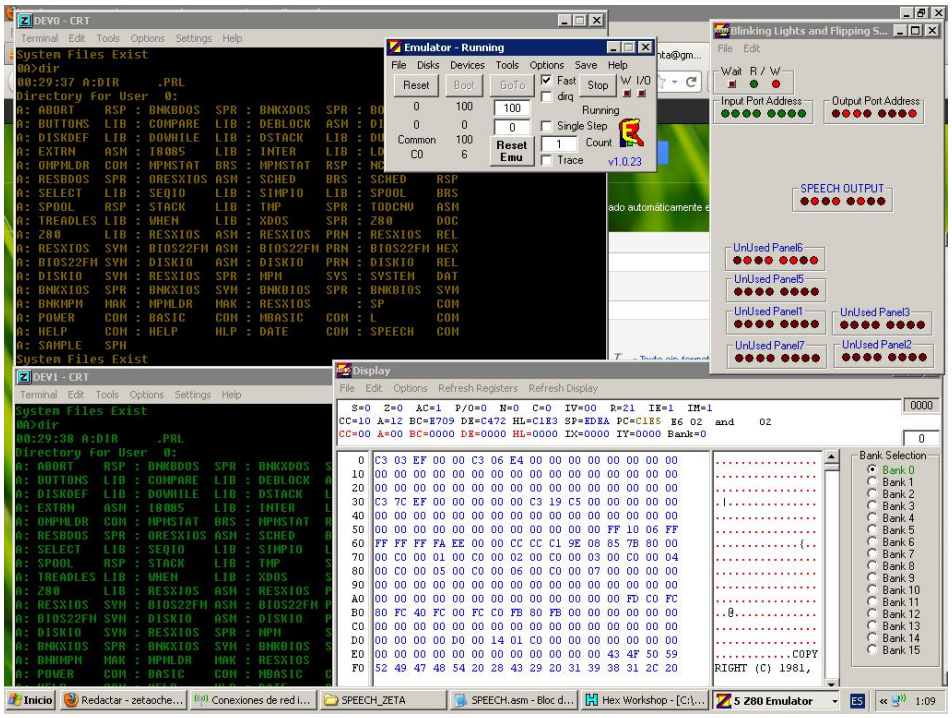
3.2 ZEMU - Z80 Emulator Joe Moore

ZEMU to emulator zaprojektowany głównie, do uruchamiania systemu CPM. Była to seria systemów operacyjnych oferowana przez firmę Digital Research Inc i latach 1970-1980[10].

Program skierowany jest do hobbystów. Oprócz standardowych możliwości takich jak podgląd, edycja pamięci, rejestrów i flag, ma możliwość emulacji stacji dyskietek, portu COM, portu szeregowego, monitora CRT, drukarki.

Na rysunku 3.2 przedstawiono interfejs aplikacji. Tak jak w przypadku Z80 SIMULATOR IDE nie jest intuicyjny, brak mu systemu pomocy, elementy interfejsu nie są opisane w wystarczającym stopniu. Osoby nie mające doświadczenia z urządzeniami wejścia/wyjścia w Zilogu Z80 będą miały problem z obsługą nawet podstawowych funkcji.

Inną wadą aplikacji jest możliwość jej uruchomienia tylko w systemie Windows.



Rysunek 3.2: ZEMU [11]

3.3 ZIM - The Z80 Machine Simulator

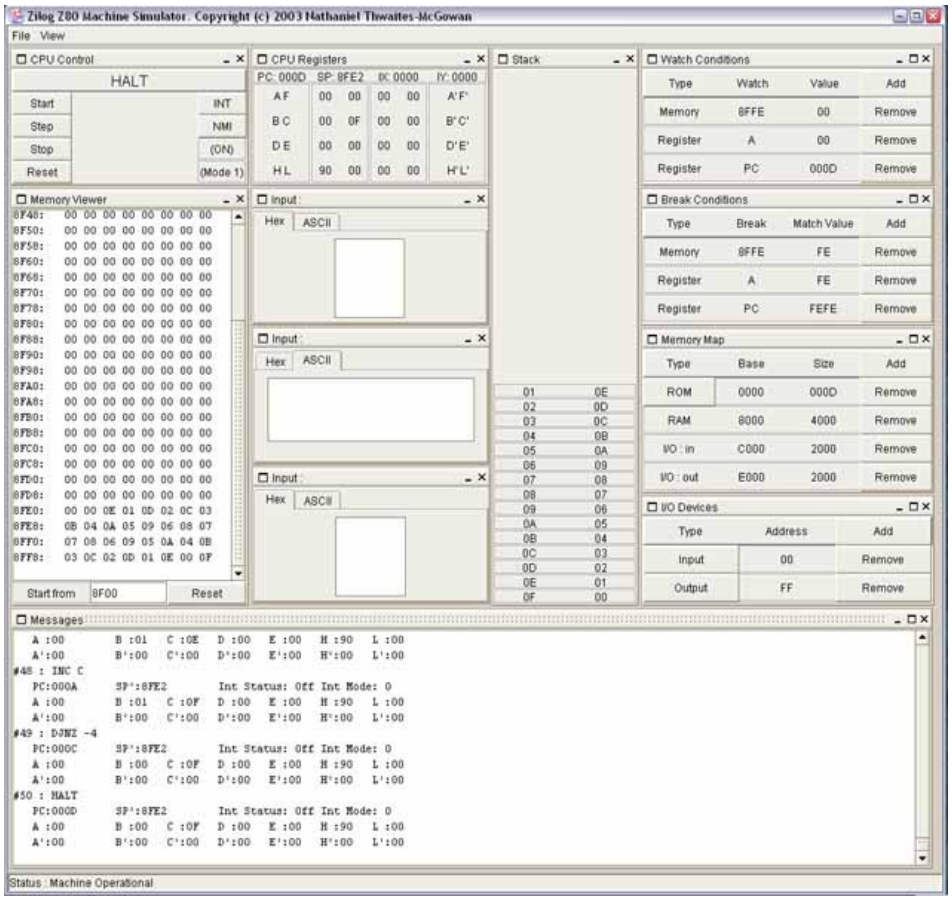
ZIM został napisany za pomocą technologi Java Web-Start. Pozwala to na uruchomienie aplikacji bezpośrednio na stronie internetowej, wraz z dostępem do lokalnych zasobów systemu, np plików[12]. Aplikacja według autora przeznaczona jest głównie dla studentów uczących się języka asemblera dla procesora Z80[13].

Aplikacja pozwala na podgląd wszystkich wewnętrznych parametrów cpu, emuluje proste urządzenia wejścia/wyjścia, umożliwia edycję pamięci, debugowanie programu, symulację przerwań. Zaletą programu jest jego wieloplatformowość dzięki zastosowaniu języka Java.

Brakuje w niej natomiast edytora asemblera, systemu pomocy, interfejs ma wiele błędów i nie jest intuicyjny. Na rysunku 3.3 przedstawiono zrzut ekranu działającej aplikacji.

3.4 Podsumowanie istniejących rozwiązań

Problemem prawie wszystkich emulatorów i symulatorów jest interfejs. Nie tyczy się to konkretnie omawianego procesora, ale ogółu tego typu aplikacji. Są one przeznaczone dla osób znających architekturę komputerową, oraz budowę i działanie konkretnego emulowanego urządzenia, z tego powodu programiści nie przywiązują do intuicyjności i systemów



Rysunek 3.3: ZIM - The Z80 Machine Simulator [14]

pomocy odpowiedniej wagi. Osoby nie posiadające wymaganej wiedzy, albo znające jedynie podstawy nie są w stanie sprawnie obsługiwać programu.

Emulatory rzadko są wielplatformowe. Dotyczy się to szczególnie aplikacji emulujących przez re-kompilacje. Aby ją wykonać wymagana jest znajomość architektur wyjściowej i docelowej. Rozwiązaniem tego problemu mogą być interpretery napisane w języku Java, tak jak "ZIM - The Z80 Machine Simulator". Rozwiązanie to wykorzystuje dużo zasobów procesora, kod emulowanej architektury jest najpierw interpretowany przez interpreter napisany w języku Java, a następnie ponownie emulowany już przez dynamiczną re-kompilacje w maszynie wirtualnej. Typowe rozwiązania napisane w języku kompilowanym pod konkretny procesor działają szybciej, kosztem wieloplatformowość. Optymalizacja w sprzęcie typu Zilog Z80 nie jest kluczową kwestią, współczesne komputery są na tyle szybkie, aby uruchomić emulator maszyny z lat 70 napisanej w Javie.

Kolejną kwestią o której warto wspomnieć jest czytelność kodu. Kod istniejących rozwiązań nie należy do przejrzystych, najczęściej cały kod emulatora zawiera się w jednym pliku z

kilkuset liniami. Osoba pragnąca wgłębić się w sam proces emulacji w takim przypadku ma utrudnione zadanie.

Podsumowując, aktualnie brakuje wieloplatformowego emulatora lub symulatora Ziloga Z80, z czytelnym, poprawnie działającym interfejsem, przejrzystym dobrze komentowanym kodem źródłowym, i system pomocy.

Rozdział 4

Projekt aplikacji

4.1 Podział aplikacji

Aplikację podzielono na 3 mniejsze moduły, xbit, z80emu-core oraz z80emu-gui z czego każdy z nich jest osobnym mniejszym projektem, który używa narzędzia „Maven" do zautomatyzowania procesu budowy oprogramowania. Ich pliki jar są przetrzymywane w platformie mymavenrepo.com jako prywatne repozytorium, zabezpieczone hasłem przy pomocy funkcji protokołu http "basic auth". Każdy z projektów może zostać wysłany do zdalnego serwera za pomocą komendy „mvn deploy:deploy". Komenda ta przeprowadza testy jednostkowe, kompiluje kod javy do kodu bajtowego JVM, dodaje zewnętrzne bliblioteki jar i umieszcza plik wykonywalny na serwerze.

Aby umożliwić integracje z serwisem mymavenrepo.com pliki pom.xml zawierają wpisy zaprezentowane w fragmencie kodu 4.2. Ponieważ repozytoria maven są ustawione jako prywatne, należy odnaleźć w systemie plik /.m2/settings.xml podać w nim dane pozwalające na autoryzacje za pomocą "http basic auth". Przykład konfiguracji przedstawiono w kodzie 4.3

Przetrzymywanie plików wykonywalnych w zewnętrznym serwisie ułatwia zarządzanie wszystkimi trzema projektami, pozwala na ich wersjonowanie, i łatwiejsze budowanie aplikacji.

4.2 XBit

Java jest językiem programowania wysokiego poziomu, kompilowanym do kodu bajtowego. Z tego powodu nie jest on zazwyczaj stosowany w emulacji, gdyż kod emulatora musi być uruchamiany w maszynie wirtualnej, co nie należy do optymalnych rozwiązań.

```
<repositories>
    <repository>
        <id>myMavenRepoRead</id>
        <url>https://mymavenrepo.com/repo/PmCL80jeU82fLVGyuUt4/</url>
    </repository>
</repositories>

<distributionManagement>
    <repository>
        <id>myMavenRepo.write</id>
        <url>${myMavenRepoWriteUrl}</url>
    </repository>
</distributionManagement>
```

Kod 4.2: fragment pliku pom.xml umożliwający integracje z serwisem myMavenRepo

```
<server>
        <id>myMavenRepo.write</id>
        <username>usename</username>
        <password>password</password>
</server>
```

Kod 4.3: fragment pliku settings.xml przechowującego dane utoryzacyjne do serwera myMavenRepo

Innym poważnym problemem Javy jest brak typów danych przechowujących tylko wartości dodatnie. James Gosling, jeden z twórców Javy tak argumentuje ich brak: „Dla mnie jako projektant języków programowania, do których tak naprawdę ostatnimi czasy się nie zaliczam, coś prostego oznaczało skończenie na założeniu że losowy deweloper będzie w stanie zapamiętać specyfikacje. Ta definicja mówi, że na przykład Java i wiele innych języków nie są proste, i w rzeczywistości wiele języków kończy z funkcjonalnościami których do końca nikt nie rozumie. Spytaj jakiegoś programistę języka C o dodatnie typy liczbowe, i odkryjesz że prawie żaden programista C faktycznie nie rozumie co dzieje się z typami bez znaku, czym jest arytmetyka liczb całkowitych. Takie rzeczy sprawiły że C jest językiem skomplikowanym. Myślę że w tej kwestii Java jest prosta"[15].

Problem ten rozwiązano tworząc własną bibliotekę o roboczej nazwie XBit. Przechowuje

ona liczby 8 i 16 bitowe, które mogą być interpretowane zarówno jako wartości całkowite jak i liczby w kodzie uzupełnień do dwóch. Biblioteka potrafi zwrócić konkretne bity, stworzyć reprezentacje liczby z pojedynczych bitów i typów prymitywnych, obudowuje operacje arytmetyczne z uwzględnieniem przepełnienia i przeniesienia. Kod projektu z jej użyciem staje się czytelniejszy i łatwiejszy do ewentualnej refaktoryzacji.

Bibliotekę zaprojektowano w taki sposób aby była możliwie jak najbardziej uniwersalna, i można ją było użyć nie tylko podczas emulacji Zilog-a Z80. ale także innych procesorów.

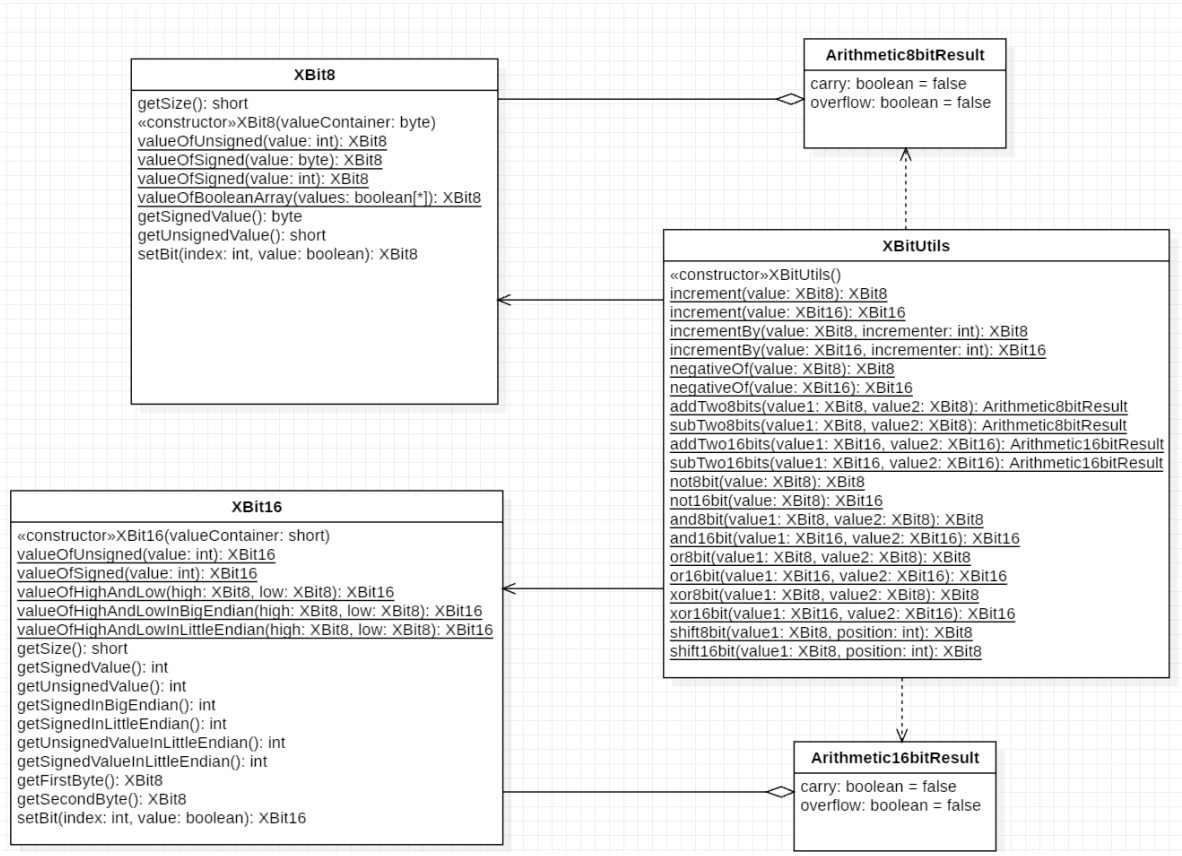
4.2.1 Możliwości

Za cel obrano następujące funkcje:

- odwzorowanie liczb 8 i 16 bitowych
- możliwość stworzenia reprezentacji liczb 8 i 16 bitowych z typów prymitywnych
- interpretacja liczb w naturalnym kodzie binarnym lub dopełnień do dwóch
- operacje na pojedynczych bitach(możliwość zmiany, odczytu bitu o danym indeksie)
- opcja odczytania grupy bitów (odczytanie kilku bitów podając indeks pierwszego i ostatniego bitu)
- interpretacja liczb 16 bitowych w formacie big endian lub little endian
- operacje arytmetyczne (dodawanie, odejmowanie)
- operacje bitowe na liczbach (negacja, alternatywa, koniunkcja, przesunięcia bitowe)
- uwzględnienie przy operacjach arytmetycznych przepełnienia oraz przeniesienia

4.2.2 Założenia projektowe xBit

Przed przystąpieniem do implementacji rozwiązania, zaprojektowano publiczny interfejs biblioteki w języku uml, zaprezentowany na grafice 4.1. Ustalono także założenia projektowe które zaprezentowano poniżej.



Rysunek 4.1: projekt uml biblioteki xbit

Klasy XBit8 i XBit16

Klasy XBit8 oraz XBit16 to reprezentacje liczb 8 i 16 bitowych. Postanowiono, że nie będą one w stanie zmienić swojej wewnętrznej wartości, swego stanu (z angielskiego nazwano by je "immutable"), tak jak obiektowe reprezentacje typów prostych w javie (Short, Long, Integer itp). Przez podjęcie tej decyzji projektowej, metody które powinny zmienić stan obiektu, klonują istniejący obiekt a następnie modyfikują odpowiednio jego stan. Przykładem takiej metody jest setBit(index: int, value: boolean) klasy Xbit8 oraz Xbit16. Funkcja ta nie zmienia bitu obiektu na rzecz którego została wykonana, a jedynie zwraca kopie obiektu, z zmodyfikowanym odpowiednim bitem.

Zalety zastosowania obiektów niezmiennych:

- Prosta implementacja oraz łatwe debugowanie kodu.
- Garbage Collector jest zoptymalizowany pod względem pracy z tego typu obiektami.
- Łatwość zapisywania obiektów do pliku lub pamięci podręcznej (z ang. cache).

- Bezpieczne używanie obiektów niezmiennych w programach wielowątkowych, co umożliwiałoby emulacje potokowości CPU. Jeden wątek w takim przypadku mógłby być odpowiedzialny za jeden stopień potoku, np osobny wątek pobierał by instrukcje z pamięci, inny dekodował instrukcję, kolejny wykonywał i tak dalej.
- Możliwość użycia obiektu jako klucz, np w HashMap.

Wadą zastosowania obiektów niezmiennych jest optymalizacja, i zwiększone użycie pamięci, co mimo wszystko nie powinno być przeszkodą dla współczesnych komputerów. Ze względu na przewagę zalet w stosunku do jednej wady postanowiono użyć obiektów niezmiennych w bibliotece.

Klasa XBitUtils, Arithmetic8bitResult, Arithmetic16bitResult

Klasa XBitUtils jest odpowiedzialna za wszystkie operacje arytmetyczne oraz bitowe. Większość z jej metod zwraca obiekty klas XBit8 lub XBit16. Wyjątkami są metody wykonujące dodawanie lub odejmowanie, które oprócz zwrócenia wyniku operacji powinny informować o wystąpieniu przeniesienia lub przepełnienia. Z tego zaprojektowano klasy Arithmetic8bitResult oraz Arithmetic16bitResult które zawierają następujące pola:

- obiekt klasy XBit8 lub XBit16 będący rezultatem operacji
- dwie zmienne typu boolean informujące o wystąpieniu przeniesienia i przepełnienia

4.3 z80emu-core

Z80emu-core to moduł mający za zadanie wykonywać emulację oraz udostępniać zestaw metod umożliwiający manipulacje tym procesem. Za cel obrano stworzenie takiego interfejsu, który pozwalał by na użycie Z80emu-core w innych projektach, które emulują urządzenia zbudowane między innymi z Ziloga Z80. Jako przykład można podać emulator przenośnej konsoli Game Boy firmy Nintendo z 1989 roku która jako główny procesor używała Ziloga Z80.

Za cel obrałem implementacje następujących funkcji:

- możliwość wykonania wszystkich 158 instrukcji procesora
- zestaw metod umożliwiających zmianę stanów rejestrów
- emulacja zewnętrznej pamięci

- możliwość podłączenia urządzeń wejścia/wyjścia
- wywołanie przerwania maskowanego oraz niemaskowanego (Z80 posiada dwa rodzaje przerwań)

4.3.1 Publiczny interfejs modułu z80emu-core

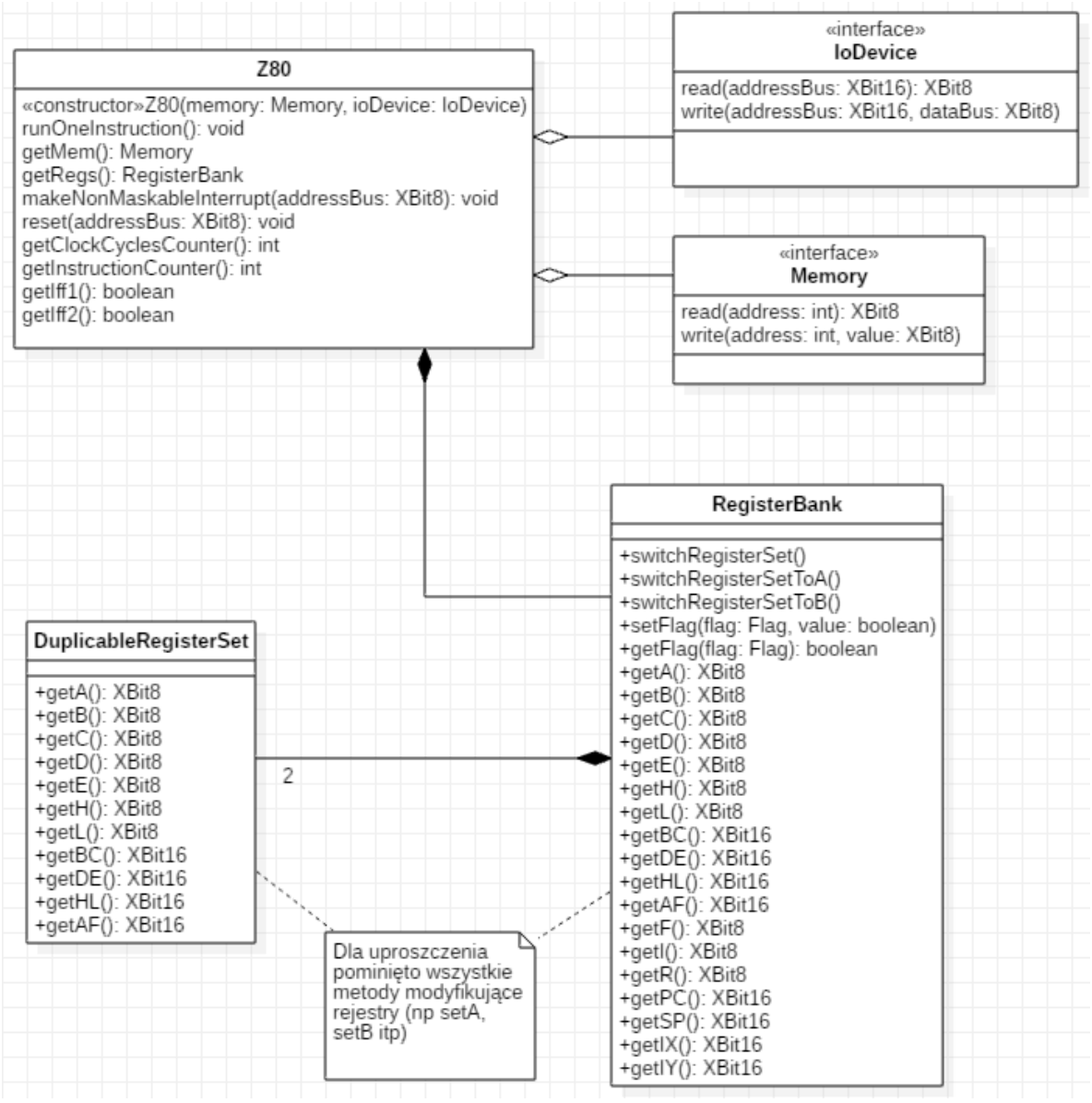
Publiczny zestaw metod służący manipulacją samym procesem emulacji jak i urządzeniem pokazano na grafice 4.2. Cała mechanika działania modułu została ukryta w trzech klasach: Z80, RegistersBank i DuplicableRegisterSet.

Klasa Z80 zawiera zestaw funkcji manipulującymi samą emulacją, oraz wartościami liczników i mniejszych rejestrów procesora. Niżej opisuje najważniejsze z nich:

- Z80(memory: Memory, ioDevice: IoDevice) - konstruktor, przyjmujący dwa parametry wymagane do poprawnego działania. Parametr „memory" oraz "ioDevice to obiekty implementujące interfejsy o tych samych nazwach. Reprezentują one kolejno moduł pamięci oraz urządzenia wejścia wyjścia podłączone do procesora. Użytkownik używający modułu Z80emu-core ma powinien samemu zaimplementować ich działanie, z zależności od otoczenia w jakim chce emulować urządzenie.
- runOneInstruction() - metoda wykonująca jedną instrukcję procesora, za jej pomocą wykonane zostanie pobieranie, dekodowanie, wykonywanie rozkazu, ale również obsługa przerwań, inkrementacja liczników.
- makeInterupt(addressBus: XBit8), makeNonMaskableInterupt(addressBus: XBit8) - metody powinny zostać wywołane między kolejnymi wywołaniami runOneInstruction(). Ich zadaniem jest zgłaszanie przerwań. Parametr addressBus to wartość jaka została by ustalona na magistrali danych podczas przerwania, gdyby zostało ono wykonane w prawdziwym urządzeniu.

Wartym przypomnienia jest, że Zilog Z80 jest procesorem posiadającym dwa zestawy rejestrów ogólnego przeznaczenia (do nich należąc rejestry A,B,C,D,E,H,L,F oraz ich 16bitowe odpowiedniki). Takie rozwiązanie jest optymalne, w przypadku gdy procesor często wykonuje obsługę przerwań. W klasycznym podejściu, podczas przerwania zestawi rejestrów ogólnego przeznaczenia zapisany zostałby na stosie, co jest czasochłonną operacją. Projektanci Ziloga Z80 postanowili stworzyć drugi alternatywny zestaw rejestrów ogólnego przeznaczenia, który zostaje przełączony jako główny podczas przerwania. W takim przypadku nie jest wymagane odłożenie wartości na stos.

W projekcie reprezentacją zestawu rejestrów ogólnego przeznaczenia jest klasa DuplicableRegisterSet. Jej dwie instancje (jedna jako główny zestaw rejestrów, druga alternatywny) przechowuje klasa RegisterBank, która posiada metody switchRegisterSet(), switchRegisterSetToA() i switchRegisterSetToB() pozwalające na przełączanie głównego zestawu rejestrów. Metody typu getA(), setA(value: XBit8) to aliasy wykonujące te operacje na aktualnie aktywnym zestawie.



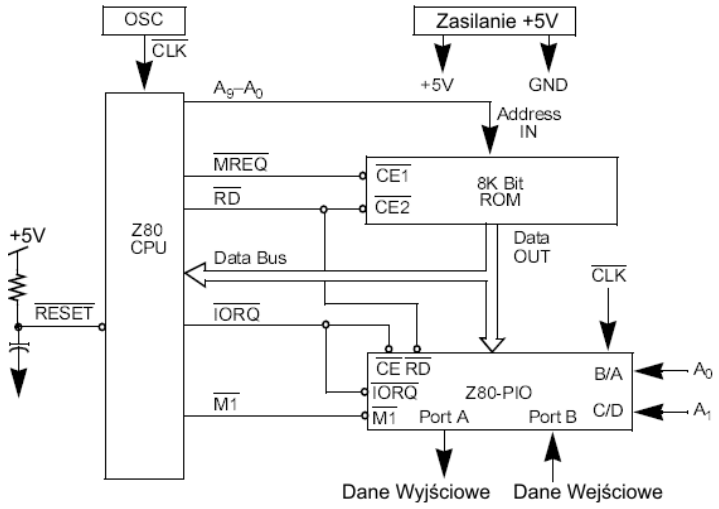
Rysunek 4.2: publiczny interfejs projektu z80emu-core zaprezentowany za pomocą diagramu klas uml

4.3.2 Elastyczność w łączeniu z innymi projektami

Postanowiono nie uruchamiać procesu emulacji w pętli głównej tak jak w klasycznym podejściu pokazanym w kodzie 2.1. Zamiast tego udostępniono jedną metodę runOneInstruction() która wykonuje jeden rozkaz procesora.

Metody manipulujące procesem emulacji (np wywołanie przerwania, edycja rejestrów) powinny zostać wywoływane między kolejnymi wywołaniami runOneInstruction(). Główna pętla emulacji powinna zostać zaimplementowania w module nadrzędnym, używającym z80emu-core. Pozwala to na większą elastyczność modułu w łączeniu go z innymi projektami.

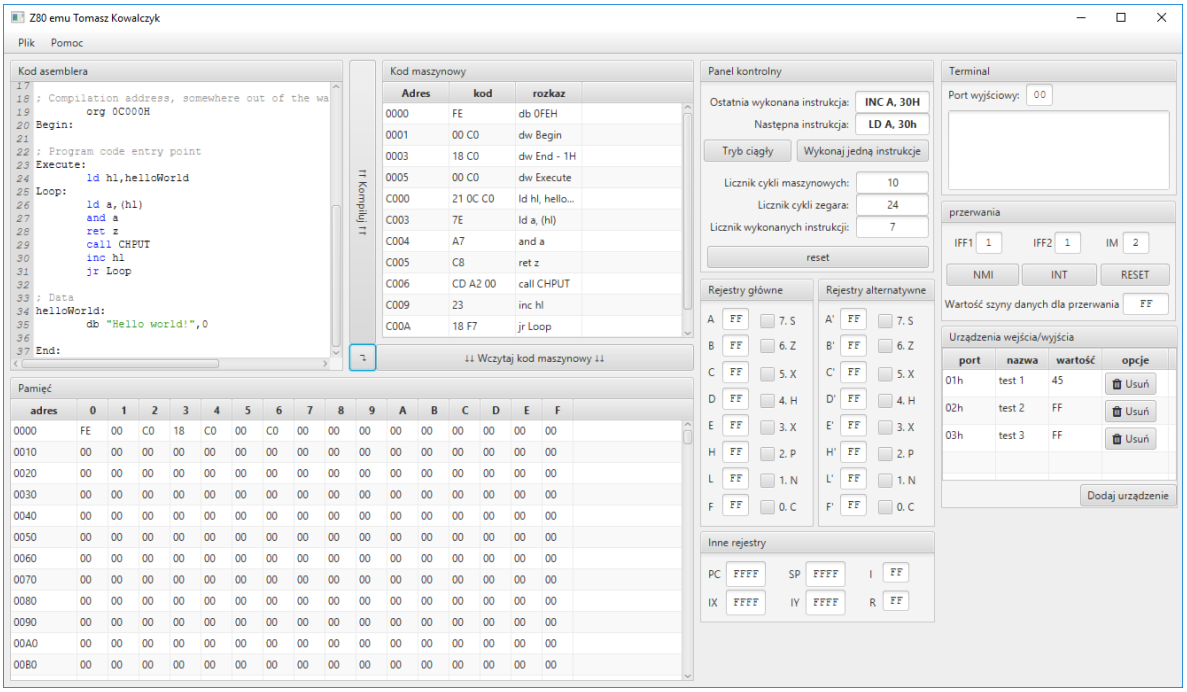
Inną ważną decyzją projektową pozwalającą na zwiększenie elastyczności projektu, było zaprojektowanie interfejsów IoDevice oraz Memory, i decyzja o zaimplementowaniu jedynie prostych reprezentacji klas implementujących te interfejsy. Aby dobrze przedstawić powód tego rozwiązania na rysunku 4.3 przedstawiono minimalny system komputerowy oparty na procesorze z80. Widać na nim że pamięć programu (na rysunku jest to 8kb ROM) oraz podłączone urządzenie wejścia/wyjścia (na rysunku z80-PIO który jest programowalnym układem wejścia/wyjścia) są osobnymi urządzeniami, i mogą one być różne w zależności od systemu komputerowego. Interfejsy IoDevice oraz Memory pozwalają na implementację zachowania takich urządzeń, jakie są wymagane dla docelowego projektu.



Rysunek 4.3: Minimalny System Komputerowy Z80 [16]

4.4 z80emu-gui

Z80emu-gui to moduł zawierający interfejs użytkownika. Został napisany z pomocą biblioteki JavaFX. Projekt interfejsu uwzględnił stworzenie makiety w języku fxml zaprezentowanej na grafice 4.4. Większość interfejsów emulatorów posiada interfejs złożony z wielu okien, które można dowolnie zamykać i otwierać, a każde z nich zawiera osobny moduł. Dla przykładu, w Z80 simulator ide, edytor pamięci, asemblera lub manipulacja urządzeniami wejścia wyjścia zawarta jest w osobnych oknach. W projekcie z80emu-gui postanowiono stworzyć interfejs zawierający wszystkie funkcje w jednym oknie.



Rysunek 4.4: Makieta interfejsu użytkownika

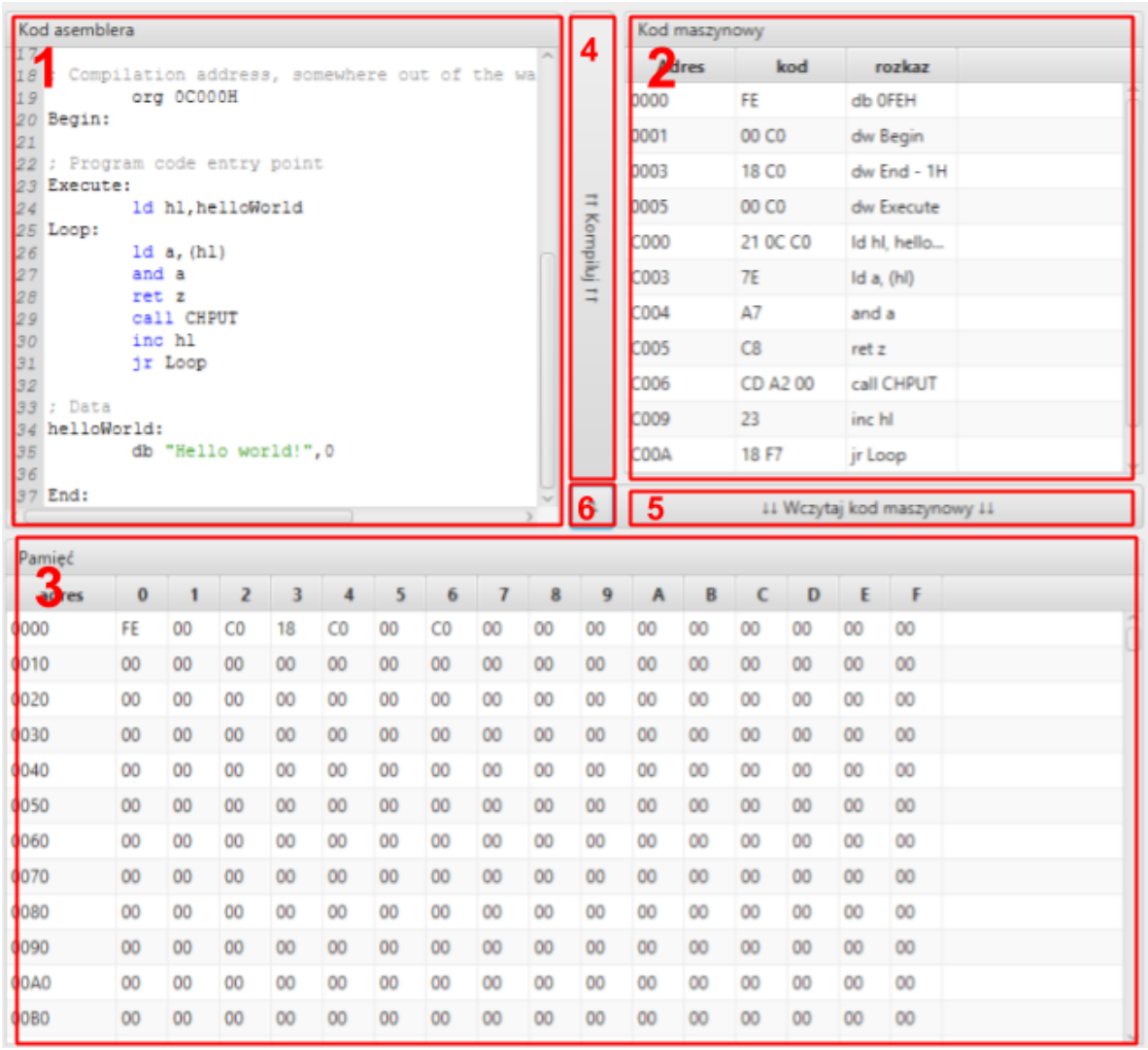
Elementy interfejsu podzielono na dwa mniejsze fragmenty w celu czytelniejszego wyjaśnienia. Część pierwsza zaprezentowana na grafice 4.5 zawiera elementy związane z manipulacją pamięcią oraz kodem programu. Opis poszczególnych elementów umieszczonych na schemacie:

1. Edytor kodu asemblera, z prostym kolorowaniem składni.
2. Widok kodu maszynowego w formie tabeli, zawierającej formacje o adresie rozkazu w pamięci, kodzie maszynowym oraz czytelnej dla człowieka nazwie instrukcji.
3. Tabela reprezentująca moduł pamięci podłączony do procesora.

4. Przycisk kompilujący kod asemblera (p. 1) do kodu maszynowego (p. 2).

5. Przycisk wczytujący kod maszynowy programu(p. 2), do pamięci procesora (p. 3)

6. Przycisk wykonujący kompilacje kodu asemblera (p. 1) a następnie wczytanie go do pamięci (p. 3).

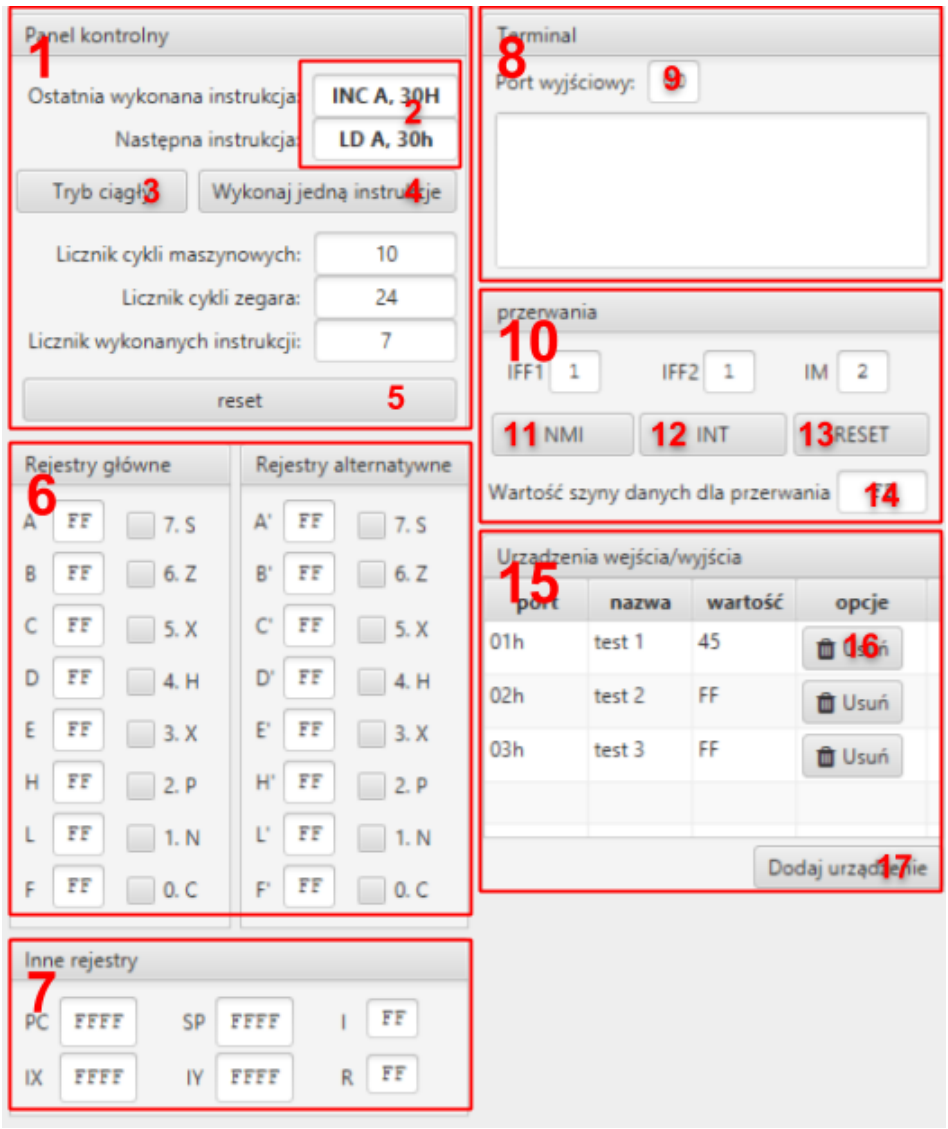


Rysunek 4.5: Makieta interfejsu użytkownika z oznaczeniami. Część 1

Drugą część interfejsu zaprezentowano na grafice 4.6. Poniżej opis poszczególnych elementów:

1. Główny panel kontrolujący proces emulacji, przedstawiający między innymi liczbę cykli maszynowych i zegara wykonywanego programu.

2. Pola prezentujące nazwę ostatniej wykonanej i następnej w kolejce instrukcji procesora.
3. Przycisk uruchamiający emulację ciągłą (ponownie jego wybranie zatrzymuje emulacje).
4. Przycisk wykonujący pojedynczą, kolejną instrukcję procesora.
5. Przycisk resetujący urządzenie.
6. Widok rejestrów głównych i alternatywnych włącznie z flagami.
7. Widok rejestrów I, R, PC, SP, IX, IY.
8. Terminal wyjściowy, emulujący na stałe przypisane urządzenie.
9. Pole tekstowe z możliwością wpisania portu terminalu wyjściowego, pod jakim będzie przyjmował dane.
10. Panel grupujący elementy interfejsu odpowiedzialne za przerwania.
11. Przycisk wywołujący NMI (z ang. Non-Maskable Interrupt), sygnał przerwania niemaskowanego.
12. Przycisk wywołujący INT (z ang. Interrupt), sygnał maskowanego przerwania.
13. Przycisk wykonujący sygnał resetu.
14. Pole tekstowe zawierające 8bitową wartość, jaką przyjmie szyna danych podczas przerwania.
15. Tabela z listą podłączonych urządzeń wejścia/wyjścia.
16. Przycisk pozwalający na usunięcie urządzenia wejścia/wyjścia.
17. Przycisk dodający nowe urządzenie wejścia/wyjścia.



Rysunek 4.6: Makieta interfejsu użytkownika z oznaczeniami. Część 2

Rozdział 5

Implementacja

W tym rozdziale przedstawiono opis implementacji projektu,

5.1 XBit

XBit to projekt mający za zadanie ułatwić operacje bitowe w javie, która interpretuje liczby całkowite w kodzie dopełnień do dwóch, i nie jest możliwa interpretacja ich w naturalnym kodzie binarnym (w skrócie NKB). Z tego powodu powstał projekt XBit implementujący taką opcję. Potrafi on odczytać liczbę n-bitową, i zwrócić jej wartość jako zmienna integer, interpretując ją jako bity w U2 lub NKB.

Wewnątrz obiektów reprezentujących liczby binarne, wartość przechowywana jest w zmiennej typu integer, którą nazwano "valueContainer". Może ona więc przyjmować teoretycznie wartości od -2147483648 do 2147483647. W praktyce liczba ta nigdy nie jest ujemna, może przyjąć wartości od 0 do $2^n - 1$ gdzie n oznacza liczbę bitów.

Dla przykładu, valueContainer obiektu Xbit8, który w zamierzeniu ma przechowywać wartości 8-bitowe, będzie mógł zawierać wartość od 0 do $2^8 - 1 = 255$.

Jeśli XBit8 będzie przechowywał liczbę binarną 11111111 (czyli odczytując w NKB to decymalne 255, a w u2 decymalnie -1). Jego wartość zmiennej valueContainer będzie wynosić binarnie 00000000 00000000 00000000 11111111. (czyli odczytując zarówno w NKB jak i w U2 będzie to 255). Niestety nadmiarowość bitów jest w tym rozwiązaniu wymagana, ponieważ wartość w kodzie u2 której najbardziej znaczącym bitem wynosi 1, jest ujemna.

Podsumowując, cała sztuczka XBit wykorzystuje fakt takiego samego zapisu liczb dodatnich w U2 i NKB, jeśli tylko będą one zapisane na większej ilości bitów niż wymagana.

5.1.1 Implementacja klasy Xbit

XBit to klasa abstrakcyjna zawierająca wspólne elementy dla klas XBit8 i XBit16 oraz, ewentualnych przyszłych klas. Poniżej opiszę warte uwagi metody.

boolean getBit(int index)

```
1  public boolean getBit(int index) {
2          if(index <0 || index > getSize()-1) {
3                  throw new NumberFormatException();
4          }
5          return ((valueContainer >> index) & 1) == 1;
6  }
```

Kod 5.4: Metoda boolean getBit(int index)

W kodzie 5.4 zaprezentowano metodę getBit zwracającą bit o danym indeksie przekazanym w parametrze. W linii nr 2 sprawdzana jest poprawność podanego indeksu. Jeśli parametr jest niepoprawny, w linii nr 3 wyrzucany jest wyjątek NumberFormatException. W linii nr 5 wykonujemy przesunięcie bitowe pola valueContainer o podany index. Ponieważ wynikiem przesunięcia bitowego jest liczba o typie integer, wykonujemy koniunkcje bitową z maską o wartości 1. Na końcu przekształcamy wynik operacji z typu integer na boolean i zwracamy.

TSelf setBit(int index, boolean value)

```
1  public TSelf setBit(int index, boolean value) {
2          int mask = 1 << index;
3          int newValue;
4          if(value) {
5                  newValue = this.getUnsignedValue() | mask;
6          } else {
7                  newValue = (this.getUnsignedValue() & ~mask);
8          }
9          return createNewOfUnsigned(newValue);
10 }
```

Kod 5.5: Metoda TSelf setBit(int index, boolean value)

Kod 5.5 prezentuje metodę modyfikujący bit o danym indeksie, o daną wartość, a następnie zwracający nową zmodyfikowany obiekt. Zwracany typ TSelf to typ generyczny reprezentujący docelową liczbę n-bitową. W linii nr 2 tworzymy maskę bitową, wykonując przesunięcie bitowe na liczbie 1 o wartość parametru "index", czyli maska będzie posiadała tylko jeden bit ustawiony na prawdę logiczną, i będzie to bit o indeksie z parametru. W liniach 5 i 7 wykonuje operacje bitowe mające na celu zmianę bitu na 1 lub 0 w zależności od parametru "value".

- Dla "value" równego 1 wykonuje się linia 5. Najpierw pobieramy obecną wartość obiektu i wykonujemy alternatywę bitową na niej, i na wcześniej zbudowanej masce. Wynik przypisujemy do zmiennej "newValue" będącej buforem.
- Dla "value" równego 10 wykonuje się linia 7. Nową wartość uzyskujemy przez koniunkcje bitową aktualnej wartości oraz negacji maski.

int getValueOfBits(int startIndexBit, int stopIndexBit)

```
1 public int getValueOfBits(int startIndexBit, int stopIndexBit) {
2         if(startIndexBit<stopIndexBit) {
3                 throw new NumberFormatException();
4         }
5
6         int buff = valueContainer >>> stopIndexBit;
7         return buff & (~(Integer.MAX_VALUE<<(startIndexBit-stopIndexBit+1)));
8 }
```

Kod 5.6: Metoda int getValueOfBits(int startIndexBit, int stopIndexBit)

Kod 5.6 przedstawia metodę przyjmującą dwa parametry, które są indeksami dwóch bitów, a zwraca wartość od indeksu pierwszego argumentu, do drugiego. Dla przykładu, jeśli wartość obiektu to binarnie 00001110, argument startIndexBit jest równy 4, a stopIndexBit jest równy 1, to metoda zwróci binarnie wartość 0111.

W liniach 1 i dwa sprawdzamy poprawność argumentów i ewentualnie wyrzucamy wyjątek. Linia 6 tworzy bufor, wykonując przesunięcie bitowe w prawo na wartości obiektu, o wartość drugiego, końcowego indeksu podanego w parametrze metody.

Linia nr 7 wykonuje koniunkcję bitową na buforze, i wyniku operacji przesunięcia bitowego w lewo wartości wypełnionej binarnej jedynkami, o wyniki operacji indeksPoczątku-indeksKońca+1.

Inne metody

Klasa XBit posiada inne metody, zbyt proste w swojej budowie aby były warte głębszego opisywania. Zostaną one jedynie wymienione poniżej w celu lepszego zobrazowania wszystkich funkcjonalności.

- public abstract short getSize() - abstrakcyjna klasa zwracająca ilość bitów jakie przechowuje docelowa liczba
- public abstract int getMinSignedValue() - zwraca minimalną liczbę ze znakiem jaka może być przechowywana w obiekcie
- public abstract int getMaxSignedValue() - zwraca maksymalną liczbę ze znakiem jaka może być przechowywana w obiekcie
- public int getMinUnsignedValue() - zwraca minimalną liczbę bez znaku jaka może być przechowywana w obiekcie (czyli zawsze przyjmuje wartość 0)
- public abstract int getMaxUnsignedValue() - zwraca maksymalną liczbę bez znaku jaka może być przechowywana w obiekcie
- public boolean isNegative() - zwraca wartość bitu o największym indeksie, który w kodowaniu u2 decyduje czy liczba jest mniejsza od zera
- public int getSignedValue() - zwraca wartość w kodowaniu U2
- public int getUnsignedValue() - zwraca wartość w kodowaniu NKB

5.1.2 Implementacja klasy Xbit8

Klasa XBit8 dziedziczy funkcjonalności po abstrakcyjnej klasie XBit, która została zbudowana w taki sposób aby klasy po niej dziedziczące były jak najmniejsze. Metody które należą do XBit8 i nie zostały odziedziczone nie są na tyle skomplikowane aby warto było je głębiej opisywać, z tego powodu zostaną wymienione poniżej i krótko opisane bez szczegółów implementacji.

- public static XBit8 valueOfUnsigned(int value) - tworzy nowy obiekt o wartości bez znaku
- public static XBit8 valueOfSigned(int value) - tworzy nowy obiekt o wartości ze znakiem

- public static XBit8 valueOfBooleanArray(boolean[] values) - tworzy nowy obiekt na podstawie tablicy elementów o typie boolean.

5.1.3 Implementacja klasy Xbit16

XBit16 to klasa reprezentująca liczbę 16 bitową. Dziedziczy ona po XBit dzięki temu nie musi implementować najbardziej podstawowych funkcji.

public static XBit16 valueOfHighAndLowInBigEndian(XBit8 high, XBit8 low)

```
1 public static XBit16 valueOfHighAndLowInBigEndian(XBit8 high, XBit8 low) {
2         ByteBuffer bb = ByteBuffer.allocate(2);
3         bb.order(ByteOrder.BIG_ENDIAN);
4         bb.put((byte)high.getSignedValue());
5         bb.put((byte)low.getSignedValue());
6         return new XBit16(bb.getShort());
7 }
```

Kod 5.7: Metoda XBit16 valueOfHighAndLowInBigEndian(XBit8 high, XBit8 low)

W kodzie 5.7 pokazano implementację metody "valueOfHighAndLowInBigEndian" tworzącą liczbę 16bitową w formacie zapisu big endian (kolejność bajtów zgodna z "ludzkim" zapisem, najbardziej znaczący bajt umieszczony jest jako pierwszy). Jako parametry przyjmowane są dwie liczby 8-bitowe o nazwach "high" i "low". Na początku tworzony jest obiekt klasy ByteBuffer należącej do standardowej biblioteki java, a następnie ustawiana jest kolejność w jakiej będą dodawane kolejne bajty (linia nr 3). Następne linie prezentują dodanie do bufora kolejno większego (linia nr 4) i młodszego bajtu (linia nr 5). Na koniec wykonano metode getShort() zwracająca docelową liczbę i na jej podstawie tworzymy reprezentacje klasy XBit16.

public static XBit16 valueOfHighAndLowInLittleEndian(XBit8 high, XBit8 low)

```
1 public static XBit16 valueOfHighAndLowInLittleEndian(XBit8 high, XBit8 low) {
2         return valueOfHighAndLowInBigEndian(low, high);
3 }
```

Kod 5.8: Metoda XBit16 valueOfHighAndLowInLittleEndian(XBit8 high, XBit8 low)

W kodzie 5.8 zaprezentowano metodę tworzącą liczbę 16-bitową w formacie zapisu little endian, czyli najbardziej znaczący bajt umieszczony jest jako ostatni (odwrotnie do "ludzkiego" zapisu gdzie najbardziej znacząca cyfra jest pierwsza). Implementacja metody jest dosyć prosta, wykonuje ona metodę valueOfHighAndLowInBigEndian() z zamienioną kolejnością parametrów.

inne metody

Poniżej wymieniono metody które są zbyt proste w implementacji aby warto było je opisywać pojedynczo.

- public static XBit16 valueOfUnsigned(int value) - tworzy nowy obiekt o wartości bez znaku
- public static XBit16 valueOfSigned(int value) - tworzy nowy obiekt o wartości z znakiem

5.1.4 Implementacja klasy XbitUtils

XbitUtils to klasa implementująca operacje arytmetyczne i bitowe na obiektach XBit8 oraz XBit16. Posiada ona dwie klasy wewnętrzne, Arithmetic8bitResult oraz Arithmetic16bitResult reprezentujące wynik operacji arytmetycznych, które oprócz samego wyniku powinny informować o wystąpieniu przeniesienia i przepełnienia. Klasy te zaprezentowano w kodach 5.9 oraz 5.10

```
1 public static class Arithmetic8bitResult {
2         public XBit8 result;
3         boolean carry = false;
4         boolean overflow = false;
5 }
```

Kod 5.9: Klasa Arithmetic8bitResult

public static XBit8 incrementBy(XBit8 value, int incrementer)

Kod 5.11 prezentuje metodę inkrementującą liczbę 8 bitową o daną wartość. Aby wykonać operację, metoda konwertuje parametr "value" na liczbę typu integer (linia nr 2) i wykonuje inkrementacje za pomocą standardowej operacji dodawania w języku java (linia nr

```
1  public static class Arithmetic16bitResult {
2          public XBit16 result;
3          boolean carry = false;
4          boolean overflow = false;
5  }
```

Kod 5.10: Klasa Arithmetic16bitResult

```
1  public static XBit8 incrementBy(XBit8 value, int incrementer) {
2          int unsignedValue = value.getUnsignedValue();
3          unsignedValue += incrementer;
4          unsignedValue = unsignedValue & XBit8.MAX_UNSIGNED_VALUE;
5          return XBit8.valueOfUnsigned(unsignedValue);
6  }
```

Kod 5.11: Metoda XBit8 incrementBy(XBit8 value, int incrementer)

3). Następnie w linii nr 4 wykonuje operacje iloczynu bitowego na uzyskanej nowej wartości, oraz masce bitowej reprezentującej największą możliwą wartość jaką może przechowywać liczba 8-bitowa (celem tej operacji jest nie dopuszczenie do sytuacji, w której wynik inkrementacji nie mieści się na ośmiu bitach). Na koniec tworzona jest i zwracana instancja klasy XBit8.

Zasada działania metody "XBit16 incrementBy(XBit16 value, int incrementer)" jest podobna, dlatego zostanie pominięta.

5.1.5 public static XBit8 negativeOf(XBit8 value)

```
1  public static XBit8 negativeOf(XBit8 value) {
2          int buf = (~value.getUnsignedValue()) & XBit8.MAX_UNSIGNED_VALUE;
3          return XBitUtils.incrementBy(
4                  XBit8.valueOfUnsigned(buf),
5                  1
6          );
7  }
```

Kod 5.12: Metoda XBit8 negativeOf(XBit8 value)

Funkcja tworząca liczbę negatywną do podanej została zaprezentowana w kodzie 5.12. Operacja polega na wykonaniu iloczynu bitowego na negacji bitowej danej wartości, oraz

liczbie 255. Następnie wykonany wynik zostaje zainkrementowany.

Zasada działania metody "XBit16 negativeOf(XBit16 value)" jest identyczna. Jedyną różnicą jest operacja iloczynu bitowego którą wykonujemy nie na liczbie 255 a 65535.

5.1.6 public static Arithmetic8bitResult addTwo8bits(XBit8 value1, XBit8 value2)

```
1  public static Arithmetic8bitResult addTwo8bits(XBit8 value1, XBit8 value2) {
2          Arithmetic8bitResult ret = new Arithmetic8bitResult();
3          int result = value1.getUnsignedValue() + value2.getUnsignedValue();
4          if(result>XBit8.MAX_UNSIGNED_VALUE) {
5                  ret.carry = true;
6          }
7          ret.result = XBit8.valueOfUnsigned(result & XBit8.MAX_UNSIGNED_VALUE);
8
9          ret.overflow =((value1.getBit(7) && value2.getBit(7) && !ret.result.getBit(7)) ||
10         (!value1.getBit(7) && !value2.getBit(7) && ret.result.getBit(7)));
11
12         return ret;
13 }
```

Kod 5.13: Metoda Arithmetic8bitResult addTwo8bits(XBit8 value1, XBit8 value2)

Kod 5.13 prezentuje metodę realizującą dodawanie dwóch liczb 8-bitowych. Wynikiem jej działania jest obiekt klasy Arithmetic8bitResult przechowujący wynik operacji, i flagi przepełnienia lub przeniesienia.

W linii nr 3, metoda wykonuje samą operacje dodawania. Linia nr 4 sprawdza czy nastąpiło przepełnienie, a linia nr 5 ustawia flagę. Linia nr 7 wykonuje "obcięcie" bitów w przypadku, gdy wynik nie będzie w stanie zmieścić się w liczbie 8-bitowej.

Linia nr 9 i 10 ustawia flagę przepełnienia według trzech zasad [17]:

1. Jeśli suma dwóch liczb dodatnich daje wynik ujemny, suma została przepełniona.
2. Jeśli suma dwóch liczb ujemnych daje wynik dodatni, suma została przepełniona.
3. W każdym innym przypadku przepełnienie nie wystąpiło.

Przyjmujemy że najbardziej znaczący bit jest bitem znaku.

Prezentacja funkcji realizującej dodawanie dwóch liczb 16-bitowych została pominięta, ponieważ odbywa się na tej samej zasadzie.

5.1.7 public static Arithmetic8bitResult subTwo8bits(XBit8 value1, XBit8 value2)

```
1 public static Arithmetic8bitResult subTwo8bits(XBit8 value1, XBit8 value2) {
2         return addTwo8bits(
3                 value1,
4                 negativeOf(value2)
5         );
6 }
```

Kod 5.14: Metoda Arithmetic8bitResult subTwo8bits(XBit8 value1, XBit8 value2)

W kodzie nr 5.14 zaprezentowano metodę realizującą odejmowanie jednej liczby 8-bitowej, od drugiej. Metoda wykorzystuje regułę $a - b = a + (-b)$ mówiącą, że odejmowanie dwóch liczb można zastąpić dodawaniem, negując drugi składnik odejmowania. Metoda realizująca odejmowanie dwóch liczb 16-bitowych działa w analogiczny sposób.

5.1.8 public static XBit8 not8bit(XBit8 value)

```
1 public static XBit8 not8bit(XBit8 value) {
2         return XBit8.valueOfUnsigned(
3                 (~value.getUnsignedValue()) & XBit8.MAX_UNSIGNED_VALUE
4         );
5 }
```

Kod 5.15: Metoda XBit8 not8bit(XBit8 value)

Funkcja not8bit(XBit8 value) zaprezentowana w kodzie 5.15 wykonuje negację bitową na liczbie 8bitowej. Operacja wykonywana jest za pomocą wbudowanej w język java opcji negacji bitowej (znak "~"). Dodatkową wykonywaną operacją jest iloczyn bitowy z liczbą 255, mająca za zadanie obcięcie niepotrzebnych bitów. Wersja metody 16-bitowa jest analogiczna.

5.1.9 public static XBit8 and8bit(XBit8 value1, XBit8 value2)

Kod 5.16 prezentuje metodę wykonującą operację sumy logicznej dwóch liczb 8-bitowych (linia nr 3). Linia nr 4 obcina wynik operacji tak, aby mieścił się w ośmiu bitach. Wersja metody dla liczb 16-bitowych działa analogicznie.

```
public static XBit8 and8bit(XBit8 value1, XBit8 value2) {
        return XBit8.valueOfUnsigned(
                (value1.getUnsignedValue() & value2.getUnsignedValue())
                & XBit8.MAX_UNSIGNED_VALUE
        );
}
```

Kod 5.16: Metoda XBit8 and8bit(XBit8 value1, XBit8 value2)

5.1.10 metody wykonujące sumę bitową i różnice symetryczną

Implementacja metod:

- public static XBit8 or8bit(XBit8 value1, XBit8 value2)
- public static XBit16 or16bit(XBit16 value1, XBit16 value2)
- public static XBit8 xor8bit(XBit8 value1, XBit8 value2)
- public static XBit16 xor16bit(XBit16 value1, XBit16 value2)
- public static XBit8 shift8bit(XBit8 value1, int position)
- public static XBit8 shift16bit(XBit8 value1, int position)

wygląda analogicznie do metody and8bit(XBit8 value1, XBit8 value2). Operacje bitowe wykonywane są za pomocą mechanizmów wbudowanych w język java, a nadmiar bitów zostaje obcięty.

Rozdział 6

Testy

6.1 ????

Bardzo ważną kwestią w projekcie było dokładne pokrycie kodu aplikacji w testach jednostkowych. Emulator mikro-kontrolera to specyficzna aplikacja. Z pozoru mało znaczący błąd może sprawić że emulator stanie się bezużyteczny.

Dla przykładu, jeśli dla 3 bajtowego rozkazu procesora zwiększymy rejestr PC o 2 zamiast o 3, to nie wykona się następna instrukcja przewidziana przez programistę. Dalsza praca emulatora stanie się nieprzewidywalna, a następna instrukcja całkowicie "wykolei" nasz program który zacznie wykonywać losowe instrukcje.

Dlatego poprawne wykonanie każdego rozkazu jest tak ważne w moim projekcie. Aby uchronić się przed tego typu prostymi błędami każdy emulowany rozkaz posiada swój test/testy jednostkowe napisane przy pomocy biblioteki Junit.

Przykładowy test dla rozkazu LD A, I. Rozkaz ten ładuje zawartość rejestru A z I:

Opisany przykład ukazuje że prosta z pozoru operacja jak pobranie wartości jednego rejestru i przeniesienie go do innego, wymaga objętościowych testów. Oprócz testowania czy poprawna wartość znajduje się w rejestrze docelowym, musimy sprawdzić także czy flagi CPU zostały ustawione na poprawnych wartościach, czy ilość przewidywanych cykli zegara została poprawnie zwiększona, czy rejestr PC został zainkrementowany.

6.2 Test-driven development

TDD to metoda pisania oprogramowania. Zakłada ona że test jednostkowy dla danej funkcjonalności powstaje jako pierwszy. Dopiero po napisaniu testu implementujemy kod programu,

a następnie testujemy za pomocą już napisanych testów. Za pomocą TDD była pisana cała aplikacja

```
public class LoadAFromITest {
    private Z80 z80;

    @Before
    public void setUp() {
        z80 = new Z80();
    }

    private void prepareZ80(XBit8 regI) throws MemoryException {
        z80.getMemory().write(0, XBit8.valueOfUnsigned(0xED));
        z80.getMemory().write(1, XBit8.valueOfUnsigned(0x57)); //ld A,I

        z80.getRegs().setF(XBit8.valueOfUnsigned(0xFF));

        z80.setIff2(true);

        z80.getRegs().setI(regI);
    }

    @Test
    public void execute() throws Exception {
        prepareZ80(XBit8.valueOfSigned(0x44));
        z80.runOneInstruction();

        Assert.assertEquals(0x44, z80.getRegs().getA().getSignedValue());

        Assert.assertEquals(false, z80.getRegs().getFlag(Flag.S));
        Assert.assertEquals(false, z80.getRegs().getFlag(Flag.Z));
        Assert.assertEquals(false, z80.getRegs().getFlag(Flag.H));
        Assert.assertEquals(true, z80.getRegs().getFlag(Flag.PV));
        Assert.assertEquals(false, z80.getRegs().getFlag(Flag.N));

        Assert.assertEquals(2, z80.getRegisterBank().getPc().getUnsignedValue());
        Assert.assertEquals(9, z80.getClockCyclesCounter());
        Assert.assertEquals(1, z80.getInstructionCounter());
    }
```



```
    @Test
    public void testFlags1() throws Exception {
```

Rozdział 7

Uwagi i wnioski

Z wymienionych celów, nie zrealizowałem jedynie emulacji wewnętrznych magistrali procesora. Nie posiadając dokumentacji technicznych opisujących wewnętrzną budowę mikroprocesora, jedyną opcją było by poddanie urządzenia inżynierii wstecznej, co już nie jest tematem tej pracy.

Bibliografia

[1] Victor Moya del Barrio *Study of the techniques for emulation programming*. 2001

[2] http://fms.komkon.org/EMUL8/HOWTO.html

[3] Mikroprocesor Z80 Jerzy Karczmarczuk

[4] Oficjalny manual

[5] https://www.atarihq.com/danb/files/emu_vol1.txt How do I write an emulator? Daniel Boris, 1999

[6] https://www.researchgate.net/publication/239665973_The_university_of_queensland_binary_translator

[7] https://www.ibm.com/developerworks/library/j-jtp12214/index.html

[8] http://www.oshonsoft.com/z80.html

[9] https://github.com/search?q=emulator+z80&type=Repositories

[10] http://www.retrotechnology.com/dri/howto_cpm.html

[11] https://www.retrobrewcomputers.org/n8vem-gg-archive/html-2012/Jul/msg00238.html

[12] http://www.natmac.net/zim/

[13] http://www.natmac.net/zim/manual/index.html

[14] http://www.natmac.net/zim/GUI6.jpg

[15] http://www.gotw.ca/publications/c_family_interview.htm

[16] https://eduinf.waw.pl/

[17] http://sandbox.mc.edu/ bennet/cs110/tc/orules.html