POLITECHNIKA ŚWIĘTOKRZYSKA
Wydział Elektrotechniki, Automatyki i Informatyki

TOMASZ KOWALCZYK
Numer albumu: 083562

# EMULATOR PROCESORA ZILOG Z80

Praca dyplomowa inżynierska
na kierunku Informatyka

Opiekun pracy dyplomowej:
dr inż. Arkadiusz Chrobot
Zakład Informatyki

Kielce, 2019

**POLITECHNIKA ŚWIĘTOKRZYSKA**
**WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI I INFORMATYKI**

**Zatwierdzam:** PRODZIEKAN ds. Kształcenia i Spraw Studenckich na Studiach Stacjonarnych Wydziału Elektrotechniki, Automatyki i Informatyki dr inż. Andrzej Stobiecki

**Rok akademicki: 2018/19**

Temat nr. TD/1597/2018/S

Dnia: **27 marca 2018**

# ZADANIE NA PRACĘ DYPLOMOWĄ

**Studiów pierwszego stopnia na kierunku**
**INFORMATYKA**

Wydano dla studenta: Tomasz Kowalczyk

**I.** Temat pracy:

## Emulator procesora Zilog Z80

**II.** Plan pracy:

1. Wstęp.....
2. Zagadnienie emulacji.....
3. Przegląd istniejących rozwiązań.....
4. Projekt aplikacji.....
5. Implementacja.....
6. Testy
7. Uwagi i wnioski

**III.** Cel pracy:

Celem pracy jest wykonanie emulatora procesora Zilog Z80. Aplikacja ta powinna umożliwiać wczytanie programu w postaci kodu maszynowego, deasemblację i wykonanie. Powinny być dostępne dwa tryby wykonania: ciągły i krokowy. W obu przypadkach emulator powinien obrazować stan rejestrów i magistrali wewnątrz procesora, jak również powinna istnieć możliwość podglądu i zmiany zawartości pamięci programu. Aplikacja powinna być zaimplementowana w języku Java.

**IV.** Uwagi dotyczące pracy: .....

**V.** Termin oddania pracy: **30 stycznia 2019**

**VI.** Konsultant:.....

**Kierownik Zakładu**
Zakładu Informatyki
Katedry Systemów Informatycznych
Wydziału Elektrotechniki, Automatyki i Informatyki
dr hab. Inż. Roman Stanisław Doniziak, prof. PŚk
*(pieczęć i podpis)*

**Opiekun pracy dyplomowej**
Aleksander Chrabąszcz
(podpis)
.....
*(imię i nazwisko)*

**Temat pracy dyplomowej celem jej wykonania otrzymałem:**

Kielce, dnia 29.04.2018 r Kowalczyk Tomasz
*czytelny podpis studenta*

# Politechnika Świętokrzyska

## WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI I INFORMATYKI

Kielce, dnia 16.02.2019

TOMASZ KOWALCZYK 083562
Imię i nazwisko studenta nr albumu

PIOTROWIEC 166, 26-070 KOPUSZNO
Adres zamieszkania

INFORMATYKA, SYSTEMY INFORMACYJNE, IV, STACJONARNE
Kierunek, specjalność, rok studiów, rodzaj studiów (stacjonarne, niestacjonarne)

DR INŻ. ARKADIUSZ CUROBOT
Opiekun pracy dyplomowej inżynierskiej/~~magisterskiej~~*

## OŚWIADCZENIE

Przedkładając w roku akademickim 2018/19 opiekunowi pracy dyplomowej inżynierskiej/~~magisterskiej~~*, powołanemu przez Dziekana Wydziału Elektrotechniki Automatyki i Informatyki Politechniki Świętokrzyskiej, pracę dyplomową inżynierską/~~magisterską~~* pod tytułem:

EMULATOR PROCESORA ZILOG Z80

oświadczam, że:

1) przedstawiona praca dyplomowa inżynierska/~~magisterska~~* została opracowana przeze mnie samodzielnie, stosownie do wskazówek merytorycznych opiekuna pracy,
2) przy wykonywaniu pracy dyplomowej inżynierskiej/~~magisterskiej~~* wykorzystano materiały źródłowe, w granicach dozwolonego użytku wymieniając autora, tytuł pozycji i miejsce jej publikacji,
3) praca dyplomowa inżynierska/~~magisterska~~* nie zawiera żadnych danych, informacji i materiałów, których publikacja nie jest prawnie dozwolona,
4) przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem stopnia zawodowego/naukowego w wyższej uczelni,
5) niniejsza wersja pracy jest identyczna z załączoną treścią elektroniczną (na CD i w systemie Archiwum Prac Dyplomowych).

Przyjmuję do wiadomości, iż w przypadku ujawnienia naruszenia przepisów ustawy o prawie autorskim i prawach pokrewnych, praca dyplomowa inżynierska/magisterska* może być unieważniona przez Uczelnię, nawet po przeprowadzeniu obrony pracy.

Zostałem uprzedzony:

1) o odpowiedzialności karnej wynikającej z art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz. U. 1994 Nr 24, poz. 83, t.j. Dz. U. 2018 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.",
2) o odpowiedzialności dyscyplinarnej na podstawie art. 307 ust. 1 i 2 ustawy z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668, z późn. zm.) „Student podlega odpowiedzialności dyscyplinarnej za naruszenie przepisów obowiązujących w uczelni oraz za czyn uchybiający godności studenta. Student może być ukarany przez rektora lub komisję dyscyplinarną".

Prawdziwość powyższego oświadczenia potwierdzam własnoręcznym podpisem.

Tomasz Kowalczyk
czytelny podpis studenta

*) niepotrzebne skreślić

# Emulator procesora Zilog Z80

**Streszczenie**

Praca ma na celu stworzenie programu emulującego procesor Zilog Z80. Aplikacja ta została podzielona na trzy niezależne części. Xbit to moduł upraszczający wykonywanie operacji arytmetycznych i bitowych na liczbach binarnych. Z80emu-core jest częścią aplikacji realizującą proces emulacji. Z80emu-gui implementuje interfejs użytkownika. Aplikacja pozwala wykonać program dla procesora Z80 w sposób ciągły lub krokowy. Interfejs użytkownika umożliwia modyfikację i podgląd rejestrów oraz pamięci procesora.
Słowa kluczowe:

- Emulator
- Zilog Z80
- Java 8
- Java FX
- Architektura 8-bitowa

# Zilog Z80 CPU emulator

**Summary**

The aim of the thesis is to develop an emulator of Zilog Z80 CPU. The application has been divided into three independent parts. Xbit is a module that simplifies arithmetic and bit operations on binary numbers. Z80emu-core is a part of the application that implements the emulation process. Z80emu-gui constitutes the user interface. The application allows the user to perform the program in a continuous or a step-by-step way.
Keywords:

- Emulator
- Zilog Z80
- Java 8
- Java FX
- 8-bit architecture

# Spis treści

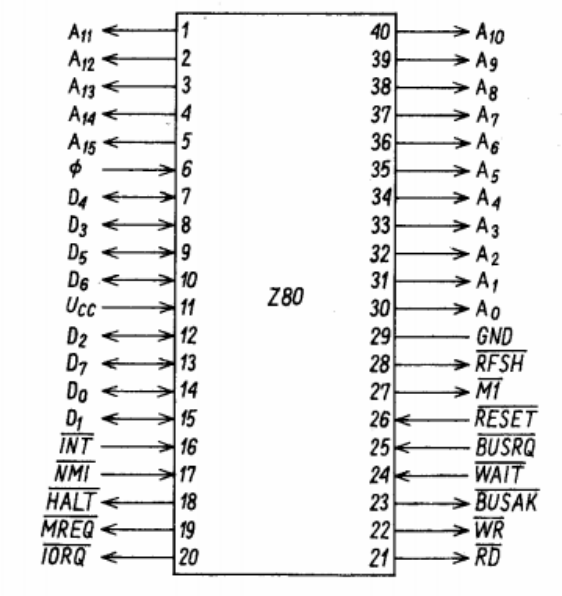
# Wstęp

Celem pracy było zaimplementowanie emulatora procesora Zilog Z80. Emulator to program komputerowy, który duplikuje zachowania systemu komputerowego, za pomocą innego system komputerowego. [7] W tym przypadku platformą docelową jest maszyna wirtualna Javy.

Procesor Zilog Z80 był bardzo popularny na rynku mikroprocesorów[6]. Stał się dostępny na rynku w roku 1976 i szybko zaczął być powszechnie stosowany w 8-bitowych systemach.

Jedną z przyczyn jego sukcesu, jest łatwy sposób sprzęgania go z innymi urządzeniami, szczególnie z kontrolerami pamięci. Inną jego zaletą jest lista rozkazów zgodna z popularnym w tamtym czasie procesorem Intel 8080, co umożliwia uruchamianie programów napisanych pierwotnie dla tego procesora [6].

Urządzenie to, mimo zalet, ma również poważne wady. Jego wewnętrzna budowa jest złożona jak na procesor ośmiobitowy. Piny magistrali danych nie są ułożone w logiczny sposób. Ich kolejność, jak ukazuje rysunek 1 (piny $D_1 - D_7$) jest w losowej kolejności.




Rysunek 1: Wyprowadzenia mikroprocesora Z80 [6]

Lista rozkazów procesora składa się z 158 pozycji, w tym 78 z nich jest zgodnych z Intel 8080[10]. Poznanie działania każdego z nich jest trudne i czasochłonne dla większości osób. Omawiany projekt emulatora ma na celu, oprócz samego wykonywania kodu programu, prezentować zmiany jakie zachodzą wewnątrz procesora. Użytkownik może prześledzić proces wykonywania dowolnej instrukcji i sprawdzić jak wpływa ona na pamięć, rejestry, czy flagi procesora. Wszystkie te wewnętrzne parametry można modyfikować podczas procesu emulacji, za pomocą graficznego interfejsu użytkownika, zaprezentowanego w rysunku 3.5.

W rozdziale pierwszym opisano proces emulacji oraz trzy sposoby jej wykonania, z naciskiem na metodę wybraną w projekcie.

Rozdział drugi zawiera przegląd wybranych emulatorów Zilog-a Z80. Skupiono się głównie na aplikacjach zawierających graficzny interfejs użytkownika tak jak opisywany projekt.

Rozdział trzeci przedstawia projekt aplikacji, wymagania funkcjonalne i niefunkcjonalne, oraz podział na moduły.

Rozdział czwarty opisuje implementacje poszczególnych modułów projektu. Wyjaśniono sposób wykonywania emulacji, realizacje operacji bitowych, oraz sposób integracji poszczególnych części aplikacji.

W rozdziale szóstym zawarto informacje na temat testów jednostkowych i manualnych aplikacji użytych w celu sprawdzenia poprawności wykonywania emulacji.

W rozdziale siódmym podsumowano projekt oraz opisano ewentualne możliwości rozwoju.

# 1. Zagadnienie emulacji

W rozdziale tym wyjaśniono, czym jest proces emulacji, na czym polega, oraz w jaki sposób może zostać on zaimplementowany. Zwrócono także uwagę na różnicę między pojęciami *symulacja* oraz *emulacja*, które są często błędnie używane zamiennie.

Emulator w dziedzinie informatyki, oznacza program który jest przystosowany do uruchomienia w określonym systemie komputerowym i pozwala na wykonywanie programów przeznaczonych dla innego tego typu urządzenia[1].

Inną ciekawą definicję emulatora podał Victor Moya del Barrio „Emulator to program komputerowy, którego zadaniem jest symulacja zachowania wszystkich komponentów danego urządzenia, używając oprogramowania uruchamianego na innym urządzeniu" [7].

Emulacje CPU można przeprowadzić na trzy sposoby:[3]

- przez interpretowanie,
- przez statyczną rekompilację,
- przez dynamiczną rekompilację.

Każda z tych metod wymaga oddzielnego przedstawienia.

## 1.1. Emulacja przez interpretowanie

Interpreter to najprostszy rodzaj emulatora. Odczytuje on w pętli kod programu z symulowanej pamięci. Odczytany bajt (lub bajty, gdyż rozkaz procesora może być wielobajtowy) zawiera informacje o rodzaju operacji jaką CPU powinien wykonać. Interpreter ma za zadanie odkodować informacje o operacji, a następnie ją wykonać. Między wykonaniem kolejnych rozkazów powinien on zmienić wirtualne parametry (np. zwiększyć o jeden wartość licznika rozkazów), sprawdzić czy nie zostało zgłoszone przerwanie, obsłużyć urządzenia wejścia/wyjścia, liczniki, kartę graficzną, lub wykonać inne operacje zależne od emulowanego urządzenia. Schemat struktury interpretera został przedstawiony na listingu 1.1.

Emulacja przez interpretowanie jest najwolniejszą formą emulacji, ale także najłatwiejszą w debugowaniu. Pozwala ona na prześledzenie wykonania operacji i podgląd

```
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++;
}
```

Listing 1.1: Schemat struktury interpretera procesora

wewnętrznych stanów urządzenia. Z tego powodu jest najczęściej wybierana przez programistów do tworzenia debuggerów procesorów oraz mikrokontrolerów.

## 1.2. Statyczna rekompilacja

Statyczna rekompilacja (ang. *static binary translaion*) to proces konwertowania kodu maszynowego na inny kod maszynowy przeznaczony dla docelowej platformy sprzętowej. Plik wykonywalny tłumaczony jest w całości i tylko raz. Wadą tego rozwiązania są problemy z tłumaczeniem rozkazów 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[13].

## 1.3. Dynamiczna rekompilacja

Dynamiczna rekompilacja (ang. *Dynamic binary translator*), w odróżnieniu od translacji statycznej, tłumaczy kod blokami podczas jego wykonywania. Dokonywana jest ona „na żądanie" w związku z tym jest wolniejsza od rekompilacji statycznej, ale rozwiązuje problem związany z tłumaczeniem rozkazów skoków pośrednich.

Raz przetłumaczony fragment kodu jest przechowywany w pamięci, na wypadek po-

| Symulator | Emulator |
|---|---|
| System zdolny do naśladowania innego systemu w pewnym stopniu. | System który naśladuje dokładne zachowanie innego systemu. |
| Może nie przestrzegać wszystkich reguł symulowanego systemu. | Ściśle przestrzega parametrów i reguł emulowanego systemu. |
| Modeluje aplikacje i zdarzenia. | Kopiuje zachowanie systemów. |

Tabela 1.1: Różnice między symulacją a emulacją [9]

trzeby jego ponownego użycia, co pozwala zwiększyć efektywność tej metody emulacji[13].

Dynamicznej rekompilacji używa w dużym stopniu maszyna wirtualna języka Java. Wczesne wersje JVM (ang. *Java Virtual Machin*) używały interpreterów, co okazało się mało wydajne. Dobrym sposobem na poprawę efektywności maszyny wirtualnej Java jest dynamiczne tłumaczenie kodu bajtowego[4].

## 1.4. Różnica między symulacją a emulacją

Różnicę między emulacją a symulacją obrazuje tabela 1.1.

W informatyce, symulator to program komputerowy, który modeluje zachowania i funkcje innego realnego systemu lub zjawiska (np. prowadzenie pojazdu). Nie jest wymagane, aby odwzorowywał wszystkie jego zachowania i funkcje. Symulator nie wykonuje realnych zadań symulowanego urządzenia i nie zastępuje go[9].

Natomiast emulator ma za zadanie „udawać" dane urządzenie/zjawisko w takim stopniu i na takim poziomie, aby był w stanie zastąpić emulowane urządzenie i funkcjonować tak jak ono[2].

W rozdziale zdefiniowano, czym jest emulacja w dziedzinie informatyki. Przybliżono jej realizacje przez interpretacje oraz statyczną i dynamiczną rekompilacje. Na koniec zwrócono uwagę, że wbrew powszechnej opinii emulacja i symulacja nie są synonimami, oraz wytłumaczono różnicę między nimi.

# 2. Przegląd istniejących rozwiązań

Procesor Zilog Z80 jest często emulowany, ze względu na jego dużą popularność. Emulatory tworzone są zarówno przez duże firmy, jak i hobbystów. Na platformie *Github* która jest przeznaczona dla projektów programistycznych, można znaleźć około dwustu repozytoriów z projektami emulującymi Z80, lub urządzenia używające tego procesora.

W tym rozdziale zaprezentowane są najciekawsze emulatory, które posiadają graficzny interfejs użytkownika i pozwalają na wgląd w wewnętrzne stany procesora (czyli najbardziej przypominające zakresem swoich funkcji aplikację będącą przedmiotem tej pracy). Przedstawione są zarówno komercyjne rozwiązania, jak i te pisane przez amatorów.

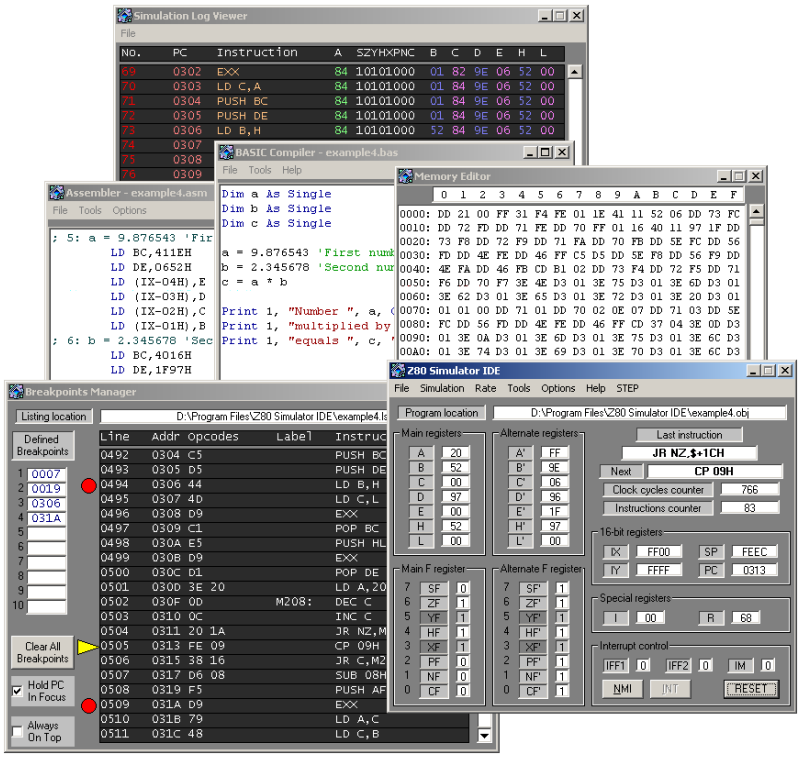
## 2.1. Z80 SIMULATOR IDE

Jest to dostępny pod adresem http://www.oshonsoft.com/z80.html płatny emulator posiadający najbardziej rozbudowany interfejs z wszystkich opisanych programów. Pozwala on na prezentowanie wewnętrznych stanów procesora, manipulację przerwaniami i portami wejścia/wyjścia. Zawiera edytor pamięci działający również podczas emulacji. Posiada on również funkcje i elementy typowe dla debuggerów, jak możliwość wstrzymania działania programu w określonym miejscu, tryb pracy krokowej, interaktywny edytor i kompilator kodu asemblera[15]. Widok interfejsu użytkownika tego programu jest zaprezentowany na rysunku 2.1.

Jedną z wad emulatora Z80 SIMULATOR IDE jest interfejs, który nie jest intuicyjny. Przykładowo, autorzy programu nie umieścili w nim informacji, o tym w jakim formacie powinny być wprowadzane wartości liczbowe. Brak w programie systemu pomocy i opisów, co może znacznie utrudniać pracę początkującym użytkownikom. Dodatkowo jest to narzędzie płatne, przeznaczone dla specjalistów i uruchamiane tylko w systemie MS Windows.

## 2.2. ZEMU - Z80 Emulator Joe Moore

ZEMU to emulator zaprojektowany głównie po to, aby umożliwiać uruchamianie systemu operacyjnego CPM, który był oferowany przez firmę *Digital Research Inc.* w latach 1970-1980[5]. Program skierowany jest do hobbystów. Oprócz standardowych możliwo-




Rysunek 2.1: Z80 SIMULATOR IDE

ści takich jak podgląd, edycja zawartości pamięci, rejestrów i flag, może on również emulować stację dyskietek, port COM, szeregowy, monitor CRT i drukarkę.

Rysunek 2.2 przedstawia interfejs aplikacji. Tak, jak w przypadku Z80 SIMULATOR IDE nie jest on intuicyjny, brak mu systemu pomocy, jego elementy 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.

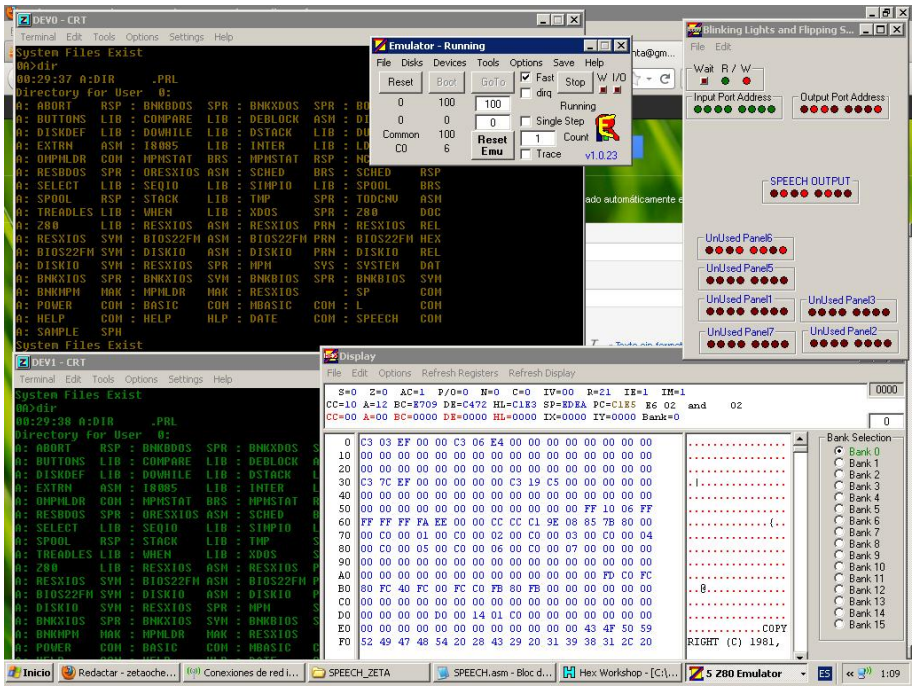
Inną wadą aplikacji jest brak możliwości jej uruchomienia w innym systemie operacyjnym niż *Windows*.

## 2.3. ZIM - The Z80 Machine Simulator

ZIM został napisany z użyciem technologii *Java Web-Start*. Pozwala ona na uruchomienie aplikacji bezpośrednio na stronie internetowej, ale również pozwala na dostęp do lokalnych zasobów komputera, np. plików. Aplikacja według autora przeznaczona jest głównie dla studentów uczących się języka asemblera dla procesora Z80[11].

Program pozwala na podgląd wszystkich wewnętrznych parametrów CPU, emuluje proste urządzenia wejścia/wyjścia, umożliwia edycję zawartości pamięci, debugowanie programu dla procesora Z80 i symulację przerwań. Zaletą emulatora jest możliwość jego uruchamiania na wielu platformach sprzętowych, dzięki zastosowaniu języka Java.

W aplikacji brakuje natomiast edytora asemblera i systemu pomocy. Interfejs użyt-



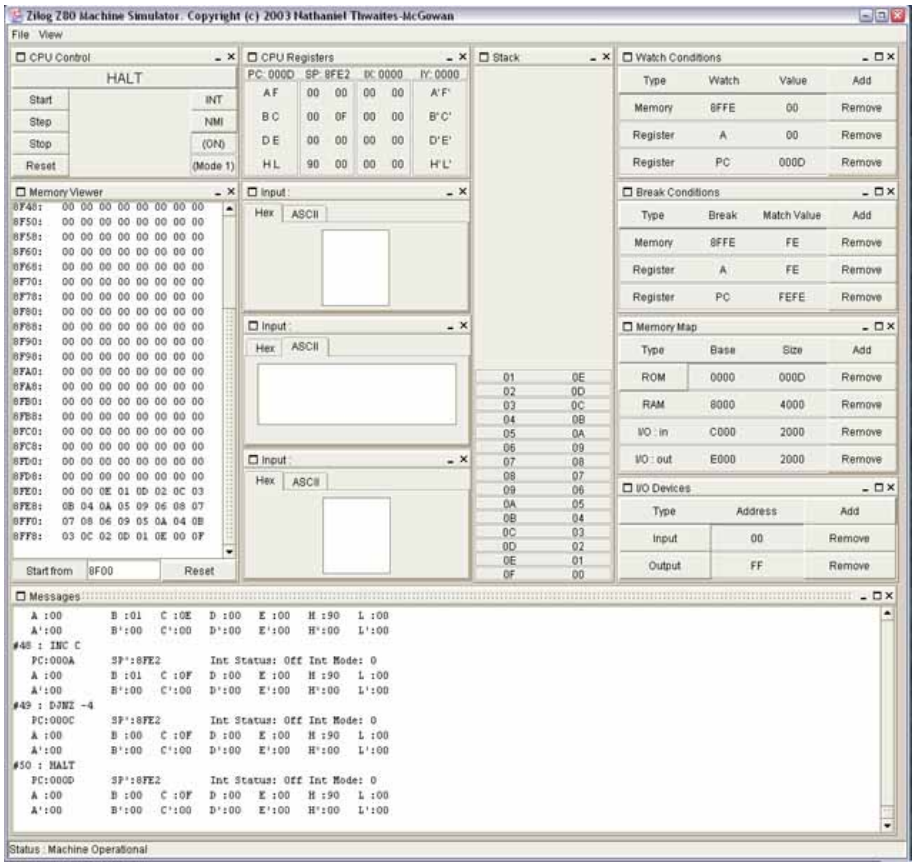

Rysunek 2.2: ZEMU

kownika ma wiele defektów i nie jest intuicyjny. Na rysunku 2.3 przedstawiono zrzut ekranu działającej aplikacji.

## 2.4. Podsumowanie istniejących rozwiązań

Problemem prawie wszystkich emulatorów i symulatorów jest ich interfejs użytkownika. Ta uwaga dotyczy nie tylko emulatorów 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 odpowiedniej wagi do intuicyjności i systemów pomocy dla interfejsów użytkownika. Osoby nieposiadające wymaganej wiedzy albo znające jedynie podstawy zagadnienia nie są w stanie sprawnie obsługiwać programu.

Emulatory nie są przeznaczone do uruchamiania na wielu platformach sprzętowych. Dotyczy to szczególnie aplikacji emulujących przez rekompilację. Do jej wykonania wymagana jest znajomość architektur wyjściowej i docelowej maszyny. Rozwiązaniem tego problemu mogą być interpretery napisane w języku Java, takie jak „ZIM - The Z80 Machine Simulator". Rozwiązanie to wykorzystuje dużo zasobów procesora. Kod przeznaczony dla emulowanej maszyny jest najpierw interpretowany przez interpreter napisany w języku Java, a następnie ponownie emulowany już za pomocą dynamicznej rekompilacji w maszynie wirtualnej. Typowe rozwiązania napisane w języku kompilowanym dla konkretnego procesora działają szybciej, kosztem możliwości uruchamiania na wielu




Rysunek 2.3: ZIM - The Z80 Machine Simulator

platformach sprzętowych. Wydajność maszyny wirtualnej języka Java nie jest przeszkodą do stworzenia w tym języku emulatora procesora Zilog Z80. Współczesne komputery są na tyle efektywne, że taki emulator może pracować z wydajnością zbliżoną do wydajności oryginalnej maszyny.

Kolejną kwestią o której warto wspomnieć jest czytelność kodu emulatorów. Kod istniejących rozwiązań nie należy do łatwych do zrozumienia. Najczęściej cały program w postaci źródłowej emulatora zawiera się w jednym pliku o wielkości kilkuset wierszy. Osoba chcąca przestudiować proces emulacji ma zatem utrudnione zadanie.

Podsumowując, aktualnie brakuje wieloplatformowego emulatora lub symulatora Zilogа Z80, z czytelnym, poprawnie działającym interfejsem użytkownika, czytelnym, dobrze opisanym kodem źródłowym i użytecznym systemem pomocy.

# 3. Projekt aplikacji

Podczas projektowania aplikacji, ustalono jej wymaganie funkcjonalne oraz niefunkcjonalne. Opracowano strukturę projektu, która zakłada podział programu na trzy mniejsze części. Zamieszczono diagramy UML przedstawiające publiczne interfejsy programistyczne tych modułów.

## 3.1. Wymagania funkcjonalne

Założono, że aplikacja będzie posiadać graficzny interfejs użytkownika umożliwiający podgląd i modyfikacje stanów wewnętrznych procesora, takich jak rejestry, pamięć i flagi. Możliwe będzie wykonanie dwóch trybów emulacji, ciągły oraz krokowy. W widocznym miejscu zostaną umiejscowione informacje o liczbie wykonanych cykli maszynowych i zegara. Aplikacja pozwoli na zgłaszanie przerwań maskowalnych i niemaskowalnych oraz wprowadzenia wartości szyny danych, jaką ustaliłoby urządzenie wywołujące przerwanie podczas zgłoszenia. Użytkownikowi zostanie udostępniona możliwość wprowadzenia kodu asemblera do aplikacji, skompilowanie go i wczytanie do pamięci. Podczas emulacji zostanie wyświetlona nazwa poprzedniej i następnej wykonanej instrukcji procesora.

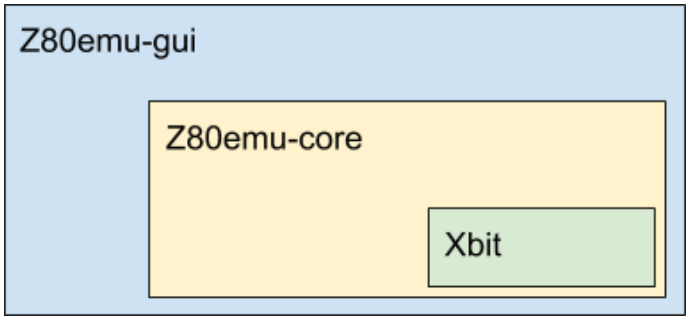
## 3.2. Wymagania niefunkcjonalne

Podstawowym założeniem, jest możliwość uruchomienia aplikacji na jak największej liczbie platform. Jako minimum zdecydowano się na systemy operacyjne *MS Windows*, *Ubuntu* i *Mac OS*. Instrukcje procesora powinny być wykonywane bezbłędnie, zgodnie z dokumentacją techniczną procesora. Projekt powinien mieć interfejs programistyczny umożliwiający podłączenia go z innymi aplikacjami, np. emulatorami urządzeń, które do swojej pracy wykorzystują Ziloga Z80.

## 3.3. Struktura aplikacji

Aplikację podzielono na trzy moduły: *Xbit*, *Z80emu-core* oraz *Z80emu-gui*, z których każdy jest osobnym modułem. Zależności między nimi pokazuje diagram 3.1. Moduł *XBit*

do swojego działania nie wymaga innych modułów. *Z80emu-core* potrzebuje modułu *XBit* do poprawnego działania i jest on jego częścią, natomiast *X80emu-gui* zawiera w sobie *Z80emu-core*, a co za tym idzie także *Xbit*.




Rysunek 3.1: Zależności pomiędzy modułami aplikacji

### 3.4. Biblioteka 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 jest wydajnym rozwiązaniem.

Innym poważnym problemem języka Java jest brak typów prostych pozwalających przechowywać wyłącznie liczby dodatnie. Problem rozwiązuje biblioteka XBit. Zawiera ona klasy, które umożliwiają przechowywanie wartości jedno i dwu bajtowych. Mogą one zostać zinterpretowane jako liczby zapisane w kodzie dopełnień do dwóch (liczby dodatnie i ujemne), lub naturalnym kodzie binarnym (tylko liczby dodatnie).
Przykładowo, liczba binarna 1110 odczytana w naturalnym kodzie binarnym (NKB), to 15 (w zapisie decymalnym), natomiast w kodzie dopełnień do dwóch (U2) to -2. XBit pozwala na obydwie interpretacje za pomocą odpowiednich metod, zwracając wynik jako typ prymitywny *int*.

Nie jest to idealne rozwiązanie. Typ *int* przechowuje cztero-bajtowe liczby z zakresu od -2 147 483 648 do 2 147 483 647, więc większość bajtów zostaje niewykorzystana. Nadmiarowość jest w tym przypadku wymagana, ponieważ zakres liczb w notacji NKB i w U2 jest inny.

Przykładowo, w przypadku gdy za pomocą XBit stworzono reprezentację liczby ośmio bitowej:
1111 0000 (w kodzie jest to NKB=240, U2=-16) to wywołując metodę interpretującą ją jako liczba bez znaku (czyli w notacji NKB), zostanie zwrócona zmienna o typie prymitywnym *int*, o wartości 240, binarne
00000000 00000000 00000000 11110000.

Natomiast jeśli wykonamy metodę interpretującą ją jako liczba ze znakiem (czyli w notacji U2) zostanie zwrócona wartość -16, binarnie
11111111 11111111 11111111 11110000.

Bibliotekę zaprojektowano w taki sposób, aby była jak najbardziej uniwersalna, i można ją było wykorzystać do budowy emulatora Zilog-a Z80, ale także innych procesorów.

### 3.4.1 Możliwości biblioteki XBit

Założono, że biblioteka *XBit* będzie spełniała następujące wymagania:

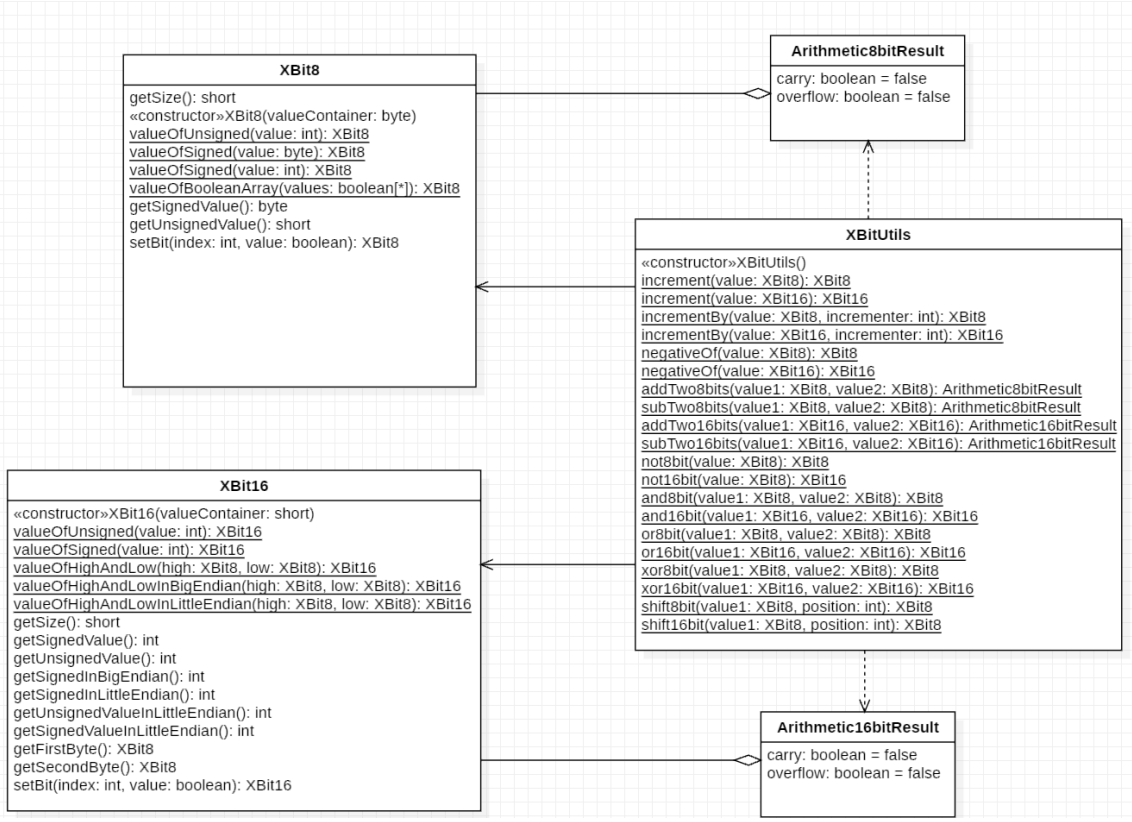
- reprezentacja liczb 8 i 16 bitowych,
- interpretacja liczb w naturalnym kodzie binarnym lub dopełnieniu do dwóch,
- operacje na pojedynczych bitach (możliwość zmiany, odczytu bitu na danej pozycji),
- opcja odczytania grupy bitów (odczytanie kilku bitów z określeniem pozycji pierwszego i ostatniego 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.

### 3.4.2 Założenia projektowe XBit

Przed przystąpieniem do implementacji rozwiązania, zaprojektowano publiczny interfejs biblioteki w języku UML, zaprezentowany na diagramie 3.2. Ustalono także założenia projektowe, które zaprezentowano poniżej.

#### Klasy XBit8 i XBit16

Klasy *XBit8* oraz *XBit16* służą do reprezentowania liczb 8 i 16 bitowych. Postanowiono, że zmiana stanu wewnętrznego ich obiektów, czyli wartości w nich przechowywanych będzie niemożliwa (są to obiekty niezmienne, z ang. *immutable*), tak jak w przypadku obiektowych reprezentacji typów prostych w języku Java (*Short*, *Long*, *Integer* itp). Konsekwencją tej decyzji jest to, że metody które powinny zmienić stan obiektu, klonują




Rysunek 3.2: Hierarchia klas biblioteki XBit

istniejący obiekt a następnie modyfikują odpowiednio stan jego kopii. Przykładem takiej metody jest *setBit(index: int, value: boolean)* występująca w obu klasach. Podprogram ten nie zmienia bitu obiektu na rzecz którego został wykonany, a jedynie zwraca kopie obiektu, ze zmodyfikowanym odpowiednim bitem.

Zalety zastosowania obiektów niezmiennych:

- prosta implementacja oraz łatwe debugowanie kodu,
- *Garbage Collector* jest przystosowany do pracy z tego typu obiektami,
- łatwość zapisywania obiektów do pliku lub pamięci podręcznej (z ang. *cache*),
- bezpieczne używanie obiektów niezmiennych w programach wielowątkowych, co ułatwia emulacje procesorów wielopotokowych; jeden wątek w takim przypadku mógłby być odpowiedzialny za jeden stopień potoku, np osobny wątek odbierałby instrukcje z pamięci, inny by je dekodował, kolejny wykonywał i tak dalej,
- Możliwość użycia obiektu jako klucza, np. w *HashMap*.

Wadą zastosowania obiektów niezmiennych jest 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 tego typu obiektów niezmiennych w bibliotece.

#### Klasy 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, informują o wystąpieniu przeniesienia lub przepełnienia. Ponieważ język Java, jako wynik metody może zwrócić tylko jeden obiekt, postanowiono zaprojektować klasy *Arithmetic8bitResult* oraz *Arithmetic16bitResult* które grupują te trzy informacje w jedną klasę, której instancja zostanie zwrócona po wykonaniu operacji arytmetycznej. Klasy te zawierają następujące pola:

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

### 3.5. 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 pozwalałby na użycie *Z80emu-core* w innych projektach, które emulują urządzenia zbudowane z użyciem Ziloga Z80. Jako przykład można podać emulator przenośnej konsoli *Game Boy* firmy *Nintendo* z 1989 roku, której procesorem jest Zilog Z80.

Wymagania projektowe względem modułu są następujące:

- możliwość wykonania wszystkich 158 rozkazów procesora,
- istnienie zestawu metod umożliwiających zmianę stanów rejestrów,
- emulacja zewnętrznej pamięci,
- możliwość podłączenia emulowanych urządzeń wejścia/wyjścia,
- umożliwienie zgłaszania przerwań maskowanych i niemaskowanych. (Z80 posiada dwa rodzaje przerwań).

#### 3.5.1 Publiczny interfejs modułu Z80emu-core

Publiczny zestaw metod służący do zarządzania procesem emulacji, jak i urządzeniem, pokazano na diagramie 3.3.

Klasa Z80 zawiera zestaw metod sterujących emulacją, oraz wartościami liczników i mniejszych rejestrów procesora. Najważniejsze z nich to:

- *Z80(memory: Memory, ioDevice: IoDevice)* - konstruktor, przyjmujący dwa parametry wymagane do poprawnego działania. Parametry „memory" oraz „ioDevice" to obiekty reprezentujące moduł pamięci oraz urządzenie wejścia/wyjścia podłączone do procesora. Użytkownik używający modułu *Z80emu-core* powinien samemu zaimplementować ich działanie, w zależności od zastosowania, dla którego chce emulować urządzenie.

- *runOneInstruction()* - metoda wykonująca pojedynczą instrukcję procesora.

- *makeInterupt(addressBus: XBit8), makeNonMaskableInterupt(addressBus: XBit8)* - metody powinny zostać wykonane między kolejnymi wywołaniami *runOneInstruction()*. Ich zadaniem jest zgłaszanie przerwań. Parametr *addressBus* to wartość jaka zostałaby ustalona na magistrali danych podczas przerwania, gdyby zostało ono wykonane w prawdziwym urządzeniu.

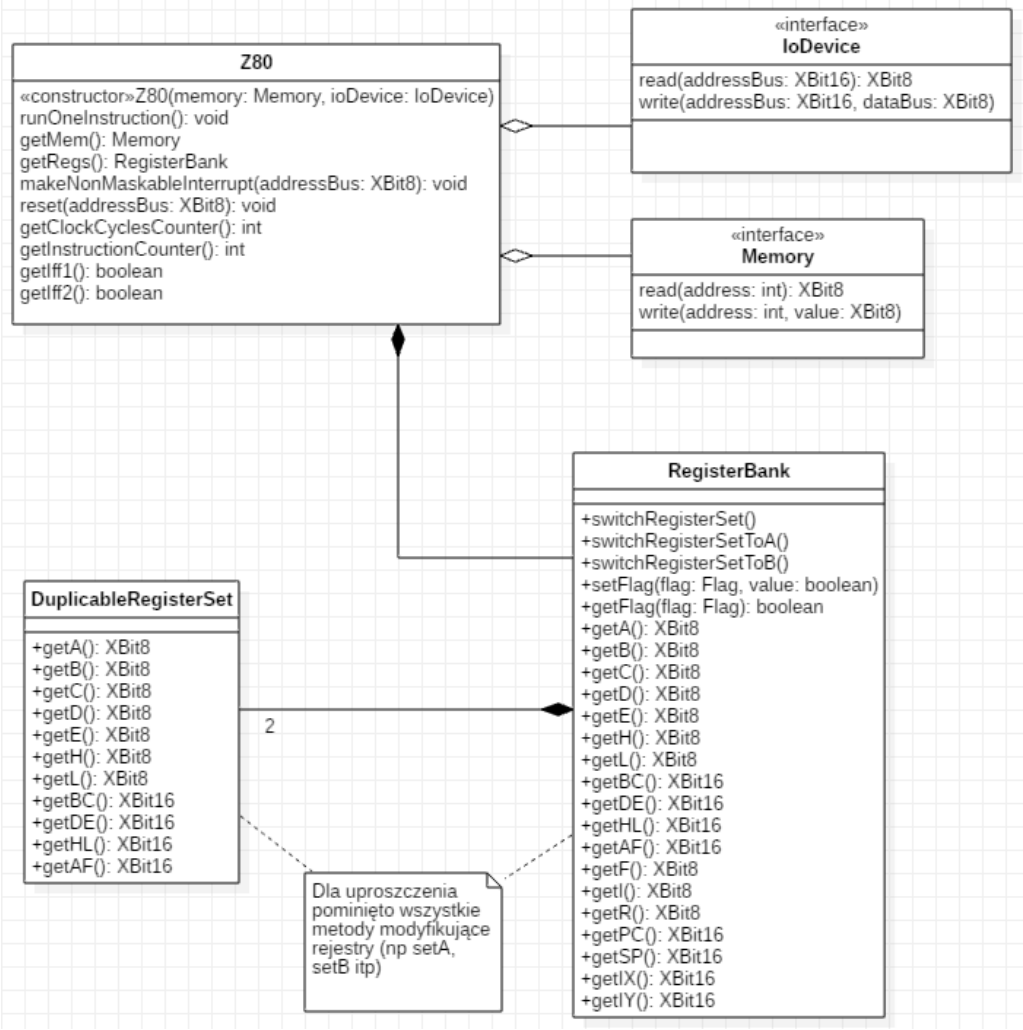
Procesor Zilog Z80 posiada dwa zestawy rejestrów ogólnego przeznaczenia (do nich należąc rejestry A,B,C,D,E,H,L,F oraz ich 16bitowe odpowiedniki). Takie rozwiązanie jest dogodne, w przypadku gdy procesor często wykonuje obsługę przerwań. W klasycznym podejściu, w ramach obsługi przerwania zestaw rejestrów ogólnego przeznaczenia zapisywany jest na stosie, co jest czasochłonną operacją. Projektanci Ziloga Z80 postanowili stworzyć drugi alternatywny zestaw rejestrów ogólnego przeznaczenia, który jest używany podczas obsługi przerwania. W takim przypadku nie jest wymagane odłożenie wartości rejestrów 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.

### 3.5.2 Łączenie z innymi projektami

Postanowiono nie uruchamiać procesu emulacji w pętli głównej tak jak w klasycznym podejściu pokazanym w kodzie 1.1. Zamiast tego udostępniono jedną metodę *runOneInstruction()* która wykonuje jeden rozkaz procesora.

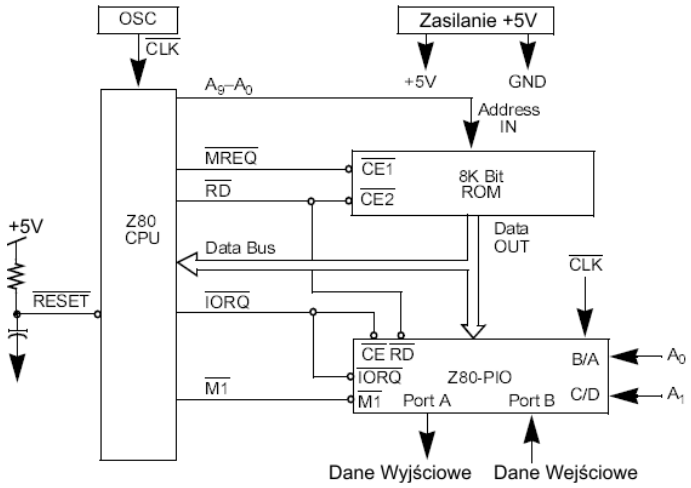
Metody sterujące procesem emulacji (np. wywołanie przerwania, edycja rejestrów) powinny zostać wywoływane między kolejnymi wywołaniami *runOneInstruction()*. Głów-




Rysunek 3.3: Diagram hierarchii klas dla modułu z80emu-core

na pętla emulacji powinna zostać zaimplementowania w module nadrzędnym, używającym *z80emu-core*. Pozwala to na większą elastyczność modułu w łączeniu go z innymi projektami.

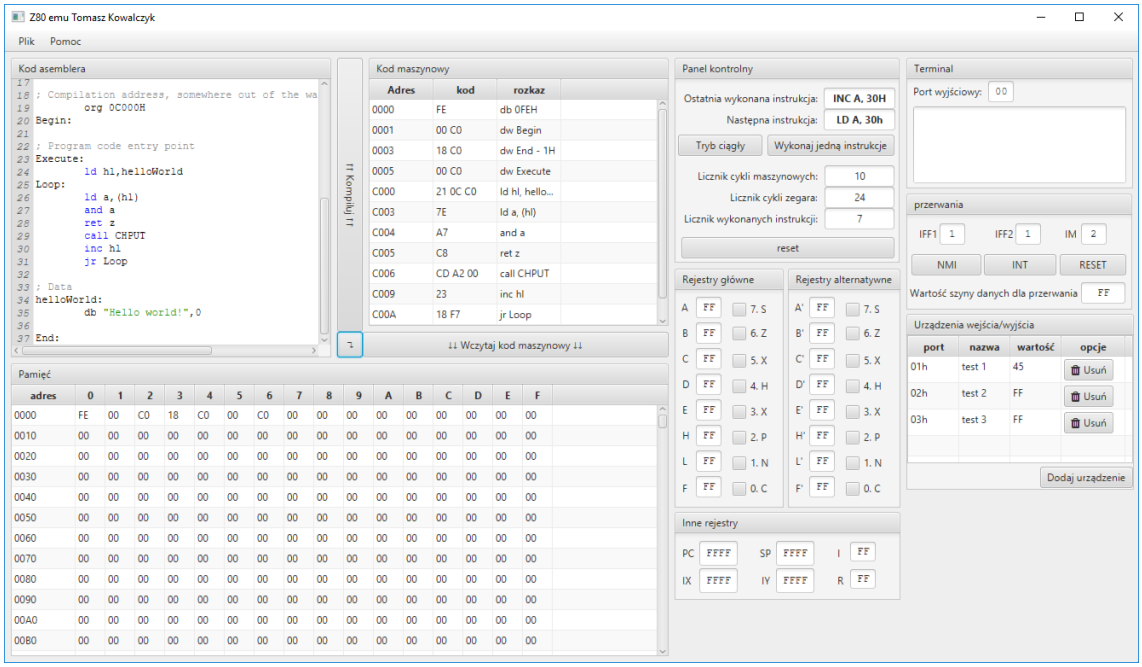
Inną ważną decyzją projektową pozwalającą na zwiększenie elastyczności projektu, było zaprojektowanie interfejsów *IoDevice* oraz *Memory* i decyzja o zaimplementowaniu jedynie prostych reprezentacji klas implementujących te interfejsy. Aby uzasadnić powód wprowadzenia tego rozwiązania na rysunku 3.4 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 3.4: Minimalny System Komputerowy Z80

## 3.6. Z80emu-gui

*Z80emu-gui* to moduł realizujący interfejs użytkownika. Został on napisany z pomocą biblioteki *JavaFX*. Projekt interfejsu obejmował stworzenie makiety w języku FXML zaprezentowanej na grafice 3.5. Większość interfejsów emulatorów posiada GUI złożone z wielu okien, które można dowolnie zamykać i otwierać, a każde z nich zawiera osobny moduł. Dla przykładu, w *Z80 simulator IDE*, edytor pamięci, asembler lub manipulacja urządzeniami wejścia wyjścia zawarte są w osobnych oknach. W projekcie *Z80emu-gui* postanowiono stworzyć interfejs zawierający wszystkie funkcje w jednym oknie.



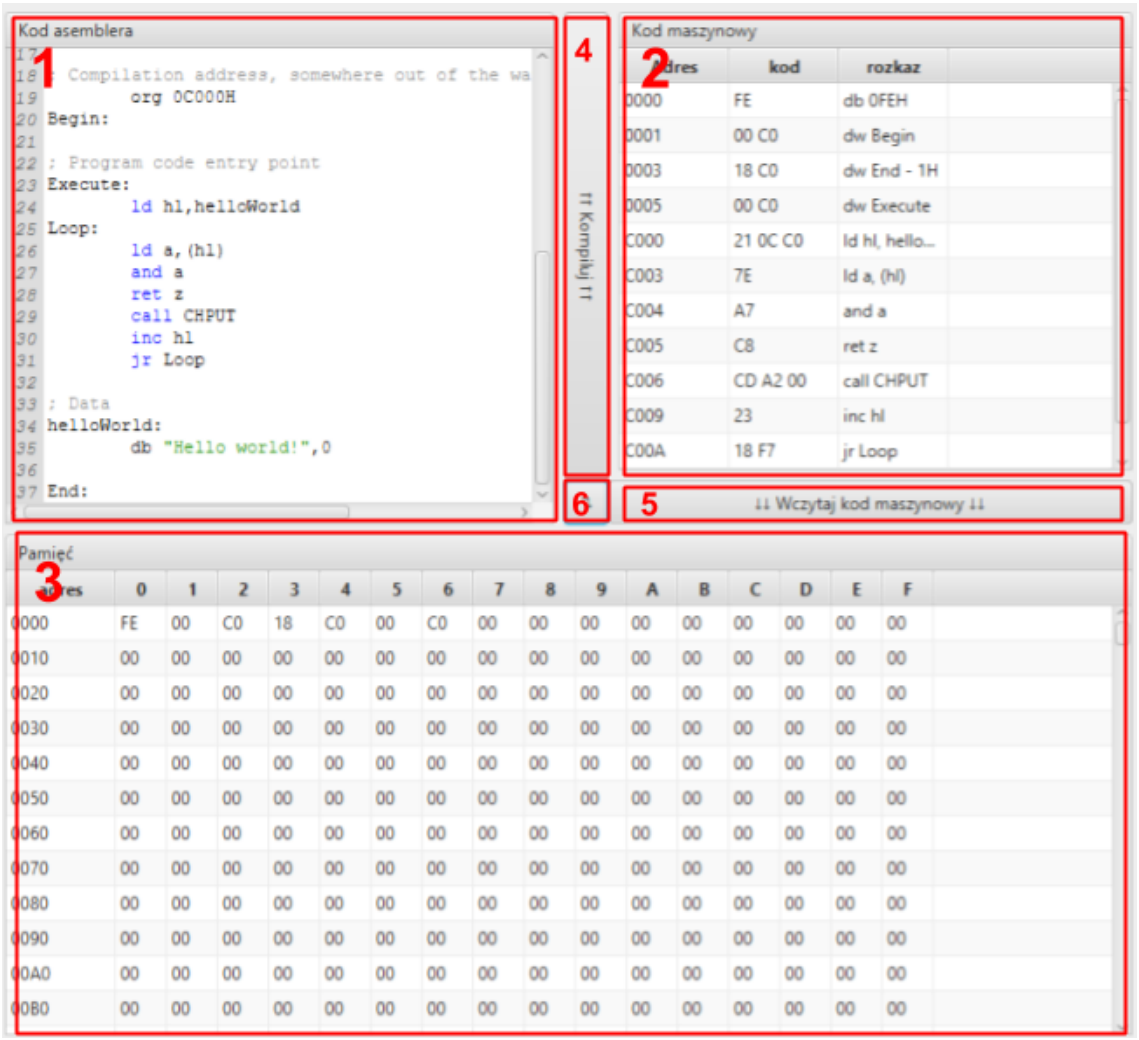

Rysunek 3.5: Makieta interfejsu użytkownika

Elementy interfejsu użytkownika podzielono na dwa zbiory w celu czytelniejszego objaśnienia ich roli. Pierwszy zbiór zaprezentowany na grafice 3.6 zawiera elementy związane ze sterowaniem 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 informacje 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).

Drugą część interfejsu zaprezentowano na grafice 3.7. Oto opis poszczególnych elementów:

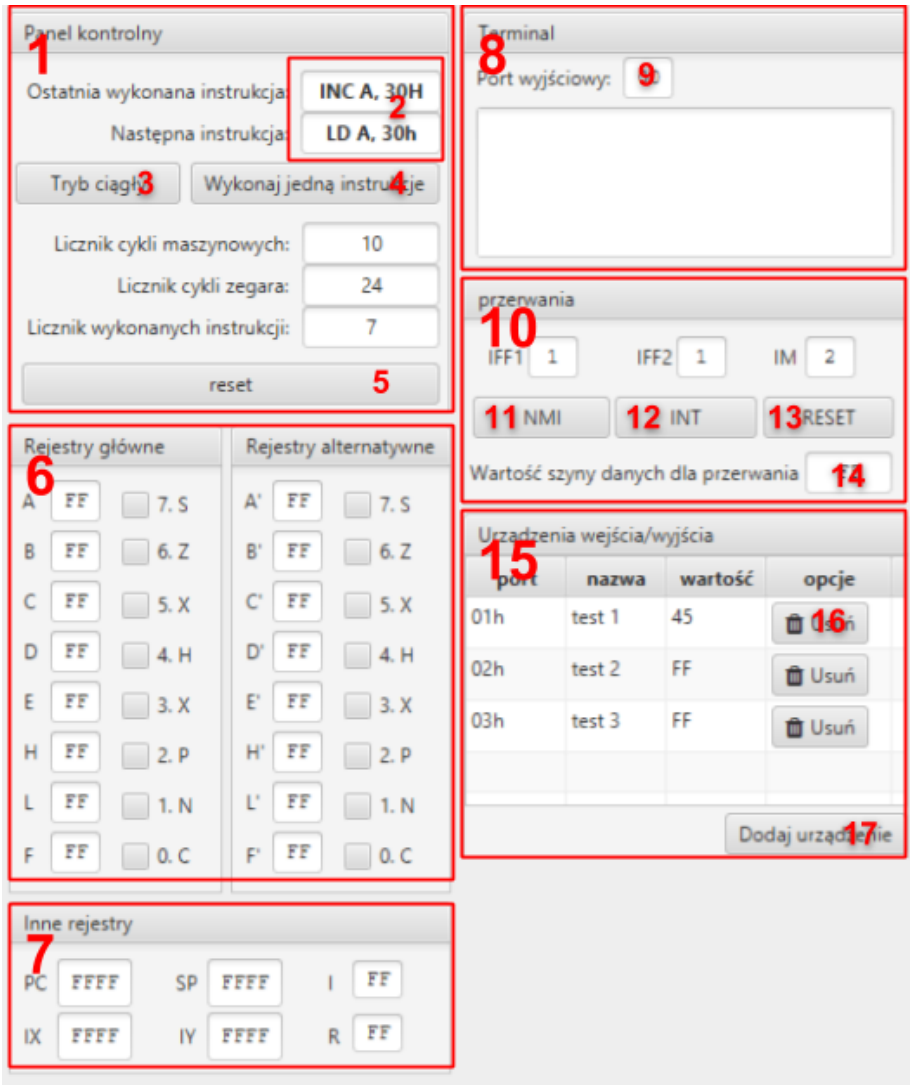
1. główny panel kontrolujący proces emulacji, przedstawiający między innymi liczbę cykli maszynowych i zegara wykonywanego programu,
2. pola prezentujące nazwę ostatniej wykonanej i następnej w kolejce instrukcji procesora,
3. przycisk uruchamiający emulację ciągłą (ponownie jego wybranie zatrzymuje emulacje),
4. przycisk wykonujący pojedynczy, następny rozkaz 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, do którego wpisywany jest numer portu terminala wyjściowego, pod jakim będzie przyjmował on dane,
10. panel grupujący elementy interfejsu odpowiedzialne za przerwania,




Rysunek 3.6: Makieta interfejsu użytkownika z oznaczeniami. Część 1

11. przycisk generujący NMI (z ang. *Non-Maskable Interrupt*), sygnał przerwania niemaskowanego,
12. przycisk generujący INT (z ang. *Interrupt*), sygnał maskowanego przerwania,
13. przycisk generujący sygnał resetu,
14. pole tekstowe zawierające 8-bitową wartość, jaką przyjmie szyna danych podczas zgłoszenia przerwania,
15. tabela z listą podłączonych urządzeń wejścia/wyjścia,
16. przycisk pozwalający na usunięcie urządzenia wejścia/wyjścia,
17. przycisk dodający nowe urządzenie wejścia/wyjścia.

W rozdziale opracowano wymagania funkcjonalne i niefunkcjonalne aplikacji. Zaprojektowano strukturę projektu podzieloną na trzy moduły. Głównym kryterium, jakie przy-




Rysunek 3.7: Makieta interfejsu użytkownika z oznaczeniami. Część 2

jęto podczas projektowania, jest możliwość ponownego użycia każdego z nich w innym projekcie emulatora. W ramach projektu stworzono także makietę interfejsu użytkownika w języku FXML.

# 4. Implementacja

W rozdziale tym przedstawiono proces implementacji projektu. Najpierw opracowano moduł *XBit*, następnie *Z80emu-core* i na końcu *Z80emu-gui*. Implementację każdego z nich opisano w osobnym podrozdziale. Zaprezentowano najciekawsze fragmenty kodu źródłowego i je opisano.

## 4.1. XBit

*XBit* to moduł mający za zadanie ułatwić operacje bitowe w języku Java, 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 moduł *XBit* implementujący taką operację. Potrafi on odczytać liczbę n-bitową i zwrócić jej wartość jako zmienną typu *int*, interpretując ją jako bity w U2 lub NKB.

Wewnątrz obiektów reprezentujących liczby binarne, wartość przechowywana jest w polu typu *int*, które nazwano *valueContainer*. Może ono przyjąć teoretycznie wartości od -2147483648 do 2147483647. Uznano jednak, że atrybut ten przechowywał będzie wartość, jaką reprezentują przechowywane bity w notacji NKB (czyli tylko wartości dodatnie). Z tego powodu wartość *valueContainer* nigdy nie będzie ujemna.
Zakres wartości tego pola to od 0 do $2^n - 1, n < 32$ gdzie $n$ oznacza liczbę bitów, jaką przechowuje dany obiekt. Ponieważ typ *int* przechowywany jest w czterech bajtach i interpretowany jest przez język Java w notacji U2, maksymalnie może przechować wartość 2 147 483 647. Dla wartości interpretowanej w NKB na czterech bajtach jest to
$2^{32} - 1 = 4\ 294\ 967\ 296$. Jest to więcej niż typ *int* może zawierać, dlatego maksymalna możliwa binarna reprezentacja, jaka może zostać zapisana w podobny sposób, to liczba przechowywana na 31 bitach.

Jeśli obiekt *XBit8* będzie przechowywał liczbę binarną 11111111 (czyli odczytując w NKB to decymalne 255, a w U2 decymalnie -1). Wartość jego pola *valueContainer* będzie wynosić w kodzie binarnym 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 liczba w kodzie U2, której najbardziej znaczący bit ma wartość 1, jest ujemna.

Podsumowując, *XBit* wykorzystuje fakt takiego samego zapisu liczb dodatnich w U2

i NKB, jeśli tylko będą one zapisane na większej liczbie bitów niż wymagana.

### 4.1.1 Implementacja klasy *XBit*

*XBit* to klasa abstrakcyjna zawierająca wspólne elementy dla klas *XBit8* i *XBit16* oraz, ewentualnych klas dziedziczących.W tym podrozdziale zawarto opis najbardziej istotnych metod tej klasy.

#### 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  }
```

Listing 4.1: Metoda boolean getBit(int index)

Na listingu 4.1 zaprezentowano metodę *getBit* zwracającą wartość bitu na pozycji o numerze przekazanym w parametrze. W linii nr 2 sprawdzana jest poprawność parametru *index*. Jeśli jest on niepoprawny, w linii nr 3 wyrzucany jest wyjątek *NumberFormatException*. W linii nr 5 metoda wykonuje przesunięcie bitowe pola *valueContainer* o podany index. Ponieważ wynikiem jest liczba o typie *int*, wykonywana zostaje koniunkcja bitowa z maską o wartości 1. Na końcu wynik operacji jest przekształcany z typu *int* na *boolean* i zwracany.

#### 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 }
```

Listing 4.2: Metoda TSelf setBit(int index, boolean value)

Listing 4.2 prezentuje metodę nadającą bitowi o określonej pozycji, zadaną wartość, a następnie zwracającą nowy zmodyfikowany obiekt. Typ *TSelf* zwracanej wartości to typ generyczny reprezentujący docelową liczbę n-bitową. W linii nr 2 metoda tworzy maskę bitową, wykonując przesunięcie bitowe na liczbie 1 o wartość parametru *index*, czyli maska będzie posiadała tylko jeden bit ustawiony na wartość 1, i będzie to bit o pozycji z parametru. W liniach 5 i 7 metoda wykonuje operacje bitowe mające na celu zmianę bitu na 1 lub 0 w zależności od parametru *value*:

- dla *value* równego 1 wykonywane są instrukcje zawarte w linii nr 5. Najpierw pobierana jest obecna wartość obiektu i wykonana alternatywa bitowa na niej i na wcześniej zbudowanej masce. Metoda przypisuje wynik do zmiennej *newValue* będącej buforem,
- dla *value* równego 0 metoda wykonuje instrukcje zawartą w linii nr 7. Nowa wartość uzyskiwana jest 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 }
```

Listing 4.3: Metoda int getValueOfBits(int startIndexBit, int stopIndexBit)

Listing 4.3 przedstawia metodę zwracającą fragment wartości binarnej, od pozycji określonej przez argument *startIndexBit*. do *stopIndexBit*, włącznie.

W liniach 2 i 3 sprawdzana jest poprawność argumentów i ewentualnie zostaje wyrzucony wyjątek. W linii 6 tworzony jest bufor, wykonując przesunięcie bitowe w prawo na wartości obiektu, o wartość parametru *stopIndexBit*. Dzięki tej operacji pozbyto się zbędnych bitów z wartości obiektu po prawej stronie. Następnym krokiem jest pozbycie się bitów po lewej stronie. W tym celu, w linii nr 7 wykonywana jest maska bitowa wypełniona taką liczbą jedynek, jak liczba bitów docelowej wartości. Aby ją wykonać, wykonujemy przesunięcie bitowe w lewo na maksymalnej liczbie, jaką może przyjąć typ *int* (w zapisie binarnym 01111111 11111111 11111111 11111111) o liczbę bitów wartości docelowej - 1. Następnie wykonano negację bitową. Bitów po lewej stronie pozbyto się za pomocą operacji iloczynu bitowego na uzyskanej masce i buforze.

**Inne metody**

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

- *public abstract short getSize()* - abstrakcyjna metoda zwracająca liczbę 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 największej pozycji, 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.

### 4.1.2 Implementacja klasy *XBit8*

Klasa *XBit8* dziedziczy 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 one na tyle skomplikowane, aby warto było je bardziej szczegółowo opisywać, z tego powodu zostaną wymienione i krótko opisane.

- *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*.

### 4.1.3 Implementacja klasy XBit16

*XBit16* to klasa reprezentująca liczbę 16 bitową. Dziedziczy ona po *XBit*. Dzięki temu nie musi implementować najbardziej podstawowych metod.

**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 }
```

Listing 4.4: Metoda XBit16 valueOfHighAndLowInBigEndian(XBit8 high, XBit8 low)

Listingu 4.4 przedstawia implementację metody *valueOfHighAndLowInBigEndian* tworzącej liczbę 16-bitową w formacie zapisu *big endian* (najbardziej znaczący bajt umieszczony jest jako pierwszy). Przyjmuje ona jako parametry dwie liczby 8-bitowe o nazwach *high* i *low*. Metoda tworzy obiekt klasy *ByteBuffer* należącej do standardowej biblioteki języka Java i konfiguruje ją, aby przechowywała dane w formacie *big endian* (linia nr 3). Następne linie prezentują operacje dodania do bufora kolejno starszego (linia nr 4) i młodszego bajtu (linia nr 5). Na koniec wykonano metodę *getShort()* zwracającą docelową liczbę i na jej podstawie tworzona jest reprezentacja 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 }
```

Listing 4.5: Metoda XBit16 valueOfHighAndLowInLittleEndian(XBit8 high, XBit8 low)

W listingu 4.5 zaprezentowano metodę tworzącą liczbę 16-bitową w formacie zapisu *little endian*, czyli najbardziej znaczący bajt umieszczony jest jako ostatni. Implementacja metody jest dosyć prosta, wykonuje ona metodę *valueOfHighAndLowInBigEndian()* z zamienioną kolejnością parametrów.

#### Inne metody

Klasa *XBit16* zawiera także 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 ze znakiem.

### 4.1.4 Implementacja klasy XbitUtils

*XbitUtils* to klasa implementująca operacje arytmetyczne i bitowe na obiektach *XBit8* i *XBit16*. Posiada ona dwie klasy wewnętrzne, *Arithmetic8bitResult* oraz *Arithmetic16bitResult* reprezentujące wynik operacji arytmetycznych i zawierające informacje o wystąpieniu przeniesienia i przepełnienia. Klasy te zaprezentowano w listingach 4.6 oraz 4.7.

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

Listing 4.6: Klasa Arithmetic8bitResult

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

Listing 4.7: Klasa Arithmetic16bitResult

#### public static XBit8 incrementBy(XBit8 value, int incrementer)

Listing 4.8 prezentuje metodę zwiększającą liczbę 8-bitową o daną wartość. Aby wykonać operację, metoda konwertuje parametr *value* na liczbę typu *int* (linia nr 2) i dodaje wartość parametru *incrementer* (linia nr 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

```java
public static XBit8 incrementBy(XBit8 value, int incrementer) {
        int unsignedValue = value.getUnsignedValue();
        unsignedValue += incrementer;
        unsignedValue = unsignedValue & XBit8.MAX_UNSIGNED_VALUE;
        return XBit8.valueOfUnsigned(unsignedValue);
}
```

Listing 4.8: Metoda XBit8 incrementBy(XBit8 value, int incrementer)

jest niedopuszczenie do sytuacji, w której wynik operacji 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 jej opis zostanie pominięty.

### 4.1.5 public static XBit8 negativeOf(XBit8 value)

```java
public static XBit8 negativeOf(XBit8 value) {
        int buf = (~value.getUnsignedValue()) & XBit8.MAX_UNSIGNED_VALUE;
        return XBitUtils.incrementBy(
                XBit8.valueOfUnsigned(buf),
                1
        );
}
```

Listing 4.9: Metoda XBit8 negativeOf(XBit8 value)

Funkcja tworząca liczbę o przeciwnym znaku do podanej została zaprezentowana w listingu 4.9. Operacja ta polega na wykonaniu iloczynu bitowego na negacji bitowej danej wartości oraz liczbie 255. Ma to na celu niedopuszczenie do sytuacji, w której wartość bufora będzie większa niż 255, a więc nie będzie mogła się zmieścić na ośmiu bitach. Następnie wykonany wynik zostaje zwiększony o jeden.

Zasada działania metody *XBit16 negativeOf(XBit16 value)* jest taka sama. Jedyną różnicą jest operacja iloczynu bitowego, którą wykonujemy nie na liczbie 255 a 65535.

### 4.1.6 public static Arithmetic8bitResult addTwo8bits(XBit8 value1, XBit8 value2)

Listing 4.10 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 operacje dodawania. W linii nr 4 umieszczono sprawdzenie, czy nastąpiło przepełnienie, a linia nr 5 ustawia flagę. Linia nr 7 wykonuje „obcię-

```
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 }
```

Listing 4.10: Metoda Arithmetic8bitResult addTwo8bits(XBit8 value1, XBit8 value2)

cie" bitów w przypadku, gdy wynik nie będzie w stanie zmieścić się w liczbie 8-bitowej.

Linie 9 i 10 zawierają operację ustawiającą flagę przepełnienia według trzech zasad [14]:

1. jeśli suma dwóch liczb dodatnich daje wynik ujemny, to znaczy, że wystąpiło przepełnienie,
2. jeśli suma dwóch liczb ujemnych daje wynik dodatni, to również oznaca wystąpienie przepełnienia,
3. w każdym innym przypadku przepełnienie nie wystąpiło.

Zakłada się, że najbardziej znaczący bit jest bitem znaku.

Prezentacja metody realizującej dodawanie dwóch liczb 16-bitowych została pominięta, ponieważ operacja ta jest przeprowadzana na tej samej zasadzie co 8-bitowa.

### 4.1.7 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  }
```

Listing 4.11: Metoda Arithmetic8bitResult subTwo8bits(XBit8 value1, XBit8 value2)

W listingu 4.11 zaprezentowano metodę realizującą odejmowanie jednej liczby 8-bitowej, od drugiej. Metoda wykorzystuje regułę $a - b = a + (-b)$ według której odejmowanie dwóch liczb można zastąpić dodawaniem, negując drugi składnik odejmowania. Metoda realizująca odejmowanie dwóch liczb 16-bitowych działa w analogiczny sposób.

### 4.1.8 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 }
```

Listing 4.12: Metoda XBit8 not8bit(XBit8 value)

Metoda *not8bit(XBit8 value)* zaprezentowana w listingu 4.12 wykonuje negację bitową na liczbie 8-bitowej. Operacja wykonywana jest za pomocą standardowego operatora negacji bitowej języka Java (znak tyldy górnej „~"). Dodatkową wykonywaną operacją jest iloczyn bitowy z liczbą 255, mający za zadanie wyzerowanie bitów starszych od ósmego. Wersja metody dla liczb 16-bitowych jest analogiczna.

### 4.1.9 public static XBit8 and8bit(XBit8 value1, XBit8 value2)

```
1 public static XBit8 and8bit(XBit8 value1, XBit8 value2) {
2         return XBit8.valueOfUnsigned(
3                 (value1.getUnsignedValue() & value2.getUnsignedValue())
4                 & XBit8.MAX_UNSIGNED_VALUE
5         );
6 }
```

Listing 4.13: Metoda XBit8 and8bit(XBit8 value1, XBit8 value2)

Listing 4.13 prezentuje metodę wykonującą operację sumy logicznej dwóch liczbe 8-bitowych (linia nr 3). Linia nr 4 zawiera operację wyzerowania bitów starszych od ósmego dla wyniku, aby mieścił się on w ośmiu bitach. Wersja metody dla liczb 16-bitowych działa analogicznie.

### 4.1.10 Metody wykonujące sumę bitową i różnice symetryczną

Implementacje 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)*.

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

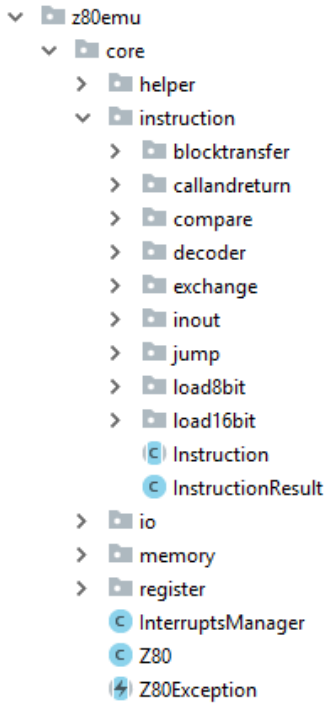
## 4.2. Z80emu-core

*Z80emu-core* to główny moduł odpowiedzialny za emulację. Korzysta on z biblioteki *XBit*. Rysunek nr 4.1 prezentuje podział kodu modułu na następujące pakiety (nazwy pakietów zostały skrócone o prefiks "org.tomaszkowalczyk94.z80emu.core." aby były bardziej czytelne):

- *helper* - zawiera klasy wspomagające wykonywanie powtarzalnych operacji,
- *instruction* - pakiet grupujący klasy implementujące rozkazy procesora; zawiera kolejne zagnieżdżone paczki mające za zadanie grupować instrukcje w kategorie np. rozkazy skoku,
- *io* - pliki odpowiedzialne za implementację symulatorów urządzeń wejścia/wyjścia podłączonych do wirtualnego CPU,
- *memory* - klasy definiujące reprezentacje pamięci procesora,
- *register* - klasy odpowiedzialne za rejestry CPU.

### 4.2.1 Metoda runOneInstruction() klasy Z80

Najważniejszą metodą w module jest *runOneInstruction()*, zaprezentowana w listingu 4.14. Pobiera ona kolejną instrukcję CPU do wykonania z pamięci (linia nr 4), następnie wykonuje ten rozkaz (linie nr 6-9) i aktualizuje liczniki (linie nr 11-16). Ważnym elementem tego podprogramu jest linia nr 1, wywołująca metodę *handleInterrupts()*, której zadaniem jest obsłużyć ewentualne przerwanie.




Rysunek 4.1: Podział na pakiety projektu Z80emu-core

### 4.2.2 Dekodowanie instrukcji

Zilog Z80 jest procesorem, który używa rozkazów o różnej długości (od 1 bajtu do 4). W związku z tym implementacja metody dekodującej instrukcje jest złożona. Nie jest możliwe przesłanie do niej jako parametr ciągu bajtów do interpretacji, ponieważ nieznana jest jego docelowa długość. Jedną z opcji byłoby nadmiarowe przesłanie zawsze czterech bajtów, ponieważ taka może być maksymalna długość rozkazu w emulowanym procesorze. Uznano jednak, że lepszym rozwiązaniem będzie przekazanie klasie dekodującej, obiektu reprezentującego pamięć. Dzięki takiemu rozwiązaniu jest możliwe odczytanie dowolnego bajtu pamięci przez klasę dekodującą. Zmniejszy to częstotliwość wywoływania metod odczytu pamięci w newralgicznej części emulatora.

W listingu 4.15 zaprezentowano fragment kodu odpowiedzialnego za dekodowanie rozkazu. Metoda *decode* przyjmuje dwa parametry:obiekt reprezentujący pamięć oraz wartość rejestru *PC*. W liniach 3-5 zamieszczone zostało dekodowanie zawartości pamięci zawierającej początkowy bajt rozkazu, który przekazano do instrukcji *switch*. Jeśli jest możliwa interpretacja rozkazu na podstawie pierwszego bajtu, zostaje zwrócony obiekt reprezentujący go tak jak w linii nr 12 lub 14. W przypadku gdy pierwszy bajt jest niewystarczający, interpretacja rozkazu zostaje oddelegowana do oddzielnych metod, które pobierają kolejną komórkę pamięci tak jak w linii nr 9.

```
1  public void runOneInstruction() throws Z80Exception {
2          interruptsManager.handleInterrupts(this);
3
4          Instruction instruction = instructionDecoder.decode(memory, registerBank.getPc());
5
6          InstructionResult result = instruction.execute(
7                  memory.read(registerBank.getPc()),
8                  this
9          );
10
11         clockCyclesCounter += result.getClocks();
12         instructionCounter++;
13
14         if(result.isAutoIncrementPc()) {
15             registerBank.incrementReg16bit(PC, result.getSize());
16         }
17 }
```

Listing 4.14: Metoda runOneInstruction()

### 4.2.3 Implementacja przykładowej instrukcji

Każdy rozkaz procesora jest reprezentowany jako osobna klasa. Przykład takiej klasy przedstawiono w listingu 4.16. Wykonuje ona instrukcje *LD A, (BC)*, która ma za zadanie wczytać do rejestru *A* wartość komórki pamięci o adresie przechowywanym w rejestrze *BC* (linie 9-15). Po wykonaniu tej operacji, metoda *execute* zwraca obiekt *InstructionResult* (linia 17), który zawiera informacje o przebiegu wykonania instrukcji, takie jak:

- liczba cykli maszynowych,
- liczba taktów zegara,
- czas wykonania w milisekundach,
- rozmiar rozkazu w bajtach (aby poprawnie zwiększyć wartość rejestru *PC*).

### 4.2.4 Przerwania

Zilog Z80 jest procesorem realizującym dwa rodzaje przerwań:

- przerwanie niemaskowalne - przerwanie, które nie może zostać wyłączone przez programistę. Wywoływane jest przez podanie sygnału na fizyczne wejście procesora NMI (ang. *non-maskable interrupt*). Po przyjęciu przerwania, procesor wykonuje restart do adresu *0066h*,

```
public Instruction decode(Memory memory, XBit16 pc)
    throws UnsupportedInstructionException, MemoryException {
        XBit8 opcode = memory.read(pc);

        int opcodeUnsignedValue = opcode.getUnsignedValue();
        XBit8 secondByte = readSecondByte(memory, pc);

        switch (opcodeUnsignedValue) {
            case 0xDD:
                return decodeDdOpcode(opcode, secondByte);
            case 0x36:
                return instructionsContainer.loadMemByHlFrom8Bit;
            case 0x0A:
                return instructionsContainer.loadAFromMemByBc;
            ...
        }
}

private Instruction decodeDdOpcode(XBit8 opcode, XBit8 secondByte)
    throws UnsupportedInstructionException {
        if(secondByte.getUnsignedValue() == 0x21) {
            return instructionsContainer.loadIxFrom16bit;
        }
        if(secondByte.getUnsignedValue() == 0x22) {
            return instructionsContainer.loadMemBy16bitFromIx;
        }
        ...
}
```

Listing 4.15: Dekodowanie rozkazu procesora

- przerwanie maskowalne - wywoływane poprzez fizyczny sygnał INT (ang. *interrupt*). Procesor może zareagować na nie na trzy sposoby:
  - tryb 0 - urządzenie przerywające umieszcza na magistrali danych instrukcje do wykonania. Ponieważ szyna danych jest 8-bitowa, rozkaz umieszczony na niej musi być jednobajtowy. Najczęściej jest to instrukcja restartu [10],
  - tryb 1 - wykonuje restart do adresu *0038h*,
  - tryb 2 - procesor wykonuje skok do adresu, którego starszy bajt jest umieszczony w rejestrze I, a młodszy dostarcza urządzenie przerywające,

Tryby może przełączać programista, używając instrukcji *IM 0*, *IM 1* i *IM 2*.

```java
public class LoadAFromMemByBc extends Instruction {
    public LoadAFromMemByBc(InstructionHelper helper) {
        super(helper);
    }

    @Override
    public InstructionResult execute(XBit8 opcode, Z80 z80) throws Z80Exception {

        XBit8 valueFromMemory = z80.getMem().read(
                z80.getRegs().getBC()
        );

        z80.getRegs().setA(
                valueFromMemory
        );

        return InstructionResult.builder()
                .machineCycles(2)
                .clocks(7)
                .executionTime(1.75f)
                .size(1)
                .build();
    }
}
```

Listing 4.16: Fragment kodu odpowiedzialnego za dekodowanie rozkazu procesora

### Przerwanie niemaskowalne

Obsługa przerwania niemaskowalnego polega na umieszczeniu na szczycie stosu aktualnej wartości rejestru *PC*. Następnie do licznika rozkazów zapisano wartość *0066h*. Fragment kodu realizujący te operacje umieszczono w metodzie *handleNmiInterrupt* zaprezentowanej w listingu 4.17

```java
private void handleNmiInterrupt(Z80 z80) throws MemoryException {
    iff2 = iff1;

    stackHelper.pushToStack(z80, z80.getRegs().getPc());
    z80.getRegs().setPc(XBit16.valueOfUnsigned(0x0066));
    nmiRequest = false;
}
```

Listing 4.17: Metoda obsługująca przerwanie niemaskowalne

### Przerwanie maskowalne

Implementacja obsługi przerwania maskowalnego pokazana na listingu 4.18, w związku z koniecznością uwzględnienia trzech sposobów reagowania na nie, jest bardziej złożona. W pierwszej kolejności odkładana jest wartość rejestru *PC* na stos (linia nr 3). Następnie sprawdzany zostaje aktywny tryb przerwania (linia nr 5), przechowywany w polu typu wyliczeniowego *interruptionMode*. W dalszej części program ulega rozgałęzieniu zależnie od aktywnego trybu przerwania:

- dla trybu nr 0 - dekodowana jest jedno-bajtowa instrukcja umieszczona na szynie danych, i wykonana (linie 7, 8),
- dla trybu nr 1 - podobnie jak w przerwaniu niemaskowalnym, wystarczy zmienić wartość rejestru *PC*, w tym przypadki na *0038h*,
- dla trybu nr 2 - wartość rejestru *PC* zostaje zmodyfikowana. Starszym bajtem jest rejestr *I*, a młodszy to wartość przesłana przez urządzenie na szynę danych (linie 15 i 16. Nowa wartość rejestru *PC* powinna być parzysta, z tego powodu zmieniono najmniej znaczący bit na *0*.

```
1  private void handleInterrupt(Z80 z80) throws Z80Exception {
2      this.iff1 = this.iff2 = false;
3      stackHelper.pushToStack(z80, z80.getRegs().getPc());
4
5      switch (interruptionMode) {
6          case IM0:
7              Instruction instruction = z80.instructionDecoder.decodeOneByte(addressBus);
8              instruction.execute(addressBus, z80);
9              break;
10         case IM1:
11             z80.getRegisterBank().setPc(XBit16.valueOfUnsigned(0x0038));
12             break;
13         case IM2:
14             XBit16 newPcValue = XBit16.valueOfHighAndLow(
15                     z80.getRegs().getI(),
16                     addressBus
17             ).setBit(0, false);
18
19             z80.getRegisterBank().setPc(newPcValue);
20             break;
21     }
22 }
```

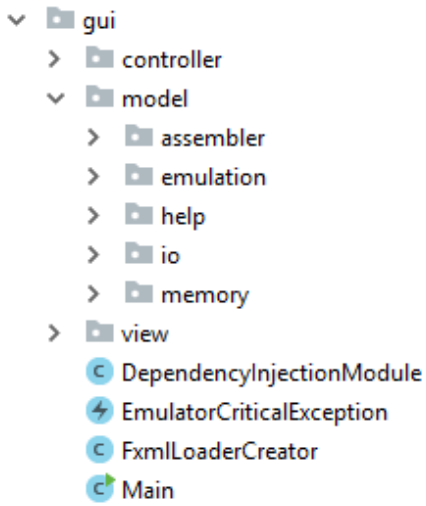
Listing 4.18: Metoda obsługująca przerwanie maskowalne

## 4.3. Z80emu-gui

Moduł *Z80emu-gui* implementuje interfejs użytkownika zaprezentowany na rysunku nr 3.5. Został on napisany z pomocą biblioteki *JavaFX*. Największą jej zaletą w porównaniu do poprzedniczki, którą była biblioteka *Swing*, jest możliwość definiowania widoku aplikacji za pomocą języków XML i CSS. Projektowanie interfejsu w *JavaFx* przypomina nieco tworzenie strony WWW.

Kod źródłowy implementujący interfejs użytkownika został podzielony na pakiety, których strukturę przedstawiono na rysunku 4.2. Postanowiono podzielić pliki zgodnie ze wzorcem *Model-View-Controller*, który dzieli aplikacje na trzy główne części. Każda z nich ma swoją reprezentację w module jako oddzielna paczka:

- *model* - implementuje logikę biznesową. W tym przypadku, między innymi zarządza procesem asemblacji i emulacji.
- *View* - opisuje widok aplikacji. W projektach zbudowanych za pomocą *javaFx* są to pliki XML i CSS. Oprócz nich mogą też być pliki języka Java, odpowiedzialne za prezentację danych. W opisywanym projekcie są to np. klasy, których zadaniem jest kolorowanie składni w edytorze kodu asemblera, tworzenie okien dialogowych lub system pomocy.
- *controller* - zadaniem kontrolera jest przetworzenie informacji od użytkownika, by na jej podstawie wywoływać metody modelu i aktualizować widok. W projekcie każda część interfejsu posiada osobny kontroler (dla przykładu, osobne kontrolery zajmują się pamięcią programu, przerwaniami, i edytorem kodu asemblera).




Rysunek 4.2: Podział na pakiety modułu z80emu-gui

### 4.3.1 Wstrzykiwanie zależności za pomocą biblioteki *Guice*

Wstrzykiwanie zależności (z ang. *Dependency Injection*) to technika programowania, której głównym założeniem jest przekazywanie gotowych skonfigurowanych już obiektów do innych (wstrzykiwanie ich), które ich wymagają. Moduł interfejsu użytkownika posiada wiele współpracujących ze sobą klas. Zarządzanie nimi stało się z czasem uciążliwe. Problem postanowiono rozwiązać z pomocą biblioteki *Guice*, która implementuje technikę wstrzykiwania zależności.

Aby opisać problem, należy przedstawić, z jakich elementów zostały zbudowane kontrolery w projekcie. Fragment jednego z nich został zaprezentowany w listingu 4.19. Kontroler ten jest zależny od modelu (linie nr 3-5) oraz posiada obiekty wstrzykiwane przez bibliotekę graficzną na podstawie plików FXML (linie nr 7 i 8). Zatem należało użyć biblioteki *Guice* nie wprowadzając przy tym konfliktów z *JavaFX*.

```
1  public class MemoryController implements Initializable {
2
3      @Inject private ValueFormatter valueFormatter;
4      @Inject private DialogHelper dialogHelper;
5      @Inject private Z80 z80;
6
7      @FXML public TableView<MemoryRowModel> memoryTable;
8      @FXML public TableColumn<MemoryRowModel, String> memoryColumnAddress;
9      ...
10 }
```

Listing 4.19: Zależności klasy *MemoryController*

Połączono *JavaFx* i *Guice* w następujący sposób. Interfejs biblioteki graficznej posiada metodę *setControllerFactory(Callback<Class<?>, Object> controllerFactory)*, potrafiącą podmienić domyślną fabrykę kontrolerów. Jako jej parametr przekazano metodę biblioteki *Guice <T> T getInstance(Class<T> type)*, która zwraca instancję danego obiektu. Dzięki takiemu rozwiązaniu, kontrolery mają możliwość używania wszystkich adnotacji ułatwiających wstrzykiwanie do nich zależności (a nawet innych kontrolerów) i jednocześnie są udekorowane w obiekty biblioteki graficznej.

### 4.3.2 Integracja z projektem Z80emu-core

*Z80emu-gui* używa do emulacji modułu *Z80emu-core*, który zawiera metodę *runOneInstruction* wykonującą kolejną instrukcję CPU. Pomiędzy jej kolejnymi wywołaniami powinny zostać zgłoszone przerwania lub odczytane wartości rejestrów, pamięci i flag. Aby umożliwić to zarówno w trybie ciągłym, jak i krokowym wymagane było umieszczenie emulacji w osobnym wątku.

W tym celu wykonano klasę *EmulatorThread*, którą pokazano w listingu 4.20. Użytkownik może uruchomić i zatrzymać emulację w dowolnym momencie za pomocą publicznych metod *pause* i *unPause*.

Metoda *pause*, która docelowo zostaje wywoływana z innego wątku, zmienia wartość pola *pause* typu *boolean* na *false*. W głównej pętli emulacji sprawdzany jest warunek *if(pause) {lock.wait(); pause = false;}*. Jeśli jest on spełniony, to wątek jest wprowadzany w stan oczekiwania.

Metoda *unPause*, która także wywoływana jest z innego wątku, odblokowuje wątek, i pozwala wznowić emulację.

Implementacja przebiegła zgodnie z założonym projektem aplikacji. Stosunkowo najbardziej czasochłonną częścią implementacji, było opracowanie klas modułu *Z80emucore*, odpowiedzialnych za realizowanie rozkazów procesora. Spowodowane to było głównie ich ilością (158 instrukcji). Moduł *XBit* spełnił swoje zadanie i znacznie uprościł kod źródłowy emulatora.

```java
public class EmulatorThread extends Thread {

    ...

    private Object lock = new Object();
    boolean pause = false;

    public void pause() {
        pause = true;
    }

    public void unPause() {
        synchronized(lock) {
            lock.notify();
        }

    }

    private void pauseEmulationIfPausedFlag() throws InterruptedException {
        synchronized(lock) {
            if(pause) {
                lock.wait();
                pause = false;
            }
        }
    }

    @Override
    public void run() {
        try {
            pause = true;
            while (true) {
                pauseEmulationIfPausedFlag();
                makeInterruptsRequests();
                z80.runOneInstruction();
                refreshGui();
            }
        } catch (Exception e) {
            dialogHelper.displayError("emulation error", e);
            debuggerController.refreshCyclicButtonText();
        }
    }

    ...
}
```

Listing 4.20: Klasa *EmulatorThread* realizująca emulację w trybie ciągłym

# 5. Testy

Ten rozdział zawiera opis metodologii przeprowadzonych testów oraz komentuje ich wyniki. Nawet niewielkie defekty w kodzie emulatora mogą spowodować błędne działanie programów przeznaczonych dla procesora Zilog Z80 lub wręcz uniemożliwić ich wykonanie. Dlatego dla każdej metody implementującej rozkaz procesora został opracowany co najmniej jeden test jednostkowy. Testy jednostkowe zaimplementowane przy pomocy biblioteki JUnit. Dodatkowo wykonano także testy manualne, polegające na wykonaniu programów na opracowanym emulatorze i *Z80 Simulator IDE*, a następnie porównaniu ich wyników.

## 5.1. Testy jednostkowe

Testy jednostkowe to programy, które mają za zadanie weryfikować działanie określonego fragmentu testowanego oprogramowania. W omawianej aplikacji zostały użyte celem sprawdzenia czy rozkazy procesora są poprawnie emulowane.

Kod testujący powstał przed napisaniem implementacji danej funkcji. Takie podejście nazywane jest *Test-driven development*. Programista najpierw określa wymagania, jakie powinna spełniać dana metoda, a następnie pisze jej implementacje. Pozwala to na opracowanie oprogramowania o lepszej jakości, ponieważ informatyk pisząc testy, ustala wymagania dla danego fragmentu kodu, które potem stara się spełnić[12].

```
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 {
    prepareZ80(XBit8.valueOfSigned(-40));
    z80.runOneInstruction();
    Assert.assertEquals(true, z80.getRegs().getFlag(Flag.S));
}

@Test
public void testFlags2() throws Exception {
    prepareZ80(XBit8.valueOfSigned(0));
    z80.runOneInstruction();
    Assert.assertEquals(false, z80.getRegs().getFlag(Flag.S));
    Assert.assertEquals(true, z80.getRegs().getFlag(Flag.Z));
}
```

Listing 5.1: Test jednostkowy instrukcji LoadAFromITest.java

Na listingu 5.1 zaprezentowano test instrukcji wczytującej zawartość rejestru *I* do rejestru *A* (rozkaz *LD A, I*). Przedstawiony przykład pokazuje, że prosta z pozoru operacja, jaką jest pobranie wartości jednego rejestru i przeniesienie go do innego, wymaga złożonych testów. Test sprawdza poprawność wartości umieszczonej w rejestrze docelowym, flag procesora, licznika cykli i rejestru *PC* za pomocą metody *assertEquals*.

## 5.2. Testy manualne

Testy manualne polegają na weryfikowaniu poprawności działania oprogramowania, przez człowieka, bez udziału narzędzi automatyzujących. Przeprowadza się je między innymi na poziomie interfejsu użytkownika aplikacji. Wykonano emulację przykładowych programów napisanych w asemblerze na opracowanym oprogramowaniu i emulatorze *Z80 Simulator IDE*, a następnie porównano wyniki działania w postaci stanu rejestrów, flag i pamięci. Kody źródłowe programów zastosowanych do testów, zaczerpnięto z książki Rodney'a Zaks'a *Programming the Z80*[8].

Listing 5.2 zawiera program użyty w jednym z testów manualnych, wskazujący w tablicy liczb, element o największej wartości. Wymaga on wprowadzenia do pamięci, począwszy od adresu *30h* tablicy, której pierwszy bajt zawiera liczbę przechowywanych elementów. Kod programu wprowadzono zarówno do pamięci stworzonego w ramach pracy emulatora, jak i *Z80 Simulator IDE*, razem z tablicą elementów: *03h*, *10h*, *40h*, *20h*.

```
MAX             LD HL, 30h
        LD B, (HL)
                LD A, 0
                INC HL
                LD (50h), HL
LOOP    CP (HL)
                JR NC, NOSWITCH
                LD A, (HL)
                LD (50h), HL
NOSWITCH INC HL
                DEC B
                JR NZ, LOOP
                RET
```

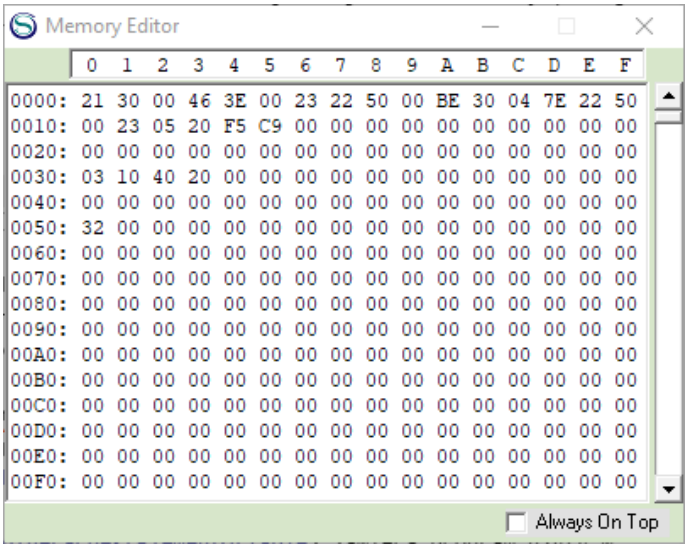
Listing 5.2: Program wskazujący największą wartość w tablicy

Zrzuty ekranów 5.1 i 5.2 prezentują pamięć obu emulatorów po wykonaniu programu. Komórki o adresach *50h-51h* przechowują adres, pod którym znajduje się znaleziona największa wartość w tablicy. Oba emulatory wskazały na liczbę 40, więc poprawnie wykonały program.

Pamięć

| adres | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0000 | 21 | 30 | 00 | 46 | 3E | 00 | 23 | 22 | 50 | 00 | BE | 30 | 04 | 7E | 22 | 50 |
| 0010 | 00 | 23 | 05 | 20 | F5 | C9 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0020 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0030 | 03 | 10 | 40 | 20 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0040 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0050 | 32 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |

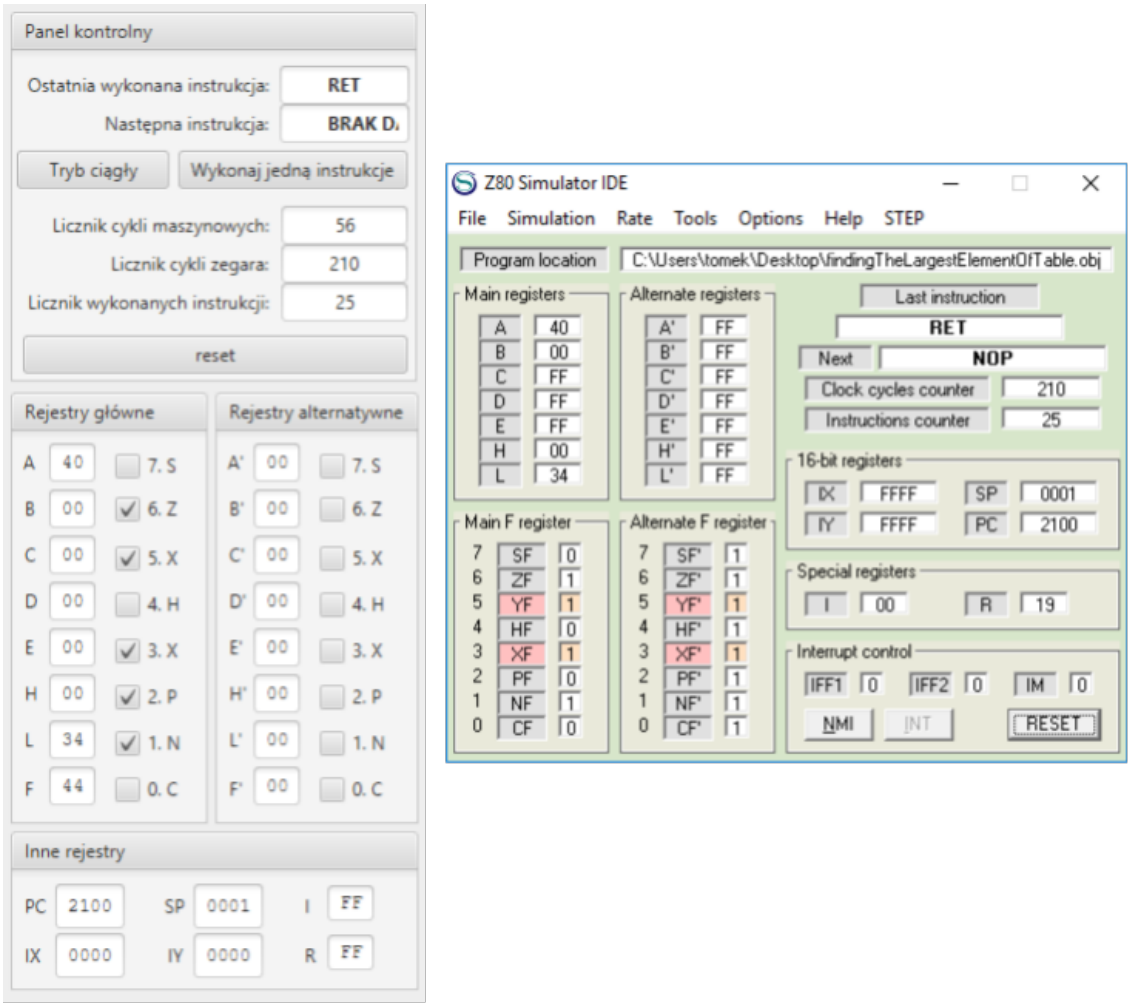
Rysunek 5.1: Zrzut ekranu aplikacji *Z80Emu* (opracowany emulator) prezentujący pamięć procesora po wykonaniu testowego programu




Rysunek 5.2: Zrzut ekranu aplikacji *Z80 Simulator IDE* prezentujący pamięć procesora po wykonaniu testowego programu

Zrzuty ekranu 5.3 przedstawiają stan rejestrów, flag i liczników po wykonaniu programu. Można zauważyć, że niektóre rejestry w emulatorze *Z80Emu* przyjmują wartość *00h*, natomiast w *Z80 Simulator IDE* wartość *FFh*. Są to rejestry, na które testowy program nie wpływa. Nie jest to błąd. Domyślnie rejestry w Zilog-u Z80 mają wartość nieokreśloną i emulatory mogą przyjąć dowolną. Oprócz nich wszystkie wartości rejestrów, flag oraz liczników zgadzają się między obydwoma aplikacjami.

Testowanie aplikacji było czasochłonnym i ważnym etapem przez specyfikę projektu. Emulator to program wrażliwy na błędy, i wymagane jest dokładne zweryfikowanie działania wszystkich emulowanych rozkazów procesora.




Rysunek 5.3: Zrzut ekranu aplikacji *Z80Emu* (po lewej) i *Z80 Simulator IDE* (po prawej) prezentujący rejestry procesora po wykonaniu testowego programu

# 6. Uwagi i wnioski

Wynikiem realizacji pracy jest aplikacja pozwalająca na emulację procesora Zilog Z80. Z celów zawartych w zadaniu na pracę dyplomową zrealizowano wszystkie, oprócz emulacji wewnętrznych magistrali procesora. Problem okazał się zbyt złożony i czasochłonny bez dostępu do dokumentacji opisującej wewnętrzną budowę procesora, której firma *Zilog* nie upublicznia.

Opisany w pracy emulator spośród innych podobnych rozwiązań wyróżnia się możliwością uruchomienia na wielu platformach systemowych, dzięki użyciu języka Java oraz czytelnym kodem źródłowym podzielonym na moduły, z których każdy może zostać ponownie użyty w innym projekcie.

Podczas pracy nad aplikacją napotkano dwa główne problemy. Jednym z nich, jest brak w języku Java typu prostego przeznaczonego wyłącznie dla liczb naturalnych (ang. *unsigned*). Rozwiązano go tworząc własną implementację liczb w bibliotece Xbit. Drugim, poważniejszym problemem okazała się wrażliwość procesu emulacji na błędy. Nawet niewielkie nieprawidłowości w wykonywanych instrukcjach procesora, powodują nieprawidłowe działanie emulowanego programu. Ten problem rozwiązano testami jednostkowymi opracowanymi dla wszystkich instrukcji procesora.

Każdy moduł aplikacji można rozwijać niezależnie. Wydajność biblioteki *XBit* może zostać zwiększona poprzez zmianę kontenera przechowującego wartość liczby, który jest typu *int*, na obiekt klasy *ByteBuffer* która jest standardowym elementem języka Java.

W module *Z80emu-core* można zaimplementować *debugger*, który dla trybu ciągłego emulacji zatrzymywałby wykonywany program w określonym miejscu.

W *Z80emu-gui* elementem, który mógłby zostać w przyszłości zaimplementowany i usprawniłby obsługę programu przez osoby dopiero uczące się zasad działania procesorów, jest wbudowany w aplikację system pomocy. Zawierałby on informacje o rozkazach, rejestrach, przerwaniach i zasadach działania procesora, które można uzyskać z ogólnodostępnych źródeł. Użytkownik mógłby kliknąć tekst z podpisem konkretnego elementu interfejsu emulatora, o którym chciałby się dowiedzieć więcej, a aplikacja przekierowałaby go do odpowiedniej informacji w systemie pomocy. Dodatkowo wersja anglojęzyczna interfejsu pozwoliłaby na dotarcie do większej liczby użytkowników.

## Bibliografia


[1] Boris D., *How Do I Write an Emulator?* (online), https://www.atarihq.com/danb/files/emu_vol1.txt (10 grudnia 2018).

[2] Bourakis A., *What's the difference between simulation and emulation* (online), http://bourakis.com/whats-the-difference-between-simulation-and-emulation/ (28 stycznia 2018).

[3] Fayzullin M., *How To Write a Computer Emulator* (online), http://fms.komkon.org/EMUL8/HOWTO.html (04 grudnia 2018).

[4] Goetz B., *Dynamic compilation and performance measurement* (online), https://www.ibm.com/developerworks/library/j-jtp12214/index.html (12 grudnia 2018).

[5] Johnson H. R., *How to start with CP/M* (online), http://www.retrotechnology.com/dri/howto_cpm.html (02 stycznia 2019).

[6] Karczmarczuk J., *Mikroprocesor Z80*, Wydawnictwa Naukowo-Techniczne, Warszawa 1987.

[7] Moya del Barrio V., *Study of the techniques for emulation programming* (online), http://www.xsim.com/papers/Bario.2001.emubook.pdf (4 grudnia 2018).

[8] Zaks R., *Programming the Z80, 3rd Edition*, Sybex, Berkeley 1981.

[9] *Difference between Simulator and Emulator* (online), https://difference.guru/difference-between-simulator-and-emulator (4 grudnia 2018).

[10] Dokumentacja techniczna Zilog Z80 CPU, 2016.

[11] Instrukcja obsługi ZIM (online), http://www.natmac.net/zim/manual/index.html (03 stycznia 2019).

[12] *Test Driven Development* (online), https://msdn.microsoft.com/pl-pl/library/test-driven-development.aspx (16 stycznia 2019).

[13] *The University of Queensland Binary Translator (UQBT) Framework* (online), https://www.researchgate.net/publication/239665973_The_university_of_queensland_binary_translator_uqbt_framework (10 grudnia 2018).

[14] *Two's Complement Overflow Rules* (online), http://sandbox.mc.edu/ bennet/cs110/tc/orules.html (04 stycznia 2019).

[15] *Z80 Simulator IDE* (online), http://www.oshonsoft.com/z80.html (28 stycznia 2018).