Emulator procesora Zilog Z80

Tomasz Kowalczyk

31 stycznia 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.

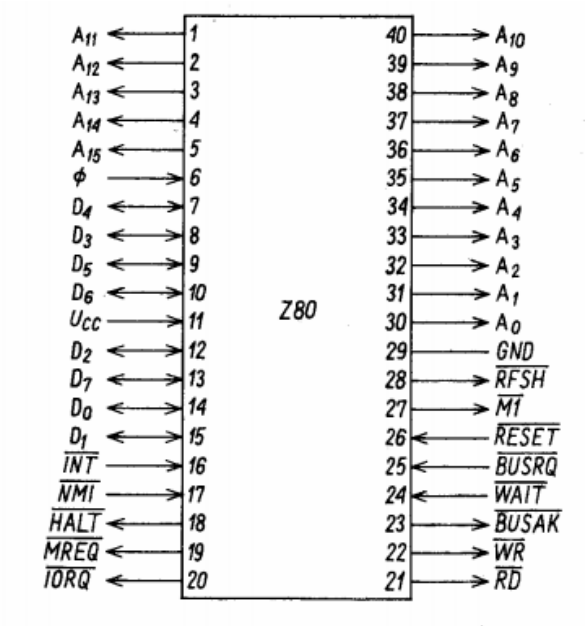
Procesor Zilog z80 był bardzo popularny na rynku mikroprocesorów[3]. Ukazał się w roku 1976, i szybko zdominował rynek 8-bitowych procesorów.

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:

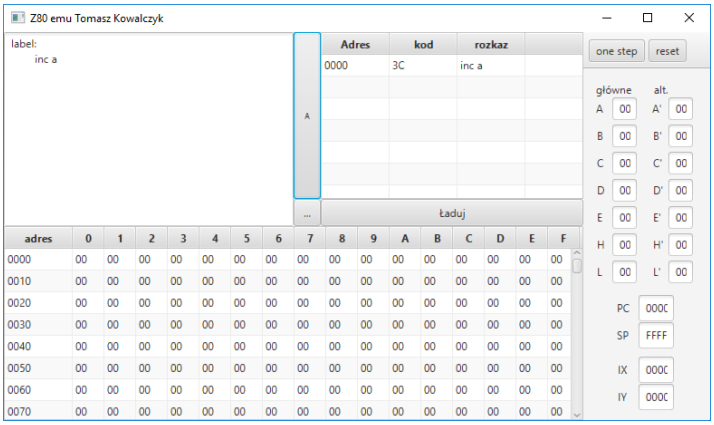
- widok kodu programu napisany w języku asembler, wraz z wynikowym kodem maszynowym
- widok pamięci w formie tabeli. Adres komórki uzyskuje się przez dodanie do siebie wartości z nazwy kolumny z wartością w nazwie wiersza. Edycje wykonuje się przez dwukrotne kliknięcie w komórkę tabeli, wpisaniu nowej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

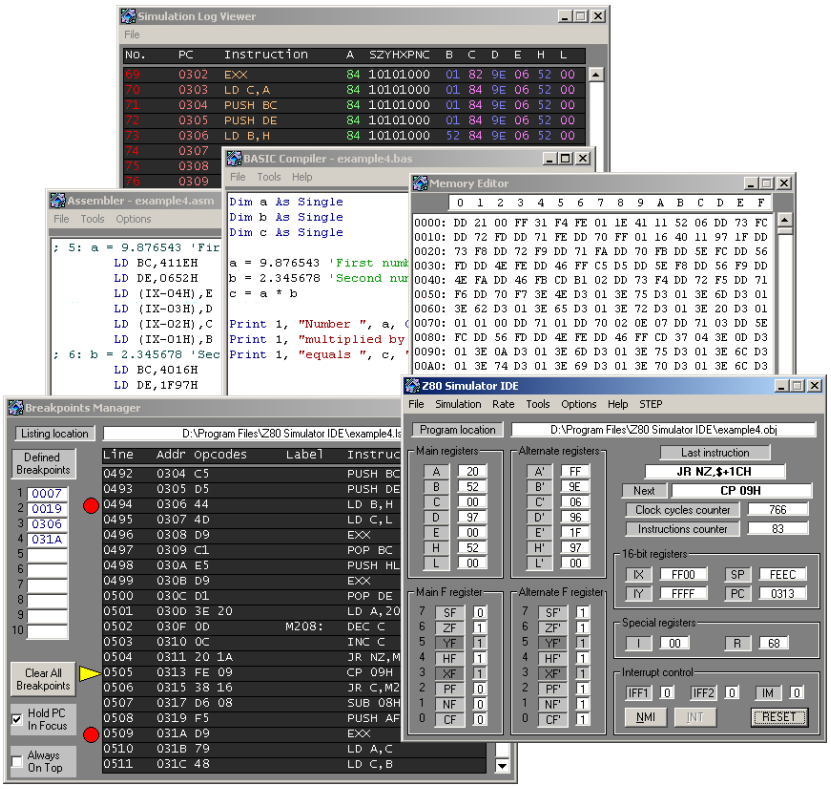
Zilog Z80 stał popularnym obiektem emulacji dzięki swojej popularności. Aplikacje te, pisano zarówno przez duże firmy, jak i hobbystów. Na jednej z usług hostingowych o nazwie „Github" która jest przeznaczona dla projektów programistycznych, można znaleźć około 200 repozytoriów z projektami emulującymi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

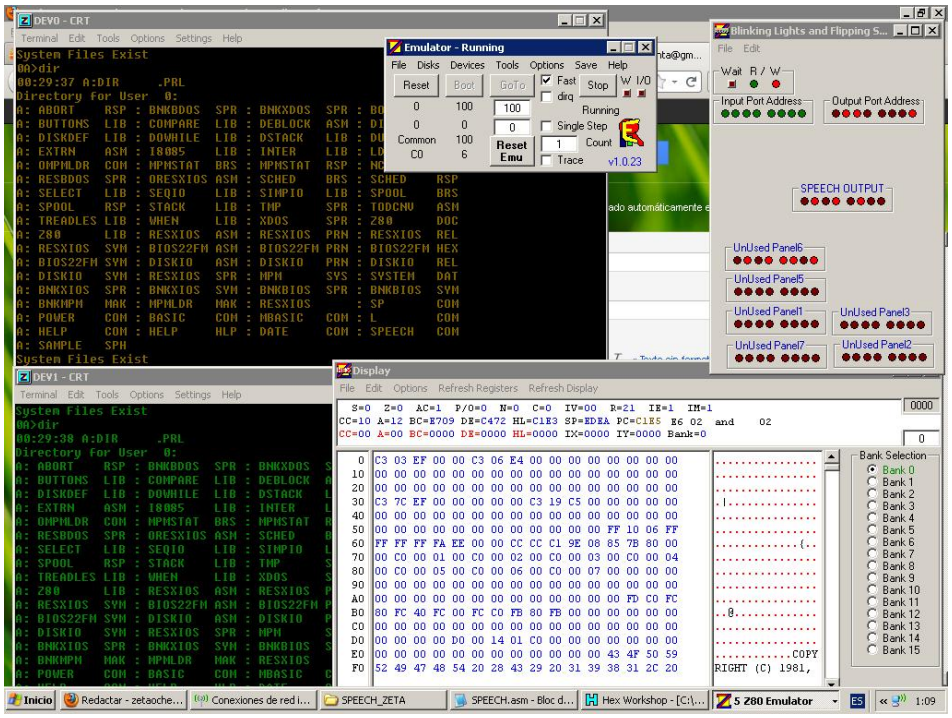
3.2 ZEMU - Z80 Emulator Joe Moore

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

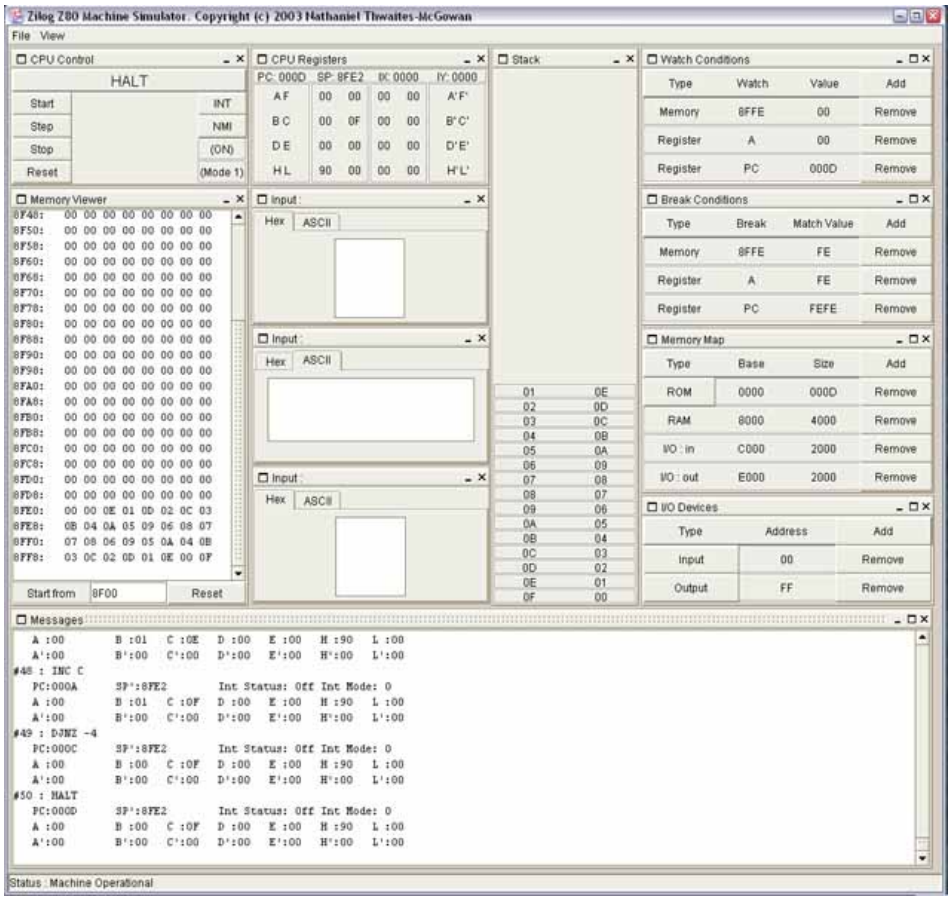
ZIM został napisany za pomocą technologi Java Web-Start. Pozwala to na uruchomienie aplikacji bezpośrednio na stronie internetowej, wraz z dostępem do lokalnych zasobów systemu, np plików[12]. Aplikacja według autora przeznaczona jest głównie dla studentów uczących się języka asemblera dla procesora 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 4.1. 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 4.2

```
<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 4.1: fragment pliku pom.xml umożliwający integracje z serwisem myMavenRepo

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

Kod 4.2: fragment pliku settings.xml przechowującego dane utoryzacyjne do serwera myMavenRepo

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

4.1.1 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.

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.

4.1.2 z80emu-core

4.1.3 z80emu-gui

Rozdział 5

Implementacja

fragmenty kodu, podział na pakiety

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:

```
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().getUnsignedV
        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));
    }

}
```

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

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