

# Emulator procesora Zilog Z80

Tomasz Kowalczyk

14 stycznia 2019

## Spis treści

<b>1</b>	<b>Wstęp</b>	<b>1</b>
1.1	Cel pracy . . . . .	1
1.2	aplikacja . . . . .	2
<b>2</b>	<b>Zagadnienie emulacji</b>	<b>2</b>
<b>3</b>	<b>Przegląd istniejących rozwiązań</b>	<b>2</b>
<b>4</b>	<b>Projekt aplikacji</b>	<b>3</b>
<b>5</b>	<b>Implementacja</b>	<b>3</b>
<b>6</b>	<b>Testy</b>	<b>3</b>
<b>7</b>	<b>Uwagi i wnioski</b>	<b>5</b>

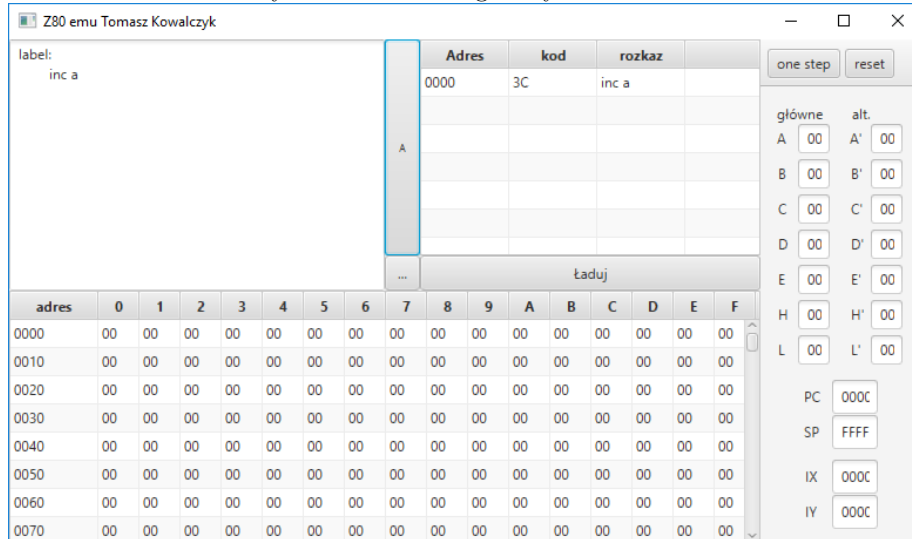
## 1 Wstęp

### 1.1 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 wewnętrznych procesora, jak również powinna istnieć możliwość podglądu i zmiany zawartości pamięci programu. Aplikacja powinna być zaimplementowana w języku Java

Z wyżej 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.

Rysunek 1: Widok główny emulatora



## 1.2 aplikacja

Interfejs użytkownika podzieliłem na 3 części.

- Widok kodu programu napisany w języku assembler, wraz z wynikowym kodem maszynowym
- Widok pamięci w formie tabeli. Aby uzyskać adres odpowiadający danej komórce, należy dodać do siebie wartość ????. Edycje wykonujemy przez dwukrotne kliknięcie na komórce, wpisaniu nowej wartości i zatwierdzeniu klawiszem Enter.
- Widok stanu wewnętrznych rejestrów procesora, wraz z przyciskami debugującymi.

W aplikacji każda wyświetlona wartość jest w systemie heksadecymalnym, i także w takiej notacji wprowadzamy wartości (oprócz pola do edycji kodu assemblera, gdzie możemy używać innych notacji).

## 2 Zagadnienie emulacji

Treść

## 3 Przegląd istniejących rozwiązań

Żadne istniejące rozwiązanie nie pozwala na podejrzenie wewnętrznych magistrali procesora

## 4 Projekt aplikacji

????

## 5 Implementacja

Treść

## 6 Testy

### 6.1 ????

Bardzo ważną kwestią w projekcie było dokładne pokrycie kodu aplikacji w testach jednostkowych. Emulator mikro-kontrolera to specyficzna aplikacja, w której nawet mały, wydawałoby się 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 dla mojego projektu. 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().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));
    }
}

```

Jak widać tak prosta operacja jak skopiowanie wartości z jednego rejestru i przeniesienie go do innego wymaga dosyć objętościowych testów. Oprócz testowania czy poprawna wartość znajduje się w rejestrze docelowym musimy sprawdzić także czy flagi CPU zostały ustawione na poprawne wartości, 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

## 7 Uwagi i wnioski

Treść