

Emulator procesora Zilog Z80

Tomasz Kowalczyk

16 stycznia 2019

Spis treści

1	Wstęp	2
2	Zagadnienie emulacji	5
2.1	emulacja przez interpretowanie	5
2.2	emulacja przez statyczną re-kompilację	6
2.3	emulacja przez dynamiczną re-kompilację	6
2.4	różnica między symulacją a emulacją	6
3	Przegląd istniejących rozwiązań	8
3.1	Motywacja	8
4	Projekt aplikacji	9
5	Implementacja	10
6	Testy	11
6.1	????	11
6.2	Test-driven development	13
7	Uwagi i wnioski	14

Rozdział 1

Wstęp

Celem pracy jest wykonanie emulatora procesora Zilog Z80. Aplikacja umożliwia wczytanie programu w postaci kodu maszynowego, deasemblację i wykonanie. Dostępne są dwa tryby wykonania: ciągły i krokowy. W obu przypadkach emulator obrazuje stan rejestrów, jak również umożliwia podgląd i zmianę zawartości pamięci programu. Aplikacja została zaimplementowana w języku Java.

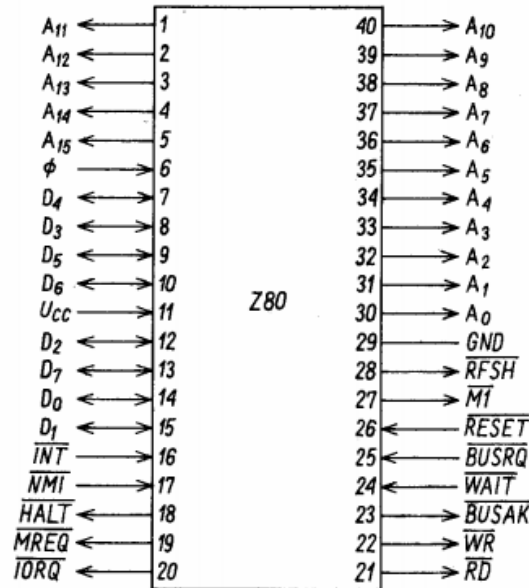
Procesor Zilog z80 był szlagierem rynku mikroprocesorowego. [3] Został wydany na rynek w roku 1976, i szybko zdominował rynek 8-bitowych procesorów.

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

Urządzenie to mimo zalet, ma 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 aplikację wykonano w języku Java 8 i biblioteki graficznej Java FX. Interfejs użytkownika został podzielony na 3 części:

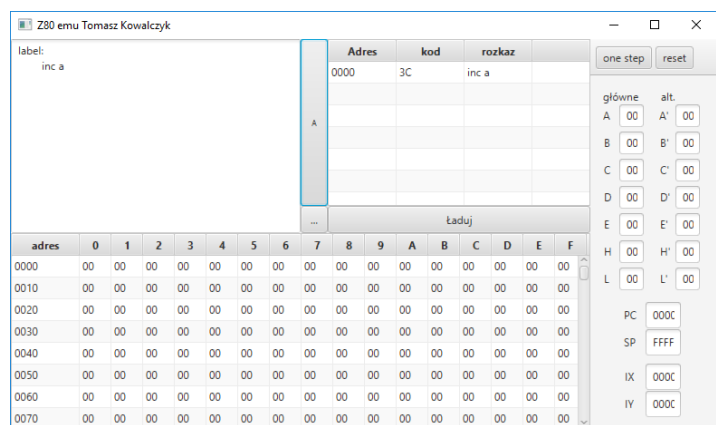
- widok kodu programu napisany w języku assembler, wraz z wynikowym kodem maszynowym
- widok pamięci w formie tabeli. Aby uzyskać adres odpowiadający danej komórce, należy dodać do siebie wartość ????. Edycje wykonujemy przez dwukrotne kliknięcie w komórkę tabeli, wpisaniu nowej wartości i zatwierdzeniu 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 wprowadzamy wartości (oprócz pola do edycji kodu assemblera, gdzie możemy 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 definicją 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-kompilację
- emulacja przez dynamiczną re-kompilację

Każda z tych metod wymaga oddzielnego omówienia.

2.1 emulacja przez interpretowanie

Interpreter to najprostszy rodzaj emulatora. Odczytuje on 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 emulacja przez statyczną re-kompilację

2.3 emulacja przez dynamiczną re-kompilację

2.4 różnica między symulacją a emulacją

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

W książce "Study of the techniques for emulation programming" Victor Moya del Barrio podaje taką to definicję 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

Żadne istniejące rozwiązanie nie pozwala na podejrzenie wewnętrznych magistrali procesora

3.1 Motywacja

dsadsadsa

Rozdział 4

Projekt aplikacji

?????

Rozdział 5

Implementacja

Treść

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 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));
}
}

```

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