SYMULATOR MIKROPROCESORA INTEL 8086

Działanie elementarnych operacji na rejestrach i pamięci

Jakub Wieroński
wierjakub@gmail.com

Spis treści

Symulator mikroprocesora INTEL 8086

1. Opis problemu

Napisać program symulujący działanie mikroprocesora INTEL 8086 w dowolnym języku programowania. Symulator powinien uwzględniać obsługę elementarnych komend assemblera MOV, XCHG, PUSH i POP. Program powinien posiadać obsługę w postaci trybu graficznego.

2. Analiza

Symulator został stworzony przy użyciu języka C# przy docelowej strukturze .NET Core 3.1, a także z wykorzystaniem silnika graficznego WPF.

Ze względu na budowę wewnętrzną mikroprocesora, najbardziej znaczącymi punktami programu stają się:

- Reprezentacja działania pamięci i rejestrów.
- Prezentacja zmian wprowadzanych w rejestrach i pamięci w trybie graficznym.
- System łączący akcje użytkownika z operacjami opartymi na języku assemblera (MOV, XCHG, PUSH, POP).

Charakterystyka problemu jest zbliżona do struktury wzorca architektonicznego MVC, co znacznie ułatwia projektowanie samej struktury symulatora i relacji łączących jego obiekty. Na wybór struktury typu MVC wpłynęła także znajomość wzorców projektowych dobrze współpracujących z tym wzorcem, w szczególności np. *Observer* a także zaznajomienie i chęć programowania zgodnie z zasadami SOLID.

W pierwszej kolejności zadaniem było zaprojektowanie ogólnej zasady działania symulatora. Rozważałem więc możliwość posłużenia się wzorcem projektowym *Command*, do wykonywania komend wprowadzonych przez użytkownika. Doszedłem jednak do wniosku, że wzorzec ten nie rozwiązywałby problemu samej interpretacji linii komend. Zmusiłoby mnie do zimplementowania również jakiegoś rodzaju fabryki, tworzącej *polecenia* na podstawie danych. Uznałem więc, że dobrym rozwiązaniem będzie przepuszczenie komend przez obiekty mogące je obsłużyć i dać dwa rezultaty. Pierwszy, to analiza operacji i adekwatne działania.

Drugi, z kolei to zwrócenie rezultatu do interfejsu użytkownika, tutaj "output". Ostatecznie do interpretacji komend wprowadzanych, czyli *kontrolera*, wykorzystałem *Chain of Responsibility*, przesyłając zbiór danych przez każde ogniwo, starające się je obsłużyć, a następnie zwracające z powrotem log z takiej operacji.

Ze względu na wiele specyficznych przypadków użycia wynikających ze specyfikacji zadania praca nad kodem źródłowym symulatora odbywała się symultanicznie z tworzeniem testów jednostkowych na zasadzie techniki TDD.

2.1. Analiza, podstawy teoretyczne

Mikroprocesor INTEL 8086 posiada w swojej specyfikacji magistralę pamięci o szerokości dwudziestu pinów, co pozwala na zaadresowanie maksymalnie do 2^{20} komórek pamięci, każda w postaci jednego bajta. Zatem mikroprocesor może operować na 1MB pamięci posługując się *adresem fizycznym*, a także zarówno jednobajtowymi, jak i dwubajtowymi *słowami*.

Jednakże, każde odwołanie do pamięci może odbyć się jedynie za pośrednictwem wyrażeń 16 bitowych. Tworzy to sytuacje w której *słowo* nie może obsłużyć – szerszej od siebie – magistrali 20 bitowej. Z tego wynika, że na każdy adres w postaci 16 bitowej przypada nam jeszcze szesnaście komórek pamięci. Toteż, aby się do nich odwołać, musimy skorzystać z *przesunięcia*.

Przesunięcie to wartość 16 bitowa, która według konwencji nie powinna przekraczać wartości 4 bitowych. Dzięki niemu możemy obliczany jest adres fizyczny. Zawartość rejestru przechowującego wartość początku segmentu danych (*DS)* przesuwana jest o cztery bity w lewą stronę, tworząc ciąg 20 bitowy. Do niego dodawane jest przesunięcie.

Do dyspozycji posiada również rejestry, każdy 16 bitowy.

Rejestry ogólnego przeznaczenia;

- AX, AH, AL – Akumlator.
- BX, BH, BL – Baza.
- CX, CH, CL – Licznik.
- DX, DH, DL – Dane.

Każdy z nich może działać niezależnie jako rejestr 8 bitowy. Przyrostek "H" oznacza pierwszą część bitów bardziej znaczących (MSB), z kolei "L" część bitów mniej znaczącą w całym słowie.

Rejestry indeksowe oraz wskaźnikowe

- SI – rejestr indeksowy źródła.
- DI – rejestr indeksowy przeznaczenia.
- SP – wskaźnik stosu.
- BP – wskaźnik bazy.

Służące głównie do wskazywania miejsc w pamięci, pod którymi znajdują się argumenty rozkazu. Są także wykorzystywane do adresowania indeksowego, bazowego i indeksowo-bazowego. Mogą również przechowywać argumenty lub wyniki operacji.

Rejestry segmentowe

- CS – rejestr segmentowy programu.
- SS – rejestr segmentowy danych.
- DS – rejestr segmentowy stosu.
- ES – rejestr segmentowy dodatkowy.

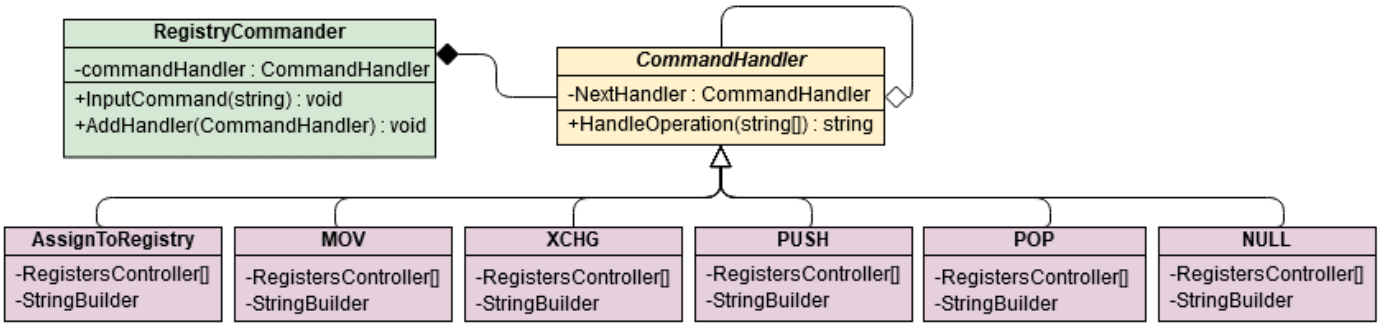
Zawartości tych rejestrów służą do obliczania fizycznego adresu komórki pamięci, gdyż zawierają one adres początku danego segmentu pamięci. W zależności do którego rodzaju pamięci użytkownik chce się odwołać, mikroprocesor wykorzysta odpowiedni z rejestrów.

2.2. Analiza obiektowa kluczowych struktur

Interpretacja wprowadzonych komend – wzorzec *Chain of Responsibility*.

Z perspektywy użytkownika najważniejszym elementem jest wprowadzenie i interpretacja linii danych na adekwatne komendy. System wprowadzonych argumentów został stworzony na zasadzie wzorca projektowego *Chain of Responsibility*.

Komenda wprowadzona przez użytkownika przeprowadzana jest przez listę wzorowaną na *Singly linked list*. Każdy element listy *ma* odwołanie do swojego następcy, zatem funkcja *HandleOperation,* może wywoływać się na swoich następcach dopóki argumenty[1] nie zostaną obsłużone, bądź nie dotrze do końca. Jeżeli wprowadzone dane nie zostaną zinterpretowane, na końcu listy znajduje się domyślnie obiekt reprezentujący wzorzec *Null object*[2], zwracający komunikat o niewspieranej komendzie. Metoda *HandleOperation* zwraca string, który przekazywany jest na sam początek wywołania łańcucha, a używany jest do wyświetlenia komunikatów powstałych podczas interpretacji.



Warto nadmienić, że zastosowanie takiej struktury daje benefity w postaci możliwości manipulowania kolejnością lub egzystencją obiektów obsługi żądania. Mogą też one posiadać tablicę obsługiwanych rejestrów i klasę StringBuilder wspierającą łączenie poszczególnych logów z procesu przetwarzania komendy. Zmniejsza też zależności między nadawcą a odbiorcą i pozwala na to, by implementacja pojedynczej procedury nie wymuszała znajomości całego systemu. Ta część programu jest kluczowa przez wzgląd, że jest głównym kontrolerem[3] całego symulatora.

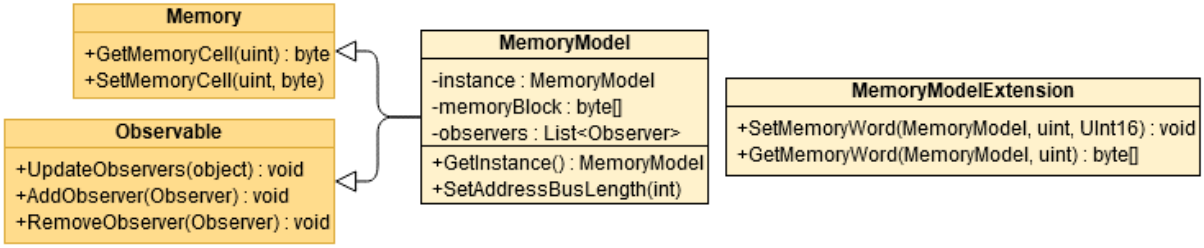
Realizacja pamięci mikroprocesora – wzorzec *Singleton* i *Observer*.

Fizyczne adresowanie pamięci mikroprocesora bierze się z jego dwudziestu pinów. Ze względu na to, że nie przewidujemy, aby zmieniły one swoją ilość, to zakładamy również, że maksymalna obsługiwana ilość komórek pamięci nie zmieni się. Nie możemy też dopuścić do tego, by w symulatorze istniały zupełnie różne instancje pamięci.

Klasa *MemoryModel*, operująca na tablicy bajtów o długości 2^{20}, może posiadać tylko i wyłącznie pojedynczą instancję, której referencje programista może pobrać za pomocą

[1] Argumenty mają postać string[], według konwencji zaczerpniętej ze standardowych funkcji main(string[] args).
[2] Częściowo niwelujemy tym jedną z wad wzorca CoR, czyli możliwość, że żadne ogniwo nie obsłuży żądania.
[3] Model-View-Controller

statycznej metody *GetInstance* tej klasy. W dodatku klasa posiada pole określające szerokość magistrali pamięci, dzięki czemu możemy wykorzystać ją także w oparciu o odmienne specyfikacje urządzeń.



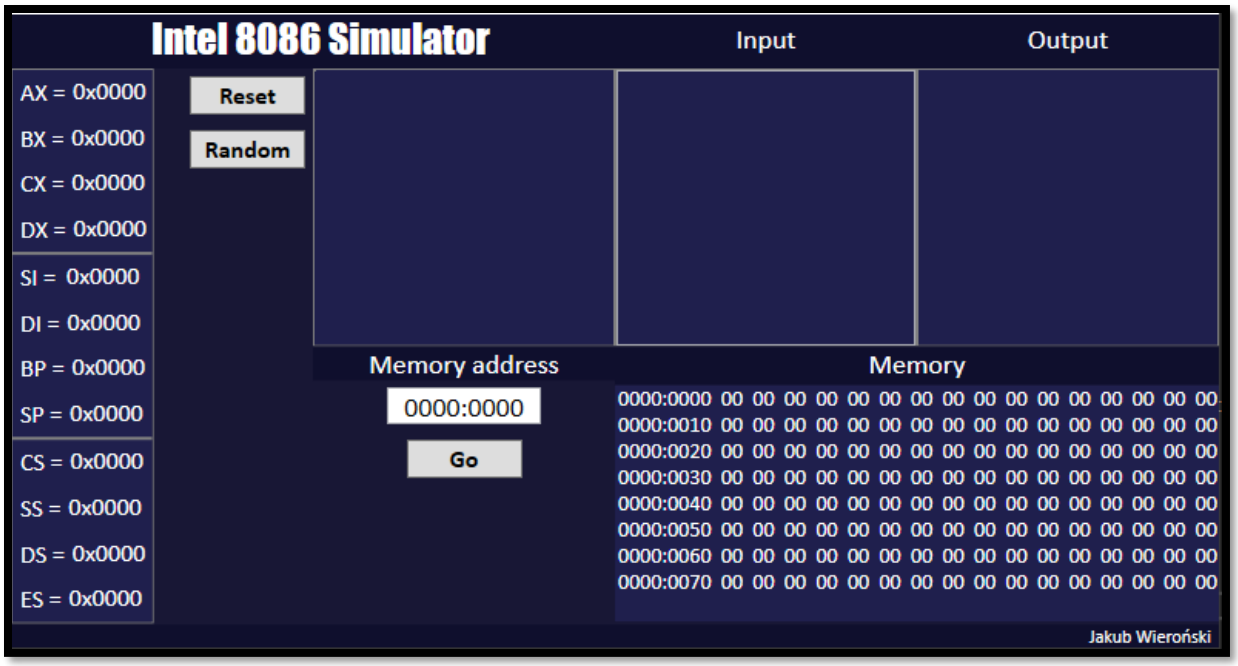
Reprezentacja pamięci wchodzi w skład wzorca architektonicznego MVC, stąd *MemoryModel* implementuje interfejs *Observable* pozwalający poinformować wszelkie obserwujące go instancje o zmianie stanu danej komórki. Przykładowo umożliwia to interfejsowi graficznemu śledzenie zmieniających się miejsc w pamięci.

Aby zmniejszyć potrzebę znajomości wewnętrznej implementacji klasy *MemoryModel* i uniezależnić jej zachowanie od środowiska w jakim zostałaby wykorzystana, stworzona została specjalna klasa definiująca metody rozszerzeń. Pozwala to programiście na stworzenie odpowiednich funkcji opartych o środowisko, które chce zasymulować. To znaczy, że nie musimy w przeładowywać podstawowych nazw funkcji wewnątrz modelu pamięci, kiedy tylko zmieniamy środowisko obsługujące pamięć w odmienny sposób – za każdym razem ingerując w klasę. Zatem powinniśmy zdefiniować odpowiednie metody rozszerzeń, które mogą przykładowo tłumaczyć nam adres pamięci w postaci *string* na *uint*, i takowymi "modułami" należałoby się posługiwać.

3. Specyfikacja zewnętrzna

Program wykorzystuje *Windows Presentation Foundation*, jako swój silnik graficzny i API. Do jego uruchomienia wykorzystujemy plik wykonywalny standardowy dla Microsoft Windows, MS-DOS oraz OS/2, o rozszerzeniu *.exe*, który nosi nazwę "Intel-8086".

3.1. Interfejs graficzny



Interfejs graficzny został podzielony na trzy sekcje.

Po lewej stronie okna aplikacji znajdują się rejestry z przypisanymi im wartościami. Będą się one zmieniać na bieżąco dzięki obserwatorom modeli ich reprezentacji klasowych i specjalnie zaprojektowanym widokom. Wartości te domyślnie są wyświetlane jako heksadecymalne, o czym świadczy przedrostek "0x". Przycisk "Reset" przywraca rejestry ogólnego przeznaczenia do stanu zerowego.
Przycisk "Random" nadaje rejestrom ogólnego przeznaczenia wartości losowe z zakresu [0, 65535].

Okna znajdujące się po prawej stronie, podpisane jako "Input" i "Output", wraz z trzecim oknem bez nazwy stanowią część programu komunikującą się z użytkownikiem.
Pole "Input" służy do wprowadzania komend, które należy zatwierdzić przyciskiem "ENTER". Każda z nich powinna zostać wprowadzona osobno.

W obszarze "Output" wyświetlają się wszelkie komunikaty odnośnie interpretacji i rezultatu z wprowadzonych do pola "Input" linii poleceń.
Okno niepodpisane stanowi zapis wszelkich argumentów przesłanych do programu. Dzięki niemu użytkownik nie musi pamiętać jaką sekwencje wprowadził uprzednio.

Obszar znajdujący się na spodzie okna aplikacji to widok pamięci symulatora. Dzięki przyciskowi "Go" możemy przenieść się do wybranej komórki po uprzednim wprowadzeniu adresu w postaci [Segment_Danych**:**Przesunięcie].
Przykładowo: 0100:000F.

3.2. Obsługa konsoli i format danych wejściowych.

Symulator został przeznaczony do obsługi podstawowych komend języka assemblera. Komendy są wprowadzane w oknie podpisanym jako "Input", a wszelkie powiadomienia wyświetlane są w miejscu podpisanym jako "Output".
Wprowadzona linia komend jest obojętna na wielkość znaków.

UWAGA: Należy pamiętać o przecinku rozdzielającym argumenty w komendach dwuargumentowych. AssignToRegistry nie jest taką komendą, więc nie wymaga przecinka.

HEX – liczba heksadecymalna. Należy pamiętać o, że wartość maksymalna dla 16 bitów wynosi FFFFh. By nie doszło do niejednoznaczności należy zapisywać liczbę
8 bitową mniejszą od szesnastu jako poprzedzoną zerem.
Przykład: 00h, FFh, 02h, 0Bh, 10h.

DEC – liczba decymalna. Należy pamiętać, że liczba 16 bitowa posiada wartość maksymalną 65535.
Przykład: 128, 255, 0.

REG – oznacza nazwę rejestru zgodną z oficjalną specyfikacją komend mikroprocesora. Przyrostek w postaci liczby "16" lub "8" odnosi się do ilości bitów, przy czym należy pamiętać, że **tylko** rejestry ogólnego przeznaczenia są w stanie być używane jako 8 bitowe.
Przykład: AX, CL, DS, SP, BP.

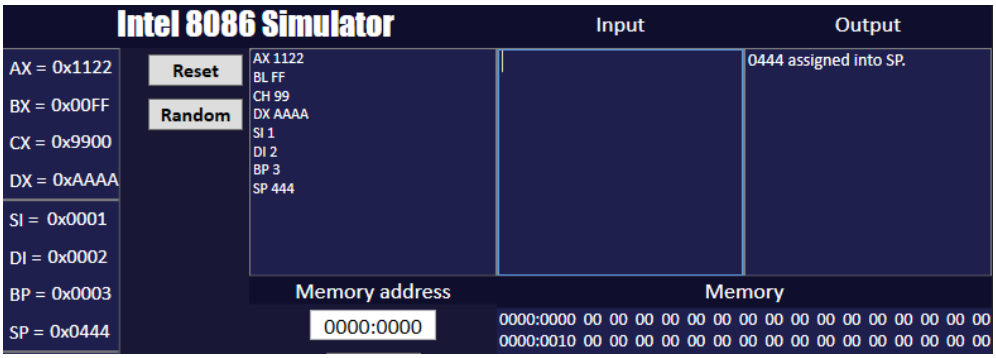
[MEM] – wyrażenie obliczane do adresu efektywnego pamięci. Musi być zgodne z typami adresowania tj. *indeksowym, bazowym, indeksowo-bazowym*, a także musi być zapisane w **nawiasach kwadratowych**.
Przykład: [SI + BP + Fh], [0], [255], [SP + 64], [BP + 0Bh]

Nazwa komendy	Wyrażenia obsługiwane	Opis
AssignToRegistry	"REG8 HEX" "REG16 HEX"	Komenda jest wyrażeniem testowym i nie istnieje w standardowych środowiskach. Pozwala ona na wpisane dowolnej liczby heksadecymalnej do dowolnego rejestru. Tylko przy tej komendzie nie trzeba podawać przyrostka "h".
MOV	"MOV REG8, HEX/DEC" "MOV REG16, HEX/DEC" "MOV REG8, REG8" "MOV REG16, REG16" "MOV REG8, REG16" "MOV REG16, REG8" "MOV REG8, [MEM]" "MOV [MEM], REG8" "MOV REG16, [MEM]" "MOV [MEM], REG16"	Wyrażenie kopiujące wskazane wartości liczbowe.
XCHG	"XCHG REG8, REG8" "XCHG REG16, REG16" "XCHG REG16, REG8" "XCHG REG8, REG16" "XCHG REG8, [MEM]" "XCHG [MEM], REG8" "XCHG REG16, [MEM]" "XCHG [MEM], REG16"	Wyrażenie zamieniające ze sobą wskazane wartości liczbowe.
PUSH	"PUSH REG16"	Wyrażenie wypychające na stos daną zawartość 16 bitowego rejestru ogólnego

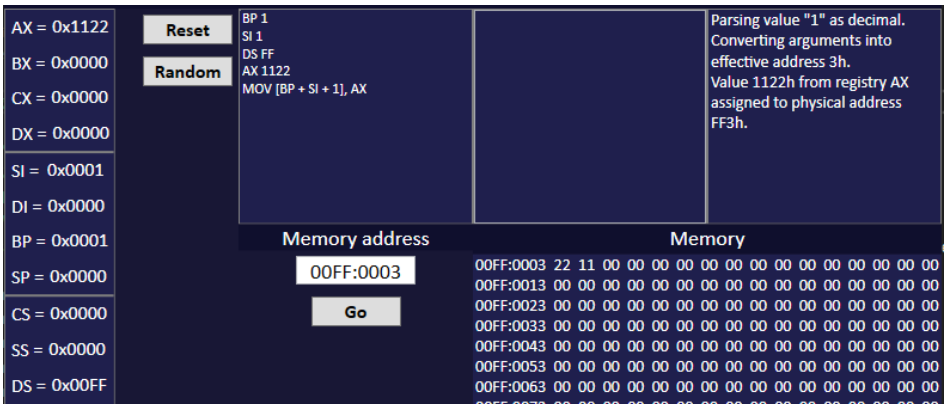
		przeznaczenia. W trakcie wypychania wartość wskaźnika wierzchołka stosu zwiększana jest o dwa.
POP	"POP REG16"	Wyrażenie kopiujące ze stosu wartość i przesyłające je do podanego rejestru 16 bitowego. Kiedy wartość wierzchołka stosu zostanie skopiowana, wskaźnik stosu zostanie zmniejszony o dwa.

3.3. Prezentacja komend na zrzutach ekranu

AssignToRegistry – Komenda testowa

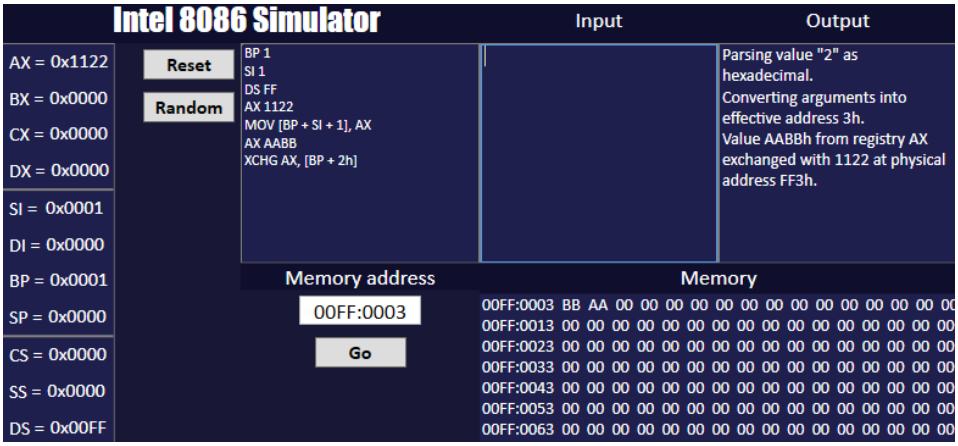


MOV

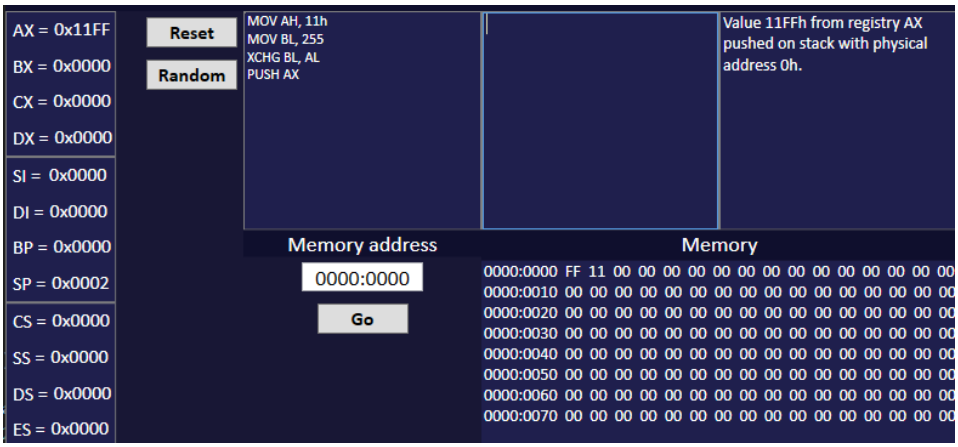

AX = 0x1122
BX = 0x0000
CX = 0x0000
DX = 0x0000
SI = 0x0001
DI = 0x0000
BP = 0x0001
SP = 0x0000
CS = 0x0000
SS = 0x0000
DS = 0x00FF
Reset
Random
BP 1
SI 1
DS FF
AX 1122
MOV [BP + SI + 1], AX
Parsing value "1" as decimal.
Converting arguments into effective address 3h.
Value 1122h from registry AX assigned to physical address FF3h.
Memory address
00FF:0003
Go
Memory
00FF:0003 22 11 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00FF:0013 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00FF:0023 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00FF:0033 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00FF:0043 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00FF:0053 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00FF:0063 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

XCHG

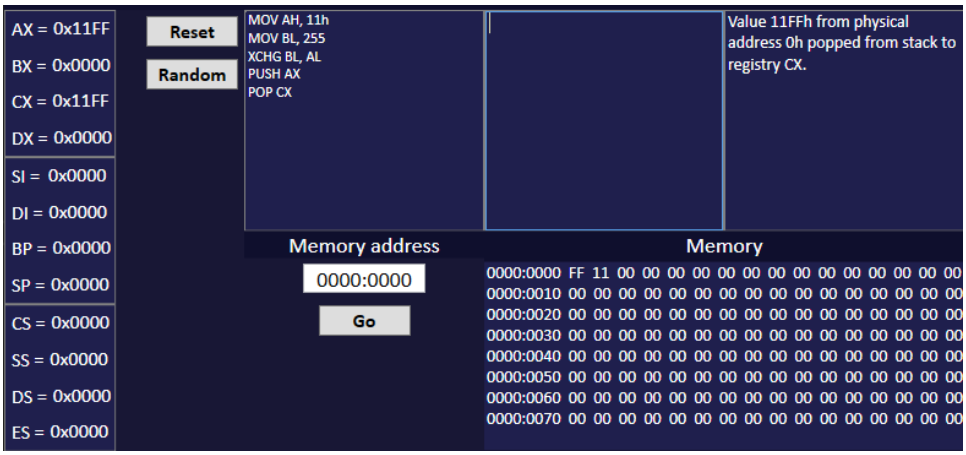

AX = 0x11FF
BX = 0x0000
Reset
Random
MOV AH, 11h
MOV BL, 255
XCHG BL, AL
BL exchanged with AL.


Intel 8086 Simulator
Input
Output
AX = 0x1122
BX = 0x0000
CX = 0x0000
DX = 0x0000
SI = 0x0001
DI = 0x0000
BP = 0x0001
SP = 0x0000
CS = 0x0000
SS = 0x0000
DS = 0x00FF
Reset
Random
BP 1
SI 1
DS FF
AX 1122
MOV [BP + SI + 1], AX
AX AABB
XCHG AX, [BP + 2h]
Parsing value "2" as hexadecimal.
Converting arguments into effective address 3h.
Value AABBh from registry AX exchanged with 1122 at physical address FF3h.
Memory address
00FF:0003
Go
Memory
00FF:0003 BB AA 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00FF:0013 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00FF:0023 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00FF:0033 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00FF:0043 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00FF:0053 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00FF:0063 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

PUSH


AX = 0x11FF
BX = 0x0000
CX = 0x0000
DX = 0x0000
SI = 0x0000
DI = 0x0000
BP = 0x0000
SP = 0x0002
CS = 0x0000
SS = 0x0000
DS = 0x0000
ES = 0x0000
Reset
Random
MOV AH, 11h
MOV BL, 255
XCHG BL, AL
PUSH AX
Value 11FFh from registry AX pushed on stack with physical address 0h.
Memory address
0000:0000
Go
Memory
0000:0000 FF 11 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:0010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:0050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:0060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:0070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

POP



3.4. Komunikaty

Symulator przez swoją linię komend posiada wiele przypadków użycia, jednak każde powinno wyświetlić odpowiedni komunikat o ewentualnym problemie, dzięki czemu użytkownik będzie na bieżąco wiedział jakie czynności wykonuje program. Możliwe komunikaty zostały podzielone ze względu na komendę z jaką współpracują.

- **AssignToRegistry**

`"Assigning fixed value to registry requires two arguments."`
Nie podano wystarczająco wiele argumentów, by obsłużyć komendę AssignToRegistry.

`"Cannot parse [valueHex] as hexadecimal."`
Liczba podana w drugim argumencie jest niepoprawna dla liczby heksadecymalnej.

- **MOV**

`"Too few arguments to function MOV."`
Nie podano wystarczająco wiele argumentów, by obsłużyć komendę MOV.

`"MOV arguments must be separated by comma."`
Argumenty nie zostały oddzielone przecinkiem.

`"Argument is missing bracket."`
Argument zawierający adres efektywny do komórki pamięci nie został oznaczony nawiasem kwadratowym.

`"Physical address [physicalAddress]h is out of range memory. Aborting operation MOV."`
Obliczony adres fizyczny przekroczył obsługiwaną ilość komórek pamięci i operacja MOV nie zostanie wykonana.

```
Value [value]h assigned to registry [destinatedRegistry] from physical address
[physicalAddress]h.
```
Podana wartość została pomyślnie przypisana do rejestru docelowego z komórki pamięci o wskazanym adresie fizycznym.

```
"Value [value]h from registry [sourcedRegistry] assigned to physical address
[physicalAddress]h."
```
Wartość z podanego rejestru została pomyślnie przypisana do wskazanej komórki pamięci o danym adresie fizycznym.

```
"[destinatedRegistry] is unsupported registry name."
"[sourcedRegistry] is unsupported registry name."
```
Rejestr docelowy/źródłowy jest niewspierany przez operację.

```
"[value]h moved into [destinatedRegistry]."
```
Wartość została pomyślnie skopiowana do rejestru docelowego.

```
"[sourcedRegistry]h moved into [destinatedRegistry]."
```
Wartość z podanego rejestru została pomyślnie skopiowana do rejestru docelowego.

```
"Cannot parse arguments as memory address.
Argument [arg] is invalid."
```
Jeden z argumentów podanych do obliczenia adresu efektywnego jest nieprawidłowy.

```
"Converting arguments into effective address [effectiveAddress]h."
```
Pomyślnie obliczono wartość adresu efektywnego.

- **XCHG**

```
"Too few arguments to function XCHG."
```
Nie podano wystarczająco wiele argumentów, by obsłużyć komendę XCHG.

```
"XCHG arguments must be separated by comma."
```
Argumenty nie zostały oddzielone przecinkiem.

```
"[firstdRegistry] is unsupported registry name."
"[secondRegistry] is unsupported registry name."
```
Rejestr pierwszy/drugi jest niewspierany przez operację.

```
"[firstRegistry] exchanged with [secondRegistry]."
```
Pomyślnie zamieniono wartości znajdujące się w obu wskazanych rejestrach.

```
"Argument is missing bracket."
```

Argument zawierający adres efektywny do komórki pamięci nie został oznaczony nawiasem kwadratowym.

```
"Physical address [physicalAddress]h is out of range memory. Aborting operation XCHG."
```

Obliczony adres fizyczny przekroczył obsługiwaną ilość komórek pamięci i operacja XCHG nie zostanie wykonana.

```
"Value [value]h from registry [registryName] exchanged with [registryValue] at
physical address [physicalAddress]h."
```

Pomyślnie zamieniono zawartość wskazanego rejestru z wartością znajdującą się pod wskazanym adresem fizycznym pamięci.

```
"Cannot parse arguments as memory address.
Argument [arg] is invalid."
```

Jeden z argumentów podanych do obliczenia adresu efektywnego jest nieprawidłowy.

```
"Converting arguments into effective address [effectiveAddress]h."
```

Pomyślnie obliczono wartość adresu efektywnego.

- **PUSH**

```
"Pushing registry on stack requires two arguments."
```

Nie podano wystarczająco wiele argumentów, by obsłużyć komendę PUSH.

```
"Cannot use command PUSH for half registers."
```

Komenda PUSH nie obsługuje rejestrów połówkowych, a całe 16 bitowe słowa, operacja zostaje anulowana.

```
"[sourcedRegistryName] is unsupported registry name."
```

Rejestr źródłowy jest niewspierany przez operację.

```
"Command interpreter couldn't find SP registry."
```

Komenda nie została zaopatrzona w obsługę rejestrów wskaźnikowych i nie można pobrać wartości wskaźnika stosu.

```
"Command interpreter couldn't find SS registry."
```

Komenda nie została zaopatrzona w obsługę rejestrów segmentowych i nie można pobrać wartości początku segmentu stosu.

```
"Value [value]h from registry [sourcedRegistryName] pushed on stack with physical
address [physicalAddress]h."
```

Pomyślnie wypchnięto zawartość wskazanego rejestru na stos o podanym adresie fizycznym.

- **POP**

```
"Pushing registry on stack requires two arguments."
```

Nie podano wystarczająco wiele argumentów, by obsłużyć komendę PUSH.

```
"Cannot use command PUSH for half registers."
```

Komenda PUSH nie obsługuje rejestrów połówkowych, a całe 16 bitowe słowa, operacja zostaje anulowana.

```
"[sourcedRegistryName] is unsupported registry name."
```

Rejestr źródłowy jest niewspierany przez operację.

```
"Command interpreter couldn't find SP registry."
```

Komenda nie została zaopatrzona w obsługę rejestrów wskaźnikowych i nie możne pobrać wartości wskaźnika stosu.

```
"Command interpreter couldn't find SS registry."
```

Komenda nie została zaopatrzona w obsługę rejestrów segmentowych i nie możne pobrać wartości początku segmentu stosu.

```
"Value [value]h from registry [sourcedRegistryName] pushed on stack with physical
address [physicalAddress]h."
```

Pomyślnie wypchnięto zawartość wskazanego rejestru na stos o podanym adresie fizycznym.

```
"The stack is empty, cannot perform operation."
```

Próba zdjęcia wartości ze stosu zakończyła się niepowodzeniem, gdyż stos jest pusty.

4. Specyfikacja wewnętrzna

4.1. Projekt sumulatora "Intel-8086"

Przestrzenie nazw

W projekcie zdefiniowane są następujące przestrzenie nazw:

Registers

Zawiera klasy reprezentujące rejestry wraz z ich *widokami* i wspólnym interfejsem pozwalającym na pracę na rejestrach, jak na kontenerach.

Console

Posiada wszelkie definicje typów wchodzących w skład wzorca *Chain of Responsibility*, który interpretuje polecenia wysyłane z bloku wprowadzania "Input".

MemorySystem

To przestrzeń nazw oddzielająca klasy operujące na pamięci operacyjnej i jej prezentacji.

Klasy

W projekcie symulatora znajduje się 29 zdefiniowanych klas.
Lista klas została podzielony według przestrzeni nazw.

Registers

```
public interface RegistersController
```

Interfejs nadający klasie zachowania odpowiednie dla kontenera rejestrów.
Funkcje, o które rozszerza klasę go implementującą:

```
public bool Contains(string registryName)
```

Zwraca TRUE lub FALSE w zależności czy kontener obsługuję podaną nazwę rejestru.

```
public byte[] GetRegistry(string registryName)
```

Pobiera wartość znajdującą się w rejestrze o podanej nazwie.

```
public void SetBytesToRegistry(string registryName, params byte[] bytes)
```

Wstawia wartość do rejestru o podanej nazwie.

```
public class GeneralPurposeRegisters : RegistersController, Observable
public class IndexRegisters : RegistersController, Observable
```

```
public class SegmentRegisters : RegistersController, Observable
public class PointerRegisters : RegistersController, Observable
```

Są to konkretne klasy pozwalające na zarządzanie obsługiwanymi przez nie rejestrami, a także rozsyłanie danych, które uległy zmianie. Wartości rejestrów przechowywane są w tablicach jednowymiarowych dwuelementowych przechowujących zmienne typu *byte*. Różnią się nazwami oraz sposobem obsługi każdego z rejestrów.

```
public class GeneralPurposeRegistersView : Observer, INotifyPropertyChanged
public class IndexRegistersView : Observer, INotifyPropertyChanged
public class PointerRegistersView : Observer, INotifyPropertyChanged
public class SegmentRegistersView : Observer, INotifyPropertyChanged
```

Klasy będące widokami dla konkretnych reprezentacji rejestrów.
Każda implementuje interfejs *Observer*, pozwalający widokowi na zostanie obserwatorem kontrolera rejestrów.
Implementują również interfejs *INotifyPropertyChanged* aktualizujący wartości wyświetlane w *UI*, dzięki MVVM.
Zmienna typu *NumeralConverter* jest zawarta w relacji *ma* w widokach i pozwala na zmianę metody wyświetlania wartości przez wstrzykiwanie zależności przez konstruktor widoków.

Console

```
public interface CommandHandler
```

Interfejs nadający klasie implementującej go możliwość dołączenia do łańcucha, według wzorca *Chain of Responsibility*.
Rozszerza klasę go implementującą o:

```
public CommandHandler NextHandler { get; set; }
```

Pole dzięki któremu można uzyskać następną instancję w łańcuchu.

```
public string HandleOperation(string[] args)
```

Funkcja obsługująca wprowadzone argumenty. Dzięki strukturze łańcucha może być ona wywoływana na każdym następującym po sobie ogniwie, aż do znalezienia pasującej interpretacji polecenia. Wówczas zwraca ona do miejsca wywołania ciąg znaków będący raportem z wykonanej operacji. W symulatorze raport ten wyświetlany jest w bloku "Output".

```
public interface CommandInterpreter
```

Interfejs nadający klasie odpowiedzialność za przetworzenie wprowadzonej linii komend pracując na interfejsie *CommandHandler*.

Rozszerza klasę go implementującą o:

```
public void InputCommand(string line)
```

Funkcja odbierająca wprowadzoną linię komend.

```
public void AddHandler(CommandHandler handler)
```

Funkcja dodająca do łańcucha następnego odbiorcę komendy.

```
public static class CommandFormatter
```

Statyczna klasa oferująca formatowanie argumentów. Została stworzona w celu uniknięcia powtarzania się kodu[4] w klasach interpretujących wprowadzone argumenty.

```
public class RegistryCommander : CommandInterpreter
```

Klasa mogąca odbierając linię poleceń i wywołać ją na swojej kompozycji typu *CommandHandler* funkcję *HandleOperation*.

Implementacja funkcji *AddHandler* oparta jest na strukturze stosu. Ostatnia dodana komenda, będzie pierwszą jaka będzie wywoływana na argumentach.

```
public class AssignToRegistry : CommandHandler
```

Klasa będąca niestandardową komendą przypisującą stałą heksadecymalną do rejestru.

```
public class MOV : CommandHandler
```

Klasa przyjmująca argumenty i interpretująca je do komendy MOV assemblera. W przypadku powodzenia przesyłana jest kopia wartości do miejsca docelowego ze wskazanego miejsca źródłowego.

```
public class XCHG : CommandHandler
```

Klasa przyjmująca argumenty i interpretująca je do komendy XCHG assemblera. W razie powodzenia wartości, z miejsca źródłowego i docelowego podanego w argumentach, są zamieniane ze sobą.

---

[4] DRY (ang. Don't Repeat Yourself, pol. Nie powtarzaj się).

```
public class POP : CommandHandler
```

Klasa obsługująca argumenty i intepretująca je do komendy POP assemblera. W przypadku powodzenia ze stosu kopiowana jest wartość do podanego miejsca docelowego. Dodatkowo wskaźnik stosu jest zmniejszany o dwa.

```
public class PUSH : CommandHandler
```

Klasa obsługująca argumenty i intepretująca je do komendy PUSH assemblera. W przypadku powodzenia na stos wypychana jest wartość z podanego miejsca źródłowego. Dodatkowo wskaźnik stosu jest zwiększany o dwa.

```
public class NULL : CommandHandler
```

Klasa będąca bezpiecznym końcem łańcucha. W przypadku, gdy argumenty nie mogą być zinterpretowane przez żaden jego element, wyświetlany jest odpowiedni komunikat zapobiegający wyjątkowi w postaci np. *NullReferenceException*.

MemorySystem

```
public interface Memory
```

Interfejs nadający zachowania odpowiednie do reprezentacji pamięci operacyjnej. Rozszerza klasę go implementującą o:

```
public byte GetMemoryCell(uint address)
```

Funkcja zwracająca komórkę pamięci o wskazanym adresie.

```
public void SetMemoryCell(uint address, byte value)
```

Funkcja wpisująca do komórki pod wskazanym adresem pamięci wartość.

```
public class MemoryModel : Memory, Observable
```

Klasa będąca modelem pamięci mikroprocesora. Komórki pamięci reprezentowane są przez tablicę jednowymiarową o długości zależnej od szerokości magistrali. Według zasady *single responsibility*, klasa reprezentuje jedynie pamięć i można jej użyć w innych kontekstach. Konstruktor klasy posiada prywatny zakres dostępu, ze względu na zastosowanie wzorca projektowego *Singleton* w postaci naiwnej.

```
public class MemoryModelExtension
```

Klasa posiadająca odpowiednie metody rozszerzeń zachowania modelu pamięci. Pozwala to na ograniczenie ingerencji w strukturę oryginalnej klasy w przypadku gdy interpretacja operacji na pamięci jest odmienna w poszczególnym środowisku. Model pamięci został przygotowany tak, by w razie potrzeby przeładowania nazwy funkcji wewnątrz klasy MemoryModel tworzyć zamiast tego rozszerzenia.

```
public class MemoryView
```

Klasa będąca widokiem pamięci, współpracująca z klasą MainWindow WPF'a. Odpowiada za wyświetlenie komórek pamięci i pozwala ona na bieżąco śledzić zmiany w niej dokonywane.

Typy nieopatrzone przestrzenią nazw

```
public interface Observable
```

Interfejs rozszerzający klasę o zachowania operujące na swoich obserwatorach.

```
public void UpdateObservers(object data)
```

Funkcja aktualizująca instancje obserwujące tą klasę.

```
public void AddObserver(Observer observer)
```

Funkcja dodająca obserwatora tej klasy.

```
public void RemoveObserver(Observer observer)
```

Funkcja usuwająca obserwatora tej klasy.

```
public interface Observer
```

Interfejs nadający klasie możliwość obserwowania inne klasy.

```
public void Update(object data)
```

Funkcja aktualizująca klasę w przypadku, gdy doszło do zmiany statusu obserwowanej klasy.

```
public interface OutputController
```

Interfejs nadający klasie możliwość wyświetlania do bloku "Output" w *UI*.

```
public void ReplaceOutput(string line)
```

Funkcja podmieniająca zawartość bloku "Output".

```
public interface NumeralConverter
```

Interfejs oznaczający klasę jako zdolną do konwersji wartości na dostosowany do wyświetlenia w *UI*.

```
public string GetName { get; }
```

Funkcja zwracająca aktualnie wykorzystywany konwerter liczb.

```
public string IntToString { get; }
```

Funkcja konwertująca wartość liczbową typu Int na odpowiedni string.

```
public class HexParser : NumeralConverter
```

Klasa wspomagająca widoki, przetwarzając wprowadzony ciąg znaków na dostosowany do systemu numerycznego.

4.2. Projekt testów jednostkowych "Intel-8086-Tests"

```
public class CommandTest
```

Klasa testująca całą funkcjonalność interpretacji komend, czyli poprawność *Chain of Responsibility*, użytego w tym celu. Metody tworzą odpowiednie rejestry, które są obsługiwane przez poszczególne reprezentacje komend i które będą użyte w głównym projekcie symulatora. Wraz z nimi instancjonowany jest również obiekt służący do odbierania rezultatu operacji – *log*.

W klasie uwzględniony jest każda zaimplementowana komenda, kombinacja ich wprowadzania i skrajne wartości, tj. *przypadki użycia*.

Każda operacja powinna zwracać odpowiednią informację do użytkownika programu, której poprawność także jest sprawdzana.

```
public class GeneralPurposeRegisterTest
public class IndexRegistersTest
public class PointerRegistersTest
public class SegmentRegistersTest
```

Klasy testujące poprawność wprowadzania i wyprowadzania danych z rejestrów.

```
public class RegistersViewTest
```

Klasa agregująca wszelkie testy na widokach poszczególnych rejestrów. Sprawdzana jest tutaj poprawność wyświetlonych danych w formatowaniu zapewnionym przez klasę *HexParser*.

```
public class MemoryModelTest
```

Klasa sprawdzająca poprawność danych wprowadzonych do klasy *MemoryModel*, a także odczytywania poszczególnych komórek pamięci.

5. Wnioski

W dobie języków wysokopoziomowych i wszechobecnych silników graficznych, API, czy też dokumentacji, symulowanie głównych zasad działania mikroprocesora Intel 8086 jest z całą pewnością znacznie prostsza, niż kiedykolwiek wcześniej. Jeżeli jako twórcy programu nie nadamy sobie żadnych wymogów, co do ilości zużytych zasobów, to ich ilość jest dla nas niemal nieograniczona. Warto zwrócić uwagę na fakt, że mikroprocesor ten pracował na jednym megabajcie pamięci, kiedy dzisiaj sam silnik graficzny potrafi zużyć ich ponad pięćdziesiąt.

Uważam też, że warto przyjrzeć się w ten sposób jak rozwijały się technologie i procesy, gdyż jest to świetne podłoże do nauki programowania i ogromna szansa na zdobycie wiedzy, lub chociażby tematu do dyskusji podczas przerwy w pracy. Trudności w stworzeniu projektu mogą wynikać głównie z początkowej fazy tworzenia – projektowania – gdzie należy zastanowić się jak poprawnie powiązać komunikację użytkownika z symulatorem i jak zasymulować kluczowe elementy procesora. Ja osobiście zdecydowałem się na ręczne wprowadzanie komend, by postawić przed sobą jeszcze trudniejsze wyzwanie, które miało skłonić mnie do większego wysiłku umysłowego, lecz jestem świadom, że symulator wciąż da się zrobić w znacznie przystępniejszej wersji, robiącej dokładnie to samo, a minimalizując nakłady pracy, jakie ja musiałem w to włożyć.

Aktualna wersja jest dostępna na Github:
https://github.com/JakubWier/Intel-8086