Piotr Turski

Kierunek: Informatyka
Specjalność: Informatyka Stosowana
Specjalizacja: Bazy danych i aplikacje internetowe
Numer albumu: 362303

Opracowanie i stworzenie asemblera dla procesora SPKFCS

Praca inżynierska
wykonana pod kierunkiem
dr. Sławomira Pawłowskiego
w Katedrze Fizyki Ciała Stałego
WFiIS UŁ

Łódź 2019

2. Wstęp

Asembler (z ang. *assembler*) jest to program tworzący kod maszynowy na podstawie kodu źródłowego (tzw. *asemblacja*) napisanego w niskopoziomowym języku asemblera. Językiem asemblera nazywać będziemy, zatem niskopoziomowy język programowania, w którym istnieje silna korelacja pomiędzy pojedynczymi dyrektywami (poleceniami) wyrażonymi w odpowiednim kodzie źródłowym, a instrukcjami w formie kodu maszynowego. Asemblery dedykowane są do konkretnych architektur i muszą uwzględniać m.in. długość słowa maszyny, sposób adresowania, czy ilość rozkazów procesora. Najczęściej pojedynczej dyrektywie wyrażonej za pomocą kodu źródłowego odpowiada jedna instrukcja kodu maszynowego.[2,3]

Kod źródłowy pierwszych języków asemblera stworzony został na wzór kodu maszynowego.[4] Każdy oddzielny rozkaz procesora otrzymywał swoją niepowtarzalną i zrozumiałą dla programistów nazwę inaczej zwaną mnemonikiem (lub ekstrakodem[12]). Dodatkowo możliwe były operacje na rejestrach oraz nadawanie nazw poszczególnym adresom tworząc w ten sposób zmienne. Wraz ze wzrostem rozwoju technologicznego powstawały nowe rozwiązania, a co za tym idzie nowe, bardziej złożone, języki asemblera i asemblery. Uwolniło to pierwszych programistów od nużącego i długotrwałego przymusu posługiwania się rozkazami w formie liczbowej oraz od konieczności przeliczania adresów.

Możliwe stało się wkrótce tworzenie makr nadających określonym zestawom derektyw oddzielną nazwę i umożliwiającym wykorzystanie tego zestawu za pomocą jednej komendy kodu źródłowego. Powstały również asemblery o krzyżowej funkcjonalności dające możliwość tworzenia kodu maszynowego dla małych architektur podczas pracy na maszynach z oddzielnym, często niepowiązanym w żaden sposób, systemem operacyjnym. Ostatecznie asemblery pozwalały na użycie abstrakcyjnych technik programowania, znanych z wysokopoziomowych języków programowania (jak konstrukcje *if, for, case* czy w końcu zastosowanie paradygmatów programowania obiektowego). Ostatecznie te niskopoziomowe języki programowania zostały wyparte przez wysokopoziomowe, jednak do dziś znajdują zastosowanie w wielu dziedzinach nowoczesnych technologii tam gdzie nadzwyczaj ważna jest optymalizacja wielkości programu lub jego szybkości wykonania. Najważniejsze z tych zastosowań to sterowniki urządzeń, systemy wbudowane czy też systemy czasu rzeczywistego.[2,3]

W dużym uproszczeniu asemblacja dokonana jest poprzez tłumaczenie mnemoników, zastosowanych zmiennych oraz ewentualnych kodów kontrolnych na odpowiednią wartość binarną. Asemblery ze względu na sposób dokonywanej asemblacji dzielą się na jedno- oraz multiprzebiegowe. Ich wspólną cechą jest zdolność określenia wielkości programu maszynowego, czyli ilości instrukcji, w pierwszym przebiegu asemblacji.

Pierwszy rodzaj asemblerów dokonuje tłumaczeniu *ad hoc*, czyli każda dyrektywa tłumaczona jest na kod maszyny w pierwszym, i jedynym, przebiegu asemblacji. Fakt ten nie pozostaje bez wpływu na możliwość wykrywania potencjalnych błędów i zagrożeń znajdujących się w kodzie źródłowym. Innymi słowy kod źródłowy tłumaczony jest bez jakiejkolwiek interpretacji asemblera, bezpośrednio na kod maszynowy, wraz ze znajdującymi się w nim błędami. Asemblery jednoprzebiegowe miały zastosowanie w pierwszej fazie rozwojów technik komputerowych. W tamtych czasach wczytanie nośnika zawierającego zapisany program w kodzie źródłowym, zabierało dość dużo czasu, a pamięć pierwszych komputerów była silnie ograniczona, przez co zastosowanie asemblerów multiprzebiegowych znacznie wydłużało procedurę asemblacji.

Działanie asemblerów multiprzebiegowych jest bardziej skomplikowane i nie ogranicza się do jednego przejścia po kodzie źródłowym. Asemblacja wieloetapowa pozwala na wykonanie, przez asembler, w pierwszym przebiegu tablicy wykorzystanych menmoników, dyrektyw, zmiennych czy adresów. W kolejnych przebiegach asembler dokonuje niezbędnych powiązań, przez co jest w stanie zbudować instrukcje z odniesieniem do uprzednio występujących zmiennych, adresów czy nawet całych instrukcji. Asemblacja multiprzebiegowa daje również możliwość wykrycia potencjalnych błędów i zagrożeń w kodzie źródłowym.[2,3]

2.1. Rys historyczny

Powstanie asemblerów oraz języków służących do ich programowania zbiega się, w oczywisty sposób, z powstaniem pierwszych komputerów posiadających zdolność przechowywania programu. Jako pierwsza, rozwój jej dziedziny informatyki, rozpoczęła Kathleen Booth w 1947 roku. Jej teoretyczna praca powstała w Birkbeck, jako opracowanie języka asemblera dla ARC2 i zainspirowana była uprzednią współpracą z Johnem von Neumann'em i Hermanem Goldstein'em w Instytucie Badań zaawansowanych w Princeton.[5,6]

Następnie w 1948 roku EDSAC (z ang. *Electric Delay Storage Automatic Calculator*) posiadał już asembler nazwany *initial orders* (z ang. rozkazy inicjujące) będący integralną częścią programu rozruchowego (z ang. *boot loader*). Został on opracowany przez David'a Wheeler'a, uznanego przez *IEEE Computer Society* za twórcę pierwszego asemblera, i posiadał jednoliterowe mnemoniki.[7,8,9] W pracach opisujących działanie jego asemblera po raz pierwszy użyty został termin asemblacja w kontekście procesu składania instrukcji procesora z wykorzystaniem informacji zawartej w kodzie źródłowym napisanym za pomocą języka asemblera.[10]

Kolejny asembler stworzył w 1955 roku Stan Poley. Wynikiem jego pracy był SOAP (z ang. *Symbolic Optimal Assembly Program*) dedykowany był dla komputera IBM 650. Pierwszy system operacyjny napisany nie tylko w języku asemblera otrzymał *Burroughs MCP* w 1961 roku. Do jego napisania wykorzystano język ESPOL (z ang. *Executive System Problem Oriented Language*), a konkretnie dialekt *Algol*. Dalszy rozwój tej dziedziny informatyki spowodował powstanie wielu programów napisanych za pomocą różnych języków asemblera. Do najpopularniejszych należy zaliczyć system operacyjny *IBM PC DOS*, wczesne wersje kompilatora języka *Turbo Pascal* czy program kalkulacyjny *Lotus 1-2-3*. W celu uzyskania najlepszej wydajności większość gier na konsolę *Sega Saturn* było napisanych w języku asemblera.[11]

We wczesnych latach osiemdziesiątych i na początku lat dziewięćdziesiątych XX wieku języki asemblera były podstawowymi językami programowania domowych komputerów jak na przykład Commodore 64, Commodore Amiga, Atari ST. W ówczesnym czasie język BASIC nie dostarczał zadowalającej prędkości wykonania programu, więc języki asemblera były jedynym sposobem pełnego wykorzystania tych architektur. Niektóre domowe komputery zawierały nawet środowiska IDE (z ang. *Integrated Develpment*

Environment) posiadające możliwość debugowania czy też asemblacji w czasie rzeczywistym.[3]

We współczesnym świecie rola niskopoziomowych języków programowania została znacznie ograniczona przez zastosowanie języków wysokopoziomowych. Jednak tak długo jak ważne będzie wykorzystanie maksymalnych zdolności sprzętowych oraz wielkość programu, języki asemblera będą mieć swoje zastosowanie. W nowoczesnych technologiach języki niskopoziomowe znajdują zastosowanie w:

- tworzeniu sterowników sprzętowych
- poprawnej obsłudze przerwań procesora
- tworzeniu programów rozruchowych[13], programów typu firmware czy wirusów
- tworzeniu programów, których wykonanie bezpośrednio zależy od zmiennych warunków w czasie rzeczywistym takich jak: nawigacja samolotu, oprogramowanie urządzeń medycznych czy systemy bezpieczeństwa
- tworzeniu algorytmów kryptograficznych odpornych na atak czasowy
- odtwarzaniu programów metodami inżynierii wstecznej (dezasemblacja[14], dekompilacja[15])
- tworzeniu kodu asemblera bezpośrednio w programach napisanych w językach o relatywnie niskim poziomie (kompilatory języków *Pascal* czy *C*)
- optymalizacji prędkości za pomocą kompilatorów optymalizacyjnych przeznaczonych do renderowania języków wysokiego poziomu do kodu o prędkości wykonania zbliżonej do kodu asemblera[16]

2.2. Rys teoretyczny

Współczesne komputery posiadają bardzo skomplikowaną architekturę, a ich projektowanie jest ogromnym wyzwaniem dla ekspertów posiadających wieloletnie doświadczenie oraz wiedzę teoretyczną, najczęściej nabytą na przestrzeni wielu lat edukacji po akademickiej. Z tego też względu, nauczanie osób początkujących najbardziej skomplikowanych architektur oraz najbardziej wyrafinowanych języków asemblera nie ma najmniejszego sensu. Pożądanymi, w tej kwestii, są proste architektury i nieskomplikowane rozwiązania języka asemblera. Symulacja procesora wielocyklowego SPKCFS, zrealizowana przez dr Sławomira Pawłowskiego (Katedra Fizyki Ciała Stałego Wydziału Fizyki i Informatyki Stosowanej Uniwersytetu Łódzkiego) w programie symulacyjnym CEDAR Logic, posiada intuicyjną i prostą architekturę MARIE (z ang.: *a **M**ashine **A**rchitecture that is **R**eally **I**ntuitive and **E**asy*)[1]. Ta 16-bitowa architektura posiada 15 różnych rozkazów procesora, które umożliwiają jej zrealizowanie nawet najbardziej skomplikowanych programów. Wyposażona jest w jednostkę arytmetyczno-logiczną, jednostkę kontrolną, kilka rejestrów oraz pamięć RAM. Charakteryzuje się:

1) słowem binarnym o określonej długości (16 bit)
2) przechowywaniem całości instrukcji programu w pamięci
3) 4-ro bitowym kodem mnemoników oraz 12-sto bitowym adresowaniem
4) adresowaniem komórek pamięci o różnej wielkości słowa (pojedyncze bity nie są adresowane)
5) 4K komórek pamięci RAM (możliwość zaadresowania z użyciem 12-sto bitowego adresowania)
6) 16-sto bitowym akumulatorem (AC), rejestrem instrukcji (IR) i buforem pamięci (MBR)
7) 12-sto bitowym licznikiem programu (PC) oraz rejestrem adresu pamięci (MAR)
8) 8-mio bitowymi rejestrami wejścia (InREG) i wyjścia (OutREG)

W ramach tego rozdziału omówione będą najważniejsze elementy architektury komputerowej oraz elementy języka asemblera procesora wielocyklowego SPKFCS

stworzonego głównie na podstawie lektury książki „*The Essentials of Computer Organization and Architecture, 4th Edition*” autorstwa Julii Labour oraz Lindy Null.

2.2.1. Najważniejsze elementy architektury komputerowej

Rzeczą oczywistą jest, że każdy procesor operuje na danych zapisanych za pomocą kodu binarnego w pamięci komputera. Jednak, aby zrozumieć, w jaki sposób dane trafiają do jednostki obliczeniowej, w spójny i uporządkowany sposób, tak, aby wyegzekwowany program spełniał zamierzony cel, należy poznać najważniejsze elementy architektury procesora.

Każdy procesor można podzielić na dwie główne części: przetwarzania oraz kontroli. Pierwsza z nich odpowiedzialna jest za przechowywanie, transport i przetwarzanie danych. Tworzy ją sieć rejestrów połączona z jednostką obróbki arytmetyczno-logicznej danych. Druga część odpowiedzialna jest za odpowiednie zsynchronizowanie wymiany i kalkulacji danych przy jednoczesnym sprawdzeniu ich zgodności. Obie te części pozwalają na pobieranie instrukcji oraz przetwarzanie ich w odpowiednim czasie i kolejności, a sprawność pracy procesora jest bezpośrednio zależna od sposobu ich zaprojektowania.

2.2.1.1. Rejestry

Rejestry to komórki pamięci podręcznej procesora, w których przechowywane są rozmaite dane w postaci binarnej jak np. adresy komórek pamięci czy licznik programu (*PC od ang. Program Counter*). Większość rejestrów może przechowywać wszystkie typy danych, ale niektóre z nich są wyspecjalizowane tylko na jeden typ jak np.: adres, dane kontrolne czy zwykłe dane liczbowe. Zlokalizowane są one bezpośrednio w procesorze, aby umożliwić szybki dostęp do danych, i posiadają, odmienny od pamięci RAM, system adresowania. Pojedynczy rejestr stanowi grupa 16-stu przerzutników typu D, mogąca tym samym zapisać 16-sto bitowy zestaw danych. Informacja jest zapisywana w rejestrach, odczytywana z nich i transportowana pomiędzy nimi. Za bezpośrednią manipulację danymi rejestrowymi odpowiada jednostka kontrolna procesora.

W tabeli 1 przedstawiono rejestry, w które został wyposażony wielocyklowy procesor SPKFCS.

L.p.	Skrót nazwy rejestru	Polska nazwa rejestru	Angielska nazwa rejestru	Ilość bitów	Opis
1	***AC***	Akumulator	Accumulator	16	Przechowuje dane potrzebne procesorowi do wykonania aktualnej operacji.
2	***MAR***	Rejestr adresu pamięci	Memory Address Register	12	Przechowuje adres komórki pamięci.
3	***MBR***	Bufor pamięci	Memory Buffer Register	16	Przechowuje dane wczytane z pamięci lub czekające na wpisanie do pamięci.
4	***PC***	Licznik programu	Program Counter	12	Przechowuje adres kolejnej instrukcji procesora.
5	***IR***	Rejestr instrukcji	Instruction Register	16	Przechowuje następną instrukcję procesora.
6	***InREG***	Rejestr wejściowy	Input Register	8	Przechowuje dane wczytane z urządzenia zewnętrznego.
7	***OutREG***	Rejestr wyjściowy	Output Register	8	Przechowyje dane czekające na wczytanie do urządzenia zewnętrznego.

Tabela 1: Spis rejestrów wielocyklowego procesora SPKFCS w architekturze MARIE

2.2.1.2. Jednostka arytmetyczno-logiczna

Jednostka arytmetyczno-logiczna (*ALU od ang. Arithmetic Logic Unit*) odpowiedzialna jest za operacje logiczne (np.: porównanie) i arytmetyczne (np.: dodawanie czy odejmowanie) na danych. Wykonywane są one podczas przebiegu programu. Posiada ona dwa wejścia dla danych oraz jedno wyjście i jest nadzorowana przez jednostkę kontrolną. Przetworzenie pojedynczej instrukcji procesora zajmuje najczęściej kilka taktów zegara.

2.2.1.3. Jednostka kontrolna

Jednostka kontrolna monitoruje wykonanie wszystkich instrukcji i czuwa nad prawidłowym transportem danych. Pobiera ona rozkaz z pamięci, bazując na wartości licznika programu PC, a następnie dekoduje go, sprawdza dane w miejscach docelowych. Ponadto wskazuje ALU, jakie operacje mają zostać wykonane i gdzie znajdują się dane wejściowe, obsługuje przerwania i dba o odpowiedni przepływ danych. Jednostka kontrolna używa rejestru statusu, aby odpowiednio reagować na zdarzenia błędne (np.: przeciążenie stosu).

2.2.1.4. BUS

Procesor komunikuje się z innymi komponentami architektury za pomocą magistrali komunikacyjnej zwanej BUS (*skrót od łac. omnibus – dla wszystkich*). Magistralę tworzy zbiór przewodników łączących określone jednostki procesora, pamięć RAM oraz inne komponenty komputera. Szybkość magistrali maleje proporcjonalnie do jej długości oraz ilości urządzeń do niej podłączonych. Magistrala zbudowana jest w taki sposób, aby pojedyncze bity informacji mogły być transportowane równolegle w tym samym czasie. Dodatkowo dane z jednego źródła mogą używać magistrali w jednym momencie, niezależnie od faktu czy dane pochodzą z rejestru procesora czy z urządzenia peryferyjnego. Z tego względu konieczne jest ustalenie zbioru reguł obowiązujących urządzenia podczas używania magistrali. Protokół magistrali komunikacyjnej dzieli najczęściej jednostki na 2 typy: master (urządzenia inicjujące) oraz slave (urządzenia odpowiadające).

Samą magistralę, ze względu na rodzaj transportowanych danych można podzielić na 4 części: szyna sterująca (kontrolna), szyna adresowa (rdzeniowa), szyna danych oraz szyna zasilająca. Szyna danych transportuje informację pomiędzy dwiema jednostkami komputera. Szyna sterująca wskazuje urządzenie uprawnione do korzystania z magistrali jednocześnie wskazując na rodzaj czynności, do jakiej uprawniona jest ta jednostka (np.: zapis do pamięci lub odczyt danych). Szyna adresowa dostarcza informacji o adresie, którego dotyczy operacja wskazana przez szynę kontrolną, a szyna zasilająca dostarcza niezbędniej energii elektrycznej.

Przebieg transportu na magistrali przebiega najczęściej dwuetapowo:

1) wskazanie adresu
2) transport danych z pamięci do rejestru (odczyt) lub transport danych z rejestru do pamięci (zapis)

a każdy etap transportu zrealizowany jest w ramach jednego taktu zegara.

2.2.1.5. Pamięć RAM

RAM (*od ang. rapid access memory*) jest podstawowym typem pamięci cyfrowej o dostępie swobodnym. Jest to rodzina pamięci posiadających możliwość łatwego i szybkiego zapisu oraz odczytu danych, bezpośrednio z dowolnej komórki pamięci. Pamięci RAM dzieli się na statyczne (*SRAM*) oraz dynamiczne (*DRAM*). Pamięci statyczne są szybsze i dłużej utrzymują informację, przez co, w odróżnieniu od pamięci DRAM, nie wymagają dodatkowego układu odświeżania. Niestety pamięć SRAM jest też dużo droższa od pamięci dynamicznej, przez co wykorzystywana jest w małych układach, dla których nie opłaca się budować odrębnego układu odświeżania, a także tam gdzie wymagana jest większa szybkość operacji zapisu/odczytu (pamięć podręczna procesora). W komputerach wymagających większej ilości pamięci stosuje się dynamiczną pamięć RAM.
Bezpośredni dostęp do każdej komórki pamięci gwarantuje system adresowania pamięci RAM. Istnieją pamięci, których każdy bit posiada swój adres, jak również takie, które adresują jedynie słowa maszyny o z góry określonej długości bitów.

2.2.1.6. Zegar systemowy

Każdy komputer zawiera generator taktu podstawowego, stanowiący najczęściej układ cewki i kondensatora lub diody pojemnościowej. Takt generatora determinuje szybkość wykonywania instrukcji procesora. Taki zegar synchronizuje również wszystkie komponenty układu komputerowego. Procesor potrzebuje określonej ilości taktów, aby wykonać pojedynczą instrukcję. Okres taktowania musi być odpowiednio dobrany, tak, aby był większy od maksymalnego czasu propagacji najważniejszych układów komputera, a jednocześnie jak najmniejszy. Widać, więc, że architektura komputera bezpośrednio wpływa na jego sprawność. Komputer może zawierać wyspecjalizowane zegary oraz zegar główny (zegar procesora). Najczęściej występującym wyspecjalizowanym zegarem jest zegar magistrali komunikacyjnej, najczęściej wolniejszy od zegara głównego.

2.2.1.7. Urządzenia wejścia i wyjścia

Urządzenia wejścia i wyjścia (*ang. Input and Output Devices, I/O*) umożliwiają komunikację komputera ze światem zewnętrznym. Są one podłączone przez odpowiedni interfejs, który tłumaczy ich system kodowania na system zrozumiały dla procesora, a w razie potrzeby, również w odwrotnym kierunku. Następnie sygnał transportowany jest przez magistralę komunikacyjną do odpowiedniego rejestru procesora, odpowiadającego za operację, odpowiednio, wejścia lub wyjścia. Wyróżnia się dwa najważniejsze sposoby takiej komunikacji:

1) in memory-mapped I/O – rejestr interfejsu jest zmapowany w pamięci komputera, w skutek czego, kosztem pamięci wzrasta szybkość
2) instruction based I/O – procesor zawiera szereg instrukcji do obsługi urządzeń wejścia i wyjścia, które oszczędzają pamięć kosztem szybkości przetworzenia.

2.2.1.8. Przerwania

Przerwania są zdarzeniami asynchronicznymi służącymi do kontroli systemu komputerowego, z którymi procesor musi umieć sobie poradzić. Przerwania obsługują:

1) żądania interfejsu wejścia/wyjścia
2) błędy arytmetyczne (np. dzielenie przez 0)
3) przeciążenia stosu
4) punkty przerwania programu zdefiniowane przez użytkownika (debugowanie)
5) instrukcje błędne
6) inne

Przerwania mogą następować w czasie wykonywania programu i kontynuować jego egzekucję po obsłużeniu przerwania przez procesor. Jeśli przerwanie wywołane jest przez krytyczny błąd programu, to sytuacja taka skutkować będzie przerwaniem programu. Żądanie przerwania może być również zamaskowane przez procesor do czasu zakończenia ważniejszych instrukcji, zależnie od nadanego im priorytetu.

2.3. Elementy języka asemblera SPKFCS

Celem asemblera jest przetworzenie języka zrozumiałego dla człowieka (programu złożonego z dyrektyw asemblera) na język binarny, zrozumiały dla procesora (zestaw następujących po sobie w odpowiedniej kolejności instrukcji procesora). Program *Assembler SPKFCS* traktuje wczytywany plik z rozszerzeniem .txt, jako pojedynczy program procesora wielocyklowego SPKFCS, a każdy wiersz tego pliku jako oddzielną dyrektywę. Aplikacja przetwarza następnie dyrektywy, bez rozróżniania wielkich i małych liter, na 16-sto bitowe instrukcje procesora. Instrukcje te wyrażone są za pomocą systemu heksadecymalnego, otrzymując w ten sposób 4-ro znakowe słowa, w których każdy znak odpowiedzialny jest za kolejne 4 bity instrukcji procesora.

	Mnemonik				Adres											
Binarnie	1	1	0	0	1	0	1	0	1	0	0	1	0	1	0	1
Heksadecymalnie	C				A				9				5			

Tabela 2: Schemat organizacji instrukcji procesora wielocyklowego SPKFCS

Asembler SPKFCS może, obok dyrektyw asemblera, przetworzyć także instrukcje procesora, wpisane bezpośrednio do programu w formie heksadecymalnej. Traktuje on takie instrukcje, jako równoważną część programu procesora wielocyklowego SPKFCS.

Poniżej omówione zostały wszystkie elementy języka asemblera SPKFCS.

2.3.1. Zasady tworzenia dyrektyw

Podczas pisania programu dla wielocyklowego procesora SPKFCS, użytkownik powinien przestrzegać szeregu zasad. Reguły te zostały podsumowane poniżej:

1) Wczytywalny plik stanowi jeden program procesora SPKFCS.
2) Każdy niepusty wiersz pliku stanowi oddzielną dyrektywę.
3) Litery małe i wielkie traktowane są tożsamo.
4) Znaki białe znajdujące się w jednym wierszu, przed i po dyrektywie, nie mają wpływu na interpretację znaczenia dyrektywy.

5) Ilość spacji i przecinków wchodzących w skład dyrektywy ma wpływ na interpretację znaczenia dyrektywy (patrz tabela 3).
6) Adresy komórek pamięci RAM podawane bezpośrednio muszą być wyrażone za pomocą systemu heksadecymalnego i zawierać dokładnie 3 znaki.
7) Nazwy zmiennych i etykiet nie mogą zaczynać się od znaku cyfry.
8) Komentarze w pliku wczytywalnym nie są dozwolone.

L.p.	Schemat poprawnego wiersza pliku wczytywanego	Ilość znaków spacji	Ilość znaków przecinka	Opis
1	Mnemonik	0	0	Dyrektywa prosta
2	Instrukcja	0	0	Heksadecymalna instrukcja procesora (4 znaki)
3	Mnemonik Adres	1	0	Dyrektywa z adresem
4	Mnemonik Zmienna	1	0	Dyrektywa ze zmienną lub etykietą
5	Etykieta, Mnemonik	1	1	Etykieta z dyrektywą prostą
6	Etykieta, Mnemonik Adres	2	1	Etykieta z dyrektywą i adresem
7	Etykieta, Mnemonik Zmienna	2	1	Etykieta z dyrektywą i zmienną
8	Zmienna, System Wartość	2	1	Deklaracja zmiennej z wpisaniem wartości liczbowej
9	Adres System Wartość	2	0	Deklaracja wartości liczbowej pod wskazanym adresem

Tabela 3: Rodzaje dyrektyw.

Objaśnienie:

Mnemonik – słowo ze zbioru mnemoników

Instrukcja – instrukcja procesora w formie heksadecymalnej (dł. 4 znaki)

Adres – heksadecymalna wartość adresu komórki pamięci (dł. 3 znaki)

Zmienna – nazwa zmiennej o dowolnej długości, niemogąca zaczynać się od cyfry

Etykieta – nazwa o dowolnej długości, niemogąca zaczynać się od cyfry, odnosząca się do komórki pamięci RAM z zapisaną instrukcją

System – identyfikator systemu liczbowego:

BIN – dwójkowy

DEC – dziesiętny

HEX – szesnastkowy

Wartość – wartość wpisywana do zmiennej odpowiednia dla zadeklarowanego systemu numerycznego

Spis oraz metody użycia poszczególnych mnemoników umieszczone zostały w tabeli 4, a schematy tworzenia dyrektyw, przy użyciu tychże mnemoników, z uwzględnieniem koniecznej ilości znaków spacji i przecinka, ukazuje tabela 3.

Pierwsze dwa schematy z tabeli 3 odwołują się do najprostszej sytuacji, w której funkcjonalną dyrektywę stanowi wiersz zawierający pojedyncze słowo. Sytuacja ta ma miejsce podczas użycia mnemoników bez zastosowania operandu lub podczas użycia instrukcji procesora. Interpreter aplikacji w takiej sytuacji, ma za zadanie jedynie odróżnić taką jednosłowną dyrektywę z użyciem mnemonika od heksadecymalnej instrukcji procesora (lub odwrotnie) i przypisać im odpowiednie wartości. Instrukcje muszą mieć formę czteroznakowego ciągu znaków, w którym heksadecymalna wartość pierwszego znaku odpowiada wartości rozkazu (mnemonika). Oczywiście, jak to wskazano za pomocą tabeli 2, heksadecymalna wartość ciągu pozostałych trzech znaków odpowiedzialna jest za adres komórki pamięci, której instrukcja będzie dotyczyła. Jeżeli użytkownik chciałby za pomocą instrukcji heksadecymalnej, zadeklarować rozkaz, który nie wymaga zastosowania adresu, to, dla spełnienia warunku długości instrukcji, powinien dopisać wartość „000” po znaku wartości heksadecymalnej rozkazu.

Schematy nr 3 i 4, ukazują budowę dyrektyw tworzonych poprzez użycie mnemoników z operandem. W tym przypadku ważne jest, aby pomiędzy mnemonikiem, a operandem znajdował się tylko jeden znak spacji. Jest to najczęściej występujący typ dyrektyw, który w uogólnieniu ma formę:

Mnemonik X

gdzie:

X - oznacza jedną z poniższych:

- zadeklarowaną zmienną
- etykietę użytą w innej części programu
- adres komórki pamięci RAM
  (trójznakowa wartość heksadecymalna)

Operandy, odpowiednie dla poszczególnych mnemoników, opisane są w tabeli 4 w kolumnie „Operand dyrektywy”.

Wszystkie powyższe schematy dyrektyw, oprócz instrukcji procesora, mogą zostać wykorzystane z użyciem etykiety (schematy nr 5-7, tabela 3). Etykieta pozwala na operowanie adresem komórki pamięci, pod którym zapisana jest instrukcja procesora,

wyrażona odpowiednią dyrektywą znajdującą się po znaku przecinka i spacji. Umożliwia to interpreterowi aplikacji *Assembler SPKFCS*, przede wszystkim, na budowanie pętli, o których mowa w punkcie 2.3.3.

Ostatnie dwa schematy z tabeli 3 opisują sposoby deklaracji zmiennej i wpisania wartości liczbowej bezpośrednio do komórki pamięci RAM o określonym przez użytkownika adresie. Zaleca się, aby deklaracje wartości znajdujących się pod określonym adresem lub w stworzonej zmiennej, były zadeklarowane na końcu kodu źródłowego. Wspólnym mianowanikiem schematów tych deklaracji jest konieczność podania systemu liczbowego i wartości odpowiadającej temu systemowi, a różnią się faktem występowania znaku przecinka. Interpreter aplikacji *Assembler SPKFCS* jest przygotowany do odczytania wartości zgodnych z trzema systemami liczbowymi:

1) binarnym – system liczbowy ***BIN***
2) dziesiętnym - system liczbowy ***DEC***
3) heksadecymalnym – system liczbowy ***HEX***

Stąd też deklaracja zmiennej o nazwie Z1, która będzie zawierać wartość heksadecymalną liczby dziesiętnej 29, powinna wyglądać następująco:

Z1, HEX 1D

Wpisując tą samą liczbę do adresu C00 za pomocą systemu binarnego, musielibyśmy zadeklarować dyrektywę w postaci:

C00 BIN 11101

Różnicą techniczną pomiędzy tymi dwoma deklaracjami jest fakt możliwości stosowania nazwy zmiennej, dla której aplikacja *Assembler SPKFCS* automatycznie znajduje wolny adres pamięci.

2.3.2. Mnemoniki i ich użycie

Wielocyklowy procesor SPKFCS zaprojektowany został z myślą o możliwości obsłużenia maksymalnie 16-stu typów rozkazów procesora, a co za tym idzie asembler tego procesora powinien rozróżniać 16 różnych mnemoników. Szczegółowe informacje dotyczące wszystkich mnemoników, obsługiwanych przez aplikację *Assembler SPKFCS*, zawarte są w tabeli 4.

L.p.	Mnemonik	Wartość rozkazu		Działanie		Operand dyrektywy
		bin	hex	Opis	Operacja na rejestrach	
1	**JnS X** *(Jump and Store)*	0000	**0**	Wpisuje wartość z PC pod wskazanym adresem (X) i pobiera instrukcję z następnej komórki pamięci (X+1).	MBR ← PC MAR ← X M[MAR] ← MBR MBR ← X AC ← 1 AC ← AC + MBR PC ← AC	Wartość heksadecymalna adresu lub zmienna (**X**)
2	**Load X**	0001	**1**	Przenosi wskazaną zawartość (X) do AC.	MAR ← X MBR ← M[MAR] AC ← MBR	Wartość heksadecymalna adresu lub zmienna (**X**)
3	**Store X**	0010	**2**	Przenosi zawartość AC do wskazanego adresu lub zmiennej (X).	MAR ← X MBR ← AC M[MAR] ← MBR	Wartość heksadecymalna adresu lub zmienna (**X**)
4	**Add X**	0011	**3**	Dodaje wskazaną zawartość (X) do AC.	MAR ← X MBR ← M[MAR] AC ← AC + MBR	Wartość heksadecymalna adresu lub zmienna (**X**)
5	**Subt X** *(Subtract)*	0100	**4**	Odejmuje wskazaną zawartość (X) od AC.	MAR ← X MBR ← M[MAR] AC ← AC - MBR	Wartość heksadecymalna adresu lub zmienna (**X**)
6	**Input**	0101	**5**	Wprowadza wartość z urządzenia zewnętrznego do AC.	AC ← InREG	Brak
7	**Output**	0110	**6**	Wyprowadza wartość z AC do urządzenia zewnętrznego.	OutREG ← AC	Brak
8	**Halt**	0111	**7**	Kończy program.	Brak	Brak

Tabela 4: Część I - Spis wszystkich mnemoników asemblera SPKFCS z wyjaśnieniami.

L.p.	Mnemonik	Wartość rozkazu		Działanie		Operand dyrektywy
		bin	hex	Opis	Operacja na rejestrach	
9	**Skipcond IDL** *(Skip instruction on condition)*	1000	**8**	Wykonuje działanie logiczne na AC. Jeśli działanie logiczne zwraca prawdę to następna w kolejności instrukcja zostaje pominięta.	Wykonuje działanie **PC ← PC + 1** jeśli działanie logiczne zwraca prawdę. Rodzaj działania logicznego zależny jest od 10-tego i 11-tego bitu indykatora działania logicznego: - *jeśli* **[11-10] = 00** *to działanie logiczne:* ***AC < 0*** - *jeśli* **[11-10] = 01** *to działanie logiczne:* ***AC = 0*** - *jeśli* **[11-10] = 10** *to działanie logiczne:* ***AC > 0***	Heksadecymalny indykator działania logicznego (**IDL**): **{000-3FF}** => IDL[11-10] = 00 **{400-7FF}** => IDL[11-10] = 01 **{800-BFF}** => IDL[11-10] = 10 ***Uwaga! Jeśli zakres IDL: {C00-FFF}*** => *IDL[11-10] = 11* ***to operacja zwraca błąd!***
10	**Jump X**	1001	**9**	Wykonuje skok do instrukcji znajdującej się pod wskazanym adresem (X).	PC ← X	Wartość heksadecymalna adresu, zmienna lub etykieta (**X**)
11	**Clear**	1010	**A**	Wpisuje 0 do AC.	AC ← 0	Brak
12	**AddI X** *(Add Indirect)*	1011	**B**	Pobiera wartość ze wskazanego adresu (X) i używa jej, jako adresu operandu dodawania z AC.	MAR ← X MBR ← M[MAR] MAR ← MBR MBR ← M[MAR] AC ← AC + MBR	Wartość heksadecymalna adresu lub zmienna (**X**)
13	**JumpI X** *(Jump Indirect)*	1100	**C**	Pobiera wartość ze wskazanego adresu (X) i używa jej, jako adresu destynacji skoku.	MAR ← X MBR ← M[MAR] PC ← MBR	Wartość heksadecymalna adresu, zmienna lub etykieta (**X**)
14	**LoadI X** *(Load Indirect)*	1101	**D**	Pobiera wartość ze wskazanego adresu (X) i używa jej, jako adresu wartości do załadowania do AC.	MBR ← X MBR ← M[MAR] MAR ← MBR MBR ← M[MAR] AC ← MBR	Wartość heksadecymalna adresu lub zmienna (**X**)
15	**StoreI X** *(Store Indirect)*	1110	**E**	Pobiera wartość ze wskazanego adresu (X) i używa jej, jako adresu, pod którym zapisuje wartość z AC.	MBR ← X MBR ← M[MAR] MAR ← MBR MBR ← AC M[MAR] ← MBR	Wartość heksadecymalna adresu lub zmienna (**X**)
16	**End**	1111	**F**	Wskazuje koniec dyrektyw programu procesora.	Brak	Brak

Tabela 4: Część II - Spis wszystkich mnemoników asemblera SPKFCS z wyjaśnieniami.

Jedynym mnemonikiem, który nie powoduje żadnej akcji procesora jest mnemonik „End”. Stworzony on został na potrzeby interpretera i wskazuje na koniec dyrektyw programu, a jego użycie nie jest konieczne. Należy tutaj rozróżnić koniec dyrektyw programu, (czyli *de facto* koniec pliku wczytywanego) od końca programu procesora, który to deklarowany jest za pomocą mnemonika „Halt” i może być użyty wielokrotnie w jednym programie procesora SPKFCS, a tym samym może również zajmować jedną z środkowych pozycji wśród dyrektyw. Deklaracja w pełni funkcjonalnej dyrektywy obu powyższych menmoników nie wymaga zastosowania z nimi dodatkowego operandu. Taka sama sytuacja dotyczy również mnemoników „Input”, „Output” oraz „Clear”. Wszystkie inne mnemoniki, do skonstruowania funkcjonalnej dyrektywy, wymagają zastosowania operandu.

Spośród funkcjonalnych mnemoników, najmniej złożone zadanie wykonuje rozkaz zakodowany mnemonikiem „Halt”, który to terminuje program. Nie wykonuje on żadnej operacji na rejestrach.

Najprostszych operacji na rejestrach dokonują mnemoniki „Clear”, „Input”, „Output” oraz „Jump”. Dokonują one jedynie prostego wpisania wartości w komórkę rejestru.

Grupę mnemoników o średniej złożoności działania stanowią: „Load”, „Store”, „Add” i „Subt”. Ich działanie wiąże się z kilkoma działaniami na rejestrach. W akcję tą zaangażowany jest bufor pamięci (MBR).

Istnieje również grupa mnemoników, których rozkazy dokonują operacji niebezpośrednich. Są nimi mnemoniki „LoadI”, „StoreI”, „AddI” i „JumpI”. W pierwszej fazie pobierają one wartość ze zmiennej lub adresu, z którym są zadeklarowane. Następnie używają tejże wartości, jako adresu docelowego działania rozkazu. Wynika z tego, iż pod adresem zmiennej tworzącej z tymi mnemonikami dyrektywę, nie może być wpisana wartość większa od maksymalnej wartości adresowania dla tej architektury (4095 dziesiętnie, czyli binarnie 12 bitów ustawionych na wartość 1).

Na szczególną uwagę zasługują mnemoniki „JnS” i „Skipcond”. Rozkazy zakodowane pod tymi mnemonikami powodują najbardziej złożone akcje procesora. W obu przypadkach, celem ich działania są manipulacje na rejestrze PC, który przechowuje adres kolejnej instrukcji procesora. Mnemonik „JnS” powoduje zapisanie wartości licznika programu pod wskazanym, przez adres lub zmienną X, adresem, a następnie rozpoczyna egzekucję programu od adresu X+1.

Mnemonik „Skipcond”, jako jedyny umożliwia zastosowanie operacji logicznej, której wynik przesądza o sposobie egzekucji dalszej części rozkazów programu. Wynik tejże

operacji logicznej zależny jest od dwóch czynników. Pierwszym z nich jest wartość 10-tego i 11-tego bitu rejestru instrukcji lub wartości heksadecymalnego indykatora działania logicznego. W zależności od tej wartości procesor dokonuje wyboru typu operacji logicznej, mając do wyboru sprawdzenie czy akumulator jest mniejszy, równy czy też większy od zera. Jeżeli, wybrana powyższym sposobem, operacja logiczna zwraca prawdę, to procesor zwiększa wartość licznika programu o 1, tym samym powodując ominięcie jednej, występującej bezpośrednio po aktualnie wykonywanej, instrukcji procesora.

2.3.3. Instrukcje warunkowe, pętle i odwołania do konkretnego adresu

Stworzenie w pełni funkcjonalnego programu wymaga zastosowania współpracujących ze sobą instrukcji warunkowych i pętli. Do zrealizowania tego celu służą etykiety i mnemoniki „Skipcond", „Jump", „JumpI" oraz „JnS", które dokunują zmian w rejestrze licznika programu PC. Wskazuje on na adres następnej instrukcji programu procesora, a co za tym idzie, rozkazy zakodowane tymi czterema mnemonikami, bezpośrednio wpływają na kierunek przebiegu instrukcji programu.
Poprzedzenie dyrektywy etykietą pozwala na wyłuskanie adresu, pod którym w pamięci RAM zapisana jest instrukcja, odpowiadająca tejże dyrektywie. Adres ten może następnie zostać użyty, jako operand innego mnemonika. W ten sposób utworzona zostaje para dyrektyw w formach:

Etykieta, Dyrektywa
Mnemonik Etykieta

gdzie:

Etykieta – nazwa etykiety
Dyrektywa – wyrażenie funkcjonalne z użyciem mnemonika
Mnemonik – dowolny mnemonik

W ten sposób asembler, komórkę docelową działania mnemonika, oznaczy adresem, pod którym znajdować będzie się instrukcja (wartość mnemonika i adresu) przetłumaczona

z dyrektywy. Metodą tą, używając mnemonika „Jump” można łatwo konstruować pętle. Przykładowa para dyrektyw tworząca pętlę wyglądałaby więc następująco:

```
E1, Add Z1
Jump E1
```

Para ta powodowałaby ciągłe dodawanie wartości zmiennej Z1 do akumulatora. W teorii pętla ta byłaby niekończona, ale w praktyce spowodowałaby błąd w momencie próby wpisania do akumulatora wartości większej od 65535, czyli od 16-bitów AC ustawionych na wartość 1.

W celu uniknięcia „nieskończonych” pętli, generujących powyższy błąd wycieku danych wynikającego z faktu przeciążenia rejestru AC, należy w odpowiednim miejscu zastosować instrukcję warunkową za pomocą mnemonika „Skipcond”. Przykładowa pętla, wykonująca dokładnie 7 przebiegów wyglądałaby zatem następująco:

```
Load Seven
E1, Subt One
Skipcond 400
Jump E1
Halt
One, Dec 1
Seven, Dec 7
```

Pierwszą dyrektywą wartość 7 przeniesiona jest do akumulatora. Następnie, dyrektywą „*E1, Subt One*” rozpoczęty zostaje pierwszy przebieg pętli: wartość akumulatora pomniejszona zostaje o 1, a następna dyrektywa sprawdza czy w AC znajduje się wartość 0. Jeśli akumulator posiada wartość większą od 0 to dyrektywa „*Jump E1*” zostaje wykonana. W liczniku programu pojawia się adres instrukcji odpowiadającej dyrektywie „*Subt One*” oznaczonej etykietą *E1*. Instrukcja ta jest następnie wykonywana.

Tą metodą wartość akumulatora jest pomniejszana w każdym przebiegu pętli o 1, aż do momentu osiągnięcia wartości 0. W momencie osiągnięcia przez akumulator wartości 0, instrukcja 8400 (zakodowana dyrektywą „*Skipcond 400*”) zwraca prawdę podczas operacji porównania akumulatora do 0. W wyniku tego licznik programu zostaje zwiększony o 1, co w praktyce przekłada się na ominięcie dyrektywy „*Jump E1*”

i wykonanie dyrektywy „*Halt*” kończącej program. Oczywiście, pomiędzy dyrektywami „*E1, Subt One*” a „*Skipcond 400*” może znajdować się więcej dyrektyw powodujących dodatkową funkcjonalność programu.

Mnemonik „*Skipcond*” ma również zastosowanie w tworzeniu rozbudowanych instrukcji warunkowych. Za jego pomocą użytkownik jest w stanie skonstruować kilka bloków kodu. Odpowiednia część kodu egzekwowana jest z zależności od wyniku operacji logicznej. Metodę tą od pętli odróżnia usytuowanie odpowiednich dyrektyw i odwołań do konkretnego adresu pamięci za pomocą różnych etykiet. Posłużyć może ona, na przykład, do porównania ze sobą dwóch liczb. Przykładowy program, zawierający etykiety *Then, Else, AwB, BwA,* wyglądałby następująco:

```
Load A
Subt B
Skipcond 400
Jump Else
Jump Then
Else, Skipcond 000
Jump AwB
Jump BwA
Then, Load A
…                    #Blok Then (A=B)
Jump Endif
BwA, Load B
…                    #Blok BwA (B>A)
Jump Endif
AwB, Load A
…                    #Blok AwB (A>B)
Jump Endif
Endif, Halt
…
```

Na początku programu załadowany zostaje wynik odejmnowania zmiennej B od zmiennej A znajduje się w akumulatorze. Dyrektywa „*Skipcond 400*” sprawdza czy akumulator jest równy 0. Sytuacja ta może mieć miejsce jedynie, gdy A = B. Jeśli

to prawda to dyrektywa „*Jump Else*” zostaje pominięta, a wykoanana zostaje dyrektywa „*Jump Then*”. W ten sposób program przechodzi do bloku „Then”, który, jak każdy inny blok w tym programie, kończy się skokiem do etykiety Endif. Jeżeli natomiast w akumulatorze jest liczba inna niż 0 to program dokonuje kolejnego porównania po przejściu do dyrektywy „*Skipcond 000*”, poprzez skok do dyrektywy z etykietą „*Else*”. Tym razem sprawdzane jest czy liczba znajdująca się w akumulatorze jest mnejsza od 0, czyli *de facto* czy zmienna B jest większa od A. Następnie, taką samą metodą jak poprzednio, egzekucja kodu zostaje uwarunkowana od wyniku odejmowania A - B. Jeżeli w akumulatorze znajduje się liczba mnejsza od 0 to dyrektywa „*Jump AwB*” zostaje pominięta i wykonany zostaje skok do etykiety „*BwA*”. W przeciwnym razie instrukcje znajdujące się w bloku *AwB* zostają wykonane.

Powyższy program może posłużyć, jako podstawa do stworzenia algorytmu Euklidesa, służącego do wyszukania największego wspólnego dzielnika (NWD):

```
While, Load A
Subt B
Skipcond 400
Jump Else
Jump Then
Else, Skipcond 000
Jump AwB
Jump BwA
Then, Load A
Output
Jump Endif
BwA, Load B
Subt A
Store B
Jump While
AwB, Store A
Jump While
Endif, Halt
```

```
#deklaracje zmiennych np.:
A, DEC 12
B, DEC 3
```

W tym przypadku obok instukcji warunkowych utworzona zostaje pętla *while*, z której wyjście znajduje się w bloku *Then*. Bloki *AwB* i *BwA* wykonują przypisanie wyniku odejmowania odpowiednio do zmiennych A lub B, a następnie skok do początku programu. Odejmowanie wartości mnejszej od większej, a następnie przypisanie wyniku do zmiennej, która posiadała większą wartość powtarzane jest do momentu, aż zmienne A i B są sobie równe. Wtedy algorytm wchodzi do bloku *Then,* w którym wartość NWD przekazana jest do urządzenia wyjściowego i program zostaje zakończony.

3. Założenia asemblera wielocyklowego procesora SPKFCS

Aplikacja *Assembler SPKFCS* została napisana w obiektowym języku programowania C# oraz wyposażona w graficzny interfejs użytkownika z obsługą w języku angielskim.[Załączniki, pkt. 1] Przeznaczona jest ona dla systemów Windows (od Windows 7 do Windows 10) i służy do nauki podstawowych metod programowania procesorów jednocześnie stanowiąc w pełni autonomiczny, krzyżowy i multiprzebiegowy asembler dyrektyw oraz programów dedykowanych dla wielocyklowego procesora SPKFCS.

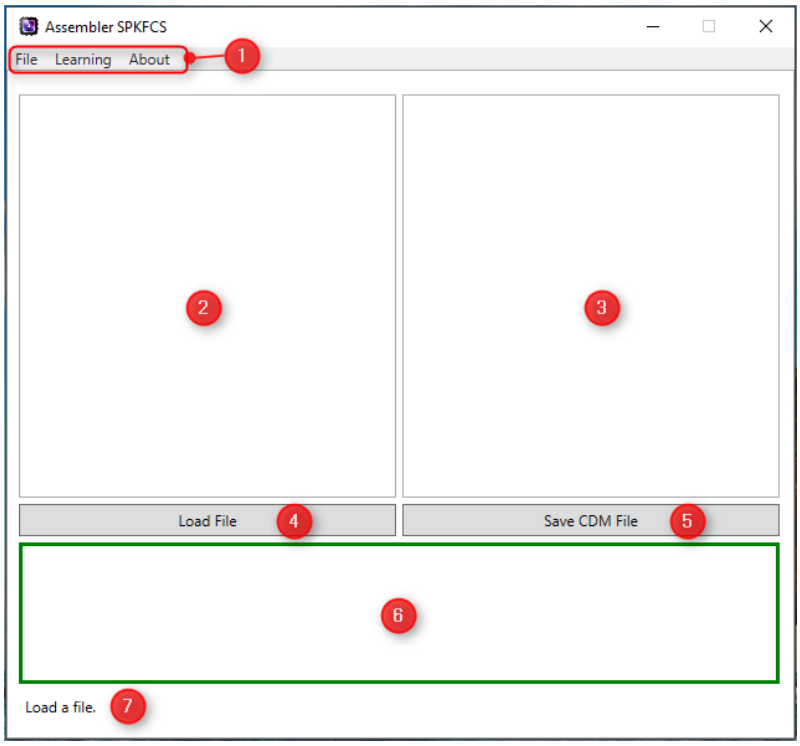
Aplikacja zakłada, że program procesora SPKFCS w formie kompletu dyrektyw będzie wczytywany przez użytkownika w formie pliku tekstowego. Asembler każdą niepustą linię wsadowego pliku tekstowego traktuje jak oddzielną i skończoną dyrektywę. W przypadku linii niezrozumiałych dla programu, wyświetlona zostaje odpowiednia informacja o błędzie. Na podstawie bazy mnemoników procesora SPKFCS, a także przy zastosowaniu odpowiednich metod umożliwiających operacje na zmiennych oraz na adresach, aplikacja dokonuje interpretacji dyrektyw, a w konsekwencji zamiany ich na instrukcje procesora w formie heksadecymalnej. Tak otrzymane instrukcje przypisuje do kolejnych adresów pamięci RAM rozpoczynając od adresu 0. Użytkownik ma możliwość zapisać program w formie heksadecymalnej w postaci pliku z rozszerzeniem .cdm . Plik ten umożliwia automatyczne wczytanie pamięci RAM w programie CEDAR Logic, w którym to architektura procesora SPKFCS została zrealizowana.

Dodatkową funkcją aplikacji jest słowniczek skrótów, mnemoników oraz metod. Umożliwia on użytkownikom poznanie podstaw z zakresu programowania procesora SPKFCS bez konieczności korzystania z dokumentacji. Funkcja ta została stworzona z myślą o wykorzystaniu aplikacji na zajęciach dydaktycznych ze studentami.

4. Funkcje graficznego interfejsu użytkownika

4.1. Opis interfejsu

Assembler SPKCFS jest prostą oraz intuicyjną aplikacją z graficznym interfejsem użytkownika typu Windows Presentation Foundation. Interfejs wyposażony jest w rozwijalne menu zakotwiczone na górnym pasku aplikacji. Okno aplikacji może znajdować się w dwóch stanach: podstawowym oraz rozszerzonym. Bezpośrednio po otworzeniu aplikacji jej okno znajduje się w stanie podstawowym. Przejście do stanu rozszerzonego następuje po uruchomieniu dowolnej funkcji słownika znajdującego się w zakładce *Learning* lub po kliknięciu w zakładkę *About*.



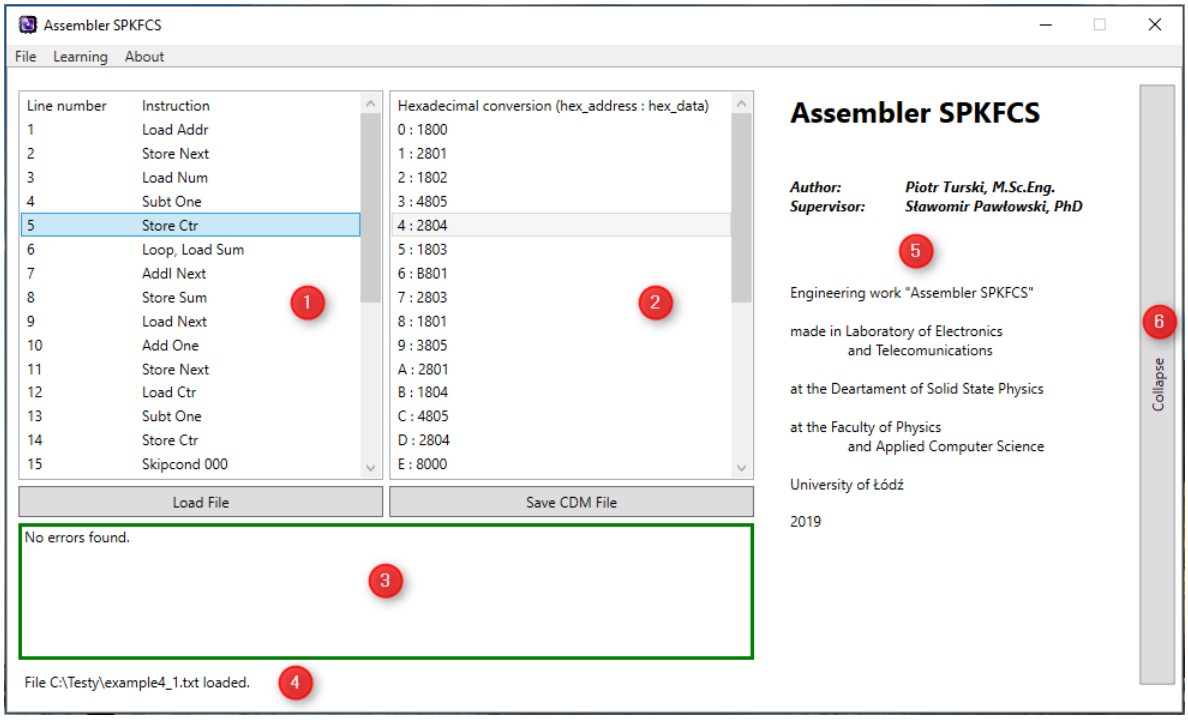
Zrzut ekranu nr 1: Widok okna aplikacji w trybie podstawowym zawierający:

1 — pasek menu górnego, 2 — okno dyrektyw pliku wejściowego, 3 — okno obrazu pamięci RAM, 4 — przycisk służący do załadowania pliku wejściowego, 5 — przycisk służący do zapisania pliku z obrazem pamięci RAM, 6 — okno zawierające ewentualne błędy, 7 — pomocnicza etykieta informacyjna

Okno aplikacji w stanie podstawowym zawiera dwa okna typu ListBox, jedno okno typu TextBox, dwa przyciski oraz etykietę. Budowę okna aplikacji Assembler SPKFCS

w stanie podstawowym, bezpośrednio po jej uruchomieniu, ukazuje zrzut ekranu nr 1. Okna typu *ListBox* usytuowane są obok siebie, tak, aby wyświetlane w nich dane mogły być z łatwością analizowane. Po załadowaniu pliku z programem procesora SPKFCS, lewe okno wyświetla instrukcje programu w formie wierszy-obiektów, a prawe okno, w tej samej formie, wyświetla odpowiadające dyrektywom komórki pamięci RAM. Analizę ułatwia także funkcja powodująca jednoczesne zaznaczenie odpowiadających sobie obiektów na obu oknach typu *ListBox*. Przyciski służące do wczytywania i zapisu plików, znajdują się bezpośrednio poniżej okien typu *ListBox* i pomiędzy oknem typu *TextBox*. Poniżej okna typu *TextBox* znajduje się etykieta, na której wyświetlane są informacje pomocnicze.

Po uruchomieniu którejkolwiek funkcji z zakładek *Learning* i *About* okno aplikacji przechodzi płynnie w stan rozszerzony odpowiednio wydłużając się w prawą stronę (patrz zrzut ekranu nr 2).

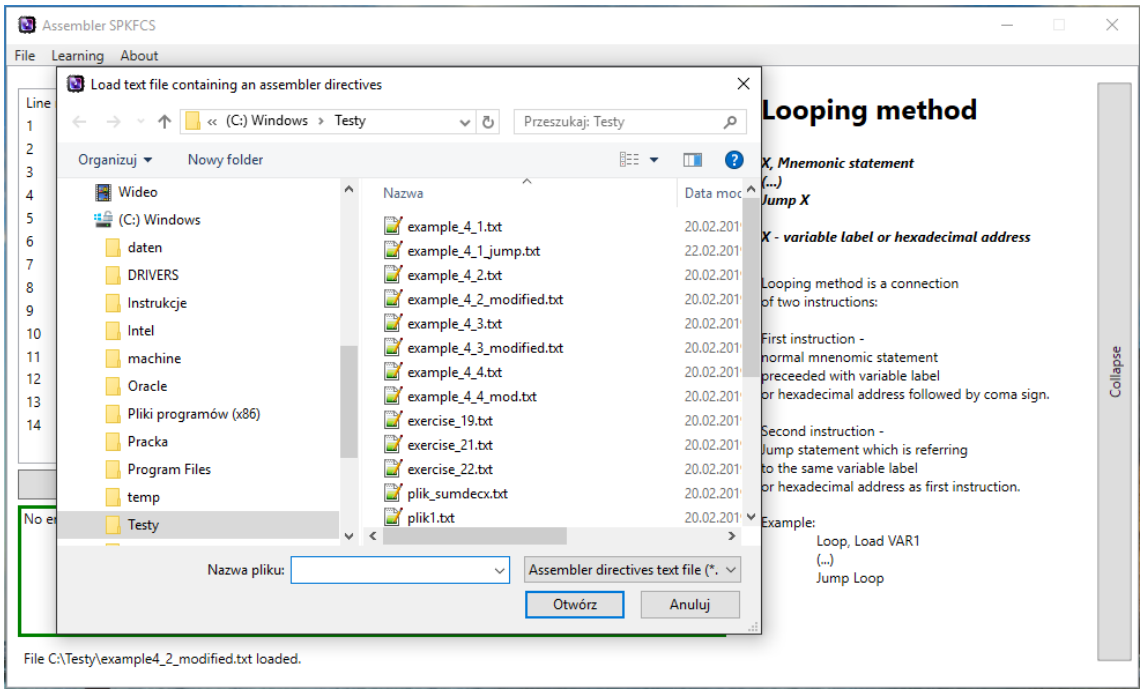


Zrzut ekranu nr 2: Widok okna aplikacji w trybie rozszerzonym, po załadowaniu pliku z poprawnymi dyrektywami, zawierający:

1 – okno dyrektyw pliku wejściowego, 2 – okno obrazu pamięci RAM, 3 – okno informujące o braku błędów dyrektyw pliku wejściowego, 4 – etykieta informująca, który plik został wczytany, 5 – rozszerzona sekcja informacyjna, 6 – przycisk umożliwiający zwinięcie sekcji informacyjnej

W ten sposób utworzona zastaje przestrzeń, na której mogą zostać wyświetlone dodatkowe informacje. Dodatkowa sekcja informacyjna wyposażona jest również w pionowy przycisk opisany tekstem *Collapse*. Przycisk ten umożliwia płynne zwinięcie sekcji informacyjnej i powrót okna aplikacji do stanu podstawowego.

Lewy przycisk opisany tekstem *Load File* umożliwia wczytanie pliku z programem procesora SPKFCS (okno wyboru pliku widoczne jest na zrzucie ekranu nr 3) oraz automatycznie uruchamia funkcję interpretera dyrektyw programu dla wielocyklowego procesora. W wyniku tego działania na lewym oknie typu *ListBox* wyświetlony zostaje zbiór dyrektyw z przyporządkowanymi im numerami linii z pliku wejściowego.



Zrzut ekranu nr 3: Widok okna aplikacji w trybie rozszerzonym z otwartym oknem wyboru wyczytywanego pliku z dyrektywami.

W prawym oknie typy *ListBox* wyświetlony zostaje obraz pamięci RAM w formie heksadecymalnej przy założeniu, że komórki pamięci RAM, które nie są wypisane w obrazie pamięci RAM zawierają jedynie wartości 0. Każda linia (za wyjątkiem linii nagłówkowej) prawego okna ma strukturę typu:

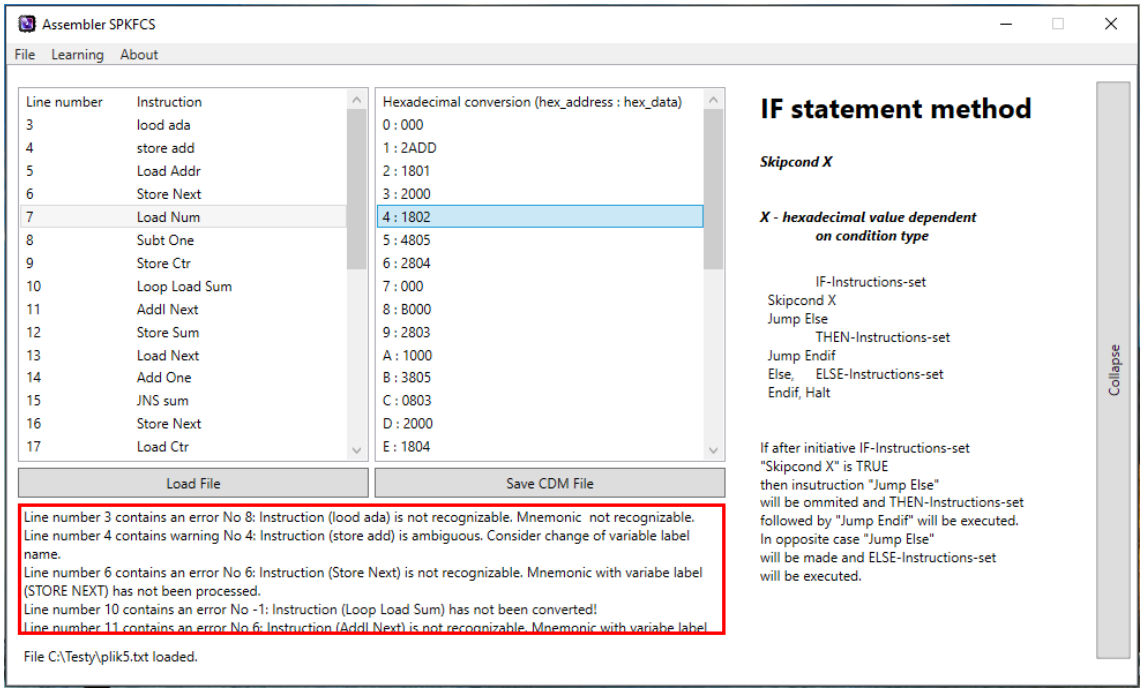
wskaźnik : wartość

gdzie:

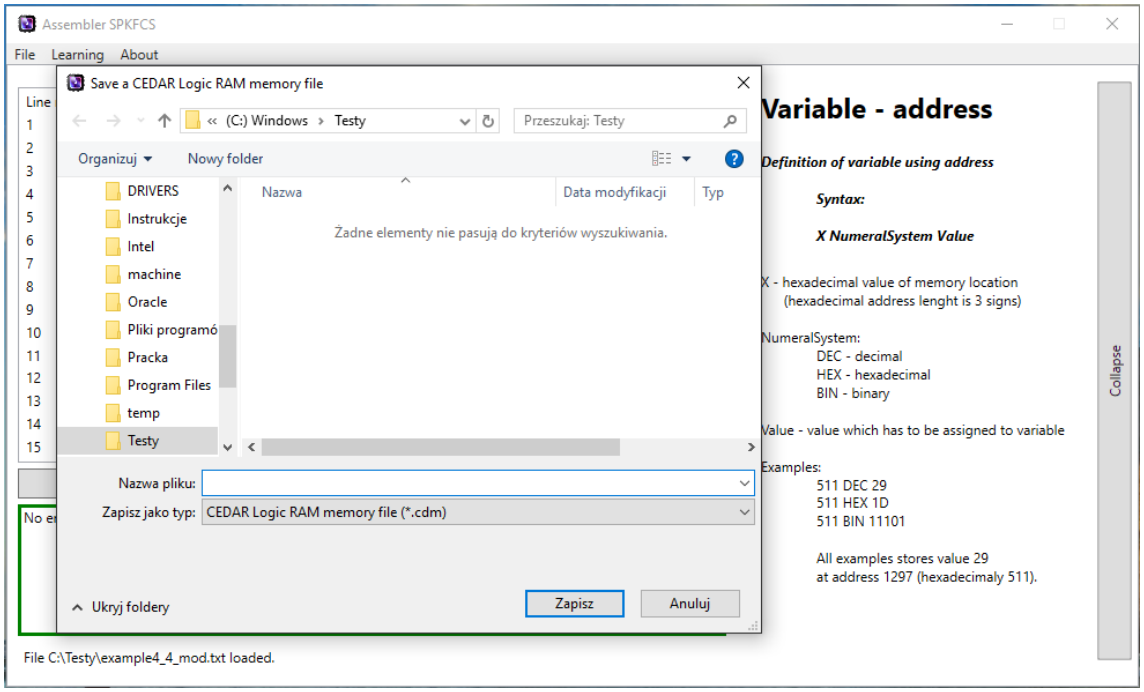
wskaźnik *– adres komórki pamięci RAM w formie heksadecymalnej*

wartość *– heksadecymalna wartość wpisana do komórki pamięci RAM*

W przypadku, gdy dyrektywa wywoła błąd interpretera, odpowiednia informacja zostaje wypisana w dolnym oknie typu *TextBox*, a obramowanie okna zmienia swoje zabarwienie na czerwone. Ten stan możemy zaobserwować na zrzucie ekranu nr 4.



Zrzut ekranu nr 4: Widok okna aplikacji w trybie rozszerzonym po załadowaniu pliku z dyrektywami wywołującymi błędy interpretera.



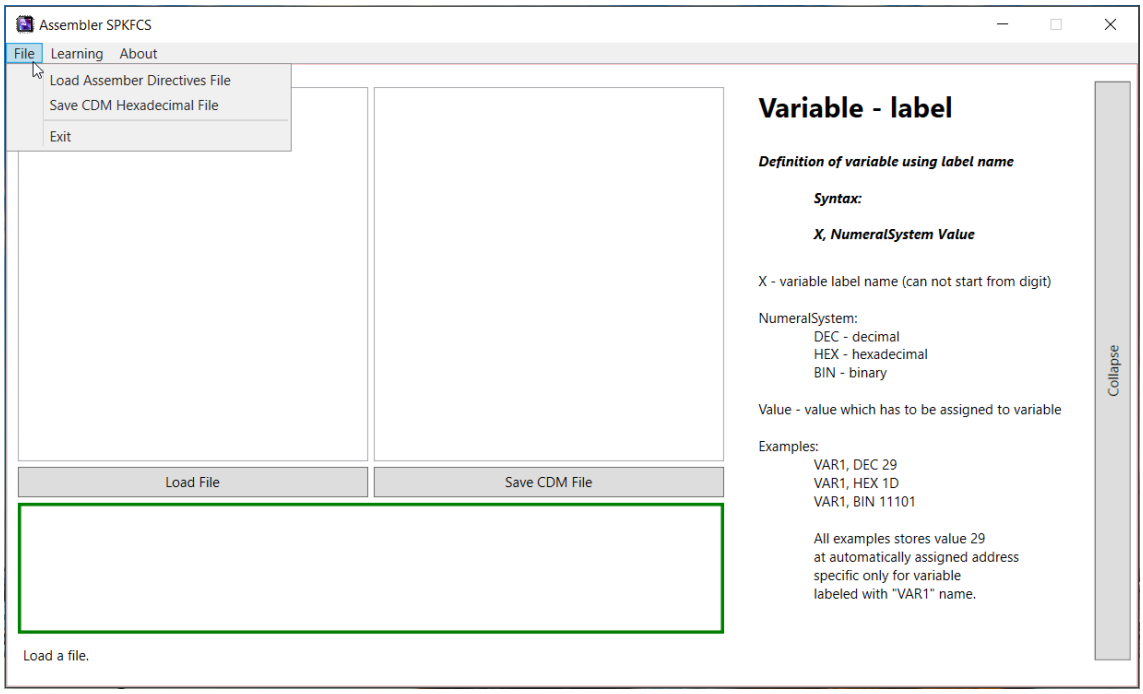
Zrzut ekranu nr 5: Widok okna aplikacji w trybie rozszerzonym z otwartym oknem zapisu pliku z obrazem pamięci RAM.

Prawy przycisk opisany tekstem *Save CDM File* umożliwia zapisanie opisanego powyżej obrazu pamięci RAM w formie pliku z rozszerzeniem .cdm (okno zapisu pliku widoczne jest na zrzucie ekranu nr 5). Plik ten umożliwia zaimportowanie obrazu pamięci RAM do modelu wielocyklowego procesora „SPKFCS" wykonanego w programie symulacyjnym CEDAR Logic. Możliwe jest zapisanie obrazu pamięci RAM, który interpreter wypisał jednocześnie uznając, że program zawiera błędy.

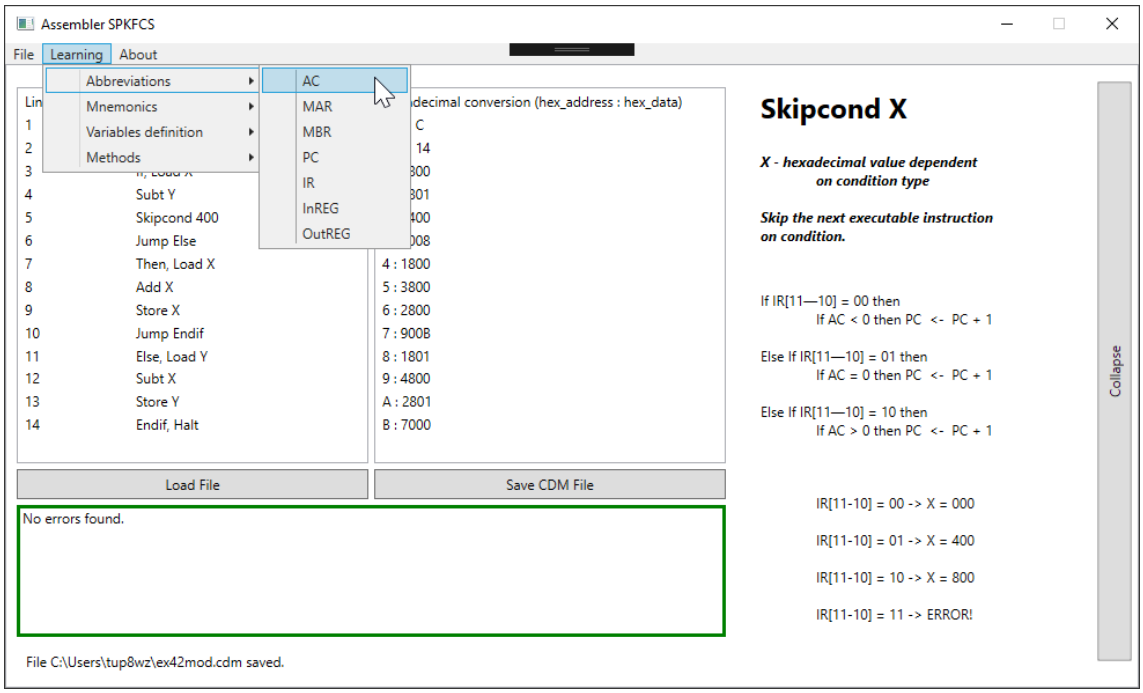
Próba zapisania obrazu pamięci RAM bez uprzedniego wczytania pliku zawierającego instrukcje programu asemblera nie powoduje otwarcia okna zapisu, a na etykiecie wypisana i podświetlona pulsacyjnie zostaje odpowiednia informacja.

4.2. Opis menu górnego

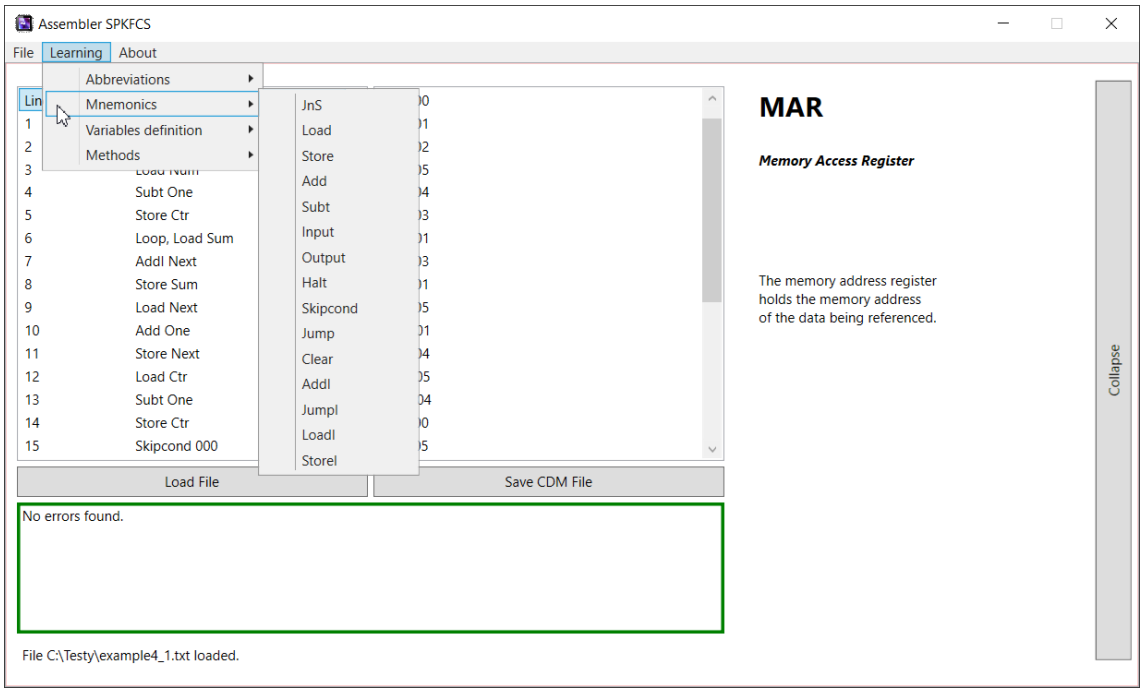
Pasek menu górnego zawiera trzy zakładki: *File*, *Learning* i *About*.
Zakładka *File* widoczna na zrzucie ekranu nr 2 zawiera opcje: *Load Assembler Directives File*, *Save CDM Hexadecimal File* oraz *Exit*. Pierwsze dwie opcje są alternatywą dla przycisków znajdujących się na oknie aplikacji. Opcja *Exit* powoduje zamknięcie aplikacji. Użycie którekolwiek opcji z sekcji *File* nie powoduje zmiany stanu okna aplikacji. Odwrotnie jest z wszystkimi opcjami sekcji *Learning* i *About*. Użycie którejkolwiek z funkcji wylistowanych w tych zakładkach spowoduje przejście okna aplikacji z trybu podstawowego w tryb rozszerzony.
Zakładka *Learning*, której zawartość widoczna jest na zrzutach ekranu nr 7, 8, 9 i 10 pełni funkcję dydaktyczną. Posiada ona rozwijalne sekcje *Abbreviations*, *Mnemonics*, *Variables definition* oraz *Methods*. Funkcje wszystkich wymienionych powyżej sekcji powodują przejście aplikacji w stan rozszerzony oraz wyświetlenie wybranych przez użytkownika objaśnień w bocznej sekcji informacyjnej. Sekcja *Abbreviations* widoczna na zrzucie ekranu nr 7 zawiera objaśnienia skrótów użytych w pozostałych sekcjach.
Sekcja *Mnemonics* widoczna na zrzucie ekranu nr 8 pozwala użytkownikowi zapoznać się ze spisem mnemoników uporządkowanych zgodnie z odpowiadającą im wartością oraz opisem ich działania. Sekcja *Variables definition* widoczna na zrzucie ekranu nr 9 zawiera wyjaśnienie sposobów deklaracji zmiennych z jednoczesnym przypisaniem ich wartości. Zmienne mogą zostać zadeklarowane za pomocą odpowiadającej im etykiety (funkcja Label name) lub bezpośrednio do adresu pamięci RAM (funkcja Address).



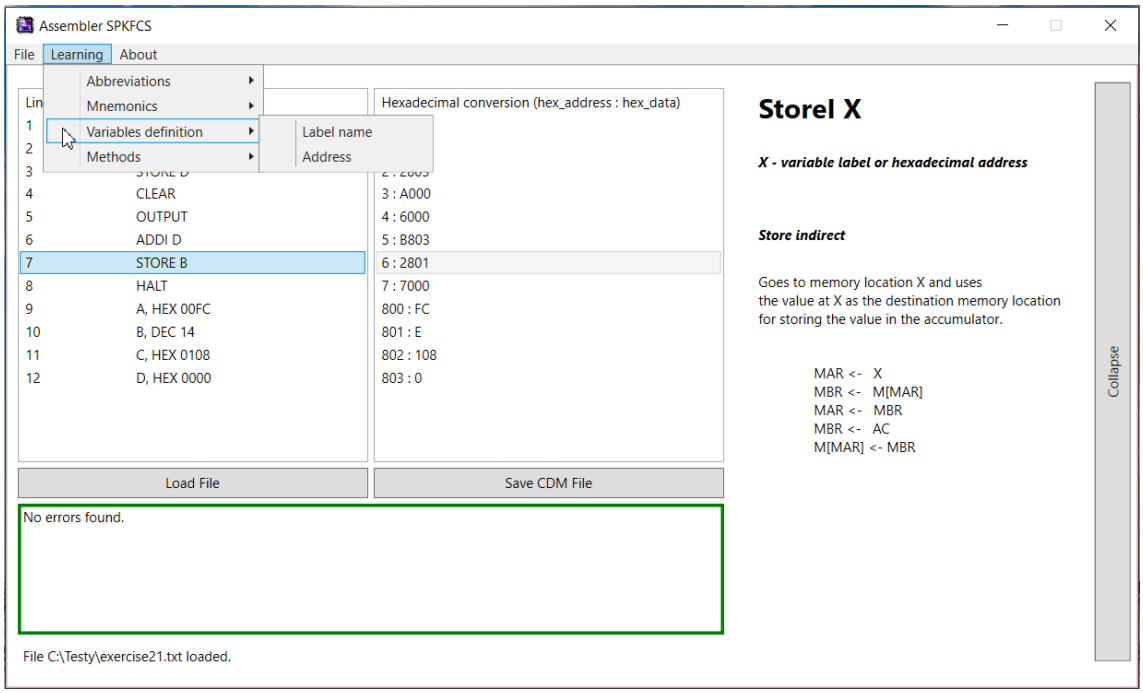
Zrzut ekranu nr 6: Widok okna aplikacji w trybie rozszerzonym po rozwinięciu sekcji File w pasku menu górnego.



Zrzut ekranu nr 7: Widok okna aplikacji w trybie rozszerzonym po rozwinięciu sekcji Learning → Abbreviations w pasku menu górnego.

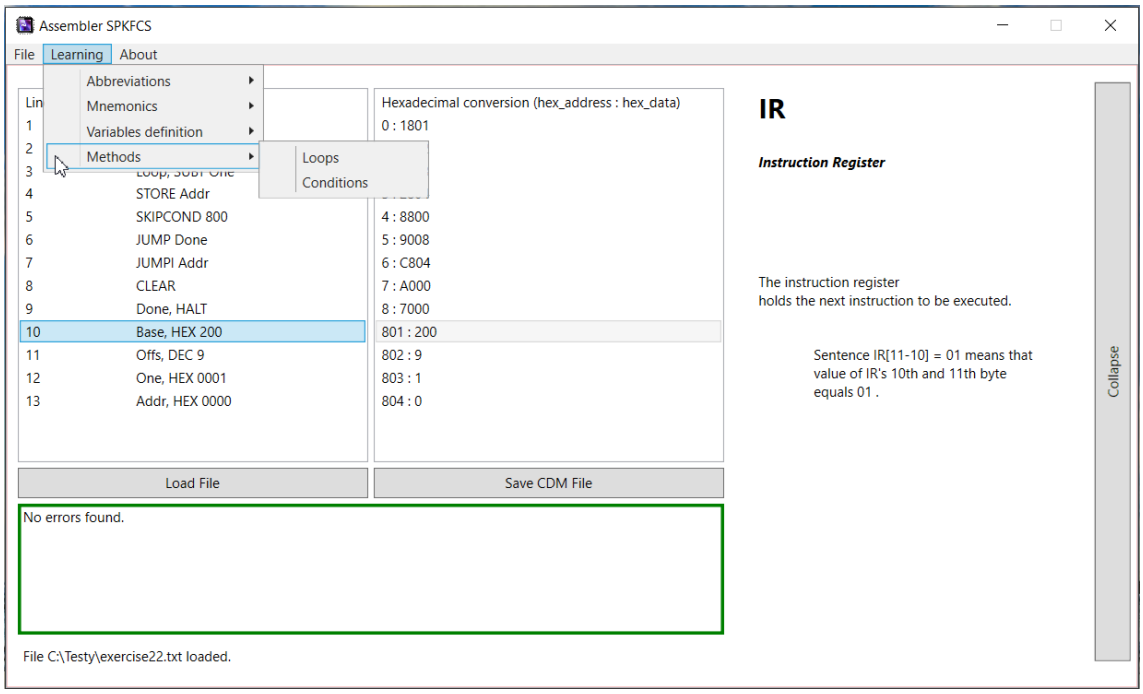


Zrzut ekranu nr 8: Widok okna aplikacji w trybie rozszerzonym po rozwinięciu sekcji Learning → Mnemonics w pasku menu górnego.



Zrzut ekranu nr 9: Widok okna aplikacji w trybie rozszerzonym po rozwinięciu sekcji Learning → Variables definition w pasku menu górnego.

Sekcja *Methods* widoczna na zrzucie ekranu nr 10 zawiera wyjaśnienia sposobów konstruowania pętli (funkcja *Loops*) oraz instrukcji warunkowych (funkcja *Conditions*).

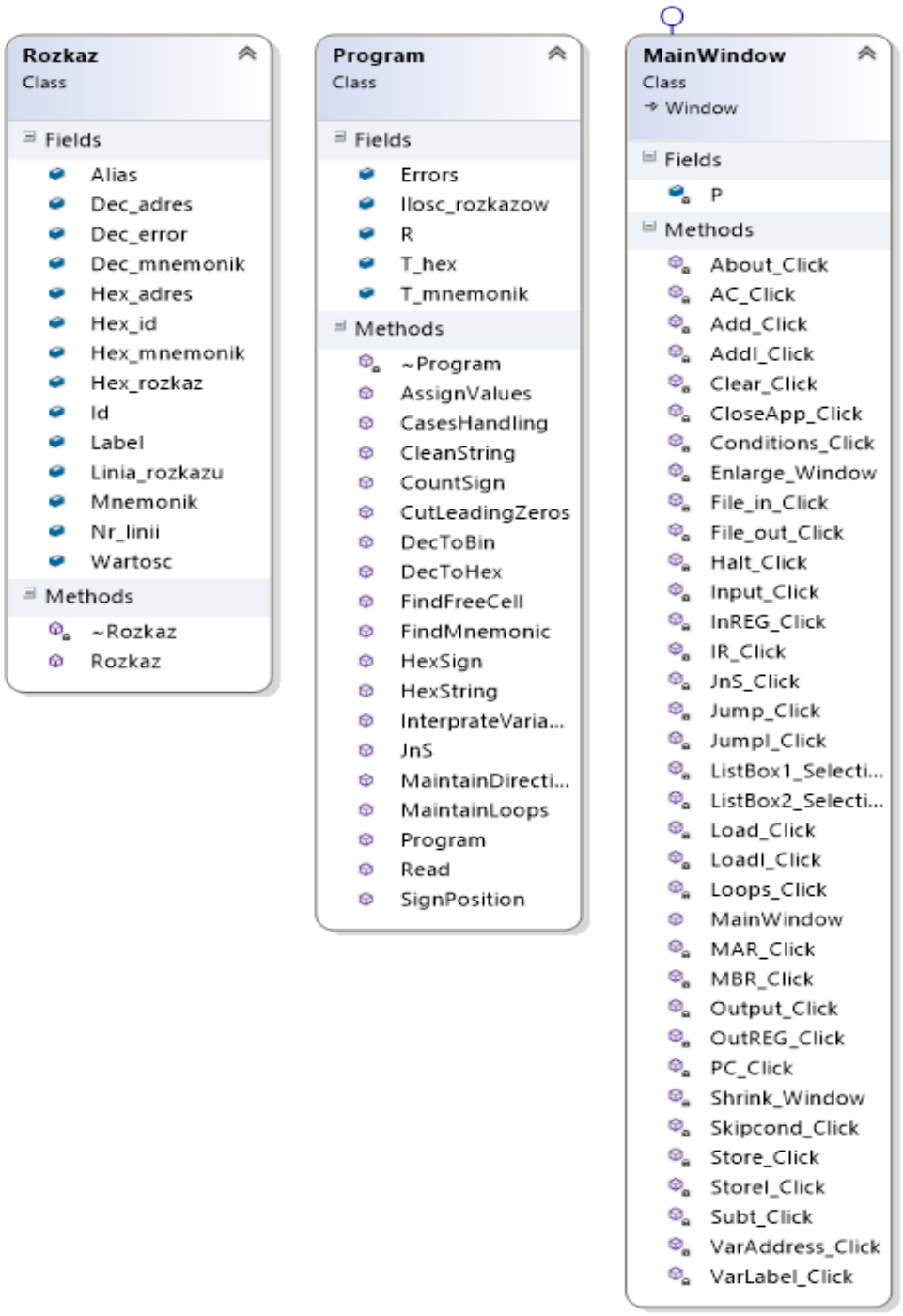


Zrzut ekranu nr 10: Widok okna aplikacji w trybie rozszerzonym po rozwinięciu sekcji Learning → Methods w pasku menu górnego.

Ostatnia zakładka menu paska górnego *About* jest bezpośrednim odwołaniem do funkcji, stąd też nie posiada podsekcji. Zawiera ona podstawowe informacja na temat aplikacji *Assembler SPKFCS*, a efekt jej działania widoczny jest na zrzucie ekranu nr 2 (rozszerzona sekcja informacyjna).

5. Opis klas i ich metod

Aplikacja *Assembler SPKFCS* została wykonana obiektowo w języku C#. Aplikacja zawiera dwie specjalnie dedykowane do niej klasy: Rozkaz oraz Program. Dodatkowo za obsługę okna aplikacji odpowiedzialna jest klasa MainWindow.xaml . Diagram klas aplikacji obrazuje zrzut ekranu nr 11. Aplikacja po uruchomieniu tworzy obiekt klasy MainWindow.xaml i uruchamia okno aplikacji.



Zrzut ekranu nr 11: Diagram klas aplikacji Assembler SPKFCS z widocznymi polami zmiennych oraz metodami.

Wewnątrz obiektu klasy MainWindow.xaml utworzony zostaje obiekt klasy Program. Następnie podczas wczytywania pliku utworzona zostaje tablica klasy Rozkaz, rozmiarem odpowiednio dostosowana do ilości niepustych linii wczytywanego pliku z dyrektywami procesora SPKFCS. W ten sposób dla każdej niepustej linii pliku z programem procesora utworzony zostaje oddzielny obiekt klasy Rozkaz, zaś tablica obiektów klasy Rozkaz (R) znajduje się w obiekcie klasy Program. Całość obsługiwana jest przez obiekt klasy MainWondow.xaml umożliwiający komunikację aplikacji z użytkownikiem.

5.1. Omówienie klas

5.1.1. Klasa Rozkaz

Klasa Rozkaz wyposażona jest w szereg pól zmiennych typu integer i string, konstruktor bezparametrowy oraz destruktor. Pola zmiennych:

```
public int Id;                  //ID Rozkazu r w obiekcie Program P
public string Hex_id;           //ID rozkazu w heksadecymalnie (adres komórki
                                RAM zawierającej rozkaz)
public int Nr_linii;            //numer linii w pliku wejściowym

public string Linia_rozkazu;    //cala linia rozkazu
public string Hex_rozkaz;       //cały rozkaz w formie heksadecymalnej

public string Mnemonik;         //mnemonik - słowo
public int Dec_mnemonik;        //mnemonik - wartość decymalna
public string Hex_mnemonik;     //mnemonik - wartość heksadecymalna

public int Dec_adres;           //wartość decymalna adresu komórki
public string Hex_adres;        //wartość heksadecymalna adresu komórki

public string Alias;            //alias adresu komórki - etykieta zmiennej
public string Label;            //etykieta adresu odniesienia (np. pętla)
public int Wartosc;             //dodatkowe pole z wartością zmiennej

public int Dec_error;           //wartość przypadku błędnego
```

tej klasy służą do analizy dyrektyw programu wielocyklowego procesora SPKFCS. Klasa Rozkaz posiada również publiczny konstruktor bezparametrowy, którego zadaniem jest wpisanie wartości do wyżej wymienionych zmiennych.
Konstruktor wpisuje wartość -1 do zmiennych typu integer, a w przypadku zmiennych typu string wpisany zostaje pusty ciąg znaków.

5.1.2. Klasa Program

Obiekt klasy Program oprócz wspomnianej wyżej tablicy obiektów klasy Rozkaz posiada również dwie pomocnicze tablice typu string i char oraz po jednej zmiennej typu string i integer:

```
public int Ilosc_rozkazow;          //ilość wszystkich obiektów klasy Rozkaz
public string[] T_mnemonik;         //tablica mnemoników (wzorzec)
public char[] T_hex;                //tablica konwersji heksadecymalnej
                                    (wzorzec)
public string Errors;               //wiadomość zawierająca wszystkie błędy
public Rozkaz[] R;                  //tablica rozkazów
```

Dodatkowo klasa Program zawiera publiczny konstruktor bezparametrowy, destruktor oraz szereg publicznych metod.

Publiczny konstruktor bezparametrowy wpisuje do zmiennej Ilosc_rozkazow wartość 0, a do zmiennej typu string (zmienna Errors) pusty ciąg znaków. Następnie tworzy wzorcowe tablice umożliwiające konwersję:

- mnemoników na odpowiadające im liczbowe wartości decymalne (kod deklaracji tablicy T_mnemonik:

```
T_mnemonik = new string[16];
T_mnemonik[0] = "JNS";
T_mnemonik[1] = "LOAD";
T_mnemonik[2] = "STORE";
T_mnemonik[3] = "ADD";
T_mnemonik[4] = "SUBT";
T_mnemonik[5] = "INPUT";
T_mnemonik[6] = "OUTPUT";
T_mnemonik[7] = "HALT";
T_mnemonik[8] = "SKIPCOND";
T_mnemonik[9] = "JUMP";
T_mnemonik[10] = "CLEAR";
T_mnemonik[11] = "ADDI";
T_mnemonik[12] = "JUMPI";
T_mnemonik[13] = "LOADI";
T_mnemonik[14] = "STOREI";
T_mnemonik[15] = "END";
```

)
- liczbowej wartości decymalnej na znak odpowiadający wartości heksadecymalnej (kod deklaracji tablicy T_hex:

```
T_hex = new char[16];
T_hex[0] = '0';
T_hex[1] = '1';
T_hex[2] = '2';
T_hex[3] = '3';
T_hex[4] = '4';
T_hex[5] = '5';
T_hex[6] = '6';
T_hex[7] = '7';
T_hex[8] = '8';
T_hex[9] = '9';
T_hex[10] = 'A';
T_hex[11] = 'B';
T_hex[12] = 'C';
T_hex[13] = 'D';
T_hex[14] = 'E';
T_hex[15] = 'F';


).
```

5.1.2.1. Metody klasy Program

Metody klasy Program, ze względu na przydatność w bezpośredniej analizie dyrektyw programu wielocyklowego procesora SPKFCS, można podzielić na metody główne oraz pomocnicze.

5.1.2.1.1. Metody główne

Metody główne służą do bezpośredniej analizy dyrektyw tworzących program wielocyklowego procesora SPKFCS w postaci tablicy obiektów klasy Rozkaz. Ich celem jest utworzenie odpowiedniego obrazu pamięci RAM, w którym kolejne instrukcje procesora są wpisane do pamięci w odpowiedniej kolejności. W klasie Program stworzone zostały poniższe metody główne:

```
public void Read(string path) ,
public void AssignValues() ,
public void InterprateVariables() ,
public void MaintainLoops() ,
public void MaintainDirectives() ,
public void CasesHandling() .
```

Metoda *Read()* jest pierwszą metodą użytą podczas analizy pliku z programem procesora SPKFCS. Otwiera ona plik z dyrektywami procesora znajdujący się w określonej przez użytkownika lokalizacji na dysku i zlicza wszystkie niepuste linie tego pliku. Następnie tworzy ona odpowiedniej wielkości tablicę o nazwie R z obiektami typu Rozkaz i do każdego obiektu tejże tablicy wpisuje pojedynczą niepustą linię pliku do zmiennej Linia_rozkazu. Przed wpisaniem wartości do zmiennej, linia pliku zostaje oczyszczona z białych znaków występujących na początku i na końcu łańcucha znaków za pomocą metody pomocniczej *CleanString()*. Dodatkowo metoda *Read()* określa wartość zmiennej Id oraz Hex_id, a także wpisuje numer linii pliku do zmiennej Numer_linii.

Metoda *AssignValues()* jest metodą główną interpretera. Dla każdego obiektu klasy Rozkaz z tablicy R[], utworzonej przez metodę *Read()*, zlicza ona ilość wystąpień znaku spacji oraz znaku przecinka w zmiennej Linia_rozkazu. Następnie na podstawie tych danych dokonywana jest wstępna analiza umożliwiająca określenie, jakim rodzajem dyrektywy procesora jest zawartość zmiennej Linia_rozkazu. Można wyróżnić dziewięć rodzajów dyrektyw procesora SPKFCS na podstawie wymienionych w tabeli 3 cech.

Różne rodzaje dyrektyw zawierające taką samą liczbę znaków spacji i znaków przecinka są następnie dywersyfikowane przez algorytm metody *AssignValues()*, a poszczególne zmienne obiektu klasy Rozkaz są uzupełniane w zależności od rodzaju analizowanego przypadku. Jeśli metoda nie jest w stanie określić rodzaju dyrektywy, to wartość zmiennej Dec_error jest zmieniana na wartość większą od 0. W ten sposób potencjalnie błędna dyrektywa jest oznaczana, co powoduje odpowiednią interpretację przypadku błędnego przez kolejne metody główne klasy Program. Metoda *AssignValues()* wszystkie litery słów zmiennej Linia_rozkazu zamienia na wielkie litery. W wyniku tego działania algorytm nie rozróżnia dyrektyw pisanych małymi i wielkimi literami.

Metoda *InterprateVariables()* zawiera algorytm, którego zadaniem jest wyszukanie, dla każdej zadeklarowanej zmiennej, wolnego adresu pamięci RAM, a następnie dopisanie tegoż adresu do wszystkich dyrektyw odnoszących się do zmiennej, której adres właśnie nadano. Dodatkowo metoda sprawdza czy zmienna nie została użyta wraz z mnemonikiem JnS, który wymaga, aby część dyrektyw języka maszynowego była umieszczona w pamięci RAM zaraz po komórce przetrzymującej wartość tej zmiennej.

Algorytm metody *MaintainLoops()* wyszykuje etykiety dyrektyw oraz zmienne o tej samej nazwie. W wyniku tego działania adres komórki zawierającej dyrektywę oznaczoną etykietą zostaje przypisany do wszystkich dyrektyw, które odnoszą się do zmiennej o takiej samej nazwie jak etykieta.

Następnie metoda *MaintainDirectives()* sprawdza poprawność wszystkich heksadecymalnych wartości obrazu pamięci RAM i dokonuje ewentualnych akcji naprawczych na podstawie danych zapisanych w każdym obiekcie typu Rozkaz.
Na koniec metoda CasesHandling() tworzy odpowiednią listę błędów opierając się na wartościach zmiennej Dec_error każdego obiektu Rozkaz. Tak utworzoną listę wpisuje do zmiennej Errors obiektu klasy Program.

5.1.2.1.2. Metody pomocnicze

Metody pomocnicze klasy Program pozwalają uniknąć redundancji kodu metod głównych. Ich wykaz oraz opis działania poszczególnych z nich znajduje się w Tabeli 5.

Metoda	Opis działania
public bool HexSign(char znak)	zwraca false, kiedy znak jest spoza zakresu {0-9} lub {A-F}
public bool HexString(string napis)	zwraca false, kiedy choć jeden znak ciągu znaków napis jest spoza zakresu {0-9} lub {A-F}
public bool JnS(string alias)	zwraca true, kiedy alias jest nazwą zmiennej użytej z mnemonikiem JnS
public int FindMnemonic(string napis)	zwraca wartość dziesiętną mnemonika gdy napis odpowiada którejkolwiek z pozycji w tablicy wzorcowej T_mnemonik w przeciwnym razie zwraca wartość -1
public int FindFreeCell(int poczatek)	analizuje tablicę obiektów klasy Rozkaz i zwraca najniższy wolny adres pamięci RAM rozpoczynając szukanie od adresu początek
public int SignPosition(string napis, char znak)	zwraca pierwszą pozycję znaku w ciągu znaków napis
public int CountSign(string napis, char znak)	zwraca ilość wystąpień znaku w ciągu znaków napis
public string DecToBin(int liczba, int i)	konwertuje liczby z systemu dziesiętnego na system binarny
public string DecToHex(int liczba, int i)	konwertuje liczby z systemu dziesiętnego na system heksadecymalny
public string CleanString(string napis)	usuwa białe znaki na początku i na końcu ciągu znaków
public string CutLeadingZeros(string napis)	usuwa znaki '0' znajdujące się po lewej stronie ciągu znaków pozostawiając przynajmniej jedną cyfrę

Tabela 5: Metody pomocnicze klasy Program z opisem ich działania

5.1.3. Klasa MainWindow.xaml

Obiekt klasy MainWindow.xaml zostaje utworzony podczas uruchamiania aplikacji i odpowiedzialny jest za prawidłową obsługę graficznego interfejsu użytkownika. Konstruktor bezparametrowy klasy MainWindow.xaml tworzy obiekt P typu Program. Pozostała funkcjonalność aplikacji *Assembler SPKFCS* jest zrealizowana za pomocą metod klasy MainWindow.xaml.

5.1.3.1. Metody klasy MainWindow.xaml

Najważniejszą z metod klasy MainWindow.xaml jest metoda

```
private void File_in_Click(object sender, RoutedEventArgs e) ,
```

która pozwala użytkownikowi na wybranie konkretnego pliku do wczytania w trybie okienkowym (patrz zrzut ekranu nr 3), i na podstawie ścieżki dostępu do tego pliku uruchamia metody główne klasy Program w odpowiedniej kolejności. Następnie do okna typu *TextBox* wpisuje zmienną Error jednocześnie odpowiednio zmieniając kolor obramowania tego okna. Na końcu metoda uzupełnia oba okna typu *ListBox*. Ponowne załadowanie innego pliku powoduje nadpisanie tablicy obiektów typu Rozkaz, a następnie analizę za pomocą metod głównych klasy Program. Metoda File_in_Click jest uruchamiana przyciskiem z opisem *Load File* oraz po kliknięciu z menu paska górnego *File → Load Assembler Directives File*.
Zapis obrazu pamięci RAM do pliku z rozszerzeniem .cdm umożliwia metoda

```
private void File_out_Click(object sender, RoutedEventArgs e) .
```

Na początku metoda ta sprawdza ilość linii w oknie z obrazem pamięci RAM i uniemożliwia zapis pliku w przypadku gdy okno to jest puste. W przeciwnym przypadku użytkownik może w trybie okienkowym (patrz zrzut ekranu nr 5) wybrać lokalizację na dysku oraz nazwę pliku, do którego obraz pamięci RAM z okna, zostanie zapisany z pominięciem linii nagłówkowej. Metoda ta jest uruchamiana przyciskiem z opisem

Save CDM File oraz po kliknięciu z menu paska górnego *File* → *Save CDM Hexadecimal File*.

Pozostałe metody klasy MainWindow.xaml wraz z krótkim omówieniem ich działania:

1) private void CloseApp_Click(object sender, RoutedEventArgs e)
   - zamyka aplikację
2) private void Enlarge_Window(object sender, RoutedEventArgs e)
   - powoduje przejście okna aplikacji ze stanu podstawowego w rozszerzony
3) private void Shrink_Window(object sender, RoutedEventArgs e)
   - powoduje przejście okna aplikacji ze stanu rozszerzonego w podstawowy
4) private void AC_Click(object sender, RoutedEventArgs e)
   - powoduje wypisanie informacji dodatkowej o skrócie AC po przejściu w rozszerzony stan okna aplikacji
5) private void MAR_Click(object sender, RoutedEventArgs e)
   - powoduje wypisanie informacji dodatkowej o skrócie MAR po przejściu w rozszerzony stan okna aplikacji
6) private void MBR_Click(object sender, RoutedEventArgs e)
   - powoduje wypisanie informacji dodatkowej o skrócie MBR po przejściu w rozszerzony stan okna aplikacji
7) private void PC_Click(object sender, RoutedEventArgs e)
   - powoduje wypisanie informacji dodatkowej o skrócie PC po przejściu w rozszerzony stan okna aplikacji
8) private void IR_Click(object sender, RoutedEventArgs e)
   - powoduje wypisanie informacji dodatkowej o skrócie IR po przejściu w rozszerzony stan okna aplikacji
9) private void InREG_Click(object sender, RoutedEventArgs e)
   - powoduje wypisanie informacji dodatkowej o skrócie InREG po przejściu w rozszerzony stan okna aplikacji
10) private void OutREG_Click(object sender, RoutedEventArgs e)
   - powoduje wypisanie informacji dodatkowej o skrócie OutREG po przejściu w rozszerzony stan okna aplikacji
11) private void JnS_Click(object sender, RoutedEventArgs e)

- powoduje wypisanie informacji dodatkowej o mnemoniku JnS po przejściu w rozszerzony stan okna aplikacji

12) private void Load_Click(object sender, RoutedEventArgs e)
    - powoduje wypisanie informacji dodatkowej o mnemoniku Load po przejściu w rozszerzony stan okna aplikacji
13) private void Store_Click(object sender, RoutedEventArgs e)
    - powoduje wypisanie informacji dodatkowej o mnemoniku Store po przejściu w rozszerzony stan okna aplikacji
14) private void Add_Click(object sender, RoutedEventArgs e)
    - powoduje wypisanie informacji dodatkowej o mnemoniku Add po przejściu w rozszerzony stan okna aplikacji
15) private void Subt_Click(object sender, RoutedEventArgs e)
    - powoduje wypisanie informacji dodatkowej o mnemoniku Subt po przejściu w rozszerzony stan okna aplikacji
16) private void Input_Click(object sender, RoutedEventArgs e)
    - powoduje wypisanie informacji dodatkowej o mnemoniku Input po przejściu w rozszerzony stan okna aplikacji
17) private void Output_Click(object sender, RoutedEventArgs e)
    - powoduje wypisanie informacji dodatkowej o mnemoniku Output po przejściu w rozszerzony stan okna aplikacji
18) private void Halt_Click(object sender, RoutedEventArgs e)
    - powoduje wypisanie informacji dodatkowej o mnemoniku Halt po przejściu w rozszerzony stan okna aplikacji
19) private void Skipcond_Click(object sender, RoutedEventArgs e)
    - powoduje wypisanie informacji dodatkowej o mnemoniku Skipcond po przejściu w rozszerzony stan okna aplikacji
20) private void Jump_Click(object sender, RoutedEventArgs e)
    - powoduje wypisanie informacji dodatkowej o mnemoniku Jump po przejściu w rozszerzony stan okna aplikacji
21) private void Clear_Click(object sender, RoutedEventArgs e)
    - powoduje wypisanie informacji dodatkowej o mnemoniku Clear po przejściu w rozszerzony stan okna aplikacji
22) private void AddI_Click(object sender, RoutedEventArgs e)

- powoduje wypisanie informacji dodatkowej o mnemoniku AddI po przejściu w rozszerzony stan okna aplikacji

23) private void JumpI_Click(object sender, RoutedEventArgs e)
    - powoduje wypisanie informacji dodatkowej o mnemoniku JumpI po przejściu w rozszerzony stan okna aplikacji
24) private void LoadI_Click(object sender, RoutedEventArgs e)
    - powoduje wypisanie informacji dodatkowej o mnemoniku LoadI po przejściu w rozszerzony stan okna aplikacji
25) private void StoreI_Click(object sender, RoutedEventArgs e)
    - powoduje wypisanie informacji dodatkowej o mnemoniku StoreI po przejściu w rozszerzony stan okna aplikacji
26) private void VarLabel_Click(object sender, RoutedEventArgs e)
    - powoduje wypisanie informacji dodatkowej o sposobie definicji zmiennej poprzez zastosowanie etykiety zmiennej wraz z wpisaniem do niej wartości po przejściu w rozszerzony stan okna aplikacji
27) private void VarAddress_Click(object sender, RoutedEventArgs e)
    - powoduje wypisanie informacji dodatkowej o sposobie definicji zmiennej poprzez zastosowanie adresu zmiennej wraz z wpisaniem do niej wartości po przejściu w rozszerzony stan okna aplikacji
28) private void Loops_Click(object sender, RoutedEventArgs e)
    - powoduje wypisanie informacji dodatkowej o sposobie zbudowania pętli po przejściu w rozszerzony stan okna aplikacji
29) private void Conditions_Click(object sender, RoutedEventArgs e)
    - powoduje wypisanie informacji dodatkowej o sposobie zbudowania instrukcji warunkowej po przejściu w rozszerzony stan okna aplikacji
30) private void About_Click(object sender, RoutedEventArgs e)
    - powoduje wypisanie informacji dodatkowej o aplikacji po przejściu w rozszerzony stan okna aplikacji
31) private void ListBox1_SelectionChanged(object sender, SelectionChangedEventArgs e)
    - po kliknięciu w obiekt znajdujący się w oknie zawierającym dyrektywy procesora powoduje zaznaczenie odpowiadającego mu obiektu z okna z obrazem pamięci RAM

32) private void ListBox2_SelectionChanged(object sender, SelectionChangedEventArgs e)
   - po kliknięciu w obiekt znajdujący się w oknie z obrazem pamięci RAM powoduje zaznaczenie odpowiadającego mu obiektu z okna zawierającego dyrektywy procesora

33) private async void BlinkingLabel(Label Name, Brush Color)
   - powoduje mruganie wybranej etykiety w wybranym kolorze

6. Pozycje literaturowe

1) “The Essentials of Computer Organization and Architecture, 4th Edition”; Julia Labour, Linda Null; 02.2014; ISBN: 9781284033144
2) *https://pl.wikipedia.org/wiki/Asembler#J%C4%99zyk_asemblera*
3) *https://en.wikipedia.org/wiki/Assembly_language#Assembly_language_syntax*
4) *https://pl.wikipedia.org/wiki/Kod_%C5%BAr%C3%B3d%C5%82owy#F#*
5) *"Kathleen Booth: Assembling Early Computers While Inventing Assembly"; 02.2019; https://hackaday.com/2018/08/21/kathleen-booth-assembling-early-computers-while-inventing-assembly/*
6) Andrew D. Booth, Kathleen H. V. Britten; 09.1947; *"General considerations in the design of an all purpose electronic digital computer (Coding for the ARC, 2nd ed)"*; Birkbeck College; London; *mt-archive.info/Booth-1947.pdf*
7) *Salomon; Assemblers and Loaders; 2012-01-17; (http://www.davidsalomon.name/assem.advertis/asl.pdf)*
8) Campbell-Kelly, Martin; "Programming the EDSAC"; IEEE Annals of the History of Computing. **2** (1): 7–36; 1980; *doi:10.1109/MAHC.1980.10009*.
9) Computer Pioneer Award 'For assembly language programming'; David Wheeler; 1985; *https://www.computer.org/web/awards/pioneer-david-wheeler*
10) *Wilkes, Maurice V.*; "The EDSAC - an Electronic Calculating Machine"; Journal of Scientific Instruments. **26** (12): 385–391; 1949; *doi:10.1088/0950-7671/26/12/301*; https://iopscience.iop.org/article/10.1088/0950-7671/26/12/301/meta
11) *"Eidolon's Inn: SegaBase Saturn"*; 07.2018; https://web.archive.org/web/20080713074116/http://www.eidolons-inn.net/tiki-index.php?page=SegaBase+Saturn
12) Wykład z Assemblera (pol.). [dostęp 2016-11-05]; http://web.archive.org/web/20111125150053/http://www.zpsb.szczecin.pl/uploads/DlaStudentow/materialy_dydaktyczne/swernikowski/asembler.pdf
13) https://pl.wikipedia.org/wiki/Program_rozruchowy
14) https://pl.wikipedia.org/wiki/Deasembler

15) https://pl.wikipedia.org/wiki/Dekompilator

16) John Markoff; *Writing the Fastest Code, by Hand, for Fun: A Human Computer Keeps Speeding Up Chips*; „The New York Times”; 11.2005; ISSN 0362-4331 [dostęp 2018-01-11] (ang.).; https://www.nytimes.com/2005/11/28/technology/writing-the-fastest-code-by-hand-for-fun-a-human-computer-keeps.html

7. Załączniki

1) Kod aplikacji *Assembler SPKFCS* w formie projektu zrealizowanego w wyskopoziomowym języku C# za pomocą środowiska kompilacyjnego Visual Studio 2015 na nośniku elektronicznym.
2) Skompilowana aplikacja *Assembler SPKFCS*