



**AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE**  
Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej

## Projekt dyplomowy

***Optymalizacja schematów blokowych algorytmicznych maszyn stanów***

*Optimization of the flowcharts representing the algorithmic state machines*

Autor:

Kierunek studiów:

Opiekun pracy:

*Wiktor Andrzej Mendalka*

Informatyka

dr hab. inż. Andrei Karatkevich, prof. AGH

Kraków, 2021/2022



*Serdecznie podziękowania opiekunowi pracy  
Profesorowi AGH, dr. inż. Andrei Karatkevich'owi  
za okazane wsparcie merytoryczne, przekazaną  
wartościową wiedzę, oraz za poświęcony czas.*



## Spis treści

<b>1. WSTĘP</b>	<b>6</b>
1.1. Cel pracy .....	6
1.2. Zakres pracy .....	6
<b>2. ALGORYTMICZNE MASZYNY STANÓW JAKO FORMALIZM OPISUJĄCY DZIAŁANIA UKŁADÓW CYFROWYCH</b>	<b>7</b>
2.1. Wprowadzenie do algorytmicznych maszyn stanów .....	7
2.2. Automaty Moore' a i Mealy' ego jako przykłady sprowadzalne do ASM.....	8
2.3. Minimalizacja liczby wierzchołków operacyjnych w algorytmicznych maszynach stanów (ASM) .....	10
<b>3. MINIMALIZACJA ASM</b>	<b>15</b>
3.1. Minimalizacja automatów skończonych .....	15
3.2. Rozpatrywane przypadki testowe .....	15
<b>4. ZAPROPONOWANE ALGORYTMY W POSTACI PSEUDOKODU</b>	<b>19</b>
4.1. Praca z plikami formatu ASM i zakres pojęć .....	19
4.2. Opis wybranych metod pomocniczych .....	20
4.3. MergeOptimization .....	22
4.4. BringOutOptimization .....	23
4.5. DuplicateOptimization .....	24
<b>5. METODA MINIMALIZACJI LICZBY WIERZCHOŁKÓW OPERACYJNYCH</b>	<b>25</b>
5.1. Opis algorytmu działania programu .....	25
5.2. Opis implementacji oraz licencje wykorzystanych prekwizytów .....	26
5.3. Instrukcja użytkownika programu .....	27
5.4. Implementacja algorytmów opisanych w rozdziałach .....	30
5.5. Testy autora .....	37
5.6. Wyniki optymalizacji .....	38
<b>6. WNIOSKI</b>	<b>41</b>
<b>BIBLIOGRAFIA</b>	<b>42</b>

# 1. Wstęp

## 1.1. Cel pracy

Celem pracy inżynierskiej jest opracowanie oprogramowania, przyjmującego jako dane wejściowe opis tekstowy pewnej algorytmicznej maszyny stanów, gdzie wierzchołki operacyjne przedstawiają zbiory mikrooperacji, a następnie optymalizującego liczbę tych wierzchołków. Działanie to pozwala na uproszczenie opisu, implementacji oraz czasu wykonywania zadanego algorytmu. Praca w zadanych wypadkach może wymagać podejścia kombinatorycznego do uzyskania lepszego lub optymalnego rozwiązania problemu. Do utworzenia danych wejściowych użyty zostanie program ASMCreator, w którym tworzony jest graficzny schemat blokowy, który następnie zostaje przekonwertowany do plików tekstowych – służących jako dane wejściowe do opracowywanego oprogramowania.

## 1.2. Zakres pracy

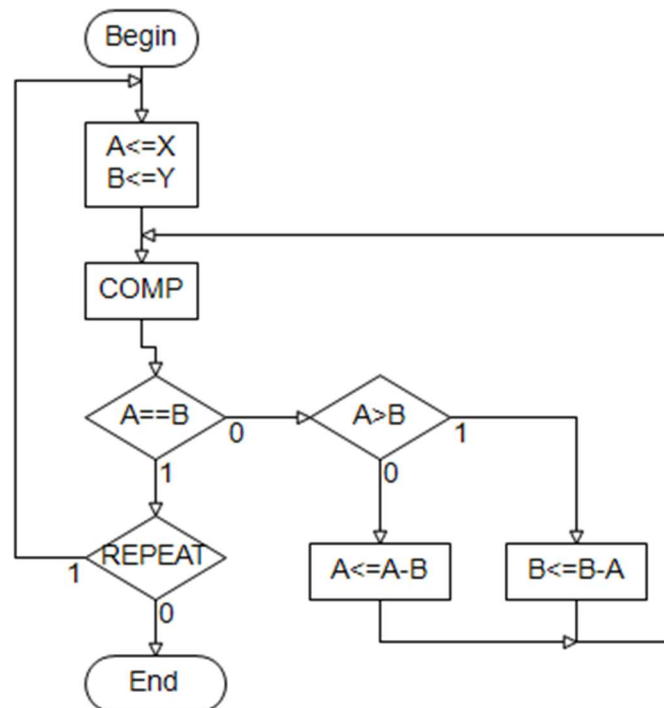
Praca ma następującą strukturę:

- Rozdział pierwszy zawiera wstęp opisujący cel i strukturę niniejszej pracy
- Rozdział drugi opisuje niezbędne pojęcia teoretyczne, potrzebne czytelnikowi do zrozumienia tematu pracy
- Rozdział trzeci przedstawia przykłady schematów blokowych, które zostały poddane optymalizacji
- Rozdział czwarty przedstawia zaproponowane przez autora algorytmy w postaci pseudokodu
- Rozdział piąty prezentuje obraną metodę, wykorzystaną do realizacji tematu pracy oraz analizę otrzymanych wyników
- Rozdział szósty zawiera wnioski ogólne podsumowujące efekty pracy

## 2. Algorytmiczne maszyny stanów jako formalizm opisujący działania układów cyfrowych

### 2.1. Wprowadzenie do algorytmicznych maszyn stanów

Algorytmiczna maszyna stanów (ASM) to jeden z modeli formalnej specyfikacji systemów sterowania tuż obok innych typów takich jak skończony automat cyfrowy FSM czy też sieci Petriego [1]. Jej głównym zastosowaniem jest projektowanie systemów cyfrowych, algorytmika oraz tworzenie systemów sterowania opartych na automatach. W ten sposób utworzona maszyna może być przekonwertowana do dowolnego języka służącego do opisu sprzętu takiego jak VHDL, a następnie poddana symulacji. Algorytmiczna maszyna stanów ma postać graficzną, dzięki czemu zyskuje się większą czytelność, a co za tym idzie – nie tylko szybsze zrozumienie realizowanego programu, ale także wychwycenie potencjalnych błędów. Strukturalnie składa się z wierzchołków startowego, wyjściowego, operacyjnych (w postaci prostokątów) oraz decyzyjnych (w postaci rombów). Wierzchołki te tworzą graf skierowany, dzięki czemu widoczny jest przebieg działania algorytmu. Wierzchołki operacyjne przedstawiają akcje wyjściowe w systemie, natomiast wierzchołki decyzyjne odpowiadają za zmiany stanów w zależności od spełnienia (bądź nie) opisanego w tym bloku warunku logicznego. Rys. 1. przedstawia przykładową algorytmiczną maszynę stanów, reprezentującą działanie algorytmu polegającą na wyznaczeniu największego wspólnego dzielnika dwóch liczb. Składa się ona z 3 stanów – IDLE – jako startowy, COMP oraz DONE – jako końcowy.



Rys. 1. Przykładowa algorytmiczna maszyna stanów [2]

## 2.2. Automaty Moore' a i Mealy' ego jako przykłady sprowadzalne do ASM

Istnieją dwa charakterystyczne typy automatów : Moore'a oraz Mealy'ego. Pierwszy z nich to taki, którego wyjście determinowane jest tylko poprzez stany wewnętrzne, natomiast drugi z nich korzysta ponadto z zadanych sygnałów wejściowych. Automat Moore' a zapisuje się jako uporządkowaną szóstkę (rys. 2) :

$$\langle Z, Q, Y, \Phi, \Psi, q_0 \rangle$$

gdzie:

$Z$  – zbiór sygnałów wejściowych

$Q$  – zbiór sygnałów wewnętrznych

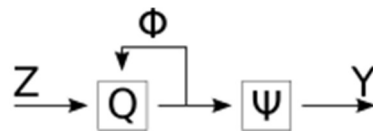
$Y$  – zbiór sygnałów wyjściowych

$\Phi$  – funkcja przejść,  $q(t + 1) = \Phi[q(t), z(t)]$

$\Psi$  – funkcja wyjść,  $y(t) = \Psi[q(t)]$ ,

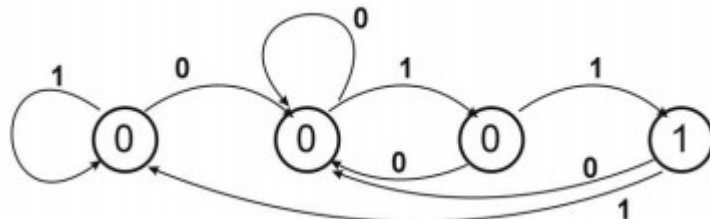
( zależy tylko od stanu, w którym znajduje się automat)

$q_0$  – stan początkowy, należy do zbioru  $Q$



Rys. 2. Ogólna postać automatu Moore'a

Rys. 3. przedstawia przykładowy automat Moore'a, akceptujący sekwencję 011



Rys. 3. Automat Moore'a wykrywający sekwencję 011



Z kolei drugi z omawianych automatów – Mealy’ego zapisywany jest jako uporządkowaną szóstkę (rys.4.):

$$\langle Z, Q, Y, \Phi, \Psi, q_0 \rangle$$

gdzie:

$Z$  – zbiór sygnałów wejściowych

$Q$  – zbiór sygnałów wewnętrznych

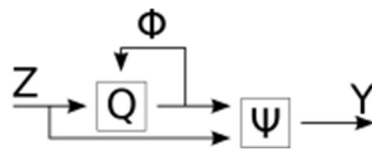
$Y$  – zbiór sygnałów wyjściowych

$\Phi$  – funkcja przejść,  $q(t + 1) = \Phi[q(t), z(t)]$

$\Psi$  – funkcja wyjść,  $y(t) = \Psi[q(t), z(t)]$ ,

(zależy od stanu, w którym znajduje się automat oraz sygnału wejściowego)

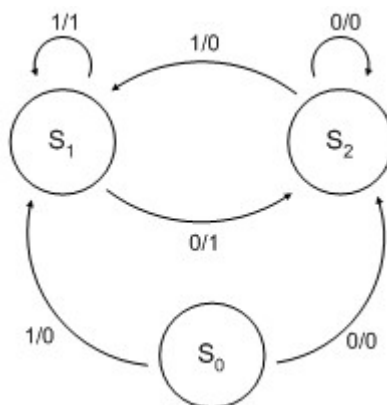
$q_0$  – stan początkowy, należy do zbioru  $Q$



Rys. 4. Ogólna postać automatu Mealy’ego

Automat ten przedstawiany jest jako skierowany graf wraz z pewnym wyróżnionym stanem – stanem początkowym. Dzięki różnym konfiguracjom stanów wejściowych możliwe jest wygenerowanie różnych sygnałów wyjściowych.

Rys. 5 przedstawia przykładowy algorytm Mealy’ego.



Rys. 5. Przykładowy automat Mealy’ego [3]

Zasadniczą różnicą jest fakt, że funkcja przejść opisana jest przy pomocy zapisu Z/Y. Oznacza to, że stan wyjść Y automatu zależy od stanu wewnętrznego automatu Q oraz stanu wejść Z. W konsekwencji liczba stanów wewnętrznych Q może być mniejsza w porównaniu do automatu Moore'a, gdyż ten sam stan Q może wystąpić dla różnych stanów wyjść Y. Synteza takich automatów polega na utworzeniu układu logicznego, który przedstawia działanie automatu. Działanie to może pozwolić na optymalizację takich układów. Cała procedura składa się z kilku kroków a jej działanie opiera się na minimalizacji funkcji logicznych. Należy zakodować sygnały wejściowe, wyjściowe oraz stany automatu przy pomocy funkcji boolowskich. Na ich podstawie tworzona jest tabela przejść. Dzięki temu możliwa jest synteza takiego układu kombinacyjnego wykorzystując przerzutniki np. D, T lub JK. Przykładowo - korzystając kolejno z metody tablic Karnaugh można otrzymać przede wszystkim zoptymalizowane postaci funkcji wzbudzeń przerzutników i funkcji wyjściowych, co kończy procedurę optymalizacji.

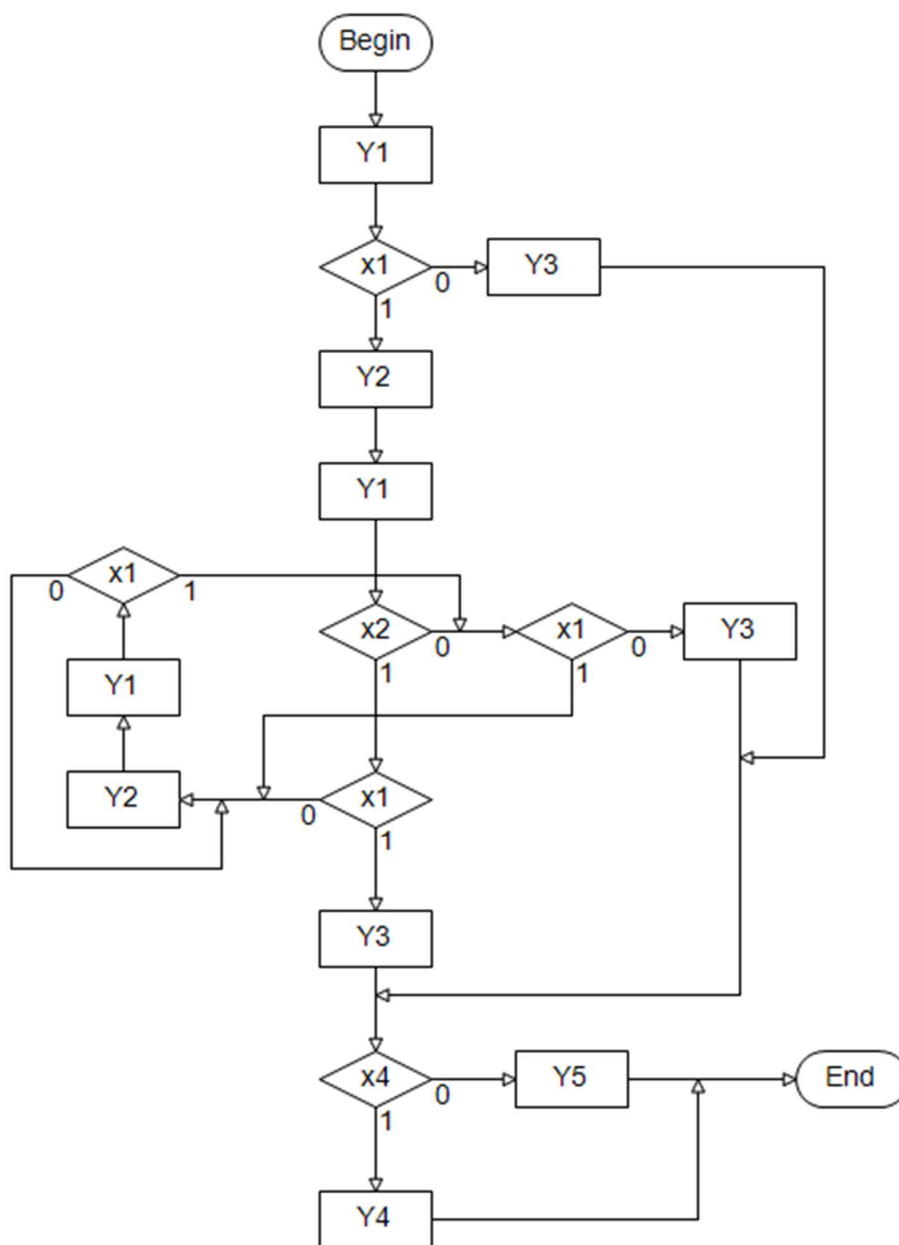
### 2.3. Minimalizacja liczby wierzchołków operacyjnych w algorytmicznych maszynach stanów (ASM)

Rys. 6. przedstawia przykład automatu, poddanego optymalizacji polegającej na zmniejszeniu liczby wierzchołków operacyjnych. Minimalizacja ASM polega na utworzeniu odpowiadającego mu automatu Moore'a i jego optymalizacji, a następnie powrocie do pierwotnej postaci. W pierwszym kroku należy oznaczyć wierzchołki operacyjne jako symbole  $a_1, a_2, \dots, a_M, a_e$ , gdzie  $a_1$  reprezentuje bloczek *Begin*,  $a_M$  ostatni bloczek przed  $a_e$  – bloczkiem końcowym *End*. Kolejno dzięki tak przetworzonym danych, konstruuje się tablicę przejść automatu Moore'a. Aby zdefiniować wspomniane przejścia korzysta się z reguły:

$$a_m x_{m1}^{e_{m1}} \dots x_{mR}^{e_{mR}} a_s.$$

Indeks górny  $e_{mr} = 1$ , oznacza przejście z wierzchołka  $a_m$  do wierzchołka  $a_s$  poprzez wierzchołek warunkowy  $x_{m1}$  przez wejście „1” natomiast zapis  $e_{mr} = 0$ , odpowiednia przez wejście „0”. Przyjęto uproszczony zapis tj. zamiast  $x_i^1 \rightarrow x_i$  oraz  $x_i^0 \rightarrow \bar{x}_i$ . Na rys. 6. widoczne jest, że w niektórych przypadkach przejście pomiędzy wierzchołkami operacyjnymi wymaga kilku przejść przez wierzchołki warunkowe. Wobec tego możliwe jest zapisanie formuły ogólnej opisującej przejście między dwoma stanami  $a_m$  oraz  $a_s$ :

$$X(a_m, a_s) = \bigvee_{r=1}^R x_{mr}^{e_{mr}}$$



Rys. 6. Przykładowy automat ASM przed optymalizacją

Jako rezultat otrzymuje się automat Moore'a z taką samą liczbą stanów co symboli opisujących dany ASM. Dzięki temu można zapisać tabelę przejść automatu Moore'a realizującego zadany ASM (tab. 1.). Jest oczywistym, że nie ma żadnych przejść ze stanu końcowego  $a_e$ , co jest widoczne w ostatnim rzędzie wspomnianej tabeli. Stan końcowy automatu jest bowiem utożsamiany ze stanem początkowym.

Tab. 1. Tabela przejść automatu Moore'a

$a_m$	$Y(a_m)$	$a_s$	$X(a_m, a_s)$	$h$
$a_1$	$Y_b$	$a_2$	1	1
$a_2$	$Y_1$	`	$x_1$	2
			$\overline{x_1}$	3
$a_3$	$Y_2$	$a_5$	1	4
$a_4$	$Y_3$	$a_{10}$	$x_4$	5
		$a_{11}$	$\overline{x_4}$	6
$a_5$	$Y_1$	$a_6$	$x_2 x_1$	7
		$a_6$	$\overline{x_2} x_1$	8
		$a_7$	$x_2 \overline{x_1}$	9
		$a_8$	$\overline{x_2} \overline{x_1}$	10
$a_6$	$Y_2$	$a_9$	1	11
$a_7$	$Y_3$	$a_{10}$	$x_4$	12
		$a_{11}$	$\overline{x_4}$	13
$a_8$	$Y_3$	$a_{10}$	$x_4$	14
		$a_{11}$	$\overline{x_4}$	15
$a_9$	$Y_1$	$a_6$	$x_1$	16
		$a_8$	$\overline{x_1}$	17
$a_{10}$	$Y_4$	$a_e$	1	18
$a_{11}$	$Y_5$	$a_e$	1	19
$a_e$	$Y_e$	$a_e$	1	20

Mając tak przygotowane dane, można przystąpić do procesu optymalizacji. W pierwszym kroku należy znaleźć kolejne podziały  $\pi_0, \pi_1 \dots \pi_{k+1} = \pi_k = \pi$ , gdzie  $\pi$  jest kolejnym podziałem oznaczającym równoważne bloki. Stany przedstawione w tab. 1 posiadające takie same wyjścia są nazwane jako 0-równoważne, a ich postać jest przedstawiona jako  $\pi_0$ , a odpowiednie elementy tego wektora są zebrane w wektory  $A_0 - A_6$ :

$$\pi_0 = \overline{a_1}, \overline{a_2}, \overline{a_5}, \overline{a_9}, \overline{a_3}, \overline{a_6}, \overline{a_4}, \overline{a_7}, \overline{a_8}, \overline{a_{10}}, \overline{a_{11}}, \overline{a_e} = \{A_0, A_1, A_2, A_3, A_4, A_5, A_6\}$$

Tak posegregowane dane zostały przedstawione w tab. 2.

Tab. 2. Uporządkowana tabela danych wg wektora  $\pi_0$

$A_i$	$a_m$	$Y(a_m)$	$A_j$	$X(a_m, A_j)$
$A_0$	$a_1$	$Y_b$	$A_1$	1
$A_1$	$a_2$	$Y_1$	$A_2$ $A_3$	$x_1$ $\overline{x_1}$
	$a_5$	$Y_1$	$A_2$ $A_3$ $A_2$ $A_3$	$x_2 x_1$ $x_2 \overline{x_1}$ $\overline{x_2} x_1$ $\overline{x_2} \overline{x_1}$
	$a_9$	$Y_1$	$A_2$ $A_3$	$x_1$ $\overline{x_1}$
$A_2$	$a_3$	$Y_2$	$A_1$	1
	$a_6$	$Y_2$	$A_1$	1
$A_3$	$a_4$	$Y_3$	$A_4$ $A_5$	$x_4$ $\overline{x_4}$
	$a_7$	$Y_3$	$A_4$ $A_5$	$x_4$ $\overline{x_4}$
	$a_8$	$Y_3$	$A_4$ $A_5$	$x_4$ $\overline{x_4}$
$A_4$	$a_{10}$	$Y_4$	$A_6$	1
$A_5$	$a_{11}$	$Y_5$	$A_6$	1
$A_6$	$a_e$	$Y_e$	$A_6$	1

Dwa stany  $a_i$  oraz  $a_j$  są k-równoważne, gdy są (k-1) – równoważne oraz prowadzą do tych samych bloków  $\pi_{k-1}$  dla tych samych wejść. Korzystając z tab. 2 można zapisać :

$$\pi_1 = \overline{a_1}, \overline{a_2}, \overline{a_5}, \overline{a_9}, \overline{a_3}, \overline{a_6}, \overline{a_4}, \overline{a_7}, \overline{a_8}, \overline{a_{10}}, \overline{a_{11}}, \overline{a_k}.$$

Zatem podziały  $\pi_1$  oraz  $\pi_1$  są sobie równe, co oznacza, że wystarczy wziąć po jednym elemencie z każdego bloku  $\pi_1$ , aby uzyskać minimalny zbiór stanów  $A'$  np:

$$A' = \{a_1, a_2, a_3, a_4, a_{10}, a_{11}, a_e\}$$

Zoptymalizowana postać przedstawiona została w tab. 3. Zawiera ona minimalną liczbę wierzchołków operacyjnych, niezbędnych do realizacji zadanego automatu na wstępie zadania.

Tab. 3. Zoptymalizowany automat Moore'a

$a_m$	$Y(a_m)$	$a_s$	$X(a_m, a_s)$	$h$
$a_1$	$Y_b$	$a_2$	1	1
$a_2$	$Y_1$	$a_3$	$x_1$	2
		$a_4$	$\overline{x_1}$	3
$a_3$	$Y_2$	$a_2$	1	4
$a_4$	$Y_3$	$a_{10}$	$x_4$	5
		$a_{11}$	$\overline{x_4}$	6
$a_{10}$	$Y_4$	$a_e$	1	7
$a_{11}$	$Y_5$	$a_e$	1	8
$a_e$	$Y_e$	$a_e$	1	9

Podsumowując, aby utworzyć zminimalizowany automat, należy go podzielić na poniższe formuły przejść:

$$Y_b \rightarrow Y_1, Y_2 \rightarrow Y_1$$

$$Y_1 \rightarrow x_1 Y_2 \vee \overline{x_1} Y_3$$

$$Y_3 \rightarrow x_4 Y_4 \vee \overline{x_4} Y_5$$

$$Y_4 \rightarrow Y_e, Y_5 \rightarrow Y_e$$

## 3. Minimalizacja ASM

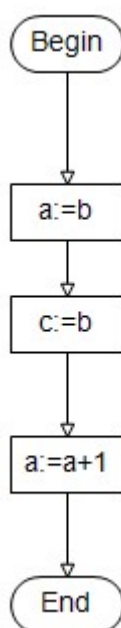
### 3.1. Minimalizacja automatów skończonych

Optymalizacja automatów skończonych polega przede wszystkim na zmniejszeniu liczby wierzchołków operacyjnych lub warunkowych. Dzięki temu część czynności może wykonać się szybciej (w mniejszej liczbie taktów procesora). Oczywiście jest, że jest to zadanie korzystne, zwłaszcza dla bardzo skomplikowanych i rozbudowanych programów. Przedmiotem tej pracy jest optymalizacja liczby wierzchołków operacyjnych. Działanie to potencjalnie może również zwiększyć możliwości optymalizacji liczby wierzchołków decyzyjnych.

Rozwiązania zaproponowane w tej pracy uwzględniają zastosowanie kilku prostszych metod optymalizacji (opisanych w dalszych rozdziałach) zamiast metody klasycznej; oferuje to dodatkowe korzyści w postaci optymalizacji elementów nieobjętych przez metodę klasyczną. Przykładem tego jest łączenie wierzchołków z różnymi operacjami.

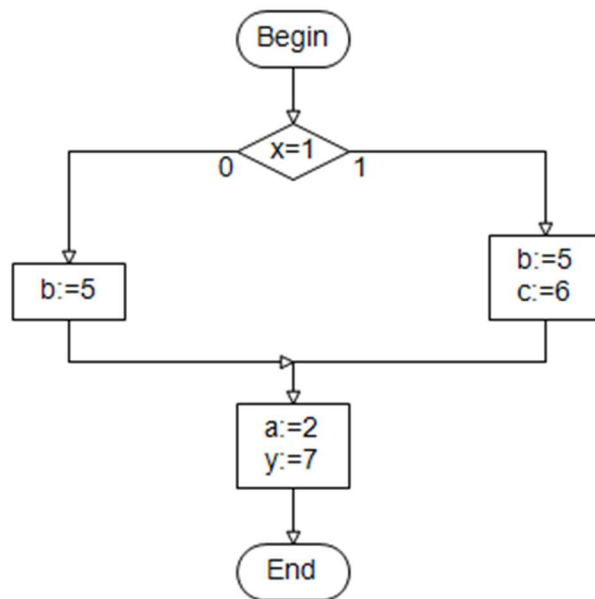
### 3.2. Rozpatrywane przypadki testowe

Poniżej zaprezentowano pierwsze przypadki testowe, uporządkowane względem skali złożoności, pozwalające na sprawdzenie działania przygotowywanego algorytmu. Jest to istotne, gdyż pewność ich działania pozwoli na podjęcie dużo bardziej skomplikowanych problemów. Dodatkowo, przypadki 1-3 zawierają kolejno problemy realizowane przez algorytmy demonstrowane w niniejszej pracy [4].



Rys. 7. Schemat blokowy nr 1

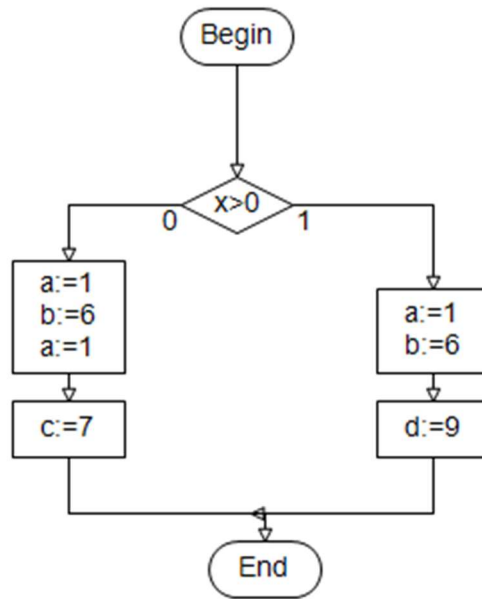
Schemat blokowy nr 1 (rys. 7) nie zawiera wierzchołków decyzyjnych. Można domniemywać, że będzie możliwe połączenie operacji pierwszej z drugą lub drugą z trzecią, lecz wszystkich naraz nie, gdyż próba jednoczesnego przypisania do zmiennej „a” dwóch różnych wartości spowoduje niejednoznaczność wyniku końcowego. Ten sposób działania odpowiada pierwszej z zaproponowanych optymalizacji tj. „MergeOptimization”



Rys. 8. Schemat blokowy nr 2

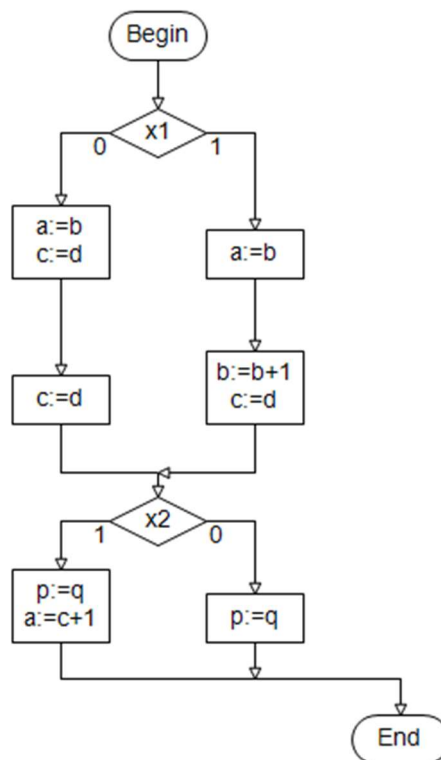
Schemat blokowy nr 2 (rys. 8) choć zawiera wierzchołek decyzyjny, to nie będzie on póki co brany pod uwagę. Można dostrzec, że operacja „b:=5” może zostać przeniesiona przed niego, gdyż wykona się niezależnie od jego wyniku. Ten sposób optymalizacji odpowiada kolejnemu z zaproponowanych algorytmów, tj. „BringOutOptimization”



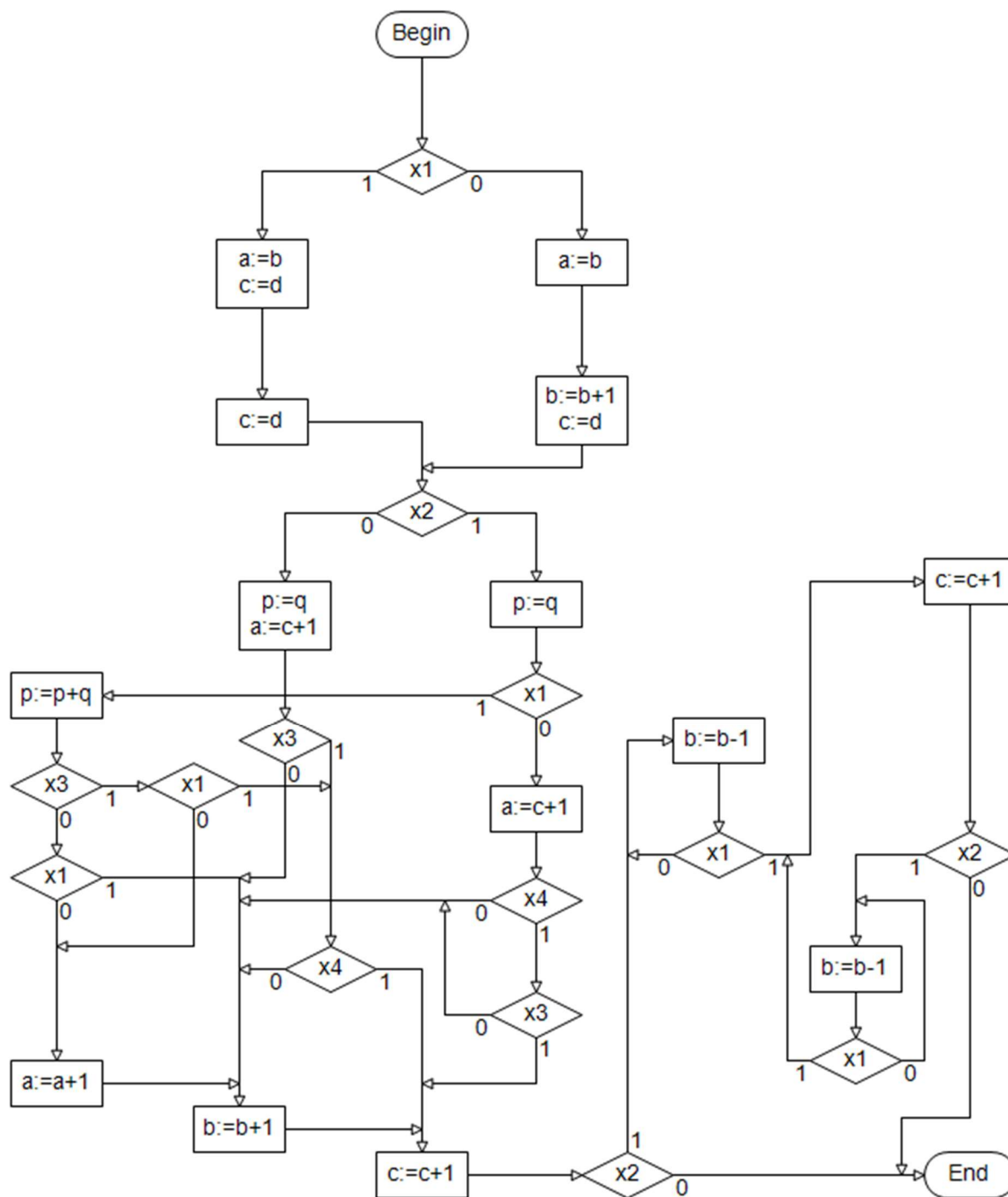


Rys. 9. Schemat blokowy nr 3

Schemat blokowy nr 3 (rys. 9) przedstawia problemy zawarte w schematach blokowych nr 1-2, wprowadzając jednocześnie nowy typ – w lewym bloku możemy znaleźć zduplikowaną operację „a:=1”, za której usunięcie odpowie algorytm „DuplicateOptimization”



Rys. 10. Schemat blokowy nr 4



Rys. 11. Schemat blokowy nr 5

Schematy blokowe zawarty na Rys. 10 i na Rys. 11 stanowią problemy będące miksami poprzednich. Ich poprawne rozwiązanie będzie stanowić punkt startowych do bardziej złożonych programów.

## 4. Zaproponowane algorytmy w postaci pseudokodu

### 4.1. Praca z plikami formatu ASM i zakres pojęć

Jak zostało wspomniane wcześniej, produktem kompilacji diagramu w programie ASMCreator są 3 pliki w formatach \*.gsa, \*.txt, \*.mic

Plik \*.gsa zawiera zarówno definicje wszystkich bloków warunkowych oraz wierzchołków operacyjnych w diagramie, łącznie z przypisanymi do nich indeksami i instrukcjami, jak i opis przejść między nimi.

Przykładowa linia z pliku \*.gsa może wyglądać następująco:

„4 Y4 2 0„

Opisuje ona blok o indeksie 4 i przypisanej instrukcji Y4, który posiada jedno dziecko/wyjście o indeksie 2. Ze względu na fakt posiadania jednego dziecka, wiemy że jest to blok warunkowy – wyjątkiem będzie jednak zawsze blok End, który jest specjalnym przypadkiem i jako jedyny w każdym diagramie nie posiada żadnego dziecka. Instrukcje opisują zbiór operacji zawartych w danym bloku warunkowym i zapisane są w formacie „Y” + liczba – wyjątkiem są bloki początkowe wraz z końcowymi, których instrukcjami są odpowiednio [Ss]tart oraz [Ee]nd.

W przypadku wierzchołków warunkowych, są one zapisane w formacie „x” + liczba i stanowią bezpośrednio odwołanie do pojedynczej funkcji logicznej.

Plik \*.txt podzielony jest na trzy sekcje:

„Micro Instructions”, zawierającej definicję listy operacji przypisanych do danej instrukcji.  
Przykład: „Y3 = y3 y4”

„Micro Operations”, opisującej kolejno symbole przypisane do danych operacji

Przykład: „y3 : a:=1”

„Logical Conditions”, definiujących kolejne funkcje logiczne wierzchołków warunkowych.

Przykład: „x1 : x>0”

Plik \*.mic nie wprowadza żadnych nowych informacji, zawiera tylko zduplikowane informacje z sekcji „Micro Instructions” pliku \*.txt

## 4.2. Opis wybranych metod pomocniczych

**PrepareInstruction(line)** – jako, że program ASMCreator przypisuje tą samą instrukcję dla różnych bloków operacyjnych zawierającą identyczną listę operacji, istnieje ryzyko niezamierzonego wpływu na treść innych bloków operacyjnych podczas modyfikacji zawartości instrukcji. Funkcja ta pomaga zapobiec takiemu zachowaniu – w przypadku, kiedy zamierzamy zmodyfikować zawartość danej linii pliku Gsa, należy wywołać funkcję **PrepareInstruction(...)** – zwróci ona nazwę instrukcji danej linii, kiedy nie wykryje zachodzenia jej współdzielenia z innym blokiem. W przeciwnym wypadku, stworzy nową instrukcję do której zostaną skopiowane poprzednie operacje oraz zmodyfikuje przekazaną linię tak, by używała nowej instrukcji. Wartością zwracaną będzie nowo wygenerowana instrukcja.

**GetChildren(line)** – zwraca tablicę zawierającą linie pliku Gsa, do których można bezpośrednio przejść z linii podanej jako argument. Wartością wynikową jest zbiór 0-2 elementów diagramu:

- 0 dla bloku operacyjnego End,
- 1 dla każdego innego bloku operacyjnego,
- 2 dla wierzchołków warunkowych.

**GetPaths()** – algorytm odnajdujący wszystkie ścieżki w zadanym diagramie. Jako ścieżkę rozumie się kilka występujących po sobie bloków operacyjnych rozgraniczonych przez wierzchołki operacyjne i/lub bloki start/end

Proponowana implementacja algorytmu za pomocą pseudokodu:

```
1. GetPaths()
2.   paths = GsaPath[]
3.   arr = int[]
4.
5.   element = next line after the start line in gsa
6.   currentPath = new GsaPath(index of element)
7.
8.   EnterPath(line: GsaLine)
9.     if line has one child
10.       add line to currentPath
11.       EnterPath(child of line)
12.     else
13.       if currentPath length greater than 0
14.         add currentPath to paths
15.       end if
16.
17.       if arr contains index of line
18.         return
19.       end if
20.
21.       add index of line to arr
22.
23.       for child in distinct children of line
24.         current = new GsaPath(line.Index)
25.         EnterPath(child)
26.       end
27.     end if
```

```
28.     end
29.
30.     return paths
31. end
```

**GetParallelPaths()** – funkcja ta grupuje ścieżki, które są rezultatem wywołanie metody **GetPaths()** na podstawie wierzchołka warunkowego będącego ich rodzicem. Funkcja ta odnajduje pary ścieżek, które rozpoczynają się we wspólnym wierzchołku warunkowym. Rezultatem działania tej funkcji jest lista par takich ścieżek. Dodatkową funkcjonalnością w niej zawartą jest wykluczenie części wspólnej ścieżek w danej parze, tj. jeśli w pewnym momencie przebiegu ścieżek następuje ich połączenie, część wspólna ścieżek nie będzie uwzględniona w zwróconym wyniku.

**IsCompatible(operation, operations)** – funkcja determinująca, czy operacja podana jako pierwszy argument może znajdować się w jednej instrukcji z operacjami pozostałymi. Przykłady wykonania:

```
IsCompatible(„a:=b”, new[] { „b:=b+1”, „c:=d”, „p:=q” }) == true
```

W tym przykładzie istnieje tylko jedna operacja przypisania do zmiennej „a”, więc operacje są ze sobą kompatybilne

```
IsCompatible(„b:=5”, new[] { „b:=a” }) == false
```

Po połączeniu tych operacji powstałaby instrukcja, w której nastąpiłaby nieznacność wyniku końcowego, w związku z czym funkcja ta zwraca wynik **false**

**GetCompatible(operations)** – funkcja zwracająca te operacje, które spełniają powyższy predykat **IsCompatible(...)**. Jako drugi argument wywołania użyty będzie cały zbiór przekazanych operacji, z wykluczeniem tej aktualnie weryfikowanej.

### 4.3. MergeOptimization

```
1. for path in GetPaths()
2.   for i = 1 to path length
3.     children = all children of every lane in gsa file
4.
5.     if count of children equal to path[i] is greater than 1
6.       continue
7.     end if
8.
9.     oldOperations = get operations for the instruction of path[i - 1]
10.    newOperations = get operations for the instruction of path[i]
11.
12.    arr = string[]
13.
14.    for operation in newOperations
15.      if IsCompatible(operation, oldOperations)
16.        add operation to arr
17.      end if
18.    end for
19.
20.    if arr length equals 0
21.      continue
22.    end if
23.
24.    instr1 = PrepareInstruction(path[i])
25.    instr2 = PrepareInstruction(path[i - 1])
26.
27.    set operations of instruction instr1 to difference of newOperations and arr
28.    set operations of instruction instr2 to union of newOperations and arr
29.  end for
30. end for
```

Celem **MergeOptimization** jest odnalezienie i połączenie ze sobą następujących po sobie operacji, kiedy jest to możliwe.

Zawieranie większej ich ilości w pojedynczej instrukcji będzie prowadziło do sytuacji, w których inne instrukcje mogą nie zawierać już żadnej operacji, a za czym idzie okazać się zbędne.

W pierwszym kroku funkcja wyszukuje w schemacie następujące bezpośrednio po sobie instrukcje – nie są rozważane te, do których następuje przejście na więcej sposobów, niż tylko od poprzedniej instrukcji, ze względu na brak możliwości dokonania połączenia w takiej sytuacji.

Następnie, następuje proces scalania instrukcji. Przeniesione zostają tylko te operacje, które spełniają predykat **IsCompatible(...)**. Pozwala to uniknąć sytuacji, w której w instrukcji znajdują się operacje powodujące niejednoznaczność wyniku.

## 4.4. BringOutOptimization

```
1. for paths in GetParallelPaths()
2.     op1 = all operations of all lines in paths[0]
3.     op2 = all operations of all lines in paths[1]
4.
5.     inter = intersection of op1 and op2
6.
7.     if inter length equals 0
8.         continue
9.     end if
10.
11.    for line in every line of every path in paths
12.        op = get operations of line
13.        set operations of PrepareInstruction(line) to difference of op and inter
14.    end for
15.
16.    add a new instruction before the parent of paths[0] with inter as the operation
    list
17. end for
```

**BringOutOptimization** znajduje identyczne operacje w równoległe położonych do siebie ścieżkach i umieszcza je w nowej instrukcji przed wierzchołkiem warunkowym wspólnym dla obu ścieżek.

Przebieg funkcji polega na wybraniu wspólnych operacji ścieżek równoległych zwróconych przez funkcję **GetParallelPaths()**, usunięciu ich z obu ścieżek przy uwzględnieniu, że mogą występować duplikaty jednej operacji i ostatecznie, umieszczeniu ich w nowo utworzonej instrukcji.

Nowa instrukcja będzie następnie podlegała inspekcji dla pozostałych zdefiniowanych optymalizacji.

## 4.5. DuplicateOptimization

```
1. for path in GetPaths()
2.     reverse path
3.     arr = string[]
4.
5.     for line in path
6.         op = get operations of line
7.
8.         for operation in GetCompatible(op)
9.             if not arr contains operation
10.                add operation to arr
11.                continue
12.            end if
13.
14.            instr = PrepareInstruction(line)
15.
16.            list = get operations of instr
17.            remove operation from list once
18.
19.            op = get operations of line
20.            set operations of instr to list
21.        end for
22.    end for
23. end for
```

Optymalizacja **DuplicateOptimization** odpowiada za usunięcie zbędnych, zduplikowanych operacji w ramach danej ścieżki.

Kluczowe dla poprawnego działania tej metody optymalizacji jest wyznaczenie, które z zawartych w ścieżce operacji są zbędne. W tym celu używany jest wariant metody **GetCompatible(...)**, który za argument przyjmuje tablicę tak, by móc pod rozważania poddać zawartość całej ścieżki.

Funkcja rozpoczyna swoje działanie od odwrócenia zawartości ścieżki ze względu na konieczność rozważania kolejnych instrukcji od początku do końca – brak wykonania tego kroku mógłby oznaczać sytuację, w której pominięta zostanie jedna z operacji, gdy ścieżka nie jest wykonywana od początku.

Działaniem, które podejmuje ta metoda jest przechodzenie po ścieżce i zapamiętywanie, które z operacji określonych przez funkcję **GetCompatible(...)** zostały już napotkane. Jeśli dana operacja wystąpiła już wcześniej, jest ona usuwana z bieżącej instrukcji. Podobnie, jak w przypadku **BringOutOptimization**, istotne jest zwrócenie uwagi na występowanie duplikatów tak, by w przypadku instrukcji zawierającej dwa lub więcej wystąpień identycznej operacji usunąć ją tylko raz.



## 5. Metoda minimalizacji liczby wierzchołków operacyjnych

### 5.1. Opis algorytmu działania programu

Algorytm postępowania w projekcie będzie dość prosty. W pierwszym kroku należy narysować przypadki testowe w programie ASMCreator i skonwertować je do ujednoczonej postaci tekstowej tj. do 3 plików z rozszerzeniami \*.gsa, \*.txt oraz \*.mic – stanowiących dane wejściowe do algorytmu. Implementowany program będzie realizować optymalizację ASM, lecz w tej części pracy skupiono się głównie na opisie minimalizacji liczby wierzchołków operacyjnych. Stanowiąc zatem będzie połowę innego – zajmującego się minimalizacją liczby wierzchołków warunkowych. Oba podprogramy będą wykonywać się naprzemiennie, aż do momentu osiągnięcia braku poprawy rozwiązania. Program, ze względu na uproszczenie nie został zaimplementowany na podstawie klasycznych metod minimalizacji liczby stanów automatu skończonego opisanego w rozdziale 2.3. Klasyczna metoda optymalizacji pozwoliłaby jednak na głębszą optymalizację. Rezultatem wyjściowym jest – zgodnie z opcjami oferowanymi przez utworzony, dalej opisany w pracy program optymalizujący - zapis do plików zminimalizowanego schematu w postaci tekstowej i opcjonalnie, równoczesny wydruk zawartości owych plików do konsoli.

Kolejność realizacji programu jest następująca: w pierwszym kroku wykonywane są operacje związane z odczytem parametrów wejściowych oraz określeniem, z których plików program pobierze dane wejściowe. Po odczytaniu zawartości plików następuje ich wstępna walidacja, a program kontynuuje swoją pracę tylko w przypadku stwierdzenia poprawności struktury plików o formatach \*.gsa, \*.txt, \*.mic. W przeciwnym razie, następuje wypisanie do konsoli stosownego komunikatu o błędzie i przerwanie programu.

Wszystkie operacje I/O znajdują się w plikach źródłowych **Program.cs** oraz **AsmOptimizer.cs**. Ze względu na ich charakter czysto implementacyjny i niską wartość merytoryczną względem omawianego tematu, szczegóły na ich temat są pominięte w tej pracy.

Następnie z pliku **Optimization.cs** zostaje załadowana lista operacji optymalizujących. Wczytywane są zarówno operacje optymalizujące liczbę wierzchołków operacyjnych jak i warunkowych w powiązanej, wcześniej wspomnianej w pracy naprzemienności. Następuje wykonanie metody **Optimize(..)**, która jest zdefiniowana w pliku **AsmOptimizer.cs** i odpowiada za uruchamianie kolejnych przebiegów optymalizacyjnych, polegających na sekwencyjnym uruchamianiu wczytanych uprzednio operacji optymalizacyjnych. Zatrzymanie wykonania następuje w momencie wykonania przebiegu, który nie spowodował zmian w żadnym z optymalizowanych plików.

Po wykonaniu wszystkich działań związanych z optymalizacją zawartości plików, następuje reindeksacja elementów zdefiniowanych w pliku \*.gsa.

Ostatecznie, następuje wywołanie adekwatnych metod związanych z zapisem wyników do konsoli lub do wybranego przez użytkownika pliku.

Praca danego algorytmu optymalizującego polega w głównej mierze na wywoływaniu w odpowiedniej kolejności licznych metod pomocniczych, zdefiniowanych w plikach klas dziedziczących po klasie abstrakcyjnej **AsmFile.cs**. Klasy te, m.in. klasa **GsaFile** posiadają funkcje definiujące odpowiednie operacje transformujące zawartość konkretnych plików struktury ASM. Pozwala to nie tylko na oddzielenie warstwy algorytmicznej danych optymalizacji od szczegółów implementacyjnych konkretnych plików struktury ASM, ale także mniejsza objętość kodu i ułatwia możliwość przeprowadzania testów.

## 5.2. Opis implementacji oraz licencje wykorzystanych prerekwizytów

Program optymalizujący napisany został w języku C#, podobnie do wyżej wspomnianego programu ASMCreator, jednakże używając aktualniejszej platformy .NET 6.

Jest to wersja LTS frameworku, która będzie wspierana do listopada 2024r., po którym to okresie będzie wciąż możliwe uruchomienie programu, jednak zalecane będzie już użycie aktualniejszej wersji .NET.

Środowisko .NET jest udostępnione na licencji MIT zezwalającej na jej użycie w tej pracy. Kod .NET jest otwarty i znajduje się na platformie GitHub: <https://github.com/dotnet>

Użyta została jedna biblioteka „**System.CommandLine**”, która nie jest częścią frameworku, ale podobnie do niego jest udostępniona na licencji MIT pod tym samym linkiem i dzieli wspólnego twórcę – firmę Microsoft

Użycie .NET pozwalało na stworzenie wieloplatformowej aplikacji o niewielkim rozmiarze (< 1 MB)

Kod aplikacji znajduje się na repozytorium platformy GitHub pod linkiem: <https://github.com/Wiciaki/Opti>

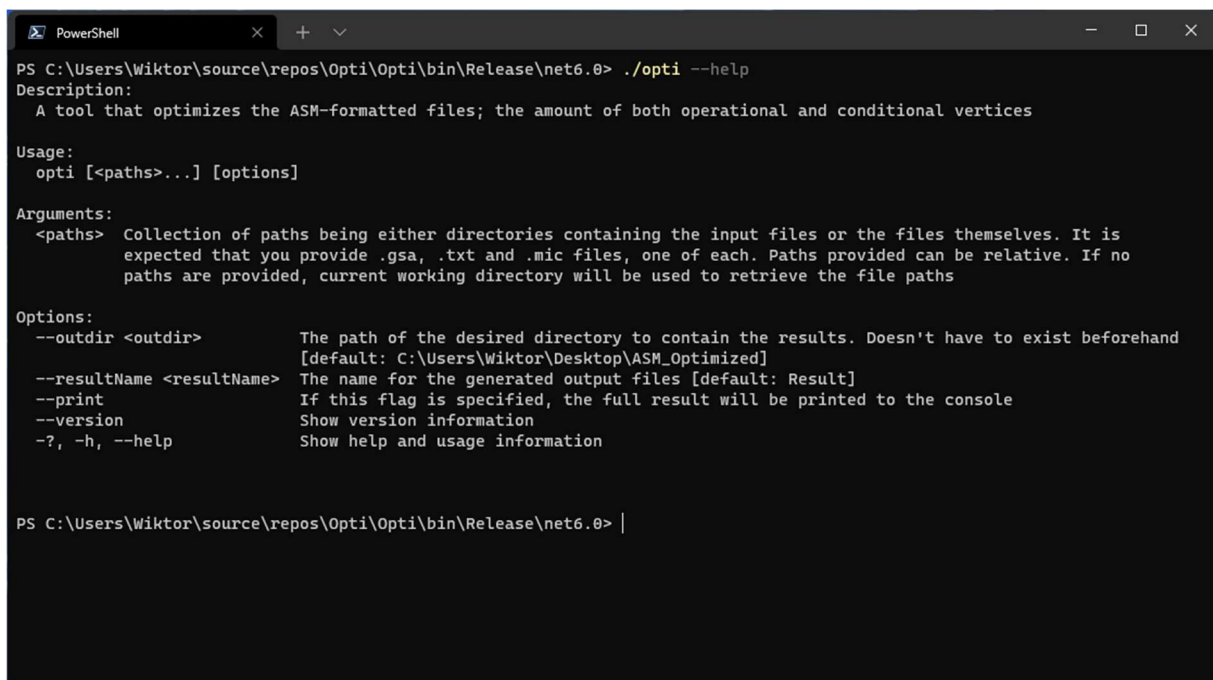
Możliwe jest skompilowanie programu pod systemy Windows, Linux, MacOS. Wymagana jest wcześniejsza instalacja środowiska .NET SDK w wersji odpowiadającej używanemu systemowi operacyjnemu, które można znaleźć pod linkiem: <https://dotnet.microsoft.com/en-us/download>

Kompilacja programu ze źródła odbywa się przy użyciu komendy „**dotnet build**” wywołanej w katalogu projektu. Możliwe jest sparаметryzowanie tej komendy, zgodnie z dostępną dokumentacją: <https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-build>

### 5.3. Instrukcja użytkowania programu

Podstawą obsługi programu jest dostępna w używanym systemie operacyjnym aplikacja terminala. Po otwarciu w niej katalogu programu, możliwe jest wywołanie go za pomocą komendy „**opti**” i podaniu jako argument ścieżkę/ścieżki dostępu do zapisanych na dysku niezoptymalizowanych plików.

Aplikacja posiada kilka opcji i możliwych flag (widocznych na Rys. 12), których szczegółowy opis można znaleźć uruchamiając program przy użyciu flagi **-help**:



```
PowerShell
PS C:\Users\Wiktork\source\repos\Opti\Opti\bin\Release\net6.0> ./opti --help
Description:
  A tool that optimizes the ASM-formatted files; the amount of both operational and conditional vertices

Usage:
  opti [<paths>...] [options]

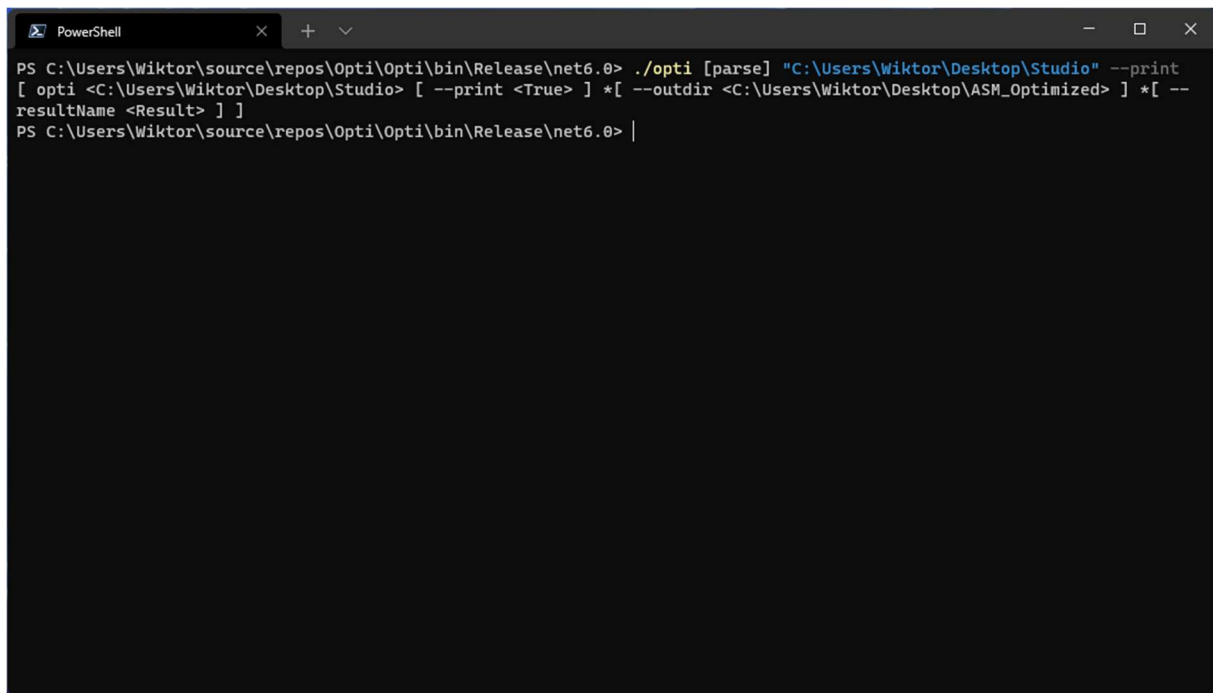
Arguments:
  <paths> Collection of paths being either directories containing the input files or the files themselves. It is
  expected that you provide .gsa, .txt and .mic files, one of each. Paths provided can be relative. If no
  paths are provided, current working directory will be used to retrieve the file paths

Options:
  --outdir <outdir>          The path of the desired directory to contain the results. Doesn't have to exist beforehand
  [default: C:\Users\Wiktork\Desktop\ASM_Optimized]
  --resultName <resultName> The name for the generated output files [default: Result]
  --print                    If this flag is specified, the full result will be printed to the console
  --version                 Show version information
  -?, -h, --help           Show help and usage information

PS C:\Users\Wiktork\source\repos\Opti\Opti\bin\Release\net6.0> |
```

Rys. 12. Fragment zrzutu ekranu przedstawiający sekcję pomocy aplikacji

Przed wykonaniem danej komendy obsługiwanej aplikacją można użyć dyrektywy „**[parse]**”, by zobaczyć i zweryfikować wejście programu przed jego wykonaniem (Rys. 13)



```
PowerShell
PS C:\Users\Wiktor\source\repos\Opti\Opti\bin\Release\net6.0> ./opti [parse] "C:\Users\Wiktor\Desktop\Studio" --print
[ opti <C:\Users\Wiktor\Desktop\Studio> [ --print <True> ] *[ --outdir <C:\Users\Wiktor\Desktop\ASM_Optimized> ] *[ --
resultName <Result> ] ]
PS C:\Users\Wiktor\source\repos\Opti\Opti\bin\Release\net6.0> |
```

Rys. 13. Fragment zrzutu ekranu demonstrujący działanie dyrektywy [parse]

Jedynym wymaganym argumentem do poprawnego uruchomienia programu jest ścieżka zawierająca dokładnie jeden egzemplarz każdego z plików w formatach \*.gsa, \*.txt, \*.mic.

Alternatywnie, program może przyjąć trzy argumenty, którymi są ścieżki do każdego z indywidualnych plików wejściowych.

Istnieje sposób na umożliwienie programowi automatycznego wykrycia lokalizacji plików wejściowych, którym jest dodanie folderu zawierającego program „**opti**” do zmiennej środowiskowej PATH. W takim wypadku, wywołanie programu bez argumentów, przy terminalu o aktywnym katalogu zawierającym pliki wejściowe, również zakończy się jego pomyślnym uruchomieniem.

Ostatni przewidziany sposób uruchomienia programu polega na przeciągnięciu na plików wejściowych lub katalogu zawierającego je na okno programu. W tym wypadku jednak nie jest możliwa parametryzacja jego uruchomienia, a także odczytanie przebiegu pracy programu z konsoli, ze względu na jej automatyczne zamknięcie (Rys. 14).

```
PowerShell
PS C:\Users\Wikt\o\Desktop\Studio> opti
I: Input checked, appears to be fully valid!
V: Duplicate operation: y3 [ a:=1 ] in [ 4 Y4 2 0 ]
V: Merged operations: [y1 [ d:=9 ]] from [ 1 Y1 6 0 ] into [ 3 Y3 1 0 ]
V: Merged operations: [y2 [ c:=7 ]] from [ 2 Y2 6 0 ] into [ 4 Y4 2 0 ]
V: Bringing out duplicate operation(s) [y3 [ a:=1 ]; y4 [ b:=6 ]] found in children of the vertex [ 5 x1 3 4 ] to [ 7 Y1 5 0 ]
I: Pass no. 1 - optimized 7 elements
I: Optimized 7 elements total
I: Optimization results saved to C:\Users\Wikt\o\Desktop\ASM_Optimized as Result_14.*
PS C:\Users\Wikt\o\Desktop\Studio> |
```

Rys. 14. Zrzut ekranu demonstrujący działanie programu

Domyślnym miejscem zapisu plików wyjściowych jest folder o nazwie „ASM\_Optimized” znajdujący się na pulpicie. Ścieżkę tę jednak można zmienić za pomocą odpowiedniego parametru przy wykonaniu programu, zgodnie z dostępną dokumentacją w sekcji pomocy.

Ze względu na charakter przeprowadzanych optymalizacji oraz fakt, że wykonanie jednej optymalizacji może otworzyć możliwość wykonania drugiej, optymalizacje są wykonywane w pętli do momentu braku znalezienia dalszych możliwości optymalizacji. Program zapisuje do konsoli informację o ilości dokonanych przebiegów algorytmów, oraz informację o podsumowaniu jego pracy i ilości zoptymalizowanych elementów

## 5.4. Implementacja algorytmów opisanych w rozdziałach

```
1. public override bool VerifyStructure()
2. {
3.     try
4.     {
5.         if (this.Content[0].TrimEnd() != "Y0")
6.         {
7.             return false;
8.         }
9.
10.        for (var i = 1; i < this.Content.Count; i++)
11.        {
12.            var line = this.Content[i];
13.
14.            if (!string.IsNullOrEmpty(line))
15.            {
16.                InstructionLine.ParseMic(line);
17.            }
18.        }
19.
20.        return true;
21.    }
22.    catch
23.    {
24.        return false;
25.    }
26. }
```

Powyżej przedstawiono jedną z głównych funkcji weryfikujących strukturę oraz poprawność wczytywanych danych wejściowych programu. Istnieją łącznie 3 odpowiednie funkcje w kodzie programu - po jednej dla każdego typu pliku wejściowego.

Dla zademonstrowanego przykładu, pochodzącego z pliku źródłowego **MicFile.cs**, weryfikowane jest w pierwszej kolejności to, czy pierwszy wiersz pliku stanowi wyrażenie „Y0” (dozwolone są znaki whitespace, ale tylko po tym wyrażeniu)

Następnie, następuje uruchomienie funkcji parsującej dla każdego pozostałego wiersza w pliku \*.mic. Funkcja ta powoduje wywołanie wyjątku w przypadku podania niekompatybilnego wiersza, więc w przypadku jego wystąpienia wyjątek zostaje obsłużony i zwracana jest informacja o niepowodzeniu procesu weryfikacji zawartości danego pliku.

```
1. public IEnumerable<GsaLine> GetChildren(GsaLine line)
2. {
3.     if (line.First != 0)
4.     {
5.         yield return this.Single(l => l.Index == line.First);
6.     }
7.
8.     if (line.Second != 0)
9.     {
10.        yield return this.Single(l => l.Index == line.Second);
11.    }
12. }
```

Wartością zwracaną przez funkcję jest typ wyliczeniowy **IEnumerable<T>**, który pozwala na użycie słowa kluczowego składni C# **yield return**, które to powoduje zwrócenie pojedynczego elementu bez zatrzymania wykonywania funkcji.

Atrybuty **First** i **Second** klasy **GsaLine** odnoszą się do indeksów dzieci danej linii pliku \*.gsa.

```

1. // algorytm rekurencyjny pozyskiwania ścieżek zawartych w diagramie
2. // ścieżka - kilka występujących po sobie bloków operacyjnych rozgraniczonych przez
   wierzchołki operacyjne i/lub bloki start/end
3. public List<GsaPath> GetPaths()
4. {
5.     var paths = new List<GsaPath>();
6.     var hashset = new HashSet<int>();
7.
8.     var element = this.GetChildren(this.GetStartInstruction()).Single();
9.     var current = new GsaPath(element.Index);
10.
11.     EnterPath(element);
12.
13.     void EnterPath(GsaLine line)
14.     {
15.         var children = this.GetChildren(line).ToList();
16.
17.         if (children.Count == 1)
18.         {
19.             current.Path.Add(line);
20.             EnterPath(children[0]);
21.         }
22.         else
23.         {
24.             if (current.Path.Any())
25.             {
26.                 paths.Add(current);
27.             }
28.
29.             if (!hashset.Add(line.Index))
30.             {
31.                 return;
32.             }
33.
34.             foreach (var child in children.GroupBy(line => line.Index).Select(g =>
   g.First()))
35.             {
36.                 current = new GsaPath(line.Index);
37.                 EnterPath(child);
38.             }
39.         }
40.     }
41.
42.     return paths;
43. }

```

Zaproponowana implementacja funkcji **GetPaths()** jest algorytmem rekurencyjnym.

W celu dokonania uniknięcia powtarzalności się indeksów sprawdzonych linii została zastosowana kolekcja typu **HashSet<T>**, która nie pozwala na umieszczenie w niej więcej niż jednej kopii danego elementu – w tym przypadku liczby typu **int**



```

1. // 'ParallelPaths' - para ścieżek pochodzących ze wspólnego wierzchołka warunkowego
2. // w przypadku, gdy ścieżki się łączą w pewnym momencie, funkcja zwraca części ścieżek
   tylko do momentu przecięcia
3. // funkcja zwraca zbiór par wszystkich równoległych ścieżek w diagramie
4. public IEnumerable<List<GsaPath>> GetParallelPaths()
5. {
6.     var lists = from path in this.GetPaths()
7.                 group path by path.Source into gr
8.                 let list = gr.ToList()
9.                 where list.Count > 1
10.                select list;
11.
12.     foreach (var list in lists)
13.     {
14.         for (var i = 0; i < list[0].Path.Count; i++)
15.         {
16.             var index = list[1].Path.FindIndex(l => l == list[0].Path[i]);
17.
18.             if (index >= 0)
19.             {
20.                 static void TruncAfter(GsaPath p, int i) => p.Path.RemoveRange(i,
p.Path.Count - i);
21.
22.                 TruncAfter(list[0], i);
23.                 TruncAfter(list[1], index);
24.             }
25.         }
26.
27.         if (list.All(path => path.Path.Count > 0))
28.         {
29.             yield return list;
30.         }
31.     }
32. }

```

Funkcja **ParallelPaths()** jest kolejną funkcją pomocniczą, która korzystając z uprzednio wspomnianej metody **GetPaths()** znajduje pary ścieżek wychodzące z jednego wierzchołka warunkowego i przygotowuje je do dalszych procedur optymalizacyjnych poprzez zwrócenie ich przyciętych części w przypadku, jeśli następuje ich połączenie.

Początkowo do zmiennej pomocniczej **lists** zapisywane są odnalezione pary ścieżek o wspólnym początku, po czym są porównywane znajdujące się na nich bloki operacyjne. Dzięki statycznej funkcji lokalnej **TruncAfter**, z list w danej parze usuwane są te bloki operacyjne, które stanowią ich część wspólną.

```

1. public class MergeOptimization : Optimization
2. {
3.     protected override int RunOptimization()
4.     {
5.         var count = 0;
6.
7.         // w każdej ścieżce, od drugiego elementu
8.         foreach (var path in Files.Gsa.GetPaths().Select(p => p.Path))
9.         {
10.            for (int i = 1; i < path.Count; i++)
11.            {
12.                // pomiń bloczki, do których wejście jest z więcej niż jednej strony
13.                if (Files.Gsa.SelectMany(Files.Gsa.GetChildren).Count(line => line
14. == path[i]) > 1)
15.                {
16.                    continue;
17.                }
18.                var oldOperations = Files.Txt.GetOperationsForInstruction(path[i -
19. 1].Instruction);
20.                var newOperations =
21. Files.Txt.GetOperationsForInstruction(path[i].Instruction);
22.                var toRemove = newOperations.Where(operation =>
23. IsCompatible(operation, oldOperations)).ToList();
24.                if (toRemove.Count == 0)
25.                {
26.                    continue;
27.                }
28.                // przeniesienie operacji z bloczka do usunięcia do bloczka wyższego
29.                Files.UpdateInstruction(Files.PrepareInstruction(path[i]),
30. newOperations.Except(toRemove));
31.                Files.UpdateInstruction(Files.PrepareInstruction(path[i - 1]),
32. oldOperations.Concat(toRemove));
33.                Print("Merged operations: {0} from {1} into {2}",
34. PrintOperations(toRemove), path[i], path[i - 1]);
35.                count += toRemove.Count;
36.            }
37.        }
38.        return count;
39.    }

```

Przedstawiony fragment kodu reprezentuje klasę będącą pierwszą z implementacji optymalizacji bloków operacyjnych. Realizuje ona problem, który polega na pozbyciu się zbędnych połączeń i redukowaniu dwóch bloków o różnych instrukcjach do jednego, tam gdzie jest to możliwe, wewnątrz jednej ścieżki.

Metoda ta odszukuje następujące po sobie w ramach ścieżki bloki operacyjne o jednym wejściu i usuwa pozostałe poza jednym, który przejmuje wszystkie operacje całej grupy.

```

1. public class BringOutOptimization : Optimization
2. {
3.     private static IEnumerable<string> GetAllOperations(GsaPath p)
4.     {
5.         return p.Path.Select(line =>
6.             line.Instruction).SelectMany(Files.Txt.GetOperationsForInstruction);
7.     }
8.     protected override int RunOptimization()
9.     {
10.        var count = 0;
11.
12.        // parallel paths - ścieżki dzielące wspólny początek i zakończenie
13.        foreach (var paths in Files.Gsa.GetParallelPaths())
14.        {
15.            var intersection =
16.                GetAllOperations(paths[0]).Intersect(GetAllOperations(paths[1])).ToArray();
17.
18.            // nie znaleziono elementów w tej optymalizacji
19.            if (intersection.Length == 0)
20.            {
21.                continue;
22.            }
23.
24.            // usuwanie części wspólnej z obu ścieżek
25.            foreach (var line in paths.SelectMany(gsaPath => gsaPath.Path))
26.            {
27.                var operations =
28.                    Files.Txt.GetOperationsForInstruction(line.Instruction);
29.                Files.UpdateInstruction(Files.PrepareInstruction(line),
30.                    operations.Except(intersection));
31.            }
32.
33.            // przenoszenie części wspólnej do nowego bloku
34.            var instruction = Files.AddInstruction(paths[0].Source, intersection);
35.
36.            var vertex = Files.Gsa.First(l => l.Index == paths[0].Source);
37.            var sourceLine = Files.Gsa.First(l => l.Instruction == instruction);
38.
39.            Print("Bringing out duplicate operation(s) {0} found in children of the
40.                vertex {1} to {2}", PrintOperations(intersection), vertex, sourceLine);
41.            count += intersection.Length;
42.        }
43.    }
44. }

```

Drugi z zaimplementowanych algorytmów optymalizacyjnych pozwala na wykrycie operacji duplikujących się operacji w obu ścieżkach wychodzących z danego wierzchołka warunkowego oraz przeniesienie ich do nowo utworzonego bloku operacyjnego u końca obu podścieżek. Wykorzystuje on wcześniej zademonstrowaną funkcję pomocniczą **GetParallelPaths()**.

```

1. public class DuplicateOptimization : Optimization
2. {
3.     protected override int RunOptimization()
4.     {
5.         // licznik dokonanych zmian
6.         var count = 0;
7.
8.         // dla każdej znalezionej ścieżki pomiędzy dwoma blokami warunkowymi
9.         foreach (var path in Files.Gsa.GetPaths().Select(gsaPath => gsaPath.Path))
10.        {
11.            // usuwać elementy będziemy od ostatniego do pierwszego bloku
operacyjnego w ścieżce
12.            path.Reverse();
13.
14.            var hashset = new HashSet<string>();
15.
16.            foreach (var line in path)
17.            {
18.                // pobierz operacje z bieżącej linii
19.                var operations =
Files.Txt.GetOperationsForInstruction(line.Instruction);
20.
21.                foreach (var operation in GetCompatible(operations))
22.                {
23.                    // już się pojawiła wcześniej, można usunąć
24.                    if (!hashset.Add(operation))
25.                    {
26.                        var instruction = Files.PrepareInstruction(line);
27.
28.                        var list =
Files.Txt.GetOperationsForInstruction(instruction).ToList();
29.                        list.Remove(operation);
30.
31.                        Files.UpdateInstruction(instruction, list);
32.
33.                        Print("Duplicate operation: {0} in {1}",
PrintOperation(operation), line);
34.                        count++;
35.                    }
36.                }
37.            }
38.        }
39.
40.        return count;
41.    }
42. }

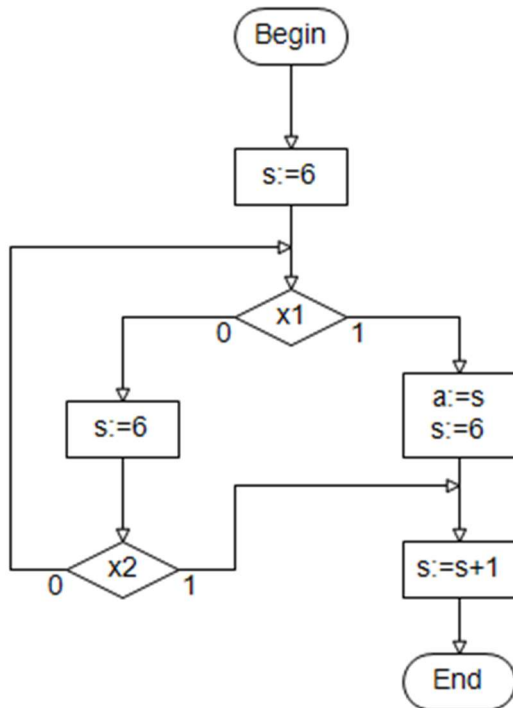
```

Trzecia i ostatnia z klas optymalizacyjnych realizuje kolejny problem, który polega na znalezieniu zduplikowanych instrukcji wewnątrz pojedynczej ścieżki i usunięciu duplikatu oraz pozostawienie tylko ostatniej z identycznych instrukcji wewnątrz ścieżki.

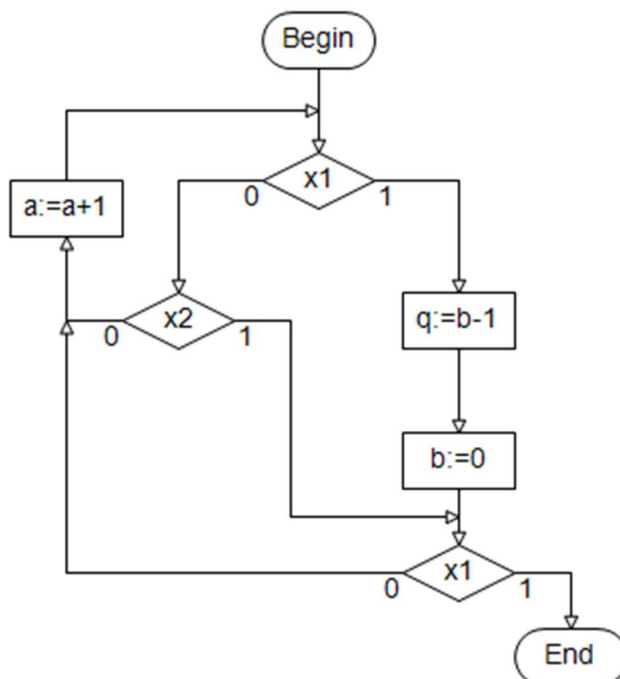
Przebieg tej funkcji polega na tym, że dla każdej ścieżki tworzona jest lista napotkanych dotychczasowo operacji. Przechodząc kolejno po blokach w ścieżce, od końca zapisywane są na liście kolejne operacje na nich się znajdujące. Gdy dana operacja znalazła się już na liście, wywoływana zostaje funkcja usuwająca operację z bloku operacyjnego, jednak tylko w przypadku gdy jej usunięcie nie wpłynie na inne korzystające z tego bloku ścieżki.

## 5.5. Testy autora

Test1 – przedstawia możliwości optymalizacji, które nie są możliwe do zrealizowania w jednym cyklu działania algorytmu.



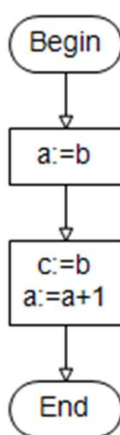
Test2 – zawiera bezpośrednio połączone wierzchołki warunkowe - jest większym wyzwaniem dla parsera.



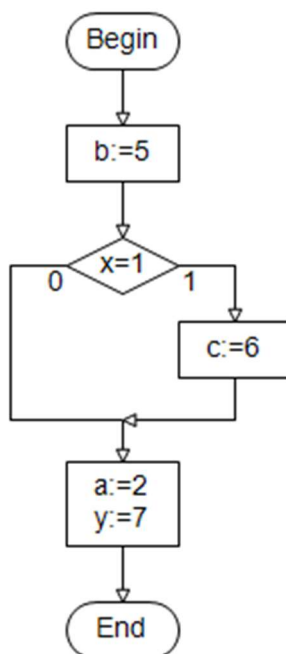
## 5.6. Wyniki optymalizacji

Na Rys. 15-18 przedstawiono kolejno w formie graficznej zoptymalizowane przypadki testowe odpowiednio dla schematów blokowych umieszczonych na Rys. 7-9 oraz Rys. 11.

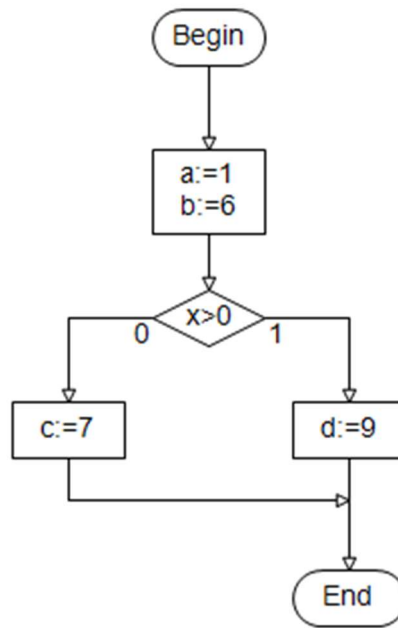
Ze względu na ograniczenia programu ASMCreator i brak możliwości wyświetlenia plików wynikowych, również po przeprowadzonej optymalizacji, rysunki zostały odtworzone przez Autora ręcznie, zgodnie z treścią plików wynikowych wytworzonego programu optymalizującego



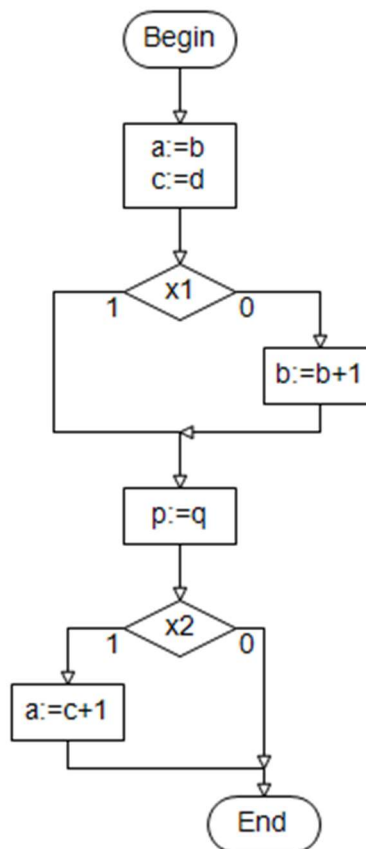
Rys. 15. Zoptymalizowany schemat blokowy z Rys. 7.



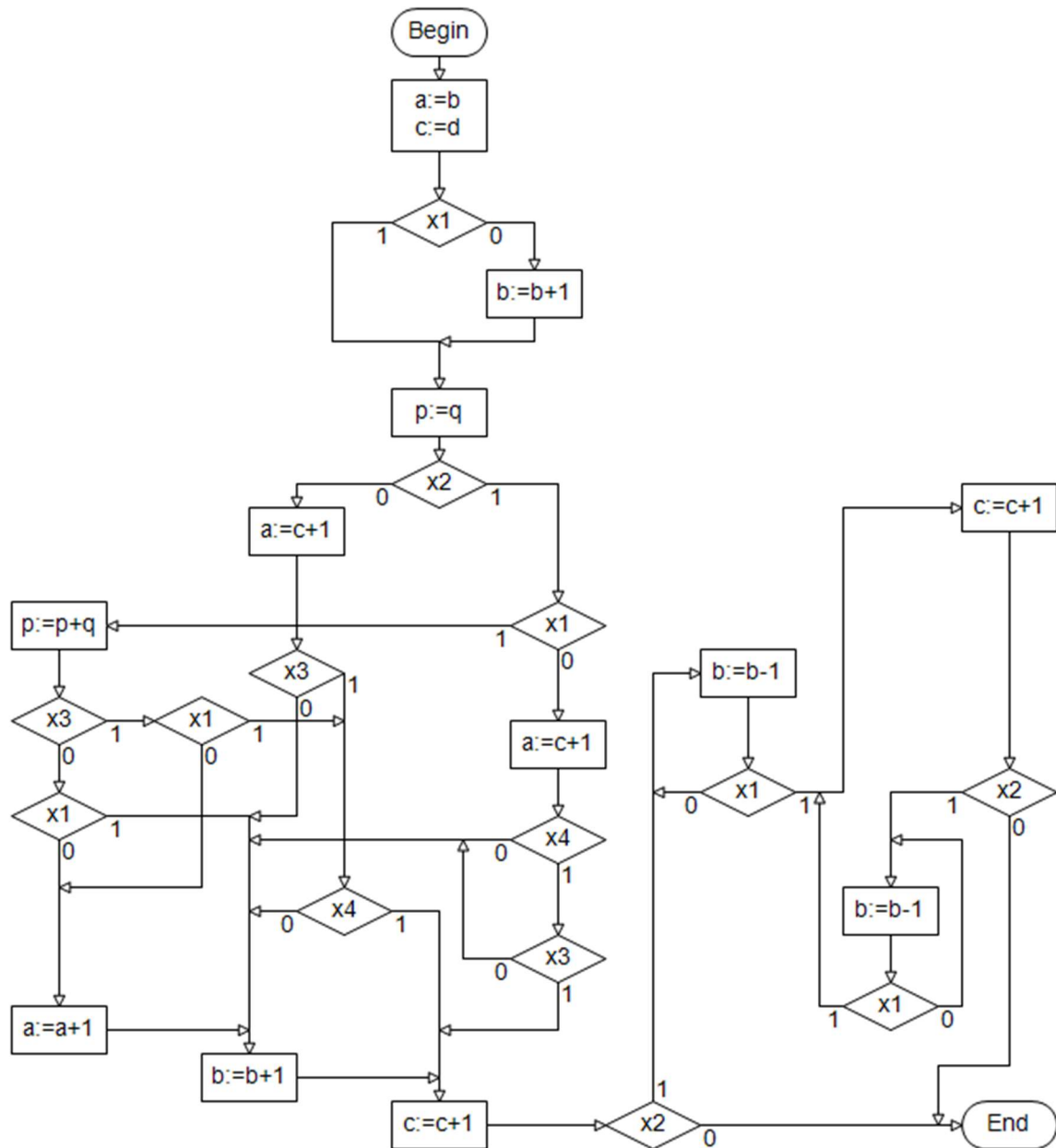
Rys. 16. Zoptymalizowany schemat blokowy z Rys. 8.



Rys. 17. Zoptymalizowany schemat blokowy z Rys. 9



Rys. 18. Zoptymalizowany schemat blokowy z Rys. 10



Rys. 19. Zoptymalizowany schemat blokowy z Rys. 11



## 6. Wnioski

ASMCreator to bardzo wygodny program do budowania schematów bloków, pozwalający na skompilowanie ich do postaci czytelnego zbioru plików tekstowych, co ułatwia dalszą obróbkę danych i pracę nad nimi. Minusem jednak jest brak możliwości działania odwrotnego.

Działanie opracowanego oprogramowania zostało przetestowane dla omówionych powyżej przypadków testowych tj. schematów blokowych 1-5, przedstawionych odpowiednio na rysunkach 7-11. Oznacza to że cel pracy został osiągnięty.

Wytworzony program jest stabilny i niezawodny, gdyż nie udało się znaleźć żadnego specyficznego przypadku, w którym by zawiódł. Na tę chwilę, Autor jest przekonany o wysokiej wartości użytkowej programu.

W planie dalszych prac warto byłoby rozszerzyć działanie programu o implementację klasycznego algorytmu minimalizacji stanów automatów, który zwiększyłby ilość odkrywanych przez program możliwości minimalizacji. Owe niezrealizowane możliwości są możliwe do dostrzeżenia w niektórych z zademonstrowanych rysunkach, m.in. na Rys. 19

Przeprowadzana optymalizacja mogłaby być również głębsza przy współpracy z podobnym programem posiadającym możliwości minimalizacji wierzchołków warunkowych. Wytworzone oprogramowanie zawiera możliwości dodania takiej współpracy w przyszłości.

Współdziałania takie powinno zachodzić, gdyż usunięcie wierzchołka jednego typu może otworzyć dany diagram na możliwość dokonania optymalizacji wierzchołków drugiego typu.

Użyteczne też mogło by się okazać opracowanie możliwości przedstawienia plików wynikowych programu w formie graficznej, co przede wszystkim byłoby dużą korzyścią dla czytelności i szybkiej, wizualnej możliwości porównania efektów początkowych z końcowymi dla danego użytkownika tej aplikacji. Tabela 4 przedstawia liczbę wykonanych optymalizacji dla poszczególnych rysunków (schematów blokowych).

Tab.4. Ilość dokonanych optymalizacji w schematach blokowych.

DIAGRAM	DOKONANE OPTYMALIZACJE
Rys. 7	3
Rys. 8	2
Rys. 9	7
Rys. 10	11
Rys. 11	11
Test 1	4
Test 2	2

## **Bibliografia**

[1] Baranov S. Finite State Machines and Algorithmic State Machines, ISBN Canada, 2018r.

[2] [http://www.barrywatson.se/dd/dd\\_algorithmic\\_state\\_machine.html](http://www.barrywatson.se/dd/dd_algorithmic_state_machine.html)

[3] [https://upload.wikimedia.org/wikipedia/commons/b/bc/Mealymachine\\_jaredwf.png](https://upload.wikimedia.org/wikipedia/commons/b/bc/Mealymachine_jaredwf.png)

[4] Baranov S., "Logic Synthesis for Control Automata", Kluwer Academic Publishers, 2012.