AsmC

asembler elektronicznej maszyny cyfrowej

ODRA 1003 / ODRA 1013

Wersja 2.5.1

2025-09-01

Klemens Czajka

Spis treści

Wstęp

Elektroniczne maszyny cyfrowe ODRA 1003 i ODRA 1013 to komputery skonstruowane i produkowane od roku 1963 w Zakładach Elektronicznych ELWRO:

https://pl.wikipedia.org/wiki/Odra_1003
https://pl.wikipedia.org/wiki/Odra_1013

Eemc ODRA 1003/1013 to emulator tych maszyn, zawierający dodatkowo modyfikacje dokonane w Zakładzie Metod Numerycznych Uniwersytetu Marii Curie-Skłodowskiej w Lublinie. Emulowane maszyny ODRA 1003, ODRA 1003 UMCS, ODRA 1013 i ODRA 1013 UMCS, to warianty tych komputerów bez i z tymi modyfikacjami.

Komputery ODRA 1003/1013 były oryginalnie wyposażone w PODSTAWOWY język programowania, w którym rozkazy i instrukcje kodowano ósemkowo, co pozwalało trzymać krótki i sprawny translator w pamięci operacyjnej podczas pisania, uruchamiania, testowania, a nawet eksploatacji programów użytkowych podawanych w postaci źródłowej. Programy w tym języku miały jedną zasadniczą wadę: były niezrozumiałe bez dobrej pamięciowej znajomości kodów ósemkowych. Komentarze do nich znajdowały się na arkuszach źródłowych i szybko się dezaktualizowały, a co gorsza, raczej zawierały objaśnienie operacji rozkazu niż algorytmu. Języki wyższego poziomu opracowane dla tych maszyn, poprzez większy rozmiar translatorów, wprowadzały dodatkowe ograniczenia zasobów (głównie cennej pamięci).

Emulator elektronicznej maszyny cyfrowej ODRA 1003 i ODRA 1013, pomyślany dla upamiętnienia i demonstracji rzeczywistych maszyn, potrzebuje języka programowania jeśli ma być modelem edukacyjnym. Język PODSTAWOWY jest zbyt nieczytelny, a języki wyższego poziomu na te maszyny są mniej przydatne dla poznania architektury ODRY 1003/1013 – są też obecnie niedostępne. Stąd myśl, by w pierwszej kolejności stworzyć język asemblera o zrozumiałej składni, dającego styczność z poszczególnymi rozkazami maszyny, a jednocześnie w znacznym stopniu samokomentującego się. Takim jest język AsmC i jego translator.

AsmC wymaga środowiska emulatora, gdyż potrzebuje dalekopisu TTY MKD-2 PL4 z bogatym zestawem znaków, prawdopodobnie nieistniejącego, oraz udostępnia asemblowanemu programowi informacje o systemie komputerowym zawarte w programie STAŁYM emulatora. Dzięki cechom emulatora, AsmC dodatkowo zyskuje możliwość wygodnego przygotowywania taśm perforowanych z tekstami źródłowymi programów, przyśpieszania pracy komputera, itp. AsmC nie jest zoptymalizowany do pracy na rzeczywistej maszynie, gdzie mógłby się okazać nieprzydatny ze względu na długi czas translacji, ale ma cechy dobrego języka i odpowiada specyfice komputera. Poza tym jest w pełni użyteczny i potrafi asemblować nawet wielkie programy. AsmC jest polskojęzyczny, oraz stara się zachować praktykę, klimat i poetykę czasów ODRY 1013.

Organizacja podręcznika

Niniejszy podręcznik zawiera uporządkowany opis asemblerowego języka programowania i jego translatora AsmC. Poszczególne rozdziały opisują kolejne aspekty z nimi związane, odwołując się przy tym często do cech i pojęć opisanych dalej, oraz posiłkując się przykładami ilustrującymi najważniejsze właściwości. Wymaga to od czytelnika wybiegania do innych części książki w celu pełnego zrozumienia. Takie podejście podyktowane jest powszechną obecnie znajomością informatyki, gdy nawet małe dzieci samodzielnie programują komputery. Książka ta opisuje asembler, jego szczegóły i specyfikę, ale nie rozwodzi się nad np. arytmetyką binarną (od której zaczynał się każdy podręcznik programowania we wczesnych latach informatyki) – polega się na

intuicji związanej z ogólną wiedzą. Szczegółów związanych konkretnie z ODRĄ 1003/1013 należy szukać np. w przywołanej poniżej bibliografii.

Pomimo iż w języku AsmC instrukcje i rozkazy można pisać małymi i wielkimi literami, to w opisach, przykładach, a szczególnie definicjach składni, wielkimi literami i znakami specjalnymi oznacza się elementy zapisywane literalnie, niezmienne, a małymi literami lub mieszanką liter małych i wielkich elementy zmienne, pod które należy wstawiać odpowiednie wartości, wyrażenia, lub opcje. Składowe alternatywne wymienia się w pionowej kolumnie, jedna pod drugą, na wspólnym tle. Pusta alternatywa (puste tło) oznacza możliwość pominięcia składowej instrukcji. Alternatywa wypełniona znakami ␣ symbolizującymi spacje oznacza wymagany odstęp jako jedną ze składowych. Kolejność zapisu poszczególnych składowych jest ustalona na sztywno. Oto przykładowa definicja składni rozkazu:

etyk ␣␣␣␣␣␣	**!**	**AC = AC +**	**[n]** **Bb**	**,**	**BON** **BO** **BN**	**;komentarz**

w której **etyk** (etykieta rozkazu), **n** (wyrażenie adresowe), **Bb** (rejestr B-modyfikacji) oraz **komentarz** są elementami zmiennymi, a reszta ustalonymi. Przecinek wolno zastąpić odstępem, a nawet całkowicie pominąć (ale oczywiście tylko wtedy, gdy nie grozi to sklejeniem w jeden dwóch oddzielnych symboli). Wokół operatorów i separatorów mogą znajdować się odstępy. Niewielkie odstępstwa od tego sposobu definiowania składni są zaznaczone osobnymi uwagami.

Bibliografia

Szczegółowe informacje dotyczące ODRY 1003 i ODRY 1013, oraz emulatora dają:

[1] artykuł „Thanasis Kamburelis, Maszyna cyfrowa ODRA 1003” zamieszczony w „Zastosowaniach Matematyki VIII (1965)”,

[2] skrypt akademicki UMCS „Światomir Ząbek: Programowanie i obsługa maszyn cyfrowych ODRA 1003 i ODRA 1013, Wydanie II, LUBLIN 1974”,

[3] skrypt Politechniki Wrocławskiej „Czesław Daniłowicz: Programowanie maszyn cyfrowych ODRA 1003 i ODRA 1013, WROCŁAW 1971”,

[4] instrukcja „Klemens Czajka: Emulator elektronicznej maszyny cyfrowej ODRA 1003 / ODRA 1013”,

[5] instrukcja „Klemens Czajka: Eemc ODRA 1003 / 1013 Program STAŁY – Rekonstrukcja”.

Program asemblera

Wymagania sprzętowe

Niektóre wymagania są związane ze środowiskiem emulatora Eemc ODRA 1003/1013:

ODRA 1003	pamięć ferrytowa ODRY 1013 jedynie polepsza wydajność asemblera, który pod jej adresami trzyma najczęściej używane zmienne
ODRA UMCS	wymagana jest jedynie gdy asemblowany program wstawia dane w kodzie THETA, do czytania których niezbędny jest czytnik taśmy perforowanej 8-kanałowej
TTY MKD-2 PL4	dalekopis z zestawem czterech pocztów znakowych z polskimi literami (AsmC jest polskojęzyczny)
PTR0	czytnik taśmy perforowanej 5-kanałowej, z którego jest czytany główny tekst asemblowanego programu
PTR2	czytnik taśmy perforowanej 5-kanałowej, gdy będą czytane dodatkowe teksty wstawiane instrukcjami WSTAW, albo kody wstawiane instrukcjami WDANE
PTR2’	czytnik taśmy perforowanej 8-kanałowej, gdy będą czytane kody wstawiane instrukcjami WDANEX
PTP5	perforator taśmy perforowanej 5-kanałowej, na który będzie wyprowadzany kod wynikowy programu
program STAŁY	powinien zawierać informacje o systemie komputerowym – w przeciwnym razie wartość symbolu standardowego SYSPARM będzie nieznacząca

Wejście do asemblera

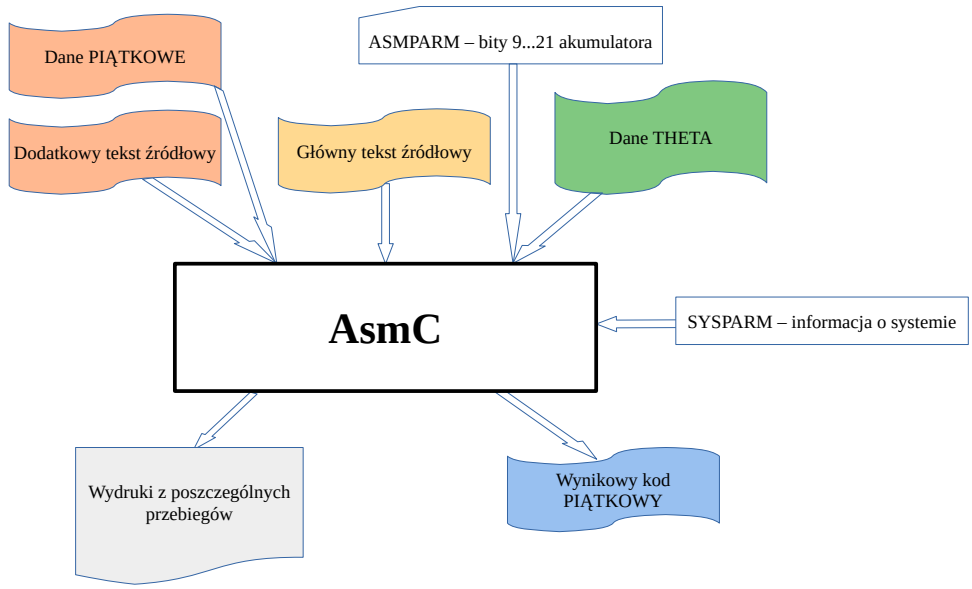
Danymi, na podstawie których AsmC tłumaczy program źródłowy napisany w języku asemblera AsmC na kod wykonywalny, są:

- SYSPARM – informacja o systemie komputerowym, zawarta w programie STAŁYM emulatora
- ASMPARM – parametry asemblacji wymagane przez asemblowany program, ustawione na przyciskach części AR akumulatora
- źródłowa taśma perforowana w kodzie ITA2 (MKD-2) z głównym tekstem asemblowanego programu
- opcjonalnie – taśmy perforowane w kodzie ITA2 z dodatkowymi tekstami źródłowymi (np. podprogramów), wstawianymi do tekstu głównego instrukcjami WSTAW
- opcjonalnie – taśmy perforowane w kodzie PIĄTKOWYM (pięciobitowym) z danymi wstawianymi do kodu wynikowego instrukcjami WDANE
- opcjonalnie – taśmy perforowane w kodzie THETA (ośmiobitowym) z danymi wstawianymi do kodu wynikowego instrukcjami WDANEX

Wyjście z asemblera

Wyjściem z asemblera są:

- wydruki na dalekopisie, stosowne dla każdego przebiegu
- kody (numery) wyświetlane na części AR rejestru rozkazów w rozkazach STOP i WE podczas montowania wymaganych taśm
- taśma perforowana z wynikowym kodem PIĄTKOWYM zasemblowanego programu



Konfiguracja wejścia/wyjścia asemblera

Kod wynikowy powstaje nawet wtedy, gdy w drugim przebiegu wykryto błędy – i wtedy jest on potencjalnie niepoprawny.

Program asemblera nie ma żadnych opcji, które mogłyby modyfikować proces asemblacji. Zbędne wydruki czy taśmy z kodem wynikowym, powstałe np. podczas próbnych asemblacji, należy po prostu odrywać i wyrzucać.

AsmC zawiera instrukcje AIF-AELIF-AELSE-AFI umożliwiające pisanie programów w sposób wariantywny, tj. dostosowany do systemu komputerowego wskazanego w SYSPARM (modelu ODRY, dalekopisu i środowiska systemu goszczącego emulator), do wymagań ASMPARM danej asemblacji ustawionych przyciskami akumulatora, lub zgodnie z opcjami ustawionymi jako wartości jakichś symboli w programie.

Informacja SYSPARM jest specyficzna dla programu STAŁEGO emulatora – w rzeczywistej maszynie byłaby nieznacząca.

Program zawłaszcza całą pamięć operacyjną, a więc zasadniczo całą maszynę, pozostawiając odłogiem jedynie dwa (niewykorzystane przy małej liczbie symboli w asemblowanym programie) obszary tablicy symboli.

Ogólna charakterystyka asemblera

AsmC jest translatorem dwuprzebiegowym:

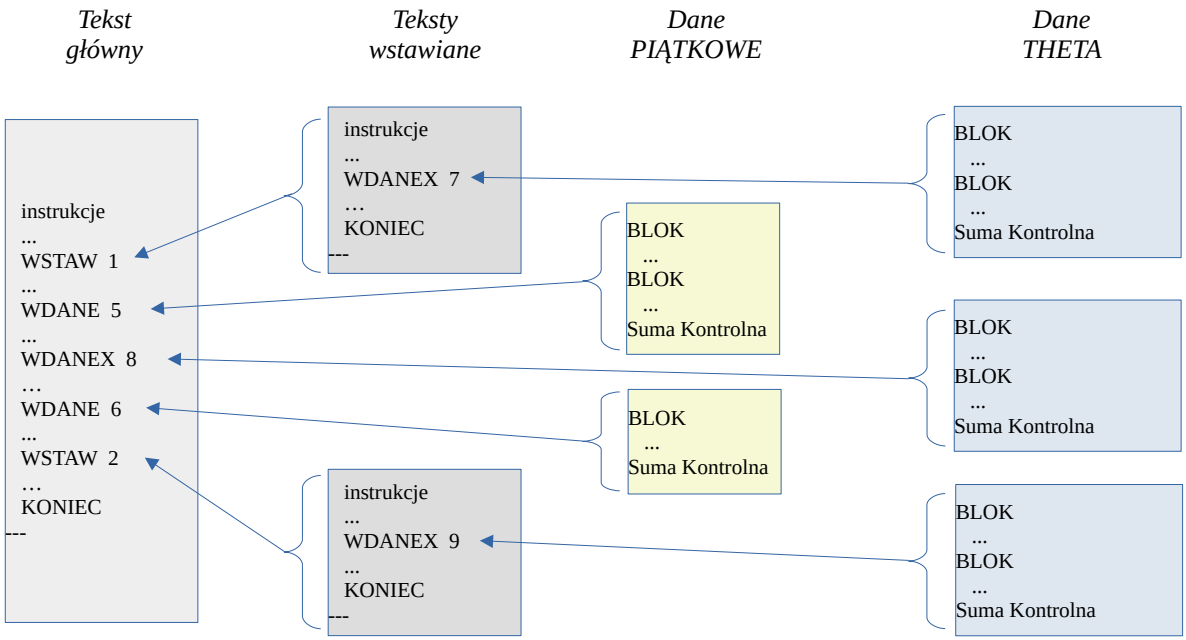
1. w pierwszym przebiegu czyta taśmy wejściowe, analizuje i listuje treść programu, wychwytuje błędy i tworzy wykaz nazw zawartych w programie,
2. w drugim przebiegu ponownie czyta taśmy wejściowe, generuje i listuje kod wykonywalny, wychwytuje dodatkowe błędy i tworzy taśmę programu ładowalnego w kodzie PIĄTKOWYM.

Drugi przebieg jest wykonywany tylko wtedy, gdy w pierwszym nie stwierdzono błędów. Drugi przebieg czyta taśmę (lub taśmy) od nowa, w tej samej kolejności co pierwszy – istotne jest więc uważne i staranne montowanie ich przez operatora, odpowiednio instruowanego przez program, by w obu przebiegach taśmy wejściowe były czytane dokładnie w ten sam sposób. Niedokładność powtórnego montowania taśmy może przede wszystkim zakłócić numerację linii.

Ogólny format tekstów źródłowych programu

Na program składają się teksty źródłowe na taśmach perforowanych. Każdy tekst składa się z wierszy zakończonych znakiem LF wysuwu papieru, oznaczającym przejście do nowego wiersza. Znaki CR powrotu karetki do początku wiersza są ignorowane. Każda instrukcja źródłowa musi być zakończona znakiem LF – również instrukcja KONIEC wskazująca koniec tekstu. Znak LF kończy instrukcję, jeśli nie jest wewnątrz stałej tekstowej lub komentarza blokowego.

Każdy tekst źródłowy, zarówno główny, jak i dodatkowy, wstawiany do głównego przy pomocy instrukcji WSTAW, musi być zakończony instrukcją KONIEC. Treść znajdująca się za instrukcją KONIEC nie jest w ogóle wczytywana. Asembler odróżnia instrukcję KONIEC w tekście głównym od instrukcji KONIEC w tekstach dodatkowych. Dzięki temu każdy tekst może być potraktowany jako główny, lub jako dodatkowy, wstawiany do innego.



Tekst główny i teksty wstawiane, oraz dane kopiowane do kodu wynikowego

W wydruku kodu wynikowego generowanego w drugim przebiegu, instrukcje KONIEC znajdujące się tekstach dodatkowych wstawianych do głównego instrukcją WSTAW, są listowane jako pseudoinstrukcje OSTAW na znak, że zakończono wstawianie.

Dane lub programy wstawiane instrukcjami WDANE i WDANEX z taśm w kodzie PIĄTKOWYM lub THETA, wstawiane są w postaci bloków pod oryginalne adresy zawarte w ich pilotach, lub pod adresy wskazane w tych instrukcjach. Zakończenie taśm danych, czy to sumą kontrolną czy odpowiednio długim ciągiem pustych rządków, listowane jest jako pseudoinstrukcja UDANE, ZDANE lub NDANE (albo odpowiednio jako UDANEX, ZDANEX lub NDANEX).

AsmC oferuje instrukcje NAF-KOF ułatwiające pisanie programów samoprzepisujących się na ścieżki ferrytowe, przeznaczonych do pracy na ODRZE 1013.

Analiza składniowa odbywa się sekwencyjnie, instrukcja po instrukcji, od lewej do prawej. Symbole, słowa kluczowe, operatory i separatory są rozpoznawane od ich początku, aż do napotkania znaku, który nie może do nich należeć. Zbyt długie nazwy są analizowane do ich końca, a dopiero potem uznawane za błędne i obcinane. Znaki specjalne są zbierane w jeden operator lub separator dopóty, dopóki tworzą zdefiniowany symbol operacji – dalsze znaki specjalne tworzą następny operator lub separator.

I tak np.:

Przykłady	*Objaśnienia*
`BLOK ADA`	Dwie nazwy, z których pierwsza jest słowem kluczowym
`BLOKADA`	Jedna nazwa, symbol, który gdzieś powinien być zdefiniowany
`BLOKADOM`	Jedna nazwa – błędna, bo za długa
`B1`	Rejestr B1
`B12`	Nazwa, symbol, który gdzieś powinien być zdefiniowany
`B8`	Nazwa, symbol, który gdzieś powinien być zdefiniowany
`A +++`	Błąd
`A ++ +`	Błąd – to samo co powyżej
`A + ++`	Dodawanie do akumulatora (licznika rozkazów + 1)
`-- - A`	Odejmowanie akumulatora od (licznika rozkazów - 1)
`---A`	Poprawne – to samo co powyżej
`A = A ///*komentarz*/ M,82`	Dzielenie długie //, a po nim komentarz blokowy – poprawnie
`A = A //*komentarz*/ M,82`	Dzielenie długie //, a po nim mnożenie – błąd
`SKZ --..++`	Poprawne
`SKZ -- .. ++`	Poprawne – to samo co powyżej
`B6=++..PODPROG`	Poprawne
`SKSB B6,PODPROG..++`	Poprawne – to samo co powyżej

Ogólna struktura wynikowych programów ładowalnych

AsmC generuje taśmę PIĄTKOWĄ z kodem programu wykonywalnego wg formatu zgodnego z programem STAŁYM. Taśma zawiera spójne bloki kodu, z których każdy jest poprzedzony pilotem zawierającym adres w pamięci operacyjnej, pod który blok ma być wczytywany do wykonania.

AsmC umożliwia definiowanie wielu bloków programu, nazywanych blokami wykonywalnymi. Jeden zdefiniowany blok w programie źródłowym, to jeden blok na taśmie wyjściowej. Program nie zawierający podziału na bloki składa się z jednego bloku wykonywalnego zaczynającego się od początku pamięci operacyjnej. Nie ma wymogu wyznaczania bloków, ale jeśli program ma być ładowany do pamięci nie na jej początku, to konieczna jest instrukcja BLOK wskazująca adres początkowy bloku wykonywalnego.

Program źródłowy może także zawierać quasi-bloki, wyłącznie opisujące format danych lub obszarów pamięci znajdujących się gdzie indziej, w celu ułatwienia odwołań do nich. Rozkazy i dane w takim pozornym bloku nie są asemblowane w postaci kodu PIĄTKOWEGO na taśmie wyjściowej. Instrukcja QBLOK wskazuje bezwzględny adres lub względną pozycję początku pozornego bloku. Względna pozycja oznacza, że jest to opis danych lub obszarów pamięci, które mogą występować w wielu miejscach pamięci, i że do wartości etykiet w QBLOKu należy dodawać adres bazowy, by uzyskiwać ostateczne adresy danych. Do bezwzględnego adresu również można dodawać liczby proste indeksujące QBLOK, co stosuje się do opisania np. postaci elementu tablicy. Składnik dodawany do adresu lub pozycji może znajdować się w rejestrze B-modyfikacji. Jeśli do adresowania danych w QBLOKu używamy stale jednego i tego samego rejestru B, to można QBLOK bazować na adresie lub pozycji indeksowanej tym rejestrem, co zwalnia z konieczności ciągłego dodawania rejestru w odwołaniach. QBLOK może być bazowany na liczbie prostej, liczbie indeksowanej, adresie prostym lub adresie indeksowanym.

Alfabet asemblera

AsmC jest programem polskojęzycznym: słowa kluczowe (z kilkoma zrozumiałymi wyjątkami), nazwy symboli, komunikaty i wydruki są w języku polskim. Z tego względu wymagany jest dalekopis z zestawem znaków TTY MKD-2 PL4, którego alfabet jest następujący:

(Dec)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
PL2	●	E	≡	A	◡	S	I	U	⌂	D	R	J	N	F	C	K	T	Z	L	W	H	Y	P	Q	O	B	G	▲	M	X	V	▼
	●	3	≡	-	◡	'	8	7	⌂	*	4	;	,	!	:	(	5	+	)	2	Ł	6	0	1	9	?	&	▲	.	/	=	▼
PL3	●	e	≡	a	◡	s	i	u	⌂	d	r	j	n	f	c	k	t	z	l	w	h	y	p	q	o	b	g	▲	m	x	v	▼
PL4	●	ę	≡	_	◡	#	Ź	Ś	⌂	ź	Ę	ń	<	ż	^	[	Ń	ą	]	Ć	ł	Ó	Ż	Ą	ó	ć	%	▲	>	|	ś	▼
(Oct)	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17	20	21	22	23	24	25	26	27	30	31	32	33	34	35	36	37

gdzie symbole graficzne: ● ≡ ◡ ⌂ ▲ ▼
reprezentują kody ITA: NU LF SP CR FS LS

W alfabecie dalekopisu, kody ITA: ●, ≡, ◡, ⌂, ▲ i ▼ występują w każdym poczcie, co utrudnia przetwarzanie łańcuchów tekstowych.

Znaki ●, ▲ i ▼, jako kody zmiany pocztu znakowego, są niedopuszczalne w łańcuchach tekstowych kodowanych 6- i 7-bitowo.

Asembler umożliwia definiowanie:

- łańcuchów tekstowych 5-bitowych: topornych, ciosanych i dłubanych, złożonych ze znaków ITA: PL4, PL3 albo PL2, w postaci kodów 5-bitowych, wśród których będą kody zmiany pocztu.
- łańcuchów tekstowych 6-bitowych, złożonych ze znaków ITA PL2 w postaci kodów 6-bitowych, wśród których nie będzie kodów zmiany pocztu, a zamiast tego najstarszy bit kodu znaku wskazuje poczet znaku.
- łańcuchów tekstowych 7-bitowych, złożonych ze znaków ITA: PL4 i PL3, w postaci kodów 7-bitowych, wśród których nie będzie kodów zmiany pocztu, a zamiast tego dwa najstarsze bity kodu znaku wskazują poczet znaku.

Wewnętrzny alfabet asemblera

AsmC wewnętrznie stosuje swoje własne, odrębne kodowanie alfabetu:

(Dec)...	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Alfabet odrębny AsmC	●	⌂	≡	☺	◡	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	_	;	,	=	?	#	%	|	^	[
	]	<	>	(	)	'	:	*	&	!	/	.	-	+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	G	H
	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	Ą	Ć	Ę	Ł	Ń	Ó	Ś	Ź	Ż	▲	a	b	c	d
	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	ą	ć	ę	ł	ń	ó	ś	ź	ż	▼
...(Dec)	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

gdzie symbole graficzne:	●	≡	◡	⌂	▲	▼	☺
reprezentują kody ITA:	NU	LF	SP	CR	FS	LS	nieobsadzone

Każdy znak występuje w tym alfabecie jednokrotnie, a ponadto pozostaje 18 pozycji kodowych bez przypisanych znaków, które można wykorzystywać do własnych celów.

Asembler umożliwia definiowanie łańcuchów tekstowych 7-bitowych odrębnych, złożonych ze znaków wewnętrznego alfabetu asemblera AsmC (w postaci kodów, wśród których nie ma kodów zmiany pocztu).

ZALECENIE:

Kodowanie alfabetu stosowane przez asembler wymaga uciążliwych translacji przy wczytywaniu i drukowaniu tekstów. W prostszych zastosowaniach posługiwanie się czterema, czy nawet trzema pocztami znaków może obyć się bez translacji, jednakże pozostaje skomplikowane stosowanie kodów zmiany pocztów, szczególnie przy konwersacji z użyciem dalekopisu. Jeżeli nie ma wyraźnej potrzeby używania wielu znaków, jeśli można obyć się alfabetem z dwoma pocztami znaków, to należy się do tego ograniczyć. Rozróżnianie wielkich i małych liter może nie być warte komplikacji w programie – najwyżej jakieś nazwy własne będą z małej litery.

Język asemblera

Język asemblera AsmC jest językiem programowania symbolicznego, odpowiadającym programowaniu na najniższym poziomie rozkazów maszynowych. Jednocześnie jego przyjazna składnia ułatwia programowanie i rozumienie czytanego tekstu programu; dlatego, miejmy nadzieję, nie odstraszy osób zaciekawionych ODRĄ 1003/1013. Składają się nań:

- komentarz blokowy /* ... */
- instrukcje asemblera – dyrektywy
- instrukcje asemblera warunkowej asemblacji, oraz emisji komunikatów
- instrukcja DS – definicja słów danych
- instrukcje maszynowe – rozkazy

Komentarz blokowy służy do wstawiania do tekstu źródłowego obszerniejszych opisów i objaśnień, mogących ciągnąć się przez wiele wierszy. Może znajdować się wszędzie tam, gdzie może znajdować się spacja, i jest traktowany jako odstęp, a poza tym ignorowany.

Instrukcje asemblera są to wskazania dotyczące adresowania pamięci, definicje symboli, żądania wstawienia dodatkowego tekstu źródłowego, żądania włączenia danych do kodu wynikowego, oraz wskazanie końca tekstu źródłowego. Instrukcje te same w sobie nie generują kodu wynikowego.

Instrukcje asemblera warunkowej asemblacji wskazują poprzez warunki, które części tekstu źródłowego mają być asemblowane, a które pomijane. Razem z nimi będą używane instrukcje BYK – emitowania komunikatu błędu, oraz NOTA – emitowania komunikatu informacyjnego (adnotacji) w ramach samokontroli.

Instrukcja DS generuje słowa w pamięci, w których generalnie są dane. Zwróćmy uwagę, że słowo wygenerowane przez instrukcję maszynową jak i przez instrukcję DS może być użyte zarówno jako dana jak i rozkaz – oba stanowią kod wynikowy.

Instrukcje maszynowe odpowiadają rozkazom maszynowym generowanym jako słowa rozkazowe i wykonywanym po uruchomieniu programu.

Oznaczenia, symbole i mnemoniki instrukcji i rozkazów generalnie odpowiadają stosowanym w [2] i [3], ze zmianami wprowadzonymi w emulatorze Eemc ODRA 1003/1013 i wymienionymi w [4]. Asembler AsmC wprowadza dodatkowe modyfikacje i uzupełnienia. Zmiany są wylistowane w rozdziale „ZMIANY oznaczeń, symboli i mnemoników w stosunku do oryginalnych”.

Ogólny format instrukcji

Generalnie jedna instrukcja zajmuje jeden wiersz programu źródłowego. Szerokość wiersza jest w praktyce nieograniczona. W jednym wierszu nie da się umieścić kilku instrukcji. Na instrukcję składają się:

- etykieta instrukcji, na początku wiersza
- zasadnicza treść instrukcji, poprzedzona odstępem i ew. korektorem kodu operacji rozkazu, oraz zakończona opcjonalnym odstępem
- komentarz liniowy, poprzedzony średnikiem

```
etyk    instrukcja                  ;komentarz liniowy
symb    instrukcja                  ;komentarz liniowy
id      instrukcja                  ;komentarz liniowy
```

Etykieta instrukcji musi zaczynać się od początku wiersza. Jest to nazwa zgodna z zasadami nadawania nazw symbolom. Etykieta definiuje symbol o odpowiedniej wartości, lub identyfikator instrukcji asemblera nie mający wartości (niektóre symbole są też identyfikatorami). Identyfikator zawsze będziemy nazywać identyfikatorem; pozostałe etykiety będziemy nazywać etykietami, symbolami, lub po prostu nazwami. Symbol może występować w wyrażeniach reprezentując swoją wartość. Identyfikator instrukcji służy jedynie do lepszej orientacji w strukturze programu.

Etykieta nie jest obowiązkowa – jeśli zostanie pominięta, to zasadnicza treść instrukcji musi zaczynać się odstępem. Etykieta jest niedozwolona w instrukcji KONIEC wskazującej koniec tekstu źródłowego.

Zasadnicza treść instrukcji zależy od jej rodzaju, i zaczyna się po odstępie za etykietą instrukcji, a kończy wraz z początkiem komentarza liniowego lub końcem wiersza. Będziemy ją nazywać krótko instrukcją lub rozkazem. Zasadnicza treść instrukcji jest opcjonalna – jeśli zostanie pominięta, to ewentualna etykieta staje się symbolem o wartości równej wartości licznika lokacji (licznika rozkazów) w tym punkcie programu. W poniższym fragmencie programu etykiety etyk1, etyk2 i etyk3 otrzymają tę samą wartość:

```
etyk1
etyk2   EQU     *               ;definicja symbolu
etyk3   A = 0                   ;rozkaz maszynowy
```

Wszędzie tam, gdzie w instrukcji lub rozkazie może znajdować się spacja, może być więcej spacji niż jedna. Tam, gdzie składnia przewiduje rozdzielanie argumentów przecinkami, przecinek może być zastąpiony (z małymi wyjątkami) spacją. Tam gdzie przecinek jest niezbędny, jego brak doprowadzi do błędu składniowego.

Komentarz liniowy jest opcjonalny i całkowicie ignorowany. Zaczyna się po znaku średnika (;) i ciągnie aż do końca wiersza. Średnik (;) zawarty wewnątrz łańcucha znaków ujętego w apostrofy, albo wewnątrz komentarza blokowego, nie jest rozpoznawany jako początek komentarza liniowego. Znaki /* znajdujące się wewnątrz komentarza liniowego nie są rozpoznawane jako otwierające komentarz blokowy. Nie da się zagnieżdżać komentarzy.

Komentarz blokowy

Komentarz blokowy zaczyna się znakami /* nie zawartymi wewnątrz łańcucha znaków ujętego w apostrofy, a kończy znakami */ w tej lub którejś kolejnej linii. Tego rodzaju komentarz pozwala uniknąć stawiania średnika w każdym wierszu komentarza. Komentarz blokowy jest traktowany jak odstęp (SP).

```
/*
- Wewnątrz komentarza blokowego średnik (;) nie rozpoczyna
  komentarza liniowego, ani nie kończy komentarza blokowego.
- Znaki /* wewnątrz komentarza blokowego nie rozpoczynają nowego,
  zagnieżdżonego komentarza. Nie da się zagnieżdżać komentarzy.
- Znaki apostrofu 'nie mają znaczenia' dla rozpoznawania końca
  komentarza blokowego.
- Komentarz blokowy kończy się dopiero po napotkaniu zamykających
  go znaków:
*/
```

Przy pomocy komentarza blokowego można wyłączyć duży fragment programu pod warunkiem, że nie ma w nim komentarzy blokowych.

Nietypowe zastosowanie komentarza blokowego umożliwia zapis instrukcji w wielu wierszach. Jeśli instrukcja może zawierać odstępy jako separatory argumentów, to zamiast lub obok takiej spacji można otworzyć komentarz blokowy i zamknąć go w następnym wierszu, a po nim umieścić następne argumenty.

Oto przykład zapisania tego samego rozkazu w jednym i w trzech wierszach:

```
pomnóż  M=[mnożnik], |M|, AM=A*M  ;mnożenie może się nie odbyć
```

```
pomnóż  M=[mnożnik],        /* jeśli to liczba -274877906944
*/      |M|,                /* to wartość bezwzględna da nadmiar
*/      AM=A*M              /* i mnożenie nie odbędzie się */
```

Słowa kluczowe i nazwy zastrzeżone

W języku AsmC rejestry maszyny, instrukcje asemblera, mnemoniki rozkazów i opcje rozkazów wskazuje się za pomocą słów kluczowych. Słowa kluczowe są to nazwy zastrzeżone przez asembler, których nie można zdefiniować do innych celów. AsmC ponadto predefiniuje wartości kilku symboli standardowych. Pozostałe nazwy są definiowalne i służą jako identyfikatory lub symbole reprezentujące wartości. Oto wykazy zastrzeżonych nazw:

Wykaz rejestrów maszyny

Oznaczenie rejestru	*Znaczenie*
A	39-bitowy rejestr akumulatora, z dodatkowym bitem technicznym
B0	„pozorny” rejestr B-modyfikacji – zawsze zawiera 0
B1	13-bitowy rejestr B-modyfikacji części N (AR) rozkazu – argumentu
B2	13-bitowy rejestr B-modyfikacji części N (AR) rozkazu – argumentu

B3	13-bitowy rejestr B-modyfikacji części N (AR) rozkazu – argumentu
B4	13-bitowy rejestr B-modyfikacji części N (AR) rozkazu – argumentu
B5	13-bitowy rejestr B-modyfikacji części N (AR) rozkazu – argumentu
B6	13-bitowy rejestr B-modyfikacji części K (NR) rozkazu – adresu następnego rozkazu
B7	39-bitowy rejestr B-modyfikacji całego rozkazu, z dodatkowym bitem technicznym, nazywany rejestrem M w rozkazach mnożenia i dzielenia stałoprzecinkowego
M	39-bitowy rejestr mnożnika w rozkazach mnożenia i dzielenia stałoprzecinkowego, wszędzie indziej nazywany rejestrem B7
AM	78-bitowy rejestr długi, tj. para rejestrów (A,M), w której bit znaku rejestru M często jest jedynie zerowany
AC	rejestr zmiennoprzecinkowy, czyli para rejestrów: akumulator A, oraz 7-bitowy rejestr cechy liczby zmiennoprzecinkowej
OM	1-bitowy rejestr zaokrągleń Ω
R	39-bitowy rejestr rozkazów, z dodatkowym bitem technicznym, który uczestniczy w operacji dzielenia

Wykaz słów kluczowych instrukcji asemblera

Instrukcja asemblera	*Znaczenie*
AIF	pierwsza klauzula warunkowej asemblacji
AELIF	kolejna klauzula warunkowej asemblacji
AELSE	ostatnia klauzula warunkowej asemblacji
AFI	koniec klauzul warunkowej asemblacji
BLOK	początek BLOKu kodu
QBLOK	początek Quasi-BLOKu pamięci – pozornego bloku
BYK	wypisanie komunikatu błędu (BYKa) w ramach samokontroli
NOTA	wypisanie adNOTAcji w ramach samokontroli
EQU	definicja symbolu
NAF	start przeadresowywania (początek segmentu programu wykonywanego NA Ferrycie)
KOF	stop przeadresowywania (KOniec segmentu programu wykonywanego na Ferrycie)
WSTAW	WSTAWianie dodatkowego tekstu źródłowego
WDANE	Wkopiowanie do kodu wynikowego DANych z taśmy 5-kanałowej
WDANEX	Wkopiowanie do kodu wynikowego DANych z taśmy 8-kanałowej
KONIEC	KONIEC danego tekstu źródłowego

Wykaz słów kluczowych rozkazów maszynowych

Rozkaz maszynowy		*Znaczenie*	
Mnemonik	*Operator*		
AL	**<<**	rozkaz	przesunięcie Arytmetyczne w Lewo
AP	**>>**	rozkaz	przesunięcie Arytmetyczne w Prawo
ALD	**<<**	rozkaz	przesunięcie Arytmetyczne w Lewo, Długie
APD	**>>**	rozkaz	przesunięcie Arytmetyczne w Prawo, Długie
LL	**<<<**	rozkaz	przesunięcie Logiczne w Lewo
LP	**>>>**	rozkaz	przesunięcie Logiczne w Prawo
CP	**><**	rozkaz	przesunięcie Cykliczne w Prawo
BF		rozkaz	przesyłanie blokowe z Bębna na ścieżkę Ferrytową
FB		rozkaz	przesyłanie blokowe ze ścieżki Ferrytowej na Bęben
BN		opcja	Bez Normalizacji zmiennoprzecinkowej
BO		opcja	Bez zaOkrąglenia zmiennoprzecinkowego
BON		opcja	Bez zaOkrąglenia i Normalizacji zmiennoprzecinkowej
CZK		rozkaz	CZytanie Klawiatury (przycisków akumulatora)
DS		instrukcja	Definicja Słów danych
DZD	**//**	rozkaz	DZielenie Długie
NIC		rozkaz	NIC nie rób
OKR	**A=A+OM**	rozkaz	zaOKRąglenie normalne bitem zaokrągleń Ω
SKBG		rozkaz	SKok przy Braku Gotowości
SKLC		rozkaz	SKok z Licznikiem Cykli prawdziwym
SKU		rozkaz	SKok przy Ujemnym
SKZ		rozkaz	SKok przy Zerze
SKD		rozkaz	SKok przy Dodatnim
SKV		rozkaz	SKok przy Nadmiarze
SKM		rozkaz	SKok przy Mniejszym
SKR		rozkaz	SKok przy Równym
SKW		rozkaz	SKok przy Większym
SKNBG		rozkaz	SKok przy Nie Braku Gotowości
SKNLC		rozkaz	SKok z Licznikiem Cykli Nieprawdziwym
SKNU		rozkaz	SKok przy Nie Ujemnym
SKNZ		rozkaz	SKok przy Nie Zerze
SKND		rozkaz	SKok przy Nie Dodatnim
SKNV		rozkaz	SKok przy Nie Nadmiarze
SKNM		rozkaz	SKok przy Nie Mniejszym
SKNR		rozkaz	SKok przy Nie Równym

SKNW		rozkaz	SKok przy Nie Większym
SKMR		rozkaz	SKok przy Mniejszym lub Równym
SKWR		rozkaz	SKok przy Większym lub Równym
SKS		rozkaz	SKok ze Śladem
SKSB		rozkaz	SKok ze Śladem w rejestrze B-modyfikacji
STOP		rozkaz	STOP-skocz
WE		rozkaz	WEjście z urządzenia zewnętrznego, z czytnika / dalekopisu
WY		rozkaz	WYjście na urządzenie zewnętrzne, na perforator / dalekopis

Wykaz predefiniowanych symboli standardowych

Predefiniowane symbole standardowe

Symbol	*Opis*
ASMPARM	Jest to wartość ustawiona przyciskami części N (AR) akumulatora na pulpicie maszyny (bity nr 9...21). Symbol przeznaczony jest do odebrania w asemblowanym programie parametrów asemblacji, dzięki którym można uzyskiwać różne warianty tego samego programu. Wartość tego symbolu jest 13-bitową liczbą bez znaku, której najstarszym bitem jest bit nr 9 przycisku akumulatora, a najmłodszym bit nr 21. Przycisk wciśnięty to bit o wartości 1, zwolniony to 0.
SYSPARM	Jest to informacja o systemie komputerowym, zawarta w programie STAŁYM emulatora. Dzięki temu symbolowi można dostosować program wynikowy do komputera, dalekopisu i środowiska emulatora. Poza środowiskiem emulatora, w rzeczywistej maszynie, wartość tego symbolu jest niezdefiniowana. Wartość tego symbolu jest 13-bitową liczbą bez znaku, której najstarszym bitem jest bit nr 9, a najmłodszym bit nr 21 słowa o adresie 0c17654. Dokładny opis informacji o systemie komputerowym znajduje się w dokumentacji programu STAŁEGO. Aktualnie jest to ciąg bitów: `FUC....JJJJSS` – gdzie: `F` 1=ODRA 1013, tj. z pamięcią ferrytową `U` 1=ODRA UMCS, tj. z urządzeniami 8-kanałowymi `C` 0=System hosta z NL=(CR,LF) 1= System hosta z NL=(LF) – bez CR `....` bity zarezerwowane, wyzerowane `JJJJ` Język dalekopisu: 0000=PL, 0001=GB, 0010=DE, 0011=FR, 0100=SC, 0101=US, 0111=RU `SS` Sterowanie dalekopisem: 00=dwa..., 01=trzy..., 10=cztery poczty wg PL, 11=trzy poczty wg RU

Nazwy

Nazwa na początku wiersza, w roli etykiety, jest definiowana jako identyfikator lub symbol. Raz zdefiniowany symbol otrzymuje wartość, której nie można zmienić. Nazwy użyte w wyrażeniach reprezentują swoje wartości, nadając im znaczenie. Dzięki temu łatwo zrozumieć sens wartości i odnaleźć wszystkie jej wystąpienia w programie. Asembler drukuje wykaz wszystkich nazw, wraz z ich wartościami i typami wartości.

Nazwa składa się z liter i cyfr `ABCDEFGHIJKLMNOPQRSTUVWXYZĄĆĘŁŃÓŚŹŻ0123456789`, przy czym pierwszym znakiem musi być litera. Litery polskie `ĄĆĘŁŃÓŚŹŻ` dozwolone są tak samo jak pozostałe łacińskie. Nazwy mogą być pisane literami wielkimi, małymi, lub mieszanką wielkich i małych. Asembler w nazwach nie rozróżnia liter wielkich i małych. Długość nazwy jest ograniczona do 7 znaków.

Nazwy zwykłe – globalne

Żadna nazwa nie może być zdefiniowana kilkakrotnie, tj. nie może wystąpić więcej niż raz w roli etykiety (z wyjątkiem kompletowania instrukcji złożonej poprzez wspólny identyfikator). Zwykła nazwa, taka jak zdefiniowano powyżej, tj. nie poprzedzona znakiem #, jest dokładnie taka jak ją napisano. Oznacza to, że nie może być zdefiniowana nigdzie indziej w całej asemblacji. Wśród wszystkich źródłowych taśm wejściowych jej definicja może pojawić się tylko jeden raz. Wymóg ten narzuca ogromną dyscyplinę na nazewnictwo symboli w bibliotece programów źródłowych.

Zwykła nazwa ma charakter globalny.

Nazwy generowane – lokalne

Wszystkie nazwy w jednej asemblacji muszą być unikalne, zdefiniowane tylko raz. Wymóg ten narzuca ogromną, wręcz niemożliwą do utrzymania, dyscyplinę na nazewnictwo symboli w bibliotece programów źródłowych. A przecież tylko niektóre nazwy w programach mają znaczenie globalne, np. nazwy funkcji, podprogramów, wspólnych zmiennych. Programy asemblerowe z natury roją się od nazw lokalnych, nieistotnych na zewnątrz.

AsmC umożliwia nadawanie symbolom nazw generowanych. Nazwę bezpośrednio poprzedzoną znakiem # asembler uzupełnia prefiksem numerycznym wstawianym zaraz za znakiem #, a przed pierwszym znakiem (literą) nazwy. Prefiks numeryczny jest jednakowy dla wszystkich nazw w danym tekście źródłowym. Każdy następny tekst, wstawiany do głównego przy pomocy instrukcji WSTAW, otrzymuje inny prefiks. Dzięki temu nazwy ze znakiem # są różne w różnych tekstach. Teraz wystarczy dbać o ich unikalność w ramach jednego tekstu.

Nazwy poprzedzone znakiem **#** są lokalne.

Prefiks numeryczny w nazwie generowanej jest ciągiem cyfr ósemkowych `01234567`. Prefiks w tekście głównym ma wartość 0, a w każdym następnym o 1 większą. Może składać się z jednej, dwóch, lub więcej cyfr. Asembler nie ogranicza długości prefiksu, ale trudno wyobrazić sobie ponad 511 instrukcji WSTAW w jednej asemblacji. Zakładając, że nigdy nie zdarzy się prefiks większy od 777, możemy śmiało dawać po znaku **#** nazwy 4-znakowe, a jeśli uznamy, że najdłuższy prefiks będzie 2-cyfrowy, możemy po **#** dawać nazwy 5-znakowe, które w połączeniu z prefiksem zmieszczą się w limicie 7 znaków.

W programach unikalnych, które nie będą wstawiane do innych programów, stosowanie nazw generowanych (lokalnych) w niczym nie pomaga, ale i nie przeszkadza. W programach uniwersalnych, które, jak przewidujemy, będą WSTAWiane w postaci źródłowej do innych programów, należy stosować nazwy lokalne z wyjątkiem tych, które muszą być znane globalnie.

Innym sposobem wstawienia tekstu dodatkowego do głównego jest scalenie tekstów w jeden na etapie przygotowywania tekstu głównego i ew. zmiana dublujących się nazw. Jeśli z jakiegoś powodu nie da się wtedy uniknąć konfliktu nazw, bądź nie chcemy ich zmieniać, to jest jeszcze możliwość osobnego zasemblowania podprogramu i wstawienia go do kodu wynikowego instrukcją WDANE lub WDANEX.

Przykłady nazw globalnych	*Przykłady nazw lokalnych*
`Program BLOK 0`	`#usuń A=0`
`Wtórne2 EQU *`	`#wtór EQU *`
`DodSłow A=A+[ŻÓŁĆ+B1]`	`#dodS A=A+[#ŻÓŁĆ+B1]`
`ŻÓŁĆ DS 1, 2`	`#ŻÓŁĆ DS 1, 2`

Odwołania do symboli i etykiet

Generalnie definicje nazw mogą znajdować się dalej w tekście źródłowym, niż odwołania do nich. Jednakże wartość nazwy musi dać się obliczyć w tej samej instrukcji, w której się ją definiuje, tj. w instrukcji, w której nazwa występuje jako etykieta.

Nazwy symboli i etykiet służą do odwoływania się do ich wartości w argumentach i wyrażeniach. AsmC wymaga, by po pierwszym przebiegu wszystkie nazwy używane w programie zostały zdefiniowane, tj. by każda znajdowała się w roli etykiety odpowiedniej instrukcji i jej wartość była obliczona, dzięki czemu możliwe jest obliczanie wartości wyrażeń i generowanie kodu w drugim przebiegu.

Szczególnym przypadkiem są wyrażenia, których wartość wpływa na wartość licznika lokacji (w instrukcjach BLOK, QBLOK, NAF-KOF), wybór klauzuli w instrukcji AIF-AELIF-AELSE-AFI, lub argumenty instrukcji WSTAW, WDANE, WDANEX, BYK i NOTA. Takie wyrażenia muszą dać się obliczyć już w pierwszym przebiegu translatora, by zarówno licznik lokacji, jak i następne etykiety rozkazów miały wartość. Pozostałe wyrażenia wystarczy, że dadzą się obliczyć w drugim przebiegu.

Etykieta instrukcji zwykle może być użyta jako argument tej instrukcji. W niektórych instrukcjach jest to jednak zabronione. W instrukcjach BLOK, QBLOK, EQU dlatego, że nazwa jest dopiero definiowana i jej wartość nie jest jeszcze obliczona. W instrukcjach NAF-KOF, WDANE i WDANEX, etykieta jest identyfikatorem i nie ma wartości. W instrukcji AIF etykieta jest identyfikatorem wszystkich instrukcji wchodzących w skład instrukcji złożonej AIF-AELIF-AELSE-AFI, ale ma też wartość licznika lokacji w miejscu instrukcji AIF, dlatego można się na nią powoływać w wyrażeniach. W instrukcji KONIEC etykieta jest całkowicie niedozwolona. Gdy asembler oddrukowuje instrukcję KONIEC jako pseudoinstrukcję OSTAW, to automatycznie nadaje jej, jako identyfikację, etykietę odpowiedniej instrukcji WSTAW, przy czym etykieta ta ma wartość licznika lokacji w miejscu instrukcji WSTAW.

Wartości adresowe

Instrukcje i rozkazy zwykle wymagają podania wartości uczestniczących bezpośrednio w operacji, albo służących do adresowania pamięci. Wartości te są lub mogą być częściami rozkazów, dlatego niezależnie od tego czy są to zwykłe liczby, czy adresy, będziemy je nazywać wartościami adresowymi, w odróżnieniu od zawartości słów pamięci, zaś wyrażenia matematyczne służące do ich obliczania wyrażeniami adresowymi.

Wartości adresowe w rozkazach są to wartości części N (AR) i części K (NR) rozkazu:

- argument bezpośredni rozkazu
- adres komórki pamięci zawierającej argument rozkazu
- adres komórki pamięci zawierającej wynik wykonania rozkazu
- adres komórki pamięci zawierającej następny rozkaz do wykonania

W instrukcji DS:

- składnik słowa liczbowego wieloadresowego
- krotność stałej (liczba powtórzeń)
- długość łańcucha słów tekstowych (atrybut S – liczba słów zajmowanych przez tekst)
  - ✗ ale nie skala słowa liczbowego stałoprzecinkowego
  - ✗ i nie cecha liczby zmiennoprzecinkowej

W pozostałych instrukcjach są to argumenty poleceń.

Zgodnie z budową rozkazów, wartość adresowa jest liczbą 13-bitową, oznaczającą albo liczbę binarną bez znaku, albo adres komórki pamięci operacyjnej, służącą jako argument bezpośredni rozkazu lub adres argumentu w pamięci operacyjnej. Taka wartość może być modyfikowana zawartością rejestru B-modyfikacji. AsmC rozróżnia cztery typy wartości adresowych:

Typ wartości adresowej *Akronim*	*Typ*	*Przykład*	*Opis*
Num	liczba prosta	`123`	liczba bez znaku, z zakresu 0...8191
NumX	liczba indeksowana	`123+B5`	liczba prosta modyfikowana rejestrem B-modyfikacji
Adr	adres prosty	`*`	adres komórki pamięci z zakresu 0...8191
AdrX	adres indeksowany	`*+B5`	adres prosty modyfikowany rejestrem B-modyfikacji

Wartość indeksowana powstaje jako wynik dodania wartości prostej i rejestru B-modyfikacji, i składa się z dwóch składowych: liczby i rejestru B-modyfikacji, przy czym może mieć tylko jeden rejestr w swoim składzie. Może to być dowolny rejestr B-modyfikacji – ograniczenie dopuszczalności danego rejestru zależy od tego, w której części adresowej rozkazu wyrażenie zostanie umieszczone:

– w części AR rozkazu – tylko rejestr B0, B1, B2, B3, B4, B5 lub B7

– w części NR rozkazu – tylko rejestr B0, B6 lub B7

– rejestr jako samodzielny argument – zgodnie z opisem instrukcji, zwykle dowolny rejestr B

Fikcyjny rejestr B0, zawsze zawierający wartość 0, może być używany tak samo jak pozostałe. Kontrola poprawności indeksowania odbywa się w chwili użycia wartości, przy wstawianiu jej do rozkazu, albo podawaniu jako argument instrukcji lub wyrażenia.

Liczba prosta może być podana bezpośrednio w postaci denotacji stałej liczbowej. Zawartość słowa definiowanego w instrukcji DS nie jest wartością w niniejszym sensie, gdyż nie może posłużyć do obliczenia części adresowej rozkazu. Adres takiego słowa jest wartością.

Każda wartość adresowa, dowolnego typu, może mieć nadaną w instrukcji EQU nazwę, która następnie jako symbol reprezentuje tę wartość w argumentach i wyrażeniach. W instrukcji EQU może powstać wartość adresowa dowolnego typu jako wynik odpowiedniego wyrażenia.

Licznik lokacji w BLOKu ma wartość typu adres prosty, równą wartości argumentu instrukcji BLOK powiększoną o odległość bieżącego słowa od początku BLOKu. Wartością etykiety rozkazu w BLOKu jest wartość licznika lokacji wskazującego adres bieżącej komórki pamięci operacyjnej.

Licznik lokacji w QBLOKu może mieć dowolny typ wartości (w odróżnieniu od BLOKu). Wartością etykiety rozkazu w QBLOKu jest wartość bazowa (argument instrukcji QBLOK) powiększona o odległość bieżącego słowa od początku QBLOKu. Suma ta jest wartością licznika lokacji w QBLOKu, wskazującego adres bezwzględny lub względny, indeksowany lub nie, typu zgodnego z typem argumentu instrukcji QBLOK.

Wartości etykiet innych instrukcji, i typy tych wartości, są podawane w opisach instrukcji.

W rozkazach często potrzebny jest argument typu adres – jeśli wtedy zostanie podany argument **n** typu liczbowego, to asembler zaakceptuje go tak, jakby był typu adresowego **n:n**. Podobnie tam gdzie może być podana wartość indeksowana, może też być podana wartość prosta. Z drugiej strony asembler nie pozwala, by w rozkazach przesunięć wielkość przesunięcia była typu adresowego i wymaga jawnej konwersji na typ liczbowy.

Liczby adresowe

Liczby adresowe, nie mylić ze słowami liczbowymi, są to liczby służące do obliczania wartości adresowych – 13-bitowych argumentów bezpośrednich, albo adresów argumentów/rozkazów podawanych w rozkazach, lub też argumentów instrukcji. Liczby adresowe mogą być co najwyżej 13-bitowe, bez znaku, z zakresu 0...8191. Liczby te mogą mieć różne denotacje: dziesiętne, binarne, ósemkowe, szesnastkowe i znakowe:

Liczby dziesiętne	– zapis przy pomocy cyfr dziesiątkowych 0...9
Liczby binarne	– zapis z prefiksem 0B przy pomocy cyfr dwójkowych 0...1
Liczby ósemkowe	– zapis z prefiksem 0C przy pomocy cyfr ósemkowych 0...7
Liczby szesnastkowe	– zapis z prefiksem 0X przy pomocy cyfr szesnastkowych 0...9,A...F
Liczby znakowe	– zapis podobny do zapisu łańcuchów tekstowych, zobacz: instrukcja DS

Litery oznaczające podstawę systemu pozycyjnego: 0B, 0C i 0X, oraz literowe cyfry szesnastkowe, mogą być wielkie lub małe.

Cyfry liczb dziesiętnych, binarnych, ósemkowych i szesnastkowych można dowolnie grupować przy pomocy znaku podkreślenia _.

Liczba znakowa musi zaczynać się od cyfrowego prefiksu wskazującego typ łańcucha tekstowego, bezpośrednio po którym musi być apostrof, i kończyć się zamykającym apostrofem. Należy pominąć literę T zapowiadającą atrybut typu, a pozostawić cyfrę. Poszczególne kody składające się na tekst są dosuwane do prawej strony (jak cyfry w liczbach), a nie do lewej (jak to jest w łańcuchach tekstowych definiowanych w instrukcjach DS). Kody znaków w tekście liczby znakowej traktowane są jak cyfry w systemie o podstawie: 32, 64 lub 128, dla tekstów odpowiednio: 5-, 6- lub 7-bitowych. Liczba znakowa nie zawiera bitów, które w tekstach niosą informację o liczbie kodów w słowie. Liczba znakowa nie ma też wyzerowanego słowa oznaczającego koniec tekstu, zatem kropka oznaczająca pominięcie tego słowa nie ma sensu.

Szczególną uwagę należy zwrócić na liczby znakowe typu 0' i 1' (toporne i ciosane), w których zawarte są również kody niewidoczne na wydruku. Kody te mogą być inne w trybie wielkich liter, niż w trybie małych liter. Niewidoczne kody zmiany pocztu mogą występować wielokrotnie. Szczegóły można znaleźć w opisie kodowania tekstów w instrukcji DS, ale uwaga: definicje słów liczbowych w instrukcjach DS mają nieco inną składnię i semantykę niż liczby adresowe.

Ponieważ stała liczbowa może mieć co najwyżej 13 bitów, to liczby znakowe mogą zawierać maksymalnie 3 znaki (nie licząc dowolnej liczby kodów NU na początku łańcucha tekstowego). Najstarsze bity, nie mieszczące się w 13-bitowej wartości, nie powodują błędu jeśli są zerowe.

Przykłady liczb adresowych

Denotacja	*Układ bitów*	*Wartość dziesiętna*	*Kody znakowe*	*Objaśnienie*
123		=123		
0b_00011_00101_11100		=3260		
0c17700		=8128		
0x1FC0		=8128		
0'5'	=0b000_00000_10000	=16	={5}	
0'AX'1	=0b000_11111_00011	=995	={▼A}	=31*32+3
1'XA'	=0b000_00011_11011	=123	={A▲}	=3*32+27
1'XAX'1	=0b000_00000_00011	=3	={A}	
2'AB'	=0b000_00011_11001	=121	={AB}	=3*32+25
6'AB'	=0b0_000011_011001	=217	={AB}	=3*64+25
7'AB'	=0b_000011_0011001	=409	={AB}	=3*128+25
9'AB'	=0b_111000_0111001	=7225	={AB}	=56*128+57

MANIFEST

Wzorem niektórych języków programowania, jako wyróżnik liczb ósemkowych przyjęto 0c. Wyróżnik 0o stosowany w innych językach, szczególnie z wielką literą O, jest zbyt podobny do dwóch zer. Wyróżnik 0C ma tę zaletę, że nawet napisany wielkimi literami odróżnia się od cyfr. Ponadto 0c można czytać jako skrót od „Octal”, co doskonale oddaje znaczenie notacji.

Również dla liczbowych kodów znaków przyjęto jawne wskazywanie po znaku / podstawy liczbowej (nawet dziesiętnej): /c, /d, /x, z dokładnie dwiema cyframi.

Licznik lokacji (licznik rozkazów)

Asembler prowadzi licznik lokacji wskazujący bieżące miejsce w pamięci, w którym następuje asemblacja rozkazu lub danej. Początkowo jest to adres komórki nr 0. Adres ten można ustawić przy pomocy instrukcji BLOK lub QBLOK, oraz modyfikować instrukcjami NAF i KOF. Wartość licznika lokacji rośnie o 1 modulo 8192 po wygenerowaniu kolejnego słowa pamięci.

Podczas asemblowania BLOKu, licznik rozkazów ma wartość typu adres prosty. W pozornym bloku (w QBLOKu) może mieć wartość każdego z czterech typów. W wyrażeniach adresowych, w instrukcji lub rozkazie, można odwoływać się do licznika rozkazów poprzez trzy symbole, których można używać tak, jak gdyby to były nazwy:

***** – wartość licznika lokacji w danym miejscu programu

++ – równoważny wyrażeniu `(*+1)`

-- – równoważny wyrażeniu `(*-1)`

Wyrażenia adresowe

Do wskazywania wartości części adresowych rozkazów, zarówno adresowych jak i liczbowych (przy czym liczbowe zwykle mogą być stosowane w roli adresowych) często stosuje się wyrażenia adresowe. Wyrażenia adresowe służą do obliczania wartości adresowych na podstawie innych wartości adresowych.

Wyrażenie adresowe jest wyrażeniem matematycznym, w którym na argumentach wykonuje się operacje przy pomocy operatorów arytmetycznych i logicznych. Obliczenia wykonuje się w kolejności od lewej do prawej, zgodnie z priorytetami operacji, w pierwszej kolejności operacje o wyższym priorytecie. Obliczane jest całe wyrażenie, nawet jeśli wynik jest przesądzony.

Kolejność wykonywania operacji można zmieniać przy pomocy nawiasów okrągłych. Wartość wyrażenia ujętego w nawiasy jest obliczana najpierw, a następnie staje się argumentem zewnętrznego wyrażenia.

Argumenty wyrażeń adresowych

Wyrazami wyrażeń adresowych – argumentami – mogą być:

- liczby naturalne 0..8191: dziesiętne, binarne, ósemkowe, szesnastkowe i znakowe, np.:

```
2289      0b101       0c127      0xC30
0'365'    1'XABX'1    2'/f12'    6'AB'     7'AB'     9'AB'
```

- etykiety stojące przed instrukcjami, np.:

```
ETYK    DS  1234
```

- symbole zdefiniowane w instrukcjach EQU, np.:

```
SYMB    EQU 1+B2
```

- licznik lokacji: `*  ++  --`
- rejestry B-modyfikacji: `B0 B1 B2 B3 B4 B5 B6 B7`
- wyrażenia ujęte w nawiasy okrągłe, np.: `(KON-POCZ)`

Priorytety operacji w wyrażeniach adresowych

Operator ? testu istnienia nazwy jest jednoargumentowy, a wszystkie pozostałe dwuargumentowe. Są to w kolejności priorytetów od najwyższego do najniższego:

Priorytet	*Operator*	*Operacja*	*Przykłady*
11	**?**	test istnienia nazwy (czy nazwa jest zdefiniowana)	`?nazwa → 0 (nie istnieje)` `?nazwa → 1 (istnieje)`
10	**:**	przeadresowanie (zmiana ścieżki)	`0c17200:0c04567 → 0c17367`
9	*****	mnożenie	`4*13 → 52`
	/	dzielenie	`13/4 → 3`
	%	reszta z dzielenia	`13%4 → 1`
	<<	przesunięcie w lewo	`0b1101<<4 → 0b11010000`
	>>	przesunięcie w prawo	`0b11101101>>4 → 0b1110`
8	**+**	dodawanie	`ETYK+1 ETYK+B5`
	-	odejmowanie	`*-ETYK ETYK-1`
7	**&**	bitowa koniunkcja (BAND)	`13&7 → 5`
6	**^**	bitowa różnica symetryczna (BXOR)	`13^14 → 3`
5	**|**	bitowa alternatywa (BOR)	`52|7 → 55`
4	**<**	mniejsze	`1<2 → 1 (prawda)`
	<=	mniejsze/równe	`3<=2 → 0 (fałsz)`
	>	większe	`1>2 → 0 (fałsz)`
	>=	większe/równe	`3>=2 → 1 (prawda)`
3	**==**	równe	`1==2 → 0 (fałsz)`
	<>	nierówne	`1<>2 → 1 (prawda)`
2	**&&**	wybór wg koniunkcji (AND)	`0&&PRAWY = 0 1&&PRAWY = PRAWY`
1	**||**	wybór wg alternatywy (OR)	`0||PRAWY = PRAWY 1||PRAWY = 1`
	<|	wybór mniejszego (MIN)	`5<|4 = 4 5<|5:5 = 5:5`
	>|	wybór większego (MAX)	`5>|4 = 5 5>|5:5 = 5:5`

Przykłady wyrażeń z nawiasami:

`(61*128):ETYK` Nawiasy zmieniają priorytet operacji: najpierw ma być tu wykonane mnożenie, a dopiero potem przeadresowanie

`AL 1+4` Instrukcja pozwalająca opuścić nawiasy bez zmiany semantyki

`A=A+(123+456)` Pierwszy znak + wyznacza rozkaz dodawania do akumulatora argumentu bezpośredniego, który musi być ujęty w nawias, bo nie istnieje rozkaz `A=A+123+456`

`A=-(123+456)` Znak – wyznacza rozkaz zmiany znaku argumentu bezpośredniego, wskazanego wyrażeniem w nawiasie, bo nie istnieje rozkaz `A=-123-456`

Wynik każdej operacji jak i całego wyrażenia jest wartością jednego z czterech typów: liczbą prostą, liczbą indeksowaną, adresem prostym lub adresem indeksowanym. Każda częściowa i ostateczna wartość jest obcinana do 13 bitów. Żadna wartość, ani częściowa, ani końcowa, nie może zawierać w sobie więcej niż jeden rejestr B-modyfikacji, nawet jeśli w następnych operacjach nastąpiłaby redukcja indeksowania.

Wyrażenie może być użyte wszędzie tam, gdzie potrzebny jest argument jednego z tych typów. W szczególności wyrażenie może być argumentem wyrażenia.

Generalnie, argument rozkazu w postaci wyrażenia adresowego może być ujęty w nawiasy. W niektórych rozkazach jest to wymóg – i wtedy nawiasy mogą być pominięte tylko gdy argument nie jest wyrażeniem (nie zawiera żadnych operatorów), lub już jest w nawiasach kwadratowych (argument pamięciowy). Otoczenie argumentu nawiasami zmienia go w wyrażenie, a tym samym zmienia semantykę niektórych instrukcji:

Przykład	*Rozkaz :TPG*	*Objaśnienie*
`A = 0`	`:040 00000 *00001 00`	Rozkaz zerowania rejestru A Część AR rozkazu mogłaby zawierać cokolwiek
`A = (0)`	`:070 00000 *00002 00`	Rozkaz załadowania A argumentem bezpośrednim zawartym w części AR, równym w tym przypadku 0
`A = 1`	`:070 00001 *00003 00`	Rozkaz załadowania A argumentem bezpośrednim, równym w tym przypadku 1 (nawiasy pominięte)
`A = (1)`	`:070 00001 *00004 00`	Rozkaz załadowania A argumentem bezpośrednim, równym w tym przypadku 1
`A = (B5+1)`	`:070 00001+B5 *00005 51`	Rozkaz załadowania A argumentem bezpośrednim, równym w tym przypadku 1 z B-modyfikacją
`A = (B5+0)`	`:070 00000+B5 *00006 51`	Rozkaz załadowania A argumentem bezpośrednim, równym w tym przypadku 0 z B-modyfikacją
`A = (B5)`	`:070 00000+B5 *00007 51`	Rozkaz załadowania A argumentem bezpośrednim, równym w tym przypadku 0 z B-modyfikacją
`A = B5`	`:060 00000 *00010 50`	Rozkaz załadowania do A zawartości rejestru B5
`A = B5+1`	błąd składniowy	Nie istnieje rozkaz dodawania zawartości rejestru B i argumentu bezpośredniego (liczby)
`A = 1+B5`	błąd składniowy	Nie istnieje rozkaz dodawania argumentu bezpośredniego (liczby) i zawartości rejestru B
`A = |X|Y|`	błąd składniowy	Operator | (BOR) w wyrażeniu zostanie wzięty za zamknięcie wartości bezwzględnej rozkazu

Widać, jak zmienia się operacja TPG i część modyfikacyjna BZ wygenerowanych rozkazów. Argument 0 bez nawiasów może być liczbą adresową w dowolnej denotacji, byleby nie symbolem i nie wyrażeniem, bo wtedy zostanie wygenerowany rozkaz jak dla argumentu niezerowego. Również argument B5 w nawiasach oznacza liczbę indeksowaną, zaś bez nawiasów oznacza zawartość rejestru B5, co powoduje wygenerowanie innego rozkazu.

Pominięcie nawiasów w dwóch przedostatnich instrukcjach jest błędem, ponieważ nie istnieje rozkaz dodawania argumentu bezpośredniego (liczby) i zawartości rejestru B-modyfikacji. W ostatnim przykładzie pierwszy znak | wyznacza rozkaz liczenia wartości bezwzględnej argumentu i oczekiwany jest taki sam znak | zamykający. Argument jest wyrażeniem adresowym

zawierającym operator | wzięty za zamknięcie rozkazu – znaki | wyznaczające rozkaz liczenia wartości bezwzględnej nie zwalniają z wymogu ujęcia argumentu w nawiasy.

Szczegóły składniowe i różnice semantyczne związane są z repertuarem rozkazów ODRY i są objaśnione przy opisie rozkazów.

UWAGA: W języku AsmC zapis wielu rozkazów upodabnia je do instrukcji np. języka C. Jest to tylko pozór, bowiem wszystkie je pisze się według ustalonych szablonów podanych przy opisie rozkazów, a przyjęta notacja ułatwia jedynie rozumienie ich funkcji. Szablony rozkazów zawierają w sobie operatory odpowiadające rozkazom, czyli wyznaczające rozkazy. Operatory w szablonach rozkazów nie wchodzą w skład wyrażeń adresowych – są to separatory. Ilustrują to trzy ostatnie przykłady z błędem składniowym, znak + jest nieoczekiwanym separatorem, a nie operatorem dodawania w wyrażeniu adresowym, zaś nazwa Y niespodziewanym i nieznanym argumentem.

MANIFEST

Priorytety operacji różnią się od tych z języka C, C++, JavaScript i pochodnych.

Zgodnie z krytyczną uwagą Dennisa Ritchiego, twórcy języka C, operacje bitowe koniunkcji (BAND), różnicy symetrycznej (BXOR) i alternatywy (BOR), mają w AsmC priorytety wyższe od operacji relacji porządkujących i porównań.

Dodatkowo w AsmC priorytet przesunięć << i >> został podniesiony i zrównany z operacjami *, / i %. W https://en.wikipedia.org/wiki/Bitwise_operation#Bit_shifts mamy uzasadnienie – przesunięcia arytmetyczne definiuje się jako równoważne mnożeniu i dzieleniu przez potęgę liczby 2. W programach asemblerowych nagminnie zamienia się mnożenia i dzielenia na przesunięcia, a to przy tradycyjnych priorytetach groziłoby błędami wskutek przeoczania braku koniecznych nawiasów.

Operator ? testu istnienia nazwy

? `?x` **Test istnienia nazwy**

Operator sprawdza, czy nazwa x istnieje gdzieś wcześniej jako etykieta instrukcji. Wynikiem testu jest liczba:

0 (FAŁSZ) – gdy nazwa x nie istnieje

1 (PRAWDA) – gdy nazwa x istnieje

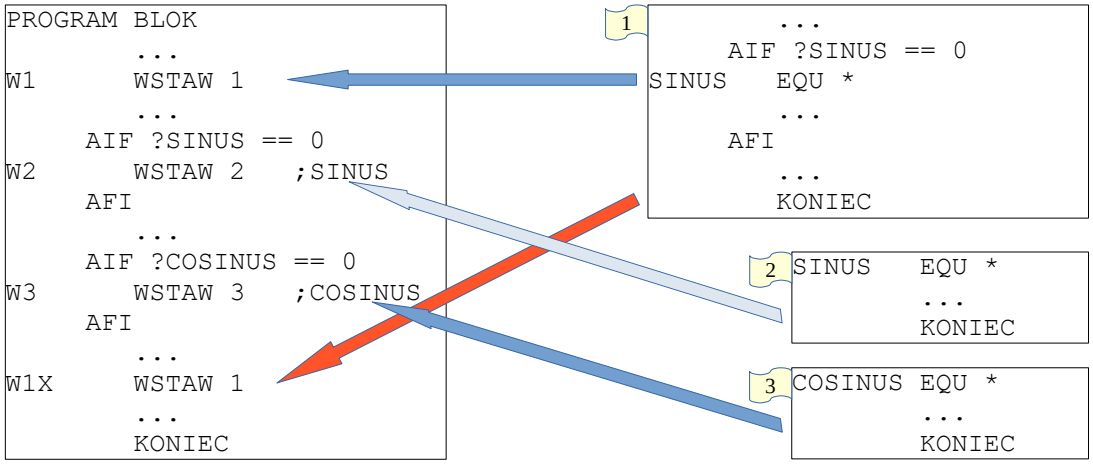
Tabelka wyników

x	?x
Nazwa x nie istnieje (jeszcze brak jej jako etykiety)	0
Nazwa x istnieje (już jest jako etykieta)	1

Operator ? jest jedynym operatorem jednoargumentowym i szczególnym ze względu na swoją rolę. Jego argumentem musi być definiowalna nazwa: symbol (w tym symbol predefiniowany), etykieta rozkazu, etykieta instrukcji, lub identyfikator. Może nim być nazwa globalna lub lokalna.

Argumentem nie może być symbol zastrzeżony: oznaczenie rejestru, słowo kluczowe instrukcji, mnemonik rozkazu lub opcji. Argumentem nie może też być wartość, znak specjalny, operator, ani wyrażenie (np. nazwa nie może być ujęta w nawiasy).

Operator ? nie tworzy odwołania do nazwy, nie wymusza jej definiowania, jak to się dzieje w innych kontekstach, gdy nazwa użyta musi być gdzieś zdefiniowana. Test istnienia nazwy jest prostą namiastką makroinstrukcji. Pozwala łatwiej sterować umiejscowieniem definicji nazw, np. unikać kolizji wielokrotnego wstawiania podprogramu:



W przykładzie pokazanym na rysunku, wskutek wykonania instrukcji W1, do tekstu programu zostanie wstawiona treść taśmy nr 1 wraz z nazwą SINUS. Instrukcja W2 nie zostanie wykonana, bo nazwa SINUS już znalazła się w tekście programu (została zdefiniowana) wskutek uprzedniego wykonania polecenia W1, natomiast instrukcja W3 zostanie wykonana i do tekstu programu zostanie wstawiona treść taśmy nr 3, bo nazwy COSINUS jeszcze nie ma (nie jest zdefiniowana). Gdyby treść taśmy nr 1 wstawiano w instrukcji W1X, a nie w W1, to funkcja SINUS w jej treści zostałaby ominięta, bowiem znajduje się w klauzuli AIF-AFI, której warunek stwierdziłby, że nazwa SINUS już jest zdefiniowana (przy braku instrukcji W1 wykonana byłaby wcześniejsza instrukcja W2 wciągająca nazwę SINUS wraz z treścią taśmy nr 2).

Operacje rachunkowe wyrażeń adresowych

UWAGI:

- W podanych tu definicjach operatorów, symbolami Num, NumX, Adr i AdrX oznaczono typy argumentów i wyników.
- Część liczbowa wyniku operacji obliczana jest jako liczba 13-bitowa bez znaku, modulo 8192. Składnik indeksowy B jest trzymany jako osobna część, oznaczająca B-modyfikację.
- Wynikiem operacji <, <=, >, >=, == i <> jest liczba 1=PRAWDA lub 0=FAŁSZ.
- W operacjach :, <, <=, >, >=, == i <> argument liczbowy jest konwertowany, gdy trzeba, do adresowego.
- W operacjach && i ||, oraz w instrukcjach AIF i AELIF, argument liczbowy typu Num o wartości 0 jest traktowany jako fałszywy, a każdy inny jako prawdziwy.

`:` x:y **Przeadresowanie**

Operator oblicza adres biorąc z pierwszego argumentu x najstarsze 6 bitów wartości jako numer ścieżki bębnowej, natomiast z drugiego y najmłodsze 7 bitów wartości jako numer strefy na ścieżce:

Argument x = xxxxxxsssssss
Argument y = tttttt yyyyyyy
Wynik x:y = xxxxxxyyyyyyy

Tabelka typów wyników

x:y		Drugi argument			
		:Num	:NumX	:Adr	:AdrX
Pierwszy argument	Num	=Adr	–	=Adr	–
	NumX	–	–	–	–
	Adr	=Adr	–	=Adr	–
	AdrX	–	–	–	–

`*` x*y **Mnożenie**
`/` x/y **Dzielenie**
`%` x%y **Reszta z dzielenia**
`<<` x<<y **Przesunięcie w lewo**
`>>` x>>y **Przesunięcie w prawo**

Są to zwykłe działania na binarnych liczbach naturalnych
x<<y jest równoważne $x*(2^y)$ modulo 8192
x>>y jest równoważne $x/(2^y)$ modulo 8192

Symbol ♠ w tabelce oznacza jeden operator, wybrany z powyższych

Tabelka typów wyników

x♠y		Drugi argument			
		♠Num	♠NumX	♠Adr	♠AdrX
Pierwszy argument	Num	=Num	–	–	–
	NumX	–	–	–	–
	Adr	–	–	–	–
	AdrX	–	–	–	–

`+` x+y **Dodawanie**

Jest to zwykłe dodawanie modulo 8192 binarnych liczb naturalnych, z zachowaniem jako składnika rejestru B-modyfikacji.

Indeksowanie dwoma rejestrami B-modyfikacji jest niedozwolone.

Tabelka typów wyników

x+y		Drugi argument			
		+Num	+NumX	+Adr	+AdrX
Pierwszy argument	Num	=Num	=NumX	=Adr	=AdrX
	NumX	=NumX	–	=AdrX	–
	Adr	=Adr	=AdrX	–	–
	AdrX	=AdrX	–	–	–

`-` `x-y` **Odejmowanie**

Jest to zwykłe odejmowanie modulo 8192 binarnych liczb naturalnych, z zachowaniem rejestru B-modyfikacji jako składnika odjemnej, bądź skasowaniem się rejestrów indeksujących odjemną i odjemnik[1].

Rejestr B-modyfikacji występujący jedynie w odjemniku jest niedozwolony.

Tabelka typów wyników

x−y		*Drugi argument*			
		−Num	−NumX	−Adr	−AdrX
Pierwszy argument	Num	=Num	–	–	–
	NumX	=NumX	=Num[1]	–	–
	Adr	=Adr	–	=Num	–
	AdrX	=AdrX	=Adr[1]	=NumX	=Num[1]

[1]) Dozwolone tylko wtedy, gdy oba operandy są indeksowane tym samym rejestrem B-modyfikacji

`&` `x&y` **Bitowa koniunkcja**
`^` `x^y` **Bitowa różnica symetryczna**
`|` `x|y` **Bitowa alternatywa**

Są to zwykłe operacje logiczne, równolegle na wszystkich bitach liczb naturalnych.

```
Argument x =        0111100001100
Argument y =        0110011001010
Wynik   x&y =       0110000001000
Wynik   x^y =       0001111000110
Wynik   x|y =       0111111001110
```

Tabelka typów wyników

x♣y		*Drugi argument*			
		♣Num	♣NumX	♣Adr	♣AdrX
Pierwszy argument	Num	=Num	–	=Num	–
	NumX	–	–	–	–
	Adr	=Num	–	=Num	–
	AdrX	–	–	–	–

Symbol ♣ w tabelce oznacza jeden operator, wybrany z powyższych

`<` `x<y` **Mniejsze**
`<=` `x<=y` **Mniejsze lub równe**
`>` `x>y` **Większe**
`>=` `x>=y` **Większe lub równe**

Wynikiem relacji porządkującej jest liczba:

0 (FAŁSZ) – gdy relacja jest nieprawdziwa

1 (PRAWDA) – gdy relacja jest prawdziwa

Tabelka typów wyników

x♦y		*Drugi argument*			
		♦Num	♦NumX	♦Adr	♦AdrX
Pierwszy argument	Num	=Num	–	=Num	–
	NumX	–	=Num[1]	–	=Num[1]
	Adr	=Num	–	=Num	–
	AdrX	–	=Num[1]	–	=Num[1]

Symbol ♦ w tabelce oznacza jeden operator, wybrany z powyższych

[1]) Dozwolone tylko wtedy, gdy oba operandy są indeksowane tym samym rejestrem B-modyfikacji

== x==y **Równe**
<> x<>y **Nierówne**

Wynikiem relacji porównania jest liczba:
0 (FAŁSZ) – gdy relacja jest nieprawdziwa
1 (PRAWDA) – gdy relacja jest prawdziwa
Operandy indeksowane różnymi rejestrami są nierówne.

Tabelka typów wyników

x♥y	*Drugi argument* ♥Num	♥NumX	♥Adr	♥AdrX
Pierwszy argument Num	=Num	=Num^2	=Num	=Num^2
NumX	=Num^2	=Num	=Num^2	=Num
Adr	=Num	=Num^2	=Num	=Num^2
AdrX	=Num^2	=Num	=Num^2	=Num

Symbol ♥ w tabelce oznacza jeden operator, wybrany z powyższych

[2]) Wynik z góry przesądzony: dla <> wynosi 1=PRAWDA, a dla == wynosi 0=FAŁSZ

&& x&&y **Wybór wg koniunkcji**
|| x||y **Wybór wg alternatywy**

Wynikiem operacji x && y jest x, gdy x jest fałszywy, zaś y w przeciwnym razie:
0 && y daje w wyniku 0
1 && y daje w wyniku y
Wynikiem operacji x || y jest x, gdy x jest prawdziwy, zaś y w przeciwnym razie:
0 || y daje w wyniku y
5 || y daje w wyniku 5

Tabelka typów wyników

x◀y	*Drugi argument* ◀Num	◀NumX	◀Adr	◀AdrX
Pierwszy argument Num	Lewy lub prawy argument i jego typ[3]			
NumX	Lewy argument jest prawdziwy, więc wartość i typ wyniku są z góry przesądzone[3]			
Adr				
AdrX				

Symbol ◀ w tabelce oznacza jeden operator, wybrany z powyższych

[3]) Liczby niezerowe, oraz wszystkie adresy i wartości indeksowane, są prawdziwe, w tym (0:0), (0:0+Bb), (0+Bb)
Liczba 0 jest fałszywa

<| x<|y **Wybór mniejszego (MIN)**
>| x>|y **Wybór większego (MAX)**

Wynikiem operacji x <| y jest x, gdy x<y, jak nie to y:
5 <| 4 daje w wyniku 4
5 <| (5:5) daje w wyniku 5:5
Wynikiem operacji x >| y jest x, gdy x>y, jak nie to y:
5 >| 4 daje w wyniku 5
5 >| (5:5) daje w wyniku 5:5
Typ wyniku m to typ wynikowego operandu

Tabelka typów wyników

x▶y	*Drugi argument* ▶Num	▶NumX	▶Adr	▶AdrX
Pierwszy argument Num	=m[4]	–	=m[4]	–
NumX	–	=m[1,4]	–	=m[1,4]
Adr	=m[4]	–	=m[4]	–
AdrX	–	=m[1,4]	–	=m[1,4]

Symbol ▶ w tabelce oznacza jeden operator, wybrany z powyższych

[1]) Dozwolone tylko wtedy, gdy oba operandy są indeksowane tym samym rejestrem B-modyfikacji

[4]) Porównania są wykonywane tak, jak dla relacji porządkujących <, <=, >, >=

Przykładowe konwersje typów i wartości

Dodanie indeksu: wystarczy suma. Wartość indeksowana składa się z dwóch części: liczby i rejestru B-modyfikacji. Jeśli zostanie użyta w rozkazie, to liczba znajdzie się w odpowiedniej części adresowej rozkazu, a numer rejestru wraz z bitem Z=1 w części BZ rozkazu. Treść rejestru będzie dodawana do tej liczby (a właściwie do rozkazu) podczas pobierania go do wykonania.	`Liczba+B5` `Adres+B5`
Usunięcie indeksu: wystarczy odjąć rejestr B-modyfikacji Odjęcie rejestru B-modyfikacji od wartości indeksowanej usuwa z niej składową, jaką był w niej ten rejestr. Pozostanie sama część liczbowa. Taka różnica, użyta jako argument rozkazu, nie generuje bitu Z=1 w części BZ rozkazu.	`LiczbaX-B5` `AdresX-B5`
Konwersja liczby na adres: albo przy pomocy operacji przeadresowania, albo przez dodanie adresu bazowego	`Liczba:Liczba` `Liczba+0:0`

Konwersja adresu na liczbę: albo przez odjęcie adresu bazowego, albo operacją bitową akceptującą argumenty liczbowe i adresowe	`Adres-0:0`
	`Adres|0`
	`AdrKOŃC-AdrPOCZ`

Standaryzacja – zamiana dowolnej wartości na standardową logiczną: 1=PRAWDA lub 0=FAŁSZ	`Wartość<>0`
Negacja wszystkich bitów liczby	`Liczba^0b1111111111111` `0c17777-Liczba`
Negacja: PRAWDA na FAŁSZ, a FAŁSZ na PRAWDA	`L==0`

Zmiana znaku liczby: liczby adresowe są bez znaku, wyrażenia są obliczane modulo 8192	`0-Liczba`
	`0-1=8191`
	`0-2=8190`

Korektor ! kodu operacji rozkazu

Wynik wyrażenia adresowego jest parą składającą się z liczby 13-bitowej bez znaku i ewentualnego rejestru B-modyfikacji. Składnik liczbowy jest zawsze obliczany modulo 8192, a ew. nadmiarowe bity obcinane. To właśnie składnik liczbowy jest umieszczany w części adresowej rozkazu, jeśli wyrażenie adresowe jest argumentem rozkazu. Zawartość rejestru B-modyfikacji będącego składnikiem wyrażenia adresowego będzie dodawana do składnika liczbowego w odpowiedniej części adresowej (lub do całego słowa rozkazowego) podczas pobierania rozkazu z pamięci. Bity nadmiarowe podczas B-modyfikacji przenoszą się na starsze pozycje słowa, w szczególności na kod operacji rozkazu. Szczególnie ryzykowne są wyrażenia takie jak np. `(B1-4)`, gdzie łatwo

zapomnieć, że takie odejmowanie jest w istocie dodawaniem wielkiej liczby, jak np. `(B1+8188)`. Dla zobrazowania, jak różni się dodawanie liczby od dodawania rejestru B, rozpatrzmy przykład.

Wyrażenie np. `(0c12345+0c06701)` ma wartość `0c1246`, która znajdzie się w odpowiedniej części adresowej po podaniu tego wyrażenia jako argumentu rozkazu:

`A=(0c12345+0c06701)`	czyli	`:070 01246 00000 00`

Natomiast wyrażenie `(0c12345+B1)`, przy B1 zawierającym również liczbę `0c06701`, podane jako argument rozkazu, da podczas B-modyfikacji sumę `0c21246`, której najstarszy bit zostanie przeniesiony na starsze bity rozkazu, zmieniając go. Tak więc rozkaz:

`A=(0c12345+B1)`	czyli	`:070 12345 00000 11`

po B-modyfikacji rejestrem B1, czyli dodaniu liczby `:000 06701 00000 00`

zmieni się na:

`[0c1246]=A=(0c1246)`	czyli	`:071 01246 00000 11`

B-modyfikacja pierwszej części adresowej (argumentu lub adresu argumentu) może zmienić kod operacji, zmieniając wymagany rozkaz w jakiś inny, B-modyfikacja drugiej części adresowej (adresu następnego rozkazu) może zmienić zarówno pierwszą część adresową jak i kod operacji, zaś B-modyfikacja rejestrem B7 może zmienić dowolne części rozkazu.

Asembler nie potrafi, ale programista powinien określić czy w danym rozkazie zawsze nastąpi przeniesienie spowodowane mechanizmem B-modyfikacji i odpowiednio zmodyfikować kod rozkazu zapisując go w notacji :TPG. Sposób ten ma jednak zasadniczą wadę: rozkaz staje się niezrozumiały – nie dość że zapisany cyferkami, to jeszcze kodujący inną operację niż potrzebna. AsmC oferuje korektor kodu operacji w postaci prefiksowego znaku wykrzyknika:

```
etyk  !  rozkaz                    ;komentarz liniowy
```

Znak wykrzyknika **!** przed rozkazem (a po etykiecie) zmniejsza o 1 kod operacji, który podczas wykonywania rozkazu zostanie z powrotem powiększony o 1 – o bit przeniesienia z dalszych części podczas B-modyfikacji. Powyższy problematyczny rozkaz może więc zostać zapisany jako:

`! A=(0c12345+B1)`	czyli	`:067 12345 00000 11`

który po B-modyfikacji rejestrem B1, tj. dodaniu liczby `:000 06701 00000 00`

zmieni się na wymagany:

`A=(0c1246)`	czyli	`:070 01246 00000 11`

Zwróćmy uwagę, że taka korekcja rozkazu ma sens tylko wtedy, gdy przeniesienie nastąpi z całą pewnością, w przeciwnym razie należy użyć innych rozkazów. Przy tym znak wykrzyknika koryguje jedynie kod operacji. Korekty innych części rozkazu należy dokonywać za pomocą odpowiednich modyfikacji wartości adresowych.

Korektor ! może być postawiony tylko przed instrukcją maszynową (rozkazem). Postawiony przed instrukcją DS, instrukcją asemblera (dyrektywą), czy instrukcją warunkowej asemblacji spowoduje błąd składniowy.

Instrukcje asemblera

Instrukcje asemblera służą ułatwieniu programowania poprzez wprowadzanie symboli, wstawianie gotowych, wcześniej przygotowanych tekstów źródłowych lub gotowych kodów programów i danych, zadawanie wymaganego adresowania pamięci, itp. Są to informacje wpływające na sposób generowania kodu programu, choć same w sobie go nie generują.

Składnia instrukcji jest prosta – po opcjonalnej etykiecie następuje słowo kluczowe instrukcji poprzedzone odstępem, a po nim, też oddzielone odstępem, argumenty:

```
etyk     słKlucz argumenty          ;komentarz liniowy
symb     słKlucz argumenty          ;komentarz liniowy
id       słKlucz argumenty          ;komentarz liniowy
```

Niektóre instrukcje nie mają argumentów. Niektóre argumenty są opcjonalne i wtedy przyjmowane są wartości domyślne. Argumenty instrukcji asemblera nie muszą być ujmowane w nawiasy.

Instrukcja EQU – definicja symbolu

```
symb     EQU     adresX          ;definicja symbolu 'symb'
```

Instrukcja EQU definiuje symbol **symb** o wartości równej wartości argumentu **adresX**.

Argument **adresX** jest wymagany i może być dowolnego typu: Num, NumX, Adr lub AdrX. Może to być dowolne wyrażenie adresowe obliczalne w pierwszym przebiegu asemblera, nie zawierające swojej etykiety **symb** – wyrażenie to może, ale nie musi być ujęte w nawiasy.

Etykieta **symb** nie jest obowiązkowa, jednak jej brak oznacza, że symbol nie jest definiowany. Instrukcja EQU bez etykiety może posłużyć obliczeniu wartości wyrażenia **adresX**, listowanej następnie w wydruku kodu programu i zaspokajającej ciekawość programisty. Może też posłużyć do sprowokowania komunikatu błędu w przypadku, gdy wyrażenie **adresX** będzie nieprawidłowe.

Przykłady:

```
start    EQU     *                ;początek pętli
zacznij                           ;równoważne 'zacznij EQU *'
MASKA    EQU     0b0000111000000  ;maska do wycinania trzech bitów
MAXTab   EQU     128              ;maksymalna wielkość tablicy
adrSumy  EQU     0c17574+0:0      ;adres sumy kontrolnej
ileTab   EQU     (konT-poczT)/2   ;liczba elementów tablicy
indTab1  EQU     poczT+B2+1       ;adres drugiego słowa el.tabl.
         EQU     1/ileTab         ;gdy BYK - tablica jest pusta
literaA  EQU     2'A'             ;kod ITA litery A lub a
znowu    EQU     start            ;'znowu' jest synonimem 'start'
```

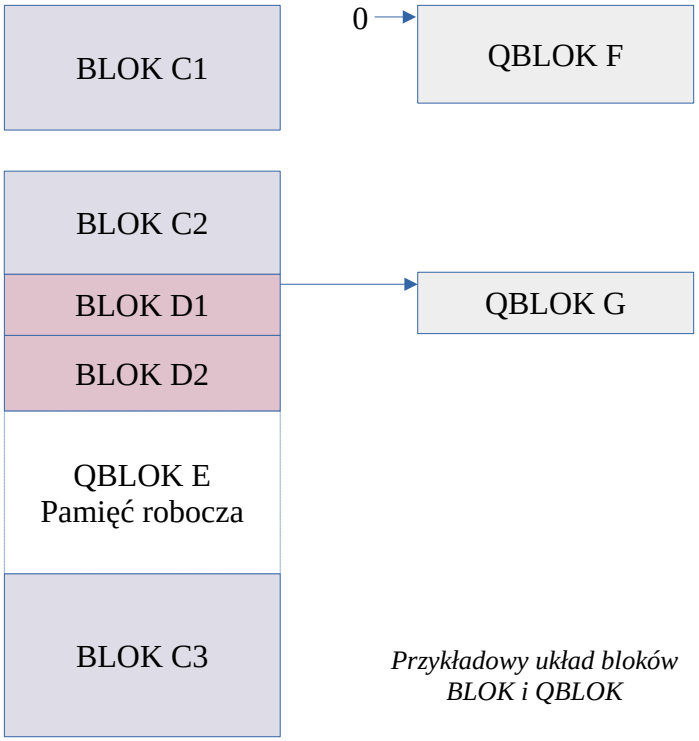
Instrukcje rozmieszczania bloków kodu w pamięci

Program wykonywalny składa się z bloków kodu programu.

Program źródłowy składa się z BLOKów kodu i bloków pozornych (QBLOKów).

Blok kodu programu, definiowany instrukcją BLOK, jest to obszar pamięci operacyjnej, w którym mają znajdować się rozkazy i dane wygenerowane przez asembler. Treść bloku programu znajdzie się na taśmie wynikowej z kodem PIĄTKOWYM, opatrzona pilotem zawierającym adres początku bloku. Dla wykonania programu, BLOKi składające się nań muszą być uprzednio załadowane z taśmy PIĄTKOWEJ do pamięci operacyjnej pod odpowiednie adresy – robi to program wprowadzający programu STAŁEGO.

Blok pozorny, definiowany instrukcją QBLOK, nie generuje kodu (słów programu), lecz tylko opisuje postać obszaru pamięci operacyjnej, który już zawiera jakieś dane lub rozkazy, będzie zapełniany danymi, albo jest obszarem roboczym programu. Blok pozorny może być zakotwiczony na stałe do pewnego adresu pamięci, lub stanowić formatkę przykładaną do różnych obszarów poprzez dynamiczne dodawanie adresu bazowego lub indeksu w poszczególnych instrukcjach. Obszarem tym może być np. element tablicy struktur, struktura będąca parametrem podprogramu, zestaw danych przygotowanych przez inny program, zestaw danych przygotowywany innemu programowi, zgrupowanie zmiennych roboczych, itp. Blok pozorny pozwala ponazywać elementy tego obszaru, umożliwiając symboliczne odwoływanie się do różnych lokacji, choćby znajdujących się poza programem, czym ułatwia programowanie.



Przykładowy układ bloków BLOK i QBLOK

```
C1  BLOK 0
    B1 = [Gp2]    ;adr.param. z D1
    A  = [Fv2+B1];2. słowo param.
    [Ex] = A      ;do kom.rob. w E
    ...
C2  BLOK 0c01000
    ...
D   WDANE 1       ;kod D1 i D2

E   QBLOK *       ;obszar roboczy
Ex  DS   0        ;wartość x
    ...
C3  BLOK *
    ...
F   QBLOK         ;postać param.
Fv1 DS   0k21
Fv2 DS   0
    ...
G   QBLOK 0c16000 ;adr.bloku D1
Gp1 DS   0
Gp2 DS   0        ;adres parametru
    ...
    KONIEC
```

W powyższym przykładzie bloki C1, C2 i C3 zostały zdefiniowane w programie, zaś D1 i D2 wkopiowane instrukcją WDANE. Blok pozorny E został zakotwiczony na stałe na adresie tuż za blokiem D2. Blok G został nałożony na blok D1 o z góry znanym adresie, by opisać symbolicznie jego postać. Blok F opisuje jakiś obszar pamięci – jaki, decyduje stosowane doń indeksowanie.

Blok, zarówno programu jak i pozorny, kończy:

- instrukcja BLOK rozpoczynająca nowy blok programu
- instrukcja QBLOK rozpoczynająca nowy blok pozorny
- instrukcja WDANE wstawiająca bloki danych z taśmy w kodzie PIĄTKOWYM
- instrukcja WDANEX wstawiająca bloki danych z taśmy w kodzie THETA
- instrukcja KONIEC zawarta w głównym tekście źródłowym

Jeśli brak instrukcji BLOK lub QBLOK, to kod generuje się jak dla BLOK 0:0, lub bezpośrednio za ostatnim blokiem wkopiowanym instrukcją WDANE lub WDANEX.

Instrukcje BLOK i QBLOK wymagają, by ewentualne przeadresowywanie rozpoczęte instrukcją NAF było zakończone instrukcją KOF.

Instrukcja BLOK – definicja bloku programu

```
etyk      BLOK    adres           ;początek bloku programu
```

Instrukcja BLOK definiuje początek bloku programu.

Argument **adres** nie jest wymagany. Jeśli jest pominięty, to domyślnie zostanie przyjęta wartość licznika rozkazów pozostała po ostatnim BLOKu – zdefiniowanym instrukcją BLOK lub wczytanym instrukcją WDANE lub WDANEX. Obecność pozornych bloków (QBLOKów) nie wpływa na domyślną wartość argumentu. BLOK bez argumentu znajdzie się w pamięci bezpośrednio za poprzednim BLOKiem. W przypadku, gdy nie było wcześniejszych BLOKów, domyślną wartością jest adres 0:0.

Jeśli jest argument **adres**, to musi być typu Adr lub Num. Może to być dowolne wyrażenie adresowe obliczalne w pierwszym przebiegu asemblera, nie zawierające swojej etykiety **etyk** – wyrażenie to może, ale nie musi być ujęte w nawiasy. Licznik lokacji * w wyrażeniu ma wartość pozostawioną po poprzedniej instrukcji. Argument typu Num jest automatycznie konwertowany do typu Adr jak w wyrażeniu Num:Num. Po obliczeniu, wartość argumentu **adres** staje się nową wartością licznika lokacji *.

Etykietę **etyk** można pominąć – jeśli jest, to jej wartością staje się nowa wartość licznika lokacji *.

Argument **adres** wyznacza adres początkowy bloku w pamięci operacyjnej. Pod ten adres blok będzie ładowany podczas wczytywania programu. Jest to zarazem adres umieszczany w pilocie bloku na taśmie wynikowej z kodem PIĄTKOWYM programu. Każdej instrukcji BLOK odpowiada pilot bloku. Wyjątkiem są bloki puste, nie zawierające ani jednego słowa kodu wynikowego – dla nich pilot nie jest tworzony.

Asembler nie kontroluje wartości argumentów BLOKów i nie wychwytuje ewentualnego nakładania się bloków na siebie, pozostawiając to zadanie trosce programisty.

Instrukcja QBLOK – definicja quasi-bloku (bloku pozornego)

```
etyk    QBLOK   adresX          ;początek bloku pozornego
```

Instrukcja QBLOK definiuje początek bloku pozornego – pewnego obszaru pamięci.

Argument **adresX** nie jest wymagany. Jeśli jest pominięty, to zostanie przyjęta domyślna wartość 0.

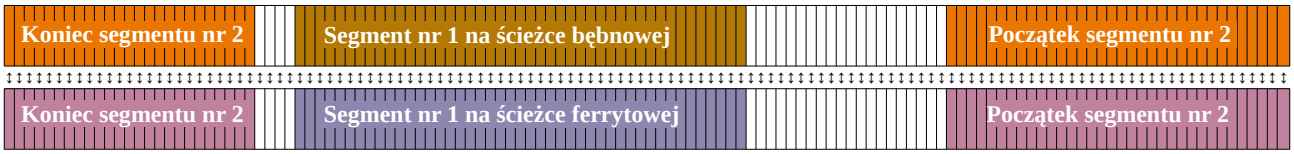
Jeśli jest argument **adresX**, to może być dowolnego typu: Num, NumX, Adr lub AdrX. Może to być dowolne wyrażenie adresowe obliczalne w pierwszym przebiegu asemblera, nie zawierające swojej etykiety **etyk** – wyrażenie to może, ale nie musi być ujęte w nawiasy. Licznik lokacji * w wyrażeniu ma wartość pozostawioną po poprzedniej instrukcji. Po obliczeniu wartość argumentu **adresX** staje się nową wartością licznika lokacji *.

Etykietę **etyk** można pominąć – jeśli jest, to jej wartością staje się nowa wartość licznika lokacji *.

Nakładanie się bloków pozornych na siebie i na bloki programu jest naturalne i stanowi metodę odwzorowywania obszarów pamięci na wiele różnych sposobów.

Instrukcje przeadresowywania kodu

ODRA 1013 jest wyposażona w szybką pamięć ferrytową, w której rozkazy i dane dostępne są praktycznie natychmiast, bez oczekiwania na obrót bębna. Komputer ODRA 1013 ma dodatkowe rozkazy przesyłania treści ścieżek bębnowych na ferrytowe i ferrytowych na bębnowe, pod odpowiadające sobie strefowo adresy. Każdy z rozkazów BF i FB umożliwia szybkie, w trakcie jednego obrotu bębna, przesłanie ciągłego segmentu komórek jednej ścieżki począwszy od wskazanego adresu, przy czym taki segment może obejmować koniec ścieżki i ciągnąć się dalej od jej początku, aż do przesłania wskazanej liczby, maksymalnie 128, komórek.



Pamięć ferrytowa to zaledwie dwie ścieżki, 2 razy po 128 komórek. Umieszczenie programu i danych w pamięci ferrytowej daje maksymalną prędkość pracy – rzecz w tym, że mało jest programów, które mieszczą się w niej w całości. Rozwiązaniem może być umieszczanie na ścieżkach ferrytowych tych segmentów programu, które mają być aktualnie wykonywane.

Problemem przy pisaniu programów jest zapewnienie prawidłowego adresowania komórek, które wczytywane do pamięci bębnowej, są następnie przenoszone na ferryt i tam wykonywane. Asembler AsmC wspomaga tę technikę, poprzez przeadresowywanie wskazanych instrukcjami NAF-KOF segmentów programu. Taki segment jest umieszczany w kodzie programu tak, jak każdy inny – w sposób ciągły wśród innych rozkazów, ale adresy słów tego segmentu są zamieniane na adresy docelowe, pod którymi znajdą się w czasie wykonywania.

Ponieważ segmenty programu mogą być wielokrotnie przerzucane między pamięcią bębnową, a ferrytową, i mogą znajdować się pod różnymi adresami, to AsmC oferuje operator przeadresowania `:` umożliwiający zastosowanie adresu doraźnie wymaganego w danej instrukcji.

Instrukcja NAF – początek przeadresowywania kodu

```
id      NAF     ścieżkaF        ;nowy adres ścieżkowy
^^^^^^^
```

Instrukcja NAF wskazuje początek przeadresowywanego segmentu programu. Koniec segmentu wskazuje instrukcja KOF.

Argument **ścieżkaF** jest wymagany i musi być typu: Num lub Adr. Może to być dowolne wyrażenie adresowe obliczalne w pierwszym przebiegu asemblera. Licznik lokacji * w wyrażeniu ma wartość pozostawioną po poprzedniej instrukcji. Po obliczeniu, adres składający się z 6 najstarszych bitów 13-bitowego wyrażenia **ścieżkaF** (nr ścieżki) i 7 najmłodszych bitów licznika lokacji ***** (nr strefy) staje się nową wartością licznika lokacji * (równoważną wyrażeniu **ścieżkaF:***).

Identyfikator **id** nie jest wymagany – jeśli jest, to instrukcja KOF kończąca przeadresowywanie może mieć ten sam identyfikator w celu łatwiejszego kojarzenia z instrukcją NAF.

Instrukcji NAF-KOF nie można zagnieżdżać – instrukcja NAF wymaga, by wcześniejsze przeadresowywanie zostało zakończone instrukcją KOF. Również instrukcje BLOK i QBLOK wymagają zakończenia aktualnego przeadresowywania instrukcją KOF.

Przeadresowywanie rozpoczęte instrukcją NAF musi być zakończone wraz z końcem bieżącej ścieżki programu. Jeżeli segment przesyłany blokowo obejmuje koniec ścieżki i ciągnie się dalej od jej początku, to AsmC wymaga użycia dwóch par instrukcji NAF-KOF: jednej nakazującej przeadresowywanie początku ścieżki, i drugiej – końca.

Rozkazy przesyłania blokowego dotyczą przesyłania na i z dwóch ścieżek ferrytowych. Możliwe jest więc przy ich użyciu przesłanie jednej ścieżki bębnowej na inną ścieżkę bębnową. Podobnie da się przesłać jedną ścieżkę ferrytową na drugą. Instrukcje NAF-KOF, swoimi mnemonikami sugerują zmianę adresów bębnowych na ferrytowe, bo taka jest najczęstsza praktyka, ale instrukcje te nie wprowadzają ograniczeń co do przeadresowywania – pozwalają zmieniać adresację dowolnych segmentów programu.

Instrukcja KOF – koniec przeadresowywania kodu

```
id      KOF             ;powrót do zwykłego adresowania
^^^^^^^
```

Instrukcja KOF wskazuje koniec przeadresowywanego segmentu programu. Początek segmentu wskazuje instrukcja NAF.

Instrukcja KOF nie ma argumentów.

Identyfikator **id** nie jest wymagany – jeśli jest, to musi być taki sam jak w instrukcji NAF w celu łatwiejszego kojarzenia pary NAF-KOF.

Instrukcji NAF-KOF nie można zagnieżdżać – instrukcja KOF jest wymagana dla zakończenia przeadresowywania przed kolejną instrukcją NAF, przed instrukcjami BLOK i QBLOK, oraz wraz z końcem bieżącej ścieżki programu.

Przykłady przeadresowywania kodu

Przykład 1.

Jest to program kopiowania danych z czytnika 8-kanałowego na samopis MAW, który posłuży do omówienia elementarnych kwestii związanych z przepisywaniem się programu na ferryt:

```
/*
   Program wymaga maszyny ODRA 1013 UMCS, gdyż przepisuje się na ferryt.
   Program ten działa w nieskończonej pętli - brak oznaczenia końca danych.
*/
        BLOK 0                          ;5  początek programu na ścieżce bębnowej
        B5 = (ET4-ET1-1) ..0:ET4        ;6  liczba komórek segmentu - 1
ET1B    EQU  *                          ;7  to jeszcze adres bębnowy
      NAF    61*128                     ;8  następne rozkazy będą na ferrycie
ET1     WE   2+8                        ;9  pierwszy rozkaz na na ferrycie
        LL   31                         ;10 drugi    rozkaz na na ferrycie
        WY   5+8          ..ET1         ;11 trzeci   rozkaz na na ferrycie
ET4     EQU  *                          ;12 to jeszcze adres ferrytowy
      KOF                               ;13 następne rozkazy będą na bębnie
ET4B    BF   61,B5,0:ET1 ..ET1          ;14 przesłanie segmentu na ferryt
        KONIEC
```

W wierszu 5 zdefiniowano blok programu zaczynający się od początku ścieżki nr 0 – bębnowej.

W wierszu 6 długość segmentu wstawiana do rejestru B5 jest obliczana na podstawie ferrytowych adresów ET1 i ET4 początku i końca segmentu programu, który ma być przesłany blokowo na ścieżkę ferrytową. Liczbę tę można by też obliczać na podstawie adresów bębnowych ET1B i ET4B. W tym samym wierszu adres następnego rozkazu do wykonania podano w postaci przeadresowania 0:ET4 adresu ferrytowego na bębnowy – można było podać zwyczajnie ET4B. Etykiety ET1B i ET4B w zaprezentowanej wersji programu są zbędne.

Rozkaz znajdujący się w wierszu 14 wykonuje się jako drugi. Przesyła treść segmentu ze ścieżki bębnowej, wskazując adres bębnowy początku segmentu w postaci przeadresowania 0:ET1 adresu ferrytowego. Po jego wykonaniu można już przekazać sterowanie do rozkazu ET1 na ferrycie – wskazanie następnego rozkazu do wykonania jest wskazaniem adresu ferrytowego ET1.

Rozkazy w wierszach 9, 10 i 11 znajdują się teraz również na ferrycie i wykonują się w pętli. Ostatni z nich nawraca do początku pętli, pod adres ferrytowy ET1. A co, gdyby nawrócił pod adres bębnowy 0:ET1, czyli pod adres 00001? Należy uważać! W tym konkretnym przykładzie, wykonałby się rozkaz bębnowy (identyczny z kopią ferrytową) i przekazałby sterowanie do następnego rozkazu na ferrycie, bo jego część adresowa K (NR) zawiera wygenerowany przez asembler adres ferrytowy, jak widać na wydruku:

```
     5.  00000          BLOK   *00000
     6.  00000            :032  00002    *00004    50
     7.  00001 ET1B     EQU    *00001
     8. =17201          NAF    *17201
     9. =17201 ET1        :266  00000    *17202    00
    10. =17202            :016  00037    *17203    00
    11. =17203            :566  00000    *17201    00
    12. =17204 ET4     EQU    *17204
    13.  00004         KOF
    14.  00004 ET4B      :136 *00001    *17201    50
    15.  00005         KONIEC  00000
```

Przykład 2.

Jest to schemat programu samoprzepisującego się na ścieżkę ferrytową, zamieszczony w [2]:

```
/*
   Program na podstawie przykładu zamieszczonego
   w paragrafie 9.2. skryptu [2]
*/
PROGR   BLOK 40*128                  ;5  początek programu na początku ścieżki
        B5 = (SEG2B-SEG1B-1)         ;6  liczba komórek segmentu - 1
        BF   61,B5,SEG1B ..SEG1F     ;7  przesłanie segmentu na ferryt

SEG1B   EQU  *                       ;9  początek przesyłanego segmentu 0c12002
NAF1  NAF    61*128                  ;10 segment będzie na ścieżce 0c17200
SEG1F   rozkaz3                      ;11 początek segmentu - adres ferrytowy
        rozkaz4
        ...
        B5 = (128-1)                     ;14 ostatnie dwa rozkazy przesyłają
        BF   61,B5,SEG2B..(61*128):SEG2B ;15 następną ścieżkę na ten sam ferryt
;       koniec ścieżki
NAF1  KOF

SEG2B   EQU  *
NAF2  NAF    61*128
        rozkazA
        rozkazB
        ...
;       koniec ścieżki
NAF2  KOF
        ...
```

W wierszu 5 zdefiniowano blok programu zaczynający się od początku ścieżki nr 40.

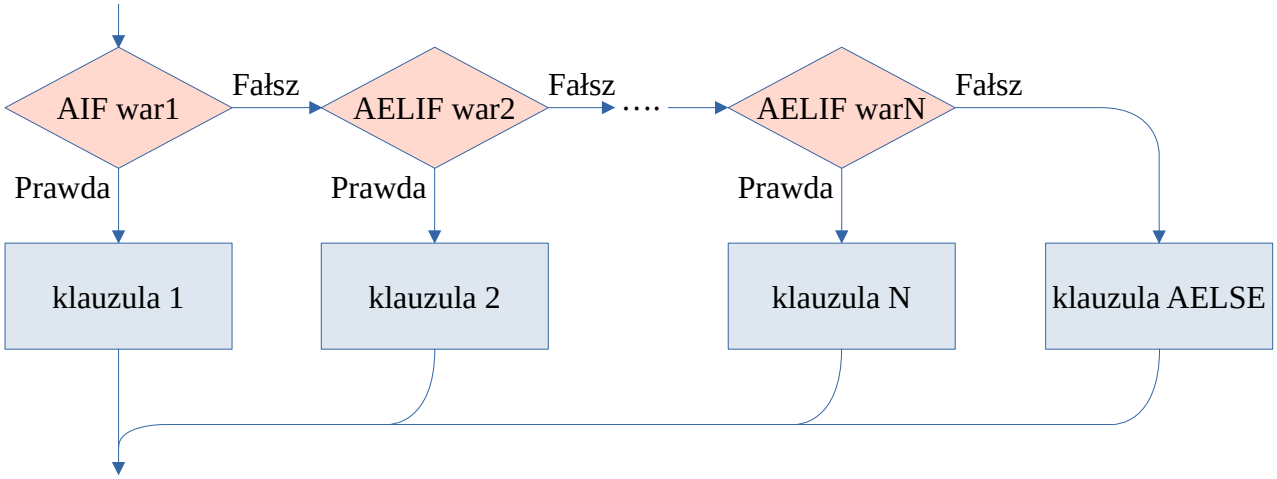
W wierszu 6, pierwszy rozkaz ścieżki ładuje do rejestru B5 pomniejszoną o 1 długość segmentu, jaki zostanie przesłany ze ścieżki bębnowej od adresu SEG1B. Adres SEG1B jest adresem bębnowym pierwszego rozkazu segmentu do przesłania.

Rozkaz w wierszu 7 przesyła segment o długości o 1 większej od liczby w rejestrze B5, spod adresu bębnowego SEG1B na ścieżkę ferrytową nr 61, i wyznacza rozkaz spod adresu ferrytowego SEG1F jako następny do wykonania. Zauważmy, że symbole SEG1B i SEG1F są adresami tego samego rozkazu „rozkaz3”, ale jeden bębnowego, a drugi ferrytowego.

W wierszach 14 i 15 znajdują się dwa ostatnie rozkazy na ścieżce ferrytowej, które przepisują następną ścieżkę bębnową również na ścieżkę ferrytową nr 61. Są analogiczne do poprzednich, ale tym razem (dla urozmaicenia przykładu) adres następnego rozkazu do wykonania został podany w postaci przeadresowania adresu bębnowego SEG2B na ferrytowy za pomocą operatora przeadresowania **:** (oszczędność na jednym symbolu).

UWAGA: Poszczególne segmenty nie muszą dokładnie wypełniać ścieżki, ale wtedy należy je odpowiednio rozmieścić na ścieżkach, gdyż rozkazy przesyłania blokowego kopiują komórki pod adresy odpowiadające sobie strefowo. Do kontroli, czy granice przeadresowywanego segmentu zgodne są z granicami ścieżki, można posłużyć się instrukcjami warunkowej asemblacji AIF-AELIF-AELSE-AFI.

Instrukcje warunkowej asemblacji




```
id      AIF war1                    ;warunek asemblacji 1. klauzuli
          Instrukcje klauzuli 1
id      AELIF war2                  ;warunek asemblacji 2. klauzuli
          Instrukcje klauzuli 2
        ...
id      AELIF warN                  ;warunek asemblacji N. klauzuli
          Instrukcje klauzuli N
id      AELSE                       ;początek klauzuli AELSE
          Instrukcje klauzuli ELSE
id      AFI                         ;koniec instrukcji warunkowej
```

Instrukcje asemblera warunkowej asemblacji wskazują poprzez warunki, które części tekstu źródłowego mają być asemblowane, a które pomijane. Jednym z zastosowań jest asemblowanie wariantu programu dostosowanego do szczególnych warunków, np. do modelu komputera. Innym – wykrywanie i sygnalizowanie sytuacji szczególnych przy pomocy instrukcji BYK i NOTA.

Instrukcję warunkową AIF-AELIF-AELSE-AFI otwiera instrukcja AIF, a zamyka instrukcja AFI. Pomiędzy nimi mogą znajdować się instrukcje AELIF oraz instrukcja AELSE, rozdzielające poszczególne klauzule, ale nie są wymagane. Każda klauzula może być pusta.

Asemblowane są instrukcje pierwszej klauzuli, której warunek **war1**, **war2**, …, **warN** jest prawdziwy – pozostałe są pomijane i ich warunki nie są obliczane. Jeśli żaden warunek nie jest prawdziwy, to asemblowana jest klauzula AELSE. Każdy warunek jest wyrażeniem adresowym, obliczalnym w pierwszym przebiegu asemblera. Wartość wyrażenia równa 0 jest fałszywa – każda inna, oraz wszystkie adresy i wartości indeksowane, w tym (0:0), (0:0+Bb), (0+Bb), są prawdziwe.

Instrukcje AIF-AELIF-AELSE-AFI można zagnieżdżać – w każdej klauzuli mogą znajdować się kolejne instrukcje AIF-AELIF-AELSE-AFI.

Instrukcja AIF – początkowa klauzula instrukcji warunkowej

```
id      AIF     war1            ;warunek asemblacji 1. klauzuli
```

Instrukcja AIF wskazuje początek instrukcji warunkowej asemblacji. Jest to także początek pierwszej klauzuli, która ma być asemblowana jeśli warunek **war1** będzie prawdziwy – i wtedy pozostałe klauzule zostaną pominięte. Klauzulę tę kończy instrukcja AELIF, AELSE lub AFI.

Argument **war1** jest wymagany i może być dowolnego typu: Num, NumX, Adr lub AdrX. Może to być dowolne wyrażenie adresowe obliczalne w pierwszym przebiegu asemblera. Licznik lokacji * w wyrażeniu ma wartość pozostawioną po poprzedniej instrukcji.

Identyfikator **id** nie jest wymagany – jeśli jest, to pozostałe instrukcje AELIF, AELSE i AFI instrukcji złożonej AIF-AELIF-AELSE-AFI mogą wskazać ten sam identyfikator w celu ułatwienia kojarzenia ich w jeden komplet.

Identyfikator **id** instrukcji AIF jest jednocześnie etykietą o wartości licznika rozkazów w tym punkcie programu.

Instrukcja AELIF – następna klauzula instrukcji warunkowej

```
id      AELIF   war2            ;warunek kolejnej klauzuli
```

Instrukcja AELIF wskazuje początek kolejnej klauzuli, która ma być asemblowana jeśli warunek **war2** będzie prawdziwy – a wtedy pozostałe klauzule zostaną pominięte. Klauzula ta jest pomijana i warunek **war2** nie jest obliczany, jeśli którakolwiek wcześniejsza klauzula instrukcji złożonej AIF-AELIF-AELSE-AFI była asemblowana. Koniec tej klauzuli wyznacza instrukcja AELIF, AELSE lub AFI.

Argument **war2** jest wymagany i może być dowolnego typu: Num, NumX, Adr lub AdrX. Może to być dowolne wyrażenie adresowe obliczalne w pierwszym przebiegu asemblera. Licznik lokacji * w wyrażeniu ma wartość równą wartości w punkcie instrukcji AIF.

Identyfikator **id** nie jest wymagany – jeśli jest, to musi być zgodny z identyfikatorem instrukcji AIF.

Instrukcja AELSE – ostatnia klauzula instrukcji warunkowej

```
id      AELSE           ;początek klauzuli AELSE
```

Instrukcja AELSE wskazuje początek klauzuli, która ma być asemblowana jeśli żaden z warunków poprzednich klauzul nie jest prawdziwy. Klauzula ta jest pomijana, jeśli któraś wcześniejsza klauzula instrukcji złożonej AIF-AELIF-AELSE-AFI była asemblowana. Koniec tej klauzuli wyznacza instrukcja AFI.

Instrukcja AELSE nie ma argumentów. Licznik lokacji * w tym punkcie programu ma wartość równą wartości w punkcie instrukcji AIF.

Identyfikator **id** nie jest wymagany – jeśli jest, to musi być zgodny z identyfikatorem instrukcji AIF.

Instrukcja AFI – koniec instrukcji warunkowej

```
id       AFI                    ;koniec instrukcji warunkowej
␣␣␣␣␣␣␣
```

Instrukcja AFI wskazuje koniec instrukcji złożonej AIF-AELIF-AELSE-AFI.

Instrukcja AFI nie ma argumentów. Licznik lokacji * w tym punkcie programu ma wartość pozostawioną po ostatnio zasemblowanej instrukcji (tj. albo po zasemblowanej klauzuli, albo w punkcie instrukcji AIF jeśli żadna klauzula nie została zasemblowana).

Identyfikator **id** nie jest wymagany – jeśli jest, to musi być zgodny z identyfikatorem instrukcji AIF.

Instrukcje sygnalizacji błędów i uwag jako samokontroli

Asembler AsmC zawiera dwie instrukcje emitowania komunikatów w trakcie translacji programu.

Jedna z nich, BYK, służy do przerywania procesu translacji programu z uwagi na niedopuszczalną sytuację. Np. pusta tablica gdy program takiej nie dopuszcza, albo wkroczenie rozkazów programu w obszar zarezerwowany na dane wprowadzane z zewnątrz instrukcjami WDANE lub WDANEX.

Druga z nich, NOTA, służy do zwracania uwagi na szczególne sytuacje w programie, które jednak nie są błędami i nie powodują zatrzymania asemblacji. Np. że są jeszcze wolne komórki pamięci ferrytowej, których użycie mogłoby przyczynić się do przyspieszenia pracy programu.

Instrukcje te zwykle umieszcza się w klauzulach instrukcji warunkowej AIF-AELIF-AELSE-AFI wykrywającej sytuacje szczególne.

Instrukcja BYK – sygnalizacja błędu

```
etyk     BYK    nr  ,  'Opis błędu'                ;komentarz
␣␣␣␣␣␣␣
```

Instrukcja BYK służy do sygnalizacji błędu wykrytego przez sam asemblowany program, powodującego przerwanie asemblacji po pierwszym przebiegu translatora. Instrukcja powoduje wydrukowanie w listingu programu z pierwszego przebiegu komunikatu błędu:

```
BYK47 llllll.kkkkkk-llllll.kkkkkk: AUTORESTRYKCJA
```

gdzie `llllll.kkkkkk-llllll.kkkkkk` wskazują numery wiersz.kolumna od-do w tekście źródłowym. Ponieważ drugi przebieg asemblera nie zostanie wykonany, to należy odnaleźć wskazane miejsce w programie źródłowym, w celu odczytania numeru i opisu błędu.

Do emitowania przez asemblowany program uwag w czasie asemblacji, nie traktowanych jako sygnał błędu, służy opisana niżej bliźniacza instrukcja NOTA.

Argument **nr** jest wymagany i musi być wyrażeniem typu Num, obliczalnym w pierwszym przebiegu asemblera. Służy on do ewidencjonowania błędów wykrywanych przez asemblowany

program podczas asemblacji – taki wykaz można zamieścić w komentarzu w programie, lub w dokumentacji programu lub całej programoteki. Wykaz błędów może być wspólny z wykazem adnotacji emitowanych instrukcjami NOTA, lub prowadzony niezależnie od nich. Numery błędów najlepiej odnotowywać ósemkowo dla łatwiejszej identyfikacji.

Argument **'Opis błędu'** jest wymagany w postaci tekstu ujętego w apostrofy (tj. typu T0, ale bez jawnego podawania atrybutu Tt typu tekstu, oraz bez jawnego wskazania atrybutu Ss liczby słów tekstu), wyjaśniającego istotę błędu. Tekst ten musi być zgodny z tekstami w stałych tekstowych typu T0'…', dopuszczalnych w instrukcji DS, i może być wielowierszowy. Ociosywanie końca tekstu oraz kropka likwidująca słowo zerowe zaznaczające koniec tekstu nie mają znaczenia. Dłuższe opisy, przekraczające maksymalną długość tekstu, należy zamieszczać w ewidencji błędów, a jeśli to ograniczenie będzie dokuczliwe, to w otoczeniu instrukcji BYK można zamieścić dodatkowe komentarze.

Pomiędzy argumentami wymagany jest przecinek, który może być otoczony dowolnie wielkimi odstępami.

Etykieta **etyk** może być pominięta – jeśli jest, to jej wartością jest wartość licznika lokacji * w tym punkcie programu. Licznik lokacji * pozostaje niezmieniony.

Instrukcja NOTA – wstawianie adnotacji

```
etyk    NOTA    nr  ,  'Tekst adnotacji'                ;komentarz
```

Instrukcja NOTA służy do wstawiania do wydruku kodu wynikowego uwag dotyczących programu, emitowanych przez sam asemblowany program. Instrukcja powoduje wydrukowanie komunikatów:

w listingu tekstu programu w pierwszym przebiegu:

```
HAK00 llllll.kkkkkk-llllll.kkkkkk: ADNOTACJA: NOTA
```

w listingu kodu w drugim przebiegu:

```
llllll.  aaaaa etyk     NOTA    nnnnn
Tekst adnotacji
```

gdzie:

`llllll.kkkkkk-llllll.kkkkkk` – nr wiersz.kolumna, od-do w tekście źródłowym
`aaaaa` – adres w kodzie wynikowym
`nnnnn` – ósemkowy numer adnotacji

Do sygnalizacji błędów wykrywanych przez sam asemblowany program w czasie asemblacji, powodujących przerwanie asemblacji po pierwszym przebiegu translatora, służy, opisana wyżej, bliźniacza instrukcja BYK.

Argumenty instrukcji NOTA zadaje się identycznie jak w instrukcji BYK i mają one analogiczne znaczenie.

Instrukcja WSTAW – wstawianie dodatkowego tekstu źródłowego

```
id      WSTAW   numer           ;komentarz na temat tekstu
```

Instrukcja WSTAW służy do wstawiania dodatkowego tekstu do głównego tekstu źródłowego. Wstawiany tekst będzie wczytywany z czytnika 5-kanałowego nr 2 w miejsce instrukcji WSTAW. Instrukcja w pierwszym i w drugim przebiegu asemblera emituje komunikat:

```
ZAŁÓŻ: numer NA PTR2  I WYSTARTUJ CPU
```

gdzie `numer` to wartość argumentu instrukcji drukowana ósemkowo

Numer **numer** w zamierzeniu jest identyfikatorem tekstu. W przypadku asemblacji programu składającego się z wielu tekstów źródłowych, zaleca się zebranie ich w komplet i ponumerowanie. Można też ponumerować unikalnymi numerami wszystkie programy w bibliotece programów. Dobrze jest spisać wszystkie wymagane teksty wraz z ich ósemkowymi numerami w dokumentacji lub w komentarzu programu.

Zamontowanie taśmy w czytniku należy potwierdzić naciśnięciem przycisku „Start CPU” na pulpicie maszyny. Rozkazy czytania tekstu mają na części N (AR) ten właśnie numer, obojętny dla samego rozkazu, ale podczas czekania na urządzenie wyświetlany na lampkach rejestru rozkazów – na podstawie tych lampek lub drukowanych komunikatów operator może łatwiej orientować się co do potrzebnych taśm.

Wstawiany tekst powinien być przygotowany wg dokładnie takich samych zasad, jak tekst główny. Koniec jego treści powinien być wskazany instrukcją KONIEC. Dla zaznaczenia, że nie jest to koniec całego programu, asembler w listingu kodu wynikowego drukuje ją jako pseudoinstrukcję OSTAW z numerem **numer** i identyfikatorem **id** zgodnymi z instrukcją WSTAW:

```
    11.  00000      ...
    12.  00001 id       WSTAW   numer
ZAŁÓŻ  numer NA PTR2  I WYSTARTUJ CPU
    13+  00001      ...
    14+  00002      ...
    15+  00003 id       OSTAW   numer
    16.  00003      ...
```

Argument **numer** jest wymagany i musi być wyrażeniem typu Num, obliczalnym w pierwszym przebiegu asemblera. Jeśli jest to liczba, to najlepiej w postaci denotacji ósemkowej. Licznik lokacji * w wyrażeniu ma wartość pozostawioną po poprzedniej instrukcji.

Identyfikator **id** nie jest wymagany – jeśli jest, to służy kojarzeniu instrukcji KONIEC, drukowanej w drugim przebiegu jako pseudoinstrukcja OSTAW, z instrukcją WSTAW.

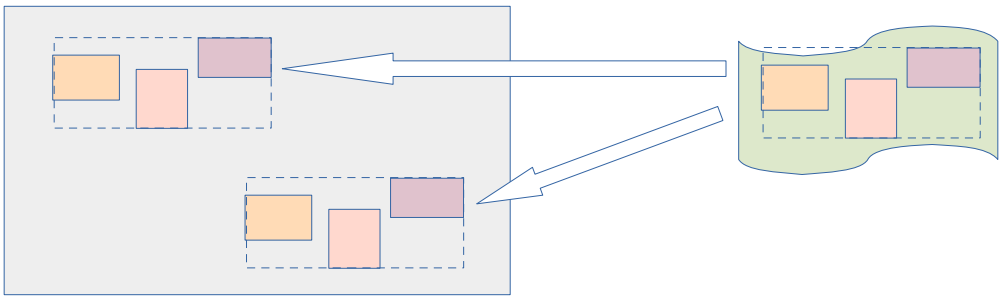
Identyfikator **id** instrukcji WSTAW jest jednocześnie etykietą o wartości licznika rozkazów w tym punkcie programu.

UWAGA: Instrukcja WSTAW we wstawianym tekście jest niedozwolona – brak tak wielu czytników taśmy perforowanej 5-kanałowej.

Instrukcje włączania do programu gotowego kodu binarnego

Biblioteka programów – programoteka – zawiera zwykle programy w postaci nie tylko źródłowej, ale również w postaci gotowego kodu binarnego, ujęte często w pakiety funkcji i podprogramów. Często też wyniki z jednego programu (dane) przekazywane są do drugiego w postaci binarnej. Programy raz przygotowane nie wymagają ponownej asemblacji i są gotowe w postaci taśm w kodzie PIĄTKOWYM lub THETA. Format tych taśm jest opisany w [5].

AsmC umożliwia wcielanie danych binarnych do kodu wynikowego asemblowanego programu, przy czym możliwe jest umieszczenie ich w pamięci z pewnym przesunięciem w stosunku do adresów zapisanych w pilotach, ale z zachowaniem wzajemnego położenia:



Kodu programu wykonywalnego, ze względu na zawarte w rozkazach absolutne adresy, zasadniczo nie należy przemieszczać, a program główny musi z góry wiedzieć na podstawie dokumentacji, które obszary pamięci operacyjnej zostaną zajęte, by wczytywane bloki, oraz bloki definiowane w programie, nie nakładały się na siebie. Można jednak żądać, by binarne dane, np. tablice wyników z innego programu, zostały wcielone w miejsce pamięci dogodniejsze dla naszego programu.

Instrukcja WDANE – włączanie danych z taśmy w kodzie PIĄTKOWYM

```
id      WDANE   numer   ,   adres     ;komentarz na temat danych
```

Instrukcja WDANE służy do wkopiowania danych zapisanych na taśmie w kodzie PIĄTKOWYM do kodu wynikowego programu. Kod będzie wczytywany z czytnika 5-kanałowego nr 2 w postaci bloków zapisanych na taśmie. Jeśli argument **adres** zostanie pominięty, to bloki zostaną wczytane pod adresy zapisane w ich pilotach. Jeśli argument **adres** zostanie podany, to pierwszy blok na taśmie zostanie wczytany pod ten adres, a wyliczone przesunięcie tego bloku zostanie zastosowane do pozostałych wczytywanych bloków – zachowane zostanie wzajemne położenie wczytywanych bloków względem siebie. Instrukcja w pierwszym i w drugim przebiegu asemblera emituje komunikat:

```
ZAŁÓŻ: numer NA PTR2  I WYSTARTUJ CPU
```

gdzie `numer` to wartość argumentu **numer** instrukcji drukowana ósemkowo

Numer **numer** w intencji jest identyfikatorem danych PIĄTKOWYCH. W przypadku programu wkopiowującego wiele danych PIĄTKOWYCH, zaleca się skompletowanie ich w jednym miejscu i

ponumerowanie. Można też ponumerować unikalnymi numerami wszystkie dane w archiwum danych. Dobrze jest spisać wszystkie wymagane dane wraz z ich ósemkowymi numerami w dokumentacji lub w komentarzu programu.

Zamontowanie taśmy w czytniku należy potwierdzić naciśnięciem przycisku „Start CPU” na pulpicie maszyny. Rozkazy czytania danych mają na części N (AR) ten właśnie numer, obojętny dla samego rozkazu, ale podczas czekania na urządzenie wyświetlany na lampkach rejestru rozkazów – na podstawie tych lampek lub drukowanych komunikatów operator może łatwiej orientować się co do potrzebnych taśm.

Wstawiane dane w listingu kodu wynikowego asembler drukuje jako BLOKi z argumentami równymi adresom, pod które bloki te są wczytywane. Bloki są wczytywane aż do napotkania sumy kontrolnej i wtedy drukowana jest instrukcja UDANE z numerem **numer** i identyfikatorem **id** zgodnymi z instrukcją WDANE. W przypadku niepoprawnej sumy kontrolnej, zamiast UDANE drukowana jest instrukcja NDANE. Jeżeli na taśmie PIĄTKOWEJ nie ma sumy kontrolnej, to zamiast UDANE drukowana jest instrukcja ZDANE:

```
     2.  00764            ...
     3.  00765  id      WDANE   numer
ZAŁÓŻ  numer NA PTR2  I WYSTARTUJ CPU
     3:  00000          BLOK   *00000
     3:  00000            :050  02000     00001    00
     3:  00001            :015  02001     00002    00
     3:  00002            ...
     3:  02000          BLOK   *02000
     3:  02000            :150  00000     00000    00
     3:  02001            ...
     3:  02002  id      UDANE   numer
     4.  02002            ...
```

Po instrukcji WDANE licznik rozkazów zawiera adres bezpośrednio za ostatnio wczytanym blokiem.

Argument **numer** jest wymagany i musi być wyrażeniem typu Num, obliczalnym w pierwszym przebiegu asemblera. Jeśli jest to liczba, to najlepiej w postaci denotacji ósemkowej. Licznik lokacji * w wyrażeniu ma wartość pozostawioną po poprzedniej instrukcji.

Argument **adres** jest opcjonalny. Jeśli jest podany, to musi być wyrażeniem typu Adr, obliczalnym w pierwszym przebiegu asemblera. Licznik lokacji * w wyrażeniu ma wartość pozostawioną po poprzedniej instrukcji.

Identyfikator **id** nie jest wymagany – jeśli jest, to służy kojarzeniu drukowanej w drugim przebiegu instrukcji UDANE, NDANE lub ZDANE z instrukcją WDANE.

Do włączania danych w kodzie THETA służy, opisana dalej, bliźniacza instrukcja WDANEX.

UWAGA: Instrukcja WDANE we wstawianym tekście jest niedozwolona – brak tak wielu czytników taśmy perforowanej 5-kanałowej.

Instrukcja WDANEX – włączanie danych z taśmy w kodzie THETA

```
id      WDANEX  numer   ,  adres    ;komentarz na temat danych
```

Instrukcja WDANEX służy do wkopiowania danych zapisanych w kodzie THETA do kodu wynikowego programu. Kod będzie wczytywany z czytnika 8-kanałowego nr 2’ w postaci bloków zapisanych na taśmie, pod adresy zapisane w pilotach tych bloków, lub zgodnie z argumentem **adres**. Instrukcja w pierwszym i w drugim przebiegu asemblera emituje komunikat:

```
ZAŁÓŻ: numer NA PTR2' I WYSTARTUJ CPU
```

gdzie `numer` to wartość argumentu instrukcji drukowana ósemkowo

Argumenty instrukcji WDANEX zadaje się identycznie jak w instrukcji WDANE i mają one takie samo znaczenie. Numer **numer** w intencji jest identyfikatorem danych THETA.

Wstawiane dane w listingu kodu wynikowego asembler drukuje jako BLOKi z argumentami równymi adresom, pod które bloki te są wczytywane. Bloki są wczytywane aż do napotkania sumy kontrolnej i wtedy drukowana jest instrukcja UDANEX z numerem **numer** i identyfikatorem **id** zgodnymi z instrukcją WDANEX. W przypadku niepoprawnej sumy kontrolnej, zamiast UDANEX drukowana jest instrukcja NDANEX. Jeżeli na taśmie THETA nie ma sumy kontrolnej, to zamiast UDANEX drukowana jest instrukcja ZDANEX:

```
     4.  02002            ...
     5.  02003  id      WDANEX  numer
ZAŁÓŻ  numer NA PTR2' I WYSTARTUJ CPU
     5!  00000          BLOK   *00000
     5!  00000            :050  02000     00001    00
     5!  00001            :015  02001     00002    00
     5!  00002            ...
     5!  02000          BLOK   *02000
     5!  02000            :150  00000     00000    00
     5!  02001            ...
     5!  02002  id      UDANEX  numer
     6.  02002            ...
```

Po instrukcji WDANEX licznik rozkazów zawiera adres bezpośrednio za ostatnio wczytanym blokiem.

Identyfikator **id** nie jest wymagany – jeśli jest, to służy kojarzeniu instrukcji UDANEX, NDANEX lub ZDANEX z instrukcją WDANEX.

Do włączania danych w kodzie PIĄTKOWYM służy opisana wcześniej bliźniacza instrukcja WDANE.

UWAGA: Instrukcja WDANEX dozwolona jest w tekście głównym programu, jak i w tekście wstawianym, jest więc atrakcyjna na maszynie UMCS, wyposażonej w czytniki taśmy perforowanej 8-kanałowej.

Instrukcja KONIEC – koniec tekstu źródłowego

```
␣␣␣␣␣␣␣ KONIEC  start           ;koniec taśmy źródłowej
```

Instrukcja KONIEC oznacza koniec tekstu źródłowego na taśmie. Na taśmie z głównym tekstem oznacza koniec programu. Na taśmie z tekstem wstawianym do głównego instrukcją WSTAW, oznacza koniec wstawianego tekstu, dzięki czemu każdy tekst źródłowy może być asemblowany jako główny lub wstawiany (o ile nie ma innych trudności). Wiersz zawierający instrukcję KONIEC musi być zakończony przejściem do nowej linii.

Asembler instrukcję KONIEC we wstawianym tekście drukuje w drugim przebiegu jako pseudoinstrukcję OSTAW z argumentem **numer** i identyfikatorem **id** zgodnymi z instrukcją WSTAW. Z tego względu etykieta instrukcji KONIEC jest niedozwolona.

Argument **start** nie jest wymagany. Jeśli go brak, to przyjęta zostanie wartość domyślna 0. Jeśli jest, to powinien być typu Adr. Argument instrukcji we wstawianym tekście jest ignorowany. W głównym tekście służy wskazaniu czytelnikowi głównego adresu startu programu. Wobec braku odpowiedniego systemu operacyjnego komputera, informacja ta nie jest wykorzystywana w żaden automatyczny sposób.

UWAGA: Brak instrukcji KONIEC może spowodować nieskończone czytanie taśmy źródłowej. Może tak się stać również wskutek braku przejścia do nowego wiersza w instrukcji KONIEC, nierozpoznania instrukcji KONIEC, niezamkniętego łańcucha tekstowego, niezamkniętego komentarza blokowego, niezamkniętej instrukcji warunkowej AIF-AELIF-AELSE-AFI, lub innego błędu we wcześniejszej instrukcji.

Instrukcja DS definicji słów danych

Instrukcja DS służy do definiowania stałych wielkości w programie, zajmujących słowa pamięci. Dla wyraźnego odróżnienia od liczb wchodzących w skład wyrażeń adresowych (nie zajmujących pamięci), będziemy nazywać je słowami danych, gdyż zajmują pamięć podobnie jak kod rozkazów. Słowa danych rezerwuje się, definiuje i inicjuje poprzez podanie ich wartości. Denotacja wartości określa jej typ, oraz wielkość rezerwowanej pamięci. Odnosi się to również do instrukcji DS w blokach pozornych, w których jednakże nie następuje inicjalizacja pamięci.

Ogólny format instrukcji jest następujący:

```
etyk    DS    defStałej1  ,  defStałej2  ,  ...    ;komentarz
```

gdzie **defStałej1, defStałej2,** ... są definicjami stałych wypełniających kolejne słowa pamięci

Poszczególne definicje stałych mogą być rozdzielone przecinkami lub odstępami. Każda definicja stałej ma następujący format ogólny:

```
[krotność]  stała
```

gdzie:

[krotność] jest opcjonalnym wyrażeniem adresowym typu Num, obliczalnym w pierwszym przebiegu translatora, ujętym w nawiasy kwadratowe, wskazującym liczbę wystąpień definiowanej stałej. Może to być liczba od 0 do 8191. Domyślna krotność wynosi 1. Błąd w instrukcji DS ustawia krotność stałej na 1

stała jest to dana, która poprzez swoją notację określa typ i wartość wielkości umieszczanej w słowie lub słowach pamięci. Między krotnością a stałą może być odstęp, ale nie musi. Rozróżnia się następujące typy stałych:
– słowo liczbowe stałoprzecinkowe
– słowo liczbowe wieloadresowe
– słowo liczbowe zmiennoprzecinkowe
– słowa łańcucha tekstowego

Przykłady instrukcji DS

```
etyk1   DS    1, 2, 3               ;trzy słowa z liczbami 1, 2 i 3
etyk2   DS    [15] 0                ;15 wyzerowanych słów
etyk3   DS    3.14                  ;liczba zmiennoprzecinkowa
etyk4   DS    6.93147181e-1         ;=ln 2
etyk5   DS    (kon-pocz)k21         ;liczba wieloadresowa
etyk6   DS    (5)k8+(7)k21+(*)k34   ;liczba wieloadresowa
etyk7   DS    'Tytuł wydruku'       ;słowa zawierające tekst
etyk8   DS    S1T2'/R/N/F/L'.       ;tekst dłubany
etyk9   DS    0'12'  2'AB'          ;liczby tekstowe w skali k38
```

Słowo liczbowe stałoprzecinkowe

Słowa liczbowe stałoprzecinkowe są to słowa będące argumentami rozkazów wykonujących operacje stałoprzecinkowego dodawania, odejmowania, zmiany znaku, wartości bezwzględnej, bitowej koniunkcji, bitowej różnicy symetrycznej, mnożenia i dzielenia (w tym dzielenia długiego), zaokrąglenia normalnego, oraz przesunięć, a więc zawierającymi liczby stałoprzecinkowe.

Liczby stałoprzecinkowe są to liczby całkowite, dodatnie, ujemne lub bez znaku, w reprezentacji binarnej uzupełnień do 2. AsmC nie pozwala na notację liczb stałoprzecinkowych z częścią ułamkową – liczba z kropką jest zmiennoprzecinkowa. Liczby stałoprzecinkowe mogą być zapisane na wskazanych bitach słowa, tj. we wskazanej skali.

Na denotację liczby stałoprzecinkowej składają się:

```
+  0123456789              K12
-  0B01
   0C01234567
   0X0123456789ABCDEF
   0'tekst toporny'
   1'tekst ciosany'1
   2'tekst dłubany'
   6'tekst 6-bitowy'
   7'tekst 7-bitowy'
   9'tekst odrębny'
```

- opcjonalny znak: **`+`** lub **`-`** (brak znaku oznacza liczbę bez znaku – naturalną)
- wartość bezwzględna, podana:
  - dziesiętnie – przy pomocy cyfr dziesiątkowych **`0...9`**
  - binarnie – z prefiksem **`0B`** przy pomocy cyfr dwójkowych **`0...1`**
  - ósemkowo – z prefiksem **`0C`** przy pomocy cyfr ósemkowych **`0...7`**
  - szesnastkowo – z prefiksem **`0X`** przy pomocy cyfr szesnastkowych **`0...9,A...F`**
  - znakowo – z prefiksem **`0'`**, **`1'`**, **`2'`**, **`6'`**, **`7'`** lub **`9'`**
    (podobnie do zapisu tekstów – zobacz: Słowa łańcuchów tekstowych)
- opcjonalna skala liczby: **`K12`** (litera k lub K z liczbą od 0 do 38 wyznaczającą skalę – brak skali oznacza domyślną skalę k38)

Litery oznaczające podstawę systemu pozycyjnego: **0B**, **0C** i **0X**, oraz literowe cyfry szesnastkowe, mogą być wielkie lub małe.

Cyfry liczb dziesiętnych, binarnych, ósemkowych i szesnastkowych można dowolnie grupować przy pomocy znaku podkreślenia **`_`**. Denotacja liczby nie może zawierać odstępów.

Liczba znakowa musi zaczynać się od cyfrowego prefiksu wskazującego typ łańcucha tekstowego, bezpośrednio po którym musi być apostrof, i kończyć się zamykającym apostrofem. Należy pominąć literę T zapowiadającą atrybut typu, a pozostawić cyfrę. Poszczególne kody składające się na tekst są dosuwane do prawej strony (jak cyfry w liczbach), a nie do lewej (jak to jest w łańcuchach tekstowych). Kody znaków w tekście liczby znakowej traktowane są jak cyfry w systemie o podstawie: 32, 64 lub 128, dla tekstów odpowiednio: 5-, 6- lub 7-bitowych. Liczba

znakowa nie zawiera bitów, które w tekstach niosą informację o liczbie kodów w słowie. Liczba znakowa nie ma też wyzerowanego słowa oznaczającego koniec tekstu, zatem kropka oznaczająca pominięcie tego słowa nie ma sensu.

Szczególną uwagę należy zwrócić na liczby znakowe typu 0' i 1' (toporne i ciosane), w których zawarte są również kody niewidoczne na wydruku. Kody te mogą być inne w trybie wielkich liter, niż w trybie małych liter. Niewidoczne kody zmiany pocztu mogą występować wielokrotnie. Szczegóły można znaleźć w opisie kodowania tekstów.

Liczba po przesunięciu do wymaganej skali musi zmieścić się na 39 bitach słowa maszynowego. Najstarsze bity, nie mieszczące się w 39-bitowym słowie, nie powodują błędu jeśli są zerowe. Maksymalny zakres liczb w skali 38 jest następujący:

liczba ze znakiem: `-274_877_906_944` … `+274_877_906_943`

liczba bez znaku: `0` … `549_755_813_887`

Przykłady liczb stałoprzecinkowych	
Denotacja	*Objaśnienie*
`500_000_000_000`	Mieści się w słowie, bo jest bez znaku
`+500_000_000_000`	Nie mieści się w słowie, bo jest ze znakiem
`+0c1234567k21`	Znak + jako kontrola, czy się zmieści w słowie
`123456789012`	
`0b10101k8`	Liczba na części TPG słowa
`0x1_EF_06k31`	
`-0'ABCDE'k34`	Na początku i końcu są kody przełączające poczty
`-1'XABCDEFX'1k34`	`=(((A*32+B)*32+C)*32+D)*32+E)*32+F` tekst ociosany z obu stron, ale i tak pomiędzy literami mogą istnieć kody przełączające poczty znakowe

UWAGA: Liczby służące do obliczania wartości adresowej – umieszczanej w rozkazie jako argument bezpośredni lub adres argumentu lub rozkazu, lub też użytej jako argument instrukcji – zapisuje się podobnie, ale nie mają one znaku ani skali, oraz są 13-bitowe (muszą mieścić się w zakresie 0 … 8191).

Słowo liczbowe wieloadresowe

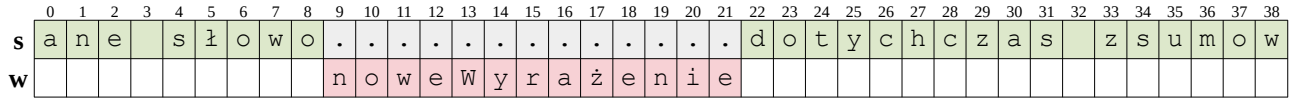
Słowo liczbowe wieloadresowe służy w sposób uogólniony do przechowywania adresów. Ponieważ adres komórki pamięci jest 13-bitowy, to w 39-bitowym słowie można zmieścić nawet trzy takie adresy. W praktyce programowania ODRY wygodnie jest, by adres zapamiętany w pamięci znajdował się w polu AR (N) lub NR (K) słowa, co oznacza, że trzeci adres musiałby znajdować się na bitach 35...38 i 0...8. AsmC umożliwia dowolne rozmieszczanie wartości adresowych w słowie za pomocą skali, podobnie jak dla liczb stałoprzecinkowych. Wartości adresowe mogą służyć do reprezentowania różnych wielkości, nie tylko adresów, i mogą zajmować mniej niż 13 bitów każda, zatem w jednym słowie może być ich wiele.

Słowo wieloadresowe wskazuje się jako sumę wartości adresowych:

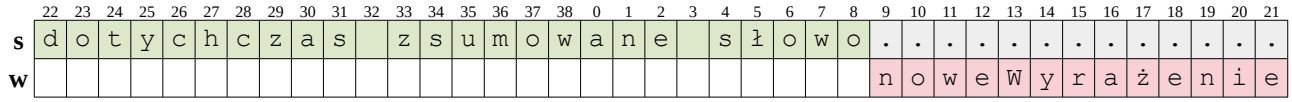
```
(wyrażenie1)K12 +(wyrażenie2)K12 +(wyrażenie3)K12 ...
```

Każdy składnik takiej sumy jest wyrażeniem adresowym w pewnej skali **K12** (litera k lub K z liczbą od 0 do 38 wyznaczającą skalę – brak skali oznacza domyślną skalę k38). Znaki **+** łączące składniki w jedno słowo wieloadresowe nie mogą być otoczone odstępami. Każde wyrażenie musi być ujęte w nawiasy okrągłe. Do wyrażeń odnoszą się ogólne zasady pisania wyrażeń adresowych – w szczególności elementy wyrażenia mogą być rozdzielane odstępami.

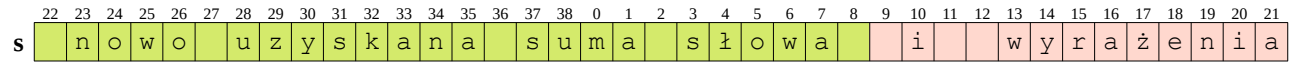
13-bitowa wartość wyrażenia może być typu Num lub Adr – wartość indeksowana jest niedozwolona. Poszczególne wyrażenia są sumowane w jedno słowo. Kolejne wyrażenie jest dodawane do słowa w sposób następujący:



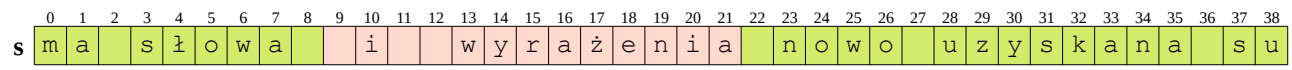
Suma **s** i składnik **w** są przesuwane cyklicznie w prawo o (38-k) bitów, gdzie k jest skalą **w**:



Przesunięte suma **s** i składnik **w** są dodawane, przy czym przeniesienie z sumowania (bowiem wykropkowane bity nie muszą być zerowe) przechodzi na starsze bity i znika dopiero po wyjściu poza 39-bitowy rejestr:



Nowo uzyskana suma **s** jest przesuwana cyklicznie w prawo o (k+1) bitów, tak, by wszystkie jej bity wróciły na właściwe pozycje:



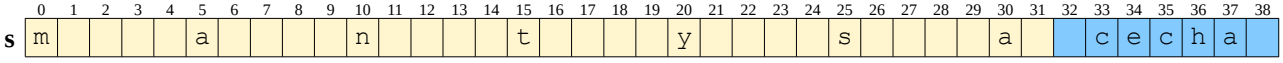
Podczas obliczania wyrażeń składających się na słowo wieloadresowe, licznik lokacji ma ustaloną wartość, a raz obliczone słowo wieloadresowe może zostać powielone w następnych komórkach pamięci zgodnie z zadaną krotnością, nie podlegając przy tym ponownemu obliczaniu.

Przykłady liczb wieloadresowych	
Denotacja	*Objaśnienie*
`(*)k21`	Słowo zawierające bieżący adres w skali 21
`[5](*)k21`	5 słów, każde zawierające adres pierwszego z nich
`(*)k21 (*)k21`	2 słowa, każde zawierające swój adres
`(pocz)k21+(kon)k34`	Słowo zawierające dwa adresy: w skali 21 i w skali 34
`(xxx)k21+(yyy)k34+(zzz)k8`	Trzy liczby, trzecia na bitach 35...38,0...8 słowa
`(2'A')k4+(2'B')k9+(2'C')k14`	Kody znaków A, B i C na 15 najstarszych bitach słowa
`(tablica+2*dłElemT)k21`	To np. adres drugiego, licząc od zera, elementu tablicy

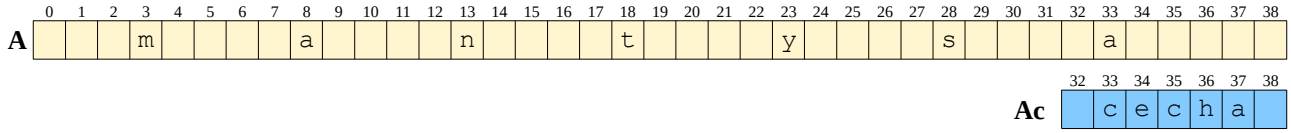
Słowo liczbowe zmiennoprzecinkowe

Słowa liczbowe zmiennoprzecinkowe są to słowa będące argumentami rozkazów wykonujących operacje zmiennoprzecinkowe: pobierania, posyłania, dodawania, odejmowania, mnożenia i dzielenia, wraz z towarzyszącymi im zaokrągleniem logicznym i normalizacją wyniku, a więc zawierającymi liczby zmiennoprzecinkowe.

Liczba zmiennoprzecinkowa jest w istocie parą dwóch liczb: mantysy i cechy. Mantysa traktowana jest jako liczba ułamkowa w skali 0, czyli z zakresu <-1, 1). Cecha jest liczbą całkowitą z zakresu <-64, +63>, wyznaczającą rząd wielkości mantysy. Para tych części jest w ODRZE 1003/1013 reprezentowana w dwóch formatach. Pierwszy format odnosi się do liczby zmiennoprzecinkowej upakowanej w słowie pamięci, lub w 39-bitowym rejestrze (zwykle w B7 – w krótkich rejestrach 13-bitowy fragment jest uzupełniany bitami zerowymi). Mantysa znajduje się tu na bitach 0...31, zaś cecha na bitach 32...38 słowa:



Drugi format odnosi się do liczby znajdującej się w rejestrze zmiennoprzecinkowym AC, składającym się z rejestru akumulatora A zawierającego mantysę, oraz z bezpośrednio niedostępnego, 7-bitowego rejestru Ac zawierającego cechę:



Wielkość liczby w reprezentacji zmiennoprzecinkowej określa wzór: **liczba = mantysa * 2^{cecha}**. Konwersję między oboma formatami zapewniają rozkazy pobierania do, i posyłania z rejestru AC.

AsmC umożliwia definiowanie słów zmiennoprzecinkowych w pierwszym formacie, w denotacji dziesiętnej lub bitowej (tj. binarnej, ósemkowej lub szesnastkowej). Na denotację składają się:

```
+  0123456789      E-12
-  012345.6789     E+12
   .0123456789     E12
   0123456789.
```

```
+  0B01.01         P-12
-  0C0123.4567     P+12
   0X012345.6EFC   P12
```

- opcjonalny znak mantysy: **+** lub **-** (brak znaku oznacza liczbę dodatnią)
- wartość bezwzględna mantysy, podana dziesiętnie, albo bitowo:
  – mantysa musi zawierać przynajmniej jedną cyfrę,
  – mantysa może zawierać kropkę oddzielającą część całkowitą mantysy od ułamkowej,
  – kropkę można pominąć, jeśli denotacja zawiera wykładnik
- **$E^{rząd}$** – wykładnik potęgi liczby 10 (**E**) wyznaczający rząd wielkości mantysy dziesiętnej, lub **P^{cecha}** – wykładnik potęgi liczby 2 (**P**) podający cechę, gdy mantysa jest zapisana bitowo

Wykładnik jest zawsze zapisywany dziesiętnie, w postaci wartości bezwzględnej z ew. znakiem **+** lub **-** (brak znaku oznacza wykładnik dodatni). Wykładnik (wraz z literą **E** lub **P** wskazującą podstawę potęgi) można pominąć, ale wtedy mantysa musi mieć kropkę.

Litery oznaczające podstawę systemu pozycyjnego: **0B**, **0C** i **0X**, wskazujące podstawę potęgi: **E** i **P**, oraz literowe cyfry szesnastkowe mogą być wielkie lub małe.

Cyfry liczb dziesiętnych, binarnych, ósemkowych i szesnastkowych można dowolnie grupować przy pomocy znaku podkreślenia **_**. Denotacja liczby nie może zawierać odstępów.

Liczba zmiennoprzecinkowa jest zapisywana w słowie w postaci znormalizowanej – by uzyskać postać nieznormalizowaną należy posłużyć się jakąś inną notacją dającą wymaganą reprezentację.

Mantysa może mieć dowolną liczbę cyfr. W zapisie dziesiętnym zostanie zaokrąglona logicznie. W zapisie bitowym (binarnym, ósemkowym lub szesnastkowym) musi być dokładna, tj. wszystkie bity znormalizowanej mantysy muszą zmieścić się na bitach 0...31 słowa.

Przykłady liczb zmiennoprzecinkowych

Denotacja	*Objaśnienie*
`1234567.89012`	Liczba zmiennoprzecinkowa, bo zawiera kropkę
`3.141592654`	= `0b11.00100100001111110110101010001`
`-123.891e-12`	= `-.000000000123891`
`-0b10100.11011`	= `-0b.10100_11011p5`
`0c17.6p-12`	= `0b.000000001111_110`
`0x1EF.6p-12`	= `0b.000111101111_0110`
`0c0.12345670457`	= `0b.00`1010011100101110111000100101111 liczba dokładna
`0c0.22345670457`	= `0b.0`100100111001011101110001001011**1** liczba za dokładna
`0c0.22345670456`	= `0b.0`100100111001011101110001001011`0` liczba dokładna
`0c0.22345670455`	= `0b.0`100100111001011101110001001011**0** liczba za dokładna

`127,99999992`
liczba taka sama jak niżej, bo nastąpi zaokrąglenie logiczne bitem czerwonym na pozycji zielonego
`127,999999999999`
	= `0b`1111111.11111111111111111111**0**
	= `0b`1111111.11111111111111111111**1**

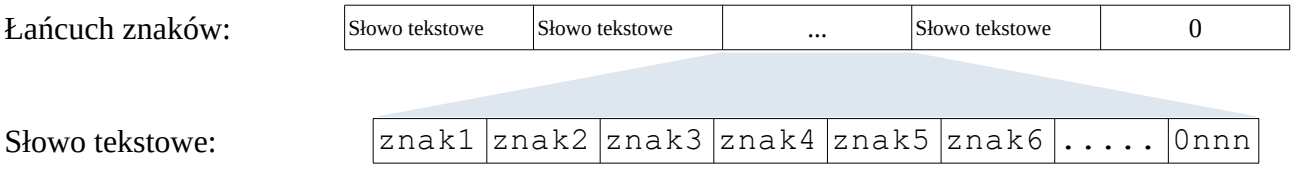
`0b1.11000111000111000111000111000 1`	słowo=`0.11100011100011100011100011100 0`0000001
`-0b1.11000111000111000111000111000 1`	słowo=`1.00011100011100011100011100011 1`0000001
`-0b1.0p+63`	słowo=`1.00000000000000000000000000000 0`111111
`0b1.0p+63`	słowo=`0.10000000000000000000000000000 0`1000000 nadmiar zmiennoprzecinkowy, cecha +64 jest zbyt wielka
`0c00000000000000000.7654321`	= `0c0.7654321`

Słowa łańcuchów znakowych i łańcuchy słów tekstowych

ODRA 1003/1013 została zbudowana głównie z myślą o obliczeniach inżynierskich i naukowych. Przetwarzanie tekstów nie było priorytetem, a więc wybór dalekopisu, już istniejącego urządzenia, do komunikacji z operatorem i drukowania wyników obliczeń, był ekonomicznie uzasadniony. Wraz z dalekopisem oraz czytnikami i perforatorami taśmy 5-kanałowej przyjęto powszechnie wtedy stosowany w telekomunikacji kod dalekopisowy nr 2 (ITA2). Specyfiką kodu ITA2 są dwa zestawy znaków: poczet liter i poczet znaków specjalnych (krótko – poczet cyfr), przełączanych znakami zmiany pocztu: LS (Letter Shift) i FS (Figure Shift). Niewielki zestaw znaków kodu ITA2 rozbudowywano o dodatkowe poczty, szczególnie gdy w grę wchodziła konieczność stosowania znaków narodowych języków innych niż angielski, a to prowadziło do bardziej złożonych sposobów przełączania pocztów, w tym wykorzystanie do tego celu również kodu NU (Null).

W emulatorze oprogramowano najbardziej typowe sposoby przełączania dwóch, trzech i czterech pocztów, opisane w [4]. W każdym z nich liczba niewidocznych kodów LS, FS i NU znajdujących się w tekście pomiędzy znakami drukowalnymi może być dowolnie wielka. Oznacza to trudność w szacowaniu wielkości pamięci potrzebnej do zapamiętania tekstu określonej długości, a także komplikuje bardziej złożone operacje na tekstach, niż zwykłe drukowanie.

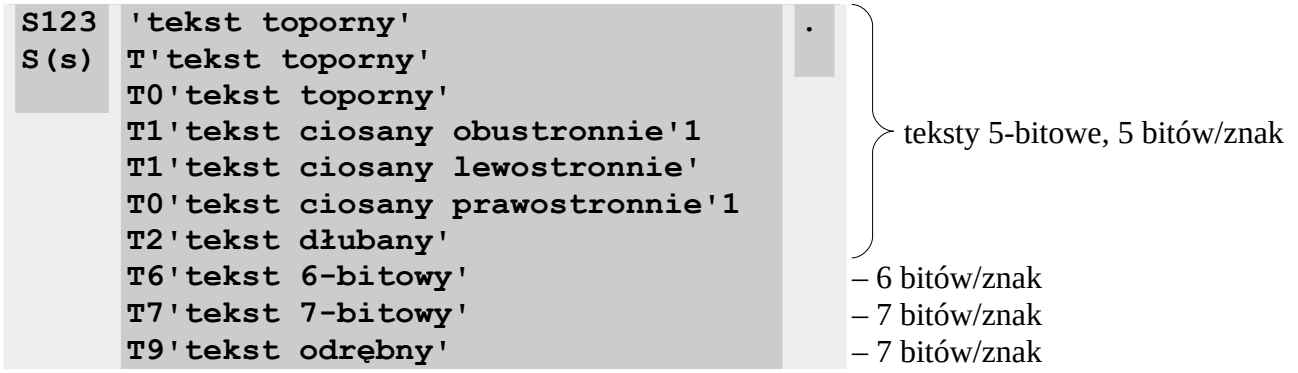
Niewielka pojemność pamięci maszyny wymusza pakowanie w jednym słowie po kilka znaków tekstu. To również utrudnia operacje takie jak dzielenie tekstu na części, czy łączenie w jedną całość. Przy małej szybkości komputera bardziej zaawansowane manipulacje są wykluczone.



- Łańcuch znaków tekstowych to ciąg słów tekstowych, zakończony wyzerowanym słowem. Można zażądać, by nie było zerowego słowa kończącego tekst.
- Każde słowo tekstu zawiera kody znaków dosunięte do lewej strony. Jeśli znaków jest mniej, niż mieści się w słowie, to wakujące bity po ostatnim znaku są zerowane.
- Na ostatnich 3 bitach słowa znajduje się licznik nnn od 0 do 7 wskazujący liczbę znaków znajdujących się w słowie, dzięki czemu wakujące bity nie muszą być wyzerowane.
- Wyzerowane całkowicie słowo zaznacza koniec tekstu. Niezerowe słowo, w którym licznik nnn jest wyzerowany, zawiera 0 znaków, ale nie oznacza końca tekstu, a więc może być użyte jako wypełniacz wycinający słowo z tekstu.

UWAGA: Reprezentacja wewnętrzna łańcuchów znakowych w AsmC jest nieco odmienna od języka PODSTAWOWEGO: w języku PODSTAWOWYM wakujące bity pozostają (jako NU) wyzerowane na początku słowa, co oznacza, że kod NU musi być obojętny przy drukowaniu i nie można go używać do przełączania pocztów znakowych. Nie ma też licznika znaków w słowie znakowym – bity licznika są zwykle wyzerowane, ale nie muszą. Rolę wypełniacza przy wycinaniu słowa z tekstu pełni więc tam słowo, w którym jedynie bity licznika są niezerowe.

AsmC udostępnia kilka cech dotyczących tekstów, które przy istniejących uwarunkowaniach pozwolą radzić sobie z podstawowymi trudnościami. Denotacja tekstu jest następująca:



Atrybut **S** – liczba słów pamięci przeznaczonych na tekst, np: `S(kon-pocz)T0'Abc123'`

Tekst, wraz z wyzerowanym słowem zaznaczającym koniec tekstu, musi zmieścić się w podanej liczbą dziesiętną lub wyrażeniem adresowym w nawiasach, liczbie słów. Niewykorzystane końcowe słowa są zerowane. Brak atrybutu S oznacza, że nie ogranicza się długości tekstu (z wyjątkiem ogólnego ograniczenia przez AsmC) i zajmuje on niezbędną liczbę słów pamięci.

Atrybut **T** – typ tekstu – sposób kodowania. Tekst może być jednego z poniższych typów:

- toporny – np. `T0'Abc123'` lub `T'Abc123'` lub `'Abc123'`
- ciosany – np. `T1'XABC'` lub `T1'XABCX'1` lub `T0'ABCX'1`
- dłubany – np. `T2'123/R/N/LABC'`
- 6-bitowy – np. `T6'ABC123'`
- 7-bitowy – np. `T7'ABC123'`
- odrębny – np. `T9'ABC123'`

Atrybut **.** – pominięcie zera na końcu tekstu, np: `S2T0'Abc12356'.`

Jest to żądanie, by tekst nie był zakończony wyzerowanym słowem zaznaczającym koniec tekstu. Kropkę stawia się na końcu całej denotacji, jako znak konkatenacji. Tekst bez kończącego zera będzie zwykle używany jako fragment większego tekstu. Atrybut S tak czy owak powoduje uzupełnienie tekstu zerami do wymaganej długości.

Treść tekstu zawsze musi być ujęta w apostrofy. Apostrof zawarty w treści tekstu musi być podwojony, jak np. `'Shakespeare''s ''Hamlet'''`.

T0 – Tekst toporny

Jest to najprostszy w notacji tekst, kodowany w 5-bitowym kodzie ITA. W łańcuchu znaków zostaną umieszczone wszystkie kody zawarte między otaczającymi apostrofami. Niewidoczne znaki sterujące LS, FS, NU i CR mogą nawet wielokrotnie powtarzać się bez widocznych na wydruku objawów, przez co tekst może stać się niepotrzebnie długi. Określenie tekstu bierze się stąd, że przy jego pisaniu nie dba się o niewidoczne niuanse – istotny jest szybki i prosty skutek.

Tekst toporny nadaje się do bezpośredniego drukowania na dalekopisie, pod warunkiem, że dalekopis jest aktualnie w trybie pracy zgodnym z trybem w którym zakodowano tekst (tryb wielkich liter, lub tryb małych liter), oraz ma wybrany poczet cyfr, w którym to znajduje się też apostrof otwierający tekst.

Apostrof otwierający tekst narzuca odpowiednie kody przełączające poczet znaków cyfrowych na wymagany przez pierwszy znak tekstu, zaś apostrof zamykający tekst narzuca odpowiednie kody przełączające poczet ostatniego znaku tekstu na poczet cyfr, w którym znajduje się apostrof. Narzucone przez apostrofy kody są częścią tekstu topornego, zatem po jego wydrukowaniu dalekopis pozostanie w poczcie cyfr.

Przykłady łańcuchów znakowych (tekstów) typu T0 – topornych	
Denotacja *Kody*	*Objaśnienie*
`'123'.`	Zakładając, że nie istnieje tryb małych liter, są to słowa:
`123` ← o takich kodach	= `0b10111001100001000000000000000000000011`
`▲123` ← lub takich	= `0b11011011110011000010000000000000000100`
`12▼▲3` ← lub jeszcze innych	= `0b10111001111111101100001000000000000101`
`'ABC'.`	Zakładając, że nie istnieje tryb małych liter, są to słowa:
`▼ABC▲` ← o takich kodach	= `0b11111000111001011101101100000000000101`
`▼AB▼▼C▲` ← lub takich	= `0b11111000111001111111111101110110110111`
`▼AB▲▼C▲` ← lub jeszcze innych	= `0b11111000111001110111111101110110110111`
`'AB12'.`	Zakładając, że nie istnieje tryb małych liter, są to słowa:
`▼AB▲12` ← o takich kodach	= `0b11111000111001110111011110011000000110`
`▼AB▲1▲2` ← lub takich	= `0b11111000111001110111011111011100110111`
`▼AB▲▲1▲2` ← lub jeszcze innych	= `0b11111000111001110111011101111011011111` `0b10011000000000000000000000000000000001`
`'Żą́ć'.`	Zakładając 4 poczty w trybie małych liter, są to słowa:
`▲Ż▲ą▲ć` ← o takich kodach	= `0b11011011011011100011101111001000000110`
`▲▼▼▲▲Ż▲ą▲ć` ← lub np. takich	= `0b11011111111111111011101110110110110111` `0b10001110111001000000000000000000000011`

Zwróćmy uwagę, że bezpośrednio za apostrofem otwierającym łańcuch znaków cyfrowych, kody przełączające na poczet cyfr są zbędne, bo apostrof jest w poczcie cyfr. Cyfry wydrukują się jako cyfry pod warunkiem, że dalekopis będzie też w poczcie cyfr. Jeśli dalekopis nie jest w poczcie cyfr, to wymagane jest najpierw przełączenie go na poczet cyfr, np. przy pomocy kodu FS zaraz za apostrofem.

Kolejne przykłady pokazują, że w łańcuchu znaków mogą znaleźć się zbędne, jałowe kody przełączające, które jednak zajmują miejsce, wymagając czasami dodatkowych słów pamięci.

W ostatnim przykładzie widzimy, jak wiele kodów trzeba do włączenia od nowa (w sposób niezależny od stanu dalekopisu), czwartego pocztu znaków (w którym jest litera Ż).

T1 – Tekst ciosany

Wyobraźmy sobie dwa teksty, które będziemy drukować jeden po drugim (dla uproszczenia literowe, w trybie wielkich liter):

Denotacje: `'THANASIS '.` `'KAMBURELIS'`
Kody: `▼THANASIS ▲` `▼KAMBURELIS▲`

Zaznaczone szarym tłem kody FS i LS na styku tekstów są w tej sytuacji zbędne, gdyż wykonują jałowe przełączenie dalekopisu na poczet cyfr i z powrotem na poczet liter, marnując czas urządzenia i (co istotne przy wielkiej liczbie takich tekstów) miejsce w pamięci.

Tekst ciosany umożliwia usunięcie z jednego lub obu krańców zbytecznych kodów przełączających poczty: LS, FS i NU (▼, ▲ i ●), a jednocześnie ustawienie takich kodów, jakie mają znaleźć się na styku tekstów. W powyższym przykładzie można to zrobić tak:

Denotacje: `T0'THANASIS X'1.` `T1'YKAMBURELIS'`
Kody bez ociosania: `▼THANASIS X▲` `▼YKAMBURELIS▲`
Kody po ociosaniu: `▼THANASIS` `KAMBURELIS▲`

Cyfra 1 za apostrofem zamykającym tekst usuwa od końca wszystkie kody przełączające, oraz jeszcze jeden znak, znajdujące się na końcu tekstu.

Cyfra 1 przed apostrofem otwierającym tekst usuwa wszystkie kody przełączające, oraz jeszcze jeden znak, znajdujące się na początku tekstu.

Kody zaznaczone w przykładzie żółtym tłem zostaną usunięte. Usuwane znaki X i Y zostały podane w denotacji tekstu tylko po to, by zostały usunięte. Mogą to być dowolne znaki różne od kodów LS, FS, NU, SP, CR i LF (bo te znajdują się w każdym poczcie), ale powinny należeć do pocztu zgodnego z pocztem dołączanego tekstu. Pomiędzy znakiem usuwanym, a znakiem pozostawionym mogą znajdować się kody przełączające, które już nie zostaną usunięte. Dzięki odpowiedniemu dobraniu znaku usuwanego, tj. dobraniu go z odpowiedniego pocztu, między znakami usuwanym i pozostawianym znajdą się odpowiednie kody przełączające, które wtedy pozostaną na krańcu tekstu. Technika ta pozwala uzyskać kody przełączające potrzebne w czasie działania programu, nie zaś potrzebne dla przejść od i do apostrofów otaczających tekst. Nie trzeba zastanawiać się, jakie to mają być kody – wystarczy podać do usunięcia znaki z odpowiednich pocztów. Ilustruje to poniższy przykład (przy założeniu, że aktualnie dalekopis działa w trybie małych liter):

Denotacje: `T0'Thanasis x'1.` `T1'xKamburelis'`
Kody bez ociosania: `▼▼Thanasis x▲` `▼x▼Kamburelis▲`
Kody po ociosaniu: `▼▼Thanasis` `▼Kamburelis▲`

Z pierwszego tekstu usunięte zostaną mała litera x, oraz kod przełączający z małych liter na cyfry (na apostrof) – tak ociosany napis pozostawi dalekopis w poczcie małych liter. Z drugiego tekstu usunięte zostaną kod przełączający z cyfr (z apostrofu) na małe litery, oraz mała litera x, a pozostaną kod przełączający z małych liter na jedną wielką literę, oraz pozostałe znaki. Bez ociosania tekst `T0'Kamburelis'` zawierałby kody `▼▼Kamburelis▲` – tzn. przełączenie z apostrofu na małe litery i przełączenie z małych liter na jedną wielką literę.

T2 – Tekst dłubany

Teksty pisane w kodzie ITA mają tę wadę, że łatwo może się do nich wkraść niepożądany, zbędny kod, niewidoczny lub trudny do zauważenia, jak np. LS, FS, CR, LF lub NU. Może być wygodniej polegać wyłącznie na widocznych kodach, zachowując pełną kontrolę nad łańcuchem znaków. Służy do tego tekst dłubany.

Asembler usuwa z tekstu dłubanego wszystkie niewidoczne 5-bitowe kody: LS, FS, CR, LF i NU. Inne znaki, w tym znak SP widoczny na wydruku jako odstęp, pozostają w tekście jako 5-bitowe kody ITA. Wyjątkiem są sekwencje kodowania przy pomocy znaku ucieczki /, kiedy cała sekwencja koduje jeden znak. Przy pomocy specjalnych sekwencji rozpoczynających się znakiem ucieczki / można kodować znaki LS, FS, CR, LF, NU i /, jak również dowolne znaki podając ósemkowo, dziesiętnie lub szesnastkowo wartość liczbową ich kodu:

Kodowanie znaków 5-bitowych przy pomocy znaku ucieczki / w tekście typu T2 – dłubanym

Notacja	*Wartość liczbowa*	*Przykład denotacji* *Kody*	*Objaśnienie*
`/L`	`31`	`T2'1/LAB'.` `1▼AB`	LS (Letter Shift) – przełączenie na poczet liter
`/F`	`27`	`T2'/LA/F2'.` `▼A▲2`	FS (Figure Shift) – przełączenie na poczet cyfr
`/R`	`8`	`T2'1/R23'.` `1⌂23`	CR (Carriage Return) – powrót karetki na początek wiersza Drukowanie znaku CR na dalekopisie emulatora niczego nie zmienia
`/N`	`2`	`T2'1/N23'.` `1≡23`	LF (Line Feed) – przejście do nowego wiersza Drukowanie znaku LF na dalekopisie emulatora wyprowadza kody New Line zgodne z systemem hosta: CRLF w Windows, LF w innych
`/Z`	`0`	`T2'1/Z23'.` `1●23`	NU (Null) – znak pusty, lub przełączenie na poczet cyrylicy
`T2'ab cd'`	`2`	`T2'1234 56'.` `123456`	Złamanie łańcucha znaków przejściem do nowej linii jest ignorowane
`/Ccc`	`0Ccc`	`T2'1/c337'.` `1▲7`	Znak o dwucyfrowym kodzie ósemkowym cc
`/Ddd`	`dd`	`T2'1/d279'.` `1▲9`	Znak o dwucyfrowym kodzie dziesiętnym dd
`/Xxx`	`0Xxx`	`T2'1/x1FA'.` `1▼A`	Znak o dwucyfrowym kodzie szesnastkowym xx
`//`	`29`	`T2'1//234'.` `1/234`	Znak dzielenia w tekście typu T2 musi być podwojony W tekstach innego typu nie da się kodować ze znakiem ucieczki
`''`	`5`	`T2'1''234'.` `1'234`	Apostrof musi być podwojony w tekście każdego typu

Zauważmy, że w tekście T2, zamiast liter można podawać znaki figurowe i na odwrót; istotne są jedynie wartości kodów. Np. tekst: `T2'/LT/Lhanasis/F'`, czyli `▼T▼hanasis▲`, możemy z równym skutkiem zakodować jako `T2'/Lt/LHaNaSiS/F'` lub `T2'/L5/LŁ-,-#8#/F'`.

T6 – Tekst 6-bitowy

Teksty typu T0, T1 i T2 są różnymi sposobami radzenia sobie z trudnościami przy posługiwaniu się 5-bitowym kodem ITA2. Trudność szacowania wielkości pamięci potrzebnej do zapamiętania tekstu określonej długości, a tym samym np. wycinania z większego tekstu jego fragmentów, przełamuje kodowanie 6-bitowe typu T6. Jest to propozycja asemblera AsmC, polegająca na uzupełnieniu kodu każdego znaku o jeden bit na najstarszej pozycji, niosący informację o poczcie do którego należy ten znak. Bit ten ma wartość 0 dla znaku z pocztu liter, zaś 1 dla znaku z pocztu cyfr. Kodowanie 6-bitowe nadawałoby się do standardowego alfabetu ITA składającego się z dwóch pocztów znakowych: wielkich liter i znaków figurowych. Teksty 6-bitowe nie wymagają kodów NU, LS i FS przełączających poczty, a więc jest oszczędność na liczbie znaków. Położenie każdego znaku w tekście jest przewidywalne, ale w jednym słowie można zmieścić tylko 6 takich znaków i wczytywanie i drukowanie wymaga odpowiedniego przekodowywania:

Słowo tekstowe:

pznak1	pznak2	pznak3	pznak4	pznak5	.....	nnn

Przykłady łańcuchów znakowych (tekstów) typu T6 – 6-bitowych

Denotacja	*Objaśnienie*
`T6'ABC123'.` `ABC123`	= 0b0000110110010011101101111100111000001110
`T6'AB` `C12'.` LF w poczcie 0 → `AB≡C12` albo w poczcie 1 →	Zakładając, że złamanie tekstu zawiera tylko znak LF: = 0b0000110110010000100111011011110011110 = 0b0000110110011000100111011011110011110
`T6'A B` `'.` SP, CR, LF w poczcie 0 → `A◡B⌂≡` albo w poczcie 1 →	Jw. Poczty znaków CR, LF, SP są dowolne: = 0b0000110001000110010010000001000000101 = 0b0000111001000110011010001000100000101

T7 – Tekst 7-bitowy

Tekst typu T7 jest 7-bitowym odpowiednikiem kodowania typu T6, ale dostosowanym do alfabetu z trzema lub czterema pocztami znaków. Różnica polega na dodaniu nie jednego, a dwóch bitów na najstarszych pozycjach, niosących informację o poczcie, do którego należy ten znak. Bity te mają wartość 00 dla znaku z pocztu wielkich liter, 01 dla znaku z pocztu cyfr, 10 dla znaku z pocztu małych liter, a 11 dla znaku z pocztu dodatkowych znaków specjalnych. W jednym słowie można zmieścić 5 znaków 7-bitowych:

Słowo tekstowe:

ppznak1	ppznak2	ppznak3	ppznak4	.....	0nnn

Można podać przykłady podobne do 6-bitowych z tą różnicą, że poczty znaków CR, LF i SP mogą być dowolnymi spośród czterech możliwych.

T9 – Tekst odrębny

Asembler AsmC wewnętrznie, dla własnej wygody, przekodowuje źródłowe teksty wejściowe napisane w 4-pocztowym kodzie dalekopisu TTY MKD-2 PL4 emulatora na własny 7-bitowy, odrębny od pozostałych, kod wewnętrzny. W kodzie tym, nazywanym alfabetem Asemblera AsmC, nie ma podziału na poczty. Kody NU, LF, SP, CR, FS i LS, które mogły wystąpić w każdym z czterech pocztów, otrzymały pojedyncze reprezentacje, a zaoszczędzone dzięki temu, nieobsadzone pozycje kodowe mogą być używane na wewnętrzne potrzeby programów. Kolejność znaków jest też dobrana pod potrzeby asemblera. Kod ten usuwa wiele kłopotów związanych z kodowaniem ITA, ale kosztem intensywnego przekodowywania z i na kod ITA przy wprowadzaniu danych i drukowaniu wyników. AsmC udostępnia ten kod w postaci tekstów typu T9, gdzie w jednym słowie można zmieścić 5 znaków 7-bitowych:

Słowo tekstowe:

znak1	znak2	znak3	znak4	.....	0nnn

Przykłady łańcuchów znakowych (tekstów) typu T9 – odrębnych

Denotacja	*Słowa tekstu, z zerowym słowem kończącym tekst*

`T9'Zażółć gęślą jaźń'`
`Zażółć gęślą jaźń`
	`= 0b1010001101110011111101111011111100101010b1110111000010011000101111000111111000101`
	`0b1100111111011000001001100101101110001010b1111101111101000000000000000000000000010`
	`0`

`T9'`
`X Y F(X,Y)`
`'`
`⌂≡␣␣␣X␣␣␣Y␣␣␣␣␣F(X,Y)⌂≡`
	`= 0b0000001000001000001000000100000010001010b1001111000010000001000000100101000001010b`
	`0b0000100000010000001000000100000010001010b0111101010001110011110011000101000001010b`
	`0b0100100000000100000100000000000000000110b`
	`0`

Uwagi do tekstów

- Liczba nnn znaków w każdym słowie tekstu jest maksymalna, z wyjątkiem ostatniego, niezapełnionego w pełni, słowa. Znaki w końcowym słowie są dosunięte do lewej, co nie jest zgodne z językiem PODSTAWOWYM, ale nie powinno powodować większych problemów jeśli kod NU jest obojętny przy drukowaniu.
- Tekst kończy się wyzerowanym słowem, co jest zgodne z językiem PODSTAWOWYM. Kropka `.` kończąca literał tekstowy oznacza, że nie chcemy mieć tego zerowego słowa.
- Wstawienie zerowego słowa w środek tekstu kończy tekst – by temu zaradzić należy wstawiać wypełniacz, tj. niezerowe słowo z wyzerowanymi trzema ostatnimi bitami nnn.
- Tekst pusty nie zajmuje pamięci – zajmuje 0 słów, oraz wyzerowane słowo kończące tekst.
- Liczby tekstowe używane w wyrażeniach nie zawierają bitów licznikowych 0nnn lub nnn.

Rozkazy maszynowe

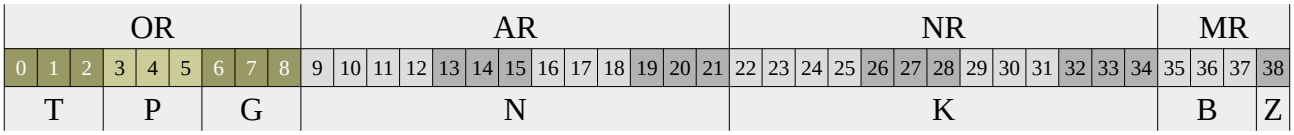
Instrukcje maszynowe, czyli instrukcje służące do generowania słów rozkazowych, będziemy nazywali krótko rozkazami. Każda instrukcja maszynowa generuje jedno słowo, które generalnie będzie zawierać rozkaz przeznaczony do wykonywania. Zwróćmy uwagę, że słowo wygenerowane przez instrukcję maszynową może być użyte również jako dana. Rozkazy mogą znajdować się też w blokach pozornych, w których jednakże nie następuje inicjalizacja pamięci.

Na wszystkie rozkazy maszynowe oddziałuje korektor ! przeciwdziałający przeniesieniu jedynki na kod operacji podczas B-modyfikacji.

Budowa rozkazów

W języku PODSTAWOWYM rozkaz zapisuje się przy pomocy cyfr ósemkowych, w formacie odpowiadającym formatowi słowa rozkazowego:

```
          :TPG NNNNN KKKKK BZ
np.       :050 10001 17207 00
```



Rozkaz maszynowy zawiera następujące części składowe:

OR część operacyjna – kod operacji, na który składają się trzy triady: T (typ), P (podstawienie) i G (grupa). W AsmC triada TPG jest generowana automatycznie na podstawie zapisu instrukcji maszynowej, stosownego dla funkcji rozkazu.

AR pierwsza część adresowa – adres argumentu lub sam argument N, w rozkazach skoku adres następnego rozkazu przy spełnionym warunku, a w niektórych rozkazach wartość nieistotna. W AsmC wartość N jest generowana na podstawie argumentu instrukcji maszynowej. Niektóre instrukcje pozwalają na pominięcie argumentu – wtedy przyjmuje się wartość 0. Niektóre instrukcje uniemożliwiają wskazanie zbędnego argumentu N (wtedy część AR jest zerowana) – wskazanie niezerowego N możliwe jest zawsze poprzez zapis w formacie :TPG.

NR druga część adresowa – adres K następnego rozkazu do wykonania, ignorowany w rozkazie wysyłającym wynik do rejestru rozkazów (wtedy część NR rozkazu wysłanego do rejestru R wskazuje następny rozkaz). Rozkazy są umieszczane w pamięci sekwencyjnie w kolejnych komórkach pamięci. Część NR rozkazu jest automatycznie wypełniana adresem następnego słowa, chyba że zostanie jawnie wskazany adres K następnej instrukcji. AsmC nie wspiera rozmieszczania rozkazów na ścieżkach w sposób optymalny, pozostawiając to automatycznemu przeplotowemu reżimowi adresowania.

MR część modyfikacyjna – triada B zawiera numer rejestru B zawierającego argument rozkazu lub użytego do B-modyfikacji rozkazu (jedno nie wyklucza drugiego). Bit Z ustawiony na 1 wskazuje B-modyfikację zawartością rejestru B. Część MR jest generowana automatycznie wg funkcji i argumentów instrukcji maszynowej. B-modyfikację innej niż AR i NR części rozkazu można wymusić podając rejestr B7 w którejś części adresowej.

Zauważmy, że budowa rozkazu uniemożliwia wskazanie w nim więcej niż jednego rejestru. Niedopuszczalne jest więc indeksowanie argumentów N i K różnymi rejestrami, lub wskazanie innego rejestru jako argument, a innego jako indeks. Asembler wychwytuje takie niespójności.

Pewna grupa rozkazów „sumowań” wykonuje operacje na argumencie bezpośrednim zapisanym w polu AR rozkazu, i zapisuje wynik w pamięci pod adresem zapisanym w tym właśnie polu AR. Argument bezpośredni jest wtedy ograniczony do wartości będącej adresem wynikowego słowa pamięci. Asembler wychwytuje niezgodność argumentu z adresem wyniku.

Istnieją rozkazy, które pobierają argument z pamięci i wynik także powinny zapisać w pamięci. Ponieważ jednak pamięć operacyjna jest zrealizowana na bębnie magnetycznym, i między pobraniem argumentu a zapisaniem wyniku musiałby upłynąć czas potrzebny na pełny obrót bębna, to rozkazy te w istocie nie zapisują wyniku z powrotem w pamięci. Asembler emituje przy tych rozkazach komunikaty HAK00 dla ostrzeżenia programisty przed czyhającą niespodzianką.

Niektóre rozkazy posyłania zmiennoprzecinkowego nie wykonują zaokrąglenia i normalizacji, pomimo że kod operacji tak by nakazywał. Asembler oznacza je zwracając na nie uwagę, ale nie traktuje tego jako błąd, lecz jako skrótowość zapisu.

B-modyfikacja rozkazu

Wykonanie rozkazu składa się z czterech etapów:

a) odszukanie rozkazu w pamięci

b) pobranie rozkazu

c) odszukanie argumentu i

d) wykonanie rozkazu z pobraniem argumentu lub zapisaniem wyniku

Etap b) – pobranie rozkazu z pamięci – polega na wczytywaniu kolejnych bitów słowa rozkazowego, poczynając od bitu Z (najmłodszego), decydującego o B-modyfikacji. Jeśli bit Z jest jedynką, to w trakcie wczytywania następuje dodawanie do rozkazu zawartości rejestru B o numerze wskazanym przez triadę B. Dodawanie to nazywane jest B-modyfikacją. Suma powstała w jej wyniku staje się zmodyfikowanym słowem rozkazowym posyłanym do rejestru rozkazów, na którym wykonywane są wszelkie dalsze działania procesora. Oryginalne słowo rozkazowe pozostaje niezmienione w pamięci. Zatem najpierw zawartość rejestru brana jest do modyfikacji adresu n lub k, albo całego rozkazu, a dopiero potem może być zmieniona jako argument operacji.

Dodawanie w trakcie B-modyfikacji jest zwykłym dodawaniem arytmetycznym w odpowiedniej skali, gdzie krótkie rejestry B1, B2, B3, B4, B5 i B6 zawierają liczby bez znaku, uzupełnione zerami z lewej i prawej strony do pełnej długości słowa:

- rejestr B7 jest dodawany do całego słowa rozkazowego
- rejestr B6 do wartości K w części NR słowa, czyli w skali 34
- rejestry B1, B2, B3, B4 i B5 do wartości N w części AR słowa, czyli w skali 21
- B-modyfikacja rejestrem B0 niczego nie zmienia – jest to rejestr fikcyjny zawierający zawsze 0, ale asembler traktuje go jak każdy inny

Jak widać, B-modyfikacja rejestrem B6 przewidziana jest zasadniczo do modyfikacji adresu następnego rozkazu w drugiej części adresowej, a rejestrami B1, B2, B3, B4 i B5 do modyfikacji pierwszej części adresowej. Wynik dodania zawartości rejestru może jednak przenieść się na starsze bity słowa rozkazowego, zmieniając nawet część operacyjną OP (TPG), czyli zmieniając słowo w jakiś inny rozkaz. W takiej sytuacji można uprzedzająco zapisać skorygowany rozkaz w postaci :TPG, albo zastosować korekcję rozkazu wykrzyknikiem.

Asembler AsmC automatycznie ustawia bity części MR rozkazu na podstawie argumentów N i K. Jeśli któryś z tych argumentów jest indeksowany, to rozkaz będzie B-modyfikowany. B-modyfikację innych części rozkazu można sprowokować podając rejestr B7 jako indeks którejś części adresowej. Asembler wychwytuje indeksowanie argumentu N lub K niewłaściwym rejestrem – rejestr B7 lub B0 jest zawsze właściwy.

Ustawianie warunków

Komputer ma jednobitowe rejestry warunków:

- U – ujemne
- Z – zerowe
- V – nadmiar (oznaczany na pulpicie maszyny i w innych publikacjach jako Nd)

Praktycznie wszystkie rozkazy ustawiają jakieś warunki, chociaż niektórych nie zmieniają. Nie ustawiają warunków rozkazy skoków warunkowych (SkZ, SkU, SkD, SkV, SkBG), skoków z licznikiem cykli (SkLC), skoku ze śladem (SkS), zatrzymania CPU (Stop), przesyłań blokowych (BF, FB), oraz niezdefiniowane (Nic). Dzielenie przez zero, lub nadmiar zmiennoprzecinkowy powodują zatrzymanie maszyny.

Rozkazy „sumowań” stałoprzecinkowych (grupy G=0,1,2,3), oraz faza „sumowania” rozkazów mnożenia i dzielenia stałoprzecinkowego (grupy G=4,5) – ustawiają warunki wg stanu sumatora, tj. stosownie do bitów przeniesień z pozycji bitowej nr 0 i z pozycji bitowej nr 1 sumatora. Oznacza to, że nadmiar przy dodawaniu liczb dodatnich ustawia warunek V, a zeruje U i Z, zaś nadmiar przy dodawaniu liczb ujemnych ustawia warunki V i U, a zeruje Z. Analogicznie dzieje się przy odejmowaniu. Jeśli wynik poślemy np. do akumulatora A, a następnie zawartość A sprawdzimy jakimś porównaniem, to warunki będą odmienne, bowiem żaden inny poza sumatorem rejestr czy komórka pamięci nie ma bitów przeniesień z pozycji bitowych nr 0 i nr 1.

Inne rozkazy ustawiają warunki wg stanu rejestru wynikowego: wg rejestru Am (czyli A) zawierającego mantysę liczby zmiennoprzecinkowej, wg rejestru AM zawierającego końcowy wynik mnożenia stałoprzecinkowego, lub wg rejestru A – wyniku (końcowego) dzielenia stałoprzecinkowego, dzielenia długiego, przesunięcia, wejścia/wyjścia, czytania klawiatury i zaokrąglenia normalnego bitem Ω. Oznacza to, że nadmiar przy przesunięciu w lewo o 1 pozycję liczby dodatniej może ustawić warunki V i U, albo V i Z, podczas gdy przy dodaniu dwóch takich samych liczb byłby to warunek V dodatni.

Rozkazy skoków warunkowych (SkZ, SkU, SkD, SkV i ich synonimiczne odmiany) wykonują skok zależnie od stanu rejestrów warunków Z, U i V. Rozkazy SkBG i skoków z licznikiem cykli (SkLC) wykonują skok na innej podstawie: stanu urządzenia lub wartości rejestru.

Notacja rozkazów w postaci :TPG

Notacja rozkazów stosowana w języku PODSTAWOWYM ułatwi w tym podręczniku objaśnianie instrukcji maszynowych. Asembler AsmC pozwala na analogiczną do języka PODSTAWOWEGO (ale nie identyczną) notację rozkazów maszynowych. Postać :TPG instrukcji maszynowej pozwala zapisać każdy rozkaz maszynowy. Przydaje się to np. przy zapisywaniu rozkazu stanowiącego daną na której wykonuje się operacje arytmetyczne, albo rozkazu w którym podczas B-modyfikacji może nastąpić przeniesienie na część operacyjną. Umożliwia też zapisanie w rozkazie nieistotnej wartości którejś części adresowej.

Instrukcja maszynowa w postaci :TPG ma następujący format:

```
etyk    ! :tpg   n , k , Bb +   ;komentarz
```

Dwukropek rozpoczynający rozkaz odróżnia postać :TPG rozkazu od innych formatów instrukcji maszynowych. Jest to wyróżnik postaci rozkazu, podobnie jak w języku PODSTAWOWYM.

Kod **tpg** jest to kod operacji rozkazu maszynowego, składający się z trzech cyfr ósemkowych stanowiących wartości kolejno triad T, P i G części operacyjnej (jak w języku PODSTAWOWYM). Dwukropek oraz cyfry kodu **tpg** nie mogą być rozdzielone między sobą odstępami.

Argument **n** jest to wartość N umieszczana w pierwszej części adresowej rozkazu, AR, jako argument lub adres argumentu rozkazu. Jest on wymagany i może być wyrażeniem dowolnego typu: Num, NumX, Adr, AdrX. Jeśli ma to być liczba ósemkowa, to musi być podana w postaci denotacji ósemkowej (inaczej niż w języku PODSTAWOWYM, gdzie wszystkie cyfry w zapisie rozkazu są uważane za ósemkowe).

Argument **k** jest to wartość K umieszczana w drugiej części adresowej rozkazu, NR, jako adres następnego rozkazu. Jest on opcjonalny. Jeśli zostanie pominięty, to zostanie dlań przyjęta wartość licznika lokacji ++, będącego odpowiednikiem znaku + w języku PODSTAWOWYM. Jeśli jest podany, to może być wyrażeniem dowolnego typu: Num, NumX, Adr, AdrX. Jeśli ma to być liczba ósemkowa, to musi być podana w postaci denotacji ósemkowej.

Argument **Bb** jest to rejestr B-modyfikacji B0, B1, B2, B3, B4, B5, B6 lub B7, którego numer jest umieszczany jako triada B w części modyfikacyjnej, MR, rozkazu. Jest on opcjonalny. Pominięty argument **Bb** może zostać zamarkowany wskazaniem numeru 0. Jeśli argument **n** lub **k** jest indeksowany, to argument **Bb** może być podany, ale nie jest to konieczne – asembler przyjmie za argument odpowiedni rejestr wraz ze znakiem **+** za nim, oznaczającym B-modyfikację.

Argument **+** oznacza B-modyfikację rozkazu rejestrem wskazanym w argumencie **Bb** (który musi być wtedy wskazany lub przynajmniej zamarkowany numerem 0 – numer 0 traktowany jest wtedy jako wskazanie rejestru B0). Pominięcie argumentu **+** to brak żądania B-modyfikacji, ale jeśli argument **n** lub **k** jest indeksowany, to rozkaz będzie B-modyfikowany. Asembler kontroluje dopuszczalność i zgodność rejestrów indeksujących argumenty **n**, **k** i **Bb**.

Przecinki rozdzielające argumenty mogą być zastąpione odstępami. W szczególnych sytuacjach, gdy nie następuje sklejenie operandów, argumenty mogą nawet się stykać. Pominięcie argumentu k musi być zamarkowane przecinkami jeśli jest podawany następny argument.

Przykłady:

W asemblerze AsmC				*W języku PODSTAWOWYM*
BLOK	0			; :401 00000:
:726	5	7	B1+	; :726 00005 00007 11
:726	5,	,	B1+	; :726 00005 + 11
:726	5			; :726 00005 + 00
:726	5+B7,	7+B7,	B7	; :726 00005 00007 71
:726	*+5+B7,	*+7+B7,	B7	; :726 00011 00013 71
:726	*+5+B7,	*+7+B7,	0	; :726 00012 00014 71
:726	*+5+B7,	*+7+B7,		; :726 00013 00015 71
:726	*+5+B7,	*+7+B7		; :726 00014 00016 71
:726	*+5+B7,	,		; :726 00015 + 71
:726	5	7	B1	; :726 00005 00007 10
:726	5	7	0 +	; :726 00005 00007 01
:766(1+2)4B1+		; stykające się argumenty		; :766 00003 00004 11

UWAGI:

- Argumentu **n** nie można pominąć
- Argument **k** można pominąć – wtedy domyślnie przyjmowane jest **++**
- Przecinki można pominąć, chyba że markują brak argumentu **k** przed argumentem **Bb**
- Wyrażenia **n** i **k** można dla przejrzystości ujmować w nawiasy, ale nie jest to konieczne.
- Argument **Bb** może być podany jako 0. Wskazanie 0+ oznacza jałową B-modyfikację
- Argumenty **Bb+** można pominąć, jeśli rejestr Bb został użyty (jawnie bądź niejawnie) w wyrażeniu **n** lub **k**

Notacja rozkazów w pozostałych postaciach

Wszystkie rozkazy maszynowe ODRY 1003/1013 zawierają w sobie drugą część adresową NR, w której zawarty jest adres następnego rozkazu do wykonania. Każdy rozkaz jest więc rozkazem skoku, co pozwala rozmieszczać rozkazy w pamięci w sposób optymalny pod względem czasu wykonania programu. Optymalizowanie programu poprzez rozrzucanie rozkazów w pamięci wprowadza jednakże trudną do ogarnięcia gmatwaninę, określaną „kodem spaghetti”, będącą zaprzeczeniem idei programowania strukturalnego. Konstruktorzy komputera, zdając sobie z tego sprawę, przewidzieli tzw. Reżim Adresowania Przeplotowego (włączany przyciskiem ZR na panelu maszyny), w którym kolejne adresy są rozstrzelone co 7 komórek pamięci. Adresowanie przeplotowe znacząco przyśpiesza pracę programu bez konieczności ręcznego rozrzucania rozkazów. Asembler AsmC nie oferuje optymalizacji rozmieszczenia rozkazów, zdając się na automatykę adresowania przeplotowego. Program wykonywalny wygenerowany przez AsmC może być wczytywany i wykonywany w trybie adresowania normalnego, albo przeplotowego. Generowane kolejno rozkazy umieszczane są w kolejnych komórkach pamięci, zatem gdy nie zachodzi potrzeba skoku do innej lokacji, wskazywanie adresu następnego rozkazu jest zbędne – asembler automatycznie wstawia adres następnej komórki pamięci.

Ogólny format instrukcji maszynowej jest następujący:

```
etyk    ! instrukcja opcje        ..nast      ;komentarz liniowy
etyk    ! SłowoKlucz argumenty    ..nast      ;komentarz liniowy
```

Etykieta rozkazu jest opcjonalna; jej wartością staje się bieżąca wartość licznika rozkazów.

Prefiksowy wykrzyknik ! korekty rozkazu jest opcjonalny.

Separator `..` jest opcjonalny – po nim musi być wyrażenie **`nast`** typu Adr lub AdrX, wskazujące adres następnego rozkazu. Pominięcie go powoduje przyjęcie bieżącej wartości licznika rozkazów `++` jako **nast**. Wartość argumentu **nast** jest wstawiana jako K do części adresowej NR rozkazu. Wyjątkiem są instrukcje skoków z zanegowanym warunkiem oraz rozkaz SKBG, gdzie wartość ta wstawiana jest jako N do części AR rozkazu (w tych przypadkach będziemy używali nazw odpowiednio z literą K lub N – np. nastN).

Etykieta rozkazu, wykrzyknik korekty, argument **`..nast`** i komentarz są wspólnymi elementami wszystkich rozkazów, więc ich opis nie będzie więcej powtarzany. Zasadnicza część instrukcji maszynowej może mieć notację podobną do instrukcji arytmetycznych języka C, analogiczną do instrukcji w innych typowych asemblerach jako słowo kluczowe operacji z operandami, a czasem do wyboru jedną i drugą. Rozkazy są pisane jednak dokładnie wg podanych schematów, z możliwością dodatkowych odstępów. Opis skupi się na zasadniczej części instrukcji maszynowej.

Przykłady:

W asemblerze AsmC			*W języku PODSTAWOWYM*
	`BLOK  0`		`; :401 00000:`
		`..ETYK`	`; :766 00000 00002 00`
	`NIC  0`	`..ETYK`	`; :766 00000 00002 00`
`ETYK`	`B1=A=A+7`		`; :472 00007 00003 10`
`PODPROG`	`M=B1, A=M, AM=A*M`		`; :064 00000 00004 10`
	`AC = AC + B7, BON`		`; :347 00000 00005 70`
	`A  = A <<< 5`		`; :016 00005 00006 00`
	`LL  5`		`; :016 00005 00007 00`
	`SKNZ  ETYK`		`; :046 00010 00002 00`
	`SKSB  B6,PODPROG`		`; :032 00011 00003 60`
	`B6=++..PODPROG`		`; :032 00012 00003 60`

UWAGI:

- Rozkaz `etyk..nast` bez zasadniczej części jest równoważny `etyk NIC 0..nast`
- Argument `..nast` można pominąć – wtedy domyślnie przyjmowane jest `..++`
- Niektóre rozkazy mają opcjonalne argumenty, jak np. NIC lub WE
- Przecinki, z nielicznymi wyjątkami, można pominąć
- AsmC oferuje dodatkowe mnemoniki dla niektórych rozkazów, jak np. SKNZ lub SKSB, opisane dalej przy rozkazach skoków warunkowych, skoków z licznikiem i rozkazie NIC

TPG=__0, __1, __2, __3 – „sumowania” stałoprzecinkowe

Rozkazy z tych grup są podstawowymi rozkazami arytmetyki stałoprzecinkowej, których wynik pojawia się na wyjściu sumatora i ustawia warunki. Wynik ten może być również posłany do akumulatora **A** i/lub do komórki pamięci operacyjnej **[n]**, rejestru B-modyfikacji **Bb** lub rejestru rozkazów **R**. Oczywiście obowiązują ograniczenia wynikające z konstrukcji maszyny: w jednej instrukcji może być tylko jedna wartość **n**, tylko jeden rejestr B-modyfikacji i tylko odpowiednie rejestry mogą być użyte do indeksowania. Instrukcje „sumowania” mają następujący format:

etyk	!	[n]=	A=	x	..nast	;komentarz	
␣␣␣␣␣␣␣		Bb=		-x			pobranie
		R=		|x|		zmiana znaku
				A-x			wartość bezwzględna
				A+x	można pisać	**A** zamiast **A+0**	odejmowanie
				x-A	można pisać	**-A** zamiast **0-A**	dodawanie
				A&x			odejmowanie przeciwne
				A^x			koniunkcja bitowa BOR
							różnica symetryczna XOR

gdzie:

[n] odwołanie do komórki pamięci poprzez wyrażenie adresowe **n** dowolnego typu: Num, NumX, Adr lub AdrX

Bb rejestr B-modyfikacji: B0, B1, B2, B3, B4, B5, B6 lub B7

R rejestr rozkazów

A rejestr akumulatora stałoprzecinkowego

x jeden z następujących argumentów:

0	liczba adresowa zero w dowolnej denotacji, nie ujęta w nawiasy
[n]	zawartość komórki pamięci o adresie zadanym wyrażeniem n dowolnego typu
Bb	rejestr B-modyfikacji: B0, B1, B2, B3, B4, B5, B6 lub B7
(n)	wyrażenie adresowe dowolnego typu ujęte w nawiasy – nawiasy ( ) można opuścić jeśli n nie zawiera operatorów, ale – nawiasy odróżniają wyrażenie (0) od argumentu 0 – nawiasy odróżniają wyrażenie (Bb) od argumentu Bb – wyrażenie (Bb) jest równoważne wyrażeniom (Bb+0) i (0+Bb) – niezerowa liczba n nawet bez nawiasów jest traktowana jak wyrażenie (n)

Przykłady:

`[zmienna] = zmienna`	Inicjalizacja komórki pamięci liczbą nieujemną
`[zmienna] = -zmienna`	Inicjalizacja komórki pamięci liczbą niedodatnią
`[zmienna] = A = 0`	Zerowanie komórki pamięci i akumulatora
`[zmienna]`	Pobranie (testowanie) zawartości komórki pamięci
`A  = |[zmienna+1]|`	Pobranie bezwzględnej wartości komórki pamięci
`B1 = ((kon-pocz)/3)`	Wstawienie liczby do rejestru B1
`A  = A + (2'H' << 5)`	Dodanie do akumulatora kodu znaku H w skali 16
`B2 = A + B2`	Dodanie do rejestru B2 zawartości akumulatora
`! B5 = (B5 - 8)`	Poprawne tylko gdy B5 >= 8

TPG=__4, __5 – mnożenia / dzielenia stałoprzecinkowe

Rozkazy z tych grup są złożone i wykonywane są w dwóch etapach:

– w pierwszym argument **x** jest pobierany do rejestru mnożnika **M** oraz na wejście sumatora, po czym jest wykonywana operacja „sumowania” jak w rozkazach z grup G=0, 1, 2, 3. Wynik tej operacji ustawia warunki i może być posłany do akumulatora **A**. Jeśli nie powstanie nadmiar, to wykonany zostanie drugi etap – w przeciwnym razie operacja zostaje przerwana.

– w drugim zawartość akumulatora (zmieniona lub wcześniejsza) jest mnożona lub dzielona przez zawartość **M**, a wynik zapisywany odpowiednio w rejestrze **AM** lub **A** i znowu ustawiane warunki.

Notacja odzwierciedla te czynności – rozkazy zapisywane są literalnie wg szablonu z wyjątkiem zmiennego argumentu x. Wokół operatorów i separatorów można stosować odstępy:

etyk ␣␣␣␣␣␣␣	**!**	**M=x**	**,**	**A=**	**M** **-M** **|M|** **A-M** **A+M** **M-A** **A&M** **A^M**	**,**	**AM=A*M** **A=A/M**	**..nast**	**;komentarz**	mnożenie dzielenie

gdzie:

M rejestr mnożnika, czyli rejestr B7

A rejestr akumulatora stałoprzecinkowego

AM 78-bitowy rejestr długi, tj. para rejestrów (A,M) czyli (A,B7), w której bit 0 rejestru B7 jest zerowany

x jeden z następujących argumentów:

0	liczba adresowa zero w dowolnej denotacji, nie ujęta w nawiasy
[n]	zawartość komórki pamięci o adresie zadanym wyrażeniem n dowolnego typu
Bb	rejestr B-modyfikacji: B0, B1, B2, B3, B4, B5, B6 lub B7
(n)	wyrażenie adresowe dowolnego typu ujęte w nawiasy

– nawiasy ( ) w tych instrukcjach można opuścić, ale
– nawiasy odróżniają wyrażenie (Bb) lub (0) od argumentu Bb lub 0
– wyrażenie (Bb) jest równoważne wyrażeniom (Bb+0) i (0+Bb)
– niezerowa liczba n nawet bez nawiasów jest traktowana jak wyrażenie (n)

Przykłady:

`M = B7, A = M, AM = A * M`	Obliczenie kwadratu liczby zawartej w rejestrze B7
`M=*-(SYMB+8)+B2, M, AM=A*M`	Iloczyn zawartości A i pewnej liczby indeksowanej
`M=[123], M, A=A/M`	Dzielenie przez zawartość komórki pamięci
`M=B5 -M AM=A*M`	Przecinki można pominąć, gdy x nie jest wyrażeniem
`M=(B5), -M AM=A*M`	Jeśli x jest wyrażeniem, a po nim ma być -M lub |M|, to brak przecinka spowoduje błąd składniowy
`M=9+B1, |M| AM=A*M`	

TPG=_16 – przesunięcia

Rozkazy przesunięć pozwalają przesuwać bity w akumulatorze **A** lub w rejestrze długim **AM**, w sumie 7 rozkazów. Są to przesunięcia logiczne, arytmetyczne i cykliczne, w lewo i w prawo.

Rozkazy przesunięć w notacji: mnemonik operand, mają następujący format:

etyk	!		i	..nast	;komentarz	
␣␣␣␣␣␣␣		**LL**				A logicznie w lewo
		LP				A logicznie w prawo
		AL				A arytmetycznie w lewo
		AP				A arytmetycznie w prawo
		CP				A cyklicznie w prawo
		ALD				AM arytmetycznie w lewo długie
		APD				AM arytmetycznie w prawo długie

Te same rozkazy przesunięć w notacji arytmetycznej mają następujący format:

etyk	!		(i)	..nast	;komentarz	
␣␣␣␣␣␣␣		**A=A <<<**				A logicznie w lewo
		A=A >>>				A logicznie w prawo
		A=A <<				A arytmetycznie w lewo
		A=A >>				A arytmetycznie w prawo
		A=A >><				A cyklicznie w prawo
		AM=AM <<				AM arytmetycznie w lewo długie
		AM=AM >>				AM arytmetycznie w prawo długie

gdzie:

A rejestr akumulatora stałoprzecinkowego

AM 78-bitowy rejestr długi, tj. para rejestrów (A,M) czyli (A,B7), w której bit 0 rejestru B7 jest zerowany

i wyrażenie adresowe typu Num lub NumX podające wielkość przesunięcia, przy czym:
– przy przesunięciach A wielkość przesunięcia określa 6 najmłodszych bitów
– przy przesunięciach AM wielkość przesunięcia określa 8 najmłodszych bitów
– starsze bity są ignorowane
– w notacji z mnemonikiem nawiasy otaczające operand są zbędne
– w notacji arytmetycznej nawiasy ( ) można opuścić jeśli argument **i** nie zawiera operatorów

Przykłady:

`A  = A  <<< 5`	Przesunięcie logiczne A w lewo o 5 bitów
`A  = A  >> (5 + B1)`	Przesunięcie arytmetyczne A w prawo
`A  = A  >>< 9`	Przesunięcie cykliczne A w prawo o 9 bitów
`AM = AM << 38`	Przesunięcie arytmetyczne AM w lewo o 38 bitów
`LL  5`	Przesunięcie logiczne A w lewo o 5 bitów
`AP  5 + B1`	Przesunięcie arytmetyczne A w prawo
`CP  9`	Przesunięcie cykliczne A w prawo o 9 bitów
`ALD  38`	Przesunięcie arytmetyczne AM w lewo o 38 bitów

TPG=716 – DzD – dzielenie długie stałoprzecinkowe

Rozkaz dzielenia długiego pozwala uzyskać nawet 252 bity ilorazu (w tym bit znaku i bit zaokrąglenia Ω), partiami po 39 bitów. Operacja dzieli zawartość rejestru **A** przez zawartość rejestru **M** (czyli B7), wykonując tyle kroków z przesuwaniem wyniku w (A,Ω), aż uzyska w rejestrze zaokrągleń bit nr (i-4) ilorazu (licząc bit znaku jako bit nr 0). Bity nie mieszczące się w rejestrze **A** są wypychane w lewo i tracone. Dzielnik w rejestrze **M** (czyli w rejestrze B7) pozostanie niezmieniony. Z argumentem **i** równym 43 wynik jest identyczny jak z rozkazu `M=B7,M,A=A/M` z grupy G=5. Rozkaz ten może być pisany w notacji: mnemonik operand, lub arytmetycznej:

etyk	**!**	**DZD**	**i**	**..nast**	**;komentarz**	dzielenie długie do bitu nr (i-4)
␣␣␣␣␣␣␣		**A//M ,**				dzielenie długie do bitu nr (i-4)

gdzie:

A rejestr akumulatora stałoprzecinkowego – dzielna

M rejestr mnożnika, czyli rejestr B7 – dzielnik

i argument bezpośredni typu Num lub NumX określający wymaganą liczbę (i-3) bitów wyniku łącznie z bitem nr 0 (bitem znaku) i bitem nr (i-4) w rejestrze zaokrągleń Ω.
Długość ilorazu w istocie określa 8 najmłodszych bitów argumentu – starsze są ignorowane.

Przykłady:

`A=[licznik]` `M=[mianown], M, A=A/M`	Licznik w A zostanie zastąpiony przez iloraz Mianownik M pozostanie niezmieniony w A bity ilorazu 0..38 w Ω = bit 39
`A = [licznik]` `A=A//M 43`	w A bity ilorazu 0..38 w Ω = bit 39
`A = [licznik]` `A=A//M,82`	w A bity ilorazu 39..77 w Ω = bit 78
`A = [licznik]` `DzD 121`	w A bity ilorazu 78..116 w Ω = bit 117
`A = [licznik]` `DzD 160`	w A bity ilorazu 117..155 w Ω = bit 156
`A = [licznik]` `DzD 199`	w A bity ilorazu 156..194 w Ω = bit 195
`A = [licznik]` `DzD 238`	w A bity ilorazu 195..233 w Ω = bit 234
`A = [licznik]` `DzD 255`	w A bity ilorazu 212..250 w Ω = bit 251 !
`Okr`	Zaokrąglenie normalne wyniku może wymagać propagacji przeniesienia na starsze słowa ilorazu
`LL 22`	Ostatnio uzyskane bity 0..21 akumulatora są to te same bity, co bity 17..38 akumulatora uzyskane w poprzednim dzieleniu

TPG=_26, _66 – We/Wy – operacje wejścia / wyjścia

Rozkaz wejścia z grupy TPG=_26 wczytuje 5-bitowy kod z urządzenia na bity 34..38 akumulatora.
Rozkaz wejścia z grupy TPG=_66 wczytuje 8-bitowy kod z urządzenia na bity 31..38 akumulatora.
Rozkaz wyjścia z grupy TPG=_26 wypisuje 5-bitowy kod na urządzenie z bitów 0..4 akumulatora.
Rozkaz wyjścia z grupy TPG=_66 wypisuje 8-bitowy kod na urządzenie z bitów 0..7 akumulatora.
Rozkazy grup TPG = _26 i _66 są zasadniczo bezargumentowe, ale AsmC pozwala podać opcjonalny argument **n**, który (choć nie ma istotnego znaczenia) w czasie wykonywania się rozkazu wyświetla się w części AR rejestru rozkazów. W przypadku pracy krokowej, oraz w przypadku emulatora pracującego na powolnych biegach, odpowiednio dobrany wskazuje stan programu:

```
etyk    ! WE     urz , n  ..nast  ;komentarz   czytanie z urządzenia nr urz
        ␣␣␣␣␣␣␣ WY                             pisanie na urządzenie nr urz
```

gdzie:

urz numer urządzenia wejścia/wyjścia (czytnika, perforatora, dalekopisu), typu Num:
0 = czytnik 5-kanałowy PTR0 5 = perforator 5-kanałowy PTP5 8 = czytnik 8-kanałowy PTR0'
1 = klawiatura dalekopisu, 6 = drukarka dalekopisu, 10 = czytnik 8-kanałowy PTR2'
2 = czytnik 5-kanałowy PTR2 13 = perforator 8-kanałowy PTP5'

n opcjonalny argument bezpośredni dowolnego typu: Num, NumX, Adr lub AdrX
brak argumentu powoduje przyjęcie wartości 0

TPG=326 – CzK – czytanie klawiatury akumulatora

Rozkaz wczytuje do akumulatora stan przycisków rejestru akumulatora A na pulpicie maszyny. Przycisk wciśnięty – to odpowiadający mu bit A ustawiony na 1, zwolniony – zresetowany na 0. Składnia i argument są analogiczne do rozkazów wejścia / wyjścia:

```
etyk    ! CZK    n  ..nast  ;komentarz   czytanie stanu przycisków akumulatora:
        ␣␣␣␣␣␣␣                           1=wciśnięty, 0=zwolniony
```

TPG=426 – Okr – zaokrąglenie normalne bitem Ω

Rozkaz dodaje bit zaokrąglenia Ω do akumulatora **A** na pozycji najmłodszego bitu nr 38. W wyniku dodawania może nastąpić propagacja przeniesienia na starsze bity. Rozkaz ma dwie notacje – mnemonik operand, oraz arytmetyczną. Argumentu **n** (dowolnego typu), który zasadniczo nie ma znaczenia, nie można wskazać w notacji arytmetycznej:

```
etyk    ! OKR    n  ..nast  ;komentarz   zaokrąglenie normalne – argument n nie
        ␣␣␣␣␣␣␣                           ma znaczenia
```

```
etyk    ! A=A+OM    ..nast  ;komentarz   zaokrąglenie normalne – argumentu n
        ␣␣␣␣␣␣␣                           nie da się wskazać
```

TPG=_36 – BF, FB – przesyłania blokowe na i z ferrytu

Rozkazy z tej grupy służą do szybkiego przesyłania zawartości komórek pamięci bębnowej do pamięci ferrytowej i odwrotnie. Przesyłanie rozpoczyna się od komórki wskazanej adresem **abębn** i powoduje przesłanie zawartości (Bb+1) kolejnych komórek jednej ścieżki – na najmłodszych siedmiu bitach części AR rejestru **Bb** ma być liczba przesyłanych komórek pomniejszona o 1. Po ostatniej komórce ścieżki następną do przesłania jest komórka numer 0 tej ścieżki. Przesyłane są zawartości komórek odpowiadających sobie strefowo. Zawartość rejestru **Bb** nie ulega zmianie. Jeśli adres **abębn** jest z zakresu pamięci ferrytowej, to w istocie wskazuje on ścieżkę bębnową zasłoniętą przez ścieżkę ferrytową – zasłonięte ścieżki bębnowe mogą więc posłużyć za schowek niedostępny dla innych rozkazów. Przesłanie zajmuje nie więcej niż 2 obroty bębna: na pozycjonowanie bębna do początkowej komórki i na przesłanie bloku komórek.

Rozkazy przesyłania blokowego mają następujący format:

```
etyk      !  BF      ferr  ,  Bb  ,  abębn  ..nast  ;komentarz
ooooooo      FB
```

bęben na ferryt
ferryt na bęben

gdzie:

ferr wyrażenie adresowe typu Num, wskazujące numer ścieżki ferrytowej na którą lub z której ma nastąpić przesłanie zawartości komórek:
61 = pierwsza ścieżka ferrytowa 62 = druga ścieżka ferrytowa

Bb rejestr B-modyfikacji B0, B1, B2, B3, B4, B5, B6 lub B7 zawierający pomniejszoną o 1 liczbę komórek do przesłania

abębn wyrażenie adresowe dowolnego typu: Num, NumX, Adr lub AdrX, wskazujące adres komórki pamięci bębnowej, od którego zaczyna się przesyłanie

Przykład:

`B2 = 127`	Liczba 127 oznacza przesłanie 128 komórek – załóżmy że jakiejś tablicy danych
`BF 61,B2,1024`	Przesłanie na pierwszą ścieżkę ferrytową 128 komórek począwszy od adresu 1024 (od początku ścieżki nr 8)
`SKSB B6,odwróć`	Jakieś operacje podprogramem **odwróć** na danych umieszczonych w pamięci ferrytowej
`B2 = 127`	Gdyby podprogram **odwróć** nie naruszył rejestru R1, to ta instrukcja byłaby zbędna
`FB 61,B2,1024`	Odesłanie z powrotem na bęben zawartości pierwszej ścieżki ferrytowej (wskazanej argumentem 61)

Bardziej złożone wykorzystanie przesyłania blokowego w maszynach wyposażonych w pamięć ferrytową polega na samoprzepisywaniu się programu, fragment po fragmencie na jedną ścieżkę ferrytową, pozostawiając drugą podprogramom lub danym. Każdy fragment, po wykonaniu się, kopiuje na ścieżkę ferrytową następny fragment i przekazuje mu sterowanie. Fragmenty programu, nie modyfikujące same siebie, nie muszą odsyłać się z powrotem na bęben. Opis wraz z przykładami znajduje się w rozdziale „Instrukcje przeadresowywania kodu”.

TPG=_46 – Sk_ – skoki warunkowe

Rozkaz skoku warunkowego powoduje przeniesienie sterowania do instrukcji pod adresem **rozg** gdy spełniony jest warunek skoku. Jeśli warunek nie jest spełniony, to sterowanie przechodzi do następnego rozkazu, **nast**. Dla lepszego oddania istoty warunku skoku, AsmC oferuje szereg dodatkowych mnemoników w uzupełnieniu podstawowych. Wśród dodatkowych mnemoników znajdują się rozkazy skoku przy przeciwnym warunku. Są to te same podstawowe rozkazy skoków warunkowych, ale powstają przez zamianę miejscami argumentów w częściach adresowych AR i NR rozkazów: adresów rozgałęzienia **rozg** i następnego **nast**. Zostało to zasygnalizowane w formatach instrukcji przez nazwanie argumentów jako **rozgK** i **nastN**, na znak, że stosują się do nich odpowiednio reguły indeksowania właściwe dla wartości K i N w częściach adresowych NR i AR: adresy **rozg** i **nastN** nie mogą być indeksowane rejestrem B6, a adresy **rozgK** i **nast** nie mogą być indeksowane rejestrami B1, B2, B3, B4 i B5.

Rozkazy skoków warunkowych mają następujący format:

etyk ⌣⌣⌣⌣⌣⌣⌣	**!**	**SKZ**	**rozg**	**..nast**	**;komentarz**	gdy zero
		SKU				gdy ujemne
		SKD				gdy dodatnie
		SKV				gdy nadmiar stp
		SKR			*synonim* `SKZ`	gdy równe
		SKM			*synonim* `SKU`	gdy mniejsze
		SKW			*synonim* `SKD`	gdy większe

etyk ⌣⌣⌣⌣⌣⌣⌣	**!**	**SKNZ**	**rozgK**	**..nastN**	**;komentarz**	gdy nie zero
		SKNU				gdy nie ujemne
		SKND				gdy nie dodatnie
		SKNV				gdy nie nadmiar stp
		SKNR			*synonim* `SKNZ`	gdy nie równe
		SKNM			*synonim* `SKNU`	gdy nie mniejsze
		SKNW			*synonim* `SKND`	gdy nie większe
		SKWR			*synonim* `SKNM`	gdy większe lub równe
		SKMR			*synonim* `SKNW`	gdy mniejsze lub równe

gdzie:

rozg adres dowolnego typu: Num, NumX, Adr lub AdrX, następnego rozkazu do wykonania gdy warunek jest spełniony – może być indeksowany rejestrem B0, B1, B2, B3, B4, B5 lub B7

nast adres dowolnego typu: Num, NumX, Adr lub AdrX, następnego rozkazu do wykonania gdy warunek nie jest spełniony – może być indeksowany rejestrem B0, B6 lub B7

rozgK adres dowolnego typu: Num, NumX, Adr lub AdrX, następnego rozkazu do wykonania gdy warunek jest spełniony – może być indeksowany rejestrem B0, B6 lub B7

nastN adres dowolnego typu: Num, NumX, Adr lub AdrX, następnego rozkazu do wykonania gdy warunek nie jest spełniony – może być indeksowany rejestrem B0, B1, B2, B3, B4, B5 lub B7

TPG=446 – SkBG – skok przy braku gotowości urządzenia

Jest to rozkaz zdefiniowany w maszynie UMCS do kontroli gotowości urządzenia wejściowego: czytnika taśmy lub dalekopisu. Czytnik jest w stanie gotowości, jeśli jego silnik jest włączony, taśma założona i spełnione są wszystkie warunki umożliwiające transmisję informacji. Dalekopis jest gotowy do transmisji informacji do maszyny, jeśli włączony został jego silnik. Rozkaz rozkaz czytania znaku musi czekać na sygnał gotowości, co może być niezauważone przez operatora.

Rozkaz :446 wykonuje skok, gdy urządzenie o numerze **Bb** nie jest gotowe. Program może wtedy zasygnalizować brak gotowości tekstem na dalekopis, lub poprzez wykonanie rozkazu STOP (co z kolei może spowodować sygnał dźwiękowy z monitora akustycznego). Aktualnie brak gotowości sygnalizowany jest dla urządzeń nr 0, 1 i 2. Pozostałe urządzenia uważane są za stale gotowe.

Zwróćmy uwagę, że rozkaz :446 ma odmienną semantykę niż pozostałe rozkazy maszyny. Numer kontrolowanego urządzenia zadaje się jako wartość pola B części MR rozkazu. Jednocześnie pole B wyznacza rejestr B-modyfikacji. Zatem B-modyfikacji tego rozkazu można dokonać tylko rejestrem o numerze zgodnym z numerem kontrolowanego urządzenia. Oto format tych instrukcji:

```
etyk    ! SKBG  Bb , rozg  ..nast  ;komentarz
```
gdy brak gotowości

```
etyk    ! SKNBG Bb , rozgK ..nastN ;komentarz
```
gdy nie brak gotowości

gdzie:

Bb numer rejestru B-modyfikacji będący zarazem numerem testowanego urządzenia

rozg adres dowolnego typu: Num, NumX, Adr lub AdrX, następnego rozkazu do wykonania gdy brak gotowości – może być indeksowany rejestrem B0, B1, B2, B3, B4, B5 lub B7

nast adres dowolnego typu: Num, NumX, Adr lub AdrX, następnego rozkazu do wykonania gdy urządzenie jest gotowe – może być indeksowany rejestrem B0, B6 lub B7

rozgK adres dowolnego typu: Num, NumX, Adr lub AdrX, następnego rozkazu do wykonania gdy urządzenie jest gotowe – może być indeksowany rejestrem B0, B6 lub B7

nastN adres dowolnego typu: Num, NumX, Adr lub AdrX, następnego rozkazu do wykonania gdy brak gotowości – może być indeksowany rejestrem B0, B1, B2, B3, B4, B5 lub B7

UWAGA: W maszynach nie UMCS jest to rozkaz niezdefiniowany i działa jako „Nic nie rób”. Dlatego rozkaz generowany instrukcją SKBG zawsze prowadzi w tych maszynach pod adres **nast**, zaś instrukcją SKNBG pod adres **rozgK**.

Przykład:

```
        SKNBG B2,gotowe          Test gotowości czytnika numer 2
        STOP  0c11111  ..++      Zatrzymanie programu z kodem 0b1001001001001
                                 i ew. sygnałem dźwiękowym z głośnika
gotowe  WE    2                  Czytanie z czytnika nr 2 taśmy 5-kanałowej
```

TPG=546, 646 – SkLC--, SkLC++ – skoki z licznikiem cykli

Rozkazy SKLC skoku z licznikiem cykli wykonują skok pod adres wskazany argumentem **adr**, gdy zawartość rejestru **Bb** przed dekrementacją lub inkrementacją jest niezerowa. Po określeniu decyzji o skoku zawartość rejestru jest pomniejszana (--) lub powiększana (++) o 1 modulo 8192 na części adresowej AR rejestru, a następnie odpowiednio do decyzji wykonywany jest skok pod adres **adr**, lub przejście do następnego rozkazu **nast**.

Instrukcje SKNLC są mnemonikami tych samych rozkazów dla skoków z przeciwnym warunkiem, tj. gdy zawartość rejestru przed dekrementacją lub inkrementacją jest zerowa, i powstają przez zamianę miejscami argumentów w częściach adresowych AR i NR rozkazów: adresów skoku **adr** i następnego rozkazu **nast**. Zostało to zasygnalizowane w formacie instrukcji przez nazwanie argumentów jako **adrK** i **nastN**, na znak, że stosują się do nich odpowiednio reguły indeksowania właściwe dla wartości K i N w częściach adresowych NR i AR: adresy **adr** i **nastN** nie mogą być indeksowane rejestrem B6, a adresy **adrK** i **nast** nie mogą być indeksowane rejestrami B1, B2, B3, B4 i B5.

Instrukcje skoków z licznikiem cykli służą do organizacji pętli i mają następujący format:

etyk	**!**	**SKLC**	**Bb**	**--** **++**	**,**	**adr**	**..nast**	**;komentarz**	z dekrementacją z inkrementacją

etyk	**!**	**SKNLC**	**Bb**	**--** **++**	**,**	**adrK**	**..nastN**	**;komentarz**	z dekrementacją z inkrementacją

gdzie:

Bb rejestr B-modyfikacji: B0, B1, B2, B3, B4, B5, B6 lub B7, zawierający liczbę cykli

adr adres dowolnego typu: Num, NumX, Adr lub AdrX, następnego rozkazu do wykonania, gdy liczba w rejestrze **Bb** przed dekrementacją lub inkrementacją jest niezerowa

adrK adres dowolnego typu: Num, NumX, Adr lub AdrX, następnego rozkazu do wykonania, gdy liczba w rejestrze **Bb** przed dekrementacją lub inkrementacją jest zerowa

Przykład:

```
         A  = 0                          Będzie sumowanie elementów tablicy w A
         B2 = tablica                    Adres początku tablicy
         B1 = (koniec-tablica)           Liczba elementów tablicy
pętla    SKNLC B1--, dalej               Albo: SKLC B1--,++..dalej – kontrola licznika
           A  = A + [B2]                 Treść pętli – dodanie elementu tablicy do A
           SKLC  B2++,pętla..pętla       Adres następnego elementu tablicy
dalej    [suma] = A
         SKSB  B6,drSumę
;        ...                             Prowadząc sumowanie począwszy od ostatniego elementu
suma     DS    0                         tablicy, wystarczyłby jeden rejestr B do indeksowania
tablica  DS    1 2 3 4 5 6 7 8 9 10      tablicy oraz odliczania liczby jej elementów
koniect  EQU   *
```

TPG=746 – SkS – skok ze śladem (w pamięci)

Rozkaz ten służy do wywoływania podprogramów i ma następującą postać :TPG:

```
:746 podpr nast  BZ
```

Jeśli bit Z jest równy 1, to w trakcie pobrania rozkazu, jeszcze przed jego wykonaniem, następuje B-modyfikacja rejestrem B wskazanym przez triadę B. Załóżmy, że po B-modyfikacji ma on powyższą postać. Działanie rozkazu jest następujące:

- zapisanie w komórce pamięci o adresie **podpr** śladu, słowa `:000 podpr nast  B0`
  – w słowie tym bity operacji TPG oraz bit modyfikacji Z są wyzerowane
- przekazanie sterowania do rozkazu w komórce pamięci o adresie (**podpr+2**)
  – komórka o adresie (**podpr+1**) zostaje „przeskoczona” i może być użyta na dowolne cele

Powrót z podprogramu następuje poprzez wykonanie rozkazu z komórki **podpr**, czyli śladu, który jest w istocie rozkazem wyzerowania sumatora (a przy tym ustawienia warunku Z) i przejścia do instrukcji pod adresem **nast**. Organizacja podprogramów w zastosowaniem rozkazów SKS i SKSB, oraz porównanie zalet i wad obu sposobów przekazywania do nich sterowania, jest opisana dalej.

Instrukcja skoku ze śladem ma następujący format:

etyk	**!**	**SKS**	**podpr**	**..nast**	**;komentarz**	skok do podprogramu z pozostawieniem śladu
ooooooo						

gdzie:

podpr adres podprogramu, argument dowolnego typu: Num, NumX, Adr lub AdrX
Pod tym adresem zapisywany jest ślad, rozkaz `:000 podpr nast  B0`
czyli rozkaz powrotu do następnego rozkazu w programie nadrzędnym (wywołującym)

Przykład:

	`A  = 1`	Argument w A
	`SKS  podpr`	Wywołanie podprogramu
	`A  = 7`	Argument w A
	`SKS  podpr`	Wywołanie podprogramu
	`...`	
`podpr`	`STOP 0..*+2`	Miejsce na ślad `:726 0, *+2, 0`
	`DS  0`	Komórka „przeskakiwana”
	`[archB5] = B5`	Przechowanie używanych rejestrów
`;`	`...`	Zasadnicza treść podprogramu
	`B5 = [archB5]  ..podpr`	Odtworzenie rejestrów i powrót po śladzie
`archB5`	`DS  0`	Miejsce na przechowanie rejestru B5

TPG=032 – SkSB – skok ze śladem w rejestrze B

Jest to osobny mnemonik dla rozkazu `:032` z grupy rozkazów „sumowania”, nadającego się do przekazywania sterowania do podprogramu z pozostawianiem adresu powrotu – śladu – w rejestrze B-modyfikacji. Organizacja podprogramów przy pomocy rozkazów SKS i SKSB jest opisana dalej:

etyk	**!**	**SKSB**	**Bb**	**,**	**podprK**	**..nastN**	**;komentarz**	skok do podprogramu

gdzie:

podprK adres podprogramu, argument dowolnego typu: Num, NumX, Adr lub AdrX
adresy **podprK** i **nastN** instrukcji zamieniają się miejscami w słowie rozkazowym

Przykład:

	`SKSB`	`B6,podprK`	`..nastN`	Jak: `B6 = nastN ..podprK` czyli: `:032 nastN, podprK ,B6`
`podprK`	`NIC`	`0`	`..B6`	Przykładowy powrót z podprogramu

TPG=766 – Nic – Nic nie rób

Jest to jeden spośród wielu niezdefiniowanych rozkazów, które nie wykonują żadnej operacji poza przejściem do następnego rozkazu pod adres **nast** – nawet nie ustawiają warunków. Kod 766 został wybrany arbitralnie na potrzeby wykonywania skoków bezwarunkowych bez zmiany warunków. Rozkaz ten może przekazać sterowanie pod adres zawarty w rejestrze B6 lub B7, w szczególności posłużyć do organizacji podprogramów z adresem powrotu zawartym w rejestrze B6. Argument **n** jest opcjonalny – w przypadku jego braku przyjmowana jest wartość 0:

etyk	**!**	**NIC**	**n**	**..nast**	**;komentarz**	przejście pod adres nast (zwykle podawany jawnie)

TPG=726 – Stop – Stop-Skocz

Rozkaz powoduje zatrzymanie programu, przy czym argument **n** wyświetlany w części AR rejestru rozkazów zwykle służy operatorowi do określenia stanu programu – przyczyny zatrzymania (w przypadku braku argumentu **n** przyjmowana jest wartość 0). Operator może wtedy wykonać odpowiednie czynności i naciśnięciem przycisku [StartCPU] wznowić pracę. Wznowienie pracy następuje poprzez wykonanie rozkazu wskazanego adresem **nast** zawartym w drugiej części adresowej rozkazu, dlatego nazywa się go rozkazem Stop-Skocz. Rozkaz postaci `STOP ..*` nazywa się stopem stabilnym (w odróżnieniu od niestabilnego, z innym adresem **nast**), bo po wznowieniu pracy następuje natychmiast ponowne zatrzymanie.

etyk	**!**	**STOP**	**n**	**..nast**	**;komentarz**	stop stabilny, gdy nast==etyk

TPG=__7 – operacje zmiennoprzecinkowe

Rozkazy tej grupy realizują operacje zmiennoprzecinkowe. Rejestr **AC** zawiera z 39-bitową mantysę w rejestrze akumulatora A, oraz 7-bitową cechę liczby zmiennoprzecinkowej w osobnym rejestrze Ac. Słowo zmiennoprzecinkowe w pamięci lub rejestrze B7 (pozostałe rejestry B-modyfikacji są tu rzadko używane) ma 31-bitową mantysę i 7-bitową cechę na najmłodszych bitach. Operacje pobrania przenoszą cechę do rejestru Ac i uzupełniają 31-bitową mantysę do 39 bitów. Operacje rachunkowe umożliwiają zaokrąglenie 39-bitowej mantysy do 31 bitów. Operacje posyłania skracają mantysę i przepisują na końcowe bity słowa wartość cechy Ac. Wszystkie operacje prócz posyłania mogą dokonywać normalizacji wyniku. Wszystkie operacje mogą zaokrąglać wynik (chociaż przy pobieraniu nie ma to znaczenia).

Rozkazy zmiennoprzecinkowe mają następujący format:

etyk	**!**	**AC=y**	**,**	**BON**	**..nast**	**;komentarz**	pobranie liczby zmp
␣␣␣␣␣␣␣		**y=AC**		**BO**			posłanie liczby zmp
		AC=AC+y		**BN**			dodawanie zmp
		AC=AC-y					odejmowanie zmp
		AC=y-AC					odejmowanie zmp przeciwne
		AC=AC*y					mnożenie zmp
		AC=AC/y					dzielenie zmp

gdzie:

BON opcja: Bez Zaokrąglenia i bez Normalizacji wyniku

BO opcja: Bez Zaokrąglenia wyniku

BN opcja: Bez Normalizacji wyniku

AC rejestr zmiennoprzecinkowy, czyli para rejestrów (Am,Ac), gdzie rolę rejestru mantysy Am pełni akumulator A, zaś Ac to niedostępny bezpośrednio 7-bitowy rejestr cechy liczby zmiennoprzecinkowej

y jeden z następujących argumentów, traktowanych jako słowo zmiennoprzecinkowe:

- [n] zawartość komórki pamięci o adresie zadanym wyrażeniem n dowolnego typu – wyrażenie n już jest w nawiasach kwadratowych, więc nie wymaga ujmowania w nawiasy okrągłe
- Bb rejestr B-modyfikacji: B0, B1, B2, B3, B4, B5, B6 lub B7 – zwykle B7

Przykłady:

`AC  = B7`	Pobranie liczby zmiennoprzecinkowej z B7 do AC
`[89]= AC  ,BON`	Tutaj opcja BON niczego nie zmienia
`AC  = AC+[89] BN`	Przecinek można pominąć
`AC  = AC*B7`	Mnożenie zmiennoprzecinkowe
`AC=AC/B1`	Zawartość B1, uzupełniona zerami, jako liczba zmp

Sporządzanie programu ładowalnego

Przekształcenie tekstu programu napisanego w języku asemblera AsmC na kod maszynowy wymaga napisania tekstu źródłowego i wykonania asemblacji programem asemblera AsmC. W środowisku emulatora Eemc ODRA 1003/1013 czynności te można wykonać nieco łatwiej niż na rzeczywistej maszynie, ale i tak proces ten wymaga cierpliwości i staranności ze względu na brak systemu operacyjnego komputera – wszelkie czynności organizacyjne musi na bieżąco wykonywać operator – człowiek obsługujący maszynę. Jedynymi narzędziami są urządzenia z ich przyciskami, programy zawarte na ścieżce STAŁEJ i niniejszy asembler.

Pisanie tekstu źródłowego, oraz asemblacja programu wymagają przygotowania maszyny, dalekopisu i innych urządzeń. Dalszy opis zakłada następującą instalację sprzętową:

1. Uruchomić emulator Eemc ODRA 1003/1013, po czym:
   - wybrać model maszyny klikając w logo modelu, np. ODRA 1013 UMCS
     – na potrzeby pisania tekstu może to być dowolny model
     – podczas asemblacji model UMCS daje największe możliwości
     – podczas asemblacji model 1013 daje największą wydajność
   - wybrać typ dalekopisu TTY MKD-2 PL4 klikając w tabliczkę z typem
     – tekst źródłowy musi być zakodowany zgodnie z bogatym w znaki polskim alfabetem
     – kodowanie tekstu musi być zgodne z konwencją przełączania czterech pocztów
2. Włączyć maszynę, tj.:
   - włączyć zasilanie instalacji komputerowej przyciskiem [StartB] na pulpicie maszyny
   - włączyć maszynę przyciskiem [Z] na pulpicie maszyny
   - włączyć tryb „Praca Ciągła CPU” przyciskiem [PC] na pulpicie maszyny

Uzyskany kod maszynowy zwykle powinien zawierać program, czyli rozkazy maszynowe, albo same bloki danych (a nawet kod pusty). Ma on postać taśmy perforowanej w kodzie PIĄTKOWYM i może być następnie eksploatowany w standardowy sposób, np. ładowany do pamięci programem STAŁYM, czy włączany do innych programów w języku asemblera AsmC.

Przygotowanie tekstu źródłowego

Tekst źródłowy programu w języku AsmC na wejściu do asemblera musi znajdować się na taśmie perforowanej 5-kanałowej, zakodowany w kodzie ITA, napisany na dalekopisie z zestawem znaków TTY MKD-2 PL4 zawierającym cztery poczty znakowe języka polskiego. Tasiemkę taką można sporządzić na dwa sposoby: pisząc na klawiaturze dalekopisu z jednoczesnym wyprowadzaniem tekstu na taśmę perforowaną, albo posłużyć się wspomaganiem oferowanym przez emulator i skonwertować tekst napisany w środowisku hosta.

Pisanie tekstu źródłowego na klawiaturze dalekopisu

1. Ustawić suwakiem wielką, nawet maksymalną, prędkość pracy emulatora
   – zmniejszy to zwłokę między naciśnięciami poszczególnych klawiszy dalekopisu
2. Włączyć perforator dalekopisu przyciskiem [O]
   – w razie potrzeby usunąć przyciskiem [U] z perforatora wcześniej wyperforowaną taśmę

3. Uruchomić program EDYTOR DALEKOPISOWY znajdujący się w pamięci STAŁEJ pod adresem **`0c17654`**, tj.:
   - ustawić na pulpicie maszyny klawiszami AR (N) rejestru rozkazów adres 0c17654
   - załadować ten adres do rejestru rozkazów przyciskiem [ŁadR]
   - nacisnąć przycisk [StartCPU]
4. Otworzyć klawiaturę dalekopisu klikając podwójnie w rysunek klawiatury
5. Wyprowadzić kody ITA resetujące stan dalekopisu na poczet liter w Trybie Wielkich Liter – oto przykładowy ciąg kodów, ułatwiający też późniejsze pozycjonowanie taśmy w czytniku:
   ●●●▼▼▼▲▼ (tj. NU, NU, NU, LS, LS, LS, FS, LS)
6. Napisać treść programu posługując się klawiaturą dalekopisu, lub klawiaturą komputera PC – instrukcja KONIEC powinna kończyć się przejściem do nowego wiersza
   UWAGA: znaki klawiatury PC nie istniejące na klawiaturze dalekopisu są ignorowane.
7. Wyprowadzić na taśmę przynajmniej dwa dodatkowe znaki (np. spacje, lub kody NL) na wypadek, gdyby inna wersja asemblera inaczej analizowała koniec wiersza
8. Zarchiwizować tekst programu, np. tak:
   - oderwać przyciskiem [U] wydruk tekstu, nazywając go, np. `program.ASM.TXT`
   - oderwać przyciskiem [U] taśmę z tekstem, nazywając ją, np. `program.ASM.PT5`
   - zarchiwizować plik `.ASM.PT5`, gdyż będzie on wejściem do asemblera

Konwersja tekstu źródłowego na kod ITA dalekopisu

Pisanie tekstu, przynajmniej dłuższego, bezpośrednio na dalekopisie jest niewygodne z powodu nieuniknionych pomyłek, których usuwanie jest uciążliwe. O wiele wygodniej jest napisać program przy pomocy edytora zwykłego tekstu, np. Notatnik w systemie Windows, i dać go do automatycznej konwersji na kod ITA przy pomocy programu EDYTOR DALEKOPISOWY na ODRZE. Procedura różni się od powyższej jedynie w punkcie 6:

6. Skonwertować na kod ITA zwykły tekst napisany np. programem Notatnik w systemie Windows, tj.:
   - zamontować przyciskiem [T] na wejściu tekstowym dalekopisu plik tekstowy programu
   - uruchomić przyciskiem [S] wejście dalekopisu z pliku tekstowego
   - po przepisaniu całej treści, przełączyć tym samym przyciskiem [S] wejście dalekopisu z pliku tekstowego na klawiaturę

UWAGI:

- Znak tabulacji w pliku tekstowym jest zamieniany na serię spacji aż do uzyskania wielokrotności 8 znaków w wierszu. Należy pamiętać, że pozycja karetki dalekopisu może pozostać po wcześniejszym użyciu. Przejście do nowego wiersza resetuje pozycję karetki.
- Nieznane dalekopisowi znaki w pliku tekstowym są traktowane jako spacje
- Wydruk na dalekopisie jest zgodny z wyperferowaną taśmą – plik wejściowy niekoniecznie

Asemblacja programu

Tekst źródłowy programu w języku AsmC, zakodowany w kodzie ITA na dalekopisie z zestawem znaków TTY MKD-2 PL4, należy podać na wejście do asemblera AsmC, który utworzy taśmę perforowaną w kodzie PIĄTKOWYM, zawierającą program wynikowy w postaci ładowalnej, czyli rozkazy maszynowe, albo same bloki danych. Możliwa jest też asemblacja programu nie zawierającego żadnego kodu – wtedy taśma z programem wynikowym nie powstaje. Specyfikacja kodu PIĄTKOWEGO znajduje się w dokumentacji programu STAŁEGO [5]. Dokumentacja asemblacji ma postać wydruków z poszczególnych przebiegów asemblera.

Na proces asemblacji można wpływać z zewnątrz programu źródłowego, o ile program źródłowy wykorzystuje symbol standardowy ASMPARM. Asembler nadaje temu symbolowi wartość ustawioną przyciskami części AR (N) akumulatora na pulpicie maszyny (bity nr 9...21).

Podczas asemblacji AsmC wyświetla kody (numery) na części AR rejestru rozkazów, gdy wykonuje rozkazy STOP lub WE podczas montowania wymaganych taśm.

Wymagane czynności to: załadowanie programu asemblera AsmC (translatora) do pamięci operacyjnej, uruchomienie asemblera, montowanie wymaganych taśm źródłowych i/lub z danymi w pierwszym przebiegu asemblacji, ponowne montowanie wymaganych taśm źródłowych i/lub z danymi w drugim przebiegu asemblacji, oderwanie i zarchiwizowanie wydruków i taśmy wynikowej, weryfikacja poprawności asemblacji i asemblowanego programu.

Załadowanie asemblera do pamięci operacyjnej

UWAGA: AsmC zajmuje w czasie pracy całą pamięć operacyjną komputera na swój kod i zmienne, więc inne programy rezydujące w pamięci zostaną najpewniej uszkodzone.

1. Ustawić suwakiem wielką, nawet maksymalną, prędkość pracy emulatora
   – AsmC nie jest w żaden sposób zoptymalizowany pod kątem szybkości działania
2. Ustawić reżim adresowania pamięci
   – AsmC może pracować albo w normalnym, albo w przeplotowym reżimie adresowania
   – przy wielkich prędkościach emulator pracuje tak, jakby cała pamięć była ferrytowa i wtedy reżim adresowania nie ma znaczenia
   – reżim adresowania ustawiony dla AsmC nie ma wpływu na generowany kod wynikowy
3. Wczytać asembler AsmC do pamięci w następujący sposób:
   - ustawić na pulpicie maszyny klawiszami AR (N) rejestru rozkazów adres **`0c17700`**
   - załadować ten adres do rejestru rozkazów przyciskiem [ŁadR]
   - nacisnąć przycisk [StartCPU] uruchamiając program STAŁY WPROWADZAJĄCY
   - zamontować przyciskiem [T] na czytniku PTR0 taśmę „AsmC.pgm.pt5” z asemblerem
   - uruchomić przyciskiem [S] czytnik PTR0 – zacznie się wczytywanie asemblera
   - po wczytaniu zdemontować przyciskami [S] i [Ø] taśmę z asemblerem z czytnika PTR0 – wejrzenie w zakamarki pamięci ukaże zajętość pamięci przez AsmC

Uruchomienie asemblera

Przed rozpoczęciem asemblacji należy zebrać w odpowiednim porządku wszystkie niezbędne taśmy źródłowe i taśmy z danymi. Asembler wspomaga pracę operatora poprzez komunikaty drukowane na dalekopisie i numery montowanych taśm wyświetlane na lampkach rejestru rozkazów.

1. Zadać parametr ASMPARM na potrzeby danej asemblacji, o ile jest to potrzebne
   – sparametryzowanie pozwala generować wymagane warianty programu
   • ustawić na pulpicie maszyny przyciski części AR (N) akumulatora (bity nr 9...21)
2. Wyzerować przyciskiem [ZerR] rejestr rozkazów
   – program asemblera rozpoczyna pracę od adresu 0
3. Nacisnąć przycisk [StartCPU] uruchamiając program asemblera
   – raz załadowany program asemblera może być uruchamiany wielokrotnie
   – po zakończeniu jednej asemblacji przycisk [StartCPU] uruchamia nową asemblację

Przebieg 1 asemblacji – ANALIZA PROGRAMU

AsmC jest asemblerem dwuprzebiegowym. Pierwszy przebieg:

- ✔ sprawdza typ dalekopisu – gdy niewłaściwy, drukuje komunikat „WYMAGANE TTY PL4” i wykonuje STOP STABILNY z kodem 0c17777
- ✔ tworzy symbole standardowe ASMPARM i SYSPARM – wartości tych symboli pozostają niezmienne do końca danej asemblacji
- ✔ analizuje sekwencyjnie program sprawdzając poprawność instrukcji – drukuje treść programu wraz z numeracją wierszy i komunikatami o stwierdzonych błędach
- ✔ sporządza wykaz wszystkich symboli znajdujących się w programie, tj. zapamiętuje nazwy, ustala typy i wartości, a na zakończenie przebiegu drukuje wykaz symboli

Oto przykładowy tekst źródłowy programu (z błędem – z jedną parą nawiasów za dużo):

```
ustaw   [#adres] = -[#adres]
        stop
#adres  equ     ((((((((((((((((((((((((((((((4  /*kontynuacja
*/              ))))))))))))))))))))))))))))))
        koniec
---                                                                    0
```

i jego listing z pierwszego przebiegu:

```
ASMC251 PRZEBIEG 1 - ANALIZA PROGRAMU                                  1
ZAŁÓŻ  00000 NA PTR0  I WYSTARTUJ CPU                                  2
WIERSZ  ---- INSTRUKCJA PROGRAMU ----                                  3
     1. ustaw   [#adres] = -[#adres]                                   4
     2.         stop
     3. #adres  equ     ((((((((((((((((((((((((((((((4  /*kontynuacja  5
     4. */              ))))))))))))))))))))))))))))))                 6
BYK20 3.48-3.49: PROBLEMATYCZNE                                        7
     5.         koniec
TEKST PROGRAMU - ZAWIERA BYKI                                          8
```

0 Wiersz źródłowy z instrukcją KONIEC musi być zakończony przejściem do nowego wiersza, w którym dobrze jest zamieścić przynajmniej dwa dodatkowe znaki (np. spacje, lub kody NL) na wypadek, gdyby inna wersja asemblera inaczej analizowała koniec wiersza

1 Listing zaczyna się tytułem ze wskazaniem wersji asemblera, w postaci `ASMCvrm`, gdzie:
`v` = numer wersji `r` = numer wydania `m` = numer modyfikacji

2 W następnym wierszu znajduje się polecenie zamontowania taśmy perforowanej numer 00000 na czytniku PTR0. Po wydrukowaniu polecenia program zatrzymuje się na rozkazie STOP z numerem taśmy.

Główny tekst programu ma zawsze numer 00000, gdyż nie istnieje żaden sposób wskazania numeru, i jest montowany na urządzeniu PTR0. Polecenia montowania dodatkowych tekstów źródłowych lub taśm z danymi zawierają numery podawane w instrukcjach WSTAW, WDANE i WDANEX, i dotyczą pozostałych urządzeń wejściowych.

Montowanie taśmy polega na założeniu taśmy przyciskiem [T] (kursor znajdzie się na jej początku) i wystartowaniu czytnika przyciskiem [S]. Montowanie jest możliwe tylko przy zatrzymanym czytniku.

Następnie zgodnie z poleceniem należy nacisnąć przycisk [StartCPU], by komputer ruszył dalej. Można też najpierw nacisnąć przycisk [StartCPU], wtedy lampka czytnika zacznie migać na znak, że należy zamontować taśmę. Jeżeli jednak w czytniku już jest zamontowana jakaś inna taśma, to naciśnięcie najpierw przycisku [StartCPU] spowoduje czytanie tej innej taśmy.

3 W trzecim wierszu znajduje się nagłówek wskazujący postać listingu. Pod nagłówkiem WIERSZ prowadzona jest numeracja wierszy programu, a obok treść wierszy (instrukcje, rozkazy).

4 Numer wiersza, dziesiętny, zakończony jest znakiem wskazującym pochodzenie wiersza:
. kropką, gdy jest to wiersz głównego tekstu programu.
+ plusem, gdy jest to wiersz z tekstu WSTAWianego
: dwukropkiem, gdy jest to wiersz z pliku WDANE – tylko w listingu z drugiego przebiegu
! wykrzyknikiem, gdy jest to wiersz z pliku WDANEX – tylko w listingu z drugiego przebiegu

5 Przykład pokazuje instrukcję napisaną w dwóch wierszach – kontynuację umożliwia komentarz blokowy.

6 Definicja symbolu adres przekracza maksymalną dozwoloną liczbę operacji na stosie (liczonych razem z nawiasami). W praktyce trudno ją przekroczyć, ale dla chcącego nic trudnego.

7 Komunikat błędu zawiera numer błędu ułatwiający diagnozę, wskazuje w którym fragmencie wiersza wykryto błąd, oraz hasłowo rodzaj błędu. Komunikat błędu rozpoczyna się słowem BYK, zaś ostrzeżenie przed potencjalnym błędem – słowem HAK.

8 Komunikat kończący listing podsumowuje przebieg analizy programu. Może to być napis:
`TEKST PROGRAMU - OK` – gdy nie stwierdzono żadnych uchybień
`TEKST PROGRAMU - SĄ HAKI` – gdy są jedynie ostrzeżenia, ale nie błędy
`TEKST PROGRAMU - ZAWIERA BYKI` – gdy są błędy

Przebieg 1 asemblacji – WYKAZ SYMBOLI

Niezależnie od poprawności programu, po pierwszym przebiegu drukowany jest wykaz symboli zdefiniowanych w programie:

```
ASMC251 WYKAZ SYMBOLI                                                    11
LP   WIERSZ  NAZWA---    WART(OCT)                                       12
1)         3. #0ADRES     00004                                         13
2)         0. ASMPARM     00000                                         14
3)         0. SYSPARM     14002                                         14
4)         1. USTAW    ? *00000                                         15
KONIEC WYKAZU SYMBOLI                                                    16
GENEROWANIE KODU WYKLUCZONE                                              17
```

11 Listing zaczyna się tytułem ze wskazaniem wersji asemblera (jak w poprzednim wydruku)

12 Następny wiersz to nagłówek dla poszczególnych kolumn wykazu:

- `LP` — numer kolejny symbolu, dziesiętny, dosunięty do lewej, zakończony nawiasem
- `WIERSZ` — numer wiersza w którym symbol zdefiniowano, dosunięty do prawej, zakończony kropką
- `NAZWA---` — *13* nazwa symbolu – nazwa generowana (lokalna) ukazywana jest tu z prefiksem numerycznym wstawionym zaraz za znakiem # (np. `#ŻÓŁĆ` jako `#0ŻÓŁĆ`)
  - *15* `?` znak zapytania za nazwą oznacza, że nazwa nie jest używana (może zbędna)
  - *14* symbole standardowe ASMPARM i SYSPARM nie wymagają definiowania i nigdy nie są zbędne
- `WART(OCT)` *15* wartość liczbowa symbolu, ósemkowo, ze wskazaniem atrybutów:
  np.: `12345`, `12345+B3`, `*12345` lub `*12345+B3`
  - `*` gwiazdka przed liczbą wskazuje typ adresowy
  - `+Bb` plus symbol rejestru za liczbą wskazuje typ indeksowany rejestrem Bb
  - `Id` napis oznaczający identyfikator instrukcji (bez wartości)
  - `----- Byk46` napis oznaczający symbol niezdefiniowany

16 Oznaczenie końca listingu

17 Komunikat, który pojawia się tylko wtedy, gdy oznajmia przerwanie asemblacji ze względu na stwierdzone błędy (BYKi). Gdy nie ma błędów, translator przechodzi do drugiego przebiegu.

Przebieg 2 asemblacji – GENEROWANIE KODU

Jeśli w pierwszym przebiegu nie wykryto żadnych błędów, to asembler przechodzi do drugiego przebiegu. Drugi przebieg:

- ✔ ponownie analizuje sekwencyjnie program sprawdzając poprawność instrukcji – w tym celu nakazuje ponownie montować taśmy z tekstami źródłowymi i danymi dokładnie w ten sam sposób i w tej samej kolejności jak w pierwszym przebiegu. Niektóre błędy wychwytuje dopiero teraz.
- ✔ generuje kod programu na taśmie PIĄTKOWEJ i drukuje go w postaci pseudo :TPG – ósemkowo, ze wskazaniem numerów wierszy źródłowych, adresów pamięci, etykiet i z instrukcjami asemblera. Wartości pól adresowych rozkazów podaje ze wskazaniem atrybutów tak jak w wykazie symboli. Oznacza rozkazy, do których odnoszą się ostrzeżenia (HAKi).
- ✔ Końcowymi komunikatami potwierdza wykonanie drugiego przebiegu asemblacji.

Oto przykładowy listing z drugiego przebiegu po usunięciu błędu w programie źródłowym (tj. uproszczeniu wyrażenia w wierszach 3-4 poprzez usunięcie jednej pary zbędnych nawiasów):

```
ASMC251 PRZEBIEG 2 - GENEROWANIE KODU                                    21
ZAŁÓŻ  00000 NA PTR0  I WYSTARTUJ CPU                                    22
WIERSZ   ADRES ETYK---- INSTRUKCJA / KOD WYNIKOWY ---                    23
     1.  00000 USTAW      :111  00004    *00001    00 PAO                24
     2.  00001            :726  00000    *00002    00
     4.  00002 #0ADRES  EQU     00004                                     25
     5.  00002          KONIEC  00000
KOD PROGRAMU - OK                                                        26
KOD WYGENEROWANY                                                         27
```

21 Listing zaczyna się tytułem ze wskazaniem wersji asemblera (jak w poprzednich wydrukach)

22 W następnym wierszu znajduje się polecenie zamontowania taśmy perforowanej numer 00000 na czytniku PTR0. Montowanie taśm będzie się odbywało po raz drugi, dokładnie tak samo jak w pierwszym przebiegu. Pozycjonowanie taśm powinno odpowiadać temu z pierwszego przebiegu, by zgodne były numeracja wierszy i wszystkie instrukcje.

23 Następny wiersz to nagłówek dla poszczególnych kolumn listingu:

`WIERSZ` numer wiersza jak w listingu z pierwszego przebiegu, dosunięty do prawej, dziesiętny, zakończony znakiem wskazującym pochodzenie wiersza

`ADRES` adres w pamięci, ósemkowo, pod którym kod został wygenerowany.
Adres jest poprzedzony znakiem:

- ◡ spacji — gdy jest to adres w zwykłej pamięci, w BLOKu
- / przekreślenia — gdy jest to adres w zwykłej pamięci, w QBLOKu
- = równości — gdy jest to adres po przeadresowaniu NAF-KOF w BLOKu
- - minusa — gdy jest to adres po przeadresowaniu NAF-KOF w QBLOKu

`ETYK----` etykieta rozkazu lub instrukcji

`INSTRUKCJA / KOD WYNIKOWY ---` *25* instrukcja ze swoim parametrem, albo *24* słowo maszynowe reprezentujące rozkaz lub daną
Parametry i słowa wydrukowane są ósemkowo.
Wartości części adresowych słowa są podane wraz ze swoimi atrybutami, np.: `12345`, `12345+B3`, `*12345` lub `*12345+B3` – jak w wykazie symboli
Napis `PAO` za kodem słowa ostrzega, że nie będzie zapisu do pamięci.
Napis `BON` za kodem słowa ostrzega, że nie będzie zaokrąglenia i normalizacji.
Tekstowy parametr instrukcji NOTA drukowany jest pod instrukcją.
Dla tablicy słów drukowany jest tylko pierwszy element tablicy, a pod nim krotność w postaci komunikatu, jak np.: `-- sztuk [8]`

26 Oznaczenie końca listingu, ze wskazaniem poprawności programu. Może to być napis:

`KOD PROGRAMU - OK` – gdy w drugim przebiegu nie stwierdzono uchybień
`KOD PROGRAMU - SĄ HAKI` – gdy są jedynie ostrzeżenia, ale nie błędy
`KOD PROGRAMU - ZAWIERA BYKI` – gdy w drugim przebiegu stwierdzono błędy

27 Komunikat informujący o wyniku asemblacji. Może to być napis:

`KOD PUSTY` – gdy nie utworzono taśmy wynikowej, bo brak kodu maszynowego
`KOD WYGENEROWANY` – gdy utworzono taśmę z kodem – kod wynikowy powstaje nawet gdy wykryto błędy i wtedy jest on potencjalnie niepoprawny.

Użytkowanie programu

W wyniku asemblacji programu w języku AsmC otrzymujemy listingi ułatwiające ponowną analizę poprawności algorytmów i usunięcie ew. błędów. Listingi mogą być jedynym oczekiwanym wynikiem, jeśli program nie zawiera instrukcji generujących kod maszynowy. Podstawowym wynikiem jest jednak taśma w kodzie PIĄTKOWYM, zawierająca wygenerowany program. Jest ona zgodna z formatem opisanym w [5].

Konwersja taśmy w kodzie PIĄTKOWYM na taśmę w kodzie THETA albo odwrotnie

Wygenerowany kod może być samodzielnym programem, podprogramem używanym przez inne programy, czy danymi w postaci słów maszynowych. Kod ten może być ładowany do pamięci przy pomocy programu STAŁEGO WPROWADZAJĄCEGO, albo wstawiany do innego programu – w języku AsmC robi się to przy pomocy instrukcji WDANE (lub WDANEX po uprzedniej zamianie na kod THETA). Konwersji taśmy PIĄTKOWEJ na THETA lub odwrotnie można dokonać programami:

- „PIĄTKOWY na THETA.pgm.pt5”
- „THETA na PIĄTKOWY.pgm.pt5”

które są przykładowymi wariantami, jakie, zadając odpowiednie opcje asemblacji, można wygenerować na podstawie tekstu źródłowego:

- „PIĄTKOWY-THETA.asm.pt5” – czyli „PIĄTKOWY-THETA.asm.txt”

Reżim adresowania programu wygenerowanego asemblerem AsmC

Asembler AsmC nie optymalizuje programu pod kątem wydajności poprzez rozmieszczenie rozkazów w pamięci z przeskokami uwzględniającymi ich czas wykonywania się. Również „ręczna” optymalizacja rozmieszczenia rozkazów przez programistę całkowicie zniszczyłaby przejrzystość programu i zaprzeczyła elementarnym zasadom programowania strukturalnego, które i tak trudno zachować w asemblerze.

Asembler AsmC może zostać wczytany i pracować w normalnym albo w przeplotowym trybie adresowania. Program przezeń generowany też można wczytać i wykonać w normalnym trybie adresowania, albo wczytać i wykonać w trybie przeplotowym. Stosowanie bardzo skutecznego, automatycznego reżimu adresowania przeplotowego, i/lub umiejętne użytkowanie pamięci ferrytowej, może znacząco przyspieszyć pracę programu. Dość powiedzieć, że czas asemblacji programu AsmC przy pomocy asemblera AsmC pracującego w trybie adresowania przeplotowego skraca się ponad trzykrotnie w stosunku do adresowania normalnego.

Program wczytany w trybie normalnym musi być wykonywany też w trybie normalnym, zaś wczytany w trybie przeplotowym musi być wykonywany w trybie przeplotowym. Jedynie programy STAŁE (patrz [5]) są odpowiednio zdublowane tak, by można je było wykonywać w dowolnym trybie adresowania.

Emulator emc ODRA 1003/1013 może pracować z różną prędkością, co usuwa problem optymalizacji pracy programu i pozwala skupić się na pozostałych cechach ODRY.

Testowanie niewielkich programów bez niszczenia rezydującego w pamięci programu AsmC

Asembler AsmC zawłaszcza całą pamięć operacyjną komputera na swój kod i swoje zmienne. Szczególnie wielki obszar pamięci przeznaczony jest na wykaz symboli asemblowanego programu, dzięki czemu możliwa jest asemblacja dużych programów. W praktyce, przy asemblacji mniejszych programów, z niewielką liczbą symboli, pozostają dwa duże obszary równej wielkości, leżące odłogiem na końcach dwóch tablic położonych jedna bezpośrednio za drugą. Drugi obszar kończy się na słowie pod adresem 0c16577. Wejrzenie w zakamarki pamięci po wykonanej asemblacji pozwala określić jego wielkość.

Leżące odłogiem obszary można wykorzystać np. do naprzemiennego asemblowania i testowania niewielkiego programu, bez konieczności ponownego ładowania asemblera, a dopiero po uzyskaniu poprawnej wersji ustalenia docelowych lokalizacji bloków kodu. Podobnie można pomiędzy jedną a drugą asemblacją wykonywać programy STAŁE (oczywiście bez wczytywania kodu w pamięć zajętą przez asembler).

UWAGA: Opisana tu możliwość testowania (wykonywania) programu w obszarach pamięci zawłaszczonych przez AsmC, ale leżących odłogiem w danej asemblacji jest specyficzna dla tej wersji asemblera i może się zmienić wraz z nowym wydaniem.

Komunikaty asemblera

Typowe komunikaty objaśniające wydruki są pokazane w rozdziale „Asemblacja programu”. Inne, wymagające reakcji operatora: instruujące, ostrzegawcze i błędów są objaśnione w poniższych wykazach. Istnieją jednak sytuacje, w których nie pojawi się żadne powiadomienie o błędzie i konieczne jest zorientowanie się co do przyczyny po innych objawach. I tak:

OCZEKIWANIE URZĄDZENIA NA ZAŁOŻENIE TAŚMY

W pewnych sytuacjach, spowodowanych zapewne jakimiś błędami, czytnik taśmy perforowanej może sygnalizować miganiem lampki oczekiwanie na założenie taśmy. Dodatkowo na lampkach części AR rejestru rozkazów wyświetli się numer taśmy podany w instrukcji WSTAW, WDANE lub WDANEX.

Jeśli czytany ma być tekst źródłowy, to prawdopodobnie brak jest instrukcji KONIEC, bo np.:

- wiersz z instrukcją KONIEC nie jest zakończony przejściem do nowego wiersza (z ew. kilkoma zbędnymi, buforowanymi znakami)
- instrukcja KONIEC znalazła się w komentarzu liniowym po zabłąkanym średniku
- instrukcja KONIEC znalazła się w niezamkniętym komentarzu blokowym lub tekście
- instrukcja KONIEC znalazła się w pominiętej klauzuli AIF-AELIF-AELSE-AFI
- bieżąca taśma istotnie zawiera początek tekstu źródłowego, a jego koniec znajduje się na następnej taśmie i należy zamontować następną taśmę z dalszym ciągiem tekstu

Jeśli czytane mają być dane, to prawdopodobnie brak prawidłowego zakończenia taśmy sumą kontrolną lub odpowiednią liczbą znaków NU, albo to w ogóle nie jest taśma z kodem PIĄTKOWYM lub THETA. Taśmy z danymi raczej nie dzieli się na części, ale jest to możliwe, a wtedy należy zamontować taśmę z następną częścią.

NIEZGODNA NUMERACJA WIERSZY

Niedokładność powtórnego montowania taśmy może sfałszować numerację linii i zaburzyć działanie translatora. Dla zminimalizowania kłopotów zaleca się na początku taśmy pozostawić pewien luz w postaci kilku pustych znaków NU. Proponowany początek taśmy pokazany jest w rozdziale „Przygotowanie tekstu źródłowego”.

NIEWIARYGODNE WIELKOŚCI

Program można „położyć” przez niewiarygodne wielkości: np. ponad 1000-cyfrową liczbę, ponad 100000 wierszy źródłowych, lub ponad 8000 znaków w wierszu. AsmC nie narzuca i nie kontroluje pewnych ograniczeń, zakładając rozsądek.

NIE WIDAĆ BŁĘDU, KTÓRY PONOĆ GDZIEŚ JEST

Jeśli brak komunikatu BYKnn, a jest informacja "PROGRAM ZAWIERA BYKI", to może błąd jest w instrukcji pomijanej warunkowo. Pomijane instrukcje są częściowo analizowane, by stwierdzić czy nie są to instrukcje AIF, AELIF, AELSE lub AFI.

Komunikaty organizacyjne

`WYMAGANE TTY PL4`

Na początku każdej asemblacji, przed pierwszym przebiegiem, AsmC sprawdza typ dalekopisu. Po wydrukowaniu wykonuje STOP STABILNY z kodem 0c17777 wyświetlanym na lampkach części AR rejestru rozkazów.

Język AsmC wymaga bogatego zestawu znaków języka polskiego, zapewnianego przez dalekopis TTY MKD-2 PL4. Zmiana dalekopisu na żądany możliwa jest jedynie przy wyłączonym zasilaniu. Należy:

- wyłączyć zasilanie przyciskiem [STOP B] na pulpicie maszyny – zgaśnie lampka „Napięcia niestabilizowane” (i wszystkie inne lampki)
- klikać w plakietkę z symbolem dalekopisu dotąd, aż typ dalekopisu zmieni się na „`TTY MKD-2 PL4`”
- włączyć zasilanie i maszynę, na nowo wczytać translator asemblera AsmC i ponowić jego start

`ZAŁÓŻ: numer NA PTRnn I WYSTARTUJ CPU`

Komunikat instruuje operatora o konieczności zamontowania taśmy na czytniku, i jest omówiony w opisach instrukcji WSTAW, WDANE i WDANEX.

`numer` to wartość argumentu instrukcji WSTAW, WDANE lub WDANEX drukowana ósemkowo. Główny tekst źródłowy programu zawsze ma numer 00000 i jest montowany na czytniku PTR0.

`PTRnn` to oznaczenie czytnika: `PTR0`, `PTR2`, `PTR0'` lub `PTR2'`

Asembler zatrzymuje się na rozkazie STOP numer, zatem `numer` wyświetla się również na lampkach części AR rejestru rozkazów.

Montowanie taśmy w czytniku możliwe jest tylko przy zatrzymanym czytniku – jeśli świeci się lampka kontrolna czytnika, to należy zastopować czytnik przyciskiem [S], oraz zdjąć zamontowaną wcześniej taśmę przyciskiem [Ø].

Należy zamontować żądaną taśmę przyciskiem [T], upewnić się, że kursor taśmy znajduje się na jej początku i wystartować urządzenie przyciskiem [S] – urządzenie jest gotowe do czytania taśmy.

Na koniec należy wystartować CPU przyciskiem [StartCPU] na pulpicie maszyny – program asemblera ruszy dalej.

UWAGA: Można najpierw nacisnąć [StartCPU], a potem montować taśmę, ale należy mieć pewność, że nie będzie czytana taśma wcześniej znajdująca się w czytniku.

Komunikaty ostrzegawcze

AsmC ostrzega przed pułapkami tkwiącymi w niektórych rozkazach, oraz wydaje komunikaty będące adnotacjami wysyłanymi z programu źródłowego instrukcją NOTA. Komunikaty te, ostrzegawcze, nie są błędami, i jako takie mają na początku oznaczenie HAK. Ich ogólny format jest następujący:

```
HAKcc odLin.odKol-doLin.doKol: ostrzeżenie
```

gdzie:

`cc`	numer ostrzeżenia – aktualnie wszystkie mają numer 00
`odLin.odKol`	numery wiersza i kolumny orientacyjnego początku komentowanej treści
`doLin.doKol`	numery wiersza i kolumny orientacyjnego końca komentowanej treści
`ostrzeżenie`	krótkie, hasłowe wskazanie ostrzeżenia

Oto wykaz komunikatów ostrzegawczych:

`HAK00 ...: ADNOTACJA: PAO`

Rozkaz pobiera argument z pamięci i ma zapisać wynik z powrotem w pamięci, ale w istocie nie zapisuje wyniku w pamięci, jak by na to wskazywała postać instrukcji. Asembler ostrzega przed potencjalnym i trudnym do wykrycia błędem. Prócz komunikatu rozkaz ten zostanie oznaczony słowem PAO w listingu z drugiego przebiegu.

`HAK00 ...: ADNOTACJA: BON`

Rozkaz zmiennoprzecinkowy ma wykonać zaokrąglenie logiczne i/lub normalizację liczby, ale tego nie robi – działa jakby miał wskazaną opcję BON. Ten komunikat w istocie nie ukazuje się, gdyż uchybienie wynika ze zgody na skrótowy zapis pewnych rozkazów bez podawania opcji BON, ale asembler oznacza taki rozkaz słowem BON w listingu z drugiego przebiegu.

`HAK00 ...: ADNOTACJA: NOTA`

Komunikat pojawia się wskutek zasemblowania instrukcji NOTA. W listingu z drugiego przebiegu pojawia się instrukcja NOTA z numerem, który jest jej pierwszym argumentem. Pod nią znajduje się treść notatki, która jest drugim argumentem. Instrukcja NOTA służy do wydawania informacji przez sam program źródłowy i zwykle będzie znajdować się wewnątrz klauzuli warunkowej instrukcji AIF-AELIF-AELSE-AFI.

Komunikaty błędów

AsmC wykrywa wiele błędów, ale dla danej instrukcji/rozkazu wydaje komunikat tylko o jednym, najwcześniej wykrytym błędzie. Zatem po usunięciu wskazanego błędu może pojawić się następny komunikat błędu. Również sam program może instrukcją BYK wydać komunikat błędu. Wykrycie w pierwszym przebiegu choćby jednego błędu powoduje wykluczenie drugiego przebiegu – generacji kodu. Komunikaty błędu mają na początku oznaczenie BYK. Ich ogólny format jest następujący:

```
BYKcc odLin.odKol-doLin.doKol: komunikatBłędu
```

gdzie:

`cc`	numer błędu – błędy ponumerowane są ósemkowo
`odLin.odKol`	numery wiersza i kolumny orientacyjnego początku błędnej treści
`doLin.doKol`	numery wiersza i kolumny orientacyjnego końca błędnej treści
`komunikat`	krótkie, hasłowe wskazanie błędu

Oto wykaz komunikatów błędu:

`BYK01 ...: CO ZA NAZWA`

Za długa nazwa symbolu. Nazwa zaczyna się od znaku literowego i składa się z cyfr i liter: 0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z Ą Ć Ę Ł Ń Ó Ś Ź Ż – kolejność leksykograficzna nazw jest zgodna z kolejnością wymienionych tu znaków. Asembler w nazwach nie rozróżnia liter wielkich i małych.

Nazwa zwykła (globalna) składa się co najwyżej z 7 znaków, nazwa generowana (lokalna) składa się co najwyżej z 6 znaków i jest poprzedzona znakiem #.

`BYK02 ...: CO ZA LICZBA`

Denotacja liczby jest błędna (np. niedozwolonymi cyframi) lub liczba jest niewłaściwa. Liczba w wyrażeniu adresowym musi być z zakresu 0...8191, bez znaku, skali i cechy.

W słowie wielkość wartości zależy od jej typu:

Liczba całkowita ze znakiem + lub - musi być z zakresu od -274877906944 do +274877906943, a bez znaku – z zakresu 0...549755813887, tj. liczba całkowita ma mieścić się na 39 bitach zakładając skalę 38 – w mniejszej skali zakres jest odpowiednio mniejszy. Na wielkość liczby w notacji znakowej mogą wpłynąć znaki niewidoczne na wydruku, szczególnie gdy zastosowano notację tekstem topornym lub ciosanym.

Skala liczby całkowitej lub członu liczby wieloadresowej musi być z zakresu 0...38.

Notacja wykładnika liczby zmiennoprzecinkowej musi być zgodna z podstawą liczbową mantysy: E12 gdy mantysa podana dziesiętnie, P12 gdy mantysa podana bitowo. Po znormalizowaniu liczby cecha musi być z zakresu -64...+63. Błąd jest wydawany zarówno dla nadmiarowej jak i podmiarowej liczby zmiennoprzecinkowej.

BYK03 ...: CO ZA TEKST

Błędnie zapisany tekst – typ tekstu jest nieprawidłowy, albo brak tekstu w stałej tekstowej.

Może też tekst jest za długi. Maksymalna długość tekstu jest ograniczona do 254 słów pamięci (plus jedno wyzerowane słowo kończące tekst), co daje maksymalnie 1778 kodów 5-bitowych. Przekroczenie tej wielkości może wynikać z niewidocznych na wydruku znaków, szczególnie w tekście topornym lub ciosanym. Zaradzić potrzebie długiego tekstu można dzieląc go na krótsze, łączone znakiem konkatenacji (.).

BYK04 ...: CO ZA /ZNAK

Błędny znak specjalny w tekście dłubanym, kodowany za pomocą znaku ucieczki (/).

Dozwolone kodowania to: /L, /F, /R, /N, /Z, // odpowiednio dla LS, FS, CR, LF, NU, / ponadto kodowanie /Ccc, /Ddd, /Xxx dowolnych znaków liczbami dwucyfrowymi: cc (ósemkowymi), dd (dziesiętnymi), xx (szesnastkowymi).

BYK05 ...: LICZBA AREALNA

Liczba zmiennoprzecinkowa jest za dokładna, tj. jej mantysa nie mieści się na 32 bitach.

Mantysa liczby zmiennoprzecinkowej dziesiętnej nie musi być dokładna i zwykle jest zaokrąglana by zmieścić się w słowie, natomiast liczby nie dziesiętnej (zapisanej bitowo – binarnie, ósemkowo lub szesnastkowo), musi być dokładna, tj. żaden bit mantysy nie może zostać odcięty czy zaokrąglony (wszystkie bity mantysy muszą zmieścić się w części słowa przeznaczonej na mantysę). Istotą notacji bitowej jest umożliwienie precyzyjnej kontroli nad reprezentacją liczby zmiennoprzecinkowej, z tym wyjątkiem że i tak zostanie znormalizowana – postać nieznormalizowaną należy uzyskać w inny sposób.

BYK06 ...: NIEETYKIETA

Na początku wiersza znajduje się słowo kluczowe instrukcji, rozkazu lub opcji, oznaczenie rejestru, albo operator lub separator.

Początek wiersza jest zarezerwowany na etykietę instrukcji, czyli na definiowalną nazwę, o czym przypomina ten komunikat. Należy albo zmienić tę nazwę, albo przed słowem zastrzeżonym wstawić odstęp.

BYK07 ...: ZNOWU TA NAZWA

Nazwa ta już została zdefiniowana, tj. znajduje się gdzieś wcześniej na początku wiersza.

Pamiętajmy, że symbol może być zdefiniowany tylko jeden raz, symbole ASMPARM i SYSPARM są z góry zdefiniowane przez asembler, a asembler w nazwach nie rozróżnia liter wielkich i małych. Tylko identyfikatory niektórych instrukcji mogą ponownie znaleźć się w polu etykiety, tj. na początku wiersza.

Błąd ten jest sygnalizowany dopiero po wczytaniu atomu następnego po etykiecie.

`BYK10 ...: ZRAZU ZAKAZANE`

Niedozwolone użycie symbolu w instrukcji EQU, BLOK lub QBLOK.

Symbolu definiowanego w polu etykiety nie można użyć w instrukcji która go definiuje, tj. nie może definiować sam siebie.

`BYK11 ...: JENO PROSTA L`

Wielkości takie jak krotność stałej w instrukcji DS, atrybut długości S tekstu, numer identyfikacyjny wstawianego tekstu lub danych w instrukcjach WSTAW, WDANE i WDANEX, numer urządzenia w rozkazach WE i WY, numer ścieżki ferrytowej w rozkazach BF i FB, albo numer adnotacji lub restrykcji w instrukcjach NOTA i BYK muszą być prostymi liczbami – wyrażeniami typu Num obliczalnymi w pierwszym przebiegu translatora. Niedozwolona jest wielkość indeksowana lub typu adresowego.

`BYK12 ...: JENO NIE ADRES`

Wielkość przesunięcia w rozkazach przesunięć LL, LP, AL, AP, CP, ALD i APD, lub wskazanie liczby bitów ilorazu w rozkazie dzielenia długiego DZD nie może być typu adresowego. Argument taki musi być liczbą typu Num lub NumX.

`BYK13 ...: JENO NIE INDEX`

Argumentu instrukcji BLOK, WDANE, WDANEX, NAF, KONIEC i wartości składnika słowa liczbowego wieloadresowego nie można indeksować rejestrem B, bowiem musi wyznaczać konkretny adres lub liczbę, by możliwa była asemblacja.

`BYK14 ...: CUDA WE WZORZE`

Błędne wyrażenie arytmetyczne.

Niezrozumiały napis zamiast spodziewanego operandu operacji w wyrażeniu adresowym.

`BYK15 ...: NIE TAKIE TYPY`

Jeden lub oba operandy operacji w wyrażeniu adresowym ma typ niedopuszczalny w tej operacji. W rozdziale „Wyrażenia adresowe” opisana jest budowa wyrażeń adresowych, priorytety operacji i dopuszczalne typy argumentów.

`BYK16 ...: NIE PRZEZ ZERO`

W operacji obliczania ilorazu lub reszty z dzielenia w wyrażeniu adresowym, dzielnik ma wartość 0.

Błąd ten jest wykrywany dopiero w drugim przebiegu asemblera, gdyż w pierwszym nie wszystkie symbole mają już wartość.

`BYK17 ...: NIEOBLICZALNE`

Wyrażenia nie da się obliczyć.

Argument instrukcji BLOK, QBLOK, WSTAW, NAF, EQU, AIF i AELIF, krotność stałej w instrukcji DS, atrybut długości S tekstu, numer identyfikacyjny i argumenty instrukcji WSTAW, WDANE i WDANEX. numer urządzenia w rozkazach WE i WY, numer ścieżki ferrytowej w rozkazach BF i FB, albo numer adnotacji lub restrykcji w instrukcjach NOTA i BYK muszą dać się obliczyć w pierwszym przebiegu translatora. W pozostałych wyrażeniach błąd ten jest wykrywany w drugim przebiegu.

`BYK20 ...: PROBLEMATYCZNE`

Zbyt złożone wyrażenie adresowe. Kolejny operator lub argument nie mieści się na stosie kalkulatora wyrażeń adresowych.

Wzór należy spłaszczyć tak, by było mniej poziomów zagnieżdżeń wyrażeń częściowych, wynikających z priorytetów operacji i nawiasów. Maksymalne zagnieżdżenie można stwierdzić na pokazanym wcześniej przykładzie z za wielką liczbą nawiasów.

Należy spróbować przeorganizować wyrażenie tak, by operacje nie oczekiwały zbyt długo na stosie na wykonanie. Np. wyrażenie `x+y*z` wymaga miejsca na stosie na przechowanie dodawania i mnożenia, zaś `y*z+x` – najpierw tylko mnożenia, a potem tylko dodawania. Może też da się usunąć zbędne nawiasy. Innym sposobem jest rozbicie wyrażenia na kilka podwyrażeń obliczanych w osobnych instrukcjach EQU.

`BYK21 ...: ZAMKNIJ NAWIAS`

Brak nawiasu okrągłego) w wyrażeniu adresowym, lub atrybucie długości tekstu S(), albo nawiasu kwadratowego] w krotności stałej w instrukcji DS, lub w odwołaniu do komórki pamięci w rozkazie maszynowym.

`BYK22 ...: NIEODGADNIONE`

Niezrozumiały rozkaz.

Może to być brak wymaganego przecinka lub argumentu tekstowego w instrukcji BYK lub NOTA, brak spodziewanego separatora lub rejestru w rozkazach maszynowych, brak separatora po definicji słowa w instrukcji DS, brak bezpośrednio po znaku `+` łączącym człony słowa wieloadresowego w instrukcji DS nawiasu `(` otwierającego kolejny człon słowa, wykrzyknik korekcji przed instrukcją nie będącą rozkazem, lub tp.

`BYK23 ...: CO ZA OPER TPG`

Błędna operacja rozkazu zapisanego w postaci :TPG - kod operacji musi składać się dokładnie z trzech cyfr ósemkowych, nierozdzielonych separatorami, znajdować się bezpośrednio za dwukropkiem i być zakończony separatorem (odstępem).

BYK24 ...: TOTO NIE REJ B

W rozkazach :TPG, BF, FB, SKBG i SKLC jednym ze spodziewanych argumentów jest rejestr B-modyfikacji, a takiego nie wskazano. Należy wskazać odpowiedni rejestr.

BYK25 ...: BRAK INKR DEKR

W rozkazie SKLC jednym ze spodziewanych argumentów (operatorów rozkazu) jest symbol ++ lub -- wskazujący operację inkrementacji lub dekrementacji zawartości rejestru B-modyfikacji, stanowiącą zasadniczą czynność rozkazu. Należy podać odpowiedni symbol.

BYK26 ...: JENO NIE TEN B

Rejestr B-modyfikacji będący składnikiem wyrażenia adresowego, jest niedozwolony w wyrażeniu stanowiącym wartość danej części adresowej rozkazu.

Zawartość tego rejestru uczestniczy w wyrażeniu poprzez mechanizm B-modyfikacji. Rejestr B6 nie może modyfikować części AR (N) rozkazu, zaś modyfikacji części NR (K) można dokonać tylko rejestrami B6, B7 i B0.

Konfuzja może wynika z użycia zanegowanego warunku w rozkazie skoku warunkowego, skoku przy braku gotowości lub skoku z licznikiem cykli, gdyż zanegowany warunek skoku uzyskuje się poprzez zamianę miejscami adresów N i K w rozkazie – adres skoku trafia wtedy do części NR, a adres następnego rozkazu do części AR rozkazu. Tym samym adres skoku z zanegowanym warunkiem może być modyfikowany tylko rejestrami B6, B7 i B0, a adres następnego rozkazu nie może być modyfikowany rejestrem B6.

BYK27 ...: MISZMASZ REJ B

Mieszanka B-rejestrów (kilku różnych rejestrów) w jednym rozkazie jest niedozwolona niezależnie od tego, w jakiej roli rejestry występują.

Budowa rozkazu pozwala wskazać w rozkazie tylko jeden rejestr, który wszakże może posłużyć zarówno do B-modyfikacji rozkazu, jak i jako argument na etapie wykonania.

Jedyny wyjątek dotyczy niejawnego udziału rejestru B7 w rozkazach mnożenia i dzielenia, w których można użyć jawnie też innego rejestru. Rejestr B7 nie jest w nich wskazywany, lecz uczestniczy w operacji. Instrukcje maszynowe tych rozkazów w języku AsmC odwołują się doń ze względów formalnych poprzez oznaczenie M.

Rejestr B-modyfikacji może też wchodzić w skład wartości symbolu użytego w instrukcji maszynowej i stąd, jako bezpośrednio niewidoczny, kolidować z innym rejestrem.

BYK30 ...: LEWY NIE PRAWY

Adres komórki pamięci po lewej stronie przypisania jest niezgodny z argumentem znajdującym się po prawej stronie.

W rozkazach „sumowania” adres komórki pamięci po lewej stronie przypisania musi być zgodny z argumentem N znajdującym się po prawej stronie, bowiem obie te wielkości są wartością jednego, wspólnego pola AR (N) rozkazu – z wyjątkiem argumentu 0, który jest asemblowany poprzez osobny kod operacji, a nie jako argument bezpośredni w części AR (N) rozkazu. Argument bezpośredni N może jednak służyć obliczeniu wartości N, -N, |N|, A-N, A+N, N-A, A&N lub A^N wstawianej do komórki [N], co pozwala jednym rozkazem nadać komórce pamięci pewną wartość obliczoną w oparciu o jej adres.

BYK31 ...: ZBYT WIELE ARG

Zbędny argument instrukcji.

Asembler kontroluje liczbę argumentów, by przez niedopatrzenie żaden nie został pominięty – być może zbędny argument powinien być elementem wyrażenia.

BYK32 ...: DOSKOCZ JAWNIE

W poprzednim rozkazie domyślny adres następnego rozkazu jest adresem następnej komórki pamięci, ale następny rozkaz nie znajduje się pod tym domyślnym adresem. Asembler domaga się, by w poprzednim rozkazie adres następnego rozkazu nie był domyślny, lecz podany jawnie.

Niezgodność jest wykrywana podczas asemblacji następnego rozkazu maszynowego, zwykle po rozpoczęciu instrukcją BLOK nowego bloku pod innym adresem niż bieżący, po instrukcji QBLOK uniemożliwiającej określenie co jest w następnej komórce pamięci, po instrukcji NAF przeadresowującej następne adresy, po instrukcji WDANE lub WDANEX wstawiającej kod pod adres inny niż bieżący. Błąd może też dotyczyć rozkazu znajdującego się w QBLOKu bazowanym nie na adresie.

Komunikat ma wykluczyć przekazanie sterowania pod niewłaściwy adres, pod którym nie ma następnego rozkazu. W poprzednim rozkazie należy po separatorze `..` jawnie podać adres następnego rozkazu. Jeżeli diagnoza asemblera jest nieprawidłowa (tj. program jest prawidłowy, ale asembler nie jest w stanie tego stwierdzić), to wystarczy podać adres w postaci argumentu `..++` by formalnie spełnić żądanie asemblera.

Komunikat ten jest wydawany tylko w drugim przebiegu.

BYK33 ...: NIEZGODNY FELC

Identyfikatory instrukcji złożonej NAF-KOF, lub AIF-AELIF-AELSE-AFI są z sobą niezgodne.

Identyfikator instrukcji KOF (jeśli podany) musi być zgodny z identyfikatorem odpowiedniej instrukcji NAF, zaś identyfikatory instrukcji AELIF, AELSE i AFI (jeśli podawane) muszą być zgodne z identyfikatorem odpowiedniej instrukcji AIF – tak jak w stolarce zgodne muszą być wycięcia łączonych z sobą elementów. Identyfikatory służą do kojarzenia w całość instrukcji tworzących wspólnie jedną instrukcję złożoną, więc błąd może wynikać ze zdekompletowania instrukcji złożonej.

BYK34 ...: WSZAK BRAK NAF

Jest instrukcja KOF, ale brak uprzedniej instrukcji NAF.

Instrukcja złożona NAF-KOF jest zdekompletowana, być może wskutek wcześniejszych błędów w programie.

BYK35 ...: DAJ KOF ZA TRK

Bieżący rozkaz znajduje się na następnej ścieżce niż poprzedni, a segment rozpoczęty instrukcją NAF nie został zakończony instrukcją KOF.

Przeadresowywany instrukcjami NAF-KOF segment musi w całości znajdować się na jednej ścieżce pamięci. Jeśli przeadresowywana ma być też następna ścieżka, to wymagane jest wskazanie nowego segmentu przy pomocy nowej pary instrukcji NAF-KOF. Asembler nie pozwala na wskazywanie segmentów rozciągających się przez kilka ścieżek, ani zaczynających się w końcowej części ścieżki, a kończących w początkowej. Należy zakończyć segment instrukcją KOF przed bieżącym rozkazem.

Komunikat ten jest wydawany tylko w drugim przebiegu.

BYK36 ...: DAJ KOF ZBORNY

Przeadresowywany segment, rozpoczęty instrukcją NAF, nie został zamknięty instrukcją KOF, a jest następna instrukcja NAF, albo zaczyna się nowy blok instrukcją BLOK, QBLOK, WDANE lub WDANEX.

Instrukcje NAF-KOF nie mogą się zagnieżdżać, jak też wyznaczony przez nie segment nie może przekraczać granic bloków. Należy dla porządku zakończyć segment instrukcją KOF przed bieżącą instrukcją.

BYK37 ...: TU NIE MIEJSCE

Instrukcja AELIF, AELSE lub AFI, a może raczej AIF, znajduje się w niewłaściwym miejscu.

Instrukcje AIF, AELIF, AELSE i AFI tworzące jedną instrukcję złożoną muszą znajdować się w programie w tej właśnie kolejności (AELIF może wystąpić wielokrotnie). Instrukcja złożona AIF-AELIF-AELSE-AFI może całkowicie znajdować się wewnątrz klauzuli innej instrukcji złożonej AIF-AELIF-AELSE-AFI, ale nie mogą się one przecinać.

Być może instrukcja złożona jest zdekompletowana – jedną z przyczyn może być odrzucenie zbyt głęboko zagnieżdżonej instrukcji AIF.

W celu ułatwienia skompletowania poszczególnych klauzul w jedną całość dobrze jest posłużyć się identyfikatorem instrukcji (wspólnym dla poszczególnych instrukcji składowych), bez którego orientacja może być utrudniona, tym bardziej że instrukcje AIF-AELIF-AELSE-AFI mogą się zagnieżdżać.

BYK40 ...: PRZEPASTNE AIF

Zbyt wiele zagnieżdżonych instrukcji złożonych AIF-...-AFI.

Asembler przewiduje wiele poziomów zagnieżdżenia instrukcji AIF-...-AFI jedna w drugiej. Należy uprościć wyznaczanie alternatyw do dopuszczalnego poziomu, który wydaje się być wystarczająco wysoki.

BYK41 ...: GDZIE WE WSTAW

W tekście źródłowym wstawianym do głównego instrukcją WSTAW znajduje się kolejna instrukcja WSTAW lub WDANE. Polecenia tego nie da się zrealizować, gdyż oba czytniki taśmy perforowanej 5-kanałowej są zajęte: jeden na główny tekst źródłowy, a drugi na wstawiany.

Należy inaczej zorganizować wstawianie tekstów źródłowych – np. niektóre zasemblować osobno i wstawiać gotowy kod instrukcją WDANE.

Kod PIĄTKOWY wstawiany instrukcją WDANE można przekonwertować na kod THETA, np. programem „PIĄTKOWY na THETA.pgm.pt5”, i wstawiać instrukcją WDANEX, bowiem w instalacji komputerowej ODRA UMCS czytnik taśmy 8-kanałowej pozostaje ciągle do dyspozycji. Kody PIĄTKOWY i THETA są opisane w [5].

BYK42 ...: JENO ODRA UMCS

Instrukcja WDANEX wymaga czytnika 8-kanałowego, dostępnego tylko w instalacji komputerowej ODRA UMCS.

Należy wyłączyć maszynę i odpowiednio zmienić model ODRY, po czym ponownie wczytać asembler i przeprowadzić asemblację.

BYK43 ...: NIEZGODNA SUMA

Kod PIĄTKOWY lub THETA, wczytywany instrukcją WDANE lub WDANEX, ma niepoprawną sumę kontrolną.

Asembler oblicza sumę kontrolną danych i porównuje z zapisaną na końcu taśmy. Na zakończenie wczytywania drukuje pseudoinstrukcję UDANE lub UDANEX – gdy suma na taśmie jest zgodna z obliczoną, pseudoinstrukcję ZDANE lub ZDANEX – gdy taśma zakończona jest wystarczającą liczbą kodów NU oznaczających koniec danych bez sumy kontrolnej, albo pseudoinstrukcję NDANE lub NDANEX – gdy suma na taśmie jest niezgodna z obliczoną (i tę sytuację komentuje komunikat). Formaty taśm w kodach PIĄTKOWYM i THETA są opisane w [5].

Niezgodna suma kontrolna wcielanego kodu oznacza, że taśma z danymi jest uszkodzona, albo że to w ogóle nie jest taśma z danymi. Objawem uszkodzonej, albo zapisanej w innym formacie taśmy może być oczekiwanie na dalsze czytanie pomimo osiągnięcia końca taśmy.

BYK44 ...: JENO BEZ NAZWY

Komunikat przypomina, że dana instrukcja nie może mieć etykiety.

Etykieta niedozwolona jest jedynie w instrukcji KONIEC – do instrukcji KONIEC nie ma potrzeby przekazywać sterowania. Inne odwołania do jej adresu należy realizować np. poprzez postawienie przed nią instrukcji EQU.

Asembler instrukcję KONIEC w tekście wstawianym instrukcją WSTAW oddrukowuje jako pseudoinstrukcję OSTAW przydając jej jako etykietę identyfikator instrukcji WSTAW. Zakaz nadawania etykiety instrukcji KONIEC wynika stąd, że każdy tekst źródłowy można wstawiać do innego tekstu, a wtedy asembler nadaje pseudoinstrukcji OSTAW w polu etykiety identyfikator odpowiedniej instrukcji WSTAW, przez co oryginalna etykieta instrukcji KONIEC zostałaby utracona.

BYK45 ...: ZA WIELE NAZW

Program zawiera zbyt wiele symboli, przekraczając możliwości asemblera.

Aktualnie możliwe jest zdefiniowanie ponad 2000 nazw. Należy zredukować liczbę symboli, by zmieścić się w tym ograniczeniu. Asembler w wykazie symboli oznacza nieużywane symbole znakami zapytania `?`. Można próbować też zmniejszyć liczbę symboli przez zastąpienie niektórych odwołaniami względnymi wobec innych.

BYK46 ...: BRAK DEF NAZW

Komunikat pojawia się pod wykazem symboli, wskazując że pewne symbole są używane, ale nigdzie nie zostały zdefiniowane. Symbole niezdefiniowane są w wykazie oznaczone napisem `----- Byk46` odsyłającym do niniejszego komunikatu błędu.

Brak definicji nazwy może wynikać ze zwykłej pomyłki w nazwie, albo z wystąpienia innych błędów.

BYK47 ...: AUTORESTRYKCJA

Program wykrył błąd w samym sobie. Komunikat jest skutkiem zasemblowania instrukcji BYK znajdującej się najpewniej w którejś klauzuli instrukcji AIF-AELIF-AELSE-AFI.

Instrukcja BYK służy do wszczęcia przez sam asemblowany program alarmu o wykryciu w sobie błędu. Takim błędem mogłaby być np. nonsensowna dla danego algorytmu tablica o zerowej liczbie elementów, albo rozciągnięcie się programu na obszar programu STAŁEGO – to programista przewiduje jakie sytuacje chciałby wyłapywać automatycznie.

Instrukcja BYK zawiera argumenty, ale ponieważ drugi przebieg asemblera zostanie wykluczony, ich wartości, numer, tekst i ew. komentarze należy odczytać w treści instrukcji BYK wydrukowanej w listingu z pierwszego przebiegu.

Prowokowanie komunikatów błędów

Program źródłowy może w określonych warunkach wychwytywanych instrukcjami AIF-AFI kazać asemblować instrukcje NOTA i BYK, generując w ten sposób komunikaty ostrzegawcze lub błędu. Błędy można też generować przy pomocy odpowiednio dobranych, ale błędnie napisanych zwykłych instrukcji, np.:

```
      AIF warunek                ;warunek oznaczający błąd
        ---       ;komunikat 1     ;w pierwszym przebiegu BYK22 - NIEODGADNIONE
        komunikat ;komunikat 2     ;w pierwszym przebiegu BYK01 - CO ZA NAZWA
        (1/0)     ;komunikat 3     ;w drugim przebiegu    BYK16 - NIE PRZEZ ZERO
      AFI
```

W zależności od rodzaju instrukcji otrzymamy inny AsmC komunikat błędu. Niektóre błędy sygnalizowane są dopiero w drugim przebiegu, jak np. BYK16. Możemy więc sobie dobrać błąd stosowny do naszych potrzeb.

Czasem możemy otrzymać odpowiedni efekt przy mniejszej fatydze – bez instrukcji AIF-AFI. Jeżeli np. nasz program nie będzie poprawny dla pewnych wartości symboli, to możemy napisać w instrukcji EQU odpowiednie wyrażenie testujące poprawność na etapie asemblacji. Na przykład, jeśli tablica musi mieć co najmniej 3 elementy, to możemy zamieścić w programie instrukcję zawierającą dzielenie przez wartość prawdziwą 1 lub fałszywą 0 i odpowiedni komentarz:

```
        EQU  1/((koniecT-poczT)>=3)     ;test rozmiaru tablicy
```

W razie modyfikacji programu zanadto skracającej tablicę otrzymamy w drugim przebiegu translatora komunikat `BYK16 - NIE PRZEZ ZERO`, który będzie wymagał zajrzenia do listingu z pierwszego przebiegu, odczytania ew. komentarza i przeanalizowania wyrażenia świadczącego o wymaganiu. Instrukcja EQU nie generuje kodu, więc nie ma znaczenia że w poprawnych sytuacjach nie spowoduje błędu. Jeżeli takie wyrażenie testujące nie jest do niczego innego potrzebne, to instrukcja EQU nie musi mieć etykiety.

Techniki programistyczne

Komputery ODRA 1003 / 1013 (również w modyfikacjach UMCS) wyposażone są w program STAŁY, umożliwiający podstawowe operacje wczytywania programu do pamięci z taśmy perforowaną 5-kanałowej, oraz wyprowadzania treści pamięci w postaci bloków kodu na taśmę perforowaną 5-kanałową. Niektóre oryginalne pakiety oprogramowania, wraz ze swoimi bibliotekami programów narzędziowych i podprogramów, jak np. język PODSTAWOWY, system PROM wspomagania uruchamiania i testowania programów, autokody (języki programowania automatycznego) MOST czy FALA, zajmują część pamięci operacyjnej i narzucają pewne ograniczenia na programy użytkownika.

Asembler AsmC nie narzuca żadnych ograniczeń, pozostawiając całą pamięć operacyjną, całą maszynę, na potrzeby tłumaczonych programów, jako że istotą asemblera jest umożliwienie pisania dowolnych, nawet bardzo obszernych programów, bez narzucania jakichkolwiek ograniczeń i konwencji. Ciężar trudności został tu przeniesiony na procedurę asemblacji wymagającą bardzo dokładnego, dwukrotnego operowania nośnikami. Podczas asemblacji maszyna zajęta jest przez asembler – w czasie wykonywania programu użytkownika asembler jest zbędny.

Przedstawione w tym rozdziale techniki programistyczne są jedynie wskazaniem typowych metod organizacji różnych elementów programów, ale nie narzucaniem ich.

Organizacja programu głównego

Rozróżnienie między programem głównym a podprogramem zależy od środowiska, w którym będą wykonywane. Program główny uruchamiany za pośrednictwem jakiegoś systemu oprogramowania, np. systemu operacyjnego, musiałby stosować się do konwencji tegoż systemu. Tu za program główny uważamy program, który nie ma nad sobą żadnego systemu.

Program główny jest uruchamiany z pulpitu maszyny.

Należy ustawić adres pierwszego rozkazu do wykonania i wystartować CPU. Program główny może mieć wiele punktów wejścia, tj. adresów od których należy go uruchamiać. Szczególnym adresem jest 0, gdyż można go ustawić jako startowy jednym przyciskiem na pulpicie maszyny.

Adres 0 wyróżnia się wśród innych pod jeszcze innym względem – komórkę pamięci o niezerowym adresie N można wygodnie zainicjować wartością zerową, dodatnią N lub ujemną -N rozkazami:

```
[N] = 0          [N] = N          [N] = -N
```

co daje trójstanowe przełączniki logiczne. Tej możliwości nie daje adres 0, więc najlepiej nie umieszczać pod nim zmiennej, a raczej rozkaz, np. początek programu.

Program główny nie musi przywracać stanu maszyny (rejestrów)

Stan maszyny, zawartość rejestrów i pamięci, przy uruchomieniu programu głównego nie jest znany i może pozostać po prawidłowym lub błędnym wykonaniu, albo zatrzymaniu przez operatora w dowolnym punkcie. Program główny nie musi zapamiętywać i na zakończenie pracy przywracać wartości rejestrów maszyny.

Program powinien kończyć pracę rozkazem STOP

Mogą istnieć programy pracujące w nieskończoność, których zatrzymanie wymusza operator stopując maszynę. Jednak programy mające do wykonania skończone zadanie powinny kończyć się rozkazem STOP, a nie np. poprzez sprowokowanie nadmiaru zmiennoprzecinkowego. W programie przewidzianym do wielokrotnego uruchamiania, jak np. w asemblerze AsmC do asemblowania kolejnych programów, może to być STOP niestabilny, po którym wystarczy nacisnąć przycisk [StartCPU] bez ponownego ustawiania adresu startu, by rozpoczęło się nowe wykonanie.

Zaleca się stosować ogólnie przyjęte konwencje komunikacji z operatorem

Stosowanie jednolitych konwencji ułatwia pracę operatorowi.

Zatrzymanie się programu po całkowitym wykonaniu się powinno następować rozkazem:

- STOP z argumentem 0 oznaczającym, że nic szczególnego się nie wydarzyło
- STOP z argumentem niezerowym, sygnalizującym szczególną przyczynę poprzez numer wyświetlany na lampkach rejestru rozkazów

Do sygnalizowania wyniku wykonania programu należy w razie potrzeby stosować załadowanie akumulatora A wartością wyświetlaną na lampkach akumulatora.

Bardziej szczegółowe informacje dot wykonania programu zaleca się podawać w postaci komunikatów drukowanych na dalekopisie.

SYSPARM – dostosowanie programu do środowiska komputerowego

Asembler AsmC oferuje standardowy symbol SYSPARM, którego wartość jest typu Num i na poszczególnych bitach zawiera informacje o systemie komputerowym, na którym właśnie działa. Informacje te pochodzą z programu STAŁEGO emulatora i są szczegółowo opisane w [5]. Należy wyraźnie zaznaczyć, że program STAŁY emulatora jest rekonstrukcją i rzeczywisty program nie zawierał w sobie informacji o konfiguracji sprzętowej, a więc odwoływanie się do symbolu SYSPARM uzależnia asemblowany program od Emulatora emc ODRA 1003/1013 – poza nim wartość symbolu SYSPARM jest niezdefiniowana i może być przypadkowa.

Wartość SYSPARM jest ustalona w danym wykonaniu asemblera – zmienia się wraz ze zmianami modelu komputera, rodzaju dalekopisu i komputera na którym działa emulator.

Przy pomocy wyrażeń oraz instrukcji AIF-AELIF-AELSE-AFI z warunkami, opartych o wartość SYSPARM, można uogólnić program źródłowy tak, by jego asemblacja generowała kod jak najlepiej dostosowany do cech sprzętu, na którym ma być używany. Należy pamiętać, że program będzie dostosowany do takiego środowiska komputerowego, na którym został zasemblowany.

Przykładowy program „test drukT5, drukT6, drukT7.asm.txt” demonstruje użycie SYSPARM do generowania stosownych kodów przejścia do nowego wiersza.

ASMPARM – przewidywanie wymaganych wariantów programu

Program użytkownika może być przygotowany w sposób wariantywny, zależnie od wymagań programisty. Dla przykładu programista może wstawić dodatkowe instrukcje na czas testowania do programu. Takie instrukcje zwykle wydłużają program, spowalniają a nawet wstrzymują jego pracę i są zbędne w przetestowanej wersji produkcyjnej. Usuwanie dodatkowych instrukcji jest uciążliwe i podatne na pomyłki, dlatego najlepiej pozostawić je, ale spowodować by na życzenie programisty zostały pominięte przez asembler.

AsmC oferuje symbol standardowy ASMPARM, którego wartością typu Num jest liczba ustawiona przyciskami części AR (N) akumulatora. Poszczególne bity mogą oznaczać wymagania programisty co do asemblacji. Przy pomocy wyrażeń oraz instrukcji AIF-AELIF-AELSE-AFI z warunkami, można spreparować program tak, by stosownie do wartości ASMPARM generowały się takie czy inne instrukcje, dając doskonale dopasowany do wymagań wariant programu. Z jednego tekstu źródłowego można wygenerować wiele różnych wariantów.

Wartość ASMPARM jest ustalana raz na początku danego wykonania asemblera i nie zmienia się aż do końca asemblacji.

Program „PIĄTKOWY-THETA.asm.txt” demonstruje użycie ASMPARM do generowania różnych wariantów programu konwersji kodów PIĄTKOWEGO i THETA.

CzK – sterowanie pracą programu przy pomocy opcji

Dostosowanie programu zgodnie z wartościami SYSPARM i ASMPARM ma charakter statyczny, wpływa na przekład programu generując stosowny kod maszynowy. Czym innym jest dynamiczne wpływanie na działanie programu w czasie jego pracy.

Typowe sposoby sterowania pracą są wbudowane w samą maszynę w postaci zatrzymywania i wznawiania przyciskami [StopCPU] i [StartCPU], ustawiania pracy krokowej zwolnieniem przycisku [PC], czy wstrzymywanie pracy przyciskami [StopAR], [StopNR] i [StopNd]. Są to mechanizmy przewidziane do testowania programów.

Bardziej zaawansowane rozwiązanie to opatrzenie programu w wariantywne lub opcjonalne fragmenty wykonywane na żądanie operatora w dowolnej chwili podczas pracy programu. Program cały czas zawiera wszystkie te fragmenty, ale wykonuje tylko wymagane. Możliwe jest to dzięki rozkazowi CzK.

Rozkaz CzK odczytuje stan przycisków akumulatora i zapisuje w rejestrze A. Do dyspozycji jest więc 39 bitów, na których można zaprojektować odpowiednie opcje pracy programu. Program powinien na odpowiednich etapach działania odczytywać stan przycisków i dalej działać zgodnie z nim. Np. przy wciśniętym przycisku drukować komunikaty obrazujące zaawansowanie wykonywania się programu, a przy zwolnionym pomijać drukowanie dla oszczędności papieru i czasu. Oczywiście samo odczytywanie stanu przycisków akumulatora i testowanie bitów wartości też zajmuje czas pracy programu.

Organizacja podprogramów

W odróżnieniu od programu głównego, który zasadniczo nie ma żadnych obowiązków, podprogram jest wykonywany po wywołaniu go przez inny program. Programem wywołującym może być program główny lub podprogram. W szczególności podprogram może wywołać sam siebie. Z podprogramami wiążą się następujące problemy:

- przekazywanie sterowania do podprogramu i powrót do programu nadrzędnego
- przekazywanie argumentów do podprogramu i zwracanie wyników
- gospodarowanie rejestrami, by jeden program nie zakłócał pracy drugiego
- gospodarowanie pamięcią operacyjną, by jeden program nie zakłócał pracy drugiego

SKS – podprogramy ze śladem w pamięci operacyjnej

Wywołanie podprogramu wymaga wskazania adresu podprogramu oraz adresu powrotnego. Programista może organizować tę operację na różne sposoby, ale komputer dysponuje rozkazem SKS przeznaczonym specjalnie do tego celu.

Rozkaz **`SKS PODPROG..POWRÓT`** powoduje:

1. zapisanie w komórce pamięci samego siebie z wyzerowanymi bitami operacji (bity 0...8) i wyzerowanym bitem Z (bit 38) – w efekcie zostanie do niej wpisany rozkaz arytmetyczny `0..POWRÓT` tj. pobrania zera do sumatora, ustawienia warunku Z, skasowania warunków U i V i przejścia pod adres POWRÓT, czyli powrotu do programu wywołującego
2. skok pod adres PODPROG+2, gdzie powinien znajdować się pierwszy rozkaz podprogramu – słowo o adresie PODPROG+1 zostaje przeskoczone

```
PROGRAM BLOK
        ...
        SKS  PODPROG  ;wywołanie nr 1
        ...
        SKS  PODPROG  ;wywołanie nr 2
        ...
        STOP
```

```
PODPROG STOP 0 ..*+2      ;ślad
        DS   0            ;słowo wolne
        ;pierwszy rozkaz
        ;następne rozkazy
        ...
        0      ..PODPROG  ;powrót po
                          ;śladzie
```

Słowo pod adresem PODPROG nie musi być zainicjowane. Konwencjonalnie wypełnia się je rozkazem `STOP 0..*+2` dla ułatwienia testowania podprogramu, jak to opisano w [2].

Słowo o adresie PODPROG+1 zostaje przeskoczone ze względu na czas potrzebny do zapisania śladu i odnalezienia adresu PODPROG+2. Słowo to może zostać użyte do dowolnego celu.

Powrót z podprogramu polega na przejściu do rozkazu śladowego zapisanego pod adresem PODPROG – wykona on przejście pod adres POWRÓT, tj. pod adres wskazany w rozkazie SKS jako adres następnego rozkazu po rozkazie SKS.

Zalety wywołania podprogramu rozkazem SKS:

- jest to prosty rozkaz wywołania podprogramu nie angażujący B-rejestrów
- wygodny skok do rozkazu śladowego zarówno przez pierwszą jak i drugą część adresową

Wady wywołania podprogramu rozkazem SKS:

- ślad jest zapisywany w pamięci (komórka ze śladem nie może być chroniona przed zapisem)
- podprogram nie może mieć wielu punktów wejścia, a przynajmniej jest to utrudnione, bo ślad byłby zapisywany raz w tym, raz w innym punkcie wejścia
- podprogram nie może mieć wielu punktów wyjścia, a przynajmniej jest to utrudnione, bo jest tylko jeden rozkaz powrotu po śladzie (rozkaz śladowy)
- podprogram nie może zwrócić ustawionych przez siebie warunków, gdyż niszczy je rozkaz powrotu umieszczany w komórce pamięci jako ślad – rozkaz śladowy jest rozkazem arytmetycznym ustawiającym warunki

SKSB – podprogramy ze śladem w rejestrze B-modyfikacji

Alternatywnym do rozkazu SKS sposobem wywołania podprogramu jest rozkaz arytmetyczny **`B6=POWRÓT..PODPR`** – powoduje on:

1. wstawienie do rejestru B6 adresu powrotu
2. przejście pod adres PODPR, gdzie powinien znajdować się pierwszy rozkaz podprogramu – podprogram może mieć wiele adresów wejściowych (punktów wejścia)

Rozkaz `B6=POWRÓT..PODPR` ma w asemblerze AsmC swój synonim i może być napisany jako **`SKSB B6,PODPR..POWRÓT`** – czyli w sposób analogiczny do rozkazu SKS.

```
PROGRAM BLOK
        ...
        SKSB B6,PODPR  ;wywołanie nr 1
        ...
        SKSB B6,PODPR  ;wywołanie nr 2
        ...
        SKSB B6,PODPR1 ;wywołanie nr 3
        ...            ;przez inny
        ...            ;punkt wejścia
        STOP
```

```
PODPR   ;pierwszy rozkaz tego wejścia
        ;drugi rozkaz
        ...
PODPR1  ;pierwszy rozkaz tego wejścia
        ;drugi rozkaz
        ...
        ;wspólne rozkazy
        ...
        NIC    ..B6  ;powrót po
                     ;śladzie w B6
```

Rejestr B6 jest tu wybrany dlatego, że modyfikuje drugą część adresową rozkazu, zawierającą adres następnego rozkazu do wykonania.

Powrót z podprogramu polega na przejściu pod adres (0+B6) z dowolnego rozkazu pozwalającego na użycie rejestru B6 w roli B-modyfikatora. Takim rozkazem może być np. `0..B6` – taki sam jak ślad w przypadku rozkazu SKS – ale jest to rozkaz arytmetyczny ustawiający warunki. Jeżeli podprogram powinien zwrócić warunek jako wynik swojego działania, to może przydać się rozkaz **`NIC 0..B6`** – niezdefiniowany, a więc nie robiący niczego, nawet nie ustawiający warunków.

Szczególną sytuację podprogram może sygnalizować programowi wywołującemu nie przez ustawienie warunku, lecz przez powrót do odpowiedniego z kilku rozkazów zamieszczonych w kolejnych lokacjach wskazanych adresem powrotu w rejestrze B6. Powrót z podprogramu polega wtedy na przejściu pod adres (0+B6), (1+B6), itd. Taki wektor rozkazów może być wygodniejszy, bowiem pozwala wykroczyć poza standardowy zestaw warunków maszynowych, a ponadto oszczędzić na rozkazach skoków warunkowych. Spójrzmy na poniższy przykład:

```
PROGRAM BLOK
        ...
        B2 = tabl+...    ;adres tabl[pocz]
        B3 = tabl+...    ;adres tabl[końc]
        B7 = ...         ;szukany klucz
        SKSB B6,BISEKT                    1
B6plus0 B2 = txtBrak   ..drBrak
B6plus1 A  = [B1]      ..jestEl
B6plus2 A  = A & 0c777   ;maska klucza
        A  = A & [B1]    ;klucz środk.el.
        A - B7 ..bisCOMP ;porównanie
        ...
drBrak  EQU  *           ;drukuj że brak
        ...
jestEl  EQU  *           ;A = el.tabl
        ...

tabl    DS   10,20,30,40,... ;tablica
        ...
        STOP
```

```
; B2 = adres pierwszego elementu tablicy
; B3 = adres ostatniego elementu tablicy
BISEKT  EQU  *
        A  = B3
        A - B2
        SKM  przezB6  ;BRAK: B2>B3        2
        A  = A + B2
        A  = A >>> 1
        B1 = A        ;adres środk.el.tabl
        NIC  0 ..B6+2 ;porównaj klucze    3
bisCOMP EQU  *        ;środk.el. z szukanym
        SKM  bisLT    ;środkowy < szukany
        SKW  ++..B6+1 ;środkowy > szukany
                      ;a gdy ==, to do B6+1
bisGT   A  = B1       ;szukaj na pocz. tabl
        B3 = A - 1 ..BISEKT
bisLT   A  = B1       ;szukaj na końcu tabl
        B2 = A + 1 ..BISEKT

przezB6 NIC  0..B6
```

Ogólnie pomyślany podprogram wyszukiwania elementu w uporządkowanej rosnąco tablicy zwraca sterowanie pod adres (B6+0) gdy w tablicy nie ma elementu o tym kluczu, pod adres (B6+1) gdy szukany element znajduje się w tablicy pod indeksem zawartym w B1, zaś pod (B6+2) gdy domaga się porównania elementu tablicy o indeksie B1 z elementem szukanym. W przykładowym wywołaniu podprogramu w linii *1* będzie to powrót sterowania odpowiednio pod adres B6plus0, B6plus1 lub B6plus2 (gdzie w istocie znajduje się algorytm porównania klucza elementu o adresie B1 z kluczem szukanym, wywoływany rozkazem w linii *3* i wracający zawsze do bisCOMP). Rozkazy B6plus0, B6plus1 i B6plus2 tworzą łatwy do adresowania względem B6 wektor punktów powrotu z podprogramu BISEKT. Nie zawsze powrót przez rejestr B6 jest wygodny – w linii *2* widać że potrzebny jest rozkaz przesiadkowy przezB6.

W szczególnych sytuacjach, w podprogramach gdzie wszystkie rozkazy powrotu odbywają się przez adres podawany w pierwszej części adresowej, wygodniejsze może być posłużenie się innym rejestrem B-modyfikacji. Może tak być np. przy powrotach za pomocą rozkazów skoków warunkowych z prawdziwym warunkiem.

Oto przykład posłużenia się rejestrem innym niż B6, np. rejestrem B1:

```
PROGRAM BLOK
        ...
        SKSB B1,PODPROG
        ...
        STOP
```

```
PODPROG ;pierwszy rozkaz
        ;następne rozkazy
        A             ;ustawienie warunku
        SKLC B1--,B1  ;powrót po
                      ;śladzie w B1
```

W rozkazie SKLC najpierw nastąpi B-modyfikacja obliczająca adres skoku (0+B1), a dopiero potem dekrementacja B1 bez zmiany warunków (można zastosować inkrementację) i skok pod adres (0+B1) – zakładamy, że adres powrotu w B1 nie jest zerowy, oraz że zmiana wartości B1 przy powrocie nie ma znaczenia.

Zalety wywołania podprogramu rozkazem SKSB:

- ślad nie jest zapisywany w pamięci (pamięć podprogramu można chronić przed zapisem)
- podprogram może mieć wiele punktów wejścia, bo adres powrotu jest zawsze w tym samym miejscu – w rejestrze B6
- podprogram może mieć wiele punktów wyjścia, bo powrót może następować pod jeden z wielu adresów względem adresu powrotu znajdującego się w rejestrze B6
- podprogram może zwrócić ustawione przez siebie warunki, których nie niszczy powrót rozkazem NIC pod adres zawarty w rejestrze B6

Wady wywołania podprogramu rozkazem SKSB:

- rozkaz wywołania podprogramu angażuje rejestr B6 do przekazania adresu powrotu
- skok powrotny z podprogramu przez rejestr B6 możliwy tylko przez drugą część adresową

Gospodarka pamięcią

Podstawowe sugestie co do sposobu korzystania z pamięci operacyjnej wynikają już z konstrukcji maszyny. Nawet połowę pamięci, komórki o adresach od 0 do 4095, można zablokować przed zapisem przy pomocy przycisków na pulpicie. Zatem początek pamięci powinien być przeznaczany na programy które nie modyfikują siebie, a nie na dane.

Najwyższe adresy zajęte są przez program STAŁY. Dzięki temu zmniejsza się ryzyko błędu spowodowanego przeniesieniem sumy z B-modyfikacji na kod operacji podczas indeksowania tablicy umieszczonej w środkowym obszarze pamięci.

Jeśli weźmiemy pod uwagę istnienie dwóch ścieżek ferrytowych tuż przed ścieżką STAŁĄ, to należy zalecić umieszczanie obszernych danych za programem, a przed ścieżkami ferrytowymi. Oczywiście, ścieżki ferrytowe można przeznaczać na (może dynamicznie ładowane) programy (podprogramy), lub na obszary robocze (może również dynamicznie ładowane).

Pamięć robocza

Program główny zwykle ma swoje zmienne globalne, dla których najprościej jest wyznaczyć stosowny obszar pamięci. Nawet wyodrębnione w podprogramy niektóre części programu głównego będą operowały na zmiennych globalnych, ale też często potrzebują pamięci roboczej tylko na czas swojego wykonania. Można próbować wykorzystywać wspólny obszar pamięci na takie lokalne potrzeby, ale wiąże się to z ryzykiem pomyłek – chyba że zastosuje się stos pamięci roboczej. Jeśli pamięci wystarcza to najlepiej każdemu nierekursywnemu podprogramowi wyznaczyć jego własny obszar roboczy.

Jako robocze można, pod określonymi rygorami, wykorzystywać komórki robocze programu STAŁEGO. Adresy **0c17577**, **0c17576**, **0c17573** i **0c17572** są zajmowane tylko na czas wywołania podprogramów programu STAŁEGO i tylko wtedy nie mogą zawierać danych programu wywołującego. Adresy **0c17575** i **0c17574** przechowują dane podprogramów wyprowadzających i mogą być używane tylko wtedy, gdy w ogóle nie używa się tych podprogramów. Odpowiednie informacje znajdują się w [5].

Rejestry robocze

W skrypcie [2] zaleca się przy budowie podprogramów stosować restrykcyjną zasadę, by wszystkie rejestry nie przenoszące informacji między programem wywołującym a podprogramem, były odtwarzane. Jest to najbardziej bezpieczna konwencja i jak najbardziej słuszna w podprogramach ogólnego użytku.

Rzecz w tym, że w programie głównym cały czas używa się rejestrów, choćby np. do organizacji pętli: jednej, za chwilę drugiej, i między tymi użyciami nie odtwarza ani zachowuje się rejestru, bo wiadomo, że te użycia są chwilowe, a sama wartość rejestru przed, jak i po użyciu staje się nieistotna. Zatem w podprogramach lokalnych, związanych z jednym programem, można rozważyć swobodniejszą konwencję, np:

- wyznaczyć kilka rejestrów jako robocze, o ulotnej wartości, nie wymagające odtwarzania, np.: A, B7, B1, B2 i B6 – zabezpieczeniem ich zawartości powinien się zajmować program wywołujący. Rejestry ulotne nadają się do przekazywania argumentów do podprogramów i odbierania z nich wyników.
- pozostałe: B3, B4 i B5 – jako rejestry o zawartości nieulotnej, których zawartość powinna pozostać niezmieniona po użyciu podprogramu – zabezpieczeniem ich zawartości powinien się zajmować podprogram. Rejestry nieulotne nadają się do przechowywania bardziej globalnych informacji.

W tej konwencji unika się częstych operacji zapamiętania i odtworzenia rejestrów w małych programikach, w których byłoby to istotniejszym obciążeniem, zaś w większych przywracanie rejestrów byłoby mniejszym obciążeniem.

Przekazywanie argumentów do i odbieranie wyników z podprogramów

Opisane w [2] konwencje dotyczące przekazywania danych do i z podprogramów za pośrednictwem akumulatora A i rejestru B7, należałoby w luźniejszej konwencji uzupełnić o rejestry B1 i B2.

Dodatkowo rejestr B6 narzuca się głównie jako adres powrotny z podprogramu wywoływanego instrukcją SKSB, a więc B6 w sposób naturalny staje się w programie nadrzędnym rejestrem roboczym.

Przypomnijmy, że instrukcja SKSB umożliwia wywoływanie podprogramu przez jeden z wielu punktów wejścia (też stanowiących informację wejściową – argument), powrót do jednego z kilku punktów powrotu (względnych wobec adresu B6 – jako informację wyjściową), oraz zwracanie jako wynik kodu warunku Z, U, D i V.

Większą liczbę argumentów można organizować jako listę (czy blok) parametrów i przekazywać adres tej listy. Jednym ze sposobów jest umieszczanie takiej listy tuż przed lub za punktem powrotu, którego adres i tak byłby przekazywany w rejestrze B6 – metoda ta jest warta rozważenia szczególnie wtedy, gdy zawartość listy jest stała (nie zmienia się podczas pracy programu).

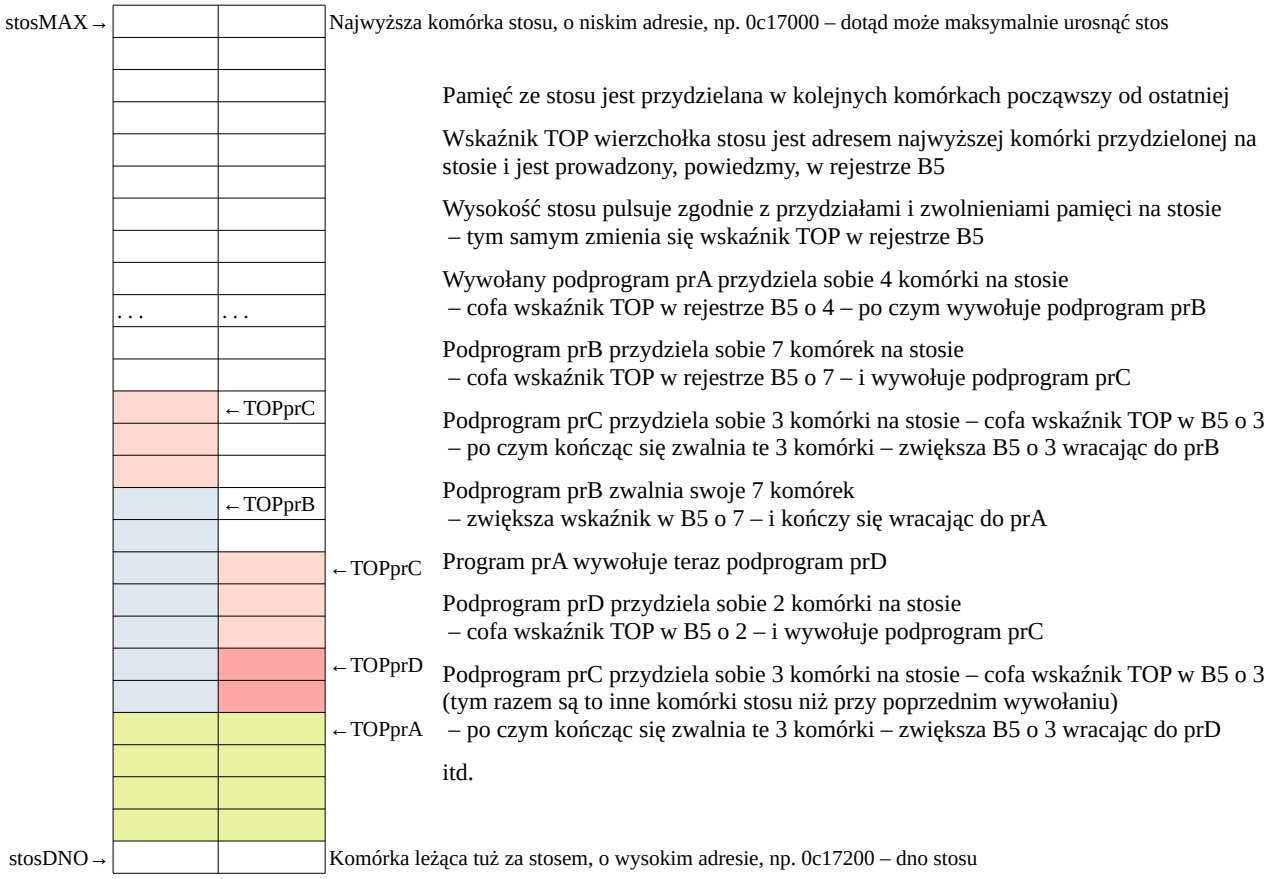
Argumenty można też przekazywać na stosie pamięci roboczej.

Stos pamięci roboczej

Jednym z najbardziej efektywnych sposobów przydzielania programom pamięci roboczej jest stos pamięci. Stos jest liniową strukturą danych (typu LIFO), w której dane dokładane są na wierzch stosu i z wierzchołka stosu są pobierane. Współczesne komputery zwykle dysponują wygodną i sprawną sprzętową obsługą stosu, ale ODRA nie ma takiego mechanizmu.

Stos jest wygodny gdy trudno zapanować nad użytkowaniem wspólnych obszarów pamięci (bo inaczej brak jej do obdzielenia wszystkich podprogramów), albo gdy mamy program rekursywny.

Stos można oprogramować samodzielnie jako tablicę słów:



Poniżej mamy prosty schemat programu głównego przygotowującego stos pamięci na potrzeby podprogramów, oraz podprogramu przydzielającego sobie pamięć roboczą na stosie.

```
PROGRAM BLOK
        B5 = stosDNO   ;wskaźnik
                       ;wierzchołka stosu
        ...
        SKSB B6,podprog
        ...
        STOP

        QBLOK 0c17000  ;stos pamięci
stosMAX DS    [128] 0  ;wielkość stosu
stosDNO EQU   *

        KONIEC
---
```

```
podprog EQU  *
      ! B5 = (B5-(robK-robP))  ;przydział
                               ;pamięci
        ...
        A   = B7       ;jakieś
        [x] = A        ;obliczenia
        ...
                               ;zwolnienie
        B5 = (B5+(robK-robP))  ;pamięci
        NIC  0 ..B6    ;powrót po śladzie

robP    QBLOK B5 ;pamięć robocza na stosie
x       DS    0        ;zmienna robocza
y       DS    0        ;zmienna robocza
robK    EQU   *
```

Program główny wyznacza blok pamięci przeznaczony na stos w postaci QBLOKu, podając jego adres i wielkość, oraz inicjuje wskaźnik wierzchołka stosu. Wskaźnik wierzchołka stosu zawsze zawiera adres najwyższego zajętego słowa. Na początku jest to adres tuż za stosem, bo jeszcze ani jedna komórka pamięci nie została przydzielona podprogramom. Od tej pory można wywoływać podprogramy posługujące się stosem na potrzeby pamięci roboczej.

Podprogram przydziela sobie pamięć w postaci segmentu o potrzebnej wielkości. Przydział polega na cofnięciu wskaźnika wierzchołka stosu o tę wielkość. Zwróćmy uwagę, że do pomniejszenia wartości w rejestrze B5 pełniącym rolę wskaźnika wystarcza jeden rozkaz. Jest to rozkaz pobrania do rejestru B5 ujemnego argumentu bezpośredniego indeksowanego rejestrem B5. Ujemny argument to w istocie argument dodatni uzupełniający wielkość do 8192, bo wyrażenia adresowe są obliczane modulo 8192. W procesie B-modyfikacji adres zawarty w B5 po zsumowaniu z argumentem „ujemnym” wykroczy poza 13 bitów części AR rozkazu na kod operacji tego rozkazu, dlatego zastosowano korekcję rozkazu znakiem wykrzyknika (zakładamy, że suma nie cofa przed adres 0c00000).

Gdy na koniec podprogramu trzeba zwolnić przydzieloną pamięć roboczą, należy wskaźnik stosu popchnąć do przodu (powiększyć o wielkość zwalnianego segmentu). Znowu łatwo to zrobić jednym rozkazem, ale tym razem dodanie wielkości segmentu za pomocą B-modyfikacji nie spowoduje przeniesienia na kod operacji i korekcja nie jest potrzebna (zakładamy, że suma nie wychodzi poza adres 0c17777).

Przydzielony segment jest niezainicjowany i zwykle zawiera „śmieci” pozostałe po programach, którym był przydzielany. Adres bloku o nazwie robP, definiującego postać segmentu, znajduje się we wskaźniku wierzchołka stosu, stąd rejestr B5 indeksuje ten QBLOK. Odwołanie do zmiennej x ma więc postać (B5+0), do zmiennej y postać (B5+1), itd. Widać, że odwołania względem B5 są proste. To wyjaśnia, dlaczego stos rośnie od większych adresów w stronę mniejszych – w przeciwnym razie wskaźnik wierzchołka stosu wskazywałby na koniec przydzielonego segmentu i odwołania do zmiennych byłyby ujemne.

Ponieważ adresy zmiennych na stosie są względne wobec wskaźnika wierzchołka stosu w rejestrze B-modyfikacji, to nie da się jednym rozkazem np. posłać do słowa x zawartości rejestru B7 – konieczne jest pośrednictwo rejestru A, a to może być istotnym utrudnieniem.

Zasadniczym problemem przy stosowaniu stosu jest oszacowanie jego wielkości. Ilość pamięci jaką należy przeznaczyć na stos zależy od głębokości wywołań podprogramów. W tym celu należy przeanalizować drzewo wywołań sumując maksymalne zapotrzebowanie na pamięć. Nie jest to łatwe zadanie, szczególnie że przy modyfikacjach programu zapotrzebowanie prawdopodobnie się zmienia. Dodatkową trudność wprowadzają programy rekurencyjne, w których głębokość wywołań zwykle zależy od danych. Gdy każda wielkość stosu może okazać się niewystarczająca należy kontrolować przekroczenie jego rozmiaru. Oto modyfikacja przydziału pamięci na stosie w powyższym schemacie, wprowadzająca odpowiednią kontrolę:

```
podprog EQU  *
      ! B5 = A = (B5-(robK-robP));przydział
        A - stosMAX              ;pamięci
        SKNM okP                 ;z kontrolą
          STOP                   ;wskaźnika
okP     EQU  *                   ;stosu
        ...
```

Podprogram powinien zwalniać pamięć przydzieloną na stosie. Przydział i zwalnianie może odbywać w wielu kawałkach, co zwiększa ryzyko błędu. Jednym z błędów może być cofnięcie wskaźnika wierzchołka stosu poniżej jego dna. Oto modyfikacja zwolnienia pamięci na stosie wprowadzająca wykrywanie tego błędu:

```
podprog EQU  *
        ...
                                ;zwolnienie
        B5 = A = (B5+(robK-robP));pamięci
        A - stosDNO             ;z kontrolą
        SKNW okZ                ;wskaźnika
          STOP                  ;stosu
okZ     EQU  *
        ...
```

Sekwencje rozkazów kontrolujących operacje przydziału i zwalniania pamięci na stosie nie tylko wydłużają programy i czas pracy, ale też wymagają użycia akumulatora A. Bardzo możliwe, że w rejestrze A przekazywany jest argument i/lub wynik podprogramu, zatem pokazane kontrole wymagają przechowania wartości A. Naturalnym byłoby zarezerwowanie na ten cel miejsca na stosie, ale przed kontrolą wymagającą użycia A nie wiadomo czy jest na nim miejsce.

Rozwiązania są różne. Przez chwilę (tj. nim nastąpi wywołanie jakiegoś podprogramu) można przechować A w oddzielnej komórce pamięci (np. komórce roboczej programu STAŁEGO).

Inne polega na umówieniu się, że każdy program zostawia na stosie jedną (lub kilka) komórek do użytku przez wywoływane podprogramy i tam można byłoby chować wartości rejestrów do czasu przydziału własnego segmentu pamięci. Rozwiązanie to ma pewne zalety, których tu nie będziemy omawiać, ale w maszynie z małą pamięcią wadą jest rezerwowanie komórek, które nie wiadomo czy będą podprogramom potrzebne.

Jeszcze inne rozwiązanie problemu polega na pozostawieniu jednej (lub kilku) wolnych komórek tuż przed stosem. Oto jak wtedy wyglądałby schemat programu głównego i podprogramu zawierającego omówione kontrole stosu:

```
PROGRAM BLOK
        B5 = stosDNO   ;wskaźnik
                       ;wierzchołka stosu
        ...
        SKSB B6,podprog
        ...
        STOP

        QBLOK 0c17000  ;stos i przedstos
        DS    0        ;przed stosem
stosMAX DS    [128] 0  ;stos pamięci
stosDNO EQU   *

        KONIEC
---
```

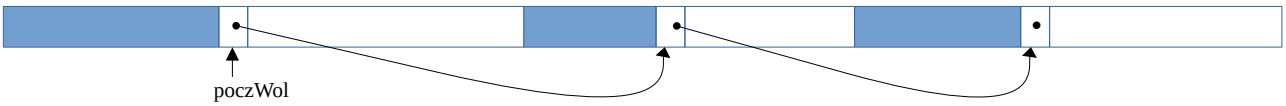
```
podprog EQU  *
      ! [archA-(robK-robP)] = A  ;argument
      ! B5 = A = (B5-(robK-robP));przydział
        A - stosMAX             ;pamięci
        SKNM okP                ;z kontrolą
          STOP                  ;wskaźnika
okP     EQU  *                  ;stosu
        ...
        A   = B7       ;jakieś
        [x] = A        ;obliczenia
        ...
                                ;zwolnienie
        B5 = A = (B5+(robK-robP));pamięci
        A - stosDNO             ;z kontrolą
        SKNW okZ                ;wskaźnika
          STOP                  ;stosu
okZ     EQU  *
      ! A = [archA-(robK-robP)]  ;wynik
        NIC  0 ..B6    ;powrót po śladzie

robP    QBLOK B5 ;pamięć robocza na stosie
x       DS    0        ;zmienna robocza
y       DS    0        ;zmienna robocza
archA   DS    0        ;przechowalnia A
robK    EQU   *
```

Zmienna archA musi znajdować się na końcu segmentu robP...robK. Teraz można w ciemno zapamiętać A w segmencie stosu, który dopiero za chwilę zostanie przydzielony (a może nie zostanie przydzielony, bo zabraknie dla niego miejsca). Nawet jeśli zabraknie miejsca, to najwyżej A zostanie zapamiętane w komórce przed stosem nie niszcząc niczego innego. Podobnie zwolnienie segmentu na stosie nie niszczy zapisanych w nim danych, więc można użyć A do operacji zwolnienia, a następnie przywrócić jego wartość na podstawie informacji zawartych w zwolnionym segmencie stosu.

Sterta pamięci roboczej

Niektóre algorytmy potrzebują dynamicznie przydzielanej pamięci, obszarów alokowanych i zwalnianych na żądanie, niezależnie od wywołań podprogramów, trwających również pomiędzy wywołaniami – pamięci na stercie. Sterta to obszar pamięci przeznaczony na dane, których przydział i zwalnianie odbywa się w sposób całkowicie dowolny i nie może być zrealizowany na stosie, w strukturze typu LIFO.

W wyniku przydziałów i zwalniań pamięci, na stercie powstaje wiele rozłącznych obszarów pamięci przydzielonej, rozdzielonych obszarami pamięci wolnej. Najprostsza implementacja polega na trzymaniu listy wszystkich wolnych obszarów sterty. Na początku cała sterta jest pusta, więc pierwsze słowo tego pustego obszaru będzie zawierać długość obszaru (na razie wielkość sterty) i adres następnego pustego obszaru (zerowy, bo go nie ma). W miarę przydziałów pamięci na stercie adres i wielkość pustego obszaru będą się zmieniać. Po zwolnieniu pamięci na stercie powstanie nowy obszar pusty. Ten nowy obszar pusty może da się, a może nie da się scalić z jednym lub dwoma wcześniej istniejącymi obszarami pustymi i wtedy liczba i wielkość obszarów pustych się zmieni. Każdy obszar pusty zawiera w swoim pierwszym słowie swoją długość i adres następnego obszaru pustego (zerowy, gdy więcej ich nie ma).



Lista pustych obszarów powinna być uporządkowana w kolejności ich adresów. Algorytmy przydziału i zwalniania obszaru na stercie powinny uwzględniać zjawisko fragmentacji pamięci, nie pozwalające znaleźć spójnego obszaru o wymaganej wielkości pomimo wystarczającej wielkości sumy pustych obszarów. Zaawansowane systemy oprogramowania zwykle stosują funkcje automatycznej defragmentacji, wymagającej jednak bardziej złożonej reprezentacji danych adresowych.

Kontrola poprawności przydziału i zwolnienia obszaru na stercie może być zbędna w programach dobrze przetestowanych, ale na czas testowania mogłaby być włączana warunkowo instrukcjami AIF-AFI podczas asemblacji, lub opcjami wykonania podczas działania.

Obsługa sterty jest na tyle złożona i czasochłonna jak na możliwości ODRY, że należy unikać jej stosowania i z tego względu więcej jej tu nie omawiamy.

Podprogramy rekursywne – funkcje rekurencyjne

Niektóre algorytmy wygodnie jest oprogramować za pomocą podprogramów rekursywnych. Szkolny przykład to zaprogramowanie wzoru rekurencyjnego funkcji silnia: 0!=1, n!=(n-1)!*n:

```
program BLOK 0
        B5 = stosDNO   ;wierzchołek stosu
        A  = [N]             ;parametr N
        SKU  błędneN
        A - [maksN]
        SKW  błędneN
        SKSB B6,silnia       ;wynik B7=N!
        STOP 0       ..program
błędneN STOP 0c17777 ..*


MAXN    EQU  14      ;maksymalna wartość N
maksN   DS   (MAXN)
N       DS   5       ;argument od 0 do 14

/*
 Funkcję silnia i jej QBLOK pamPOCZ
 umieścimy gdzieś przed stosem pamięci
 ze względu na wzór w definicji stosu
*/

        QBLOK 0c17000  ;stos pamięci
        DS    0        ;przed stosem
stosMAX DS    [maksN*(pamKON-pamPOCZ)] 0
stosDNO EQU   *

        KONIEC
---
```

```
silnia  EQU  *
      ! B5 = (B5-(pamKON-pamPOCZ))
        [archN] = A             ;A=N na stos
        SKD  silnia2
        B7 = [jeden] ..silnia9  ;B7=0!=1
silnia2 EQU  *
        A  = B6                 ;B6 na stos
        [archB6] = A
        A  = [archN]            ;N ze stosu
        A  = A - [jeden]        ;N=N-1
        SKSB B6,silnia          ;B7=(N-1)!
        A  = [archN]            ;N ze stosu
        M=B7, M, AM=A*M         ;B7=N!
        A  = [archB6]
        B6 = A                  ;B6 ze stosu
        A  = [archN]            ;A ze stosu
silnia9 EQU  *
        B5 = (B5+(pamKON-pamPOCZ))
        NIC  0 ..B6

jeden   DS   1

pamPOCZ QBLOK B5       ;pamięć robocza silni
archB6  DS    0        ;przechowalnia B6
archN   DS    0        ;przechowalnia N
pamKON  EQU   *
```

Podprogramy rekursywne najlepiej wywoływać rozkazem SKSB, jako że adres powrotny (ślad) z nadrzędnego wywołania nie może być zniszczony przez ślad kolejnego rekursywnego wywołania.

Podprogram rekursywny musi dynamicznie przydzielać sobie pamięć roboczą, np. na stosie. Oszacowanie wielkości stosu dla funkcji silnia jest tu proste – jeśli wynik ma się zmieścić w rejestrze B7, to wielkość argumentu nie może przekroczyć liczby 14. Podprogram wpisuje na stos 14 segmentów pamPOCZ...pamKON dla N=14, oraz dodatkowo jedno słowo dla N=0.

Przykład zakłada, że funkcja silnia nie zmienia zawartości żadnego rejestru prócz wynikowego B7. Widać, jak wiele dodatkowych rozkazów zabierają operacje organizacyjne. Nawet zrezygnowanie z odtwarzania wartości akumulatora A nie przyniesie wielkich oszczędności. Adres powrotny we wszystkich wywołaniach funkcji silnia, prócz wywołania z programu głównego, jest ten sam, a mimo to rozrzutnie zajmuje miejsca na stosie. Normalnie funkcja ta powinna być zwykłą pętlą, a przy takim ograniczeniu argumentu należałoby ją nawet zastąpić tablicą 15 z góry policzonych wyników, bo kod tego podprogramu jest dłuższy – zajmuje 17 słów nie licząc stosu i wywołania.

W przypadku niektórych funkcji rekurencyjnych oszacowanie głębokości rekursywnych wywołań może być trudne. Ponieważ dynamiczna kontrola przepełnienia stosu kosztuje jakże cenne czas i wielkość, należy próbować oszacować z góry wymaganą wielkość stosu na podstawie dopuszczalnych danych.

Należy też unikać pułapek, jak np. w programie rekursywnym obliczania elementu ciągu Fibonacciego wg wzoru F(0)=0, F(1)=1, F(n)=F(n-2)+F(n-1), gdzie na ogromny czas pracy wpływa wielokrotne obliczanie tych samych wartości w poszczególnych gałęziach rozwinięcia funkcji.

Rekursywność podprogramu może być ukryta w sekwencji wywołań, np. podprogram P1 wywołuje P2, podprogram P2 wywołuje P3, … itd. aż któryś z nich znowu wywołuje P1. W tej sytuacji nie tylko P1 jest rekursywny, ale prawdopodobnie również P2, P3, …

ZALECENIE:

Jeśli można zmienić algorytm rekurencyjny na iteracyjny, to należy tak zrobić.

ZMIANY oznaczeń, symboli i mnemoników w stosunku do oryginalnych

Według		*Znaczenie*
AsmC	*[2],[3]*	
AC	A*	Akumulator zmiennoprzecinkowy składający się akumulatora mantysy Am (czyli A) i akumulatora cechy Ac [1]
&	∧	Operator koniunkcji [2]
^	÷	Operator różnicy symetrycznej [2]
*	✠	Operator mnożenia [2]
[N]	n	Oznaczenie argumentu w komórce pamięci o adresie N [3]
WE 0	We1	Rozkaz czytania o kodzie TPG :026, z czytnika 5-kanałowego nr 1 [4]
WE 1	We2	Rozkaz czytania o kodzie TPG :126, z dalekopisu [4]
WE 2		Rozkaz czytania o kodzie TPG :226, z czytnika 5-kanałowego nr 2 [4]
WY 5	Wy1	Rozkaz pisania o kodzie TPG :526, na perforator 5-kanałowy nr 1 [4]
WY 6	Wy2	Rozkaz pisania o kodzie TPG :626, na dalekopis [4]
WE 8	We1'	Rozkaz czytania o kodzie TPG :066, z czytnika 8-kanałowego nr 1 [4]
WE 10		Rozkaz czytania o kodzie TPG :266, z czytnika 8-kanałowego nr 2 [4]
WY 13	Wy1'	Rozkaz pisania o kodzie TPG :566, na perforator 8-kanałowy nr 1 [4]
LL	Lw	Rozkaz przesunięcia Logicznego w Lewo [5]
LP	Pr	Rozkaz przesunięcia Logicznego w Prawo [5]
SKBG		Rozkaz SKoku gdy Braku Gotowości urządzenia wejściowego [6]
SKV	SkNd	Rozkaz SKoku gdy nadmiar stałoprzecinkowy [7]
SKLC --	KC-	Rozkaz SKoku gdy Licznik Cykli był niezerowy (koniec cyklu z minusem) [8]
SKLC ++	KC+	Rozkaz SKoku gdy Licznik Cykli był niezerowy (koniec cyklu z plusem) [8]
BF 61	BF1	Rozkaz przesyłania blokowego z Bębna na ścieżkę Ferrytową nr 1 [9]
BF 62	BF2	Rozkaz przesyłania blokowego z Bębna na ścieżkę Ferrytową nr 2 [9]
FB 61	FB1	Rozkaz przesyłania blokowego ze ścieżki Ferrytowej nr 1 na Bębnową [9]
FB 62	FB2	Rozkaz przesyłania blokowego ze ścieżki Ferrytowej nr 2 na Bębnową [9]
NIC		Rozkaz NIC NIE RÓB o kodzie TPG :766 [10]
SKSB		Rozkaz SKoku ze Śladem w rejestrze B [11]
SKNBG		Rozkaz SKoku gdy Nie Brak Gotowości [12]
SKND		Rozkaz SKoku gdy Nie Dodatnie [12]
SKNU		Rozkaz SKoku gdy Nie Ujemne [12]
SKNV		Rozkaz SKoku gdy Nie Nadmiar [12]
SKNZ		Rozkaz SKoku gdy Nie Zero [12]
SKNLC --		Rozkaz SKoku gdy Licznik Cykli był zerowy [8] [12]
SKNLC ++		Rozkaz SKoku gdy Licznik Cykli był zerowy [8] [12]
SKM		Rozkaz SKoku gdy Mniejsze [13]
SKR		Rozkaz SKoku gdy Równe [13]
SKW		Rozkaz SKoku gdy Większe [13]
SKNM		Rozkaz SKoku gdy Nie Mniejsze [13]
SKNR		Rozkaz SKoku gdy Nie Równe [13]
SKNW		Rozkaz SKoku gdy Nie Większe [13]
SKMR		Rozkaz SKoku gdy Mniejsze lub Równe [13]
SKWR		Rozkaz SKoku gdy Większe lub Równe [13]

[1] Żadne rozkazy nie powołują się na oznaczenie Am akumulatora mantysy, ani na oznaczenie Ac rejestru cechy. Zatem symbol AM oznaczający rejestr długi będący połączeniem rejestrów A i B7 (M) nie koliduje z oznaczeniem Am rejestru mantysy, a symbol AC oznaczający akumulator zmiennoprzecinkowy jako parę rejestrów (Am,Ac) nie koliduje z oznaczeniem Ac rejestru cechy.

[2] Operatory &, ^ i * są powszechnie stosowane w językach programowania do zapisu koniunkcji, różnicy symetrycznej i mnożenia. Dodając argument o braku niektórych z tych symboli w wielu fontach, należy przyjąć za zasadną rezygnację z oryginalnych symboli ∧ (koniunkcji), ÷ (różnicy symetrycznej) i ✠ lub · (mnożenia) stosowanych w dokumentacji ODRY.

[3] Nawiasy kwadratowe jako oznaczenie argumentu w komórce pamięci są stosowane w niektórych asemblerach. Tutaj dodatkowym uzasadnieniem jest brak rozróżnienia małych i wielkich liter w nazwach symboli.

[4] Oryginalne mnemoniki rozkazów we/wy zmieniono tak, by wartość triady T kodu TPG rozkazu była argumentem instrukcji, jak również z powodu braku informacji o mnemoniku rozkazu czytania z drugiego czytnika. Mnemoniki rozkazów we/wy 8-kanałowego powstały przez analogię do 5-kanałowych, ale z argumentem powiększonym o 8.

[5] Mnemoniki rozkazów przesunięć logicznych zmieniono tak, by nie było wątpliwości, że są to przesunięcia logiczne, a nie arytmetyczne. Wzorem są oznaczenia w innych asemblerach.

[6] Mnemonik rozkazu skoku przy braku gotowości urządzenia utworzono z braku takiej informacji.

[7] Mnemonik rozkazu skoku przy nadmiarze zmieniono z SkNd na SKV, gdyż kolidowałby ze „skokiem przy nie dodatnim”. Nadmiar często oznacza się właśnie literą V.

[8] Oznaczenia -- i ++ w rozkazach SKLC zgodne są z operatorami dekrementacji i inkrementacji w innych językach. Symbol ++ kojarzy się też z symbolem następnego adresu w instrukcjach języka PODSTAWOWEGO. Mnemoniki KC- i KC+ zmieniono na SKLC ze wskazaniem dekrementacji lub inkrementacji licznika tak, by tak jak inne rozkazy skoków mówiły o skoku pod adres N (AR), a nie o skoku pod adres K (NR). Być może naruszono ducha ówczesnych czasów, ale jest to zgodne z innymi asemblerami.

[9] W rozkazach przesyłania blokowego numer ścieżki ferrytowej uzupełniono tak, by był pełnym numerem ścieżki pamięci, dzięki czemu łatwiej orientować się, gdzie jest pamięć ferrytowa. Jednocześnie wyodrębniono go w argument instrukcji. Jest to o tyle istotne, że przy pomocy tych rozkazów można przerzucać zawartość ścieżek na dowolne inne, co AsmC akceptuje zezwalając na dowolne przeadresowywanie przy pomocy instrukcji NAF-KOF.

[10] Instrukcja NIC generuje rozkaz o kodzie TPG :766, który jest niezdefiniowany i jako taki nie wykonuje żadnych operacji, nawet ustawienia warunków, prócz przejścia do następnego rozkazu. Został wybrany spośród wielu rozkazów niezdefiniowanych jako najwłaściwszy do powrotu z podprogramu bez niszczenia ustawionego warunku, oraz bez posiłkowania się śladem zapisywanym w pamięci operacyjnej.

[11] Instrukcja SKSB jest synonimem rozkazu o kodzie TPG :032. Rozkaz SKSB B6,podprog..++ jest zgodny z ogólnym sposobem zapisu rozkazów i czytelniejszy od równoważnego B6=++..podprog, oraz ma mnemonik podobny do skoku SKS podprog..++ ze śladem w pamięci operacyjnej.

[12] Instrukcje skoków z zanegowanymi warunkami powstają poprzez zamianę miejscami wartości części adresowych zwykłych rozkazów skoków warunkowych. Skutkiem tego jest zamieniona dopuszczalność rejestrów B-modyfikacji w adresach skoków.

[13] Dodatkowe mnemoniki instrukcji skoków warunkowych, wprowadzone w AsmC w celu ułatwienia rozumienia sensu rozkazów skoków po odejmowaniach pełniących rolę porównań, są synonimami rozkazów: SKU, SKZ, SKD, oraz SKNU, SKNZ, SKND.

WYKAZ operatorów, separatorów i wyróżników

Oznaczenie	*Znaczenie*	*Przykłady*
≡ *(LF)*	Koniec wiersza	
◡ *(odstęp)*	Separator atomów (leksemów)	`SKLC B2-- ETYK`
_	Separator – grupowanie cyfr liczby	`DS 123_456_789`
;	Separator – początek komentarza liniowego	`A=5 ;komentarz`
,	Separator części rozkazu lub instrukcji	`M=[ETYK+B3],M,A=A/M`
=	Separator rozkazu wyznaczający przypisanie (posłanie wyniku)	`A=B7`
==	Operator relacji równości (EQ) w wyrażeniu adresowym	`AIF aaa==bbb`
?	Operator testu istnienia nazwy w wyrażeniu adresowym	`AIF ?funkcja`
#	Generator nazw lokalnych	`#et SKZ #błąd`
%	Operator reszty z dzielenia (MOD) w wyrażeniu adresowym	`str EQU (adr-0:0)%128`
| |	Separator rozkazu obliczania wartości bezwzględnej (ABS)	`A=|[zmienna]|`
|	Operator alternatywy bitowej (BOR) w wyrażeniu adresowym	`odd EQU adres|1`
||	Operator alternatywy logicznej (OR) w wyrażeniu adresowym	`albo EQU x==0&&y||z`
^	Separator rozkazu obliczania różnicy symetrycznej (BXOR)	`A=A^14`
^	Operator różnicy symetrycznej (BXOR) w wyraż. adresowym	`AIF (x==0)^(y==0)`
[ ]	Separator odwołania do komórki pamięci w rozkazie	`A=[zmienna]`
[ ]	Separator deklaracji krotności słowa lub tekstu	`DS [50]0, [3]'abc'`
<	Operator relacji mniejszości (LT) w wyrażeniu adresowym	`AIF aaa<bbb`
<=	Operator relacji mniejsze równe (LE) w wyrażeniu adresowym	`AIF aaa<=bbb`
<<	Separator rozkazu przesunięcia arytmetycznego w lewo	`A=A<<5`
<<	Operator przesunięcia w lewo w wyrażeniu adresowym	`k16 EQU z<<5`
<<<	Separator rozkazu przesunięcia logicznego w lewo	`A=A<<<5`
<>	Operator relacji nierówności (NE) w wyrażeniu adresowym	`AIF aaa<>bbb`
<|	Operator wyboru mniejszej (MIN) w wyrażeniu adresowym	`min EQU 5<|x<|y`
>	Operator relacji większości (GT) w wyrażeniu adresowym	`AIF aaa>bbb`
>=	Operator relacji większe równe (GE) w wyrażeniu adresowym	`AIF aaa>=bbb`
>>	Separator rozkazu przesunięcia arytmetycznego w prawo	`A=A>>17`
>>	Operator przesunięcia w prawo w wyrażeniu adresowym	`nrś EQU z>>7`
>>>	Separator rozkazu przesunięcia logicznego w prawo	`A=A>>>13`
>><	Separator rozkazu przesunięcia cyklicznego w prawo	`A=A>><5`
>|	Operator wyboru większej (MAX) w wyrażeniu adresowym	`max EQU 5>|x>|y`

( )	Operator (nawiasy) podnoszący priorytet w wyraż. adresowym	`ile  EQU  (kon-pocz)*3`
:	Separator – wyróżnik rozkazu w postaci TPG	`:726 00000 00000 00`
:	Operator przeadresowania w wyrażeniu adresowym	`SKD  (61*128):ETYK`
' '	Separator – ogranicznik tekstu	`DS  'tekst'`
*	Separator rozkazu mnożenia stałoprzecinkowego	`M=B3,M,AM=A*M`
*	Separator rozkazu mnożenia zmiennoprzecinkowego	`AC=AC*B7`
*	Symbol bieżącego adresu	`etyk EQU  *`
&	Separator rozkazu obliczania koniunkcji (BAND)	`A=A&7`
&	Operator koniunkcji bitowej (BAND) w wyrażeniu adresowym	`wyc  EQU  adres&1`
&&	Operator koniunkcji logicznej (AND) w wyrażeniu adresowym	`AIF  (x==2)&&(y==2)`
!	Korektor operacji rozkazu ze względu na B-modyfikację	`et ! B5 = (B5-8)`
/	Separator rozkazu dzielenia stałoprzecinkowego	`M=B3,M,A=A/M`
/	Separator rozkazu dzielenia zmiennoprzecinkowego	`AC=AC/B7`
//	Separator rozkazu dzielenia stałoprzecinkowego długiego	`A=A//M,82`
//	Znak specjalny / w tekście dłubanym	`DS  T2'//'`
/R	Znak specjalny CR w tekście dłubanym	`DS  T2'/R'`
/N	Znak specjalny LF w tekście dłubanym	`DS  T2'/N'`
/L	Znak specjalny LS w tekście dłubanym	`DS  T2'/L'`
/F	Znak specjalny FS w tekście dłubanym	`DS  T2'/F'`
/Z	Znak specjalny NU w tekście dłubanym	`DS  T2'/Z'`
/Ccc	Znak o kodzie ósemkowym cc w tekście dłubanym	`DS  T2'/c37'`
/Xxx	Znak o kodzie szesnastkowym xx w tekście dłubanym	`DS  T2'/x1F'`
/Ddd	Znak o kodzie dziesiętnym dd w tekście dłubanym	`DS  T2'/d31'`
/* */	Separator – ogranicznik komentarza blokowego	`/* komentarz */`
.	Separator – kropka ułamkowa w liczbie zmiennoprzecinkowej	`DS  123.45`
.	Separator – usunięcie słowa zerowego kończącego tekst	`DS  T0'tekst'.`
..	Separator rozkazu – wskazanie adresu następnego rozkazu	`A=B1 ..ETYK`
-	Separator rozkazu zmiany znaku liczby stałoprzecinkowej	`A=-[zmienna]`
-	Separator rozkazu odejmowania stałoprzecinkowego	`A=[zmienna]-A`
-	Separator rozkazu odejmowania zmiennoprzecinkowego	`AC=AC-B7`
-	Separator – znak ujemnej liczby stałoprzecinkowej ze znakiem	`DS  -274_877_906_944`
-	Separator – znak ujemnej liczby zmiennoprzecinkowej	`DS  -123.45e3`
-	Separator – znak ujemnej cechy liczby zmiennoprzecinkowej	`DS  123.45e-3`
-	Operator odejmowania w wyrażeniu adresowym	`B1=(adres-1)`
--	Separator rozkazu dekrementacji rejestru B-modyfikacji	`SKLC B2--,ETYK`
--	Symbol bieżącego adresu pomniejszonego o 1	`SKLC B2--,--`
+	Separator rozkazu dodawania stałoprzecinkowego	`A=A+[zmienna]`

+	Separator rozkazu dodawania zmiennoprzecinkowego	`AC=AC+B7`
+	Separator – znak dodatniej liczby stałoprzecinkowej ze znakiem	`DS  +274_877_906_943`
+	Separator – znak dodatniej liczby zmiennoprzecinkowej	`DS  +123.45e3`
+	Separator – znak dodatniej cechy liczby zmiennoprzecinkowej	`DS   123.45e+3`
+	Operator dodawania w wyrażeniu adresowym	`B1=(adres+1)`
++	Separator rozkazu inkrementacji rejestru B-modyfikacji	`SKLC B2++,ETYK`
++	Symbol bieżącego adresu powiększonego o 1	`SKU  ++ ..ETYK`
0B	Wyróżnik liczby w notacji dwójkowej	`A=A+0b10011`
0C	Wyróżnik liczby w notacji ósemkowej	`A=A+0c23`
0X	Wyróżnik liczby w notacji szesnastkowej	`A=A+0x13`
0'	Wyróżnik tekstu topornego	`DS  0'AB',  T0'AB'`
1'	Wyróżnik tekstu ciosanego	`DS  1'AB'1  T1'AB'1`
2'	Wyróżnik tekstu dłubanego	`DS  2'/r/n' T2'/r/n'`
6'	Wyróżnik tekstu 6-bitowego	`DS  6'AB',  T6'AB'`
7'	Wyróżnik tekstu 7-bitowego	`DS  7'AB',  T7'AB'`
9'	Wyróżnik tekstu odrębnego	`DS  9'AB',  T9'AB'`
E	Wyróżnik rzędu (cechy) liczby zmiennoprzecinkowej dziesiętnej	`DS  123e9`
P	Wyróżnik cechy liczby zmiennoprzecinkowej bitowej	`DS  0x123p9`
K	Wyróżnik skali liczby stałoprzecinkowej	`DS  123k21`
S	Wyróżnik atrybutu długości (liczby słów) tekstu	`DS  S2'ABCD'`
T	Wyróżnik atrybutu typu tekstu	`DS  T2'ABCD'`

WYKAZ rozkazów maszynowych

W zamieszczonych tu tabelkach wypisano wszystkie rozkazy ODRY. Rozkazy są uporządkowane rosnąco wg triad G, P i T kodu TPG. W poszczególnych kolumnach znajdują się:

- kod TPG rozkazu, złożony z trzech triad T, P i G.
- notacja rozkazu w języku asemblera AsmC
- kody warunków po wykonaniu rozkazu

Kolorem niebieskim oznaczono rozkazy przesyłania blokowego dostępne dopiero w ODRZE 1013.

Kolorem zielonym oznaczono rozkazy dostępne w ODRZE UMCS, tj. wejścia/wyjścia 8-kanałowego (numery tych urządzeń są o 8 większe od odpowiedników 5-kanałowych) i skok przy braku gotowości, oraz podkreślono, że w miejsce konwertera analogowo-cyfrowego zainstalowano drugi czytnik.

Kolorem ceglanym zaznaczono rozkaz niezdefiniowany (o mnemoniku Nic), ale arbitralnie wybrany jako najbardziej odpowiedni do wykonywania skoków bez ustawiania warunków.

Puste pozycje są to rozkazy niezdefiniowane, działające dokładnie tak samo jak rozkaz NIC: nie wykonują żadnej operacji, nie ustawiają warunków, a jedynym skutkiem jest przejście do następnego rozkazu wyznaczonego przez adres w części NR (K) rozkazu. Rozkazy niezdefiniowane podlegają B-modyfikacji tak jak wszystkie inne.

W wykazie rozkazów zastosowano symboliczny zapis funkcji rozkazów podobny do instrukcji języka JavaScript lub C, a w nim następujące oznaczenia:

`x`	wartość wchodząca na drugie wejście sumatora
`s`	wartość wynikowa z sumatora
`A`	zawartość rejestru akumulatora
`B`	zawartość rejestru B, tj. B0, B1, B2, B3, B4, B5, B6 lub B7 (w rozkazie SkBG chodzi też o nr rejestru)
`M`	zawartość rejestru B7
`R`	rejestr rozkazów, do którego pobrana lub posłana wartość staje się rozkazem do wykonania
`AM`	zawartość pary rejestrów A i M (tj. bity 0..38 rejestru A, oraz bity 1..38 rejestru M=B7)
`AC`	zawartość pary rejestrów: Am=A (mantysa), oraz Ac (cecha) liczby zmiennoprzecinkowej
`n`	wartość pola AR (N) rozkazu (argument bezpośredni)
`[n]`	zawartość komórki pamięci o adresie n
`Ω`	wskazanie, że ustawiana jest wartość rejestru zaokrągleń
`M=#`	wskazanie, że po wykonaniu rozkazu pozostanie pewna wartość w rejestrze roboczym M=B7
`uzv`	wskazanie ustawiania kodu warunku: u – ujemne, z – zero, v – nadmiar stałoprzecinkowy: mała litera – dany warunek jest ustawiany, . – że ten warunek nie jest ustawiany, duża litera – że wystąpi dany warunek, 0 – że ten warunek nie wystąpi
`uzvn`	jw. a ponadto drugie wskazanie ustawiania nadmiaru: n – może wystąpić nadmiar w fazie mnożenia lub dzielenia stałoprzecinkowego, / – może wystąpić dzielenie przez zero, ● – nastąpi Stop, * – może wystąpić nadmiar zmiennoprzecinkowy
`PAO`	wskazanie, że wartość wynikowa nie jest posyłana do pamięci – w rozkazy TPG=`x11` i TPG=`x51`
`BON`	wskazanie, że nie wykonuje się zaokrąglenia i normalizacji – operacje zmiennoprzecinkowe TPG=`2x7`

:000	0	0Z0
:100	-0	0Z0
:200	|0|	0Z0
:300	A-0	uz0
:400	A+0	uz0
:500	0-A	uzv
:600	A&0	0Z0
:700	A^0	uz0
:010	[n]	uz0
:110	-[n]	uzv
:210	|[n]|	0zv
:310	A-[n]	uzv
:410	A+[n]	uzv
:510	[n]-A	uzv
:610	A&[n]	uz0
:710	A^[n]	uz0
:020	B	uz0
:120	-B	uzv
:220	|B|	0zv
:320	A-B	uzv
:420	A+B	uzv
:520	B-A	uzv
:620	A&B	uz0
:720	A^B	uz0
:030	(n)	0z0
:130	-(n)	uz0
:230	|(n)|	0z0
:330	A-(n)	uzv
:430	A+(n)	uzv
:530	(n)-A	uzv
:630	A&(n)	uz0
:730	A^(n)	uz0
:040	A= 0	0Z0
:140	A=-0	0Z0
:240	A=|0|	0Z0
:340	A=A-0	uz0
:440	A=A+0	uz0
:540	A=0-A	uzv
:640	A=A&0	0Z0
:740	A=A^0	uz0
:050	A= [n]	uz0
:150	A=-[n]	uzv
:250	A=|[n]|	0zv
:350	A=A-[n]	uzv
:450	A=A+[n]	uzv
:550	A=[n]-A	uzv
:650	A=A&[n]	uz0
:750	A=A^[n]	uz0
:060	A= B	uz0
:160	A=-B	uzv
:260	A=|B|	0zv
:360	A=A-B	uzv
:460	A=A+B	uzv
:560	A=B-A	uzv
:660	A=A&B	uz0
:760	A=A^B	uz0
:070	A= (n)	0z0
:170	A=-(n)	uz0
:270	A=|(n)|	0z0
:370	A=A-(n)	uzv
:470	A=A+(n)	uzv
:570	A=(n)-A	uzv
:670	A=A&(n)	uz0
:770	A=A^(n)	uz0

:001	[n]= 0	0Z0
:101	[n]=-0	0Z0
:201	[n]=|0|	0Z0
:301	[n]=A-0	uz0
:401	[n]=A+0	uz0
:501	[n]=0-A	uzv
:601	[n]=A&0	0Z0
:701	[n]=A^0	uz0
:011	[n]= [n]	uz0 PAO
:111	[n]=-[n]	uzv PAO
:211	[n]=|[n]|	0zv PAO
:311	[n]=A-[n]	uzv PAO
:411	[n]=A+[n]	uzv PAO
:511	[n]=[n]-A	uzv PAO
:611	[n]=A&[n]	uz0 PAO
:711	[n]=A^[n]	uz0 PAO
:021	[n]= B	uz0
:121	[n]=-B	uzv
:221	[n]=|B|	0zv
:321	[n]=A-B	uzv
:421	[n]=A+B	uzv
:521	[n]=B-A	uzv
:621	[n]=A&B	uz0
:721	[n]=A^B	uz0
:031	[n]= (n)	0z0
:131	[n]=-(n)	uz0
:231	[n]=|(n)|	0z0
:331	[n]=A-(n)	uzv
:431	[n]=A+(n)	uzv
:531	[n]=(n)-A	uzv
:631	[n]=A&(n)	uz0
:731	[n]=A^(n)	uz0
:041	[n]=A= 0	0Z0
:141	[n]=A=-0	0Z0
:241	[n]=A=|0|	0Z0
:341	[n]=A=A-0	uz0
:441	[n]=A=A+0	uz0
:541	[n]=A=0-A	uzv
:641	[n]=A=A&0	0Z0
:741	[n]=A=A^0	uz0
:051	[n]=A= [n]	uz0 PAO
:151	[n]=A=-[n]	uzv PAO
:251	[n]=A=|[n]|	0zv PAO
:351	[n]=A=A-[n]	uzv PAO
:451	[n]=A=A+[n]	uzv PAO
:551	[n]=A=[n]-A	uzv PAO
:651	[n]=A=A&[n]	uz0 PAO
:751	[n]=A=A^[n]	uz0 PAO
:061	[n]=A= B	uz0
:161	[n]=A=-B	uzv
:261	[n]=A=|B|	0zv
:361	[n]=A=A-B	uzv
:461	[n]=A=A+B	uzv
:561	[n]=A=B-A	uzv
:661	[n]=A=A&B	uz0
:761	[n]=A=A^B	uz0
:071	[n]=A= (n)	0z0
:171	[n]=A=-(n)	uz0
:271	[n]=A=|(n)|	0z0
:371	[n]=A=A-(n)	uzv
:471	[n]=A=A+(n)	uzv
:571	[n]=A=(n)-A	uzv
:671	[n]=A=A&(n)	uz0
:771	[n]=A=A^(n)	uz0

:002	B= 0		0Z0
:102	B=-0		0Z0
:202	B=|0|		0Z0
:302	B=A-0		uz0
:402	B=A+0		uz0
:502	B=0-A		uzv
:602	B=A&0		0Z0
:702	B=A^0		uz0
:012	B= [n]		uz0
:112	B=-[n]		uzv
:212	B=|[n]|		0zv
:312	B=A-[n]		uzv
:412	B=A+[n]		uzv
:512	B=[n]-A		uzv
:612	B=A&[n]		uz0
:712	B=A^[n]		uz0
:022	B= B		uz0
:122	B=-B		uzv
:222	B=|B|		0zv
:322	B=A-B		uzv
:422	B=A+B		uzv
:522	B=B-A		uzv
:622	B=A&B		uz0
:722	B=A^B		uz0
:032	B= (n)	SKSB B,k	0z0
:132	B=-(n)		uz0
:232	B=|(n)|		0z0
:332	B=A-(n)		uzv
:432	B=A+(n)		uzv
:532	B=(n)-A		uzv
:632	B=A&(n)		uz0
:732	B=A^(n)		uz0
:042	B=A= 0		0Z0
:142	B=A=-0		0Z0
:242	B=A=|0|		0Z0
:342	B=A=A-0		uz0
:442	B=A=A+0		uz0
:542	B=A=0-A		uzv
:642	B=A=A&0		0Z0
:742	B=A=A^0		uz0
:052	B=A= [n]		uz0
:152	B=A=-[n]		uzv
:252	B=A=|[n]|		0zv
:352	B=A=A-[n]		uzv
:452	B=A=A+[n]		uzv
:552	B=A=[n]-A		uzv
:652	B=A=A&[n]		uz0
:752	B=A=A^[n]		uz0
:062	B=A= B		uz0
:162	B=A=-B		uzv
:262	B=A=|B|		0zv
:362	B=A=A-B		uzv
:462	B=A=A+B		uzv
:562	B=A=B-A		uzv
:662	B=A=A&B		uz0
:762	B=A=A^B		uz0
:072	B=A= (n)		0z0
:172	B=A=-(n)		uz0
:272	B=A=|(n)|		0z0
:372	B=A=A-(n)		uzv
:472	B=A=A+(n)		uzv
:572	B=A=(n)-A		uzv
:672	B=A=A&(n)		uz0
:772	B=A=A^(n)		uz0

:003	R= 0	0Z0
:103	R=-0	0Z0
:203	R=|0|	0Z0
:303	R=A-0	uz0
:403	R=A+0	uz0
:503	R=0-A	uzv
:603	R=A&0	0Z0
:703	R=A^0	uz0
:013	R= [n]	uz0
:113	R=-[n]	uzv
:213	R=|[n]|	0zv
:313	R=A-[n]	uzv
:413	R=A+[n]	uzv
:513	R=[n]-A	uzv
:613	R=A&[n]	uz0
:713	R=A^[n]	uz0
:023	R= B	uz0
:123	R=-B	uzv
:223	R=|B|	0zv
:323	R=A-B	uzv
:423	R=A+B	uzv
:523	R=B-A	uzv
:623	R=A&B	uz0
:723	R=A^B	uz0
:033	R= (n)	0z0
:133	R=-(n)	uz0
:233	R=|(n)|	0z0
:333	R=A-(n)	uzv
:433	R=A+(n)	uzv
:533	R=(n)-A	uzv
:633	R=A&(n)	uz0
:733	R=A^(n)	uz0
:043	R=A= 0	0Z0
:143	R=A=-0	0Z0
:243	R=A=|0|	0Z0
:343	R=A=A-0	uz0
:443	R=A=A+0	uz0
:543	R=A=0-A	uzv
:643	R=A=A&0	0Z0
:743	R=A=A^0	uz0
:053	R=A= [n]	uz0
:153	R=A=-[n]	uzv
:253	R=A=|[n]|	0zv
:353	R=A=A-[n]	uzv
:453	R=A=A+[n]	uzv
:553	R=A=[n]-A	uzv
:653	R=A=A&[n]	uz0
:753	R=A=A^[n]	uz0
:063	R=A= B	uz0
:163	R=A=-B	uzv
:263	R=A=|B|	0zv
:363	R=A=A-B	uzv
:463	R=A=A+B	uzv
:563	R=A=B-A	uzv
:663	R=A=A&B	uz0
:763	R=A=A^B	uz0
:073	R=A= (n)	0z0
:173	R=A=-(n)	uz0
:273	R=A=|(n)|	0z0
:373	R=A=A-(n)	uzv
:473	R=A=A+(n)	uzv
:573	R=A=(n)-A	uzv
:673	R=A=A&(n)	uz0
:773	R=A=A^(n)	uz0

:004	M=0, M, AM=A*M	0Z00	Ω
:104	M=0,-M, AM=A*M	0Z00	Ω
:204	M=0,|M|,AM=A*M	0Z00	Ω
:304	M=0,A-M,AM=A*M	0Z00	Ω
:404	M=0,A+M,AM=A*M	0Z00	Ω
:504	M=0,M-A,AM=A*M	uzv0	Ω
:604	M=0,A&M,AM=A*M	0Z00	Ω
:704	M=0,A^M,AM=A*M	0Z00	Ω
:014	M=[n], M, AM=A*M	uz0n	Ω
:114	M=[n],-M, AM=A*M	uzvn	Ω
:214	M=[n],|M|,AM=A*M	uzvn	Ω
:314	M=[n],A-M,AM=A*M	uzvn	Ω
:414	M=[n],A+M,AM=A*M	uzvn	Ω
:514	M=[n],M-A,AM=A*M	uzvn	Ω
:614	M=[n],A&M,AM=A*M	uz0n	Ω
:714	M=[n],A^M,AM=A*M	uz0n	Ω
:024	M=B, M, AM=A*M	uz0n	Ω
:124	M=B,-M, AM=A*M	uzvn	Ω
:224	M=B,|M|,AM=A*M	uzvn	Ω
:324	M=B,A-M,AM=A*M	uzvn	Ω
:424	M=B,A+M,AM=A*M	uzvn	Ω
:524	M=B,M-A,AM=A*M	uzvn	Ω
:624	M=B,A&M,AM=A*M	uz0n	Ω
:724	M=B,A^M,AM=A*M	uz0n	Ω
:034	M=n, M, AM=A*M	uz00	Ω
:134	M=n,-M, AM=A*M	uz00	Ω
:234	M=n,|M|,AM=A*M	uz00	Ω
:334	M=n,A-M,AM=A*M	uzv0	Ω
:434	M=n,A+M,AM=A*M	uzv0	Ω
:534	M=n,M-A,AM=A*M	uzv0	Ω
:634	M=n,A&M,AM=A*M	uz00	Ω
:734	M=n,A^M,AM=A*M	uz00	Ω
:044	M=0,A= M, AM=A*M	0Z00	Ω
:144	M=0,A=-M, AM=A*M	0Z00	Ω
:244	M=0,A=|M|,AM=A*M	0Z00	Ω
:344	M=0,A=A-M,AM=A*M	0Z00	Ω
:444	M=0,A=A+M,AM=A*M	0Z00	Ω
:544	M=0,A=M-A,AM=A*M	uzv0	Ω
:644	M=0,A=A&M,AM=A*M	0Z00	Ω
:744	M=0,A=A^M,AM=A*M	0Z00	Ω
:054	M=[n],A= M, AM=A*M	uz0n	Ω
:154	M=[n],A=-M, AM=A*M	uzvn	Ω
:254	M=[n],A=|M|,AM=A*M	uzvn	Ω
:354	M=[n],A=A-M,AM=A*M	uzvn	Ω
:454	M=[n],A=A+M,AM=A*M	uzvn	Ω
:554	M=[n],A=M-A,AM=A*M	uzvn	Ω
:654	M=[n],A=A&M,AM=A*M	uz0n	Ω
:754	M=[n],A=A^M,AM=A*M	uz0n	Ω
:064	M=B,A= M, AM=A*M	uz0n	Ω
:164	M=B,A=-M, AM=A*M	uzvn	Ω
:264	M=B,A=|M|,AM=A*M	uzvn	Ω
:364	M=B,A=A-M,AM=A*M	uzvn	Ω
:464	M=B,A=A+M,AM=A*M	uzvn	Ω
:564	M=B,A=M-A,AM=A*M	uzvn	Ω
:664	M=B,A=A&M,AM=A*M	uz0n	Ω
:764	M=B,A=A^M,AM=A*M	uz0n	Ω
:074	M=n,A= M, AM=A*M	uz00	Ω
:174	M=n,A=-M, AM=A*M	uz00	Ω
:274	M=n,A=|M|,AM=A*M	uz00	Ω
:374	M=n,A=A-M,AM=A*M	uzv0	Ω
:474	M=n,A=A+M,AM=A*M	uzv0	Ω
:574	M=n,A=M-A,AM=A*M	uzv0	Ω
:674	M=n,A=A&M,AM=A*M	uz00	Ω
:774	M=n,A=A^M,AM=A*M	uz00	Ω

:005	M=0, M, A=A/M	0Z0●	Ω
:105	M=0,-M, A=A/M	0Z0●	Ω
:205	M=0,|M|,A=A/M	0Z0●	Ω
:305	M=0,A-M,A=A/M	uz0●	Ω
:405	M=0,A+M,A=A/M	uz0●	Ω
:505	M=0,M-A,A=A/M	uzv●	Ω
:605	M=0,A&M,A=A/M	0Z0●	Ω
:705	M=0,A^M,A=A/M	uz0●	Ω
:015	M=[n], M, A=A/M	uz0/	Ω
:115	M=[n],-M, A=A/M	uzv/	Ω
:215	M=[n],|M|,A=A/M	uzv/	Ω
:315	M=[n],A-M,A=A/M	uzv/	Ω
:415	M=[n],A+M,A=A/M	uzv/	Ω
:515	M=[n],M-A,A=A/M	uzv/	Ω
:615	M=[n],A&M,A=A/M	uz0/	Ω
:715	M=[n],A^M,A=A/M	uz0/	Ω
:025	M=B, M, A=A/M	uz0/	Ω
:125	M=B,-M, A=A/M	uzv/	Ω
:225	M=B,|M|,A=A/M	uzv/	Ω
:325	M=B,A-M,A=A/M	uzv/	Ω
:425	M=B,A+M,A=A/M	uzv/	Ω
:525	M=B,M-A,A=A/M	uzv/	Ω
:625	M=B,A&M,A=A/M	uz0/	Ω
:725	M=B,A^M,A=A/M	uz0/	Ω
:035	M=n, M, A=A/M	uz0/	Ω
:135	M=n,-M, A=A/M	uz0/	Ω
:235	M=n,|M|,A=A/M	uz0/	Ω
:335	M=n,A-M,A=A/M	uzv/	Ω
:435	M=n,A+M,A=A/M	uzv/	Ω
:535	M=n,M-A,A=A/M	uzv/	Ω
:635	M=n,A&M,A=A/M	uz0/	Ω
:735	M=n,A^M,A=A/M	uz0/	Ω
:045	M=0,A= M, A=A/M	0Z0●	Ω
:145	M=0,A=-M, A=A/M	0Z0●	Ω
:245	M=0,A=|M|,A=A/M	0Z0●	Ω
:345	M=0,A=A-M,A=A/M	uz0●	Ω
:445	M=0,A=A+M,A=A/M	uz0●	Ω
:545	M=0,A=M-A,A=A/M	uzv●	Ω
:645	M=0,A=A&M,A=A/M	0Z0●	Ω
:745	M=0,A=A^M,A=A/M	uz0●	Ω
:055	M=[n],A= M, A=A/M	uz0/	Ω
:155	M=[n],A=-M, A=A/M	uzv/	Ω
:255	M=[n],A=|M|,A=A/M	uzv/	Ω
:355	M=[n],A=A-M,A=A/M	uzv/	Ω
:455	M=[n],A=A+M,A=A/M	uzv/	Ω
:555	M=[n],A=M-A,A=A/M	uzv/	Ω
:655	M=[n],A=A&M,A=A/M	uz0/	Ω
:755	M=[n],A=A^M,A=A/M	uz0/	Ω
:065	M=B,A= M, A=A/M	uz0/	Ω
:165	M=B,A=-M, A=A/M	uzv/	Ω
:265	M=B,A=|M|,A=A/M	uzv/	Ω
:365	M=B,A=A-M,A=A/M	uzv/	Ω
:465	M=B,A=A+M,A=A/M	uzv/	Ω
:565	M=B,A=M-A,A=A/M	uzv/	Ω
:665	M=B,A=A&M,A=A/M	uz0/	Ω
:765	M=B,A=A^M,A=A/M	uz0/	Ω
:075	M=n,A= M, A=A/M	0z0N	Ω
:175	M=n,A=-M, A=A/M	uz0/	Ω
:275	M=n,A=|M|,A=A/M	uz0N	Ω
:375	M=n,A=A-M,A=A/M	uzv/	Ω
:475	M=n,A=A+M,A=A/M	uzv/	Ω
:575	M=n,A=M-A,A=A/M	uzv/	Ω
:675	M=n,A=A&M,A=A/M	uz0/	Ω
:775	M=n,A=A^M,A=A/M	uz0/	Ω

:006			
:106			
:206			
:306			
:406			
:506			
:606			
:706			
:016	LL n	A=A<<<n	uz0
:116	LP n	A=A>>>n	uz0 Ω
:216	AL n	A=A<<n	uzv
:316	AP n	A=A>>n	uz0 Ω
:416	CP n	A=A><n	uz0 Ω
:516	ALD n	AM=AM<<n	uzv Ω
:616	APD n	AM=AM>>n	uz0 Ω
:716	DZD n	A=A//M,n	uzv/ Ω
:026	WE 0,n		uz.
:126	WE 1,n		uz.
:226	WE 2,n		uz.
:326	CZK n		uz.
:426	OKR n	A=A+OM	uzv
:526	WY 5,n		uz.
:626	WY 6,n		uz.
:726	STOP n		...
:036			
:136	BF 61,B,n		...
:236	BF 62,B,n		...
:336	FB 61,B,n		...
:436	FB 62,B,n		...
:536			
:636			
:736			
:046	SKZ n	SKR n	...
:146	SKU n	SKM n	...
:246	SKD n	SKW n	...
:346	SKV n		...
:446	SKBG B,n		...
:546	SKLC B--,n		...
:646	SKLC B++,n		...
:746	SKS n		...
:056			
:156			
:256			
:356			
:456			
:556			
:656			
:756			
:066	WE 8,n		uz.
:166			
:266	WE 10,n		uz.
:366			
:466			
:566	WY 13,n		uz.
:666			
:766	NIC n		...
:076			
:176			
:276			
:376			
:476			
:576			
:676			
:776			

:007		
:107	AC=[n], BON	uz.0
:207	[n]=AC, BON	uz.0
:307	AC=AC+[n],BON	uz.*
:407	AC=AC-[n],BON	uz.*
:507	AC=[n]-AC,BON	uz.*
:607	AC=AC*[n],BON	uz.* M=#
:707	AC=AC/[n],BON	uz/* M=#
:017		
:117	AC=[n], BO	uz.0
:217	[n]=AC, BO	uz.0 BON
:317	AC=AC+[n],BO	uz.*
:417	AC=AC-[n],BO	uz.*
:517	AC=[n]-AC,BO	uz.*
:617	AC=AC*[n],BO	uz.* M=#
:717	AC=AC/[n],BO	uz/* M=#
:027		
:127	AC=[n], BN	uz.0
:227	[n]=AC, BN	uz.0 BON
:327	AC=AC+[n],BN	uz.*
:427	AC=AC-[n],BN	uz.*
:527	AC=[n]-AC,BN	uz.*
:627	AC=AC*[n],BN	uz.* M=#
:727	AC=AC/[n],BN	uz/* M=#
:037		
:137	AC=[n]	uz.0
:237	[n]=AC	uz.0 BON
:337	AC=AC+[n]	uz.*
:437	AC=AC-[n]	uz.*
:537	AC=[n]-AC	uz.*
:637	AC=AC*[n]	uz.* M=#
:737	AC=AC/[n]	uz/* M=#
:047		
:147	AC=B, BON	uz.0
:247	B=AC, BON	uz.0
:347	AC=AC+B,BON	uz.*
:447	AC=AC-B,BON	uz.*
:547	AC=B-AC,BON	uz.*
:647	AC=AC*B,BON	uz.* M=#
:747	AC=AC/B,BON	uz/* M=#
:057		
:157	AC=B, BO	uz.0
:257	B=AC, BO	uz.0 BON
:357	AC=AC+B,BO	uz.*
:457	AC=AC-B,BO	uz.*
:557	AC=B-AC,BO	uz.*
:657	AC=AC*B,BO	uz.* M=#
:757	AC=AC/B,BO	uz/* M=#
:067		
:167	AC=B, BN	uz.0
:267	B=AC, BN	uz.0 BON
:367	AC=AC+B,BN	uz.*
:467	AC=AC-B,BN	uz.*
:567	AC=B-AC,BN	uz.*
:667	AC=AC*B,BN	uz.* M=#
:767	AC=AC/B,BN	uz/* M=#
:077		
:177	AC=B	uz.0
:277	B=AC	uz.0 BON
:377	AC=AC+B	uz.*
:477	AC=AC-B	uz.*
:577	AC=B-AC	uz.*
:677	AC=AC*B	uz.* M=#
:777	AC=AC/B	uz/* M=#

WYKAZ programów przydatnych z AsmC

Do programu asemblera AsmC poleca się dodatkowe programy i podprogramy pomocnicze – narzędziowe i demonstracyjne – których analiza pomoże też rozwiać jakieś niejasności:

„AsmC - asembler” – zasadniczy program asemblera

`„AsmC.pgm.pt5”` program ładowalny asemblera – kod PIĄTKOWY programu asemblera – jest to obraz taśmy perforowanej 5-kanałowej

---

„EDYTOR DALEKOPISOWY” – program w pamięci STAŁEJ komputera

`0c17654` program przydatny do konwersji programu źródłowego z postaci pliku .txt na tasiemkę perforowaną w kodzie ITA PL4 akceptowaną przez AsmC

„PIĄTKOWY-THETA” – pakiet programów konwersji taśm z kodem binarnym

`„PIĄTKOWY-THETA.asm.txt”` – program źródłowy w postaci pliku tekstowego

`„PIĄTKOWY-THETA.asm.pt5”` – program źródłowy w postaci taśmy w kodzie ITA PL4

`„PIĄTKOWY na THETA.pgm.pt5”` – program konwersji kodu PIĄTKOWEGO na THETA (postać ładowalna przykładowego wariantu programu konwersji)

`„THETA na PIĄTKOWY.pgm.pt5”` – program konwersji kodu THETA na PIĄTKOWY (postać ładowalna przykładowego wariantu programu konwersji)

„drukT - drukowanie tekstów” – przykładowe podprogramy drukowania tekstów

`„drukT5.asm.txt”` – podprogram źródłowy drukowania na dalekopisie tekstu typu T0

`„drukT6.asm.txt”` – podprogram źródłowy drukowania na dalekopisie tekstu typu T6

`„drukT7.asm.txt”` – podprogram źródłowy drukowania na dalekopisie tekstu typu T7

`„test drukT5, drukT6, drukT7.asm.txt”` – test powyższych podprogramów

TABELKA ZNAMIONOWA asemblera AsmC

Asembler elektronicznej maszyny cyfrowej ODRA 1003 / ODRA 1013	
Nazwa programu	AsmC
Wersja	2.5.1
Data wydania	2024-11-07
Autor	Klemens Czajka
Licencja	Freeware
Minimalne wymagania sprzętowe	ODRA 1003 + program STAŁY dalekopis TTY MKD-2 PL4 perforator taśmy PTP5 czytnik taśmy perforowanej PTR0
Maksymalne wymagania sprzętowe	ODRA 1013 UMCS + program STAŁY rekonstrukcja dalekopis TTY MKD-2 PL4 perforator taśmy PTP5 czytnik taśmy perforowanej PTR0 czytnik taśmy perforowanej PTR2 czytnik taśmy perforowanej PTR0’
Punkt wejścia (adres startowy)	0c00000
Liczba zastrzeżonych słów kluczowych	71
Liczba standardowych symboli	2
Maksymalna liczba definiowalnych nazw	2024 + 2 symbole standardowe
Maksymalna długość nazwy	7 znaków (nazwy globalne) 6 znaków + prefiks # (nazwy lokalne)
Maksymalna długość tekstu	254 słów + 1 słowo zerowe, tj. liczba kodów: 1778 (5-bitowych), 1524 (6-bitowych), 1270 (7-bitowych)
Wysokość stosu kalkulatora wyrażeń	31 operatorów, 31 operandów
Wysokość stosu instrukcji AIF-AFI	8 poziomów
Zajętość pamięci operacyjnej	Cała