Mikrokomputery

Mikroprocesor Z80

Jerzy Karczmarczuk

Mikroprocesor
Z80

Mikrokomputery

Komitet Redakcyjny

Sekretarz WOJCIECH CELLARY
ZUZANNA GRZEJSZCZAK
ANDRZEJ KOBUS
Przewodniczący ROMUALD MARCZYŃSKI
PIOTR MISIUREWICZ
WOJCIECH NOWAKOWSKI
MACIEJ STOLARSKI
HALINA TEMPCZYK
JÓZEF WINKOWSKI
JAN ZABRODZKI

Jerzy Karczmarczuk

Mikroprocesor Z80

Wydawnictwa Naukowo-Techniczne
Warszawa 1987

Opiniodawca Wojciech Cellary
Redaktor Ewa Zdanowicz
Redaktor techniczny Elżbieta Gontarz
Okładkę i strony tytułowe projektował Juliusz Rybicki

681.3

W książce opisano architekturę i zasady działania mikroprocesora Z80. Przedstawiono listę rozkazów procesora — podano mnemoniczny zapis każdego rozkazu, jego kod, długość w bajtach oraz czas wykonania w taktach zegarowych. Omówiono techniki programowania w języku asemblera Z80.

Książka jest przeznaczona dla szerokiego kręgu Czytelników interesujących się mikrokomputerami.

S.102869

© Copyright by Wydawnictwa Naukowo-Techniczne
Warszawa 1987

All rights reserved
Printed in Poland

ISBN 83-204-0861-X

WNT Warszawa 1987. Wydanie I. Nakład 30.000+200 egz.
Ark. wyd. 11,5. Ark. druk. 11,0. Format B-5. Papier offset
kl. III, 70 g. Pracę oddano do składu w maju 1986 r.
Podpisano do druku w listopadzie 1987 r. Druk ukończono
w listopadzie 1987 r. Symbol Et/82114/WNT
Drukarnia im. Rewolucji Październikowej - Warszawa
Zam. 2748/11/86 K-13

Spis treści

Przedmowa/7

Część pierwsza. Budowa i działanie mikroprocesora Z80/9

1. Wstęp/9

1.1. Przedstawiamy bohatera/9
1.2. Podstawowe cechy użytkowe Z80/10
1.3. Dodatkowe układy rodziny Z80/11
1.4. Reprezentacja liczb w procesorze i podstawowe operacje arytmetyczne i logiczne/12

2. Budowa i organizacja wewnętrzna Z80/16

2.1. Podstawowe dane techniczne/16
2.2. Wyprowadzenia zewnętrzne procesora/17
2.3. Organizacja wewnętrzna procesora/19
2.4. Rejestry/21

3. Działanie procesora/26

3.1. Cykle pracy procesora/26
3.2. Wykonanie cyklu M1/27
3.3. Cykl odczytu i zapisu pamięci oraz urządzeń zewnętrznych/29
3.4. Cykl przyjęcia przerwania/31
3.5. Żądanie wyłączności dostępu do magistral. Zerowanie procesora/34

4. Zasady programowania w języku wewnętrznym/36

4.1. Tryby adresowania pamięci przez procesor/36
4.2. Mnemonika asemblera Z80. Konwencje zapisu argumentów operacji/38
4.3. Kody rozkazów i ich struktura/42

5. Lista rozkazów procesora Z80/46

5.1. Podział rozkazów na grupy. Konwencje notacyjne/46
5.2. Grupa rozkazów przesyłania jednego bajtu/48

5.3. Grupa rozkazów przesłań dwubajtowych/53
5.4. Grupa rozkazów zamiany/56
5.5. Grupa 8-bitowych rozkazów arytmetycznych i logicznych/57
5.6. Grupa rozkazów przesunięć/64
5.7. Grupa 16-bitowych rozkazów arytmetycznych/68
5.8. Grupa rozkazów przesyłania i przeszukiwania bloków/71
5.9. Grupa rozkazów sterujących/74
5.10. Grupa rozkazów skoków/76
5.11. Grupa rozkazów wywołań podprogramów i rozkazów powrotu/80
5.12. Grupa rozkazów adresujących bity/84
5.13. Grupa rozkazów wejścia i wyjścia/85

Część druga. Techniki programowania w języku asemblera/89

6. Architektura programów w języku asemblera/89

6.1. Od algorytmu do programu/89
6.2. Podprogramy i wykorzystanie stosu/91
6.3. Podprogramy przemieszczalne. Współprogramy/95
6.4. Rozgałęzienia i pętle/99
6.5. Programy sterowane danymi. Techniki interpretacji i ich wykorzystanie/105

7. Operacje arytmetyczne i logiczne/108

7.1. Operacje na bitach i ich zastosowanie/108
7.2. Prosta arytmetyka na liczbach całkowitych. Konwersja liczb z postaci zewnętrznej na dwójkową i odwrotnie/110
7.3. Złożone procedury arytmetyczne/116

8. Struktury danych i ich przetwarzanie/125

8.1. Tablice jednowymiarowe/125
8.2. Tablice wielowymiarowe i struktury listowe/133

Dodatek A. Program deasemblera Z80/140

Dodatek B. Wybrane procedury arytmetyczne/152

Dodatek C.1. Tablica kodów rozkazów mikroprocesora Z80 uszeregowana alfabetycznie/161

Dodatek C.2. Tablica kodów rozkazów mikroprocesora Z80 uszeregowana według kodów/166

Literatura/171

Skorowidz/172

Przedmowa

Książka ta nie jest pierwszą pozycją w naszej literaturze zawierającą informacje na temat mikroprocesora Z80. Listę rozkazów oraz skrótowy opis innych cech procesora można znaleźć np. w [7], [9] lub [11], a także w przygotowywanej do druku przez WNT książce T. Kręglewskiego, P. Misiurewicza i K. Sachy „Przewodnik po technice mikrokomputerowej”. Celem autora niniejszej książki — ze względu na jej monograficzny charakter — był pełny, wyczerpujący opis możliwości tego procesora z punktu widzenia programisty, a także omówienie techniki programowania w języku asemblera. Opisano więc dość dokładnie przebiegi czasowe w procesorze oraz pewne szczegóły dotyczące np. systemu przerwań, niekiedy pomijane w literaturze. Podano również „nieoficjalne”, niepublikowane przez firmę Zilog rozkazy procesora. Rozkazy te zostały przez autora przetestowane, jednakże nie może to gwarantować ich skuteczności dla każdego egzemplarza Z80.

Druga część książki jest podręcznikiem programowania w języku asemblera Z80. Takiej pozycji dotychczas u nas nie wydano, chociaż praca [7] zawiera nieco przykładów programów. O potrzebie takiej książki świadczy popularność pozycji [10] i [12] oraz innych książek Rodnaya Zaksa poświęconych programowaniu innych mikroprocesorów.

Książka jest przeznaczona dla osób mających już pewne, choćby niewielkie doświadczenie w programowaniu, niekoniecznie w języku asemblera. Może służyć zawodowym projektantom oprogramowania, ale także wielu osobom dysponującym mikrokomputerem opartym na Z80 i chcącym rozszerzyć jego możliwości.

Związek umiejętności programowania ze znajomością listy rozkazów jest nieco podobny do relacji między umiejętnością tłumaczenia, czy nawet mówienia w danym języku, a znajomością słownika. Trzeba trenować, ale niezłym punktem startowym może być czytanie gotowych rozwiązań. Temu celowi są poświęcone przykłady. Jest ich sporo, niektóre są dość złożone. Ich

głównym zadaniem jest możliwie wszechstronna demonstracja sposobów korzystania z różnych rozkazów Z80, trybów adresowania oraz metod organizacji i optymalizacji programów. Każdemu większemu przykładowi towarzyszą sugestie dotyczące samodzielnych ćwiczeń.

Bardzo ważnym elementem programowania jest znajomość podstawowych algorytmów numerycznych bądź innych, w oderwaniu od konkretnego języka programowania, zwłaszcza zaś od języka asemblera, który charakteryzuje się tym, że idea algorytmu może być kompletnie zakryta szczegółami technicznymi. Chyba nie będzie przesady w stwierdzeniu, że bardziej skomplikowanego algorytmu praktycznie nie można nauczyć się jedynie na poziomie asemblera. Tak więc jakość programów pisanych na niskim poziomie, tj. bliskim sprzętowi, jest też wykładnikiem ogólniejszej wiedzy programisty o danych i algorytmach. Książki [1] i [6] zawierają wiele interesujących przykładów poświęconych np. arytmetyce, tak sformułowanych, że przełożenie ich na język asemblera nie powinno być bardzo trudne. W niniejszej książce niektórym rozwiązaniom na poziomie asemblera towarzyszą procedury napisane w języku wyższego poziomu zbliżonym do Pascala. Ich celem jest czytelny i zwarty, ale precyzyjny opis użytego algorytmu, co powinno ułatwić czytanie procedury w języku asemblera.

Jerzy Karczmarczuk

Kraków, czerwiec 1985

Budowa i działanie mikroprocesora Z80

1. Wstęp

1.1. Przedstawiamy bohatera

Wkrótce po wypuszczeniu na rynek przez firmę Intel mikroprocesora znanego jako 8080 (produkowanego także przez innych producentów, między innymi również w Polsce) od firmy tej odłączyła się grupa pracowników, która założyła własne przedsiębiorstwo o nazwie Zilog Inc. (10460 Bubb Road, Cupertino, California 95014). W latach 1976—77 w firmie Zilog zaprojektowano, a następnie wypuszczono na rynek 8-bitowy mikroprocesor o nazwie Z80, który wkrótce stał się szlagierem rynku mikroprocesorowego. Firma Zilog później zaprojektowała i inne procesory, w tym dość znaną 16-bitową rodzinę Z8000, jednak swojego pierwszego sukcesu już nie powtórzyła.

Z80 charakteryzował się dużym stopniem złożoności, co było w roku 1977 ewenementem, a jednocześnie stosunkowo dużą prostotą i wygodą w sprzęganiu go z układami współpracującymi, szczególnie z pamięcią. Obecnie Zilog nie jest jedynym producentem Z80. Wytwarza go również firma Mostek, japońska firma NEC, międzynarodowa firma SGS ATES z siedzibą we Włoszech, a od pewnego czasu Z80 jest produkowany w NRD pod nazwą procesor U880. Produkowane są również procesory o nieco innych parametrach, jak np. NSC 800, które są zgodne programowo z Z80.

Mikroprocesor Z80 powstał według zamierzeń twórców jako daleko idące rozszerzenie i udoskonalenie mikroprocesora Intel 8080. Dzięki programowej zgodności z 8080 miał przejąć część wybuchowo w owym czasie rozwijającego się rynku zastosowań. Zamierzenie to się udało. Systemy komputerowe oparte na Z80 mogły od razu skorzystać z pokaźnego banku programów pracujących pod kontrolą systemu operacyjnego CP/M, a nawet wpłynęły

w pewien sposób na rozwój tego systemu. Wypuszczony mniej więcej w tym samym czasie mikroprocesor Intel 8085, który również był udoskonaleniem 8080, nie był już w stanie zagrozić popularności Z80.

Chęć zachowania zgodności z 8080, a jednocześnie stworzenia urządzenia znacznie odeń doskonalszego nie we wszystkich aspektach korzystnie wpłynęła na projekt. Pewne elementy Z80 zaprojektowano niejednolicie i niezbyt dogodnie. Lista rozkazów sprawia wrażenie przeładowanej, a mimo to pewnych rzeczy bardzo brakuje, np. wygodnego adresowania pośredniego. Konstrukcja rozkazów nie jest regularna, przebiegi czasowe podczas wykonania rozkazu są dość zawiłe, co powoduje, że Z80 ustępuje szybkością działania niektórym innym procesorom 8-bitowym. Tym niemniej w dziedzinie procesorów 8-bitowych, których gwiazda świeci ciągle jasno mimo szybkiego rozwoju procesorów 16- i 32-bitowych, Z80 zajmuje utrwaloną pozycję.

1.2. Podstawowe cechy użytkowe Z80

Mikroprocesor Z80 jest procesorem 8-bitowym, tj. może przetwarzać bezpośrednio dane z zakresu liczb całkowitych od 0 do 255 lub — inaczej je interpretując — od −128 do 127, z tym że dysponuje również kilkoma rozkazami arytmetycznymi 16-bitowymi, co ułatwia manipulację adresami. Może bezpośrednio adresować 2^{16}, tj. 65536 komórek pamięci.

W odróżnieniu od 8080 mikroprocesor Z80 wymaga zasilania pojedynczym poziomem napięcia +5 V i taktowania jednofazowym zegarem. Obecnie są sprzedawane następujące wersje Z80: podstawowa o maksymalnej częstotliwości zegara 2,5 MHz, wersja Z80A o częstotliwości do 4 MHz, wersja Z80B — do 6 MHz, a niedawno pojawiła się wersja Z80H — 8 MHz. Wersja 2,5 MHz w Ameryce i Europie Zachodniej jest już wycofywana.

Wszystkie sygnały kontrolne i sterujące w Z80 są zgodne ze standardami TTL i nie są multipleksowane z danymi, jak w 8080. Dane nie są również multipleksowane z adresami, co z kolei jest niedogodną cechą 8085. Mikroprocesor Z80 można więc łączyć z pamięcią i innymi układami współpracującymi bez specjalnych, wyrafinowanych układów sterujących i demultipleksujących. Dodatkową zaletą Z80 jest generowanie tzw. *adresu odświeżania*, który może być wykorzystany do regeneracji zawartości pamięci dynamicznych, co w przypadku użycia innych procesorów wymaga stosowania specjalnych układów. Z80 może również na sygnał zewnętrzny odłączyć swoje szyny danych i adresów od magistrali systemowej, przez wprowadzenie ich w stan wysokiej impedancji, i w ten sposób umożliwić na pewien czas urządzeniom zewnętrznym przejęcie kontroli nad przepływem informacji w systemie.

Z80 dysponuje stosunkowo dużym zestawem wewnętrznych rejestrów — w sumie 208 bitów dostępnych użytkownikowi. Dzięki temu więcej informacji

można przechować wewnątrz procesora i program jest bardziej zwarty, a co za tym idzie szybszy niż w przypadku procesorów 8080 i 8085. Procesor Z80 ma dwa 16-bitowe rejestry indeksowe; jest to bardzo wygodne w operowaniu tablicami i ułatwia implementację języków wyższego poziomu. Wbudowany rejestr stosu umożliwia łatwą konstrukcję i obsługę podprogramów. Wreszcie tzw. *rejestr wektora przerwań* pozwala na elastyczną reakcję procesora na przerwania ze strony urządzeń zewnętrznych bez potrzeby stosowania skomplikowanych układów sprzętowych.

Rozkazy dotyczące wejścia i wyjścia są dość rozbudowane. Z80 może adresować pełne 2^{16} portów[1)] wejścia/wyjścia, a ich adresy konstruować dynamicznie. Dysponuje również rozkazami transmisji blokowej, co może być bardzo użyteczne w przypadku kontaktu z szybkimi urządzeniami zewnętrznymi, takimi jak dyski.

1.3. Dodatkowe układy rodziny Z80

Firma Zilog oprócz procesora Z80 skonstruowała również rodzinę innych układów o dużym stopniu scalenia współpracujących z tym procesorem i lepiej dostosowanych do jego specyfiki niż analogiczne urządzenia firmy Intel. Nie będą one tutaj szczegółowo omawiane, w literaturze dostępnej w Polsce niektóre zostały już opisane [7]. Wypada tylko wspomnieć o bardzo dobrze zaprojektowanych modułach wejścia/wyjścia:

— równoległym PIO (ang. *parallel input/output*), który jest dwukanałowym portem 8-bitowym mogącym samodzielnie generować przerwania procesora i autonomicznie rozpoznawać wykonanie przez procesor rozkazu RETI (omówionego w p. 5.11 i oznaczającego zakończenie wykonywania podprogramu obsługi przerwania);

— szeregowym SIO (ang. *serial input/output*), który można uznać za mały procesor komunikacyjny dokonujący nie tylko konwersji danych z równoległej na szeregową i odwrotnie, ale obsługujący dość skomplikowane protokoły transmisji synchronicznej.

Ponadto do typowych członków rodziny Z80 należą: programowalny zegar/licznik CTC (ang. *clock-timer circuit*) oraz układ bezpośredniego dostępu do pamięci DMA (ang. *direct-memory-access controller*). Układ DMA w bardzo efektywny sposób wykorzystuje charakterystykę przebiegów czasowych w szynach danych i adresów podczas pracy procesora i umożliwia zewnętrznym urządzeniom bezpośredni dostęp do pamięci w trybie nie zmniejszającym szybkości wykonania programu; umożliwia także automatyczne poszukiwanie w pamięci programowanych wzorców bitowych.

---

1) Wyjaśnienie pojęcia „port" na str. 30

Układy o dużym stopniu scalenia współpracujące z procesorem są często równie ważne jak sam procesor. Nie do pomyślenia jest zaprojektowanie efektywnego układu mikroprocesorowego, w którym zasadniczy wpływ na koszty, pobór mocy i zajmowaną powierzchnię będą miały układy wspomagające skonstruowane z pojedynczych bramek.

1.4. Reprezentacja liczb w procesorze i podstawowe operacje arytmetyczne i logiczne

Rozdział ten nie ma na celu wyjaśniania podstaw pracy mikroprocesorów początkującym czytelnikom, których odsyłamy do innych pozycji literaturowych (np. [5]), lecz sprecyzowanie pojęć występujących w dalszym opisie języka wewnętrznego Z80.

Aczkolwiek procesor dysponuje rozkazami mogącymi operować pojedynczymi bitami, podstawową jednostką danych, na której wykonuje się operacje arytmetyczne i logiczne jest *bajt* (ang. *byte*). Jest to zespół 8 bitów stanowiący najmniejszą adresowalną jednostkę pamięci i najmniejszą porcję danych przesyłaną z rejestrów procesora do pamięci oraz między rejestrami. W dalszym ciągu terminem bajt będziemy określać zarówno miejsce w pamięci (np. mowa będzie o pierwszym bajcie określonej tablicy), jak i samą jego zawartość (tj. 8-bitową daną niezależnie od miejsca, gdzie się znajduje, i od interpretacji). Bajt może mieć interpretację ściśle uzależnioną od sprzętu — np. kod rozkazu, czysto umowną — np. znak alfanumeryczny, może też być interpretowany jako liczba.

Chcieliśmy unikać określenia „słowo maszynowe" ze względu na niejednoznaczność, która pojawiła się w literaturze. Przez wiele lat terminem tym określano podstawową adresowalną jednostkę informacji — w naszym przypadku bajt. Ponieważ jednak bardzo wiele operacji arytmetycznych, np. operacje na adresach, „przeciętne" rachunki numeryczne wykorzystujące liczby całkowite itp., używa jednostki 16-bitowej, tj. pary bajtów, pojawiła się tendencja, aby terminem *słowo maszynowe* określać właśnie parę bajtów. Znalazło to swoje odbicie w przyjętej mnemonice asemblera Z80 i innych procesorów 8-bitowych. Tak więc pojęcie „słowo" będzie oznaczało po prostu kolejną parę bajtów.

W przyjętej konwencji bity w bajcie numeruje się od 0 do 7, od prawej strony do lewej w dwójkowym układzie pozycyjnym, tj. najmniej znaczący jest bit o numerze 0, najbardziej — o numerze 7.

Z80 może wykonywać *operacje arytmetyczne* na liczbach dwójkowych i dwójkowo-dziesiętnych, w tzw. *kodzie BCD* (ang. *binary coded decimal*). Liczby dwójkowe reprezentowane jako 1 bajt dopuszczają dwie naturalne interpretacje:

liczby całkowitej nieujemnej (z przedziału 0...+255) oraz liczby ze znakiem (z przedziału −128...+127). Na przykład konfiguracja bitów

10101011

może być traktowana jako liczba 171 lub jako −85. Liczby ujemne są reprezentowane zgodnie z konwencją zapisu uzupełnieniowego do dwóch (lub uzupełnienia dwójkowego) [4], wg którego aby zmienić znak liczby należy zamienić wszystkie bity na przeciwne, a następnie do wyniku dodać 1. Jeśli wykonamy tę operację na powyższym przykładzie otrzymamy

01010101

co, jak łatwo sprawdzić, jest dwójkowym zapisem liczby 85. Zasadniczą korzyścią zapisu uzupełnieniowego, z punktu widzenia prostoty konstrukcji procesora, jest jednolite traktowanie liczb bez znaku i liczb ze znakiem przez arytmometr, który w ogóle ich nie rozróżnia przy wykonywaniu operacji. Zadaniem programisty jest więc właściwa interpretacja wyniku dodawania czy odejmowania, w oparciu o dodatkową informację dostarczaną przez układ arytmometru.

W tym zapisie rozpoznanie znaku liczby jest proste: jeśli najbardziej znaczący (lewy) bit liczby jest równy 0 — liczba jest dodatnia, jeśli jest równy 1 (ustawiony) — liczba jest traktowana jako ujemna.

Podstawowe pojęcia, które będą dalej potrzebne to nadmiar oraz przeniesienie. Są one czasem mylone, a początkujący użytkownicy Z80 przyzwyczajeni do procesora 8080, który nie informował o nadmiarze mogą mieć kłopoty z jego prawidłowym wykorzystaniem.

Przeniesienie (ang. *carry*) najłatwiej określić jako pojawienie się jedynki w fikcyjnym bicie o numerze 8 rejestru zawierającego wynik operacji arytmetycznej. Jeśli dodamy do siebie przykładowe liczby −85 i 85 traktując je jako liczby bez znaku, to otrzymamy:

```
  10101011
 +01010101
 ---------
 100000000
```

Ośmiobitowy wynik tej operacji, który zostanie przesłany do miejsca przeznaczenia jest równy zeru. Procesor zauważa jednak pojawienie się bitu przeniesienia i dysponuje rozkazami umożliwiającymi jego testowanie.

Dodając do siebie dwie liczby 85 otrzymamy:

```
  01010101
+ 01010101
 ---------
  10101010
```

co może być interpretowane jako liczba bez znaku 170 bądź liczba ujemna −86 w kodzie uzupełnieniowym. Żadne przeniesienie nie pojawia się, jednak zaistniała sytuacja również zostaje rozpoznana i zasygnalizowana jako *nadmiar* (ang. *overflow*). Nadmiar pojawia się, gdy suma dwóch liczb dodatnich w kodzie uzupełnieniowym da wynik ujemny. Jest on sygnalizowany również, gdy suma dwóch liczb ujemnych daje wynik dodatni. Wtedy mamy jednocześnie do czynienia z przeniesieniem.

Analogiczne zjawiska wystąpią w operacji odejmowania, która jest równoważna zmianie znaku odjemnika i dodaniu go do odjemnej. Wtedy przeniesienie można traktować jako pożyczkę z fikcyjnego bitu odjemnej.

Ani przeniesienie, ani nadmiar nie powodują powstania żadnej sytuacji wyjątkowej z punktu widzenia sprzętu i program może je zignorować, jeśli nie są użyteczne.

Z80 może także wykonywać operacje dodawania i odejmowania liczb 16-bitowych, używając do tego par rejestrów 8-bitowych. Wtedy sygnalizowane jest pojawienie się przeniesienia z 15 bitu, tj. najbardziej znaczącego bitu całości. Również nadmiar odnosi się do całości, lecz nie wszystkie, na pierwszy rzut oka podobne operacje sygnalizują go, co jest mankamentem procesora i wymaga uwagi ze strony programisty (patrz p. 5.7).

Przeniesienie odgrywa też dużą rolę w operacjach przesunięć bitów, które są istotnym elementem arytmetyki na liczbach wielobajtowych oraz w konstrukcji programów mnożenia i dzielenia. Wbudowanych rozkazów mnożenia i dzielenia Z80 nie posiada.

Jak już powiedzieliśmy, oprócz arytmetyki dwójkowej Z80 może wykonywać operacje na liczbach BCD. W tym zapisie bajt należy traktować jako dwie 4-bitowe połówki (tetrady) zawierające po jednej cyfrze dziesiętnej. Każda cyfra jest liczbą dwójkową, a określenie „dziesiętna" oznacza jedynie, że „legalne" są liczby 0...9, a liczby 10...15 wychodzą poza dopuszczalny zakres. Nasza przykładowa liczba dwójkowa 171 w zapisie BCD nie jest liczbą legalną. Konfiguracja bitów traktowana jako liczba dwójkowa o wartości 85, interpretowana jako liczba w kodzie BCD ma wartość 55. Zapis BCD jest mniej ekonomiczny od dwójkowego; jego główną zaletą jest znacznie prostsza konwersja liczb z i do postaci znakowej.

Z80 nie dysponuje żadnymi specjalnymi rozkazami arytmetycznymi dla liczb BCD. Chcąc na przykład dodać dwie liczby BCD o wartości 55 wykonujemy normalny rozkaz dodawania dwójkowego. Dodanie do siebie dwóch liczb 5 (dwójkowo: 0101) wynosi 1010 i nie jest poprawną cyfrą BCD. Program może jednak zlecić wykonanie specjalnego *rozkazu korekcyjnego* DAA, który w razie potrzeby przeniesie dziesiątkę do lewej połówki bajtu. Czterobitowy wynik dodawania nie jest jednak w ogólności jednoznaczny. I tak 2, tj. 0010, może być sumą dwóch jedynek lub dwóch dziewiątek. W tym drugim przypadku mamy

przeniesienie z bitu nr 3 do bitu nr 4, bez zauważenia którego prawidłowa interpretacja dodawania nie byłaby możliwa. Z80 sygnalizuje więc także i to przeniesienie.

Tyle tytułem wstępu. Dokładniejsze omówienie pierwotnych operacji arytmetycznych i łączenie ich w bardziej skomplikowane (z wykorzystaniem sygnałów przeniesienia, nadmiaru itp.) nastąpi przy opisie poszczególnych rozkazów arytmetycznych.

Operacje logiczne wykonywane przez Z80 to typowe 8-bitowe operacje boolowskie: *suma logiczna* (OR), *iloczyn logiczny* (AND), *negacja* oraz *różnica symetryczna* zwana też dysjunkcją lub wyłączającym „lub" (XOR — ang. *exclusive-OR*). Operacje te nie generują przeniesienia, również pojęcie nadmiaru w tym przypadku nie ma sensu. Projektanci Z80 wprowadzili jednak dodatkowy wskaźnik, który może być zmieniany przez operacje logiczne i który sygnalizuje, czy bajt wynikowy zawiera parzystą czy nieparzystą liczbę bitów równych 1. *Wskaźnik parzystości* może być wykorzystywany do sprawdzania przekłamań podczas transmisji danych. W typowych programach bywa używany raczej rzadko.

Z80 dysponuje w ograniczonym zakresie możliwością operacji na pojedynczych bitach bajtu znajdującego się w rejestrze bądź w pamięci za pomocą wyspecjalizowanych rozkazów. Rozkazy te nie pozwalają wykonywać operacji dwuargumentowych ani negować bitu, lecz można go sprawdzać, ustawiać albo zerować.

Do operacji logicznych wykonywanych przez Z80 można również zaliczyć dość bogaty repertuar rozkazów przesunięć bitów. Bity w bajcie można przesuwać w lewo lub w prawo, cyklicznie, cyklicznie z wykorzystaniem pomocniczego 1-bitowego rejestru przeniesienia jako przedłużenia bajtu do 9 bitów. Przy przesunięciach w prawo można zażądać powielenia wartości bitu znaku (bitu nr 7), co ma istotne znaczenie dla szybkiej realizacji operacji mnożenia i dzielenia liczby ze znakiem przez potęgę dwójki.

2. Budowa i organizacja wewnętrzna Z80

2.1. Podstawowe dane techniczne

Szczegółowe informacje zawarte w tym i następnym rozdziale nie są konieczne, aby zacząć programować w języku wewnętrznym procesora. Potrzebne jest natomiast posiadanie podstawowej wiedzy o rejestrach wewnętrznych oraz możliwościach ich wykorzystania. Jak wiadomo z doświadczenia, oprócz samodzielnej praktyki niezłą metodą uczenia się programowania jest czytanie nie tylko podręczników, lecz także dobrych, sprawdzonych programów, najlepiej dobrze udokumentowanych. Bliższa znajomość procesora, sygnałów sterujących generowanych i odbieranych przezeń oraz niektórych przebiegów czasowych w procesorze może bardzo ułatwić zrozumienie wyboru tych czy innych rozwiązań programowych.

Oczywiście, znajomość parametrów technicznych jest niezbędna do projektowania systemu. Podstawowym źródłem informacji są opracowania firmy Zilog [13, 14], jednak należy pamiętać, że dokładne charakterystyki elektryczne mogą zależeć od poszczególnych producentów mikroprocesora Z80.

Z80 jest pojedynczym układem scalonym, zaprojektowanym w technologii NMOS z bramką krzemową. Istnieje również jego wersja CMOS używana m.in. przez firmę EPSON, co świadczy o popularności procesora, ale tą wersją nie będziemy się dalej zajmować. Produkowany jest on w obudowie typu *dual-in-line* z 40 wyprowadzeniami.

Zakres temperatur, w którym procesor może pracować wynosi od 0 do 70 stopni C. (Wersje wojskowe mogą się charakteryzować szerszym zakresem.)

Mikroprocesor Z80 wymaga pojedynczego zasilania napięciem $U_{CC} = +5\ V \pm 5\%$. Zakres dopuszczalnych napięć (względem ziemi) na dowolnym wyprowadzeniu wynosi od $-0{,}3$ V do $+7$ V. Maksymalna rozpraszana moc wynosi szacunkowo 1,5 W. Maksymalny prąd zasilania wynosi 150 mA dla wersji

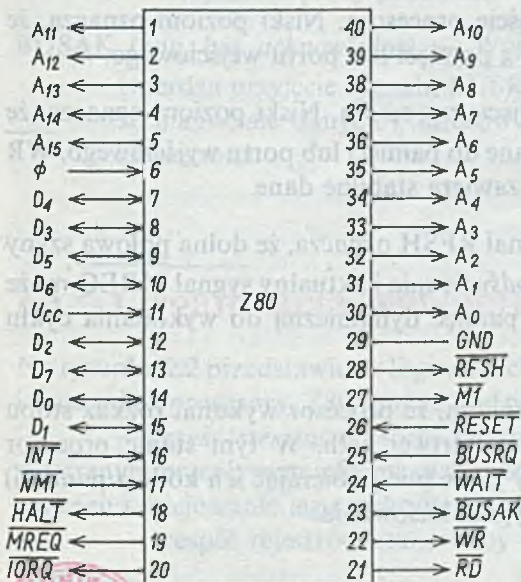
podstawowej i 200 mA dla wersji A; prąd typowy — 90 mA. Dane powyższe mogą ulegać wahaniom w zależności od producenta. Maksymalny prąd upływności szyn wynosi ok. 10 μA.

Właściwe formowanie sygnałów taktujących odbywa się wewnętrznie, Z80 wymaga jednak zewnętrznego generatora sygnału prostokątnego o napięciu w granicach: od (−0,3 V...+0,45 V) — niskie — do (U_{CC}−0,6 V...U_{CC}+ +0,3 V) — wysokie. Maksymalna pojemność wejścia zegarowego wynosi 35 pF.

Jak już wspominaliśmy, wszystkie wyprowadzenia zewnętrzne są zgodne ze standardem TTL. Oznacza to, że niski poziom napięcia wejściowego może się wahać od −0,3 V do 0,8 V, a wysoki od 2,0 V do U_{CC}. Poziomy generowane przez procesor: niski wynosi +0,4 V, wysoki — +2,4 V. Wszystkie sygnały sterujące mogą się pojawiać na wyprowadzeniach całkowicie asynchronicznie.

2.2. Wyprowadzenia zewnętrzne procesora

Na rysunku 2.1 przedstawiono schematyczne rozmieszczenie wyprowadzeń procesora Z80. Pozioma kreska nad nazwą wyprowadzenia oznacza, że niski poziom napięcia (logiczne 0) traktuje się jako poziom aktywny, zaś poziom wysoki (logiczne 1) — jako poziom spoczynkowy. Kierunek strzałki informuje, czy jest to wejście czy wyjście procesora. Opis jest jedynie orientacyjny, szczegóły dotyczące wykorzystania wyprowadzeń zostaną omówione w następnych rozdziałach.



Rys. 2.1. Wyprowadzenia mikroprocesora Z80

POLITECHNIKA ŚLĄSKA BIBLIOTEKA GŁÓWNA

GND (ang. *ground*) — Poziom ziemi. Wszystkie napięcia na wyprowadzeniach definiuje się względem tego poziomu.

U_{CC} — Napięcie zasilania +5 V.

ϕ — Wejście zegarowe.

$A_0...A_{15}$ (A od ang. *address bus*) — Wyjście procesora. Jednokierunkowa, trójstanowa 16-bitowa szyna adresów służąca do adresowania zarówno pamięci, jak i portów wejścia/wyjścia.

$D_0...D_7$ (D od ang. *data bus*) — Dwukierunkowa, trójstanowa 8-bitowa szyna danych służąca do przesyłania wszystkich danych z i do procesora.

$\overline{M1}$ — Wyjście procesora. Sygnalizuje wykonywanie tzw. *pierwszego cyklu maszynowego* (M1), w którym procesor pobiera rozkaz z pamięci, dekoduje go i inicjuje jego wykonanie.

$\overline{MREQ}$ (ang. *memory request*) — Trójstanowe wyjście procesora. Niski poziom oznacza, że procesor żąda kontaktu z pamięcią i że szyna adresów zawiera już stabilny adres.

$\overline{IORQ}$ (ang. *input-output request*) — Trójstanowe wyjście procesora. Niski poziom oznacza, że procesor wykonuje rozkaz wejścia lub wyjścia i żąda kontaktu z odpowiednim rejestrem portu wejścia/wyjścia. Szyna adresów zawiera odpowiedni adres. Sygnał $\overline{IORQ}$ jest generowany również jako potwierdzenie przyjęcia przerwania.

$\overline{RD}$ (ang. *read*) — Trójstanowe wyjście procesora. Niski poziom oznacza, że procesor będzie czytał dane z pamięci lub portu wejściowego.

$\overline{WR}$ (ang. *write*) — Trójstanowe wyjście procesora. Niski poziom oznacza, że procesor będzie zapisywał dane do pamięci lub portu wyjściowego. $\overline{WR}$ informuje, że szyna danych zawiera stabilne dane.

$\overline{RFSH}$ (ang. *refresh*) — Wyjście. Sygnał RFSH oznacza, że dolna połowa szyny adresów zawiera tzw. *adres odświeżania* i aktualny sygnał $\overline{MREQ}$ może zostać wykorzystany przez pamięć dynamiczną do wykonania cyklu odświeżania.

$\overline{HALT}$ — Wyjście. Sygnał $\overline{HALT}$ oznacza, że procesor wykonał rozkaz stopu HALT i przeszedł do stanu martwej pętli. W tym stanie procesor wykonuje cyklicznie rozkazy puste (nie pobierając ich kodu z pamięci) i czeka na przerwanie lub sygnał kasowania.

$\overline{\mathrm{WAIT}}$ — Wejście. Sygnał $\overline{\mathrm{WAIT}}$ informuje procesor, że pamięć bądź inny układ zewnętrzny nie nadąża za procesorem i żąda wstrzymania pracy procesora na okres jednego lub więcej cykli zegarowych.

$\overline{\mathrm{INT}}$ (ang. *interrupt request*) — Wejście. Sygnał $\overline{\mathrm{INT}}$ — jeśli nie zostanie zignorowany — spowoduje przerwanie pracy procesora (po zakończeniu wykonywania aktualnego rozkazu) i wykonanie podprogramu obsługi przerwania. Jest to tzw. przerwanie maskowalne.

$\overline{\mathrm{NMI}}$ (ang. *non-maskable interrupt*) — Wejście. Sygnał $\overline{\mathrm{NMI}}$ ma pierwszeństwo przed $\overline{\mathrm{INT}}$ i przerywa bezwarunkowo pracę procesora po zakończeniu wykonywania aktualnego rozkazu. Reakcja procesora jest wyzwalana opadającym zboczem napięcia na tym wyprowadzeniu. Jest to tzw. przerwanie niemaskowalne.

$\overline{\mathrm{RESET}}$ — Wejście. Sygnał $\overline{\mathrm{RESET}}$ „kasuje" procesor. Dopóki ten sygnał jest aktywny, procesor zawiesza wszystkie wyprowadzenia trójstanowe. Następnie praca Z80 jest wznawiana „od początku", tj. procesor zeruje licznik rozkazów oraz niektóre rejestry wewnętrzne zanim zacznie wykonywać kolejne cykle pracy.

$\overline{\mathrm{BUSRQ}}$ (ang. *bus request*) — Wejście procesora. Sygnał $\overline{\mathrm{BUSRQ}}$ ma wyższy priorytet niż $\overline{\mathrm{NMI}}$ i jest zawsze przyjmowany po zakończeniu aktualnego cyklu maszynowego. Powoduje przejście szyny adresów, danych i innych wyprowadzeń trójstanowych w stan wysokiej impedancji i wstrzymanie pracy procesora.

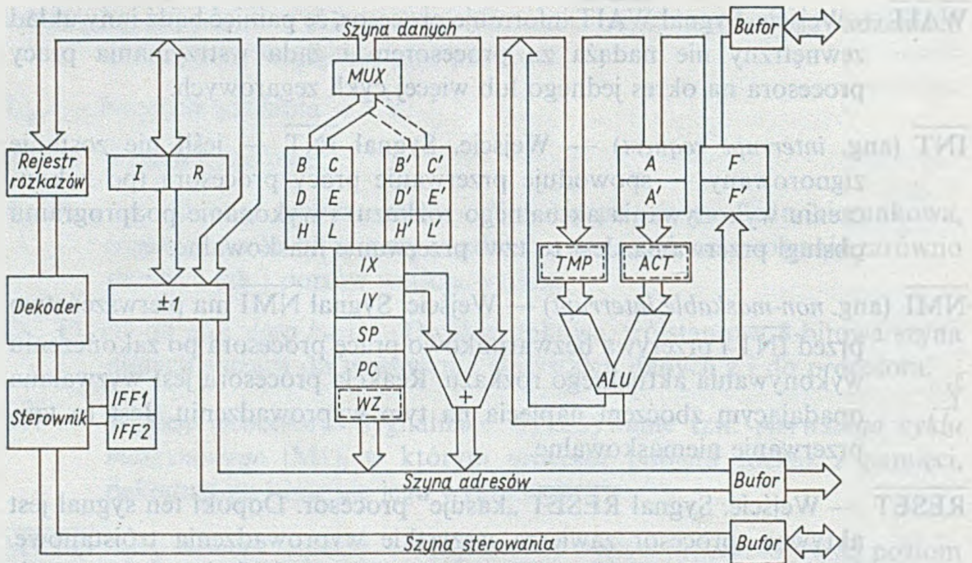
$\overline{\mathrm{BUSAK}}$ (ang. *bus acknowledge*) — Wyjście procesora. Sygnał $\overline{\mathrm{BUSAK}}$ potwierdza przyjęcie sygnału $\overline{\mathrm{BUSRQ}}$. Informuje urządzenia zewnętrzne, że magistrale danych i adresów są do dyspozycji, a praca procesora zawieszona.

2.3. Organizacja wewnętrzna procesora

Na rysunku 2.2 przedstawiono logiczny schemat architektury Z80. Podobnie jak i inne mikroprocesory, Z80 można podzielić ogólnie na trzy części:

— *część sterująca*, której zadaniem jest kontrola i zarządzanie wewnętrznym przepływem informacji w procesorze, w szczególności dekodowanie rozkazu i inicjowanie jego wykonania;

— zespół rejestrów stanowiący wewnętrzną pamięć operacyjną procesora;



Rys. 2.2. Wewnętrzna organizacja mikroprocesora Z80

— główny układ wykonawczy noszący nazwę *arytmometru* lub *jednostki arytmetyczno-logicznej* (ang. *arithmetic-logic unit* – ALU).

Jednostka arytmetyczno-logiczna zajmuje się właściwym wykonaniem takich rozkazów jak 8-bitowe dodawanie, odejmowanie, logiczne (boolowskie) mnożenie itp. Niektóre rozkazy, jak np. skoku, nie przechodzą przez ALU, gdyż zmieniają wyłącznie stan części sterującej procesora. ALU zazwyczaj bywa rysowana w kształcie litery V. Na rysunku widać drugi, mniejszy obiekt o podobnym kształcie. Jest to specjalny sumator, który pozwala na szybkie wykonywanie operacji dodawnia 8-bitowej liczby do zawartości rejestru indeksowego, umożliwiając łatwą organizację względnego adresowania pamięci.

Część sterująca procesora również dysponuje małym sumatorem, którego zadaniem jest automatyczne zwiększanie o 1 zawartości licznika rozkazów oraz licznika odświeżania.

Całość jest spięta zespołami połączeń, które noszą nazwę szyn. Termin szyna bywa w literaturze używany zamiennie z terminem magistrala. W niniejszej książce dla jednoznaczności pozostawimy termin *szyna* dla określenia wewnętrznego kanału informacji w procesorze, natomiast przez *magistralę* będziemy rozumieć kanał zewnętrzny — element systemu mikroprocesorowego, do którego podłączone są szyny procesora. Mamy więc wewnętrzną 8-bitową *szynę danych, szynę adresów*, a także zespół wielu połączeń, którymi przebiegają sygnały sterujące i kontrolne procesora, umownie zwany *szyną sterowania.* Aczkolwiek magistrala adresowa jest fundamentalną częścią każdego układu

komputerowego, szyna adresów z punktu widzenia wewnętrznej organizacji procesora nie pełni żadnej wyróżnionej roli — jest po prostu 16-bitowym wyjściem procesora. Oczywiście przedstawienie na tak schematycznym rysunku jak rys. 2.2 wszystkich istotnych połączeń nie jest możliwe.

2.4. Rejestry

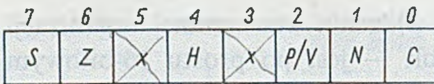
Przedstawienie wszystkich elementów, których rolą jest zapamiętywanie przez pewien czas informacji przepływającej przez procesor nie jest możliwe. Mamy wiele przerzutników, buforów i rejestrów pomocniczych, które nie są dostępne użytkownikowi. Na rysunku 2.2 rejestry, do których program nie ma bezpośredniego dostępu są oznaczone linią przerywaną.

Rejestrami dostępnymi programowo są:

— akumulator A (ang. *accumulator*),
— rejestr wskaźników stanu F (ang. *flag register*),
— rejestry ogólnego przeznaczenia B, C, D, E, H i L,
— rejestr wskaźnika stosu SP (ang. *stack pointer*),
— rejestry indeksowe IX i IY (ang. *index registers*),
— rejestr wektora przerwań I (ang. *interrupt vector*),
— licznik odświeżania pamięci R (ang. *memory refresh*),
— licznik rozkazów PC (ang. *program counter*).

Ponadto procesor dysponuje dwoma jednobitowymi rejestrami IFF1 i IFF2 (ang. *interrupt enable flip-flops*) informującymi, czy procesor jest w stanie przyjąć przerwanie maskowalne. Dwa przerzutniki są potrzebne, gdyż przerwanie niemaskowalne automatycznie blokuje możliwość przyjmowania przerwań maskowalnych przez procesor, lecz może odtworzyć stan poprzedni po powrocie z podprogramu obsługi przerwania.

Podstawowym rejestrem zawierającym jeden z argumentów operacji arytmetycznych i logicznych jest 8-bitowy *akumulator*. Do akumulatora jest również przesyłany wynik operacji wykonanej przez ALU. Jak widać na rys. 2.2 ALU ma jeszcze „boczne wyjście", którym są przesyłane dane do rejestru F. Rejestr ten przechowuje pomocniczą informację dotyczącą wyniku ostatnio wykonanej operacji: czy wynik jest równy zeru, jaki ma znak, czy nastąpiło przeniesienie lub nadmiar itp. Rejestr F jest 8-bitowy. Jeśli odpowiedni warunek jest spełniony, bit odpowiadający temu warunkowi jest ustawiany. Bity te nie są bezpośrednio odczytywane przez program (aczkolwiek można to uczynić, choć niezbyt wygodnie), lecz automatycznie sprawdzane przez procesor podczas wykonywania rozkazów skoków warunkowych oraz rozkazów dotyczących arytmetyki BCD. Strukturę rejestru F pokazano na rys. 2.3.



Rys. 2.3. Struktura rejestru wskaźników stanu F

Oznaczenia są następujące:

0. C (ang. *carry*) — wskaźnik przeniesienia. Jest ustawiany (C = 1), jeśli podczas operacji arytmetycznej lub przesunięcia bit równy 1 „wyszedł poza rejestr".

1. N — wskaźnik odejmowania. Jest ustawiany (N = 1), jeśli ostatnio wykonaną operacją było odejmowanie, a zerowany — jeśli dodawanie. Jest potrzebny w arytmetyce BCD do prawidłowego wykonania operacji korekcji dziesiętnej.

2. P/V (ang. *parity/overflow*) — wskaźnik nadmiaru/parzystości. Ponieważ nadmiar może być sygnalizowany w operacjach arytmetycznych, dla których liczba jedynek w bajcie jest informacją bez znaczenia, projektanci Z80 oszczędzili jeden bit. I tak bit nr 2 rejestru F służy do wskazywania nadmiaru operacji arytmetycznych oraz parzystości w przypadku operacji logicznych. Jest ustawiany (P/V = 1), jeśli pojawił się nadmiar lub jeśli liczba ustawionych bitów jest parzysta. Mnemonicznym sposobem zapamiętania, przy jakiej parzystości wskaźnik ten jest ustawiany, jest zauważenie, że słowo „nieparzyste" brzmi po angielsku „odd", a litera O przypomina cyfrę 0.

3. x — bit nie używany. Nie jest sprawdzany ani przez rozkazy skoków warunkowych, ani przez rozkaz korekcji dziesiętnej. ALU umieszcza w nim jednak jakąś informację, na ogół jest to kopia bitu nr 3 jednego z argumentów lub wyniku operacji. (Patrz np. [8].)

4. H (ang. *half-carry*) — wskaźnik przeniesienia z bitu nr 3 do bitu nr 4. Omawialiśmy potrzebę istnienia tego wskaźnika wspominając o zasadach arytmetyki BCD.

5. x — bit nie używany. (Patrz bit 3.)

6. Z (ang. *zero*) — wskaźnik zera. Ten bit jest ustawiany (Z = 1), jeśli wynikiem operacji arytmetycznej lub logicznej jest zero.

7. S (ang. *sign*) — wskaźnik znaku. Jest to kopia najbardziej znaczącego bitu wyniku operacji i wynosi 1, jeśli wynikiem jest liczba ujemna w dwójkowym kodzie uzupełnieniowym.

Z80 dysponuje stosunkowo dużą liczbą 8-bitowych *rejestrów ogólnego przeznaczenia* (roboczych). Ośmiobitowe rejestry B, C, D, E, H i L można również wykorzystać parami BC, DE i HL, traktowanymi jako rejestry 16-bitowe. Oznaczenia alfabetyczne nie są w pełni regularne. Nazwy H i L są historyczne. Już mikroprocesor Intel 8008 zawierał 16-bitową parę rejestrów HL, której zadaniem było przechowywanie adresów: H — bardziej znaczącej połowy (ang. *high*), L — mniej znaczącej (ang. *low*). Wykorzystanie par BC, DE i HL nie jest w pełni symetryczne. Każdy rejestr ma swoją specyfikę, co odbija się na regularności listy rozkazów i utrudnia jej zapamiętanie. Na przykład istnieje

rozkaz skoku do adresu będącego zawartością pary HL, a podobne rozkazy dla BC i DE nie istnieją. Rejestr HL służy również jako akumulator w 16-bitowych rozkazach arytmetycznych, zaś rejestr BC jako licznik w rozkazach transmisji blokowej, a także jako rejestr adresujący port wejścia/wyjścia.

Jak widać na rys. 2.2 rejestry BC, DE, HL, a także A i F są podwojone. W rzeczywistości Z80 ma dwa zestawy (banki) tych rejestrów. Drugi bank rejestrów ma oznaczenia primowane: A', F' i B'...L'. Nie można jednak jednocześnie korzystać z obu banków, np. nie jest możliwe bezpośrednie przekazanie zawartości z rejestru „nieprimowanego" (głównego) do „primowanego" (alternatywnego). Można używać albo A i F, albo A' i F' oraz, niezależnie, albo BC, DE i HL, albo odpowiednich rejestrów alternatywnych. Z80 dysponuje *rozkazami przełączania banków:* EX AF, AF' i EXX, które noszą również nazwę *rozkazów zamiany*, a bywają w literaturze nazywane niezbyt ściśle rozkazami wymiany. W rzeczywistości rozkazy te nie wymieniają zawartości banków, tylko przedefiniowują bank główny na alternatywny i odwrotnie. W programie mamy do czynienia wyłącznie z rejestrami banku głównego, a jedyne operacje na banku alternatywnym to rozkazy zamiany. Typowym wykorzystaniem banku alternatywnego jest szybkie inicjowanie podprogramu obsługi przerwań. Taki podprogram, który może zostać uruchomiony w nie dającym się przewidzieć momencie musi przechować zawartość wszystkich zmienianych przezeń rejestrów, a następnie je odtworzyć. Niektóre mikroprocesory przyjmując przerwanie zapamiętują w pamięci operacyjnej zawartość wszystkich rejestrów, jednak Z80 przechowuje jedynie zawartość licznika rozkazów. Jeśli w rachubę nie wchodzi przerwanie podprogramu obsługi, co zmusiłoby program do zapamiętania krytycznych rejestrów jeszcze raz, rozkazy zamiany są szybsze i ekonomiczniejsze pamięciowo niż sekwencyjne zapamiętania zawartości rejestrów w pamięci, np. na stosie.

Z80 ma dwa 16-bitowe *rejestry indeksowe* oznaczane IX i IY. Jest również możliwe niezależne wykorzystanie 8-bitowych połówek tych rejestrów, które będą dalej oznaczane jako HX, HY, LX i LY. Jest to niestandardowa cecha procesora, która nie znajduje się w oficjalnej dokumentacji firmy Zilog. Jednak w niniejszej książce rejestry te zostaną opisane.

Większość rozkazów operujących rejestrami H i L ma swój odpowiednik dla rejestrów indeksowych. Kody tych rozkazów są dwubajtowe: pierwszy bajt to szesnastkowo DD dla rejestru IX i FD dla rejestru IY, drugi bajt to odpowiedni kod dotyczący rejestru HL. W szczególności na rejestrach indeksowych można wykonywać 16-bitowe operacje arytmetyczne, np. dodanie do IX zawartości BC. Najważniejszą rolę rejestry indeksowe pełnią w tzw. *rozkazach indeksowanych*. W rozkazach tych adres słowa pamięci, z którego pobiera się (lub przesyła) dane, oblicza się przez dodanie 8-bitowej stałej, traktowanej jako liczba z przedziału $-128...+127$ i wbudowanej w rozkaz, do zawartości odpowiedniego rejestru indeksowego. Daje to możliwość wygodnego

operowania elementami (stosunkowo niewielkich) tablic w przypadku, gdy z góry są znane indeksy elementów, natomiast nie jest znany adres początku tablicy. Tak więc, w rzeczywistości rejestry indeksowe powinny nosić nazwę rejestrów bazowych.

Wskaźnik stosu SP jest 16-bitowym rejestrem specjalnego przeznaczenia. Zawiera on adres tzw. *wierzchołka stosu* (który, jak się okaże, jest w rzeczywistości jego spodem). *Stos* jest tablicą umieszczoną w dowolnym miejscu pamięci i służy głównie do przechowywania adresów powrotów z podprogramów oraz innych zmiennych, które po powrocie z podprogramu należy odtworzyć. Każdy rozkaz wywołania podprogramu CALL umieszcza w pamięci — w słowie o adresie zawartym w SP — adres powrotu, tj. adres pierwszego bajtu po rozkazie CALL, a następnie automatycznie zmniejsza zawartość SP o 2. Zawartość par rejestrów BC, DE, HL oraz indeksowych, a także pary rejestrów AF można przesłać na stos odpowiednim rozkazem PUSH. I w tym przypadku zawartość SP zmniejsza się o 2. Stos więc rośnie w dół. Rozkazy powrotu z podprogramu i rozkazy pobrania zawartości wierzchołka stosu do odpowiedniego rejestru 16-bitowego automatycznie cofają stos, tj. zwiększają zawartość SP o 2. Nie istnieją żadne zabezpieczenia przed przekroczeniem w jedną lub drugą stronę granic tablicy przewidzianej na stos.

Licznik rozkazów PC jest obsługiwany przez moduł sterowania mikroprocesora. Przechowuje on adres następnego rozkazu do wykonania. Jego zawartość jest automatycznie zwiększana o 1 podczas pobierania i dekodowania każdego kolejnego bajtu rozkazu. Jawna zmiana jego zawartości jest równoważna rozkazowi skoku. W rzeczywistości rozkaz skoku nie polega po prostu na nadaniu nowej wartości rejestrowi PC. Aby skrócić nieco czas wykonywania tej operacji, procesor Z80 korzysta z pomocniczego rejestru WZ, który nie jest dostępny programowi. Jego rola zostanie przedstawiona podczas omawiania cykli pracy procesora. Nie jest ona istotna z punktu widzenia programisty, pomaga jednak nieco lepiej zrozumieć strukturę cyklu rozkazowego.

Licznik odświeżania R jest 8-bitowym rejestrem specjalnego przeznaczenia. Każdorazowo podczas dekodowania rozkazu pobranego z pamięci szyna adresów nie jest używana przez pewien krótki czas przez procesor. Projektanci Z80 wpadli na pomysł, aby wykorzystać ten czas i przesyłać na magistralę adresową zawartość automatycznie zwiększanego o 1 licznika. Adres ten może być wykorzystany przez układy sterujące pamięcią dynamiczną do wykonania tzw. *cyklu odświeżania*. Ze względu na standardy pamięci dynamicznych obowiązujące w czasie gdy Z80 był projektowany, na szynę adresów przesyłane i zwiększane o 1 jest tylko 7 bitów rejestru R. Najbardziej znaczący bit rejestru R nie ulega zmianie. Zawartość rejestru R jest podawana na mniej znaczącą połowę szyny adresów. W tym czasie na bardziej znaczącą połowę szyny adresów jest podawana zawartość rejestru I, co było rozwiązaniem ad hoc, lecz może być

praktycznie wykorzystane przez układy zewnętrzne. Na ogół programista ma niewielki pożytek z rejestru R, może go jednak użyć jako programowanego licznika. Można go również wykorzystać jako prymitywny generator liczb pseudolosowych, jeśli odczyt zawartości tego rejestru następuje niezbyt często i w nieregularnych odstępach.

Ośmiobitowy rejestr I zwany *rejestrem wektora przerwań* umożliwia (opcjonalnie) elastyczną reakcję procesora na zewnętrzne przerwanie maskowalne. Umieszcza się w nim bardziej znaczącą połowę adresu słowa pamięci, w którym mieści się adres podprogramu obsługi przerwania. Mniej znaczącą połowę adresu powinien dostarczyć magistralą danych układ generujący przerwanie. W ten sposób można łatwo obsługiwać wiele urządzeń zewnętrznych za pomocą niezależnych podprogramów obsługi, jednakże sam procesor nie jest w stanie rozstrzygać konfliktów między konkurującymi urządzeniami zewnętrznymi — muszą to zapewnić odpowiednie układy dodatkowe.

3. Działanie procesora

3.1. Cykle pracy procesora

Praca mikroprocesora Z80, jak każdego procesora o klasycznej architekturze, polega na powtarzaniu pojedynczego *cyklu rozkazowego* składającego się z następujących faz:

1. Pobranie z pamięci operacyjnej kodu rozkazu począwszy od adresu umieszczonego w liczniku rozkazów.
2. Dekodowanie rozkazu i przygotowanie stanu procesora do wykonania rozkazu.
3. Wykonanie rozkazu, zmiana zawartości pamięci lub rejestrów procesora.
4. Modyfikacja licznika rozkazów o liczbę zależną od długości rozkazu i jego typu.

Ponieważ rozkazy mogą być wielobajtowe, a ich wykonanie może pociągać za sobą wielokrotne odwoływanie się procesora do pamięci, każdy cykl rozkazowy jest w rzeczywistości skomplikowaną sekwencją prostszych elementów. Niektóre procesy mogą być i w rzeczywistości są wykonywane współbieżnie, np. modyfikacja licznika rozkazów może następować jednocześnie z wykonaniem rozkazu lub jeszcze wcześniej, natychmiast po tym, gdy zawartość PC została wykorzystana do zaadresowania pamięci.

Cykl rozkazowy Z80, podobnie jak procesora Intel 8080, dzieli się na *cykle maszynowe* (lub *cykle procesora*). Cykl rozkazowy może zawierać jeden cykl maszynowy, jednak na ogół zawiera ich kilka — do 6. Każdy cykl maszynowy wiąże się z pojedynczym odwołaniem się procesora do pamięci lub portu wejścia/wyjścia w celu odczytu lub zapisu i z wykonaniem wewnątrz procesora odpowiedniego przesłania między rejestrem a szyną danych (lub między

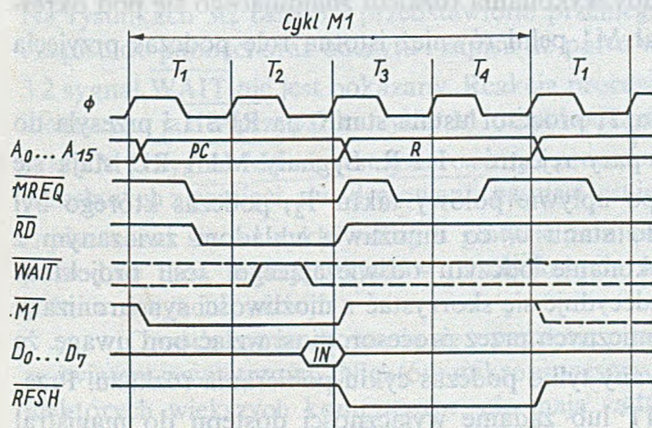
rejestrami). Może wiązać się z reakcją procesora na zewnętrzne sygnały sterujące, np. $\overline{\text{BUSRQ}}$ lub $\overline{\text{WAIT}}$.

Każdy cykl maszynowy jest więc również złożony i składa się z kilku *cykli zegarowych* (*taktów*) — od 3 do 6, typowo 4. Będą one oznaczane T_1, T_2 itp. Struktura cyklu maszynowego nie jest jednorodna; występuje częściowe nakładanie: proces rozpoczęty w jednym cyklu maszynowym może zakończyć się w następnym, jednocześnie z procesem, który należy do drugiego cyklu. Dzięki temu częściowemu paralelizmowi osiąga się kilkunastoprocentową oszczędność czasu. Jest 8 rodzajów cykli maszynowych:

1. Podstawowy cykl pobrania i dekodowania rozkazu. Będzie nazywany cyklem M1. Trwa 4 lub 5 taktów zegarowych.
2. Cykl odczytu lub zapisu pamięci. Trwa typowo 3 takty, ale może być wydłużony przez sygnał $\overline{\text{WAIT}}$.
3. Cykl odczytu lub zapisu portu wejścia/wyjścia.
4. Cykl przyjęcia sygnału $\overline{\text{BUSRQ}}$.
5. Cykl przyjęcia sygnału przerwania maskowalnego.
6. Cykl przyjęcia przerwania niemaskowalnego.
7. Cykl pracy jałowej po wykonaniu rozkazu HALT.
8. Cykl kasowania (zerowania) i inicjacji pracy procesora.

3.2. Wykonanie cyklu M1

Przebiegi czasowe i stan wyprowadzeń procesora podczas wykonania cyklu M1 są pokazane na rys. 3.1. Szczegóły tego i podobnych wykresów nie są zazwyczaj interesujące dla przeciętnego programisty, zawierają jednak podstawowe informacje dla projektantów układów i są potrzebne przy konstrukcji programów



Rys. 3.1. Przebieg czasowy podczas wykonania cyklu M1

działających w czasie rzeczywistym. Ponieważ schematyczne oznaczanie wysokiego i niskiego poziomu na szynach danych i adresów nie ma większego sensu, opis tych szyn będzie skrótowo podawał ich zawartość.

Na początku pierwszego taktu procesor umieszcza 16-bitową zawartość licznika rozkazów na szynie adresów oraz ustala stan 0 na wyprowadzeniu $\overline{M1}$. Następnie, w połowie taktu, jednocześnie z opadającym zboczem sygnału zegarowego zmienia stan $\overline{MREQ}$ z 1 na 0 oraz ustala stan 0 na wyprowadzeniu $\overline{RD}$. W ten sposób podczas drugiego taktu pamięć może wykonać odczyt i wyprowadzić zawartość odpowiedniego bajtu na magistralę danych. Stan sygnałów $\overline{MREQ}$ i $\overline{RD}$ (a także w cyklach omawianych poniżej $\overline{IORQ}$ i $\overline{WR}$) jest ustalany, gdy stany na szynie adresów (ewentualnie także na szynie danych) ustaliły się, można więc użyć bezpośrednio $\overline{MREQ}$ i $\overline{RD}$ do wyzwalania odczytu pamięci, bez wprowadzania dodatkowych opóźnień. Jeśli pamięć jest za wolna, specjalne układy elektroniczne mogą podczas taktu T_2 wygenerować sygnał $\overline{WAIT}$, który jest testowany podczas opadającego zbocza sygnału zegarowego. Jeśli WAIT jest nieaktywny, procesor przyjmuje dane i przechodzi do wykonania drugiej połowy cyklu M1, w przeciwnym razie odczekuje jeden takt i przy następnym opadającym zboczu powtarza testowanie $\overline{WAIT}$, co może trwać dowolnie długo.

Podczas drugiej połowy taktu T_2, gdy układy pamięci przygotowują dane, procesor zwiększa licznik rozkazów o 1. Sygnał $\overline{M1}$ wydaje się nadmiarowy, układom pamięciowym jest oczywiście wszystko jedno, w jakim celu dokonuje się odczytu i wykorzystują jedynie sygnały $\overline{MREQ}$ i $\overline{RD}$ (oraz $\overline{WR}$). Można jednak wykorzystać ten sygnał do poinformowania układów zewnętrznych, że procesor przystępuje do wykonania rozkazu, a nie pobiera danych z pamięci. Umożliwia to np. oddzielić obszary pamięci przeznaczone na program i dane, założyć wykrywane sprzętowo *pułapki* (ang. *breakpoints*) przerywające program w przypadku próby wykonania rozkazu znajdującego się pod określonym adresem itp. Sygnał $\overline{M1}$ pełni również istotną rolę podczas przyjęcia przerwania przez procesor.

Na początku taktu T_3 procesor ustala stan 0 na $\overline{RFSH}$ i przesyła do szyny adresów zawartość pary rejestrów I i R. Sygnały $\overline{M1}$ i $\overline{RD}$ stają się nieaktywne, zaś $\overline{MREQ}$ po upływie połowy taktu T_3, podczas którego był nieaktywny, przechodzi do stanu 0, co umożliwia układom związanym z pamięcią dynamiczną wykonanie odczytu odświeżającego. Jeśli projektant układu opartego na Z80 zdecyduje się skorzystać z możliwości synchronizacji odświeżania pamięci dynamicznych przez procesor, musi wziąć pod uwagę, że sygnał $\overline{RFSH}$ jest generowany tylko podczas cyklu pobierania rozkazu. Przedłużający się sygnał $\overline{WAIT}$ lub żądanie wyłączności dostępu do magistral

$\overline{\text{BUSRQ}}$, a także przedłużający się stan zerowania $\overline{\text{RESET}}$ powyżej 1 ms może spowodować utratę informacji przez pamięć. Niebezpieczne jest również nadmierne obniżenie częstotliwości zegara. Projektant układu dysponujący danymi technicznymi pamięci może obliczyć najmniejszą dopuszczalną częstotliwość taktowania zakładając, że procesor powtarza „najgorszy" możliwy rozkaz: EX (SP), HL, który zawiera tylko jeden cykl M1, a trwa 19 taktów. Typowo wyniesie ona 1,2 MHz dla pamięci 16 K bajtów wymagającej odświeżania co 2 ms.

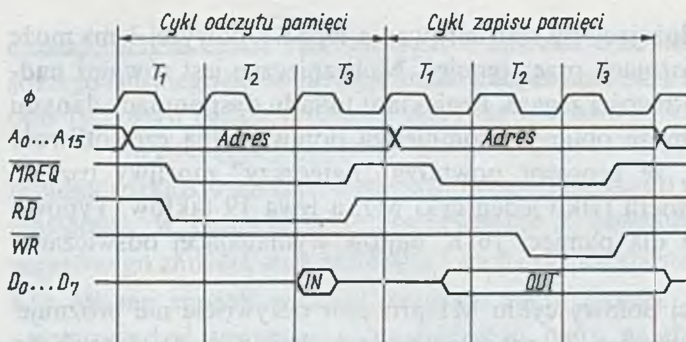
Podczas drugiej połowy cyklu M1 procesor oczywiście nie próżnuje. Podczas narastającego zbocza T_3 procesor przesyła bajt z szyny danych do rejestru rozkazów, a następnie przystępuje do dekodowania rozkazu. Jeśli jest to rozkaz nie wymagający kontaktu z pamięcią, np. jest przesłaniem między rejestrami lub operacją arytmetyczną na zawartości akumulatora i innego rejestru, całość wykonania rozkazu kończy się w pierwszym cyklu maszynowym.

W tym miejscu można omówić jeszcze *cykl pracy jałowej* (lub cykl procesora w stanie stopu). Po wykonaniu rozkazu HALT normalna praca procesora ulega zatrzymaniu. Procesor przeprowadza szyny danych i adresów w stan wysokiej impedancji i przestaje się kontaktować z pamięcią. Na zewnątrz jest wysyłany sygnał $\overline{\text{HALT}}$ informujący o stanie procesora, z którego może go wyprowadzić jedynie przerwanie lub zerowanie. Nie znaczy to, że wtedy procesor „nie robi nic". Należy pamiętać, że sygnał $\overline{\text{RFSH}}$ jest generowany tylko podczas wykonywania cyklu M1 przez procesor. W związku z tym stan procesora po wykonaniu rozkazu HALT jest dynamiczny: procesor powtarza jednobajtowy rozkaz pusty NOP generując normalne sygnały odświeżania.

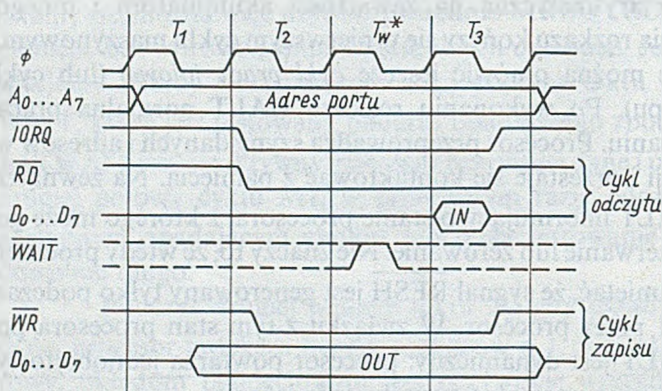
3.3. Cykl odczytu i zapisu pamięci oraz urządzeń zewnętrznych

Na rysunkach 3.2 oraz 3.3 przedstawiono przebiegi czasowe podczas odczytu i zapisu do pamięci oraz odczytu i zapisu do portu wejścia/wyjścia. Na rysunku 3.2 sygnał $\overline{\text{WAIT}}$ nie jest pokazany. Reakcja procesora na jego wystąpienie jest identyczna jak w przypadku cyklu M1. Cykl odczytu pamięci jest podobny do cyklu M1. Sygnał $\overline{\text{M1}}$ nie jest generowany i procesor robi inny użytek z danych przesłanych z pamięci, ale adresowanie pamięci jest identyczne. Zewnętrznie cykl zapisu różni się od cyklu odczytu głównie zamianą sygnału $\overline{\text{RD}}$ na $\overline{\text{WR}}$, co powoduje inną reakcję układów pamięciowych. Występują również drobne różnice w synchronizacji.

Oprócz komunikowania się z pamięcią procesor musi kontaktować się ze światem zewnętrznym. Niektóre mikroprocesory, a także jednostki centralne niektórych większych komputerów, nie mają żadnych specjalnych instrukcji



Rys. 3.2. Przebieg czasowy cyklu odczytu i zapisu pamięci



Rys. 3.3. Przebieg czasowy cyklu odczytu i zapisu portu wejścia/wyjścia

wejścia lub wyjścia, natomiast odwołanie do wyróżnionych adresów pamięci jest interpretowane jako żądanie odczytu lub zapisu rejestru przypisanego urządzeniu zewnętrznemu, np. niezależnemu procesorowi komunikacyjnemu. Procesor Z80 może, niezależnie od adresowania pamięci, adresować 65536 rejestrów wejścia/wyjścia (choć niektóre pozycje literaturowe, np. [7] podają liczbę 256). Na ogół każde fizyczne urządzenie zewnętrzne jest traktowane przez procesor jako kilka lub kilkanaście rejestrów. Te rejestry będą w dalszym ciągu nazywane *portami wejścia/wyjścia*. Szczegóły dotyczące połączeń procesora z portami, sprzętowej organizacji systemu przerwań itp. nie zostaną w tej książce omówione.

Należy zauważyć, że w przypadku rozkazu wymagającego kontaktu z urządzeniami wejścia/wyjścia procesor automatycznie wprowadza opóźnienie o jeden takt między T_2 a T_3, oznaczany jako T_w^*. Daje to zewnętrznym układom pewien luz czasowy potrzebny do zdekodowania adresu portu i ewentualnie do przygotowania sygnału $\overline{\text{WAIT}}$ dla procesora.

3.4. Cykl przyjęcia przerwania

Przerwaniem nazywamy wymuszone odpowiednim sygnałem sterującym wstrzymanie automatycznego generowania kolejnych adresów rozkazów do wykonania i wywołanie specjalnego podprogramu obsługi. Przerwania znajdują zastosowanie we współpracy procesora z asynchronicznie działającymi urządzeniami zewnętrznymi. Procesor obsługujący np. powolną drukarkę nie musi nieustannie sprawdzać, czy urządzenie jest gotowe do przyjęcia następnego znaku, może wykonywać inne czynności, a gdy drukarka zgłosi gotowość, zasygnalizuje o tym za pomocą przerwania. Innym możliwym zastosowaniem przerwań jest cykliczne przerywanie pracy procesora celem diagnostyki lub aby umożliwić współbieżne (naprzemienne) wykonywanie wielu programów. Przyjęcie przerwania powoduje automatycznie zapamiętanie na stosie zawartości licznika rozkazów, co umożliwia poprawny powrót do programu wykonywanego przed przerwaniem. Nie jest natomiast zapamiętywana zawartość żadnych innych rejestrów; podprogram obsługi przerwania musi zadbać o to sam.

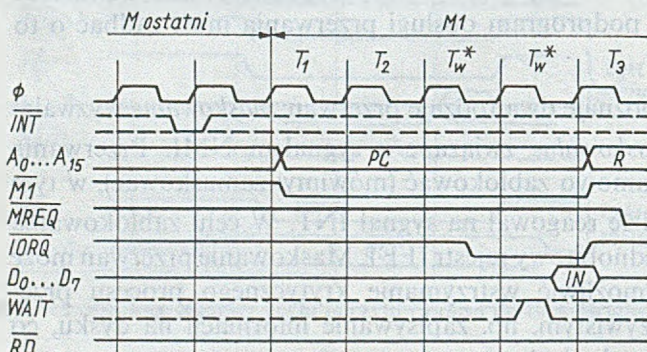
Procesor Z80 rozpoznaje dwa rodzaje przerwań: *maskowalne* wyzwalane sygnałem $\overline{\mathrm{INT}}$ i *niemaskowalne* związane z sygnałem $\overline{\mathrm{NMI}}$. Przerwania maskowalne można programowo zablokować (mówimy: zamaskować), w tym sensie, że procesor nie będzie reagował na sygnał $\overline{\mathrm{INT}}$. W celu zablokowania przerwań procesor zeruje jednobitowy rejestr IFF1. Maskowanie przerwań może być stosowane, aby uniemożliwić wstrzymanie krytycznego procesu przebiegającego w czasie rzeczywistym, np. zapisywanie informacji na dysku, co mogłoby zdezorganizować cały dysk.

Przerwania niemaskowalne zostały przewidziane do celów specjalnych, np. do sygnalizacji awarii, spadku napięcia zasilania itp., a także jako „ostatnia deska ratunku" w przypadku zawieszenia pracy systemu, np. zapętlenia albo zatrzymania rozkazem HALT przy wyłączonych lub zablokowanych przerwaniach maskowalnych. Sygnały $\overline{\mathrm{INT}}$ oraz $\overline{\mathrm{NMI}}$ są testowane przez procesor podczas narastającego zbocza ostatniego taktu ostatniego cyklu maszynowego poprzedniego rozkazu. Jeśli sygnały $\overline{\mathrm{INT}}$ i $\overline{\mathrm{NMI}}$ są jednocześnie aktywne, to przerwanie niemaskowalne ma pierwszeństwo. W obu przypadkach stan 0 na wyprowadzeniu oznacza zgłoszenie przerwania, jednak reakcja procesora na sygnał $\overline{\mathrm{NMI}}$ jest nieco inna niż na sygnał $\overline{\mathrm{INT}}$, co jest często pomijane w literaturze opisującej skrótowo procesor Z80. Otóż w przypadku przerwania maskowalnego procesor reaguje wyłącznie na poziom sygnału. Natomiast w przypadku $\overline{\mathrm{NMI}}$, testowany przez układy sterujące procesora bufor wyprowadzenia jest zerowany opadającym zboczem sygnału $\overline{\mathrm{NMI}}$. Oznacza to, że jeśli

podczas wykonywania podprogramu obsługi przerwania niemaskowalnego stan $\overline{\text{NMI}}$ nie ulegnie zmianie i będzie nadal 0 — nie spowoduje to następnego przerwania. Podobny problem w przypadku przerwania maskowalnego jest rozstrzygany inaczej: po przyjęciu przerwania ($\overline{\text{INT}}$ lub $\overline{\text{NMI}}$) — przerwania maskowalne są blokowane.

Po przyjęciu przerwania maskowalnego procesor zeruje oba rejestry IFF1 i IFF2. Po przyjęciu przerwania niemaskowalnego procesor przepisuje zawartość IFF1 do IFF2, a następnie zeruje IFF1.

Z punktu widzenia programu reakcja procesora na przerwanie niemaskowalne jest bardzo prosta: procesor wywołuje podprogram mieszczący się pod adresem 102 (szesnastkowo 66). Należy pamiętać, że przerwania maskowalne są wtedy zablokowane i nie zostają automatycznie odblokowywane. Powrót z podprogramu obsługi zawiera dodatkowe komplikacje, które zostaną omówione w następnych rozdziałach.



Rys. 3.4. Przebieg czasowy cyklu przyjęcia przerwania

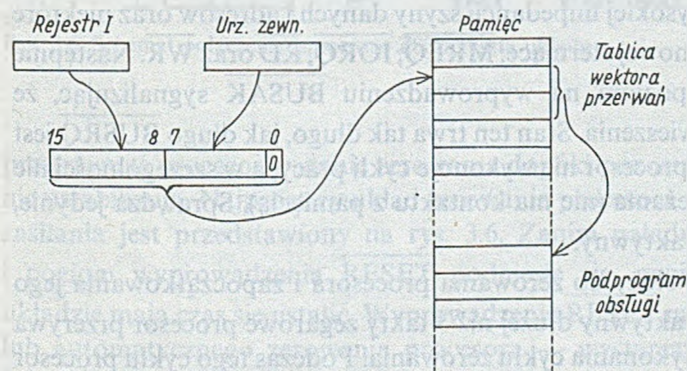
Jeśli przerwanie maskowalne zostanie przyjęte, procesor generuje specjalny cykl M1 (rys. 3.4) różniący się od normalnego tym, że zamiast $\overline{\text{MREQ}}$ uaktywnia się $\overline{\text{IORQ}}$, tj. procesor może pobrać następny bajt z portu wejściowego, a nie z pamięci. Sygnał $\overline{\text{RFSH}}$ jest generowany normalnie, a do szyny adresów procesor wysyła zawartość PC, co oczywiście nie ma większego sensu, lecz było prostsze do zaprojektowania niż specjalna obsługa szyny adresów. Dwa dodatkowe takty oczekiwania T_w^* są automatycznie wykonywane przez procesor, aby np. ułatwić współpracę procesora z układami organizującymi pierwszeństwo przerwań i rozstrzygającymi konflikty. Na uwagę zasługuje jeszcze brak sygnału $\overline{\text{RD}}$. Daje to układom zewnętrznym możliwość prostego odróżnienia między normalnym odczytem $\overline{\text{RD}}$ a reakcją na przerwanie: $\overline{\text{M1}}$ w połączeniu z $\overline{\text{IORQ}}$.

Mikroprocesor Z80 dysponuje trzema trybami reakcji na przerwania maskowalne oznaczanymi symbolami: IM 0, IM 1 i IM 2 (ang. *interrupt modes*). Przełączanie między tymi trybami jest programowane i zostanie omówione później.

W *trybie 0* procesor traktuje specjalny cykl M1 jako normalny cykl pobrania rozkazu, tj. traktuje zawartość szyny danych jako kod rozkazu do wykonania. Tę daną powinno dostarczyć urządzenie zewnętrzne. Aczkolwiek możliwe jest przesłanie w ten sposób rozkazu wielobajtowego, najczęściej spotykanym rozwiązaniem jest przesłanie kodu tzw. *rozkazu restartu* RST, który jest krótkim, jednobajtowym rozkazem wywołania podprogramu mieszczącego się pod jednym z adresów 0, 8, 16, 24, 32, 40, 48 lub 56.

W *trybie 1* specjalny cykl M1 jest ignorowany, a reakcja procesora przypomina przyjęcie NMI z tym, że zostaje wywołany podprogram mieszczący się pod adresem 56 (szesnastkowo 38).

Tryb 2 jest najbardziej wszechstronny i elastyczny. Procesor oczekuje od urządzenia zewnętrznego nie kodu rozkazu, lecz jednego bajtu danych traktowanego jako mniej znacząca połowa adresu. Jako bardziej znaczący bajt adresu procesor pobiera ustawianą programowo zawartość rejestru I, a następnie dokonuje odczytu słowa pamięci mieszczącego się pod wskazanym adresem. Te dwa odczytane bajty traktuje jako adres podprogramu obsługi przerwania. Na rys. 3.5 pokazano schematycznie powyższą sekwencję.



Rys. 3.5. Wywołanie procedury obsługi przerwania w trybie 2

Jeśli przerwania są generowane przez wiele urządzeń zewnętrznych, to każdemu można w ten sposób zapewnić osobną procedurę obsługi przerwania. Urządzenie powinno podać tylko identyfikującą je liczbę potraktowaną jako jednobajtowy indeks w tablicy, której początek wyznacza zawartość rejestru I. Tablica ta nosi nazwę *wektora przerwań*.

Indeks generowany przez urządzenie przerywające musi być parzysty! Aby się zabezpieczyć przed przypadkowym zaadresowaniem dwóch sąsiednich

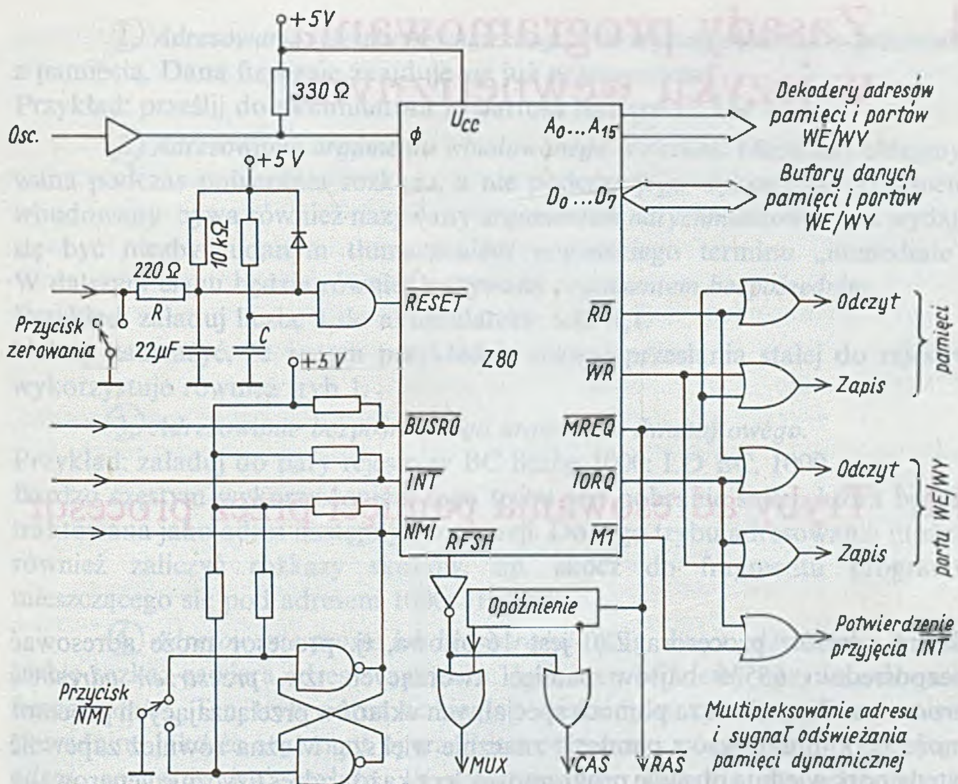
bajtów należących do różnych elementów wektora przerwań, najmniej znaczący bit indeksu jest ignorowany i zawsze traktowany jako zero. (Jest to cecha procesora nie zawsze właściwie opisywana w literaturze.)

3.5. Żądanie wyłączności dostępu do magistral. Zerowanie procesora

Jeśli Z80 jest częścią bardziej złożonego układu (np. wieloprocesorowego) lub jeśli współpracuje z szybkim urządzeniem zewnętrznym (które potrafi niezależnie od procesora adresować pamięć w celu przesyłania większych bloków danych), może się okazać konieczne chwilowe odłączenie procesora od reszty układu, przede wszystkim od magistral danych i adresów, by nie zakłócił on przepływu informacji między pamięcią a innymi elementami układu. To odłączenie jest możliwe dzięki sygnałowi BUSRQ, który ma pierwszeństwo przed przerwaniami. Ten sygnał jest testowany przy końcu każdego cyklu maszynowego, może więc przerwać w połowie wykonywanie rozkazu, a w szczególności przerwać w połowie odczyt dwubajtowego adresu z pamięci, co może zdezorganizować program, jeśli zawartość pamięci uległa zmianie podczas zawieszenia pracy procesora. Po otrzymaniu sygnału BUSRQ procesor przeprowadza w stan wysokiej impedancji szyny danych i adresów oraz niektóre wyprowadzenia (trójstanowe) sterujące: MREQ, IORQ, RD oraz WR. Następnie procesor ustala niski poziom na wyprowadzeniu BUSAK sygnalizując, że znajduje się w stanie zawieszenia. Stan ten trwa tak długo, jak długo BUSRQ jest aktywny i w tym czasie procesor nie wykonuje cykli pracy, a w szczególności nie generuje sygnału odświeżania (nie ma kontaktu z pamięcią). Sprawdza jedynie, czy BUSRQ jest nadal aktywny.

Sygnał RESET służy do zerowania procesora i zapoczątkowania jego pracy. Jeśli RESET jest aktywny dłużej niż 3 takty zegarowe procesor przerywa pracę i przystępuje do wykonania cyklu zerowania. Podczas tego cyklu procesor zawiesza wszystkie wyprowadzenia trójstanowe, a pozostałe sygnały sterujące są nieaktywne. Licznik rozkazów oraz rejestry I i R są zerowane. Oba rejestry IFF1 i IFF2 są zerowane, czyli przerwania są maskowane. Po odblokowaniu przerwań procesor będzie gotowy do ich przyjęcia w trybie 0 (jest to tryb zgodny z reakcją procesora Intel 8080).

Jest celowe, aby system zbudowany na procesorze Z80 (zresztą na innych mikroprocesorach również) był tak zaprojektowany, żeby w momencie włączenia zasilania procesor wykonywał cykl zerowania. Uniknie się w ten sposób startu programu od przypadkowego adresu, a ponadto można



Rys. 3.6. Schemat połączenia procesora Z80 z resztą układu

przetrzymać procesor w stanie zerowania dopóki pozostałe elementy układu się nie ustabilizują. Najprostszy układ zerowania procesora w momencie włączenia zasilania jest przedstawiony na rys. 3.6. Zanim naładuje się kondensator *C* i poziom wyprowadzenia RESET podniesie się, poziomy napięć w całym układzie mają czas się ustalić. Wyprowadzenie RESET może służyć do ręcznego lub automatycznego zerowania procesora — wystarczy je zewrzeć z ziemią, jednak ze względu na znaczną pojemność *C* bezpieczniej jest rozładować ją przez oporność *R*. Doskonalszą konstrukcją będzie układ, który np. po ręcznym wciśnięciu przycisku zerowania, w krótką chwilę później automatycznie podniesie poziom RESET uniemożliwiając w ten sposób przetrzymanie procesora zbyt długo w stanie zerowania (co może spowodować uszkodzenie zawartości pamięci dynamicznych).

4. Zasady programowania w języku wewnętrznym

4.1. Tryby adresowania pamięci przez procesor

Szyna adresów procesora Z80 jest 16-bitowa, tj. procesor może adresować bezpośrednio 65536 bajtów pamięci tworzących tzw. *przestrzeń adresową procesora*. Oczywiście za pomocą specjalnych układów przełączających procesor może się kontaktować z pamięcią znacznie większą. Można również zapewnić wtedy odpowiednią obsługę programową, lecz każdy adres fizycznie generowany przez procesor jest 16-bitowy. Na poziomie języka wewnętrznego istnieją jednak bardziej rozbudowane możliwości adresowania wynikające z faktu, że procesor może automatycznie wykonywać pewne operacje arytmetyczne na adresach przed ich wyprowadzeniem na szynę. Ponadto, niektóre operacje procesora korzystają jedynie z rejestrów wewnętrznych i nie wymagają kontaktu z pamięcią.

Ogólnie sposób, w jaki rozkaz odnosi się do danych, na których dokonuje operacji nazywa się *trybem adresowania danych*. Sposób definiowania trybów adresowania nie jest jednolicie przyjęty, również nazewnictwo występuje w kilku wariantach, w związku z tym należy starać się raczej zrozumieć ich znaczenie niż nauczyć się nazwy, tym bardziej że w niektórych rozkazach występuje kombinacja kilku trybów adresowania. W tej książce przyjmiemy, że procesor Z80 dysponuje 10 trybami adresowania, co jest liczbą przyjętą w większości publikacji dotyczących Z80. Nie jest to wiele w porównaniu z niektórymi innymi procesorami. Niestety, różne rozkazy dopuszczają różne tryby adresowania; pod tym względem takie procesory jak Motorola 6809 czy 68000 są zaprojektowane znacznie przyjemniej; prawie każdy argument dowolnej operacji może być otrzymany w dowolnym trybie adresowania.

Tryby adresowania dla Z80 są następujące:

1. *Adresowanie rejestru wewnętrznego.* Nie wymaga kontaktu procesora z pamięcią. Dana fizycznie znajduje się już w procesorze.
Przykład: prześlij do akumulatora zawartość rejestru C: LD A,C.

2. *Adresowanie argumentu wbudowanego w rozkaz.* Dana jest otrzymywana podczas pobierania rozkazu, a nie podczas jego wykonania. Argument wbudowany bywa również nazywany *argumentem natychmiastowym*, co wydaje się być niezbyt udanym tłumaczeniem angielskiego terminu „immediate". W dalszym ciągu będzie również nazywany *argumentem bezpośrednim.*
Przykład: załaduj liczbę 1 do akumulatora: LD A,1.
Należy zauważyć, że w tym przykładzie rozkaz przesłania stałej do rejestru wykorzystuje również tryb 1.

3. *Adresowanie bezpośredniego argumentu dwubajtowego.*
Przykład: załaduj do pary rejestrów BC liczbę 1000: LD BC, 1000.
Bardzo częstym wykorzystaniem tego trybu jest pobranie stałej, która będzie traktowana jako adres następnych operacji. Do tego trybu adresowania można również zaliczyć rozkazy skoków, np. skocz do fragmentu programu mieszczącego się pod adresem 1000: JP 1000.

4. *Adresowanie proste lub bezpośrednie* (ang. *direct addressing*). W tym trybie rozkaz zawiera adres argumentu. Adres ten będziemy nazywali *adresem bezpośrednim.* Ten tryb adresowania jest niekiedy nazywany adresowaniem pośrednim, jako że argument operacji jest otrzymany pośrednio, poprzez jego adres, co może być mylące. W rzeczywistości Z80 nie ma adresowania pośredniego, w którym rozkaz zawiera adres słowa mieszczącego adres argumentu.
Przykład: przekaż zawartość akumulatora do bajtu pamięci o adresie 1000: LD (1000), A.

5. *Adresowanie pośrednie zawartością rejestru.* Adres argumentu znajduje się w parze rejestrów, najczęściej HL, ale dla niektórych rozkazów możliwe jest użycie BC lub DE, a także rejestrów indeksowych oraz rejestru stosu.
Przykład: dodaj do akumulatora zawartość bajtu pamięci, którego adres znajduje się w HL: ADD A,(HL).
Adresowanie pośrednie zawartością rejestru stosu różni się od poprzedniego tym, że z rozkazem jest związana jednoczesna modyfikacja zawartości rejestru stosu.
Przykład: umieść na stosie zawartość pary BC: PUSH BC.

6. *Adresowanie indeksowane* (ang. *indexed addressing*). Adres argumentu jest otrzymywany przez dodanie do zawartości jednego z rejestrów indeksowych IX lub IY 8-bitowej stałej wbudowanej w rozkaz. Stała ta jest nazywana *przesunięciem* (ang. *displacement*) i jest traktowana jako liczbą ze znakiem mieszcząca się w granicach od −128 do 127.

Przykład: dodaj do akumulatora zawartość dziesiątego elementu tablicy, której początek (adres zerowego elementu) jest wskazywany przez rejestr IX: ADD A,(IX + 10).

7. *Adresowanie względne* (ang. *relative addressing*) — względem licznika rozkazów. Jest podobne do poprzedniego (6), z tym że rolę rejestru indeksowego pełni PC. Służy do programowania skoków względnych.
Przykład: skocz o 10 bajtów w przód: JR 10.
Ważną cechą Z80 będącą często źródłem pomyłek jest fakt, że przesunięcie definiuje się względem wartości PC w momencie wykonania rozkazu, a wtedy licznik rozkazów zawiera już adres następnego rozkazu, a nie aktualnie wykonywanego.

8. *Adresowanie stronicy zerowej*. Z80 dysponuje kilkoma specjalnymi, jednobajtowymi rozkazami wywołania podprogramów umieszczonych pod adresami 0, 8, 16, 24, 32, 40, 48 i 56. Odpowiedni adres jest konstruowany przez procesor z kodu rozkazu.
Przykład: wywołaj podprogram o adresie 8: RST 8.

9. *Adresowanie pojedynczych bitów*. Z80 dysponuje możliwością testowania i zmiany pojedynczego bitu dowolnego rejestru ogólnego przeznaczenia lub bajtu pamięci. Jest to oczywiście skrótowe potraktowanie tej operacji, w rzeczywistości procesor musi z pamięci pobrać cały bajt.
Przykład: wyzeruj bit 0 akumulatora RES 0, A.

10. *Adresowanie niejawne* (ang. *implied addressing*). Niektóre rozkazy są na sztywno związane z określonym rejestrem, np. wszystkie jednobajtowe operacje arytmetyczne wykorzystują akumulator A jako jeden z argumentów i jako miejsce przesłania wyniku.
Przykład: odejmij 10 od zawartości akumulatora: SUB 10.

Uwaga. Adresy są przechowywane w pamięci w określonej kolejności: mniej znaczący bajt jest zapamiętywany pod niższym adresem, bardziej znaczący bajt pod wyższym. Obowiązuje to dla wszystkich dwubajtowych przesłań z i do pamięci, dla adresów stanowiących część rozkazu, dla adresów zapamiętywanych na stosie itp.

4.2. Mnemonika asemblera Z80. Konwencje zapisu argumentów operacji

Mimo że lista rozkazów Z80 jest w porównaniu z procesorem 8080 znacznie rozbudowana, język asemblera jest prostszy i bardziej przejrzysty. Jego projektanci zdecydowali się ograniczyć liczbę nazw operacji, np. nie ma oddzielnych nazw operacji dla załadowania rejestru wartością pobraną z pamięci i dla przesłania do pamięci wartości rejestru. Wszystko to są przesłania

identyfikowane wspólną nazwą operacji LD (ang. *load*). Natomiast źródło oraz przeznaczenie przesyłanej informacji jest określane przez *argumenty operacji*. W związku z tym, postać argumentów może być w pierwszym momencie uznana za skomplikowaną. Rozkazy mogą zawierać od 0 do 2 argumentów, oddzielanych przecinkami. Argumentami mogą być liczby, adresy danych oraz oznaczenia rejestrów. Oczywiście, każdy sensowny asembler umożliwi również symboliczne zapisywanie argumentów, np. użycie identyfikatora etykiety zamiast adresu liczbowego. Poniżej opiszemy sposób kodowania programów przyjmowany przez większość standardowych asemblerów Z80, choć różne asemblery określają rozmaicie np. sposób zapisu liczb szesnastkowych, a także składnię etykiet. Będziemy używali wyłącznie etykiet alfanumerycznych, o nazwach rozpoczynających się od litery.

Pole etykiety	Pole rozkazu	Pole argumentów

Rys. 4.1. Format zapisu rozkazu w języku asemblera

Rozkazy będą zapisywane w polach, w półsztywnym formacie. Na rysunku 4.1 przedstawiono format zapisu rozkazu. *Pole etykiety* może zawierać identyfikator etykiety, który będzie traktowany przez asembler jako nazwa stałej o wartości równej adresowi rozkazu, przed którym etykieta jest umieszczona. *Pole operacji* będzie zawierać nazwę rozkazu. *Pole argumentów* oddzielone od pola operacji jedną lub większą liczbą spacji będzie zawierać argumenty oddzielane przecinkami. Wiersz lub jego część rozpoczynająca się od średnika będzie traktowana jako *komentarz*.

W dalszym ciągu obowiązywać będzie konwencja, że liczby całkowite zapisane bez specjalnego omówienia jako argumenty operacji będą liczbami dziesiętnymi, chyba że zostanie zaznaczone inaczej. Liczby szesnastkowe jedno- i dwubajtowe będą natomiast poprzedzone znakiem #, np. # FA00. Wyjątkiem będą kody rozkazów, które zwyczajowo zapisuje się szesnastkowo, gdyż wtedy zapis jest bardziej zwarty, a zapis dziesiętny i tak w tym przypadku nie ma żadnych walorów mnemonicznych. W tablicach wszystkie liczby będą szesnastkowe. Argumentami operacji będą, oprócz liczb, również bardziej skomplikowane wyrażenia zawierające operatory arytmetyczne: +, −, * i /, przy czym wartość wszystkich wyrażeń będzie obliczana w arytmetyce całkowitej i obcinana do 8 lub 16 bitów w zależności od oczekiwanego wyniku. Obcięcie może spowodować, że suma dwóch liczb dodatnich będzie ujemna! Wszystkie operacje będą wykonywane po kolei, bez zwracania uwagi na standardowo przyjmowane pierwszeństwo operatorów arytmetycznych. Jest to niewygodne i przestarzałe, jednak zachowamy tę konwencję, gdyż dla prostoty przestrzega jej większość obecnie spotykanych asemblerów. Oprócz czterech operatorów arytmetycznych

są używane również operatory relacji: =, >, <, > =, < = i < >, które dostarczają 1 w przypadku spełnienia relacji przez wyrażenie i 0 w przeciwnym razie.

Bardzo charakterystyczną dla asemblera Z80 notacją jest odróżnianie liczby będącej bezpośrednim argumentem operacji od adresu argumentu. Liczba będąca argumentem operacji jest zapisywana wprost, bez dodatkowych elementów składniowych, np. rozkaz przesłania do akumulatora liczby 1000 ma postać:

LD A,1000

Natomiast jeśli rozkaz zawiera liczbę będącą adresem argumentu operacji przesłania, arytmetycznej czy innej, liczba ta jest ujęta w nawiasy. I tak, rozkaz załadowania do akumulatora zawartości bajtu pamięci o adresie 1000 ma postać:

LD A,(1000)

Powyższa konwencja była wielokrotnie krytykowana jako myląca i utrudniająca tradycyjne wykorzystanie nawiasów w przypadku argumentów będących złożonymi wyrażeniami, ale już się utrwaliła. Jest również wykorzystywana w niektórych rozkazach adresujących rejestry, jeśli zawartością rejestru jest adres argumentu, np. rozkaz przesłania zawartości akumulatora do komórki pamięci adresowanej przez parę rejestrów HL ma postać:

LD (HL),A

Konwencja ta zawiera pewne drobne niekonsekwencje, np. rozkaz skoku do rozkazu o adresie zawartym w HL zapisujemy jako

JP (HL)

co może być mylące, gdyż argumentem operacji skoku jest sam adres, tj. zawartość pary HL, a nie liczba umieszczona pod tym adresem.

Na zakończenie należy wspomnieć o tzw. *dyrektywach asemblera*, tj. o elementach programu, które nie generują bezpośrednio kodu wynikowego, lecz wpływają na pracę programu tłumaczącego. Ograniczymy się do najczęściej używanych dyrektyw.

ORG *adres*

Występowanie dyrektywy ORG (ang. *origin*) jest związane z faktem, że dla niewielkich systemów komputerowych opartych na 8-bitowym mikroprocesorze najczęściej mamy do czynienia z kompilacją programów w ustalony z góry obszar pamięci, a generownie kodu przemieszczalnego (który może być załadowany i wykonany w różnych obszarach przestrzeni adresowej) jest raczej charakterystyczne dla większych systemów. Argument dyrektywy ORG podaje

adres, od którego następujące po tej dyrektywie rozkazy będą umieszczone w pamięci podczas wykonania programu. Jednakże asembler niekoniecznie składuje kod wynikowy od razu w pamięci pod tym adresem!

END

Dyrektywa END oznacza koniec programu źródłowego. Jest ona użyteczna w przypadku kolejnego kompilowania kilku programów.

etykieta EQU *wyrażenie*

Dyrektywa EQU (ang. *equal*) przypisuje *etykiecie* wartość otrzymaną przez obliczenie *wyrażenia*. Otrzymana w ten sposób wartość jest traktowana jako stała i nie może być zmieniana.

etykieta DEFL *wyrażenie*

Dyrektywa DEFL (ang. *define label*) różni się od EQU tym, że wartość *etykiety* może zostać przedefiniowana następną dyrektywą DEFL. Symbole zmienne mają największe zastosowanie w makrodefinicjach.

DEFB *wyrażenie,wyrażenie,....,wyrażenie*

Dyrektywa DEFB (ang. *define byte*) umieszcza w programie dane. Wartościami *wyrażeń* powinny być liczby jednobajtowe.

DEFW *wyrażenie,wyrażenie,....,wyrażenie*

Dyrektywa DEFW (ang. *define word*) umieszcza w programie dane dwubajtowe (słowa). W celu zachowania zgodności z konwencją obowiązującą dla adresów, bardziej znaczący bajt jest umieszczany pod większym adresem. Jak widać, powyższe dwie dyrektywy są związane z tworzeniem kodu wynikowego, jednak nie jest to kompilacja rozkazów.

DEFM *tekst,tekst,....,tekst*

Dyrektywa DEFM (ang. *define message*) służy do umieszczania w programie danych tekstowych. Argumenty dyrektywy są ciągami znaków ograniczonymi znakami apostrofów lub cudzysłowem, zależnie od asemblera (od którego zależy również maksymalna długość tekstu). Asembler tłumaczy każdy znak na jego kod ASCII (najczęściej, choć spotyka się i inne kody).

DEFS *wyrażenie*

Dyrektywa DEFS (ang. *define space*) służy do rezerwacji w pamięci obszaru o długości równej wartości *wyrażenia*. Jest równoważna przedefiniowaniu ORG. Między ostatnim rozkazem (lub daną) poprzedzającym DEFS a następnym, asembler wytworzy przerwę o odpowiedniej długości.

IF *wyrażenie*

ciąg rozkazów i dyrektyw

ENDIF

Jest to para dyrektyw kompilacji warunkowej. Jeśli wartość *wyrażenia* jest różna od zera, to *ciąg rozkazów i dyrektyw* jest normalnie kompilowany. W przeciwnym razie ciąg rozkazów aż do dyrektywy ENDIF jest pomijany.

nazwa MACRO *par1,par2,...,parn*

ciąg rozkazów i dyrektyw

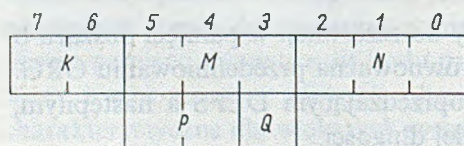
ENDM

Jest to dyrektywa makrodefinicji. Definiuje ona *nazwę* nowej, złożonej operacji. Parametry formalne *par1*, *par2* itd. występują w makrodefinicji jako argumenty i w wywołaniu są zastępowane przez dowolne wyrażenia.

4.3. Kody rozkazów i ich struktura

Rozkazy Z80 na ogół są wielobajtowe, gdyż zawierają w sobie dane w postaci argumentów bezpośrednich i adresów. Jednak o tym, jaki to jest rozkaz decyduje w zasadzie pierwszy bajt zawierający kod rozkazu. Możliwych kodów jest więc 256. Tylko 12 spośród nich nie zostało zaimplementowanych na 8080, niewiele więc można było w ten sposób dodać. Daleko idące rozszerzenie listy rozkazów Z80 osiągnięto w ten sposób, że wprowadzono rozkazy o kodach dwu- i trójbajtowych. Kody (szesnastkowo) CB, DD, ED i FD, które nie są wykorzystywane przez 8080, są traktowane jako przedrostki (prefiksy) właściwego kodu operacji. Kodami prefiksowanymi zajmiemy się później, teraz opiszemy strukturę kodu jednobajtowego. Wszystkie kody rozkazu, a także cały rozkodowany rozkaz będzie zapisywany szesnastkowo.

Bajt rozkazu można podzielić na trzy segmenty, które określają klasę operacji oraz rodzaj argumentów. Środkowy segment można podzielić na dwa mniejsze. Strukturę bajtu kodu rozkazu przedstawiono na rys. 4.2. Do przedstawionego schematu nie należy rozkaz HALT o kodzie 76 będący rozkazem zatrzymania pracy procesora. (Reakcja procesora na rozkaz HALT została omówiona w poprzednim rozdziale.)



Rys. 4.2. Struktura bajtu zawierającego kod rozkazu

Pierwsze 2 bity (segment *K*) określają (z wyjątkami) ogólną klasę operacji. I tak, jednobajtowe rozkazy przesłań mają pierwsze 2 bity równe 01. Pozostałe 6 bitów kodu rozkazu dzieli się na 2 grupy określające rejestry: przeznaczenia i źródłowy. Każdy rejestr jest opisany 3-bitowym kodem według schematu:

Rejestr	Kod (dwójkowo)
B	000
C	001
D	010
E	011
H	100
L	101
(HL)	110
A	111

Schemat ten obowiązuje dla wszystkich rozkazów adresujących rejestry. Ściśle mówiąc (HL) nie jest rejestrem, lecz bajtem pamięci adresowanym przez HL, jednak w wielu rozkazach formalnie można traktować go jako rejestr. (W opisie procesora 8080 nosi on nazwę wirtualnego rejestru M.) Kod 110 jest używany również w rozkazach wykorzystujących adresowanie indeksowane.

Przykładowo: rozkaz o kodzie 7A ma postać dwójkową

01 111 010

co natychmiast tłumaczy się na LD A,D.

Podobnie, wszystkie jednobajtowe rozkazy arytmetyczne i logiczne zaczynają się od bitów 10. Ostatnie 3 bity określają rejestr, który obok akumulatora jest argumentem operacji, a środkowe specyfikują dokładniej operację. I tak, rozkaz o kodzie 86, czyli dwójkowo 10 000 110, jest rozkazem dodania do akumulatora bajtu pamięci: ADD A,(HL). Bity 000 oznaczają operację dodawania 8-bitowego bez przeniesienia.

Struktura rozkazów rozpoczynających się od bitów 00 i 11 jest mniej regularna. Wszystkie rozkazy skoków, wywołań podprogramów i powrotów z nich zaczynają się od 11 — z wyjątkiem skoków względnych, które zaczynają się od 00 i wykorzystują kody nie używane przez 8080. Rozkazy o kodach

00 *ddd* 100 (inkrementacja zawartości rejestru),

00 *ddd* 101 (dekrementacja zawartości rejestru) oraz

00 *ddd* 110 (ładowanie do rejestru jednobajtowego agrumentu bezpośredniego)

wykorzystują 3 środkowe bity do identyfikacji rejestru według schematu podanego powyżej. Litera *d* oznacza tu cyfrę dwójkową.

W kodach prefiksowanych drugi bajt kodu rozkazu rozpoczynającego się od CB również ma strukturę opisaną powyżej. Pierwsze 2 bity definiują klasę

operacji. Ostatnie 3 bity zawsze określają rejestr, na którym operacja jest wykonywana. Od bitów 00 rozpoczynają się rozkazy przesunięć i obrotów (przesunięć cyklicznych). (W tym niektóre rozkazy, które nie figurują na oficjalnej liście rozkazów.) Środkowe 3 bity specyfikują operację. Niektóre z tych rozkazów są nadmiarowe i powtarzają istniejące już rozkazy obrotów akumulatora. Istnieją więc dwa rozkazy: RLA i RL A, które wykonują dokładnie tę samą operację. Drugi z nich jest dłuższy i wolniejszy, i pozornie nie powinien być nigdy stosowany, jednak istnieje ze względu na regularność schematu adresowania rejestrów w rozszerzonych rozkazach, a ponadto inaczej ustawia rejestr wskaźników stanu, co może być przydatne.

Bity 01 identyfikują operację BIT testującą pojedynczy bit rejestru, bity 10 — operację SET ustawiającą bit, zaś 11 — operację RES zerującą bit. Numer bitu jest określany przez środkowe 3 bity kodu. Przykładowo, drugi bajt rozkazu o kodzie CB EA ma postać dwójkową 11 101 010 i jest dekodowany jako RES 5,D.

Rozkazy prefiksowane przez ED mają strukturę częściowo regularną. W powyższy schemat wchodzą rozkazy wejścia/wyjścia o dynamicznie określanym adresie portu. Struktura drugiego bajtu rozkazu jest wtedy następująca:

01 *ddd* 000 dla operacji wejścia oraz
01 *ddd* 001 dla operacji wyjścia

ddd określa rejestr, którego (lub do którego) zawartość jest przesyłana. Dla *ddd* równego 110 w przypadku operacji wejścia dana pobrana z portu wejściowego nie jest przesyłana do pamięci pod adres zawarty w HL, natomiast w oparciu o informację przeczytaną z portu wejściowego zmienia zawartość rejestru wskaźników stanu F. Ten rozkaz jest nieoficjalny.

W tej klasie również występują rozkazy nadmiarowe. Rozkazy o kodach jednobajtowych umożliwiają ładowanie par rejestrów BC, DE oraz HL dwubajtowym argumentem bezpośrednim, ale jedynie HL dopuszcza tryb adresowania prostego. Rozkazy LD BC,(*nn*), LD (*nn*),BC (gdzie *nn* symbolizuje liczbę 16-bitową) i podobne dla DE należą do klasy rozkazów prefiksowanych przez ED. W związku z tym rozkazy LD HL,(*nn*) i LD (*nn*),HL występują w dwóch wersjach: zwykłej i prefiksowanej.

Najbardziej charakterystyczne dla rozszerzonych kodów są rozkazy rozpoczynające się od prefiksów DD i FD. Służą one do wykonywania operacji związanych z rejestrami indeksowymi. Ogólnie, jeśli rozkaz dotyczący rejestru H, L, HL lub (HL) poprzedzimy bajtem DD, skonstruujemy rozkaz dotyczący jednej z połówek lub całego rejestru IX, względnie bajtu pamięci — bądź adresowanego wprost przez IX, bądź z jednobajtowym przesunięciem, które jest zapisywane jako trzeci bajt rozkazu. Podobnie rozkazy prefiksowane przez FD dotyczą rejestru IY. Przykładowo, rozkaz zmniejszenia o 1 zawartości rejestru HL: DEC HL o kodzie 2B, poprzedzony bajtem FD staje się rozkazem: DEC IY.

Jest to rozkaz adresujący jedynie rejestr indeksowy, nie należy go mylić z rozkazem indeksowanym. Rozkaz przesłania zawartości rejestru H do pamięci pod adres zawarty w rejestrze HL: LD (HL),H ma kod 74. Rozkaz DD 74 10 dekodowany jako LD (IX+16), H jest typowym rozkazem wykorzystującym adresowanie indeksowane. W tym przypadku tylko element (HL) został zamieniony na odpowiednie wyrażenie wykorzystujące rejestr indeksowy, rejestr H nadal figuruje w tym rozkazie. Natomiast prefiks DD przed rozkazem LD L,H zamienia go na rozkaz LD LX,HX. Rozkazów dotyczących połówek rejestrów IX i IY nie ma na oficjalnej liście i w związku z tym nie ma oficjalnych nazw dla połówek rejestrów indeksowych. Będziemy je oznaczać HX i HY — bardziej znaczące bajty, oraz LX i LY — mniej znaczące. Te oznaczenia są już spotykane w niektórych publikacjach, np. [8]. Są też wyjątki, tj. nie każdy rozkaz dotyczący HL można prefiksować (a właściwie można prefiksować, lecz prefiks DD czy FD jest ignorowany przez procesor). Takim wyjątkiem jest np. rozkaz wymiany zawartości rejestrów HL i DE: EX DE,HL (o kodzie EB). Wszystkie znane autorowi egzemplarze procesora Z80 ignorują prefiksowanie tego rozkazu i chociaż pozycja [9] wspomina o rozkazach EX, DE,IX oraz EX DE,IY, należy sądzić, że takie rozkazy nie istnieją. Rozkazy prefiksowane przez ED również ignorują prefiksy DD i FD (przed ED).

Indeksowane rozkazy przesunięć oraz operacje na bitach mają kody trójbajtowe. Prefiksowane są przez FD CB lub DD CB. Charakterystyczne w ich strukturze jest to, że przesunięcie nadal występuje jako trzeci bajt rozkazu, tj. zaraz za CB, a właściwy jednobajtowy kod specyfikujący operację jest czwartym bajtem rozkazu. Struktura tego bajtu jest następująca:

bb ddd 110

Dla *bb* równego 00 bity *ddd* specyfikują operację przesunięcia. Bity 01, 10 lub 11 identyfikują operację BIT, SET oraz RES podobnie jak dla rozkazów nieprefiksowanych. Wtedy *ddd* określa numer bitu.

Indeksowane rozkazy operacji na bitach również dopuszczają nieoficjalne rozszerzenia. Zauważmy, że ostatnie 3 bity są zawsze równe 110. Jest to informacja nadmiarowa, żadnych innych rozkazów rozpoczynających się od DD CB oraz FD CB nie ma. Zamieniając 110 na inny kod rejestru w rozkazach przesunięć, SET oraz RES otrzymamy rozkaz wykonujący operację na argumencie otrzymanym w trybie indeksowanym, jednak wynik jest również przesyłany do dodatkowego rejestru. Rozszerzenie notacji asemblera dla tych rozkazów będzie polegało na dodaniu trzeciego argumentu. Nie jest to rozszerzenie rozpoznawane przez typowe asemblery Z80. Na przykład rozkaz SET 3,(IX),D o kodzie DD CB 00 DA ustawia trzeci bit bajtu pamięci adresowanego przez IX, a ponadto przesyła cały wynikowy bajt do rejestru D. Rozkazy testujące bity nie podlegają temu rozszerzeniu.

5. Lista rozkazów procesora Z80

5.1. Podział rozkazów na grupy. Konwencje notacyjne

Ten rozdział jest w całości poświęcony dokładnemu omówieniu wszystkich rozkazów mikroprocesora Z80. Zaznaczone zostaną czasy wykonania w taktach zegarowych oraz stan rejestru wskaźników po wykonaniu rozkazu. Programiści chcący szybko i efektywnie programować w języku asemblera muszą pamięciowo opanować rozkazy także pod względem zmian stanu rejestru wskaźników, gdyż ustawianie wskaźników nie zawsze odpowiada intuicji, np. fakt, że 8-bitowe operacje arytmetyczne modyfikują inne wskaźniki niż operacje inkrementacji czy operacje arytmetyczne 16-bitowe może być mylący dla początkujących. W dalszym ciągu obowiązywać będzie następująca konwencja dotycząca wskaźników:

0 — oznacza, że bit wskaźnika jest zerowany,
1 — bit wskaźnika jest ustawiany,
↕ — bit jest zerowany lub ustawiany w zależności od wyniku operacji, tj. przeniesienie występuje lub nie, wynikiem jest zero lub liczba różna od zera itp..
* — bit nie jest zmieniany przez operację,
? — wartość bitu jest przypadkowa.

Bity nr 3 oraz nr 5 rejestru wskaźników będą ignorowane. Oficjalnie wartość tych bitów jest nieokreślona. W rzeczywistości w jednobajtowych operacjach arytmetycznych i logicznych do tych bitów jest wpisywany trzeci i piąty bit bajtu wynikowego. W większości innych operacji bity te nie są zmieniane. Więcej szczegółów na ten temat można znaleźć w [8]. Ponieważ „ręczne" sprawdzanie pojedynczych bitów rejestru wskaźników jest nie-

wygodne — trzeba przesłać zawartość F na stos, a następnie ściągać ją do rejestru roboczego — to w dalszych rozważaniach bity nr 3 i nr 5 pominiemy.

Sposób testowania zawartości rejestru F zostanie omówiony razem z grupą rozkazów skoków, gdyż podstawowe wykorzystanie wskaźników stanu jest związane ze skokami warunkowymi. Niektóre bity rejestru F są testowane automatycznie przy wykonywaniu operacji korekcji dziesiętnej.

Uwaga. Użytkowników przenoszących na Z80 programy napisane dla procesora 8080 mogą spotkać niespodzianki wynikające stąd, że w rejestrze F ten sam bit służy do testowania parzystości oraz nadmiaru. Procesor 8080 nie testuje nadmiaru, ale może testować parzystość, w związku z tym programy testujące parzystość po wykonaniu operacji arytmetycznych nie są równoważne dla obu procesorów.

Symbolem *n* będzie oznaczana jednobajtowa stała liczbowa (dwie cyfry szesnastkowe), a *nn* — stała dwubajtowa. Przypominamy, że w rozkazie mniej znaczący bajt takiej stałej, np. adresu, następuje przed bardziej znaczącym.

Symbolem *r*, ewentualnie *r*1, *r*2 będą oznaczane 8-bitowe rejestry A, B, C, D, E, H i L, a także HX, LX, HY i LY w niektórych rozkazach. „Wirtualny rejestr” (HL) w razie potrzeby będzie traktowany osobno, gdyż rozkazy zawierające adresowanie pośrednie trwają dłużej. Oznaczenie 16-bitowych rejestrów nie będzie jednolite, gdyż w zależności od grupy operacji jako dopuszczalny rejestr będzie traktowana para AF, względnie rejestr stosu SP, a ponadto w niektórych rozkazach może wystąpić tylko HL.

Symbolami *ddd* i *sss* będą oznaczane 3-bitowe kody rejestrów.

Listę rozkazów można podzielić na 12 grup:

1. Jednobajtowe rozkazy przesłań.
2. Dwubajtowe rozkazy przesłań.
3. Rozkazy zamiany.
4. Jednobajtowe rozkazy arytmetyczne i logiczne.
5. Rozkazy obrotów i przesunięć.
6. Dwubajtowe rozkazy arytmetyczne.
7. Rozkazy przesłań i przeszukiwania bloków bajtów.
8. Rozkazy sterujące stanem procesora.
9. Rozkazy adresujące pojedyncze bity.
10. Rozkazy skoków.
11. Rozkazy wywołań podprogramów i rozkazy powrotów.
12. Rozkazy wejścia/wyjścia.

5.2. Grupa rozkazów przesyłania jednego bajtu

Tę grupę można podzielić na podgrupy w zależności od użytych trybów adresowania. Należą do niej również rozkazy specjalne, nie należące do ogólnego schematu. Trójbitowy kod *ddd* oznacza rejestr będący *odbiorcą danej* (ang. *destination*), a *sss* – *rejestr źródłowy* (ang. *source*). Kod rejestru 110 (oznaczający HL) w tej podgrupie nie jest używany.

A. Przesłania między rejestrami A, B, C, D, E, H i L

W tej grupie (oraz we wszystkich innych) pierwszy argument zawsze oznacza odbiorcę danej. Tak więc bity *ddd* opisują rejestr *r*1, a *sss* – rejestr *r*2.

Mnemonika: LD *r*1,*r*2 Kod: 01 *ddd sss*

Długość rozkazu w bajtach: 1
Liczba cykli maszynowych: 1
Liczba taktów: 4
Zawartość rejestru F: nie ulega zmianie

Przykład (mnemonika i kod):

LD C,H 4C

(Załaduj do rejestru C zawartość rejestru H) □

Uwagi

Początkujący programiści czasami zapominają, że dopiero wykonanie operacji arytmetycznej lub logicznej ustawia wskaźniki stanu, a np. załadowanie zera do akumulatora wcale nie oznacza, że wskaźnik Z jest ustawiony.

B. Przesłanie między rejestrem a pamięcią w trybie adresowania pośredniego przez HL

Mnemonika: LD *r*,(HL) Kod: 01 *ddd* 110
LD (HL),*r* 01 110 *sss*

Długość rozkazu: 1
Liczba cykli maszynowych: 2
Liczba taktów: 7
Zawartość rejestru F: nie ulega zmianie

Przykład

LD H,(HL) 66
LD (HL),A 77

(Załaduj do rejestru H bajt pamięci adresowany przez HL, a następnie pod adres zawarty w HL — już na ogół inny — prześlij zawartość akumulatora.) □

Uwagi

Z tej grupy jest wyłączone *ddd* lub *sss* równe 110. Kod równy 01 110 110 oznacza HALT.

C. Nieoficjalne rozkazy przesłań między rejestrami, wykorzystujące połówki rejestrów indeksowych: HX, LX, HY, LY

Pierwszy bajt kodu rozkazu (prefiks): DD dla rejestru IX
FD dla rejestru IY

Mnemonika: identyczna jak w podgrupie A
Drugi (podstawowy) bajt kodu: identyczny jak w podgrupie A
(Kod rejestru 100 odpowiada HX lub HY, zaś 101 — LX lub LY.)
Długość rozkazu: 2
Liczba cykli maszynowych: 2
Liczba taktów: 8

Przykład

LD B,HX	DD 44
LD LY,HY	FD 6C

(Załaduj do B zawartość górnej połowy rejestru IX; zawartość górnej połowy IY skopiuj w dolnej połowie IY.) □

Uwagi
Jednoczesne operowanie rejestrami IX i IY lub ich połówkami nie jest możliwe w jednym rozkazie. Nie jest również możliwe jednoczesne operowanie rejestrami H i L oraz połówkami rejestrów indeksowych.

Prefiksowanie rozkazu LD *r1*,*r2* jest ignorowane, gdy kod żadnego z rejestrów nie odpowiada H lub L.

D. Przesłania między rejestrem a pamięcią w trybie adresowania indeksowanego

Prefiks: DD lub FD
Mnemonika: LD *r*,(IX+*d*)
LD (IX+*d*),*r*

i podobnie dla IY. *d* jest jednobajtową liczbą z przedziału −128...+127 nazywaną *przesunięciem.* Dla ujemnych przesunięć jest dopuszczalny zapis zawierający znak odejmowania, np. (IX−4) itp., zaś dla *d* równego zeru większość asemblerów przyjmuje uproszczony zapis: LD *r*,(IX) itp. Faktyczny adres argumentu jest sumą zawartości rejestru indeksowego i przesunięcia.

Podstawowy bajt rozkazu: identyczny jak w podgrupie B
Trzeci bajt rozkazu: przesunięcie
Długość rozkazu: 3
Liczba cykli maszynowych: 5
Liczba taktów: 19
Zawartość rejestru F: nie ulega zmianie

Przykład:

LD A,(IX – 1) DD 46 FF
LD (IY),L FD 75 00

(Załaduj do akumulatora bajt pamięci o adresie o 1 mniejszym niż wartość IX; prześlij zawartość L do pamięci pod adres zawarty w IY.)

Uwagi

Jak widać, wykorzystanie indeksowania jest pamięcio- i czasochłonne. W porównaniu z rozkazami podgrupy B kosztuje to dodatkowe 3 cykle maszynowe (12 taktów) potrzebne na pobranie prefiksu, bajtu przesunięcia i na wykonanie operacji dodawania przesunięcia do zawartości rejestru indeksowego.

W tych rozkazach wystąpienie rejestrów H lub L jest traktowane normalnie.

E. Przesłanie argumentu bezpośredniego do rejestru

Mnemonika: LD *r,n* Kod: 00 *ddd* 110
Drugi bajt: ← *n* →
Długość rozkazu: 2
Liczba cykli maszynowych: 2
Liczba taktów: 7
Zawartość rejestru F: nie ulega zmianie

Przykład

LD B,127 06 7F

(Załaduj do B liczbę 127.) □

F. Przesłanie argumentu bezpośredniego do pamięci w trybie adresowania pośredniego przez HL

Mnemonika: LD (HL),*n* Kod: 00 110 110 (36)
Drugi bajt: ← *n* →
Długość rozkazu: 2
Liczba cykli maszynowych: 3
Liczba taktów: 10
Zawartość rejestru F: nie ulega zmianie

Przykład

LD (HL),10 36 0A

(Liczbę 10 prześlij do pamięci pod adres zawarty w HL.) □

G. Przesłanie argumentu bezpośredniego do pamięci w trybie adresowania indeksowanego

Prefiks: DD lub FD
Mnemonika: LD (IX+*d*),*n*
LD (IY+*d*),*n*
Podstawowy bajt rozkazu: 36, jak w podgrupie F
Długość rozkazu: 4
Liczba cykli maszynowych: 5
Liczba taktów: 19
Zawartość rejestru F: nie ulega zmianie

Przykład LD (IX+1),0 DD 36 01 00
(Wyzeruj bajt pamięci o adresie o 1 większym niż zawartość IX.) □

Uwagi

W następnych punktach większość rozkazów dotyczących rejestrów indeksowych będzie omawiana razem z rozkazami dotyczącymi H, L czy HL ze względu na standardowy sposób ich konstrukcji.

H. Przesłania między akumulatorem a bajtem pamięci adresowanym pośrednio przez rejestry BC lub DE

Mnemonika:	LD (BC),A	Kod:	00 000 010	(02)
	LD A,(BC)		00 001 010	(0A)
	LD (DE),A		00 010 010	(12)
	LD A,(DE)		00 011 010	(1A)

Długość rozkazu: 1
Liczba cykli maszynowych: 2
Liczba taktów: 7
Zawartość rejestru F: nie ulega zmianie

Uwagi

Jest to przykład nieregularności listy rozkazów Z80. Rozkazy te dopuszczają jako rejestr jedynie akumulator i ich format jest inny niż dla analogicznego rozkazu wykorzystującego (HL).

I. Przesłania między akumulatorem a bajtem pamięci adresowanym wprost

Mnemonika:	LD (*nn*),A	Kod:	00 110 010	(32)
Następne bajty:			← *nl* →	
			← *nh* →	
	LD A,(nn)		00 111 010	(3A)
			← *nl* →	
			← *nh* →	

Długość rozkazu: 3
Liczba cykli maszynowych: 4
Liczba taktów: 13
Zawartość rejestru F: nie ulega zmianie

Przykład

LD A,(4097) 3A 01 10

(Prześlij do akumulatora bajt pamięci o adresie 4097.) □

Uwagi

Przez *nl* oznaczyliśmy mniej znaczący (*low*) bajt adresu, przez *nh* — bardziej znaczący (*high*).

J. Przesłania między akumulatorem a rejestrami I i R

Są to rozkazy prefiksowane kodem ED. Podzielimy je na dwie klasy w zależności od zawartości wskaźników stanu.

Prefiks: ED
Mnemonika: LD I,A Kod: 01 000 111 (47)
LD R,A 01 001 111 (4F)
Długość rozkazu: 2
Liczba cykli maszynowych: 2
Liczba taktów: 9
Zawartość rejestru F: nie ulega zmianie
Mnemonika: LD A,I Kod: 01 010 111 (57)
LD A,R 01 011 111 (5F)
Długość i czas trwania rozkazu: jak wyżej

Zawartość rejestru F:

S	Z	H	P/V	N	C
↕	↕	0	IFF2	0	*

Uwagi

Nietypową cechą tych rozkazów w porównaniu z innymi przesłaniami jest modyfikacja rejestru wskaźników stanu. Najbardziej znaczący bit rejestrów I lub R ustawia wskaźnik znaku w rejestrze F, choć bit ten na ogół nie jest interpretowany jako znak. Modyfikowany jest również wskaźnik zera. Do wskaźnika parzystości/nadmiaru przepisywana jest zawartość kopii rejestru blokady przerwań. Przypominamy, że do IFF2 przepisywana jest zawartość zasadniczego rejestru blokady przerwań IFF1 w momencie przyjęcia przerwania niemaskowalnego. W ten sposób podprogram obsługi przerwania niemaskowalnego może sprawdzić, czy przed jego wywołaniem przerwania maskowalne były zamaskowane czy nie. Zawartości rejestru IFF1 bezpośrednio odczytać nie można.

5.3. Grupa rozkazów przesłań dwubajtowych

I tę grupę podzielimy na podgrupy w zależności od użytego trybu adresowania i prefiksu. Jednak ze względu na regularność i prostotę, wszystkie rozkazy wykorzystujące rejestry indeksowe zostaną omówione razem z rozkazami wykorzystującymi HL. Ich długość jest zawsze większa o 1, rozkaz trwa o jeden cykl maszynowy (4 takty) dłużej niż w przypadku HL. Wartości te będą podawane w nawiasach przy odpowiednim opisie.

Żaden rozkaz z tej grupy nie zmienia rejestru F, z wyjątkiem rozkazu POP AF, który ładuje rejestr F zawartością pamięci adresowaną przez stos.

Podobnie jak w przypadku pary HL, w parach BC, DE i AF pierwszy rejestr będzie się odnosił do bardziej znaczącego bajtu argumentu lub adresu.

A. Dwubajtowe przesłania argumentu bezpośredniego do rejestru

Oznaczenia rejestrów:

Rejestr	*dd*
BC	00
DE	01
HL	10
SP	11

Mnemonika: LD *dd*,*nn* Kod: 00 *dd*0 001
← *nl* →
← *nh* →

Długość rozkazu:	3	(4 dla IX lub IY)
Liczba cykli maszynowych:	3	(4)
Liczba taktów:	10	(14)

Przykład

LD BC,65535	01 FF FF
LD IY,100	FD 21 64 00

B. Dwubajtowe przesłania między rejestrem a pamięcią w trybie adresowania wprost

W tej grupie występują powtórzone rozkazy o identycznych funkcjach i różnym formacie. Jak już wspominaliśmy, procesor 8080 dopuszczał jedynie wykorzystanie rejestru HL i rozszerzenie tych rozkazów wymagało użycia prefiksu ED. Tylko oryginalne, nieprefiksowane rozkazy wykorzystujące HL dopuszczają prefiksy DD i FD. Zaczniemy od grupy prefiksowanej bajtem ED.

Prefiks: ED
Mnemonika: LD (*nn*),*dd* Kod: 01 *dd*0 011
← *nl* →
← *nh* →
LD *dd*,(*nn*) 01 *dd*1 011
← *nl* →
← *nh* →

Długość rozkazu: 4
Liczba cykli maszynowych: 6
Liczba taktów 20

Nieprefiksowane rozkazy wykorzystujące HL (rozszerzenie na IX i IY jest standardowe) mają następujący format:

Mnemonika: LD (*nn*,)HL Kod: 00 100 010 (22)
← *nl* →
← *nh* →
LD HL,(*nn*) 00 101 010 (2A)
← *nl* →
← *nh* →

Długość rozkazu: 3 (4 dla IX lub IY)
Liczba cykli maszynowych: 5 (6)
Liczba taktów: 16 (20)

Przykład

LD HL,(1024) 2A 00 04
LD HL,(1024) ED 6B 00 04

(Prześlij do HL zawartość słowa pamięci o adresie 1024 (oraz 1025).) □

C. Rozkazy wykorzystujące rejestr stosu

Do tej grupy należą rozkazy ładowania rejestru stosu SP zawartością rejestru HL lub rejestrów indeksowych (standardowo prefiksowanych) oraz rozkazy przesłań między rejestrem a parą bajtów pamięci adresowaną pośrednio przez SP. Pierwsza podgrupa to:

Mnemonika: LD SP,HL Kod: 11 111 001 (F9)
Długość rozkazu: 1 (2 dla IX lub IY)
Liczba cykli maszynowych: 1 (2)
Liczba taktów: 6 (10)

Przykład

LD SP,IX DD F9

(Skopiuj do rejestru SP zawartość IX.) □

Druga podgrupa umożliwia przesyłanie do pamięci zawartości rejestrów ogólnego przeznaczenia, a także rejestru F.
Oznaczenia rejestrów:

Rejestr	*qq*	
BC	00	
DE	01	
HL	10	(również IX oraz IY)
AF	11	

Mnemonika: POP *qq*		Kod:	11 *qq*0 001
Długość rozkazu:	1	(2 dla IX lub IY)	
Liczba cykli maszynowych:	3	(4)	
Liczba taktów:	10	(14)	

Mnemonika: PUSH *qq*		Kod:	11 *qq*0 101
Długość rozkazu:	1	(2 dla IX lub IY)	
Liczba cykli maszynowych:	3	(4)	
Liczba taktów:	11	(15)	

Przykład

PUSH DE   D5
POP IX   DD E1 □

Uwagi
Podczas wykonania rozkazu PUSH pierwszą operacją jest zmniejszenie zawartości rejestru SP o 2 (pozycja [7] na str. 71 omyłkowo podaje wartość 1). Następnie zawartość dwubajtowego rejestru źródłowego jest przesyłana do pamięci, mniej znaczący bajt pod adres będący zawartością SP, bardziej znaczący pod SP + 1, tak jakby procesor wykonywał nieistniejący rozkaz: „LD (SP),*qq*". PUSH AF pozwoli przesłać do pamięci zawartość rejestru F, a następnie np. odczytać bity nr 3 lub 5.

Operacja POP wykonuje operację odwrotną. Najpierw przesyła dwa bajty z pamięci do rejestru: „LD *qq*,(SP)", a następnie zwiększa zawartość SP o 2. Przez POP AF można jawnie przesłać dowolny bajt do rejestru wskaźników stanu, co może służyć do symulowania sytuacji, które w rzeczywistości nie miały miejsca, np. wystąpienie nadmiaru.

5.4. Grupa rozkazów zamiany

W tej grupie znajdują się rozkazy dotyczące alternatywnego banku rejestrów, a ponadto rozkazy, które zostały wprowadzone ze względu na efektywność, choć można je zastąpić sekwencjami innych rozkazów przesyłania. Zawartość rejestru F nie ulega zmianie, z wyjątkiem rozkazu EX AF,AF'.

A. Rozkaz wymiany zawartości DE i HL

Mnemonika: EX DE,HL Kod: 11 101 011 (EB)
Długość rozkazu: 1
Liczba cykli maszynowych: 1
Liczba bajtów: 4

Uwagi
Rozkaz ten nie korzysta z innych rejestrów ani z pamięci danych i daje dużą oszczędność czasu i pamięci, gdy trzeba np. chwilowo zapamiętać zawartość HL, co jest często przydatne w dwubajtowych obliczeniach arytmetycznych. Jak już zostało zaznaczone, autorowi nie jest znany model Z80, który reaguje na prefiks DD lub FD.

B. Wymiana zawartości HL z parą bajtów leżącą na wierzchołku stosu

Ten rozkaz można prefiksować przez DD lub FD.
Mnemonika: EX (SP),HL Kod: 11 100 011 (E3)
Długość rozkazu: 1 (2 dla IX lub IY)
Liczba cykli maszynowych: 5 (6)
Liczba taktów: 19 (23)

Przykład

EX (SP),IY FD E3

(Wymień parę bajtów pamięci adresowaną przez SP z zawartością rejestru IY.) □

Uwagi
Zawartość rejestru L jest wymieniana z (SP), a H z (SP + 1). Sam rejestr SP nie ulega zmianie. Rozkaz ten jest użyteczny w konstrukcji złożonych struktur sterowania: podprogramów z parametrami, podprogramów o dynamicznie obliczanych adresach, współprogramów itp.

C. Rozkazy dotyczące alternatywnego banku rejestrów

W tej grupie znajdują się dwa rozkazy, jeden dotyczy rejestrów A i F, drugi zaś pozostałych: B, C, D, E, H i L.

Mnemonika: EX AF,AF' Kod: 00 001 000 (08)
Długość rozkazu: 1
Liczba cykli maszynowych: 1
Liczba taktów: 4
Zawartość rejestru F: zmienia się odpowiednio

Mnemonika: EXX Kod: 11 011 001 (09)
Długość rozkazu: 1
Liczba cykli maszynowych: 1
Liczba taktów: 4
Zawartość rejestru F: nie ulega zmianie

Uwagi

Jak już wspominaliśmy, rozkazy zamiany należy traktować raczej jako przedefiniowanie nazw rejestrów A na A', B na B' itd. niż jako rzeczywiste przepisywanie ich zawartości. Znajdują one zastosowanie, gdy trzeba szybko zapamiętać stan danych w procesorze, np. przy wywoływaniu krytycznych podprogramów lub obsłudze przerwań (jednopoziomowej).

5.5. Grupa 8-bitowych rozkazów arytmetycznych i logicznych

Ta grupa rozkazów stanowi podstawę prawie wszystkich obliczeń i jest niewiele zastosowań mikroprocesora, które nie wykorzystują rozkazów z tej grupy. Niektóre z nich, np. rozkaz dodawania ADD, mają swoje odpowiedniki 16-bitowe noszące tę samą nazwę mnemoniczną. Inne, np. odejmowanie bez pożyczki SUB, są wyłącznie 8-bitowe. W związku z tym, w rozkazach o jednoznacznych nazwach standardowa notacja asemblera Z80 opuszcza jawne określenie rejestru A jako jednego z argumentów i miejsca przesłania wyniku, gdyż obowiązuje to wszystkie rozkazy (niejawny tryb adresowania). Natomiast w przypadku dwuznaczności nazwa A w rozkazie figuruje. Ze względu na czytelność i konsystencję, stosowanie tej konwencji nie wydaje się najlepsze, podporządkujemy się jej, aby zachować zgodność ze standardem.

Konstrukcja tej grupy rozkazów jest dość regularna i opis pierwszej podgrupy nie będzie musiał być dalej powtarzany. Środkowe trzy bity podstawowego bajtu rozkazu (segment *M*) określają rodzaj wykonywanej operacji, według następującego schematu:

Operacja	bbb	
ADD	000	
ADC	001	(ang. *add with carry*)
SUB	010	(ang. *subtract*)
SBC	011	(ang. *subtract with carry*)
AND	100	
XOR	101	
OR	110	
CP	111	(ang. *compare*)

A. Rozkazy dodania zawartości rejestru do akumulatora

W tej grupie rozkazów rejestrem oznaczonym w kodzie rozkazu przez *sss* może być A, B, C, D, E, H, L, połówka rejestru indeksowego, (HL), (IX+*d*) oraz (IY+*d*). Rozkazy korzystające z rejestru indeksowego są standardowo prefiksowane przez DD lub FD. Rozkazy wykorzystujące adresowanie indeksowane zawierają trzeci bajt — odpowiednie przesunięcie. W tej podgrupie znajdą się dwa rozkazy: ADD, który tylko dodaje zawartość rejestru do akumulatora oraz ADC, który dodatkowo dodaje jeszcze bit przeniesienia.

Mnemonika: ADD A,*r* Kod: 10 000 *sss*
ADC A,*r* 10 001 *sss*

Długość rozkazu: 1 dla rejestru wewnętrznego i (HL)
2 dla HX itp.
3 dla (IX+*d*) itp.

Liczba cykli maszynowych: 1 dla rejestru wewnętrznego
2 dla HX itp. oraz (HL)
5 dla (IX+*d*) itp.

Liczba taktów: 4 dla rejestru wewnętrznego
7 dla (HL), 8 dla HX itp.
19 dla (IX+*d*) itp.

Zawartość rejestru F:

S	Z	H	P/V	N	C
↕	↕	↕	V	0	↕

Przykład

ADD A,A 87
ADC A, (IY+56) FD 8E 38
ADC A, LX DD 8D

(Drugi: dodaj do akumulatora zawartość bitu nr 0 rejestru F i bajtu pamięci o adresie będącym sumą zawartości IY i 56.) □

Uwagi

Symbol V na pozycji bitu parzystości/nadmiaru oznacza, że bit ten jest ustawiany, jeśli wystąpił nadmiar, tj, nastąpiło przeniesienie z bitu 6 do 7, które zmieniło bit 7.

B. Rozkazy odejmowania zawartości rejestru od akumulatora i rozkazy porównania

W tej podgrupie mamy trzy rozkazy: SUB — wykonujący zwykłe 8-bitowe odejmowanie w dwójkowym kodzie uzupełnieniowym; SBC — rozkaz odejmowania z pożyczką, w którym dodatkowo odejmuje się zawartość wskaźnika przeniesienia; CP — rozkaz porównania, który z punktu widzenia pracy arytmometru i ustawiania wskaźników stanu jest równoważny SUB, jednak wynik odejmowania nie jest nigdzie zapamiętywany, w szczególności nie modyfikuje akumulatora. Rozkaz CP służy do sprawdzania relacji równości, mniejszości itp. między danymi.

Mnemonika:	SUB *r*	Kod:	10010 *sss*
	SBC A,*r*		10011 *sss*
	CP *r*		10111 *sss*

Długość i czas trwania rozkazu: identyczny jak w przypadku ADD i ADC.
Zawartość rejestru F:

S	Z	H	P/V	N	C
↕	↕	↕	V	1	↕

Przykłady

SBC A,A	9F
CP (HL)	BE

(Drugi: odejmij od akumulatora zawartość bajtu pamięci adresowanego przez HL bez zapamiętania wyniku.) □

Uwagi

Rozkaz SUB A może służyć do szybszego zerowania akumulatora niż rozkaz LD A,0. Rozkaz SBC A,A zeruje akumulator, jeśli wskaźnik przeniesienia był wyzerowany, w przeciwnym razie zawartość A wyniesie #FF. Rozkaz CP A ustawia wskaźnik zera nie zmieniając innych rejestrów.

Sytuację, w której generuje się nadmiar i przeniesienie najprościej zrozumieć traktując odejmowanie jako dodawanie liczby z przeciwnym znakiem.

Po wykonaniu rozkazów odejmowania wskaźnik N oznaczający odejmowanie jest ustawiany. Jest to potrzebne, aby rozkaz korekcji dziesiętnej był wykonany prawidłowo.

C. Rozkazy operacji logicznych między akumulatorem a rejestrem

W tej podgrupie mamy rozkazy: 8-bitowej sumy logicznej OR, iloczynu AND oraz różnicy symetrycznej XOR.

Mnemonika:		Podst. kod:	
	AND *r*		10 100 *sss*
	XOR *r*		10 101 *sss*
	OR *r*		10 110 *sss*

Długość rozkazu i czas trwania: jak poprzednio

Zawartość rejestru F:

	S	Z	H	P/V	N	C
dla rozkazu AND	↕	↕	1	P	0	0
dla OR oraz XOR	↕	↕	0	P	0	0

Przykłady

XOR A	AF
AND (IX+254)	DD A6 FE

(Drugi: pomnóż logicznie akumulator przez bajt pamięci o adresie o 2 mniejszym niż zawartość IX.) □

Uwagi

Rozkaz XOR A również może służyć do szybkiego zerowania akumulatora. (Oczywiście rejestr F jest ustawiany inaczej niż w przypadku SUB A.) Rozkazy OR A i AND A, które zachowują zawartość akumulatora nie zmienioną mogą służyć do zerowania wskaźnika przeniesienia oraz do testowania zera lub znaku liczby przesłanej do akumulatora.

Ogólnie, operacje logiczne oprócz generowania warunków logicznych mogą służyć również do maskowania, tj. wycinania fragmentów bajtu z rejestru lub pamięci.

Symbol P w pozycji wskaźnika parzystości/nadmiaru oznacza, że wskaźnik ten jest ustawiany w zależności od parzystości wyniku.

Zachowanie się wskaźników N (odejmowania) i H (pomocniczego przeniesienia) w operacjach logicznych na ogół nie jest specjalnie ważne ani interesujące. Pod tym względem jednak w literaturze panuje bałagan: autor spotkał się z 5 różnymi wariantami, co można rozmaicie interpretować. Dane przedstawione w pracach [7] i [9] nie w pełni zgadzają się z materiałami publikowanymi przez firmę Zilog, z testami wykonanymi przez autora ani ze sobą.

D. Rozkazy operacji arytmetycznych i logicznych wykorzystujących jednobajtowy argument bezpośredni

W tym punkcie zbierzemy razem operacje dodawania, odejmowania, porównania oraz operacje logiczne. Podobnie jak w podgrupach A, B i C środkowe trzy bity kodu rozkazu identyfikują operację według schematu podanego na początku p. 5.5.

Mnemonika:	ADD A,*n*	Kod:	11 000 110 (C6)
	ADC A,*n*		11 001 110 (CE)
	SUB *n*		11 010 110 (D6)
	SBC A,*n*		11 011 110 (DE)
	AND *n*		11 100 110 (E6)
	XOR *n*		11 101 110 (EE)
	OR *n*		11 110 110 (F6)
	CP *n*		11 111 110 (FE)

Drugi bajt rozkazu: ← *n* →
Długość rozkazu: 2
Liczba cykli maszynowych: 2
Liczba taktów: 7
Zawartość rejestru F: identyczna jak w grupach A, B i C

Przykład:

ADD A,20 C6 14
CP 1 FE 01

(Dodaj 20 do akumulatora i sprawdź, czy wynik jest większy, mniejszy czy równy 1 (tj. 257).) □

E. Rozkazy zwiększenia (inkrementacji) i zmniejszenia (dekrementacji) o 1 zawartości rejestru

W tej grupie rejestrem może być dowolny rejestr 8-bitowy łącznie z połówkami rejestrów indeksowych, a także (HL), (IX + *d*) i (IY + *d*) zgodnie ze standardowym schematem prefiksowania. Kod rejestru jest oznaczony przez *ddd*.

Mnemonika:	INC *r*	Kod:	00 *ddd* 100
	DEC *r*		00 *ddd* 101

Długość rozkazu: 1 (2 dla HX itp., 3 dla (IX + *d*) itp.)
Liczba cykli maszynowych: 1 (2 dla HX itp., 3 dla (HL)
6 dla (IX + *d*) itp.)
Liczba taktów: 4 (8 dla HX itp., 11 dla (HL)
23 dla (IX + *d*) itp.)

Zawartość rejestru F:

		S	Z	H	P/V	N	C
Dla rozkazów	INC	↕	↕	↕	V	0	*
Dla rozkazów	DEC	↕	↕	↕	V	1	*

Przykłady

INC L	2C
DEC LX	DD 2D
DEC (IY−8)	FD 35 F8

(Trzeci: zmniejsz o 1 zawartość bajtu pamięci o adresie o 8 mniejszym niż wartość IY.) □

Uwagi

Dość istotną różnicą między tymi rozkazami a pozostałymi rozkazami arytmetycznymi jest brak modyfikacji wskaźnika przeniesienia.

F. Pomocnicze rozkazy modyfikujące akumulator i wskaźnik przeniesienia

W tej podgrupie mamy rozkaz CPL (ang. *complement*), który odwraca wszystkie bity rejestru A na przeciwne (uzupełnienie do 255) oraz rozkaz NEG (ang. *negate*), który zmienia znak liczby w akumulatorze. W dwójkowym kodzie uzupełnieniowym CPL i NEG nie są równoważne, jednak zmianę znaku akumulatora można otrzymać wykonując CPL, a następnie dodając 1. Rozkaz NEG został wprowadzony dla wygody, a nie dla efektywności, gdyż jest rozkazem prefiksowanym o zwiększonej długości i czasie wykonywania.

Bardzo użyteczną możliwością jest bezpośrednie sterowanie wskaźnikiem przeniesienia przez programistę. Jawny rozkaz zerowania wskaźnika przeniesienia nie istnieje, można to osiągnąć wykonując operację OR A lub AND A. Procesor Z80 dysponuje natomiast rozkazem ustawiania wskaźnika przeniesienia: SCF (ang. *set carry flag*) oraz rozkazem jego odwrócenia: CCF (ang. *complement carry flag*).

Mnemonika: CPL Kod: 00 101 111 (2F)
Długość rozkazu: 1
Liczba cykli maszynowych: 1
Liczba taktów: 4
Zawartość rejestru F:

S	Z	H	P/V	N	C
*	*	1	*	1	*

Prefiks: ED
Mnemonika: NEG Kod: 01 000 100 (44)
Długość rozkazu: 2

Liczba cykli maszynowych: 2
Liczba taktów: 8
Zawartość rejestru F:

S	Z	H	P/V	N	C
↕	↕	↕	V	1	↕

Mnemonika: SCF Kod: 00 110 111 (37)
CCF 00 111 111 (3F)

Długość rozkazu: 1
Liczba cykli maszynowych: 1
Liczba taktów: 4
Zawartość rejestru F:

	S	Z	H	P/V	N	C
Dla rozkazu SCF:	*	*	0	*	0	1
Dla rozkazu CCF:	*	*	?	*	0	↕

Uwagi

Zawartość wskaźnika pomocniczego przeniesienia H po wykonaniu rozkazu CCF jest często w tabelach podawana jako nieokreślona lub nieznana. Według testów autora procesor umieszcza tam poprzednią zawartość wskaźnika C.

G. Rozkaz korekcji dziesiętnej

Jest to rozkaz DAA (ang. *decimal adjust accumulator*), który umożliwia wykonywanie obliczeń w kodzie BCD. Wszystkie operacje dodawania i odejmowania są wykonywane dwójkowo. Jeśli jednak wynik dodawania tetrad jest liczbą większą niż 9, np.:

$$\begin{array}{r} 0101 \\ +0111 \\ \hline 1100 \end{array}$$

należy od niego odjąć dziesięć, a jedynkę przenieść do następnej tetrady. Sprowadza się to po prostu do dodania liczby 6:

$$\begin{array}{r} 1100 \\ +0110 \\ \hline 10010 \end{array}$$

Jeśli dodawanie samo wygenerowało przeniesienie, np. $9+8 = 17 = 16+1$, dodanie 6 również odtwarza prawidłowy wynik BCD: $16+7$. Oczywiście korekcję należy przeprowadzić i dla bardziej znaczącej tetrady. Teraz jawnie widać zastosowanie wskaźnika H. Korekcja, tj. dodanie 6 jest wykonywana, jeśli wynik dodawania jest większy lub równy 10, tj. w tetradzie ustawione są bity 3 i 1 lub 3 i 2, lub jeśli ustawiony jest wskaźnik H.

Nieco inaczej przebiega korekcja w przypadku odejmowania, co jest rozpoznawane przez ustawienie wskaźnika N. Wtedy od tetrady należy odjąć 6, jeśli wystąpiło przeniesienie, np.:

$$\begin{array}{r} 0011 \\ -1000 \\ \hline 1011, \end{array}$$

tj. dziesiętnie: (1)3 − 8 = 5 = 11 − 6.

Mnemonika: DAA Kod: 00 100 111 (27)
Długość rozkazu: 1
Liczba cykli maszynowych: 1
Liczba taktów: 4
Zawartość rejestru F:

S	Z	H	P/V	N	C
↕	↕	↕	P	*	↕

Uwagi
Rozkaz DAA jedynie poprawia wynik po wykonaniu operacji arytmetycznej. Nie służy do zamieniania liczb zapisanych dwójkowo na liczby w kodzie BCD! Wszystkie liczby biorące udział w obliczeniach w kodzie BCD powinny od początku być prawidłowo zapisane. Metody konwersji z postaci dwójkowej na BCD i odwrotnie zostaną podane w następnych rozdziałach.

5.6 Grupa rozkazów przesunięć

Grupa ta stanowi naturalne uzupełnienie rozkazów arytmetycznych i logicznych. Rozkazy przesuwające o 1 lub więcej bitów zawartość rejestrów są niezbędne w niektórych technikach przesyłania danych z i do komputera (sterowana programowo transmisja szeregowa) oraz do wykonywania operacji mnożenia i dzielenia.

Z80 dysponuje bogatym zestawem rozkazów przesunięć logicznych, arytmetycznych, cyklicznych 8-bitowych i 9-bitowych (wykorzystujących wskaźnik przeniesienia jako dziewiąty bit rejestru). Procesor 8080 dopuszczał jedynie przesunięcia zawartości akumulatora. Te rozkazy istnieją nadal w Z80, ale oprócz nich jest znacznie bogatsza grupa rozkazów prefiksowanych umożliwiających przesunięcia zawartości wszystkich rejestrów. Ta podgrupa powtarza rozkazy z podgrupy pierwszej, jednak występują istotne różnice w zachowaniu się rejestru wskaźników stanu, tak że te podgrupy wymagają osobnego omówienia.

W tej grupie rozkazów dla jednoznaczności będziemy operować terminem *przesunięcia* (ang. *shift*) w przypadku, gdy bit przesunięty poza rejestr, tj. bit nr 7 przy przesunięciach w lewo lub bit nr 0 przy przesunięciach w prawo, jest

„gubiony". Natomiast przesunięcia cykliczne, w których bit przesunięty poza rejestr wraca do niego z przeciwnej strony będą nazywane *obrotami* (ang. *rotations*).

Konstrukcja rozkazów w tej grupie jest dość regularna. Trzy środkowe bity kodu rozkazu specyfikują operację według poniższego schematu (nazwy tych operacji są bezpośrednio nazwami rozkazów prefiksowanych przez ED, natomiast rozkazy dotyczące wyłącznie akumulatora zawierają jeszcze jednoliterowy sufiks A).

Operacja	Kod: *ooo*	
RLC	000	(ang. *rotate left circular*)
RRC	001	(ang. *rotate right circular*)
RL	010	(ang. *rotate left*)
RR	011	(ang. *rotate right*)
SLA	100	(ang. *shift left arithmetic*)
SRA	101	(ang. *shift right arithmetic*)
* SLI	110	(ang. *shift left and increment*)
SRL	111	(ang. *shift right logical*)

Operacje RLC i RL przesuwają cyklicznie zawartość rejestru w lewo, a RRC i RR w prawo. Operacje RLC i RRC wykonują obroty na 8 bitach rejestru, a bit przesunięty poza rejestr oprócz powrotu z drugiej strony jest dodatkowo wpisywany do wskaźnika przeniesienia. Natomiast operacje RL i RR traktują wskaźnik przeniesienia jako dziewiąty bit rejestru.

Przesunięcia dzielą się na logiczne i arytmetyczne, przy czym różnica jest istotna dla przesunięć w prawo. Przy przesunięciu w lewo, tj. przy operacji SLA, do zerowego bitu rejestru jest wpisywane zero. Operacja SLI różni się od SLA tym, że do bitu zerowego wpisywane jest 1. Operacja SLI wyróżniona w powyższej tabeli gwiazdką nie figuruje na oficjalnej liście rozkazów. Jej użyteczność wydaje się zresztą nieco mniejsza niż pozostałych.

Przy przesunięciu logicznym w prawo do bitu nr 7 rejestru jest wpisywane zero. Natomiast przy przesunięciu arytmetycznym wartość tego bitu pozostaje nie zmieniona. Wiąże się to z interpretacją tego bitu jako znaku liczby. Ponieważ podstawowym zastosowaniem tej operacji jest dzielenie liczby przez 2 (i wyższe potęgi dwójki), takie ustalenie daje prawidłowy wynik również dla liczb ujemnych.

Oprócz powyższych operacji w tej grupie znajdą się jeszcze dwa rozkazy cyklicznych przesunięć tetrad użyteczne w konwersji i formatowaniu liczb w kodzie BCD, a także w konwersji liczb dwójkowych z postaci wewnętrznej na ciągi cyfr szesnastkowych.

A. Rozkazy obrotów akumulatora

W tej podgrupie mamy tylko 4 rozkazy.

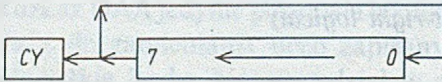
Mnemonika:	RLCA	Kod:	00 000 111	(07)
	RRCA		00 001 111	(0F)
	RLA		00 010 111	(17)
	RRA		00 011 111	(1F)

Długość rozkazu: 1
Liczba cykli maszynowych: 1
Liczba taktów 4
Zawartość rejestru F:

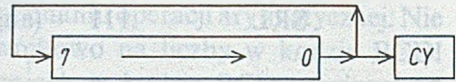
S	Z	H	P/V	N	C
*	*	0	*	0	↕

Uwagi

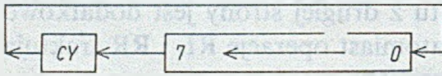
Najwygodniej zrozumieć działanie tych operacji analizując rys. 5.1, 5.2, 5.3, 5.4.



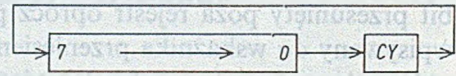
Rys. 5.1. Przepływ informacji podczas wykonania rozkazu RLC



Rys. 5.2. Przepływ informacji podczas wykonania rozkazu RRC



Rys. 5.3. Przepływ informacji podczas wykonania rozkazu RL



Rys. 5.4. Przepływ informacji podczas wykonania rozkazu RR

B. Rozszerzone rozkazy obrotów i przesunięć

Wszystkie rozkazy tej podgrupy są prefiksowane przez CB. Argumentem operacji oznaczanym kodem *sss* może być dowolny rejestr 8-bitowy, (HL) oraz (IX + *d*) i (IY + *d*) zgodnie ze standardowym schematem prefiksowania przez DD i FD, jednak bez połówek rejestrów indeksowych.

Prefiks: CB

Mnemonika:	RLC *r*	Kod:	00 000 *sss*
	RRC *r*		00 001 *sss*
	RL *r*		00 010 *sss*
	RR *r*		00 011 *sss*
	SLA *r*		00 100 *sss*
	SRA *r*		00 101 *sss*
	SLI *r*		00 110 *sss*
	SRL *r*		00 111 *sss*

Długość rozkazu:	2	(3 dla indeksowanych)
Liczba cykli maszynowych:	4	(6 dla indeksowanych)
Liczba taktów:	15	(23 dla indeksowanych)

Zawartość rejestru F:

S	Z	H	P/V	N	C
↕	↕	0	P	0	↕

Przykłady

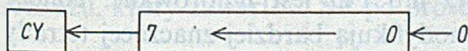
RLC A	CB 07
SLI H	CB 34
RR (IY+12)	FD CB 1D 0C
RRC (IX),H	DD CB 0C 00

(Czwarty: przesuń cyklicznie w prawo zawartość bajtu pamięci adresowanego przez IX wykorzystując bit przeniesienia jako wydłużenie. Wynik dodatkowo wpisz do rejestru H. Na przykład jeśli (IX) zawierało # 6A, tj. 106, a wskaźnik C był ustawiony, wynikiem będzie # B5 (181), a C zostanie wyzerowane.) □

Uwagi

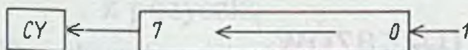
Poprzedzenie rozkazu np. RRC H prefiksem DD nie zmienia go w rozkaz „RRC HX". Jest to odejście od dotychczasowych nieoficjalnych rozszerzeń rozkazów dotyczących rejestrów H i L. Rozkaz ten zostanie wtedy zinterpretowany jako RRC (IX+*d*),H — inne nieoficjalne rozszerzenie. Następny bajt po podstawowym zostanie zinterpretowany jako przesunięcie, operacja zostanie wykonana na zawartości odpowiedniej komórki pamięci, a wynik dodatkowo zostanie przesłany do rejestru H. Podobnie zachowują się i inne rozkazy tej grupy.

Przesyłanie informacji w operacjach przesunięć obrazują rysunki 5.5, 5.6, 5.7 i 5.8.

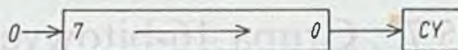


Rys. 5.5. Przepływ informacji podczas wykonania rozkazu SLA

Rys. 5.6. Przepływ informacji podczas wykonania rozkazu SRA



Rys. 5.7. Przepływ informacji podczas wykonania rozkazu SLI

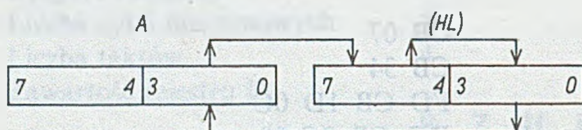


Rys. 5.8. Przepływ informacji podczas wykonania rozkazu SRL

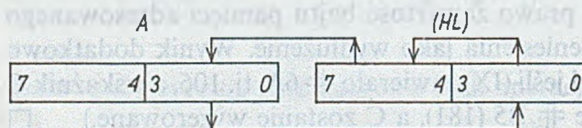
C. Rozkazy obrotów tetrad (cyfr dziesiętnych)

Poprzednie rozkazy przesuwały zawartość rejestru tylko o 1 bit w lewo lub prawo. W tej podgrupie mamy dwa rozkazy, które dokonują przesunięcia od

razu o 4 bity, ale ponieważ argumenty operacji są inne niż w powyższych przypadkach, rozkazów tych nie da się prosto zastąpić przez 4-krotne powtórzenie jednego z rozkazów omówionych w podgrupie B. Te rozkazy to RRD (ang. *rotate right digit*) i RLD (ang. *rotate left digit*), które wymieniają tetrady między prawą połową akumulatora a bajtem pamięci adresowanym przez HL. Schemat tych obrotów jest przedstawiony na rys. 5.9 i 5.10. Rozkazy RRD i RLD są prefiksowane przez ED.



Rys. 5.9. Przepływ informacji podczas wykonania rozkazu RLD



Rys. 5.10. Przepływ informacji podczas wykonania rozkazu RRD

Prefiks: ED
Mnemonika: RRD Kod: 01 100 111 (67)
RLD 01 101 111 (6F)
Długość rozkazu: 2
Liczba cykli maszynowych: 5
Liczba taktów: 18
Zawartość rejestru F:

S	Z	H	P/V	N	C
↕	↕	0	P	0	*

Uwagi
Prefiksowanie tych rozkazów przez DD lub FD jest ignorowane. Należy zauważyć, że rozkazy RRD i RLD nie modyfikują bardziej znaczącej tetrady akumulatora.

5.7. Grupa 16-bitowych rozkazów arytmetycznych

W tej grupie rolę akumulatora pełni rejestr HL, a dla operacji dodawania także jeden z rejestrów indeksowych zgodnie ze standardowym schematem prefiksowania. Drugim argumentem operacji może być dowolny 16-bitowy rejestr ogólnego przeznaczenia albo rejestr stosu. Rejestr ten będzie oznaczany przez *rr*

jako symboliczny argument operacji i kodowany dwoma bitami *ss* według następującego schematu:

Rejestr *rr*	kod: *ss*
BC	00
DE	01
HL	10 (również IX lub IY)
SP	11

A. Rozkaz dodawania

Mnemonika: ADD HL,*rr* Kod: 00 *ss*1 001
ADD IX,*rr*
ADD IY,*rr*

Długość rozkazu: 1 (2 dla IX i IY)
Liczba cykli maszynowych: 3 (4)
Liczba taktów: 11 (15)
Zawartość rejestru F:

S	Z	H	P/V	N	C
*	*	?	*	0	↕

Przykłady

ADD HL,BC 09
ADD IY,IY FD 29
ADD HL,SP 39 □

Uwagi
Wskaźnik H jest ustawiany przeniesieniem z bitu nr 11 rejestru HL. Operacja dodawania zawartości rejestru stosu do HL lub rejestru indeksowego jest jedynym sposobem pobrania tej zawartości do dalszych operacji. Nie ma rozkazu, który zawartość SP po prostu przesyła.

B. Rozkazy dodawania z przeniesieniem i odejmowania z pożyczką

Te rozkazy są prefiksowane przez ED i nie rozszerzają się na rejestry indeksowe

Prefiks: ED
Mnemonika: ADC HL,*rr* Kod: 01 *ss*1 010
SBC HL,*rr* 01 *ss*0 010
Długość rozkazu: 2
Liczba cykli maszynowych: 4
Liczba taktów: 15

Zawartość rejestru F:

		S	Z	H	P/V	N	C
Dla rozkazu	ADC:	↕	↕	?	V	0	↕
Dla rozkazu	SBC:	↕	↕	?	V	1	↕

Przykłady

ADC HL,DE	ED 5A
SBC HL,SP	ED 72

□

Uwagi

Rozkazy ADC i SBC są dłuższe niż rozkaz ADD, ale wygodniejsze do wykonywania obliczeń na liczbach 16-bitowych ze względu na ustawianie wskaźników znaku i zera. Głównym zastosowaniem rozkazu ADD są operacje arytmetyczne na adresach, dla których wskaźniki znaku i zera nie mają znaczenia. Wskaźnik H jest ustawiany podobnie jak w przypadku ADD.

C. Rozkazy inkrementacji i dekrementacji

Te rozkazy mają podstawowy kod jednobajtowy i dopuszczają prefiksy DD lub FD.

Mnemonika:	INC *rr*		Kod: 00 *ss*0 011
	DEC *rr*		00 *ss*1 011
Długość rozkazu:		1	(2 dla IX lub IY)
Liczba cykli maszynowych:		1	(2)
Liczba taktów:		6	(10)

Zawartość rejestru F:

S	Z	H	P/V	N	C
*	*	*	*	*	*

Przykłady

INC BC	03
DEC IX	DD 2B
INC SP	3B

(Trzeci: wycofaj ze stosu ostatni bajt. Rzadko wystąpi samodzielnie, gdyż na ogół na stosie umieszcza się całe słowa.) □

Uwagi

Nieprzyjemną cechą tych rozkazów jest brak modyfikacji wskaźników, co utrudnia ich wykorzystanie do organizacji pętli. Na przykład, żeby sprawdzić, czy zawartość HL po dekrementacji jest równa zeru, trzeba sprawdzić osobno H i L przy użyciu akumulatora.

5.8. Grupa rozkazów przesyłania i przeszukiwania bloków

Przesyłanie większych bloków danych (np. tablic), przemieszczanie programów itp. jest dość często używaną operacją. Podobnie często spotyka się przeszukiwanie tablic w celu znalezienia określonego bajtu lub grupy bajtów (np. fragmentu tekstu). Wymaga to zorganizowania odpowiednich pętli, co może być nieco uciążliwe. Lista rozkazów procesora zawiera kilka interesujących ułatwień. Należą do nich rozkazy przesyłania bajtu danych z automatyczną inkrementacją lub dekrementacją zawartości rejestru HL i DE: LDI i LDD; rozszerzenie rozkazów porównania CP również automatycznie modyfikujących zawartość HL i DE: CPI i CPD; a także rozkazy będące sprzętową realizacją pętli przesyłających cały blok danych: LDIR i LDDR lub przeszukujących blok według wzrastających lub malejących adresów: CPIR i CPDR.

A. Rozkazy przesyłania z automatyczną inkrementacją lub dekrementacją i rozkazy przesyłania bloków

Przed wykonaniem rozkazów tej grupy program powinien zapewnić prawidłową zawartość następujących rejestrów:

HL — adres pierwszego bajtu przesyłanego bloku,
DE — adres pierwszego bajtu miejsca przeznaczenia,
BC — liczba przesyłanych bajtów.

Wykonanie rozkazu LDI (ang. *load and increment*) składa się z następujących kroków:

1. Bajt pamięci adresowany zawartością HL jest przesyłany pod adres zawarty w DE.

2. Zawartości HL oraz DE zostają zwiększone o 1.

3. Zawartość BC zostaje zmniejszona o 1.

Jeśli przesyłanie kolejnego bajtu nie wymaga dodatkowych operacji testujących lub przetwarzających, to wygodniejsze jest użycie rozkazu LDIR (ang. *load-increment-repeat*), w którym oprócz powyższych trzech kroków jest wykonywane dodatkowo:

4. Zawartość BC jest sprawdzana. Jeśli wynosi zero, program przechodzi do wykonania następnego rozkazu, w przeciwnym razie kroki 1, 2, 3 i 4 są powtarzane.

Rozkazy LDD (ang. *load and decrement*) i LDDR (ang. *load-decrement-repeat*) różnią się od LDI i LDIR tym, że zawartość rejestrów HL i DE jest zmniejszana o 1. Użycie LDIR i LDDR nie jest równoważne, gdy zachodzi

częściowe pokrywanie się bloku źródłowego i obszaru przeznaczenia. Na przykład aby przesunąć blok danych o 1 bajt w górę przestrzeni adresowej należy przesuwać blok od końca wykorzystując rozkaz LDDR. Użycie LDIR w tym kontekście spowoduje zapisanie całego bloku zawartością pierwszego bajtu. To też oczywiście może znaleźć zastosowanie.

Wszystkie rozkazy tej grupy są prefiksowane przez ED.

Prefiks: ED
Mnemonika: LDI Kod: 10 100 000 (A0)
LDD 10 101 000 (AB)
Długość rozkazu: 2
Liczba cykli maszynowych: 4
Liczba taktów: 16
Zawartość rejestru F:

S	Z	H	P/V	N	C
*	*	0	↕	0	*

Uwagi

Wskaźnik P/V jest ustawiany w zależności od zawartości rejestru BC. Jeśli po wykonaniu rozkazu zawartość BC jest równa zeru wskaźnik P/V jest zerowany, w przeciwnym wypadku ustawiany.

Prefiks: ED
Mnemonika: LDIR Kod: 10 110 000 (B0)
LDDR 10 111 000 (B8)
Długość rozkazu: 2
Liczba cykli maszynowych: 4 gdy zawartość BC = 0
5 w przeciwnym wypadku
Liczba taktów: 16 gdy zawartość BC = 0
21 w przeciwnym wypadku
Zawartość rejestru F:

S	Z	H	P/V	N	C
*	*	0	0	0	*

Uwagi

Należy teraz wyjaśnić co oznaczają podane wyżej czasy wykonania i odpowiedzieć na pytanie, co się stanie, jeśli podczas wykonywania rozkazu nastąpi przerwanie.

Organizacja wykonanią tych rozkazów jest bardzo prosta i nie wymaga ani wyrafinowanego mikroprogramu, ani dużej komplikacji sprzętowej. Układ sterujący procesora po wykonaniu LDI lub LDD testuje wskaźnik P/V i jeśli jest on ustawiony, tj. przesłanie bloku nie zostało zakończone, licznik rozkazów ulega cofnięciu o 4. W ten sposób rozkaz LDIR lub LDDR zostaje ponownie pobrany z

pamięci do wykonania. W związku z tym nie należy się spodziewać jakichś kolosalnych oszczędności czasowych w porównaniu z jawnie zaprogramowaną pętlą.

W ten sam sposób są zorganizowane omawiane niżej rozkazy przeszukiwań bloków oraz rozkazy blokowej transmisji między pamięcią a portami wejścia/wyjścia.

Jeśli podczas przesyłania bloku procesor przyjmie przerwanie, licznik rozkazów będzie nadal wskazywał na instrukcję LDIR czy podobną i ta będzie mogła być kontynuowana bez przeszkód, jeśli tylko zawartość rejestrów BC, DE i HL oraz F zostanie odtworzona.

B. Rozkazy porównania z inkrementacją lub dekrementacją oraz rozkazy przeszukiwania bloku

W tej podgrupie mamy rozkazy: CPI (ang. *compare and increment*), CPD (ang. *compare and decrement*), CPIR (ang. *compare-increment-repeat*) i CPDR (ang. *compare-decrement-repeat*). Ich organizacja jest podobna do oganizacji rozkazów LDIR i LDDR i nie będziemy jej powtarzać.

Rozkazy CPI i CPD porównują zawartość akumulatora z bajtem pamięci adresowanym przez HL. Ich wykonanie składa się z następujących kroków:

1. Zawartość bajtu pamięci adresowanego przez HL jest pobierana i porównywana z akumulatorem, Jeśli są równe, to wskaźnik Z jest ustawiany, w przeciwnym razie zerowany. Odpowiedniej modyfikacji podlegają również wskaźniki S i H, podobnie jak w przypadku rozkazu CP.

2. Zawartość HL w rozkazie CPI jest zwiększana o 1; w rozkazie CPD zmniejszana o 1.

3. Zawartość BC jest zmniejszana o 1. Jeśli wynikiem jest zero, wskaźnik P/V jest zerowany, w przeciwnym razie ustawiany. Rozkazy CPIR i CPDR zawierają jeszcze następny krok:

4. Sprawdzane są wskaźniki Z oraz P/V. Jeśli szukany bajt został znaleziony, tj. wskaźnik zera jest ustawiony lub przeszukiwanie bloku zostało zakończone, tj. zawartość BC stała się równa zeru i wskaźnik P/V został wyzerowany, wykonywanie rozkazu ulega zakończeniu. W przeciwnym razie rozkaz się powtarza od kroku 1.

Rozkazy w tej podgrupie są prefiksowane przez ED.

Prefiks:	ED			
Mnemonika:	CPI	Kod:	10 100 001	(A1)
	CPD		10 101 001	(A9)
Długość rozkazu:	2			
Liczba cykli maszynowych:	4			
Liczba taktów:	16			

Zawartość rejestru F:

S	Z	H	P/V	N	C
↕	↕	↕	↕	1	*

Uwagi

Wykonanie rozkazu CPI nie jest równoważne wykonaniu rozkazu CP (HL) oraz odpowiednich modyfikacji zawartości rejestrów HL i BC. Wskaźniki są ustawiane inaczej, w szczególności wskaźnik C nie jest modyfikowany, gdy tymczasem rozkaz CP mógł go zmienić.

Prefiks: ED

Mnemonika: CPIR Kod: 10 110 001 (B1)

CPDR 10 111 001 (B9)

Długość rozkazu: 2

Liczba cykli maszynowych: 4 gdy zawartość BC = 0

5 w przeciwnym wypadku

Liczba taktów: 16 gdy zawartość BC = 0

21 w przeciwnym wypadku

Zawartość rejestru F:

S	Z	H	P/V	N	C
↕	↕	↕	↕	1	*

Uwagi

Po zakończeniu wykonywania rozkazów CPIR i CPDR program może sprawdzić, czy szukany bajt został znaleziony testując wskaźnik zera. Jeśli jest on wyzerowany oznacza to, że poszukiwanie zakończyło się niepowodzeniem. Wtedy nie potrzeba już nic innego sprawdzać. Natomiast w przypadku sukcesu, HL zawiera liczbę o 1 większą niż adres bajtu pamięci, który był równy zawartości akumulatora.

Oczywiście rozkazy CPIR i CPDR nie nadają się do testowania innych relacji między bajtami niż równość.

5.9. Grupa rozkazów sterujących

W tej niewielkiej grupie znajdują się rozkazy, które nie wpływają na dane dostępne programowi, lecz jedynie na wewnętrzny stan procesora. Nie jest zmieniana zawartość żadnego z rejestrów ogólnego przeznaczenia ani rejestru F. Oczywiście zawartość licznika rozkazów ulega normalnemu zwiększeniu.

Do tej grupy należą: rozkaz pusty NOP, rozkaz zatrzymania procesora HALT, rozkazy zablokowania i odblokowania przerwań maskowalnych DI (ang. *disable interrupt*) i EI (ang. *enable interrupt*) oraz rozkazy ustawiania trybu przyjęcia przerwań (ang. *interrupt modes*) IM 0, IM 1 oraz IM 2.

A. Rozkaz pusty i rozkaz zatrzymania

Mnemonika: NOP Kod: 00 000 000 (00)
HALT 01 110 110 (76)
Długość rozkazu: 1
Liczba cykli maszynowych: 1
Liczba taktów: 4

Uwagi
Rozkaz pusty może służyć do generowania opóźnień czasowych w programie, albo jako tymczasowy wypełniacz „dziur" w jeszcze niekompletnym, rozwijanym programie.

Rozkaz HALT jest stosowany głównie w sytuacji, gdy procesor czeka na przerwanie nie mając nic do roboty. Może to być korzystniejsze niż wprowadzenie procesora w martwą pętlę, gdyż dzięki wyprowadzeniu $\overline{HALT}$ układy zewnętrzne mają dostęp do informacji o stanie procesora.

Przypominamy, że stan procesora po zatrzymaniu jest dynamiczny: procesor wykonuje cykle równoważne rozkazom NOP generując normalne sygnały odświeżania w drugiej połowie cyklu maszynowego.

B. Rozkazy zablokowania i odblokowania przerwań maskowalnych

Rozkaz DI zeruje oba przerzutniki blokady przerwań IFF1 oraz IFF2. Rozkaz EI je ustawia. Charakterystyczną, ważną cechą rozkazu EI jest wprowadzenie krótkiego opóźnienia odblokowania przerwań. Ewentualny sygnał przerwania, który nadchodzi podczas wykonywania rozkazu EI nie jest testowany przez procesor po zakończeniu cyklu rozkazowego i procesor przed przyjęciem przerwania wykona jeszcze następny rozkaz. Dzięki temu, procesor zdąży jeszcze wykonać rozkaz powrotu z podprogramu obsługi przerwania, jeśli nastąpi to zaraz po wykonaniu EI. (Patrz rozkaz RETI w p. 5.12).

Mnemonika: DI Kod: 11 110 011 (F3)
EI 11 111 011 (FB)
Długość rozkazu: 1
Liczba cykli maszynowych: 1
Liczba taktów: 4

C. Rozkazy ustawiające tryb przyjęcia przerwania maskowalnego

Są to rozkazy prefiksowane przez ED. Różnica między tymi trybami została już omówiona w rozdz. 3 podczas opisu cyklu przyjęcia przerwania. Niezależnie od trybu, po przyjęciu przerwania procesor je maskuje zerując IFF1 i IFF2.

Tryb 0. W tym trybie procesor pobiera następny rozkaz nie z pamięci, lecz z portu wejściowego uaktywniając sygnał $\overline{IORQ}$ zamiast $\overline{MREQ}$. Należy zwrócić uwagę, że procesor nie dysponuje bezpośrednio informacją o naturze przerwania i na szynę adresów jest wysyłana zawartość PC, a nie adres portu (którego?). Ten problem musi zostać rozwiązany sprzętowo.

Tryb 1. Przyjęcie przerwania w tym trybie powoduje wywołanie podprogramu obsługi mieszczącego się pod ustalonym adresem 56.

Tryb 2. Po przyjęciu przerwania w tym trybie, procesor uaktywnia sygnał $\overline{IORQ}$ i odczytuje jeden bajt danych, który jest traktowany jako mniej znacząca połowa adresu wektora przerwań będącego tablicą w pamięci, gdzie znajduje się adres podprogramu obsługi przerwania. Jako bardziej znaczącą połowę procesor pobiera zawartość rejestru I. Ponieważ zawartość rejestru I jest ustawiana programowo rozkazem LD I,A, a ponadto adres podprogramu obsługi można zmienić, program może w zależności od kontekstu rozmaicie reagować na przerwania nawet przy identycznych stanach układów zewnętrznych. Przypominamy uwagę, że elementy wektora przerwań mają parzyste adresy.

Prefiks:	ED			
Mnemonika:	IM 0	Kod:	01 000 110	(46)
	IM 1		01 010 110	(56)
	IM 2		01 011 110	(5E)

Długość rozkazu:	2
Liczba cykli maszynowych:	2
Liczba taktów:	8

Uwagi

Trybem ustawianym w cyklu zerowania procesora jest tryb 0.

Niektóre asemblery stosują nieco różniącą się notację tych rozkazów: IM0, IM1, IM2.

5.10. Grupa rozkazów skoków

Skoki (łącznie z wywołaniami podprogramów, które również zawierają w sobie operacje skoku) stanowią jedyną strukturę sterowania na poziomie programowania w języku asemblera. Skoki w połączeniu z testowaniem warunków związanych ze stanem rejestru F są jedynym sposobem podejmowania decyzji przez program. Te znane i banalne fakty powodują jednak, że z punktu widzenia architektury programów, ich sprawności, a także czytelności, jest to bardzo ważna grupa rozkazów.

Procesor Z80 dysponuje trójbajtowymi rozkazami skoków (tzw. skoków długich) do dowolnego adresu pamięci — JP (ang. *jump*), rozkazami skoków

względnych — JR (ang. *jump relative*) oraz bardzo użytecznym w konstrukcji krótkich pętli rozkazem skoku względnego z automatyczną dekrementacją rejestru B — DJNZ (ang. *decrement and jump if not zero*). Skoki mogą być bezwarunkowe oraz warunkowe, testujące zawartość rejestru wskaźników stanu.

Żaden rozkaz skoku nie zmienia zawartości rejestru F.

A. Rozkazy skoków długich JP

Tę podgrupę nazwaliśmy *skokami długimi*, a nie bezwzględnymi w odróżnieniu od względnych, aby uniknąć nieporozumienia terminologicznego i pomylenia ze skokami bezwarunkowymi.

Będziemy sprawdzali zawartość 4 wskaźników stanu: S, Z, P/V oraz C. Dla każdego z tych wskaźników istnieją dwa rozkazy skoku warunkowego: jeśli wskaźnik jest ustawiony oraz jeśli jest wyzerowany. Jest więc testowanych 8 warunków. Skok jest wykonywany, jeśli warunek jest spełniony, w przeciwnym razie procesor przechodzi do następnego rozkazu. Oznaczenia mnemoniczne warunków i ich 3-bitowe kody przedstawia poniższy schemat:

Warunek *cc*	kod *ccc*	
NZ	000	(ang. *non zero* — wskaźnik Z wyzerowany)
Z	001	(ang. *zero* — wskaźnik Z ustawiony)
NC	010	(ang. *non carry* — wskaźnik C wyzerowany)
C	011	(ang. *carry* — wskaźnik C ustawiony)
PO	100	(ang. *parity odd* — wskaźnik P/V wyzerowany)
PE	101	(ang. *parity even* — wskaźnik P/V ustawiony)
P	110	(ang. *plus* — wskaźnik S wyzerowany)
M	111	(ang. *minus* — wskaźnik S ustawiony)

W omawianej podgrupie znajdują się trójbajtowe skoki zawierające adres jako dwubajtowy argument bezpośredni, a także rozkaz skoku do adresu zawartego w HL lub rejestrze indeksowym według znanego schematu prefiksowania. Te ostatnie mogą być tylko bezwarunkowe.

Mnemonika:	JP *nn*	Kod:	11 000 011	(C3)
	JP *cc, nn*		11 *ccc* 010	
Następne dwa bajty:			← *nl* →	
			← *nh* →	

Długość rozkazu: 3
Liczba cykli maszynowych: 3
Liczba taktów: 10

Przykłady

JP NC,23552	D2 00 5C
JP Z,65535	CA FF FF

(Jeśli ostatnia operacja arytmetyczna lub logiczna nie wygenerowała przeniesienia, skocz pod adres 23552; jeśli wynikiem było zero, skocz pod 65535.) □

Uwagi
Z logicznego punktu widzenia rozkaz skoku można traktować jako rozkaz wewnętrznego przesłania argumentu do licznika rozkazów. Zauważmy jednak, że musiałoby to trwać dłużej niż 10 taktów: nie można tego zrobić podczas pobierania argumentu, gdyż drugi bajt musi zostać poprawnie zaadresowany. Jednak ostatnie dwa cykle rozkazu skoku trwają tylko 3 takty i procesor nie ma czasu, aby wykonać przesłanie całego argumentu jeszcze podczas trwania rozkazu skoku. Oszczędność uzyskuje się dzięki dodatkowemu ukrytemu rejestrowi WZ, do którego procesor przesyła adres skoku podczas pobierania go z pamięci. Następnie, zawartość WZ, a nie PC zostaje użyta do pobrania kolejnego rozkazu. W tym czasie zawartość WZ jest zwiększona o 1, a wynik przesyłany do PC. (Patrz [12].)

Następne rozkazy są jedynym sposobem skoku pod adres obliczany dynamicznie w programie. Wykorzystują rejestry HL, IX lub IY.

Mnemonika: JP (HL) Kod: 11 101 001 (E9)
Długość rozkazu: 1 (2 dla IX lub IY)
Liczba cykli maszynowych: 1 (2)
Liczba taktów: 4 (8)

Przykład

JP (IY) FD E9

(Skocz pod adres zawarty w rejestrze IY.)

Uwagi
Rozkazy wykorzystujące IX lub IY nie mogą zawierać żadnego przesunięcia, trzeci bajt już należy do następnego rozkazu. To jest jeszcze jeden argument na rzecz tezy, że racjonalniej byłoby pisać: „JP HL" czy „JP IY".

B. Rozkazy skoków względnych

Poniższe rozkazy wykorzystują adresowanie względne; bazą jest zawartość licznika rozkazów. Umożliwiają skoki w zakresie od −128 do +127 bajtów od adresu następnego rozkazu. Podstawowymi korzyściami z ich stosowania są oszczędność pamięci (gdyż rozkazy te są dwubajtowe) oraz ułatwienie pisania programów przemieszczalnych.

Pewną wadą jest nieco dłuższy czas wykonywania w porównaniu ze skokami długimi, gdyż procesor musi wykonać odpowiednią operację dodawania adresu względnego do zawartości PC. Ponadto czas wykonywania tych rozkazów zależy od tego, czy badany warunek jest spełniony czy nie, co utrudnia ich wykorzystanie w programowaniu niektórych procesów czasu rzeczywistego, gdzie np. procesor generuje zadane z góry, ściśle określone opóźnienia czasowe wykonując odpowiednią liczbę przebiegów pętli.

Rozkazy warunkowych skoków względnych testują jedynie cztery warunki, w dodatku inaczej kodowane niż dla skoków długich. Dlatego wszystkie rozkazy skoków warunkowych zostaną wypisane jawnie.

Mnemonika:	JR *n*	Kod:	00 011 000	(18)
	JR NZ,*n*		00 100 000	(20)
	JR Z,*nn*		00 101 000	(28)
	JR NC,*n*		00 110 000	(30)
	JR C,*n*		00 111 000	(38)

Drugi bajt rozkazu: ← *n* →
Długość rozkazu: 2
Liczba cykli maszynowych: 3 jeśli warunek spełniony
2 jeśli warunek nie spełniony
Liczba taktów: 12 jeśli warunek spełniony
7 jeśli warunek nie spełniony

Przykład

JR C,−6 38 FA

(Jeśli wskaźnik przeniesienia jest ustawiony, skocz pod adres o 4 mniejszy niż adres rozkazu JR.) □

Uwagi

Czasami w opisie skoków względnych w literaturze spotyka się skrótowe, symboliczne przedstawienie wyniku tej operacji:

$$PC \leftarrow PC + n$$

Aby ten zapis był zgodny z odpowiednim zapisem dla np. dowolnego jednobajtowego rozkazu przesłania zawartości rejestrów:

$r1 \leftarrow r2$ i następnie $PC \leftarrow PC + 1$

jako argument operacji dodawania należałoby uważać zawartość licznika rozkazów w momencie pobierania rozkazu, a nie jego wykonywania. Zdaniem autora tę konwencję należy stosować z dużą ostrożnością, gdyż po pierwsze nie odpowiada ona rzeczywistemu procesowi, a po drugie w przypadku skoków względnych wprowadza pewne zamieszanie: *n* trzeba traktować jako liczbę zawartą między −126 a +129, co nie przystaje do interpretacji bajtu przesunięcia jako liczby w dwójkowym kodzie uzupełnieniowym.

C. Rozkaz skoku względnego z dekrementacją

Dość typową konstrukcją występującą w wielu programach jest *pętla* przebiegana z góry zadaną liczbę razy; liczba przebiegów jest umieszczana w rejestrze, który za każdym przebiegiem jest dekrementowany, i gdy osiągnie zero pętla jest przerywana.

Z80 dysponuje bardzo dogodnym rozkazem DJNZ, którego wykonanie składa się z dwóch kroków:

1. Zawartość rejestru B jest zmniejszana o 1.
2. Jeśli wynik jest równy zeru, program przechodzi do następnego rozkazu, w przeciwnym razie wykonuje się skok względny. Należy zaznaczyć, że dekrementacja B w tym przypadku nie modyfikuje zawartości rejestru F.

Mnemonika:	DJNZ *n*	Kod:	00 010 000 (10)
Drugi bajt rozkazu:			← *n* →
Długość rozkazu:	2		
Liczba cykli maszynowych:	3	jeśli B < > 0	
	2	jeśli B = 0	
Liczba taktów:	13	jeśli B < > 0	
	8	jeśli B = 0	
Zawartość rejestru F:		nie ulega zmianie	

Przykład

DJNZ −2 10 FE

(Jest to jednorozkazowa pętla: wykonuj ten rozkaz tak długo, aż wartość B stanie się równa zeru.) □

5.11. Grupa rozkazów wywołań podprogramów i rozkazów powrotu

Z80 podobnie jak większość innych mikroprocesorów dysponuje bardzo prostym, ale efektywnym i wygodnym mechanizmem wywołań traktowanych jako *skoki do podprogramu ze śladem*, tj. z zapamiętaniem informacji pozwalającej podprogramowi wrócić, czyli wykonać skok do miejsca w programie mieszczącego następny rozkaz po wywołaniu.

Informacja ta, czyli po prostu zawartość licznika rozkazów w momencie wykonania rozkazu, jest automatycznie umieszczana na stosie. Procesor wykonuje (nieistniejący na poziomie języka asemblera) rozkaz „PUSH PC”, a następnie zwykły skok (długi).

Omawiana grupa zawiera rozkazy CALL (warunkowe i bezwarunkowe) oraz tzw. *rozkazy restartu* RST — będące specjalnymi, krótkimi rozkazami wywołań podprogramów mieszczących się na stronicy zerowej pamięci (o czym już była mowa wcześniej).

Rozkaz powrotu można wyobrazić sobie jako wykonanie (nieistniejącego) rozkazu „POP PC”, co powoduje przekazanie sterowania właściwemu fragmentowi programu. W tej grupie znajdują się warunkowe i bezwarunkowe

rozkazy powrotu RET, a także specjalne rozkazy powrotu z podprogramów obsługi przerwań: RETI i RETN. Żaden rozkaz z tej grupy nie modyfikuje zawartości rejestru F.

A. Rozkazy wywołań długich

Termin „długi" należy rozumieć tak jak dla skoków: argumentem bezpośrednim rozkazu jest adres, tj. liczba dwubajtowa. Symbol *cc* w mnemonice i trójka bitów *ccc* w kodzie mają znaczenie identyczne jak w przypadku skoków.

Mnemonika:	CALL *nn*	Kod:	11 001 101	(CD)
	CALL *cc,nn*		11 *ccc* 100	
Następne dwa bajty rozkazu:			← *nl* →	
			← *nh* →	

Długość rozkazu:	3	
Liczba cykli maszynowych:	5	jeśli warunek spełniony
	3	w przeciwnym przypadku
Liczba taktów:	17	jeśli warunek spełniony
	10	w przeciwnym przypadku

Przykład

CALL NZ,32767 C4 FF 7F

(Jeśli wartość wskaźnika Z jest różna od zera wywołaj podprogram o początkowym adresie 32767.) □

Uwagi

Nie ma rozkazów pozwalających wywołać podprogram o dynamicznie obliczanym adresie. Należy to zorganizować programowo. Zostanie to omówione w rozdziale 6.

B. Rozkazy restartu

Są to krótkie, jednobajtowe rozkazy używane dla oszczędności pamięci (a również i czasu) w stosunku do wywołań długich w przypadku, jeśli program zawiera wiele wywołań jednej lub kilku krytycznych procedur. Szczególnym wykorzystaniem jest obsługa przerwań: rozsądnym wykorzystaniem trybu 0 przez urządzenie zewnętrzne jest przesłanie procesorowi rozkazu RST ze względu na jego niewielką długość. Mnemonika rozkazów RST zawiera jako argument adres podprogramu, który jest liczbą jednobajtową, jednak kodowanie nie jest bezpośrednie i opiera się na następującym schemacie:

Adres *n*		kod *ttt*
#00	(0)	000
#08	(8)	001
#10	(16)	010
#18	(24)	011
#20	(32)	100
#28	(40)	101
#30	(48)	110
#38	(56)	111

czyli: adres $n = 8 * ttt$.

Mnemonika: RST *n* Kod: 11 *ttt* 111
Długość rozkazu: 1
Liczba cykli maszynowych: 3
Liczba taktów: 11

Przykład

RST #20 E7 □

C. Rozkazy powrotu

Podobnie jak dla rozkazów skoków długich i rozkazów CALL mamy do dyspozycji rozkaz powrotu bezwarunkowego i rozkazy warunkowe testujące takie same warunki jak rozkazy JP i CALL, i tak samo kodowane.

Mnemonika: RET Kod: 11 001 001 (C9)
RET *cc* 11 *ccc* 000
Długość rozkazu: 1
Liczba cykli maszynowych: 3 jeśli warunek spełniony
1 w przeciwnym przypadku
Liczba taktów: 11 jeśli warunek spełniony
5 w przeciwnym przypadku

Przykład

RET C D8 □

Uwagi
Umieszczanie adresów powrotu na stosie przez rozkaz CALL i automatyczne ich zdejmowanie przez RET umożliwia łatwą organizację *wywołań rekursywnych.* Używanie tego samego stosu adresowanego poprzez SP zarówno do wywołań/powrotów, jak i do przechowywania danych może być zarówno błogosławieństwem, jak i przekleństwem programisty. Z jednej strony operując stosem program może się dowiedzieć skąd był wywoływany, można łatwo zorganizować wywoływanie podprogramów o obliczanych dynamicznie adresach i w przypad-

ku rekursji bezpiecznie przechowywać parametry i zmienne lokalne. Z drugiej strony jakiekolwiek błędy w obsłudze stosu nie tylko zaburzają dane przetwarzane przez program, ale mogą zdezorganizować przepływ sterowania.

D. Rozkazy powrotu z podprogramów obsługi przerwań

Ta podgrupa zawiera rozkaz RETI przydatny do zakończenia podprogramu obsługi przerwania maskowalnego oraz rozkaz RETN użyteczny w podprogramach obsługi przerwania niemaskowalnego.

Z punktu widzenia programu rozkaz RETI nie różni się niczym od zwykłego rozkazu RET z wyjątkiem tego, że jest dłuższy i trwa dłużej, co trudno nazwać zaletą. Jest to jednak rozkaz, który jest sprzętowo rozpoznawany przez niektóre układy specjalnie zaprojektowane do współpracy z Z80, oczywiście jeśli są one podłączone do wspólnej magistrali danych z procesorem oraz do niektórych sygnałów sterujących, w szczególności sygnału $\overline{M1}$. Na przykład równoległy port wejścia/wyjścia Zilog PIO (znany również jako Mostek MK 3883) można przyłączyć i zaprogramować tak, aby generował przerwania maskowalne. Pobranie przez procesor rozkazu RETI zostanie potraktowane przez PIO jako sygnał zakończenia podprogramu obsługi przerwania i PIO podniesie poziom swojego wyjścia INT bez ingerencji procesora.

Jak już zostało opisane, przyjęcie przez procesor przerwania $\overline{NMI}$ powoduje zablokowanie przerwań maskowalnych przez wyzerowanie rejestru IFF1, ale wcześniej zapamiętanie jego zawartości w rejestrze IFF2. Wykonanie rozkazu RETN spowoduje przepisanie zawartości IFF2 z powrotem do IFF1 odtwarzając poprzedni stan gotowości procesora do przyjęcia sygnału $\overline{INT}$.

Rozkazy omawianej podgrupy są prefiksowane przez ED.

Prefiks:	ED			
Mnemonika:	RETI	Kod:	01 001 101	(4D)
	RETN		01 000 101	(45)

Długość rozkazu: 2
Liczba cykli maszynowych: 4
Liczba taktów: 14

Uwagi

Zgodnie z komentarzem dotyczącym rozkazu EI, jeśli ostatnimi rozkazami podprogramu obsługi przerwania maskowalnego będzie sekwencja:

```
EI
RETI        ; (względnie RET)
```

ewentualne przerwanie zostanie przyjęte dopiero po wykonaniu rozkazu powrotu, gdy stos znajdzie się w takim samym stanie jak przed przyjęciem przerwania.

5.12. Grupa rozkazów adresujących bity

Jest to koncepcyjnie bardzo prosta i bardzo użyteczna w niektórych zastosowaniach grupa zawierająca rozkazy BIT testujące pojedynczy bit rejestru lub komórki pamięci, rozkazy SET ustawiające i RES zerujące (ang. *reset*) pojedynczy bit.

Wszystkie rozkazy tej grupy są prefiksowane przez CB. Rozkazy te mogą adresować dowolny rejestr 8-bitowy ogólnego przeznaczenia oraz (HL), (IX + *d*) i (IY + *d*), a więc możliwe są jeszcze prefiksy DD i FD. Podobnie jednak jak dla rozkazów przesunięć, użycie połówek rejestrów indeksowych nie jest możliwe. Każdy rozkaz prefiksowany przez DD będzie operował na bajcie pamięci adresowanym przez (IX + *d*) i przesunięcie *d* musi być zawarte w rozkazie jako trzeci bajt, natomiast w przypadku rozkazów SET i RES wynik zostanie przesłany i do pamięci, i do odpowiedniego rejestru. Kodowanie rejestru oznaczane bitami *sss* jest takie samo jak poprzednio. Numer *b* testowanego lub modyfikowanego bitu w rejestrze będzie oznaczany przez 3 bity *bbb* w normalnym kodzie dwójkowym, tj. *bbb* równe 000 oznacza bit 0, 111 — bit 7.

A. Rozkazy testujące

Ponieważ bit może być albo 0 albo 1, jedynym wskaźnikiem racjonalnie modyfikowanym przez te rozkazy jest wskaźnik Z.

Prefiks: CB
Mnemonika: BIT *b*, *r* Kod: 01 *bbb rrr*
Długość rozkazu: 2 (4 dla indeksowanego)
Liczba cykli maszynowych: 2 (3 dla (HL), 5 dla IX lub IY)
Liczba taktów: 8 (12 dla (HL), 20 dla IX lub IY)
Zawartość rejestru F:

S	Z	H	P/V	N	C
?	↕	1	?	0	*

Przykład

BIT 4,H CB 64
BIT 0,(IY + 127) FD CB 7F 46 □

B. Rozkazy zerujące i ustawiające bit

Prefiks: CB
Mnemonika: RES *b*,*r* Kod: 10 *bbb rrr*
SET *b*,*r* 11 *bbb rrr*

Długość rozkazu: 2 (4 dla indeksowanego)
Liczba cykli maszynowych: 2 (4 dla (HL), 6 dla indeks.)
Liczba taktów: 8 (15 dla (HL), 23 dla indeks.)
Zawartość rejestru F: nie ulega zmianie

Przykład

```
SET 7,A            CB FF
RES 0,(IX)         DD CB 00 86
```

(Ustaw najbardziej znaczący bit akumulatora; wyzeruj najmniej znaczący bit komórki pamięci adresowanej przez IX.) □

Uwagi

Nieoficjalne rozszerzenia rozkazów indeksowanych można zapisać jako np.

```
SET 2,(IX),D       DD CB 00 D2
```

jednak nie jest to rozpoznawane przez typowe asemblery. Te rozszerzenia należy zresztą uznać za przypadkową cechę sprzętową (lub mikroprogramu) Z80. Powtarzamy, że dane podane w niniejszej książce są oparte na nieoficjalnych informacjach z innych źródeł i testach wykonanych przez autora. Nie powinny one służyć jako podstawa konstrukcji użytkowego oprogramowania bez dokładnego niezależnego sprawdzenia.

5.13. Grupa rozkazów wejścia i wyjścia

Ostatnią omawianą w tym rozdziale grupą są rozkazy wejścia i wyjścia. W odróżnieniu od niektórych mikroprocesorów, które w ogóle nie dysponują specjalnymi rozkazami adresującymi urządzenia zewnętrzne i pewna liczba adresów dekodowana sprzętowo służy tym celom, Z80 posiada całkowicie niezależną przestrzeń adresową dla rozkazów wejścia i wyjścia. Z punktu widzenia programu te rozkazy są po prostu przesłaniami z lub do procesora, a urządzenia zewnętrzne odróżniają je od rozkazów dotyczących pamięci dzięki sygnałowi $\overline{IORQ}$. Zachowanie się procesora podczas wykonywania cyklu czytania lub pisania z portu wejścia/wyjścia zostało przedstawione na rys. 3.3.

Do tej grupy należą rozkazy wejścia IN i wyjścia OUT, a także rozkazy transmisji z automatyczną modyfikacją rejestrów i rozkazy transmisji blokowej: INI (ang. *input and increment*), INIR (ang. *input-increment-repeat* itp. dla pozostałych nazw mnemonicznych), IND, INDR, OUTI, OTIR, OUTD i OTDR realizowane podobnie jak LDI itd.

Rozkazy IN i OUT można podzielić na dwie podgrupy. Do pierwszej będą należeć rozkazy zawierające jednobajtowy argument bezpośredni będący adresem portu i mogące adresować jeden z 256 portów zewnętrznych (naprawdę więcej, ale w bardzo specyficzny, wymagający dodatkowego omówienia sposób).

W drugiej podgrupie znajdują się rozkazy wykorzystujące pełną przestrzeń adresową procesora. Adres portu jest umieszczany w rejestrze BC. Również rozkazy transmisji blokowych wykorzystują tę opcję, jednak należy zdać sobie sprawę z tego, że podczas wykonywania tych rozkazów zawartość rejestru B ulega zmianie.

A. Rozkazy adresujące jawnie port zewnętrzny

W tej podgrupie rejestrem zawierającym wysyłaną lub odbieraną informację jest zawsze akumulator.

Jak wspomnieliśmy, te rozkazy zawierają jako jednobajtowy argument adres portu. Jest on przesyłany na mniej znaczącą połowę szyny adresów. W tym czasie procesor przesyła na bardziej znaczącą połowę szyny adresów zawartość akumulatora. W przypadku rozkazów wejścia, w których zawartość akumulatora zostanie przepisana wprowadzoną daną, ten mechanizm może służyć do adresowania 65536 portów, jednak w przypadku rozkazów wyjścia akumulator musi zawierać daną a nie dowolny adres, więc w tym sensie pożytek z tego ustalenia jest niewielki, choć bardzo wyrafinowane układy zewnętrzne i tu mogą znaleźć jakieś zastosowania.

Mnemonika: IN A, (*n*) Kod: 11 011 011 (DB)
OUT (*n*), A 11 010 011 (D3)
Drugi bajt rozkazu: ← *n* →
Długość rozkazu: 2
Liczba cykli maszynowych: 3
Liczba taktów: 11
Zawartość rejestru F: nie ulega zmianie

Przykład

IN A,(254) DB FE

(Przeczytaj jeden bajt z portu o adresie # FE i umieść zawartość w akumulatorze.) □

B. Rozkazy adresujące port zawartością BC

W tej podgrupie zawartość rejestru C jest przesyłana na mniej znaczącą połowę szyny adresów, tj. na linie $A_0...A_7$, a zawartość B — na $A_8...A_{15}$. Rejestrem zawierającym wysyłaną lub odbieraną daną może być dowolny rejestr 8-bitowy ogólnego przeznaczenia zgodnie ze standardowym kodowaniem, jednak wyłączone z tego są HX itp., a także (HL). Kod 110 ma sens tylko dla rozkazu IN i jest traktowany nietypowo: otrzymana dana nie jest nigdzie przesyłana, jedynie rejestr F ulega modyfikacji, podobnie jak dla innych rozkazów tej podgrupy. Mnemonika rozkazu ma wtedy postać: IN F, (C).

W większości prostych układów opartych na procesorze Z80 porty zewnętrzne są podłączone jedynie do mniej znaczącej połowy szyny adresów i zawartość B jest wtedy ignorowana.

Rozkazy tej podgrupy są prefiksowane przez ED i nie dopuszczają prefiksów DD czy FD.

Prefiks: ED
Mnemonika: IN *r*,(C) Kod: 01 *sss* 000
OUT (C),*r* 01 *ddd* 001
Długość rozkazu: 2
Liczba cykli maszynowych: 3
Liczba taktów: 12
Zawartość rejestru F:

	S	Z	H	P/V	N	C
Dla rozkazów IN:	↕	↕	0	P	0	*
Dla rozkazów OUT:	*	*	*	*	*	*

Przykłady

IN A,(C) ED 78
OUT (C),C ED 49

(Prześlij do akumulatora bajt danych z portu adresowanego rejestrem BC; wyślij do portu adresowanego przez BC zawartość rejestru C.) □

C. Rozkazy transmisji z inkrementacją/dekrementacją rejestru i rozkazy transmisji blokowej

Wykonanie rozkazu INI lub IND składa się z następujących kroków:

1. Przesłanie bajtu z portu adresowanego przez BC do komórki pamięci adresowanej przez HL.

2. Zawartość HL ulega zwiększeniu o 1 dla rozkazu INI, zmniejszeniu o 1 dla rozkazu IND.

3. Zawartość B ulega zmniejszeniu o 1. (Zawartość C pozostaje bez zmian!)

Wykonanie rozkazu OUTI oraz OUTD różni się pierwszym krokiem; transmisja następuje z pamięci do portu wyjścia.

Rozkazy transmisji blokowej zawierają dodatkowy krok:

4. Testuj B: jeśli zawartość B jest równa zeru — przerwij; w przeciwnym razie powtórz kroki 1, 2, 3 i 4.

Jak widać możliwa jest transmisja od 1 do 256 bajtów.

Wszystkie rozkazy tej podgrupy są prefiksowane przez ED.

Prefiks:	ED			
Mnemonika:	INI	Kod:	10100010	(A2)
	IND		10101010	(AA)
	OUTI		10100011	(A3)
	OUTD		10101011	(AB)

Długość rozkazu: 2
Liczba cykli maszynowych: 4
Liczba taktów: 16
Zawartość rejestru F:

S	Z	H	P/V	N	C
?	↕	?	?	1	*

Uwagi

Wskaźnik Z jest ustawiany, jeśli po wykonaniu rozkazu rejestr B został wyzerowany, w przeciwnym razie zawartość Z jest zerem.

Prefiks:	ED			
Mnemonika:	INIR	Kod:	10110010	(B2)
	INDR		10111010	(BA)
	OTIR		10110011	(B3)
	OTDR		10111011	(B8)

Długość rozkazu: 2
Liczba cykli maszynowych: 5 jeśli B < > 0
4 jeśli B = 0
Liczba taktów: 21 jeśli B < > 0
16 jeśli B = 0
Zawartość rejestru F:

S	Z	H	P/V	N	C
?	1	?	?	1	*

Uwagi

Wykorzystanie bardziej znaczącej połowy szyny adresów w tych rozkazach nie jest typowe, ale jest możliwe. Można w ten sposób informować układy zewnętrzne, ile pozostało bajtów do końca transmisji. Można także jednym rozkazem obsłużyć wiele portów wejścia, np. w niektórych mikrokomputerach klawiatura jest traktowana jako składająca się z kilku sekcji matryca wyłączników podłączona bezpośrednio do magistral systemowych. Jej stan można zbadać jednym lub kilkoma rozkazami INIR, co może być prostsze i tańsze w realizacji sprzętowej niż klawiatura z pełnym dekodowaniem znaków.

Zwróćmy uwagę na zawartość wskaźnika Z równą 1. Jedynka oznacza, że transmisja uległa zakończeniu i rejestr B został wyzerowany. Przerwanie transmisji przed jej zakończeniem zostawia oczywiście wyzerowany wskaźnik Z.

Część druga

Techniki programowania w języku asemblera

6. Architektura programów w języku asemblera

6.1. Od algorytmu do programu

Celem tej książki nie może być — choćby ze względu na jej objętość — wyczerpujące przedstawienie zasad programowania na poziomie bliskim sprzętowi, tj. w języku wewnętrznym. Literatura na ten temat jest dość skąpa, ale istnieje. Zainteresowanego Czytelnika odsyłamy do książki Grabowskiego i Koślacza [5] zawierającej wiele ogólnych dobrych rad, a także sporo nieźle udokumentowanych przykładów w języku asemblera procesora Intel 8080, oraz do książki [7], w której można znaleźć porównanie procesorów Z80 i 8080 na kilku prostych przykładach.

Rozpoczynający programowanie w języku asemblera powinien od razu zdać sobie sprawę z tego, że opanowanie listy instrukcji procesora ma jeszcze mniejszy udział w efektywnym programowaniu niż opanowanie pełnej składni języka programowania wyższego poziomu. Droga od algorytmu do programu jest bardzo długa; ze względu na obfitość wariantów, programowanie w języku asemblera wymaga od programisty znacznie większej dyscypliny niż programowanie np. w Pascalu i angażuje go w zajmowanie się szczegółami nieistotnymi z punktu widzenia rozwiązywanego problemu. Mając do dyspozycji dość rozbudowaną listę rozkazów procesora Z80 (charakteryzującą się nadmiarowością i brakiem regularności), wybór jednego z kilku wariantów wydaje się kwestią stylu. Na przykład sprawdzenie, czy bit nr 1 w akumulatorze jest ustawiony, można zrealizować przez użycie rozkazu

```
BIT 1,A
```

ale również przez użycie rozkazu

```
        AND 02
```

jeśli poprzednią zawartość akumulatora można zniszczyć. Długość tych rozkazów jest taka sama, różnica w czasach wykonania prawie żadna (drugi jest krótszy o 1 takt zegarowy). Pierwszy jest czytelniejszy. A może okaże się, że jeszcze inny wariant jest lepszy, np.

```
        RRA
        RRA
```

ze sprawdzeniem wskaźnika przeniesienia zamiast wskaźnika Z.

Bardzo często pogoń za oszczędnością kilku bajtów lub kilku mikrosekund kosztuje wiele niepotrzebnych godzin pracy programisty. Często jednak również drobna nieefektywność wynikająca z nieuwagi, niekompetencji czy nonszalancji programisty powiela się w wielu tysiącach rozprowadzonych systemów mikrokomputerowych!

Twórcy oprogramowania systemowego już od pewnego czasu odchodzą od programowania w języku asemblera. Prawie cały system operacyjny UNIX i nowsze wersje systemu CP/M są pisane w języku „C". Jednakże tam, gdzie każda mikrosekunda lub każdy bajt jest na wagę złota, gdy trzeba w określonym polu pamięci zmieścić bardzo sprawny program — asembler nadal pozostaje niezastąpiony, gdyż kompilator, który brałby pod uwagę wszystkie możliwości optymalizacji musiałby być olbrzymi.

Jeśli autor tej książki może podzielić się swoim osobistym doświadczeniem — to przed przystąpieniem do kodowania dużego programu w języku asemblera bardzo dobrym zwyczajem jest napisanie go w jakimś znanym języku wyższego poziomu, posiadającym dobre cechy strukturalne i rozsądny poziom autodokumentacyjności, np. w Pascalu czy „C". Jest to znacznie lepsze rozwiązanie niż konstrukcja diagramów blokowych zarówno ze względu na zwartość i czytelność, jak i na możliwość niezależnej weryfikacji algorytmu. Znajomość języków wyższego poziomu i tak jest pewną koniecznością — jest to najczęściej stosowany w światowej literaturze informatycznej sposób prezentacji algorytmów i nie do pomyślenia jest efektywne programowanie przy znajomości jedynie języka asemblera. Pewną wadą (a może jednak błogosławieństwem?) tego podejścia jest konceptualne utrudnienie programiście stosowania pewnych chwytów optymalizacyjnych niskiego poziomu, jak np. nietypowe przekazywanie sterowania, czy pisanie samomodyfikujących się programów. Będziemy starali się ich unikać, choć nie uważamy, że podobne techniki zasługują jedynie na dyskwalifikację.

Dalszy ciąg tego rozdziału jest poświęcony podstawowym strukturom sterowania: podprogramom, pętlom i rozgałęzieniom, a także strukturom sterowania wyższego poziomu, które mają pewien związek z technikami interpretacji. Interpretatory i programy interpretowane wbrew pozorom nie

należą do wyszukanych i wyspecjalizowanych technik programowania, lecz pozwalają konstruować oprogramowanie znacznie bardziej zwarte, choć wolniejsze niż w kodzie maszynowym, a to w technice mikroprocesorowej ma olbrzymie znaczenie.

6.2. Podprogramy i wykorzystanie stosu

Podprogramami lub *procedurami* nazywamy fragmenty (sekcje) programu przekazujące sobie sterowanie za pomocą rozkazów wywołań i powrotów. Podział programu na podprogramy jest podstawową techniką *modularyzacji programu* — rozbicia go na dające się objąć w całości, autonomiczne fragmenty. Można śmiało stwierdzić, że techniki składania programu z procedur i zapewnienie prawidłowej komunikacji między procedurami należą do kanonów dobrego programowania każdego programisty, niezależnie od komputera i języka programowania.

W naszym przypadku jest to zagadnienie szczególnie ważne z dwóch względów:

1. Krótkie, 8-bajtowe dane, na których procesor wykonuje bezpośrednie operacje, powodują to, że przetwarzanie każdej większej struktury danych, np. wielobajtowej liczby całkowitej lub zmiennopozycyjnej, wymaga dość długiej sekwencji operacji. Te same sekwencje wystąpią wielokrotnie podczas obliczania bardziej skomplikowanego wyrażenia. W związku z tym rozkazy wywołań odpowiednich procedur będą się pojawiały bardzo często, o wiele częściej niż w przypadku procesorów dysponujących dłuższym słowem maszynowym.

2. Język asemblera nie daje żadnych ułatwień w organizacji przekazywania argumentów procedurom ani dostarczania na zewnątrz wyników. Jest to jeden z głównych powodów ogólnego odwrotu od programowania w języku asemblera i stworzenie języków takich jak PL/M czy „C", które z jednej strony są językami „niskiego poziomu", pozwalającymi w miarę efektywnie posługiwać się strukturami danych charakterystycznymi dla danej architektury komputera: bajty, adresy, bity itp., a z drugiej strony dającymi programiście mocne i wygodne w konstrukcji struktury sterowania: pętle, wyrażenia warunkowe, procedury z parametrami itp.

Niniejszy rozdział jest poświęcony konstrukcji podprogramów w języku asemblera ze zwróceniem uwagi na sposoby przakazywania parametrów oraz możliwą optymalizację.

Jako dość prosty i przejrzysty, ale niebanalny przykład procedury weźmy podprogram obliczający wartość współczynnika dwumianu Newtona $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. Aby uniknąć komplikacji związanej z mnożeniem i dzieleniem

(i to w odpowiedniej kolejności, aby uniknąć przepełnienia) skorzystamy z rekurencyjnej definicji współczynnika Newtona: $\binom{n}{0}=\binom{n}{n}=1$ oraz $\binom{n}{k}= =\binom{n-1}{k}+\binom{n-1}{k-1}$. Trzeba z góry zaznaczyć, że ta procedura nadaje się do praktycznych zastosowań tylko dla niewielkich n i k (rzędu kilku). Jest wtedy bardzo szybka. Dla większych n i k jest bardzo nieefektywna czasowo, gdyż ma złożoność rosnącą wykładniczo z n. Jednak łatwo można ją ulepszyć. Zostawimy to jednak jako samodzielne ćwiczenie.

```
; Procedura BIN(n,k) obliczania wartości symbolu
; Newtona 'n po k'
; Parametry:
; Rejestr B: n,    rejestr C: k
; Wynik: w rejestrze HL.
BIN     LD HL,0001      ; początkowa wartość wyniku
        LD A,C          ; wartość k
        OR A
        RET Z           ; powrót jeśli k=0
        CP B
        RET Z           ; powrót jeśli k=n
        DEC B           ; n <- n-1
        PUSH BC         ; zapamiętanie na stosie n-1 i k
        CALL BIN        ; wywołanie BIN(n-1,k)
        POP BC          ; odzyskanie parametrów
        DEC C           ; k <- k-1
        PUSH HL         ; zapamiętanie wartości BIN(n-1,k)
        CALL BIN        ; wywołanie BIN(n-1,k-1)
        POP DE          ; odzyskanie wartości BIN(n-1,k)
        ADD HL,DE       ; suma BIN(n-1,k)+BIN(n-1,k-1)
        RET
```

Uważny Czytelnik dostrzeże, że dla wywołań z n i k różnymi od zera i od siebie umieszczanie wartości 1 w HL jest niepotrzebne. Dlaczego w miejscu rozkazu RET Z nie umieścić warunkowego skoku do sekcji procedury, która dopiero gdy trzeba zwraca wartość 1 w HL? Jak wynika z praktyki nie jest to jednak opłacalne ani pamięciowo, ani czasowo; strata czasu na wykonanie skoków jest zbyt wielka.

W programowaniu w języku asemblera bardzo często okazuje się, że podejmowanie decyzji i wykonywanie skoków warunkowych, aby ominąć wykonanie pojedynczego rozkazu lub ich krótkiej sekwencji, może zająć więcej miejsca i czasu niż wykonanie tych rozkazów przed sprawdzeniem warunku. Ponadto zwiększanie liczby rozkazów skoku w programie ujemnie wpływa na

jego czytelność. Oznacza to, że nieraz bardziej opłacalne będzie niepotrzebne wykonanie niektórych rozkazów niż ich ominięcie. Aby wykorzystywać efektywnie tę technikę programista musi jednak pokonać pewien psychiczny opór.

W powyższym przykładzie mieliśmy do czynienia z dwoma parametrami, które łatwo było przekazać w rejestrach. W przypadku większej liczby parametrów ich wartości należy umieścić w pamięci. Można je zgromadzić w tablicy, której adres jest stały i znany procedurze, jednak jest to bardzo sztywne rozwiązanie. Można również adres tej tablicy przekazać w rejestrze — często dobrym rozwiązaniem jest wykorzystanie rejestru indeksowego, gdyż wtedy procedura ma ułatwione adresowanie wielu elementów tablicy bez dodatkowych obliczeń.

W większych programach na ogół występuje pewna grupa procedur, które są bardzo często wywoływane. Przykładem może być procedura drukująca pojedynczy znak na urządzeniu zewnętrznym albo procedura diagnozująca błędy wykryte przez program. Wtedy samo przygotowanie parametrów przekazywanych podprogramowi i umieszczenie ich wartości lub adresów w rejestrach może być dość kosztowne. Jeśli parametry są stałymi, tj. nie wymagają obliczeń, można je umieścić w programie bezpośrednio po rozkazie wywołania i przekazać procedurze za pomocą następującej techniki:

```
        CALL PROC
PAR     DEFB n1,n2,...      ; dane przekazane procedurze PROC
        ...                 ; dalszy ciąg programu

PROC    POP HL              ; adres tablicy PAR
        ...
```

Zdejmując ze stosu adres powrotu, procedura w rzeczywistości umieszcza w rejestrze HL adres tablicy parametrów. Teraz jednak powstaje problem powrotu do wykonania dalszego ciągu programu. W tym celu procedura musi dysponować informacją o długości tablicy parametrów: może to być np. jeden z parametrów lub tablica może mieć ustaloną długość — często po prostu jest to jeden bajt: kod drukowanego znaku lub numer błędu. Jeśli długość tablicy zostanie już obliczona, a następnie umieszczona w rejestrze DE, a rejestr HL nadal zawiera adres tablicy, to powrotu dokonuje się za pomocą następującej sekwencji rozkazów:

```
        ADD HL,DE
        JP (HL)
```

Przedstawiona technika może się wydawać „nieczystym” trikiem programowania, ale obrazuje ona bardzo charakterystyczne dla programowania na poziomie asemblera silne powiązanie struktur danych i struktur sterowania: używanie adresów fragmentów programu jako danych i odwrotnie.

Innym wykorzystaniem tej możliwości jest wywołanie procedury o dynamicznie obliczanym adresie, umieszczanym np. w rejestrze HL. Niestety rozkaz „CALL (HL)" analogiczny do JP (HL) nie istnieje. Adres powrotu można jednak umieścić na stosie jawnie

```
        LD DE,RETADD
        PUSH DE
        JP (HL)
RETADD  . . .           ; dalszy ciąg programu
```

W przypadku, gdy nie można poświęcić DE ani BC, mamy rozwiązanie alternatywne

```
        PUSH HL         ; adres procedury
        LD HL,RETADD
        EX (SP),HL      ; wymieniony na adres powrotu
        JP (HL)
RETADD  . . .           ; dalszy ciąg programu
```

Najekonomiczniejszym i najprostszym rozwiązaniem jest napisanie i użycie specjalnego podprogramu, który działa jak „stacja przesiadkowa"

```
        CALL CALLHL
        . . .           ; dalszy ciąg programu
CALLHL  JP (HL)
```

Procedura CALLHL przekaże sterowanie właściwemu podprogramowi z prawidłowym adresem powrotu na stosie. Z tej techniki jeszcze kilkakrotnie skorzystamy.

Z kolei, jeśli adres wywoływanej procedury znajduje się w rejestrze DE a nie HL, nieistniejący rozkaz skoku „JP (DE)" można zrealizować za pomocą sekwencji rozkazów

```
        PUSH DE
        RET
```

Pominęliśmy omówienie *rozkazów restartu* RST. Ich zastosowanie jest dość oczywiste: są one po prostu krótkimi rozkazami wywołań podprogramów i opłaca się ich używać ze względu na oszczędność pamięci. Nie kryją one w sobie żadnych możliwości ulepszeń. Należy pamiętać, że obszar stronicy zerowej jest niewielki i dłuższe podprogramy wywoływane przez RST i tak będą musiały być przedłużane skokami do innych pól pamięci. Ponadto w wielu systemach dolny obszar pamięci jest zarezerwowany, mieszczą się tam procedury inicjacji pracy procesora po kasowaniu i procedury obsługi przerwań.

Na zakończenie zwrócimy uwagę na niebezpieczeństwo czyhające na program używający stosu. Przykładowa procedura BIN, wskutek rekursywnych

wywołań, charakteryzuje się dużym obciążeniem stosu: 4 bajty przy każdym rekursywnym wywołaniu — 2 na adres powrotu i 2 na zapamiętanie parametrów lub wyniku cząstkowego. Może więc dojść do *przepełnienia stosu* — sytuacji, gdy kolejna operacja PUSH lub CALL spowoduje zniszczenie fragmentu programu lub danych, lub w ogóle wypadnięcie poza fizyczną przestrzeń adresową w danym systemie. Niech w dwu bajtach pamięci adresowanych etykietą SLIMIT znajduje się adres, który jest ustaloną dolną granicą zawartości SP. Zejście poniżej ma zostać potraktowane jako błąd. Przed krytycznymi rozkazami CALL lub PUSH należy wywołać następującą procedurę:

```
SLTEST  LD HL,(SLIMIT)   ; granica
        SBC HL,SP        ; odjęcie aktualnego adresu stosu
        RET C
        ...              ; diagnoza i czynności porządkowe
```

W SLIMIT można zostawić kilka bajtów rezerwy: na samo wywołanie SLTEST, na ewentualne przerwania itp. oraz aby móc zignorować niejednoznaczność 1 bajtu wynikającą z faktu, że wynik operacji SBC zależy od stanu wskaźnika przeniesienia. (Uniezależnienie się od niego przez dowolną operację ustalającą jego stan, np. przez rozkaz OR A albo SCF, może nie być opłacalne.)

6.3. Podprogramy przemieszczalne. Współprogramy

Przemieszczalnością będziemy określać własność podprogramu polegającą na tym, że może on być ładowany do różnych pól pamięci i nadal będzie wykonywany poprawnie. Jeśli gdziekolwiek w programie występuje pełny, dwubajtowy adres, np. w rozkazach LD *r*,(*nn*), CALL *nn* itp., to przesunięcie fragmentu programu zawierającego ten adres spowoduje dezorganizację programu. Taki program będziemy nazywali *absolutnym* albo *zakotwiczonym*. Jeśli jednak jeden fragment programu przekazuje sterowanie drugiemu przez skok względny JR *n*, to przesunięcie bloku zawierającego zarówno rozkaz skoku, jak i rozkaz, do którego jest wykonywany skok nie zaburzy poprawności programu. Taki właśnie program będziemy nazywali *przemieszczalnym*. Moduł przemieszczalny nie może zawierać jawnie żadnych adresów danych ani procedur należących do tego modułu. Wszystkie skoki muszą być względne.

Niektóre mikroprocesory, np. Motorola 6809, zostały tak zaprojektowane, aby ułatwić pisanie programów przemieszczalnych. Umożliwia to konstruowanie bibliotek niezależnych podprogramów bez konieczności dysponowania skomplikowanym programem ładującym i jest bardzo ważne w systemie operacyjnym z wieloprogramowością takim jak OS-9. Niestety Z80 nie jest pod tym względem wygodny. Z wyjątkiem skoków względnych nie ma wbudowanych

mechanizmów ułatwiających pisanie programów przemieszczalnych, lecz przy pewnym wysiłku można pisać programy niezależne od położenia zajmowanego przez nie w pamięci. Może to znaleźć zastosowanie np. w przypadku dopisywania podprogramów w kodzie maszynowym do programów w językach wyższego poziomu, jeśli kompilator generuje od razu wykonywalny kod absolutny i automatycznie przydziela pamięć dla podprogramów poza kontrolą programisty. Przy pisaniu programu przemieszczalnego należy pamiętać, że:

— Po pierwsze, wszystkie dane w programie muszą być adresowane względnie. Adres musi być obliczany przez dodanie adresu względnego (przesunięcia) do pewnego adresu bazowego związanego z położeniem, które w pamięci zajmuje program. Zauważmy, że w tym celu program musi dysponować informacją, w którym miejscu się zajmuje. Zmienne lokalne można przechowywać na stosie. W przypadku większej liczby danych adresowanych w dowolnej kolejności standardowy sposób operowania danymi na stosie przez PUSH i POP może być kłopotliwy, można jednak przekazać adres stosu do rejestru indeksowego

```
LD IX,0000
ADD IX,SP
```

i użyć adresowania indeksowanego.

— Po drugie, wszystkie skoki muszą być względne. Jeżeli adres, do którego należy skoczyć jest zbyt oddalony, można umieścić w programie pewną liczbę skoków „przesiadkowych".

— Po trzecie, wszystkie wywołania procedur muszą być zorganizowane specjalnie, ponieważ nie ma względnego rozkazu CALL. Jeśli jednakże potrafimy rozwiązać problem pierwszy, tj. adresowanie lokalnych danych, będziemy umieli również zorganizować przesłanie ich na stos i zrealizować dynamiczne wywołania.

Skąd procedura może się dowiedzieć, w którym miejscu pamięci się znajduje? W tym celu musimy dysponować pewnym niewielkim blokiem pamięci o ustalonym adresie absolutnym. Umieśćmy w nim następującą sekwencję rozkazów:

```
RETADDR POP HL
        JP  (HL)
```

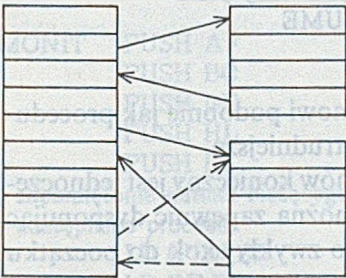
Teraz wywołanie

```
        CALL RETADDR
        . . .
```

dostarczy w rejestrze HL adres powrotu. Dysponując procedurą RETADDR można zorganizować względne wywołania podprogramów za pomocą następującego makrorozkazu:

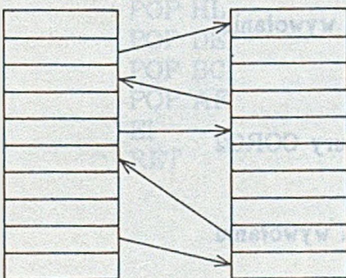
```
CALLR   MACRO PROC
        CALL RETADD
        LD BC,7         ; długość tego i dalszych 3 rozkazów
        ADD HL,BC
        PUSH HL
        JR PROC         ; wywołanie podprogramu PROC
        ENDM
```

Zakończenie tego rozdziału jest poświęcone *współprogramom* lub *współprocedurom* (zwanym także koprocedurami — ang. *coroutines*). Klasyczne podprogramy stanowią konstrukcje ściśle hierarchiczne: przekazywanie sterowania odbywa się w sposób pokazany na rys. 6.1. Zdarzają się jednak sytuacje, w



Rys. 6.1. Organizacja wywołań podprogramów

których fragmenty programu występują na równych prawach, jak partnerzy w grze, i wygodniej byłoby zorganizować przepływ sterowania na zasadach partnerskich, jak to przedstawiono na rys. 6.2. Współprocedury znajdują np.



Rys. 6.2. Organizacja wywołań współprogramów

zastosowanie w symulowaniu procesów współbieżnych, co jest interesujące z punktu widzenia zastosowań mikroprocesorów do celów sterowania i kontroli. Także we współczesnych standardach systemów operacyjnych zapoczątkowanych przez UNIX, które dotarły już na teren procesorów 8-bitowych, wiele składników systemu to współprocedury, które komunikują się ze sobą i programem użytkownika przez wspólne bufory i stanowią dla niego wirtualne urządzenia wejścia/wyjścia. Wypada więc o tej koncepcji powiedzieć parę słów.

Podstawową operację przekazującą sterowanie współprogramowi będziemy nazywali RESUME (wznowić). Nie istnieje ona na poziomie asemblera. Jej zadaniem jest spowodowanie wykonania skoku do instrukcji następującej po ostatnio wykonanym RESUME przez współprogram (a więc pełni rolę powrotu) z jednoczesnym zapamiętaniem miejsca wznowienia (a więc jest również wywołaniem). W przypadku dwóch współprocedur pierwszym wykonanym RESUME jest zwykły CALL pozostawiający na stosie adres powrotu. Następne będą się posługiwały „stacją przesiadkową" podobną do omówionej w poprzednim punkcie procedury CALLHL. Każde RESUME będzie wywołaniem CALL RESUME procedury, która ma postać

```
RESUME  POP HL          ; zdjęcie adresu powrotu ze stosu
        EX (SP),HL      ; wymiana na adres pozostawiony przez
                        ; poprzednie CALL RESUME
        JP (HL)         ; przekazanie sterowania
```

Parametry można przekazać współprogramowi podobnie jak procedurze, z tym, że wykorzystanie stosu jest teraz nieco trudniejsze.

W przypadku większej liczby współprogramów konieczny jest jednoczesny dostęp do wielu adresów powrotu, czego nie można zapewnić dysponując jednym stosem. Wtedy zrealizujemy RESUME jako zwykły skok do początku współprocedury poprzedzony zapamiętaniem adresu powrotu nie na stosie, lecz w statycznych zmiennych lokalnych. Każda współprocedura w momencie otrzymania sterowania odzyskuje ten adres i skacze w odpowiednie miejsce

```
CORO1   LD HL,(MEMC1)   ; pobranie adresu restartu
        JP (HL)         ;  i skok
COINI1  . . .           ; wykon. przy pierwszym wywołaniu
        . . .
        LD BC,RET1      ; adres restartu
        LD (MEMC1),BC
        JP CORO2        ; wywołanie współprocedury CORO2
RET1    . . .
        . . .
MEMC1   DEFW COINI1     ; zmienione po pierwszym wywołaniu
```

Takie rozwiązanie jest celowe, gdy współprocedury są od siebie niezależne i CORO2 nie ma dostępu do informacji, gdzie należy przechowywać adres restartu dla CORO1. Możemy mieć także do czynienia z sytuacją, gdy współprogramy nie przekazują sobie sterowania bezpośrednio, lecz przez jeden wspólny program monitorujący. Tak jest np. gdy procesor obsługuje większą liczbę podobnych urządzeń za pomocą mniejszej liczby procedur, z których każda pracuje na różnych zestawach danych lokalnych. Najwygodniej jest wtedy dysponować jednym programem nadrzędnym, który zajmie się przełączaniem.

Jako przykład skonstruujemy fragment takiego monitora. Niech system zawiera N niezależnych procesów — współprocedur, z których każda ma swój własny stos i lokalne dane adresowane przez rejestr IX. Procesy nie będą się ze sobą komunikowały, natomiast będą okresowo przerywane, a procedurą obsługi przerwania będzie nasz monitor. Jego zadaniem będzie cykliczne przekazywanie sterowania wszystkim procesom po kolei. Monitor będzie dysponował tablicą PROCTAB zawierającą adresy wierzchołków stosów poszczególnych procesów w odpowiedniej kolejności. Tablica ta, o długości $2 \cdot N + 2$ bajtów, będzie obsługiwana jako cykliczna kolejka. Zmienna FIN będzie adresem $(N+1)$-ego, dodatkowego elementu tablicy PROCTAB odpowiadającego bieżącemu obsługiwanemu procesowi. Zmienna NPROC zawiera długość kolejki w bajtach, tj. $2 \cdot N$.

```
MONIT   PUSH AF            ; przechowanie rejestrów; na bieżącym
        PUSH BC            ;  stosie jest już adres restartu
        PUSH DE            ;  zapamiętany w momencie przerwania
        PUSH HL
        PUSH IX
; zapamiętanie adresu bieżącego stosu i uruchomienie
; następnego procesu
        LD (FIN),SP        ; zapamiętanie wierzchołka stosu
        LD DE,PROCTAB
        LD HL,PROCTAB+2
        LD BC,(NPROC)
        LDIR               ; przesunięcie kolejki
        LD SP,(PROCTAB)    ; stos wznawianego procesu
        POP IX             ; odtworzenie rejestrów
        POP HL
        POP DE
        POP BC
        POP AF
        EI                 ; odblokowanie przerwań
        RET                ; restart
```

6.4. Rozgałęzienia i pętle

Przypuśćmy, że program ma wykonać dwie sekwencje instrukcji w zależności od tego, czy wartość pewnego wyrażenia jest równa zeru czy nie. Instrukcję

if $t = 0$ **then** *seq*1 **else** *seq*2;

tłumaczymy na język asemblera Z80 następująco:

```
TESTT   . . .          ; ustawienie rejestru F w zależności
                       ;  od wartości t
        JR NZ,ELSE
THEN    . . .          ; sekwencja seq1
        JR ENDIF
ELSE    . . .          ; sekwencja seq2
ENDIF   . . .          ; dalszy ciąg programu.
```

W przypadku braku członu **else** *seq*2 adres ELSE pokrywa się z EXIT; sekwencja *seq*2 nie istnieje i instrukcja JR EXIT jest pusta i zbędna.

Czy jest tu coś do dodania oprócz uwagi, że niekiedy może się opłacać przeformułowanie powyższej pełnej instrukcji if w poniższy sposób?

if $t <> 0$ **then** *seq*2 **else** *seq*1;

Otóż czasami mamy do czynienia nie z pojedynczą alternatywą, lecz całą kaskadą warunków:

if *test*1 **then** *seq*1 **else**
  if *test*2 **then** *seq*2 **else**
    ....
    if *testn* **then** *seqn* **else** *seq*0;

A w języku asemblera:

```
TEST1   . . .          ; sprawdzanie warunku 1
        JR NZ,TEST2    ; jeśli niespełniony – skok
SEQ1    . . .          ; jeśli spełniony – wykonać seq1
        JR EXIT        ; ...i zakończyć instrukcję
TEST2   . . .          ; sprawdzanie warunku 2
        JR NZ,TEST3    ; . . .
SEQ2    . . .
        JR EXIT
TEST3   . . .
        . . .
TESTN   . . .
        JR NZ,SEQ0
SEQN    . . .
        JR EXIT
SEQ0    . . .          ; żaden warunek nie spełniony
EXIT    . . .          ; dalszy ciąg programu.
```

W przypadku gdy warunki wykluczają się wzajemnie, a poszczególne sekwencje instrukcji nie niszczą wskaźnika Z, istnieje możliwość uzyskania pewnej oszczędności pamięci kosztem przedłużania czasu wykonania. Można po prostu usunąć wszystkie instrukcje JR EXIT z wyjątkiem ostatniej. Nie jest to chwyt charakterystyczny jedynie dla asemblera, ale na ogół tego typu oszczędności są opłacalne podczas programowania na niskim poziomie. Typowym

przykładem jest okresowe czytanie znaków z klawiatury i podejmowanie decyzji w zależności od przeczytanego znaku. Wtedy oczywiście opóźnienie wywołane dodatkowymi testami jest zupełnie bez znaczenia.

W przypadku wielokrotnego rozgałęzienia bardziej zwartą i efektywniejszą konstrukcją niż kaskada instrukcji warunkowych jest instrukcja typu **case** w Pascalu czy **switch** w Algolu lub „C" względnie „COMPUTED GOTO" w Fortranie. Niech sterowanie będzie przekazane jednej z kilku sekcji w zależności od wartości akumulatora numerowanej kolejno od zera. Umieszczamy w programie tablicę zawierającą instrukcje skoku (dla ogólności niech to będą skoki długie)

```
JPTAB   JP SEQ1
        JP SEQ2
        JP SEQ3
        . . .
        JP SEQN
```

a sterowanie przekazujemy następująco:

```
        LD HL,JPTAB     ; początek tablicy
        LD C,A          ; przechowanie akumulatora
        ADD A,A         ; pomnożenie przez dwa
        ADD A,C         ; razem: pomnożenie przez trzy
        LD C,A          ; BC zawiera adres skoku
        LD B,0          ; względem JPTAB
        ADD HL,BC
        JP (HL)         ; skok pośredni
```

I tu można oszczędzić sporo pamięci umieszczając w tablicy nie instrukcje skoku, lecz jedynie adresy sekcji. Niech tablica ta nosi nazwę ADRTAB. Powyższy program zastąpimy wtedy następującym:

```
        LD HL,ADRTAB    ; początek tablicy adresów
        ADD A,A         ; pomnożenie przez 2
        LD C,A
        LD B,0
        ADD HL,BC       ; adres odpowiedniego elementu
        LD A,(HL)       ; pobranie jego zawartości
        INC HL
        LD H,(HL)
        LD L,A          ; do HL
        JP (HL)         ; i skok
```

Zauważmy, że tablica skróciła się o czynnik 2/3, a program organizujący skoki wcale się bardzo nie wydłużył. Jeszcze większa oszczędność jest możliwa,

jeśli sekcje odpowiadające różnym warunkom są krótkie i wystarczy umieścić w tablicy jednobajtowe adresy względne, przy czym wygodniej jest odliczać te adresy nie względem początku tablicy, lecz względem adresu, pod którym jest umieszczone przesunięcie.

Niech symbol $ oznacza adres aktualnie tłumaczonej instrukcji. (Tu jest miejsce dużej liczby pomyłek: wiele asemblerów traktując symbol dolara jako oznaczenie licznika rozkazów w momencie wykonywania tłumaczonego programu przypisuje mu adres następnej instrukcji. My jednak użyjemy symbolu $ jedynie w dyrektywie DEFB.) Tablica ADRTAB zostanie skonstruowana następująco:

```
ADRTAB DEFB SEQ1-$
       DEFB SEQ2-$
       . . .
       DEFB SEQN-$
```

sam zaś program:

```
       LD HL,ADRTAB   ; adres początku tablicy
       LD C,A         ; przekazanie wartości A do BC
       LD B,0
       ADD HL,BC      ; adres odpowiedniego elementu
       LD C,(HL)      ; pobranie zawartości
       ADD HL,BC      ; obliczenie adresu
       JP (HL)        ;  i skok
```

Nie będzie dużą przesadą stwierdzenie, że większy program praktycznie cały czas swojego „życia" spędza w jakiejś pętli. Organizacja pętli jest więc „żelaznym" elementem programowania. Pętlę, która ma się wykonywać z góry określoną liczbę razy, nie większą niż 256, najprościej jest zorganizować przy użyciu rozkazu DJNZ. Fragment programu

```
       LD B,NUM
LOOP   . . .
ENDLOOPDJNZ LOOP
```

wykona pętlę: LOPP...ENDLOOP, NUM razy. Należy tylko pamiętać o tym, że rozkaz DJNZ najpierw dekrementuje rejestr B, a potem sprawdza jego zawartość i ewentualnie wykonuje skok warunkowy. Tu jest często miejsce pomyłek. Powyższa pętla zawsze wykona się przynajmniej raz. Pętla, która może w ogóle nie zostać wykonana wymaga sprawdzenia warunku na początku. Przy użyciu DJNZ można to zorganizować następująco:

```
        LD B,NUM
        INC B
LOOP    DJNZ ENDLOOP
        . . .
        DJNZ LOOP+2
ENDLOOP. . .            ; dalszy ciąg programu
```

Organizując pętle przy użyciu innych rejestrów niż B i sprawdzając jawnie wskaźniki stanu, należy mieć na uwadze następujące własności listy rozkazów Z80:

1. Inkrementacja lub dekrementacja rejestru 16-bitowego nie zmienia rejestru F. Jeśli wartość HL równa zeru ma przerwać pętlę, to można to zorganizować następująco:

```
        DEC HL
        LD A,H
        OR L
        JR NZ,LOOP      ; skok do początku pętli
```

Oczywiście, jedynkę można odjąć inaczej

```
        LD DE,0001
        . . .
LOOP    . . .           ; początek pętli
        . . .
        SBC HL,DE
        JR NZ,LOOP      ; skok do początku pętli
```

Jednak np. instrukcja ADD HL,DE nie zmienia wskaźnika Z. Trzeba wtedy tak przeformułować warunek kończący pętlę, aby można się posłużyć wskaźnikiem przeniesienia.

2. Rozkazy inkrementacji i dekrementacji rejestru 8-bitowego nie zmieniają wartości wskaźnika przeniesienia. Zmiana wartości rejestru z 0 na 255 jest wykrywalna dzięki wskaźnikowi S. (Jednak wtedy trzeba zastosować długi skok warunkowy JP M,..., gdyż odpowiedni „JR M" nie istnieje.)

Na zakończenie rozdziału podamy przykład procedury generowania impulsów prostokątnych o określonej częstotliwości i czasie trwania. Będzie to przykład konstrukcji pętli o określonej charakterystyce czasowej. Impulsy będą generowane przez wysyłanie na przemian bitów 0 i 1 do odpowiedniego urządzenia wyjściowego — dla ustalenia uwagi niech wysyłany będzie bit nr 0 przesyłanego bajtu; adresem portu będzie zaś 254 (# FE). Liczba na początku komentarza będzie oznaczała długość rozkazu wyrażoną w taktach zegarowych. Zapis np. 7/12 oznacza 7 taktów, jeśli rozkaz skoku nie zostanie wykonany, a 12 przy skoku wykonanym. Rzeczywista częstotliwość w hertzach i długość w sekundach będzie zależeć od częstotliwości zegara taktującego procesor.

```
; Procedura generowania impulsów prostokątnych.
; Parametry wejściowe:
; DE - długość okresu, HL oraz B - długość trwania fali.
; Długość generowania fali jest podana w jednostkach absolut-
; nych, a nie jako liczba okresów. Jest to liczba 3-bajtowa,
; rejestr B zawiera najbardziej znaczący bajt, rejestr HL dwa
; mniej znaczące.
        DI              ; zablokowanie przerwań
        CALL GEN
        EI              ; odblokowanie
        RET
;
GEN     LD (MEMDE),DE   ; zapamiętanie wartości DE
        XOR A           ; wyzerowanie akumulatora
        INC B           ; test zera nastąpi po dekrementacji
;
LOOP    OUT (#FE),A     ;11    wysłanie 0 lub 1
        XOR 01          ;7          zmiana  0 <-> 1
        LD C,A          ;4          chwilowe przechowanie
        LD DE,(MEMDE)   ;20       pobranie długości okresu
        SBC HL,DE       ;15    odjęcie
        JR NC,LOP1      ;7/12  skok jeśli HL >= DE
        DEC B           ;4          dekrementacja pierwszego bajtu
LOP1    DEC DE          ;6     początek pętli opóźniającej
        LD A,D          ;4
        OR E            ;4
        JR NZ,LOP1      ;7/12  ...i koniec
        OR B            ;4          sprawdzenie czy B=0
        RET Z           ;5/11  powrót jeśli koniec
        LD A,C          ;4          odtworzenie wartości A
        JP LOOP         ;10         skok i następne pół okresu
MEMDE   DEFS 2
```

Charakterystyka czasowa tej pętli nie jest idealna. Dobrym, choć niełatwym treningiem będzie próba poprawienia tej procedury. Zewnętrzna pętla jest sterowana odejmowaniem od licznika (B,HL) zawartości DE. W ten sposób, jeśli okres impulsów jest dłuższy, to odpowiednio mniej zostanie ich wygenerowanych. Jeśli wartość HL była większa lub równa DE, to skok do LOP1 zostanie wykonany, co potrwa 12 taktów. Jeśli HL było mniejsze, to niewykonanie skoku i dekrementacja B razem zabiorą 11 taktów. Tę niedokładność potraktujemy jako nieznaczącą. Pętla opóźniająca zaczynająca się od LOP1 trwa $26 \cdot DE - 5$ taktów. Cały półokres impulsu — $26 \cdot DE + 87$ taktów. Tak więc, jeśli częstotliwość zegara wynosi w, a żądaną częstotliwością fali prostokątnej jest f, odpowiednią wartość DE można obliczyć ze wzoru

$$f = w/(52 \cdot DE + 174)$$

Procedura może generować impulsy o częstotliwości do kilku tysięcy Hz. Aby na przykład otrzymać częstotliwość równą 300 Hz przy częstotliwości zegara równej 3,5 MHz, należy rejestrowi DE nadać wartość równą 221. Jak łatwo obliczyć, przy powyższych parametrach wartości B = 2 i HL = 1528 spowodują wygenerowanie ciągu impulsów o czasie trwania równym 1 s. Obecność stałego składnika 87 powoduje, że dla mniejszych wartości DE czas generowania będzie zależał nieco od DE. Wzięcie odpowiedniej poprawki wymaga już bardziej zaawansowanej arytmetyki.

6.5. Programy sterowane danymi. Techniki interpretacji i ich wykorzystanie

Aczkolwiek ten rozdział ma dość techniczny charakter, zagadnienie w nim poruszane wiąże się również z pewną filozofią dotyczącą stosunku programisty do komputera i programów.

Decydując się na napisanie programu w języku asemblera praktycznie w pełni podporządkowujemy się technicznym wymogom procesora. Formułujemy nasz program w takim języku, jaki jest wygodny dla maszyny. Programując w języku wyższego poziomu, takim jak np. Pascal czy Fortran, jesteśmy pod tym względem swobodniejsi, czarną robotę wykonuje za nas *kompilator* (ang. *compiler*).

Można jednak na problem spojrzeć inaczej: zamiast dostosowywać program do komputera można dostosować komputer do programu. Dzięki odpowiedniemu oprogramowaniu można nauczyć komputer rozumienia dowolnego języka programowania i wykonywania jego instrukcji bez potrzeby tłumaczenia ich na kod maszynowy. Takie oprogramowanie nosi nazwę *interpretatora* (ang. *interpreter*). Na ogół interpretator języka programowania jest o wiele bardziej zwarty i prostszy do napisania niż kompilator tego języka, gdyż kompilator musi jednocześnie operować dwoma zupełnie odmiennymi modelami semantycznymi. Programy interpretowane są często znacznie krótsze niż odpowiednie programy w kodzie maszynowym. Na przykład instrukcja drukowania w typowym interpretowanym języku, jakim jest Basic może mieć postać

```
PRINT X
```

i zajmować w pamięci 2 bajty, jeden na jednobajtowy kod instrukcji PRINT i jeden na nazwę X. Kompilacja tej instrukcji na kod maszynowy może dać w wyniku

```
LD HL,X          ; tu X oznacza adres zmiennej
CALL PRINT       ; podprogram drukowania
```

co zajmuje już 6 bajtów. (Pomijając drukowanie znaku końca wiersza.) Programy w kodzie interpretowanym mogą być prawie w pełni przenośne — podobnie jak teksty źródłowe programów w językach wyższego poziomu. Zabezpieczenie przed błędami wykonania i ich diagnozy są łatwiejsze do zorganizowania. Podstawową ceną, jaką trzeba zawsze zapłacić jest prędkość wykonania. Ponadto, część pamięci jest zajęta przez rezydujący program interpretatora.

Okazuje się jednak, że w niewielkich systemach mikroprocesorowych prościej jest zainstalować niewielki interpretator niż kompilatory, programy ładujące itp. Nieefektywność czasowa wcale nie musi być wielka. Wiąże się to z zauważonym już uprzednio faktem, że w przypadku procesorów dysponujących krótkim słowem maszynowym i niezbyt wyrafinowaną listą rozkazów, dominującym elementem programu są wywołania podprogramów i większość czasu program spędza w tych podprogramach, np. wykonujących obliczenia zmiennopozycyjne czy szukających bądź przemieszczających bloki danych. Nawet tak prosta operacja jak przesłanie liczby zmiennopozycyjnej do lub z procedury wymaga wielobajtowego ciągu instrukcji.

Również nieefektywność pamięciowa nie musi być bardzo znacząca. Część organizacyjna interpretatora z reguły jest niewielka, najwięcej pamięci potrzeba na podprogramy wykonawcze. Także w przypadku programów kompilowanych będą one zajmować najwięcej miejsca. W przypadku interpretatora w pamięci będą rezydować też te podprogramy, które nie są wykorzystywane. Jest to dużym obciążeniem w przypadku interpretatorów uniwersalnych, jak np. interpretator Basicu, który jest standardowym wbudowanym programem w większości małych komputerów domowych. Są też wyspecjalizowane, małe interpretatory zawierające tylko kilka niezbędnych podprogramów w kodzie maszynowym, mogące być jednak dowolnie rozszerzane przez procedury, które dopisuje się już w kodzie interpretowanym. W ten właśnie sposób jest zbudowana większość realizacji języka programowania FORTH, który został pomyślany pierwotnie jako język do pisania zwartych, szybkich i przenośnych programów, do sterowania urządzeń radioastronomicznych przy użyciu minikomputerów. Kilkadziesiąt procent interpretatora języka FORTH jest napisane w nim samym.

Typowy interpretator składa się z następujących części (pominiemy tu procedury wejściowe, które czytają program źródłowy np. z klawiatury i tłumaczą na wynikowy kod interpretowany; takie procedury będą występować również w kompilatorach, tylko kod pośredni jest następnie tłumaczony dalej na kod maszynowy):

1. Pętli interpretatora, która pobiera następną instrukcję programu z miejsca wskazywanego przez rejestr pełniący rolę licznika rozkazów oraz inkrementuje ten rejestr.

2. Dekodera, który zamienia kod instrukcji na adres właściwego podprogramu i przekazuje mu sterowanie poprzez odpowiednie instrukcje rozgałęzienia.

3. Pakietu podprogramów wykonawczych.

Część organizacyjna interpretatora jest szczególnie prosta, jeśli kodami interpretowanych rozkazów są po prostu adresy procedur wykonawczych. Niech rolę licznika rozkazów pełni rejestr HL'. Cała pętla interpretatora to następujący program:

```
INTERP EXX
       LD C,(HL)      ; pobranie 2-bajtowego adresu z (HL)
       INC HL         ; i (HL+1) do BC
       LD B,(HL)
       INC HL         ; inkrementacja HL
       PUSH BC
       EXX
       RET            ; skok do procedury
```

Każda procedura wykonawcza kończy działanie nie przez RET, lecz przez JP INTERP. Ułatwia to wykorzystanie stosu do przekazywania danych między procedurami.

Jako przykład zastosowania techniki interpretacji proponujemy zawarty w Dodatku A program deasemblera tłumaczący rozkazy w kodzie maszynowym Z80 na mnemonikę asemblera. Ze względu na nieregularność listy rozkazów Z80, program deasemblera musi być dość skomplikowany. Można tego uniknąć, ale kosztem sporego powiększenia rozmiaru tablic zawierających zdekodowane instrukcje. W każdym razie przy użyciu technik bezpośrednich deasembler Z80 zajmowałby przynajmniej 2 do 3 K bajtów pamięci. Deasembler załączony w Dodatku A zajmuje poniżej 1 K bajta. Jest zorganizowany jako prosty interpretator o jednobajtowych rozkazach. Zawiera 8 procedur wykonawczych, więc w jednym bajcie rozkazu oprócz numeru procedury jest jeszcze miejsce na parametry przekazywane procedurze. Połowę całego programu zajmują dane, które są właściwym programem interpretowanym wraz ze swoimi danymi: mnemonicznymi nazwami rozkazów, nazwami rejestrów itp. Pomysł tego deasemblera jest naszkicowany w książce [2].

7. Operacje arytmetyczne i logiczne

7.1. Operacje na bitach i ich zastosowanie

W tym rozdziale zajmiemy się operacjami, które traktują informację zawartą w bajcie lub grupie bajtów jako ciąg bitów w oderwaniu od wartości numerycznej bajtu, choć bardzo ważnym zastosowaniem tych operacji jest organizacja obliczeń arytmetycznych na wielobajtowych strukturach danych. Są to operacje przesunięć i obrotów, a także maskowanie i wycinanie fragmentów bajtów. I tak, najprostszym ciągiem operacji, który mnoży zawartość rejestru BC przez 2 jest

```
SLA C      ; C:=C*2 bit przeniesienia zostanie
RL  B      ; przekazany do rejestru B
```

Choć podobna operacja dotycząca rejestru HL jest cztery razy krótsza (i odpowiednio szybsza)

```
ADD HL,HL
```

więcej czasu i pamięci może zająć przesyłanie informacji między rejestrami. Ponadto pierwszy wariant w naturalny sposób przedłuża się o następne operacje RL rejestrów zawierających bardziej znaczące cyfry wielobajtowej liczby. Dla rejestru DE jest prostsze rozwiązanie

```
EX DE,HL
ADD HL,HL
EX DE,HL
```

Umiejętne użycie operacji logicznych może znacznie uprościć strukturę decyzyjną programu. Na przykład mamy zaprogramować instrukcję

if *przeniesienie* **then** $a := n1$ **else** $a := n2$

Klasycznym rozwiązaniem jest

```
        LD A,n2         ; wstępna decyzja
        JR NC,EXIT
        LD A,n1
EXIT    ...             ; dalszy ciąg programu
```

Wstępne przypisanie wartości *n*2 jest już znanym chwytem pozwalającym uniknąć jednego rozkazu skoku. Ten sam efekt osiągniemy jednak w ogóle bez skoków

```
        SBC A,A         ; # FF gdy jest przeniesienie lub 0
        AND n1-n2       ; n1-n2 lub 0
        ADD A,n2        ; n1 lub n2
```

Powyższe instrukcje stanowią przykład maskowania warunkowego. *Maskowanie*, tj. wymuszone zerowanie (może być również ustawianie bądź pomijanie) określonych bitów przez zastosowanie pewnej operacji logicznej z przygotowaną uprzednio konfiguracją bitów zwaną *maską*, może mieć i inne zastosowania. Jako przykład napiszmy ciąg instrukcji zliczający, ile jest ustawionych bitów w rejestrze C. Wynik zostanie umieszczony w akumulatorze. Klasycznym rozwiązaniem jest 8-krotne przesunięcie rejestru z dodawaniem do akumulatora jedynki, gdy pojawi się przeniesienie. Korzystając z przedstawionej powyżej nauki unikniemy skoku omijającego inkrementację akumulatora.

```
        XOR A
        LD B,08
LOOP    RR C            ; obrót C; mógłby być w lewo
        ADC A,00        ; dodanie bitu przeniesienia
        DJNZ LOOP       ; powrót do początku pętli
```

Krytyczny Czytelnik zauważy, że zamiast wykonywać 8 razy pętlę można sprawdzić, czy po obrocie zawartość C nie jest już równa zeru. Istnieje jednak jeszcze szybsze rozwiązanie. Otóż wykonanie operacji X **and** $(X-1)$ zeruje najmniej znaczący ustawiony bit X. Teraz wygodniej będzie zaczynać od argumentu w akumulatorze. Wynik zostanie umieszczony w rejestrze C.

```
        LD C,00
        OR A
        JR Z,EXIT       ; jeżeli od początku zero
LOOP    INC C
        LD B,A          ; przechowanie argumentu X
        DEC A           ; X - 1
        AND B           ; (X-1) and X
        JR NZ,LOOP
EXIT    ...             ; dalszy ciąg programu
```

W podobny sposób można osiągnąć i inne efekty. Na przykład wyizolowanie najmniej znaczącego bitu z X (zerując wszystkie inne) najprościej otrzymać przez obliczenie X **and** $(-X)$, gdzie $-X$ jest negacją liczby w dwójkowym kodzie uzupełnieniowym.

Rozkazy BIT, SET i RES są nadmiarowe, w tym sensie, że ich efekt można osiągnąć używając odpowiedniej maski. Główną zaletą rozkazów jednobitowych jest to, że nie zmieniają one zawartości innych rejestrów i nie potrzeba specjalnie angażować akumulatora.

Przy rozwiązywaniu zadań związanych z maskowaniem niezastąpiona jest operacja XOR. Przypuśćmy, że w bajcie pamięci adresowanym przez HL mamy umieścić daną zawartą w akumulatorze, ale nie całą, lecz tylko fragment bajtu. W rejestrze B znajduje się maska bitowa. Te bity akumulatora, które odpowiadają bitom maski równym 1 należy umieścić w (HL), pozostałe bity komórki pamięci nie ulegają zmianie. Najkrótszym rozwiązaniem jest

```
XOR (HL)
AND B
XOR (HL)
LD  (HL),A
```

7.2. Prosta arytmetyka na liczbach całkowitych. Konwersja liczb z postaci zewnętrznej na dwójkową i odwrotnie

Najprostszą konwersją liczby dwójkowej na postać nadającą się do wydruku jest konwersja na postać szesnastkową. Wystarczy podzielić ciąg bitów tworzący liczbę (dowolnej długości) na tetrady i do liczby będącej wartością tetrady dodać kod zewnętrzny (np. ASCII) znaku „0". W kodzie ASCII litery nie następują bezpośrednio po cyfrach, więc jeśli cyfra przekracza „9" należy jeszcze dodać 7. Podprogram drukowania jedno- i dwubajtowych liczb w układzie szesnastkowym jest zawarty w programie deasemblera w dodatku A.

Podobnie przeczytanie i zamiana liczby szesnastkowej na postać wewnętrzną nie sprawia dodatkowych trudności; każdy bajt tekstu będzie odpowiadał jednej tetradzie.

Z oczywistych względów wszystko co dotyczy konwersji szesnastkowej można prawie bez zmian zastosować do konwersji liczb BCD, dla których każda tetrada odpowiada również cyfrze. Jest to nawet prostsze, gdyż nie potrzeba osobno rozpatrywać cyfr większych od „9". (Pomijając sprawdzanie poprawności.)

Wprowadzanie liczby dziesiętnej i jej konwersja z postaci znakowej na wewnętrzną polega na czytaniu kolejnych cyfr i dodawaniu odpowiedniej wartości do już skonstruowanej części liczby pomnożonej przez 10. Tak więc już na tym poziomie dobrze jest dysponować podprogramem mnożenia. Z kolei wyprowadzanie liczby całkowitej w postaci dziesiętnej wymaga dzielenia przez 10, aby zrekonstruować poszczególne cyfry.

Zapiszemy te procedury w uproszczonym Pascalo-podobnym języku, który nie wymaga osobnego omawiania. Procedurę czytania liczby i umieszczania jej w zmiennej *N* najprościej jest zorganizować w postaci pętli

```
N:=0;
readchar (c);
while ASCII ('0') <= ASCII(c) <= ASCII('9') do
    begin
        N:=N*10+(ASCII(c)-ASCII('0'));
        readchar(c)
    end;
```

Z kolei procedurę drukowania wartości zmiennej *N* najłatwiej jest napisać rekursywnie. Nie znając wartości liczby trudno jest od razu określić jej pierwszą cyfrę, natomiast ostatnią można wyznaczyć jako resztę z dzielenia przez 10. Tę cyfrę można odciąć biorąc część całkowitą ilorazu i powtórzyć operację. Cyfry zapamiętane na stosie będą następnie drukowane w odwrotnej kolejności.

```
procedure print(N);
    if N < 10 then printchar(N+ASCII('0'))
        else begin
                print(N div 10);
                printchar(N mod 10+ASCII('0'))
            end;
```

ASCII('0') jest kodem cyfry „0” wynoszącym 48(#30). Kod dziewiątki jest o 9 większy.

Teraz powyższe procedury zapiszemy w języku asemblera dla dwubajtowych liczb całkowitych traktowanych jako dodatnie. Czytanie będzie polegało na pobieraniu znaków z tablicy o adresie umieszczonym w zmiennej TEXT. Wynik zostanie dostarczony w rejestrze HL. Wykorzystanie uniwersalnej procedury wykonania mnożenia przez 10 nie jest w tym przypadku opłacalne. Mnożenie przez potęgę dwójki jest bardzo proste, a $10 = 8 + 2$. Czasami korzysta się jednak z gotowej procedury mnożącej. Jest to celowe, gdy mamy do czynienia z liczbami wielobajtowymi. Można wtedy zaoszczędzić pamięć, a strata czasu przy konwersji liczb na ogół jest bez znaczenia.

```
        LD HL,0000    ; inicjacja wartości wyniku
        LD D,0        ; bardziej znaczący bajt cyfry
        LD IX,(TEXT)  ; adres tablicy znaków
LOOP    LD A,(IX)     ; pobranie znaku
        SUB 48        ; odjęcie kodu ASCII zera
        RET C         ; powrót jeśli kod za mały
        CP 10
        RET NC        ; kod za duży?
        LD B,H
        LD C,L        ; zapamiętanie w BC
        ADD HL,HL     ; pomonożenie przez 2
        ADD HL,HL     ; przez 4
        ADD HL,BC     ; razem: przez 5
        ADD HL,HL     ; ostatecznie przez 10
        LD E,A
        ADD HL,DE     ; dodanie wartości cyfry
        INC IX        ; zwiększenie adresu znaku
        JR LOOP       ; i skok do początku pętli
```

Czy można napisać prostą procedurę dzielenia przez 10? Jest to bardziej skomplikowany problem. Jednak dla małych liczb istnieje procedura o wiele prostsza niż procedura uniwersalna. Należy zauważyć, że liczbę 1/10 można zapisać jako 1/(8+2) i przedstawić w postaci rozwinięcia geometrycznego:

$$\frac{1}{(8+2)} = \frac{1}{8}\cdot\left(1 - \frac{1}{4} + \frac{1}{16} - \frac{1}{64} + \frac{1}{256} - \frac{1}{1024} + \ldots\right)$$

Dla liczb mniejszych niż 256 tylko pierwsze 3 człony dadzą przyczynki różne od zera. Dla liczb dwubajtowych trzeba ich 7. W podobny sposób można konstruować wyspecjalizowane procedury do dzielenia przez 15 itp. Im dzielnik jest bliższy potędze dwójki, tym procedura łatwiejsza do zaprogramowania i szybciej zbieżna. To już zostawimy jako samodzielne ćwiczenie.

Prostym i użytecznym przykładem użycia wyspecjalizowanej procedury mnożenia liczb całkowitych jest kongruencyjny generator liczb pseudolosowych z przedziału 0...65535, działający według algorytmu:

$$X(n+1) = \big((X(n)\cdot 65) + 7\big) \bmod 65536$$

(Współczynniki są przykładowe. Nie jest to idealny generator, ale może być praktycznie wykorzystany i jest bardzo szybki.) Operując liczbami dwubajtowymi automatycznie używamy reszt modulo 65536. Kolejne wyniki będą przekazywane w rejestrze HL.

```
RANDOMLD D,H
        LD E,L          ; przechowanie HL w DE
        LD B,06         ; liczba przebiegów pętli
RLOOP   ADD HL,HL
        DJNZ RLOOP      ; razem: * 64
        ADD HL,DE       ; razem: * 65
        LD DE,0007
        ADD HL,DE
        RET
```

Procedura mnożenia liczb 8-bitowych jest bardzo prosta i można jej opis bez trudu znaleźć w literaturze. Napiszemy procedurę, nieco bardziej uniwersalną, mnożenia liczby 16-bitowej przez 8-bitową. Posłuży ona jako główny podprogram w procedurze mnożenia liczb 16-bitowych i oczywiście będzie mogła służyć także do mnożenia liczb 8-bitowych. Liczby będą uważane za dodatnie, bez znaku (tj. do 65535 i do 255).

```
; Procedura MULT21 mnożenia liczby 16- przez 8-bitową.
; Mnożna:    rejestr DE
; Mnożnik:   rejestr A
; Najbardziej znaczący bajt wyniku:  rejestr A
; Dwa mniej znaczące bajty wyniku:   rejestr HL
MULT21
        LD HL,0000      ; inicjacja wyniku
        LD B,08         ; inicjacja pętli
NXBIT   ADD HL,HL       ; przesunięcie wyniku w lewo
        RLA             ; sprawdzenie cyfry mnożnika
        JR NC,FIN21     ; opuszczenie zera
        ADD HL,DE       ; dodanie mnożnej
        ADC A,00        ; zapamiętanie przeniesienia
FIN21   DJNZ NXBIT      ; powrót do początku pętli
        RET
```

Charakterystycznym elementem procedury MULT21 jest wykorzystanie akumulatora jednocześnie jako mnożnika i rejestru zapamiętującego trzeci bajt wyniku. Można to uzyskać dzięki temu, że przy każdym obrocie zawartości akumulatora zwalnia się miejsce na jeden bit z jego prawej strony.

Napiszmy teraz procedurę mnożenia liczb dwubajtowych umieszczonych w rejestrach HL i DE. Metoda postępowania będzie polegała na pomnożeniu L przez DE, a następnie H przez DE i dodaniu do siebie obu wyników po przesunięciu ostatniego wyniku o jeden bajt w lewo.

```
; Procedura MULT22 mnożenia liczb dwubajtowych
; Argumenty: zawartości rejestrów  HL i DE
; Bardziej znaczące bajty wyniku:  HL
; Mniej znaczące bajty wyniku:   DE
MULT22
        LD C,H          ; przechowanie H
        LD A,L
        CALL MULT21     ; pomnożenie DE * L
        PUSH HL         ; zapamiętanie wyniku na stosie
        LD H,A
        LD A,C          ; poprzednie H do akumulatora
        LD C,H          ; trzeci bajt poprzedniego wyniku
        CALL MULT21     ; pomnożenie DE * H
        POP DE          ; E już zawiera końcowy wynik
        LD B,C          ; dwa środkowe bajty wyniku do BC
        LD C,D
        ADD HL,BC       ; i dodane do HL, a przeniesienie
        ADC A,00        ; poprawi wartość H
        LD D,L          ; drugi bajt wyniku
        LD L,H          ; trzeci
        LD H,A          ; i ostatni poprawiony
        RET
```

Procedura ta jest czytelniejsza, a także szybsza niż procedura, która mnożyłaby liczby przesuwając cały 16-bitowy rejestr.

Teraz napiszmy procedurę dzielenia liczby 16-bitowej przez 8-bitową, dostarczającą iloraz całkowity i resztę. To pozwoli nam skonstruować podprogram dokonujący konwersji liczby dwójkowej na zewnętrzną postać dziesiętną. Metoda postępowania użyta w tej procedurze jest klasyczna: od dzielnej odejmuje się odpowiednio przesunięty dzielnik. Jeśli wynik odjęcia jest nieujemny, bitem ilorazu jest 1, w przeciwnym razie 0. Dzielna i dzielnik są traktowane jako liczby dodatnie bez znaku.

Dzielnik musi być większy od bardziej znaczącego bajtu dzielnej, aby iloraz mieścił się w jednym bajcie.

```
; Procedura DIV21 dzielenia całkowitego
; Dzielna:   HL
; Dzielnik:  D
; Iloraz:    A
; Reszta:    HL (właściwie tylko w L)
; Po zakończeniu procedury rejestr E zawiera dzielnik
; Procedura nie niszczy rejestru C.
```

```
DIV21   LD B,09        ; inicjacja pętli
        XOR A          ; inicjacja ilorazu
        LD E,A         ; przygotowanie DE=256*dzielnik
        JR HOP         ; start od środka pętli
NXBIT   SRL D
        RR E           ; podzielenie dzielnika przez 2
HOP     SBC HL,DE      ; odjęcie; przeniesienie = 0
        JR NC,QBIT     ; skok jeśli dzielna większa
        ADD HL,DE      ; restytucja; tu przeniesienie=1
QBIT    RLA            ; dopełnienie bitu ilorazu
        DJNZ NXBIT     ; skok do początku pętli
        CPL            ; poprawienie bitów ilorazu
        RET
```

Charakterystyczną cechą procedury DIV21 jest bezpośrednie wykorzystanie bitu przeniesienia do konstrukcji ilorazu. Wymaga to na końcu odwrócenia bitów ilorazu przez użycie rozkazu CPL. Możliwe jest proste ulepszenia tej procedury tak, by reagowała na przepełnienie wynikające z niewłaściwych argumentów. Zostawimy to jako samodzielne ćwiczenie.

Procedura DIV21 sama nie wystarczy do konwersji liczby dwójkowej na postać dziesiętną, gdyż nie nadaje się do podzielenia liczby większej niż 2559 przez 10. Zastosujemy algorytm (patrz np. pozycja [6]) dzielenia przez 10 dowolnie długiej liczby składającej się z bajtów $u[1]$, $u[2]$,..., $u[n]$. Ilorazem jest liczba $w[1]$, $w[2]$,..., $w[n]$, a resztą r.

$r := 0;$
for $j := 1$ **to** n **do**
begin
$w[j] := (r*256 + u[j])$ **div** 10;
$r := (r*256 + u[j])$ **mod** 10
end;

Dla uproszczenia użyjemy tylko dwubajtowej dzielnej umieszczonej w rejestrze HL. Procedura dokonująca konwersji zawartości HL na postać zewnętrzną zapisze otrzymane w wyniku konwersji cyfry w tablicy TEXT.

```
; Procedura PRINT drukowania liczby zawartej w HL
; Cyfry są umieszczane w tablicy TEXT
PRINT   LD IX,TEXT     ; adres tablicy znaków
PRINTR  LD DE,10       ; dzielnik
        OR A           ; przeniesienie = 0
        SBC HL,DE      ; sprawdzenie czy liczba < 10
        ADD HL,DE      ; restytucja
```

```
        JR C,PRCHAR      ; skok gdy pojedyncza cyfra
        LD C,L           ; chwilowe zapamiętanie
        LD L,H
        LD H,D           ; D = 0. HL := HL div 256
        LD D,E           ; D := E = 10
        CALL DIV21       ; iloraz w A, reszta w L
        LD H,L
        LD L,C           ; HL := 256 * L + poprzednie H
        LD C,A           ; zapamiętanie
        LD D,E           ; odtworzenie D = 10
        CALL DIV21
        PUSH HL          ; zapamiętanie reszty < 10
        LD H,C
        LD L,A           ; HL zawiera iloraz
        CALL PRINTR      ; wywołanie rekursywne
        POP HL           ; odtworzenie reszty
PRCHAR  LD A,L           ; druk znaku
        ADD 48           ; dodanie kodu zera w ASCII
        LD (IX),A        ; umieszczenie w tablicy
        INC IX           ; inkrementacja indeksu
        RET
```

7.3. Złożone procedury arytmetyczne

Zajmiemy się teraz zagadnieniem operacji na liczbach dłuższych niż dwubajtowe.

Poniższa procedura LONGADD dodaje do siebie dwie liczby całkowite zajmujące dowolną liczbę bajtów. Adres pierwszej liczby jest zawarty w rejestrze DE, adres drugiej — w HL. Wynik zostanie przepisany w miejsce zajmowane przez drugą liczbę. Liczby są zapamiętywane od najmniej znaczących bajtów w górę.

```
; Procedura LONGADD dodawania dwóch długich liczb
; Parametry:
; Adres najmniej znaczącego bajtu pierwszego argumentu - w DE
; Adres najmniej znaczącego bajtu drugiego argumentu - w HL
; liczba bajtów zajmowanych przez jedną liczbę - w B
; wynik operacji zajmie w pamięci miejsce drugiej liczby.
LONGADD
        XOR A            ; wyzerowanie przeniesienia
ADDLOP  LD A,(DE)        ; pierwszy argument dodany
        ADC A,(HL)       ; do drugiego z przeniesieniem
        LD (HL),A        ; zapamiętanie wyniku
        INC HL           ; następne bajty
        INC DE
        DJNZ ADDLOP
        RET
```

Podobnie konstruujemy procedurę odejmowania dwóch liczb, zamieniając rozkaz ADC na SBC. Procedury te będą się nadawały dla liczb dodatnich lub ujemnych w dwójkowym kodzie uzupełnieniowym. Gdyby procedury dodawania i odejmowania stanowiły elementy pakietu użytkowego należałoby dopisać do nich sekwencje rozkazów sprawdzających wystąpienie przepełnienia. Dopisując rozkaz DAA po rozkazie dodawania lub odejmowania otrzymamy procedurę wykonującą prawidłowe operacje na liczbach BCD. Liczbami w kodzie BCD nie będziemy się szczegółowo zajmować. Są one wygodne np. w rachunkowości, gdzie wprowadzanie i wyprowadzanie informacji numerycznej odgrywa podstawową rolę i gdzie ważne jest, by nie zgubić żadnej cyfry wyniku. Wypada zaznaczyć, że z reguły są to liczby dodatnie, znak towarzyszy liczbie osobno, np. w dodatkowym bajcie. Jako przykład operacji przydatnej w arytmetyce BCD skonstruujemy fragment programu konwersji liczby dwójkowej mniejszej od 100 na kod BCD. Trzeba obliczyć iloraz i resztę z dzielenia przez 10. Można to zrealizować na wiele sposobów. Posłużmy się dla wprawy operacjami charakterystycznymi dla BCD. Wynikiem będzie suma bardziej znaczącej tetrady pomnożonej przez 16 oraz mniej znaczącej.

```
; Procedura BINTOD przekształcania jednobajtowej liczby
; z postaci binarnej na BCD.
; Parametry: argument znajduje się w (HL)
; Wynik – w akumulatorze
BINTOD
        XOR A           ; wyzerowanie akumulatora
        RLD             ; bardziej znacząca tetrada do A
        ADD A,00        ; przygotowanie wskaźników stanu
        DAA             ; liczba BCD w zakresie 0 : 15
        LD B,04         ; licznik pętli
MUL16   ADD A,A         ; pomnożenie A przez 2
        DAA             ; ... ale w BCD
        DJNZ MUL16      ; koniec pętli mnożenia przez 16
        RLD             ; mniej znacząca tetrada do A
        ADD A,00        ; przygotowanie rejestru F
        DAA
        ADD A,(HL)      ; dodanie częściowego wyniku
        DAA             ; poprawienie
        RET
```

Również w przypadku długich liczb dwójkowych bardzo często spotyka się reprezentację zawierającą osobno znak liczby, a osobno jej moduł. Ma ona oczywiste wady z punktu widzenia wygody wykonywania operacji dodawania i odejmowania. Konieczny jest wtedy dodatkowy bit na znak, co w praktyce sprowadza się do całego bajtu. Dodając liczby do siebie trzeba sprawdzić znaki i ewentualnie wywołać procedurę odejmowania zamiast dodawania. Jeszcze więcej czasu i pamięci wymaga konieczność rzeczywistej zmiany znaku liczby, tj.

odjęcie jej od zera, gdy wynik operacji odejmowania wyjdzie ujemny (przeniesienie, a nie nadmiar!). Wykonywanie w pętli operacji:

```
LD A,00
SBC (HL)
```

można nieco przyspieszyć zamieniając powyższe rozkazy na:

```
SBC A,A
SUB (HL)
```

Rozkaz NEG dla długich liczb na niewiele się przydaje. Dla dwubajtowej liczby umieszczonej w HL prostym rozwiązaniem jest:

```
EX DE,HL
LD HL,0000
SBC HL,DE          ; przy wyzerowanym przeniesieniu
```

Zalet osobnego traktowania znaku liczby jest jednak znacznie więcej niż wad. Po pierwsze, o wiele sprawniej organizuje się konwersję liczb na postać zewnętrzną i odwrotnie. Następnie, procedury mnożenia i dzielenia liczb ze znakiem są bardziej skomplikowane niż procedury mnożenia i dzielenia liczb tylko dodatnich. Dodatkowy bajt na znak może być również użyty do przechowywania informacji o długości liczby — liczby zajmujące więcej niż kilkadziesiąt bajtów mają tylko specjalne zastosowania. W dodatkowym bajcie można jeden bit poświęcić na rozszerzenie zakresu liczby do pełnej potęgi dwójki, co czasami się bardzo przydaje. Można wreszcie w tym dodatkowym bajcie umieścić inne specjalne znaczniki, np. bit oznaczający, że wartość liczby wynosi zero. Jawny rozkaz zmiany znaku liczby w porównaniu z kodem uzupełnieniowym tu nie kosztuje prawie nic.

Korzyść z umieszczania liczb w pamięci począwszy od najmniej znaczących bajtów (to, czy w górę czy w dół, jest oczywiście kwestią konwencji) jest dość oczywista dla operacji dodawania i odejmowania, a także dla mnożenia. Z dzieleniem jest odwrotnie, ale tu można zastosować technikę programowania rekursywnego. Poniższy fragment programu dzieli przez 2 długą liczbę, której adres jest umieszczony w HL, a długość w B.

```
        XOR A
DIV2L   DEC B
        JR Z,FIND2     ; skok, gdy ostatni bajt liczby
        INC HL
        CALL DIV2L
        DEC HL
FIND2   LD A,(HL)
        RRA            ; przesunięcie w prawo wpisuje na
        LD (HL),A      ; pozycję 7 poprzedni bit 0
        RET
```

Czytelnik stwierdzający, że jest to program mało sensowny, gdyż prościej jest zacząć od obliczenia adresu końca liczby, będzie miał tutaj rację. Jednak, jeśli liczba nie zajmuje spójnego bloku pamięci, lecz jest jednokierunkową listą złożoną z segmentów powiązanych adresami, rekursja będzie wręcz podstawowym narzędziem przetwarzania takich struktur.

Metoda mnożenia i dzielenia długich liczb (dodatnich) nie jest koncepcyjnie bardziej skomplikowana niż metoda papieru i ołówka znana ze szkoły. Algorytmy mnożenia i dzielenia zostały bardzo dokładnie opisane np. w pracy [6]. W książce Knutha [6], a także w pracy [1] są również opisane nieklasyczne, szybsze algorytmy mnożenia i dzielenia, które, o dziwo, wydają się być ignorowane przez twórców znanych autorowi implementacji języka Basic na 8-bitowe mikrokomputery.

Typowym przykładem nieklasycznego, szybkiego algorytmu mnożenia jest mnożenie liczb podwójnej długości. Niech B oznacza tu bazę układu pozycyjnego; dla liczb traktowanych jako ciąg jednobajtowych cyfr wynosi ona 256.
Liczba x podwójnej długości ma w bazie B postać

$$x = B \cdot x_1 + x_0$$

gdzie x_1 i x_0 są liczbami o pojedynczej długości, pełniącymi rolę cyfr w tej bazie. Podobnie liczba y ma postać

$$y = B \cdot y_1 + y_0$$

Rozwinięcie iloczynu

$$z = x \cdot y = (B \cdot x_1 + x_0) \cdot (B \cdot y_1 + y_0)$$

zawiera 4 człony, wiąże się więc z koniecznością 4-krotnego wywołania procedury mnożącej bajty. A może prostsza będzie jedna procedura mnożąca bezpośrednio długie ciągi bitów skonstruowana tak jak MULT21 z p. 7.1? Uniknie się wtedy 4-krotnego organizowania pętli. Jednak organizacja pętli zabiera bardzo mało czasu w porównaniu z koniecznością operowania wynikami pośrednimi o zwiększonej długości.

Ulepszenie algorytmu polega na wprowadzeniu pomocniczych zmiennych:

$$u = x_1 - x_0$$
$$v = y_0 - y_1$$

co daje w wyniku

$$z = B \cdot (B+1) \cdot x_1 \cdot y_1 + B \cdot u \cdot v + (B+1) \cdot x_0 \cdot y_0$$

Zmniejsza to liczbę mnożeń z 4 do 3 kosztem zwiększenia liczby dodawań, które wykonuje się o wiele szybciej. Posługując się tą techniką rekursywnie, można zmniejszyć liczbę mnożeń z 16 do 9 w przypadku liczb 4-krotnej długości. (W dodatku B zamieszczono realizację tej procedury dla liczb czterobajtowych.)

Opisany poniżej algorytm szybkiego dzielenia, a właściwie obliczania odwrotności liczby również nie jest zbyt popularny, mimo że w niektórych kalkulatorach był realizowany układowo. Opiera się on na eleganckim wykorzystaniu znanego schematu iteracyjnego Newtona. Przypominamy zasadę tego schematu. Należy znaleźć wartość x_0 będącą rozwiązaniem równania $f(x)=0$. Zakładając, że początkowe przybliżenie x jest bliskie x_0, można $f(x_0)$ rozwinąć wokół x z dokładnością do członów liniowych i odwikłać x, otrzymując następne przybliżenie iteracyjne:

$$x := x - f(x)/f'(x)$$

W naszym przypadku chcemy obliczyć $x=1/y$, tj. $1/(x \cdot y)-1=0$. Schemat iteracyjny ma postać:

$$x := x \cdot (2 - x \cdot y)$$

Procedura ta jest zbieżna kwadratowo, tj. liczba znaczących bitów podwaja się przy każdej iteracji, jednak wymaga pewnego przeskalowania liczb, gdyż odwrotność liczby całkowitej jest oczywiście ułamkiem. W praktyce, dla n-bitowej liczby x zamiast bezpośrednio jej odwrotności oblicza się $2^{(2n+1)}/x$, co w wyniku daje liczbę n-bitową (lub $(n+1)$-bitową, gdy x jest potęgą dwójki). Istotne jest, że w podanym schemacie obliczeniowym wcale nie potrzeba od razu stosować mnożenia liczb pełnej długości, gdyż na początku iteracji i tak tylko pierwsze bity są istotne. Największe zastosowanie ma ta metoda w arytmetyce zmiennopozycyjnej, gdzie skalowanie liczb jest automatycznie wbudowane w system. W dodatku B przedstawiono realizację powyższego algorytmu dla liczb czterobajtowych.

Programując na poziomie asemblera można, a czasami należy, unikać stosowania tzw. oczywistych rozwiązań wyższego poziomu, które mogą być bardzo nieefektywne. Wspomnieliśmy już, że dla niewielkich liczb najszybszą metodą dzielenia przez 10 może być wzięcie przybliżenia 13/128 lub 51/512. W przypadku arytmetyki BCD czasami trzeba dzielić przez 10 jednobajtową liczbę powstałą z mnożenia dwóch cyfr dziesiętnych. Tu najefektywniejsza jest po prostu zwykła pętla odejmująca. Nie tylko mamy gwarancję, że liczba przebiegów pętli nie przekracza 9, ale jeśli mnożone cyfry były przypadkowe, średnia liczba potrzebnych operacji odejmowania wynosi ok. 3. Innym klasycznym przykładem jest znany algorytm Euklidesa obliczający największy wspólny podzielnik (ang. *greatest common denominator* — GCD) dwóch liczb całkowitych. Algorytm ten najprościej opisać wyrażeniem

GCD(*x*,*y*) :=
  if *y* = 0 **then** *x* **else** *GCD*(*y*,*x* **mod** *y*);

co wymaga obliczania reszty z dzielenia. Można go sformułować i bez dzielenia

GCD(*x*,*y*) :=
  if *y* = 0 **then** *x* **else** *GCD*(*min*(*x*,*y*),*abs*(*x* − *y*));

ale iteracji będzie teraz znacznie więcej i może się to nie opłacić. Dysponując jednak prostymi i szybkimi operacjami na poziomie asemblera, jakimi są operacje dzielenia i mnożenia przez potęgę dwójki, można zrealizować tzw. *algorytm binarny* obliczający największy wspólny podzielnik. Idea polega na dzieleniu obu liczb przez dwa póki są one parzyste, co daje wspólny czynnik będący potęgą dwójki. Następnie, jeśli jedna z liczb jest nadal parzysta, należy ją dzielić przez dwa do skutku. Gdy już obie są nieparzyste, można mniejszą odjąć od większej, co da wynik dodatni i parzysty. Wynik ten znowu redukujemy itd. Poniższa procedura jest realizacją tej metody.

```
; Procedura BINGCD obliczania największego wspólnego
; podzielnika dwóch dodatnich dwubajtowych liczb całkowitych
; Argumenty: w rejestrach HL i DE.
; Wynik: w rejestrze HL
BINGCD
        LD B,01          ; początkowa potęga dwójki plus 1
RED     BIT 0,L
        JR NZ,CHKDE      ; skok gdy HL nieparzyste, DE : ??
        BIT 0,E
        JR NZ,HLRED      ; skok gdy HL parzyste, DE nie
        SRL H
        RR L             ; HL := HL / 2
        SRL D
        RR D             ; DE := DE / 2
        INC B            ; zwiększenie potęgi dwójki
        JR RED           ;i powrót
HLRED   EX DE,HL
DERED   SRL D            ; tu DE jest parzyste
        RR E             ; DE := DE / 2
CHKDE   BIT 0,E
        JR Z,DERED       ; pętla jeśli nadal parzyste
        SBC HL,DE        ; odjęcie; wynik jest parzysty
        JR Z,EXIT        ; koniec iteracji
        JR NC,HLRED      ; redukcja wyniku odjęcia
        ADD HL,DE        ; wynik ujemny, trzeba poprawić
        EX DE,HL
        CCF              ; wyzerowanie przeniesienia
        SBC HL,DE        ; teraz już będzie dodatni
        JR HLRED         ; i kolejna redukcja
EXIT    EX DE,HL         ; wynik do HL
LOOP    DEC B
        RET Z            ; koniec mnożenia przez potęgę 2
        ADD HL,HL        ; następny czynnik 2
        JR LOOP
```

Argumenty procedury BINGCD muszą mieć wartości różne od zera.

Jako samodzielny problem badawczy pozostawimy Czytelnikowi napisanie procedury szybszej, a równie zwartej. (W przypadku sukcesu proszę natychmiast zawiadomić autora.)

Zagadnieniem zasługującym na osobny komentarz jest organizacja danych w pamięci. Każdy użytkowy pakiet podprogramów numerycznych powinien zapewnić dogodny sposób przekazywania liczb procedurom. W przypadku liczb długich, zwłaszcza o zmiennej długości, może to być niebagatelny problem wymagający zastosowania procedur dynamicznego przydziału i odzyskiwania pamięci. Jednak nawet przy ustalonej długości, np. przy wybranym raz na zawsze standardzie zapisu liczb zmiennopozycyjnych, musimy wykonać czarną robotę, którą zrobiłby za nas kompilator języka wyższego poziomu, a mianowicie zorganizować pamięć roboczą na wyniki pośrednie. Programista przystępujący do pracy nad oprogramowaniem zawierającym dużo obliczeń arytmetycznych powinien pamiętać, że wąskim gardłem systemu może się okazać nie powolność procedury mnożącej, lecz transmisja danych między różnymi polami pamięci. Dobrze jest wtedy dysponować niedużą tablicą o ustalonym adresie do przechowywania stałych lub częściej używanych zmiennych. Jednakże podstawową strukturą danych, bez której bardziej złożone problemy obliczeniowe stają się koszmarem, jest stos, na którym będą przechowywane argumenty i wyniki częściowe złożonej sekwencji operacji. (Nie musi mieć on nic wspólnego z rejestrem SP.)

Zakończenie rozdziału zawiera kilka komentarzy poświęconych arytmetyce zmiennopozycyjnej. Zagadnienie jest zbyt obszerne, aby poświęcić mu tutaj dużo miejsca. Jest bardzo dobrze omówione w książce [6]; patrz także [4]. Kilka algorytmów zrealizowanych dla mikroprocesora Intel 8080 zawiera pozycja [5].

Jak wiadomo, klasyczną reprezentacją liczb zmiennopozycyjnych jest para liczb zwanych powszechnie cechą (lub wykładnikiem) c i mantysą m. Wartość liczby x jest określona wzorem:

$$x = m \cdot 2^c$$

Cecha jest liczbą całkowitą z przedziału określonego swoją długością i wyznacza zakres liczby zmiennopozycyjnej. Z praktycznych względów najczęściej spotykaną długością cechy w systemach opartych na procesorze 8-bitowym jest jeden bajt. Daje to c zawarte między -128 a $+127$ i zakres liczb dodatnich zawartych w przybliżeniu między 10^{-38} a 10^{+38}. Nie zawsze jest to wystarczające. Oczywiście bardzo rzadko pojawia się konieczność operowania liczbami z obu końców zakresu, zwłaszcza jednocześnie, gdy nic nie można zrobić odpowiednim przeskalowaniem. Takie problemy można spotkać we współczesnej teorii grawitacji, ale nie jest to typowe wykorzystanie procesora 8-bitowego. Niestety w bardziej typowych zastosowaniach mamy czasami do czynienia z iloczynem lub ilorazem liczb różnej wielkości i przy odrobinie pecha, z którym każdy

programujący musi się liczyć, wynik może dość silnie zależeć od kolejności wykonywanych działań: o jedno mnożenie za dużo i dostajemy niedomiar w postaci zera, albo utratę wielu cyfr znaczących, względnie nadmiar i błąd.

16-bitowa cecha z kolei wydaje się przesadą i marnowaniem pamięci. Pewnym rozwiązaniem łatwo realizowalnym na poziomie asemblera jest rezygnacja z podstawy cechy równej 2. (Jest to czasami spotykane w dużych komputerach, np. nie każdy wie, że IBM 370 jest pod tym względem maszyną o podstawie 16.)

Przypominamy, że przy podstawie równej 2 mantysę traktuje się jako ułamek dwójkowy — liczbę stałopozycyjną o położeniu kropki binarnej tuż na lewo od najbardziej znaczącego bitu *m*. Znormalizowana mantysa jest większa lub równa 1/2 i mniejsza od 1. Oznacza to, że najbardziej znaczący bit mantysy jest zawsze równy 1. Pozwala to wykorzystać ten bit do innych celów. Typowym wykorzystaniem jest przeznaczenie go na bit znaku.

Naturalnymi kandydatami na nietradycyjną podstawę są: 16 ze względu na istnienie operacji RLD i RRD przesuwających tetrady, oraz 256. W tym ostatnim przypadku, jeśli na cechę wykorzystamy tylko 7 bitów, a 8 bit przeznaczymy na znak, największą potęgę 10 mieszczącą się w zakresie przesuniemy do ok. 152, co wydaje się już w pełni wystarczające. To rozwiązanie zostało wykorzystane w niektórych mikrokomputerach (np. New Brain), ale nie zdobyło większej popularności. Częściowo wynika to z konserwatywności twórców oprogramowania komercyjnego. Istniejące rozwiązania po prostu się przepisuje, jak tego dowodzi np. drobny błąd w dzieleniu małych liczb ujemnych w wersji Basicu Microsoft, zauważony na wielu komputerach pracujących z różnymi procesorami. Są jednak i inne powody nieużywania podstawy większej niż 2, np. 256. Wykorzystanie bitów mantysy jest wtedy gorsze: o znormalizowanej mantysie wiadomo tyle, że jej najbardziej znaczący bajt jest różny od zera, tj. jest to liczba nie mniejsza niż 1/256. Przy dodawaniu, jak wiadomo, pierwszą operacją jest wyrównanie wykładników z odpowiednim przesunięciem (przeskalowaniem) jednej z mantys. Przy podstawie 256 to przesunięcie jest znacznie szybsze, gdyż przesuwamy całe bajty, ale traci się też od razu całe 8 bitów dokładności. Oczywiście można to nadrobić wprowadzając dodatkowy bajt mantysy, lecz to z kolei obciąża pamięć i wydłuża czas liczenia. Wydaje się jednak, że podstawa 256 i czterobajtowa mantysa przy równoczesnym zastosowaniu szybkich procedur mnożących i dzielących góruje szybkością nad najczęściej stosowanym i zaakceptowanym przez IEEE jako nieoficjalny standard rozwiązaniem z trójbajtową mantysą i jednobajtową cechą o podstawie 2.

Przy podstawie 256 konieczność przesuwania mantysy wystąpi rzadziej niż dla podstawy 2, tym bardziej że w typowych rachunkach na ogół dodaje się liczby o zbliżonych rzędach wielkości. Tu nasuwa się uwaga pod adresem autorów dobrej i pożytecznej książki [5], którzy na str. 227 zauważają, że jeśli różnica cech jest odpowiednio duża, przesuwanie jednej z mantys zepchnie ją do

zera i dodawanie liczb jest operacją pustą. Narzucającym się wnioskiem jest możliwość optymalizacji polegającej na przerwaniu dodawania przed przesuwaniem mantysy. Wniosek słuszny, ale ta optymalizacja przyniesie częściej szkodę niż pożytek. Uniknie się niepotrzebnej pracy w znikomym procencie przypadków, ale odpowiedni test będzie zabierał czas zawsze. Jest to tak klasyczny przykład, że autor radzi zawsze o nim pamiętać w momencie przystępowania do ulepszania programu.

Czternaste przykazanie programisty brzmi:

Nie naprawiaj, czego nie zepsułeś.

Samodzielnym ćwiczeniem niech będzie odkrycie pozostałych 13.

8. Struktury danych i ich przetwarzanie

8.1. Tablice jednowymiarowe

Programując w języku asemblera stosunkowo rzadko mamy do czynienia ze strukturami danych o dużym stopniu złożoności strukturalnej, ze względu na brak dogodnych mechanizmów opisu danych i struktur sterowania. Natomiast *tablice* traktowane jako spójne, jednolicie adresowane bloki pamięci wystąpią w większości niebanalnych programów. Niestety, nawet tak proste operacje jak przeszukiwanie czy przesyłanie bloków mogą być kłopotliwe do zaprogramowania i wbudowane rozkazy LDIR czy CPIR nie rozwiązują całości problemu. Jako pierwszy przykład operacji bardziej złożonej niż zwykłe przemieszczenie bloku napiszmy procedurę odwracającą kolejność elementów tablicy bajtów.

```
; Procedura INVERT odwracania tablicy
; HL - adres tablicy minus 1
; DE - długość tablicy w bajtach
; procedura nie niszczy rejestru BC
INVERT
        EX DE,HL
        ADD HL,DE       ; HL - adres ostatniego elementu
INVLOP  INC DE          ; DE - adres pierwszego elementu
        SBC HL,DE       ; czy ostatni adres >= pierwszy?
        RET C           ; koniec, gdy przekroczona połowa
        ADD HL,DE       ; restytucja HL
        LD A,(HL)       ; pobranie ostatniego elementu
        EX AF,AF'
        LD A,(DE)       ; pobranie pierwszego elementu
        LD (HL),A       ; przesłanie na miejsce ostatniego
        EX AF,AF'
        LD (DE),A       ; na miejsce pierwszego
        DEC HL          ; redukcja tablicy od góry
        JR INVLOP
```

Na uwagę zasługuje zignorowanie przypadku nieparzystej długości tablicy: środkowy element zostanie pobrany i wpisany ponownie w to samo miejsce. Okazuje się, że dołączenie do procedury testu wykonywanego podczas każdego przebiegu pętli jest zupełnie nieopłacalne. Sytuacja może być inna, jeśli elementami tablicy nie są pojedyncze bajty, lecz dłuższe segmenty. Wtedy program od razu zaczyna się komplikować.

Drobną wprawką będzie przeformułowanie tej procedury tak, by startowała od HL zawierającego adres tablicy, a nie adres o 1 mniejszy, i była tak samo efektywna czasowo i pamięciowo. (A może lepsza?) Czy odejmowanie górnego adresu od dolnego — zamiast odwrotnie — rozwiąże wspomniany wyżej problem elementu środkowego?

Z pomocą procedury INVERT skonstruujemy teraz pouczającą i sprawną procedurę przesuwającą cyklicznie elementy tablicy (bajty lub dłuższe bloki — jest to w tym wypadku bez znaczenia), bez potrzeby używania dodatkowej pamięci.

Aby N-elementową tablicę obrócić o M elementów w dół, tj. aby pierwsze M elementów znalazło się na końcu tablicy, należy wykonać następującą sekwencję operacji:

1. Odwrócić kolejność pierwszych M elementów tablicy.
2. Odwrócić całą tablicę.
3. Odwrócić kolejność pierwszych $N-M$ elementów tablicy.

Niech dla ilustracji $N = 8$, a $M = 3$. Wtedy mamy:

początkowo:	1	2	3	4	5	6	7	8
po kroku 1:	3	2	1	4	5	6	7	8
po kroku 2:	8	7	6	5	4	1	2	3
po kroku 3:	4	5	6	7	8	1	2	3

Aby tablicę przesunąć cyklicznie w górę, należy w miejsce M wziąć $N-M$ lub zmienić porządek operacji. Drugi sposób i jego warianty pozostawiamy jako samodzielne ćwiczenie.

```
; Procedura ROTABL przesuwania cyklicznego elementów
; (bajtów) tablicy.
; HL - adres tablicy minus 1
; BC - całkowita długość tablicy
; DE - liczba elementów, o którą należy dokonać obrotu
;        (długość pierwszego segmentu tablicy)
ROTABL
        PUSH DE        ; DE - długość pierwszego segmentu
        PUSH HL        ; HL - adres tablicy minus 1
        CALL INVERT    ; odwrócenie pierwszego segmentu
        POP HL
        PUSH HL        ; odtworzenie adresu tablicy minus 1
        PUSH BC
```

```
POP DE         ; DE - długość tablicy
CALL INVERT    ; odwrócenie całości
POP DE         ; DE - adres tablicy minus 1
PUSH BC
POP HL         ; HL - długość tablicy
POP BC         ; BC - długość pierwszego segmentu
CCF            ; wyzerowanie przeniesienia
SBC HL,BC      ; HL - długość drugiego segmentu
EX DE,HL       ;   w DE. HL - adres tablicy - 1
JP INVERT      ; ostatnie wywołanie i powrót
```

Przypominamy, że procedura ROTABL nie wymaga dodatkowej pamięci. Nieefektywność wynikająca z faktu 2-krotnego przesyłania każdego elementu okazuje się mniejsza niż spowodowana koniecznością skomplikowanego obliczania adresów w przypadku innej, bardziej bezpośredniej techniki. Natomiast powyższa procedura zawiera pewne niezbyt czytelne elementy: „żonglowanie" zawartością stosu. Tutaj prostsze będzie zapamiętanie rejestrów w bezpośrednio adresowanych komórkach pamięci, nawet jeśli jest to nieco mniej efektywne czasowo i pamięciowo.

Drugim przykładem złożonej operacji jest wyszukiwanie słów w tekście, tj. sprawdzenie, czy i gdzie w tablicy znajduje się określony ciąg bajtów. Jest to problem o dużym znaczeniu zarówno praktycznym, jak i teoretycznym (patrz np. pozycja [1]). Procedura przedstawiona poniżej jest efektywnym rozwiązaniem dla słów niezbyt długich lub takich, które nie zawierają dłuższych powtarzających się podsłów.

```
; Procedura SEARCH szukania w tablicy ciągu bajtów.
; Parametry wejściowe:
; TEXT - słowo mieszczące adres tablicy
; WORD - słowo mieszczące adres szukanego wzorca
; TXTLEN - słowo mieszczące długość tablicy
; WRDLEN - słowo mieszczące długość wzorca
; Parametr wyjściowy:   HL
; Jeśli wskaźnik Z jest ustawiony, HL zawiera adres wewnątrz
;  tablicy, od którego rozpoczyna się szukany wzorzec.
;  Jeśli wskaźnik Z jest wyzerowany poszukiwanie zakończyło
;  się niepowodzeniem.
SEARCH
      LD HL,(WRDLEN) ; długość wzorca oraz
      LD BC,0001     ;   liczba 1
      EXX            ; będą w alternatywnym banku
      LD HL,(TEXT)   ; adres tablicy
      LD BC,(TXTLEN) ; długość tablicy
```

```
RSTRT  LD DE,(WORD)   ; adres początku wzorca
       LD A,(DE)      ; pobranie pierwszego znaku
       CPIR           ; szukanie pierwszego znaku
       RET NZ         ; powrót gdy niepowodzenie
       PUSH HL        ; na stosie: adres drugiego znaku
NXTC   EXX
       SBC HL,BC      ; gdy był to ostatni znak wzorca
       EXX            ;   HL' się wyzerowało
       JR Z,FOUND     ; sygnalizacja sukcesu
       INC DE
       LD A,(DE)      ; następny znak wzorca
       CPI            ; porównuje, HL:=HL+1, BC:=BC-1
       JR Z,NXTC      ; jeśli sukces: następny znak
       POP HL
       RET PO         ; koniec tablicy: niepowodzenie
       JR RSTRT
FOUND  POP HL         ; adres pierwszego znaku + 1
       DEC HL
       RET
```

Zupełnie odmienny sposób postępowania należy przyjąć, jeśli przeszukiwany tekst nie jest statyczną tablicą, lecz jest przetwarzanym przez procesor strumieniem znaków o nieokreślonej długości, np. czytanym z urządzenia wejściowego, i przeszukiwanie należy przerwać po napotkaniu wzorca lub specjalnego znaku końca. Zostawimy ten problem jako bardzo pouczające ćwiczenie.

Końcowym przykładem w tym rozdziale będzie konstrukcja struktury zwanej *stogiem* (ang. *heap*, po polsku nazywanej również kopcem lub stertą — w zależności od rolniczych upodobań piszącego). Jest to struktura, która ma organizację jednowymiarowej tablicy, jednak semantycznie odpowiada *drzewu binarnemu*. Stóg służy do konstrukcji bardzo efektywnej procedury sortującej tablicę, znanej jako *heapsort* (patrz np. [6], t. III). Ponadto — co może mieć spore znaczenie w niektórych systemach mikroprocesorowych — służy on do zarządzania tzw. *kolejką priorytetów* zawierającą deskryptory zgłoszeń procesów (np. wejścia/wyjścia), które należy obsłużyć. Zgłoszenie o najwyższym priorytecie w kolejce jest obsługiwane najwcześniej i usuwane z kolejki. Kolejne przychodzące zgłoszenia mogą mieć różne priorytety.

Programista może zastosować jedną z dwóch narzucających się prymitywnych metod bezpośrednich. Pierwsza polega na umieszczaniu nowego zgłoszenia na końcu kolejki i przeszukiwaniu jej, aby znaleźć element o najwyższym priorytecie. Czas przeszukiwania będzie proporcjonalny do długości kolejki. Drugim rozwiązaniem jest utrzymywanie kolejki stale posortowanej: nowy element będzie umieszczany na odpowiedniej pozycji. Wtedy pobieranie

zgłoszenia do obsługi jest natychmiastowe, ale czas wprowadzania nowego zgłoszenia będzie proporcjonalny do długości kolejki. Należy wtedy przeglądać kolejkę tak długo, aż relacja między kolejnym elementem a nowo wprowadzanym zmieni się na przeciwną. Jednocześnie należy elementy przesuwać o jedną pozycję, aby wytworzyć „dziurę" na nowy element, jeśli kolejka jest zwartą strukturą typu tablicy, a nie np. listą wiązaną adresami.

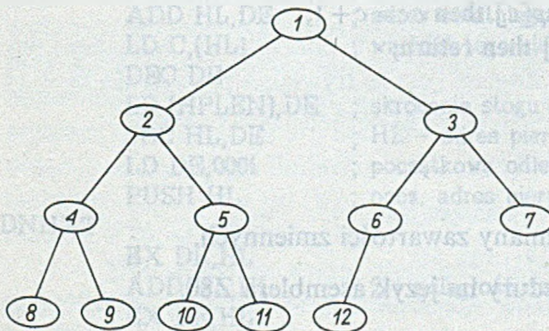
Jeśli system ma obsłużyć dużą liczbę N zgłoszeń, całkowity czas operacji w obu przypadkach będzie proporcjonalny do kwadratu N. Użycie stogu umożliwia osiągnąć czas proporcjonalny do logarytmu długości kolejki zarówno przy wprowadzaniu, jak i usuwaniu elementu, co da całkowity czas operacji proporcjonalny do $N \cdot \log N$.

Dla uproszczenia będziemy rozważać stóg złożony z jednobajtowych liczb, a relacją porządkującą je będzie zwykła mniejszość. Stóg jest zdefiniowany jako drzewo binarne o następujących właściwościach:

1. Wartość elementu leżącego w węźle drzewa jest mniejsza (lub równa) od wartości elementów w węzłach potomnych.

2. Drzewo jest specyficznie zrównoważone: wszystkie poziomy węzłów licząc od korzenia są wypełnione, z wyjątkiem ostatniego poziomu, w którym może z jednego końca brakować elementów.

Rysunek 8.1 przedstawia strukturę stogu o 12 elementach. Liczby w węzłach oznaczają kolejność wypełniania drzewa, tj. stóg o 5 elementach ma strukturę odpowiedniego poddrzewa. Jednocześnie ten rysunek wskazuje sposób realizacji stogu za pomocą jednowymiarowej tablicy. Liczby oznaczają po prostu kolejne indeksy elementów tablicy numerowane od 1. Dla węzła o numerze (indeksie) I węzły potomne mają numery $2 \cdot I$ oraz $2 \cdot I + 1$.



Rys. 8.1. Struktura stogu o 12 elementach

Napiszemy dwie procedury operujące stogiem. Pierwsza o nazwie UPH dołączy N-ty element do stogu o $N-1$ elementach. Stóg będzie przeglądany od końca, tj. od liści w kierunku korzenia, z przesuwaniem elementów w dół i umieszczeniem nowego elementu na odpowiedniej pozycji, podobnie jak w liniowej kolejce. Teraz jednak długość listy do przejrzenia jest proporcjonalna do

wysokości drzewa, tj. do log N, a nie do długości tablicy. Druga procedura o nazwie DOWNH, będzie służyła do usuwania elementu ze stogu. Elementem usuwanym będzie zawsze korzeń drzewa, ale należy wtedy odtworzyć strukturę stogową za pomocą następującej metody. Ostatni element stogu umieszcza się na pierwszej pozycji, a następnie przesuwa się w głąb drzewa, aż zajmie właściwą pozycję, przesuwając kolejne elementy stogu w górę. Przy przesuwaniu w dół nowego elementu wybiera się zawsze tę gałąź potomną, której wierzchołek ma mniejszą wartość, co gwarantuje, że po zakończeniu operacji będziemy mieli prawidłowy stóg. Procedury te są następujące:

```
procedure UPH(N);
 begin N := N+1;
   i := N;
     while i > 1 do
       begin p := i div 2;
         if heap[p] < = heap[i] then return;
         heap[p] :=: heap[i];
         i := p
       end
 end;
procedure DOWNH(N);
 begin heap[1] := heap[N]; N := N-1;
     i := 1;
     while (c := 2*i) < = N do
       begin
         if c < N then
           if heap[c+1] < heap[c] then c := c+1;
         if heap[i] < = heap[c] then return;
         heap[i] :=: heap[c];
         i := c
     end
end;
```

Symbol :=: oznacza operację wymiany zawartości zmiennych.

Przetłumaczymy te procedury na język asemblera Z80.

```
; Procedury UPH i DOWNH operujące stogiem.
; Parametry wejściowe:
; HPAD1 - słowo mieszczące adres o 1 mniejszy od pocz. stogu,
; HPLEN - słowo mieszczące aktualną długość stogu
; Rejestr A - wartość nowo wprowadzanego elementu w UPH
```

```
UPH
        LD DE,(HPLEN)
        INC DE
        LD (HPLEN),DE   ; zwiększenie długości stogu o 1
        LD HL,(HPAD1)
        ADD HL,DE
        LD (HL),A       ; wpisanie A do ostatniego elementu
        LD C,A          ; zapamiętanie jego wartości w C
UPNXT   PUSH HL         ; na stosie adres elementu
        SRA D           ; dzielenie zawartości DE przez 2
        JR NZ,NZERO     ;   ze sprawdzeniem, czy nie jest
        RR E            ;    już równa zeru
        JR Z,UPFIN
        RL E
NZERO   RR E
        SBC HL,DE       ; HL - adres przodka elementu
        LD A,(HL)       ; pobranie
        CP C            ;  i porównanie z elementem
        JR C,UPFIN      ; gdy przodek < element: koniec
        LD (HL),C       ; element na miejsce przodka
        EX (SP),HL      ; na stosie adres przodka
        LD (HL),A       ; przodek na miejsce elementu
        POP HL          ; adres przodka ze stosu
        JR UPNXT        ; powtórzenie pętli o poziom wyżej
UPFIN   POP HL          ; przywrócenie początkowego stosu
        RET
;------------------------------------------------------------
DOWNH
        LD HL,(HPAD1)   ; adres stogu minus 1
        LD DE,(HPLEN)   ; długość stogu
        ADD HL,DE       ; adres ostatniego elementu stogu
        LD C,(HL)       ; wartość tego elementu
        DEC DE
        LD (HPLEN),DE   ; skrócenie stogu
        SBC HL,DE       ; HL - adres pierwszego elementu
        LD DE,0001      ; początkowa odległość do potomka
        PUSH HL         ; pocz. adres pierwszego elementu
DNNXT
        EX DE,HL
        ADD HL,HL       ; 2 * odległość do potomka
        EX DE,HL
        LD HL,(HPLEN)   ; całkowita długość stogu
        SBC HL,DE
        JR C,DNFIN      ; jeśli za daleko - koniec
        LD HL,(HPAD1)
        ADD HL,DE       ; adres potomka; nie zmienia wsk. Z!
        LD A,(HL)       ; pobranie wartości lewego potomka
        JR Z,DNCMP      ; jeśli to ostatni element stogu
```

```
        INC HL          ; adres prawego potomka
        INC DE
        CP (HL)         ; porównanie lewy - prawy
        JR NC,RIGHT     ; skok gdy prawy mniejszy
        DEC HL          ; z powrotem lewy
        DEC DE
RIGHT   LD A,(HL)       ; pobranie wartości potomka
DNCMP   CP C            ; porównanie potomek - element
        JR NC,DNFIN     ; jeśli potomek większy: koniec
        LD (HL),C       ; element na miejsce potomka
        EX (SP),HL      ; adres elementu ze stosu
        LD (HL),A       ; potomek na miejsce elementu
        JR DNNXT        ; na stosie: adres potomka
DNFIN   POP HL
        RET
```

Proponujemy następujące samodzielne ćwiczenia:

Przepisać dwie powyższe procedury tak, by operowały tablicą dwubajtowych słów. Napisać alternatywną do DOWNH procedurę usuwającą element ze stogu w następujący sposób: usunięty pierwszy element nie będzie zastępowany ostatnim; stóg będzie się przesuwał w górę tak, że dziurę wypełni potomek o mniejszej wartości i dopiero miejsce zwolnione na końcu zajmie ostatnie element stogu. Porównać efektywnoścć obu sposobów postępowania.

Jak wspomnieliśmy, procedury zarządzające stogiem służą jako podstawa efektywnej i szeroko stosowanej metody sortowania. Jako przykład weźmy 10-elementową tablicę zawierającą początkowo liczby:

5, 14, 2, 3, 10, 8, 17, 4, 14 oraz 0.

Zakładając teraz, że dla kolejnych $m = 1,2,3,...,9$ początkowy m-elementowy segment tablicy jest prawidłowym stogiem (jest to trywialna prawda dla $m = 1$), poniższa pętla dołączy do stogu po kolei wszystkie elementy tablicy

for m := 1 **to** 9 **do** *UPH*(m);

Kolejność elementów tablicy będzie następująca:

0, 2, 5, 4, 3, 8, 17, 14, 14 oraz 10

co jak łatwo sprawdzić odpowiada kolejności w stogu. Aby ułożyć (np. wydrukować) elementy od najmniejszego do największego, wystarczy wykonać pętlę

```
for m := 10 downto 1 do begin
                          writeln(heap[1]);
                          DOWNH(m)
                        end;
```

8.2. Tablice wielowymiarowe i struktury listowe

Charakterystyczna dla wielowymiarowych tablic konieczność odwzorowania ciągu kilku indeksów na pojedynczy adres jest dość kłopotliwa na poziomie asemblera. Rozważmy d-wymiarową tablicę o indeksach liczonych od zera:

$$A[0\,..\,n(1)-1, 0\,..\,n(2)-1, ..., 0\,..\,n(d)-1]$$

o całkowitej liczbie jednobajtowych elementów równej iloczynowi

$$n(1)\cdot n(2)\cdot ...\cdot n(d)$$

Jeśli adres tej tablicy oznaczymy przez A, to adres elementu $A[i(1), i(2), ..., i(d)]$ będzie dany wyrażeniem:

$$A + i(1) + n(1)\cdot i(2) + n(2)\cdot i(3) + ... + n(1)\cdot n(2)\cdot ...\cdot n(d-1)\cdot i(d)$$

Konstrukcja tego wyrażenia opiera się na konwencji, że przy przebieganiu kolejnych elementów najczęściej zmienia się ostatni wskaźnik, potem przedostatni itd., i że tablica zajmuje spójny obszar pamięci począwszy od adresu A. Adres ten można obliczyć za pomocą następującej pętli:

```
adrrel := i1;
for k := 2 to d do adrrel := adrrel* n(k)+i(k);
adr := A+adrrel;
```

Stosunkowa powolność procedury adresowania elementów tablic wielowymiarowych wskutek konieczności wykonywania mnożenia powoduje, że mimo niewygody czasami trzeba podprogram operujący tablicami napisać w języku asemblera. Wiąże się to z tym, że dysponując kompilatorem języka wyższego poziomu o niezbyt wyrafinowanych możliwościach optymalizacyjnych przy każdym wyrażeniu indeksowanym będzie generowany kod wykonujący odpowiedni iloczyn. Często jednak sposób operowania indeksami jest na tyle regularny, że obejdzie się bez wywoływania podprogramów mnożenia, zastępując je sekwencjami dodawania. Poniższy przykład obrazuje mnożenie dwóch macierzy: $A[0\,..\,n1-1, 0\,..\,n2-1]$ oraz $B[0\,..\,n2-1, 0\,..\,n3-1]$ dające w wyniku macierz $C[0..n1-1, 0..n3-1]$.

Dla ustalenia uwagi elementy macierzy będą dwubajtowe. Zostanie użyty klasyczny schemat obliczeniowy

```
for i:=0 to n1-1 do
for j:=0 to n3-1 do
  begin C[i,j]:=0;
        for k:=0 to n2-1 do
              C[i,j]:=C[i,j]+A[i,k]*B[k,j]
  end;
```

W języku asemblera Z80 procedura mnożenia dwóch macierzy ma postać:

```
; Procedura MATMULT mnożenia dwóch macierzy o elementach
; dwubajtowych.
; Parametry:
; AMAT - słowo zawierające adres macierzy A
; BMAT - słowo zawierające adreS macierzy B
; CMAT - słowo zawierające adres macierzy wynikowej C
; N1 - słowo zawierające dwubajtową liczbę n1 (liczba wierszy
;  macierzy A)
; N2 - słowo zawierające dwubajtową liczbę n2 (liczba kolumn
;  macierzy A, lub liczba wierszy macierzy B)
; N3 - słowo zawierające dwubajtową liczbę n3 (liczba kolumn
;  macierzy B, lub liczba kolumn macierzy C)
;
;  Procedura wykorzystuje osobny, nie zamieszczony tutaj
; podprogram mnożenia MULT o następujących parametrach:
; adres dwubajtowej mnożnej:     w rejestrze HL,
; adres dwubajtowego mnożnika:   w rejestrze DE
; dwubajtowy wynik:              w rejestrze BC.
; Procedura MULT nie niszczy rejestrów DE,HL ani IX.
;
; Na wstępie: definicja makrorozkazu do obsługi pętli
DJNZ2   MACRO ADR
        POP BC          ; zdjęcie BC ze stosu
        DEC BC          ; dekrementacja
        LD A,C
        OR B
        JR NZ,ADR       ; skok jeśli nadal różne od zera
        ENDM            ; koniec makrodefinicji
;______________________________________________
MATMULT
        EXX
        LD HL,(CMAT)    ; bieżący adres w macierzy wynikowej
        EXX             ; zostanie przechowany w HL'
        LD HL,(N3)
        ADD HL,HL       ; dla efektywności wewnętrznej pętli:
        LD (N32),HL     ;     liczby są dwubajtowe
        LD HL,(AMAT)    ; adres mnożnej
        LD BC,(N1)      ; indeks pierwszej pętli: "i"
ILOOP   PUSH BC
        LD DE,(BMAT)    ; adres mnożnika
        LD BC,(N3)      ; indeks drugiej pętli: "j"
JLOOP   PUSH BC
        PUSH DE
        PUSH HL
        LD IX,0000      ; IX przechowa sumę częściową
        LD BC,(N2)      ; indeks wewnętrznej pętli: "k"
```

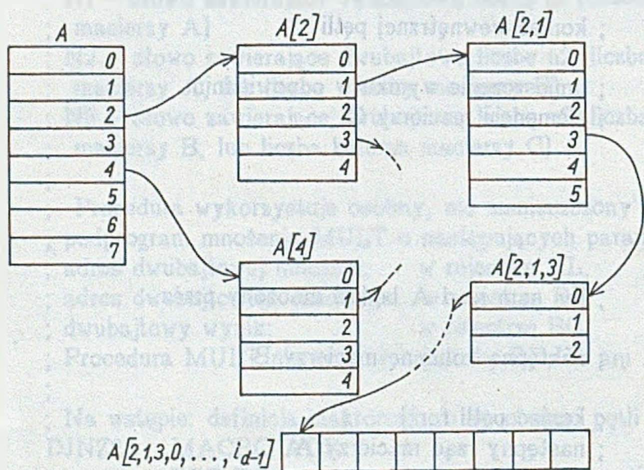
```
KLOOP PUSH BC
      CALL MULT       ; wywołanie procedury mnożenia
      ADD IX,BC       ; dodanie wyniku do sumy częściowej
      INC HL
      INC HL          ; następny element w rzędzie A
      EX DE,HL
      LD BC,(N32)     ; długość rzędu w bajtach
      ADD HL,BC       ; następny element w kolumnie B
      EX DE,HL
      DJNZ2 KLOOP     ; koniec wewnętrznej pętli
      EXX
      PUSH IX         ; umieszczenie wyniku w odpowiednim
      POP DE          ; elemencie macierzy C
      LD (HL),E
      INC HL
      LD (HL),D
      INC HL
      EXX
      POP HL          ; ten sam rząd A będzie mnożony przez
      POP DE
      INC DE          ; następną kolumnę macierzy B
      INC DE
      DJNZ2 JLOOP     ; koniec pętli for j
      INC HL          ; następny rząd macierzy A
      INC HL
      DJNZ2 ILOOP     ; koniec pętli for i
      RET
```

Jak widać, wartości indeksów pętli nigdy nie zostały użyte do konstrukcji adresów. Można było dekrementować te indeksy do zera zamiast inkrementować do odpowiednich granic, co przyspieszyło operację sprawdzania warunków kończących pętle.

Umieszczanie w pamięci pod dynamicznie obliczanymi adresami dwubajtowych słów nie jest wygodne, trzeba to robić na raty. Jeśli jednak tablica jest wypełniona kolejno, w dół przestrzeni adresowej, pożytecznym chwytem w niektórych sytuacjach może być niestandardowe użycie rejestru stosu i przesyłanie elementów przez kolejne PUSH nie stowarzyszone z POP, a na końcu odtworzenie pierwotnej wartości rejestru SP. Podobnie można zorganizować przeglądanie tablicy w górę adresów. Użycie stosu do innych celów jest jednak wtedy mocno ograniczone, a jeśli system dopuszcza przerwania, dane mogą ulec uszkodzeniu.

Drugą standardową metodą indeksowania elementów tablicy wielowymiarowej jest, jak wiadomo, wykorzystanie struktury listowej i indeksowanie przez użycie pośredniego adresowania. Tablica d-wymiarowa jest traktowana

jako jednowymiarowa, ale złożona z obiektów będących tablicami $(d-1)$-wymiarowymi itd. Każdy taki obiekt jest w tablicy nadrzędnej reprezentowany przez swój adres. Początkowy adres tablicy A nie odnosi się więc bezpośrednio do bloku danych, lecz jest adresem jednowymiarowej tablicy — wektora zawierającego adresy podtablic. Dopiero ostatnie tablice na tej liście zawierają rzeczywiste dane. Strukturę tak zorganizowanej tablicy przedstawiono na rys. 8.2.



Rys. 8.2. Listowa organizacja tablicy wielowymiarowej

Gdy program operuje elementami tablicy w sposób nieregularny, przypadkowy i żadnej mądrzejszej metody niż pełne indeksowanie zastosować się nie da, powyższy schemat postępowania może dać duże oszczędności czasowe w porównaniu z metodą wykorzystującą mnożenie. Ceną, jaką trzeba wtedy zapłacić, jest dodatkowa pamięć przeznaczona na adresy. Jest natomiast dodatkowa korzyść związana z faktem, że dane nie muszą zajmować spójnego bloku pamięci. Tablica nie musi być prostokątna, można np. operować tablicą dwuwymiarową, w której każda kolumna będzie miała inną liczbę wierszy.

Poniższa procedura oblicza adres elementu tablicy o dowolnej liczbie wymiarów w oparciu o tablice wartości indeksów.

```
; Procedura IADR obliczająca adres elementu d-wymiarowej
; tablicy.
; Parametry:
; ATAB - słowo zawierające adres tablicy A, tj. adres jej
;   pierwszej listy adresów podtablic.
; DIM - słowo zawierające liczbę wymiarów, mniejszą od 256.
; LEN - słowo zawierające długość elem. tablicy w bajtach
; ITAB - słowo zawierające adres tablicy kolejnych
;   indeksów: i1, i2, ..., id. Indeksy są jednobajtowe.
```

```
IADR
        LD HL,(ITAB)     ; adres tablicy indeksów
        EXX              ;do HL'
        LD HL,(ATAB)     ; adres tablicy A do HL
        LD BC,(DIM-1)    ; liczba wymiarów
        JR LFIN
ILOOP   EXX
        LD A,(HL)        ; pobranie kolejnego indeksu
        INC HL
        EXX
        LD E,A           ;do rejestru DE
        LD D,00
        EX DE,HL
        ADD HL,HL        ; kolejny numer bajtu w liście
        ADD HL,DE        ; adres elementu listy
        LD A,(HL)
        INC HL
        LD H,(HL)
        LD L,A           ; adres następnej listy do HL
LFIN    DJNZ ILOOP
        EXX
        LD A,(HL)        ; ostatni, d-ty indeks
        LD BC,(LEN)
        LD HL,0000
DLOOP   OR A
        JR Z,EXIT        ; skok, gdy indeks = 0
        ADD HL,BC        ; względny adres kolejnego elementu
        DEC A            ; redukcja indeksu
        JR DLOOP
EXIT    PUSH HL          ; przesłanie zawartości HL'
        EXX
        POP DE           ;do DE
        ADD HL,DE        ; ostateczny adres elementu
        RET
```

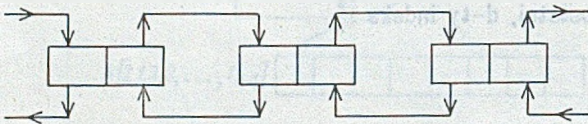
Ponieważ rozkaz pobierania pojedynczego bajtu z pamięci do rejestru B: „LD B,(NUM)” nie istnieje, zastosowano nieskomplikowany chwyt polegający na pobraniu całego BC.

Na zbliżonych schematach adresowania będą się opierać prawie wszystkie typowe programy wykorzystujące struktury listowe. Właściwy użytek z adresów, np. usuwanie elementów, wprowadzanie nowych — w powyższym kontekście byłoby to rozszerzenie tablicy o nowy wymiar — należy już do technik programowania na wyższym poziomie i przykłady w języku asemblera nie będą tu zbyt pouczające. Jest jednak problem, który warto poruszyć.

Struktury listowe i ogólniej struktury danych wiązane adresami znajdują liczne zastosowania wszędzie tam, gdzie mamy do czynienia ze złożonymi

obiektami, które trzeba np. sortować lub rekonfigurować wzajemnie w inny sposób (patrz np. procedury obsługujące stóg). Niestety, jednokierunkowe listy, które nie pozwalają wrócić do poprzedniego elementu, nie są zbyt wygodne w wielu zastosowaniach. Często pomaga zastosowanie rekursji, jak wspomniano w rozdz. 7, ale przy długich listach oznacza to obciążenie stosu liczbą bajtów równą podwojonej długości listy, co może być nie do przyjęcia w niewielkim systemie mikroprocesorowym. Innym rozwiązaniem jest więc zastosowanie list dwukierunkowych: każdy element listy zawiera dwa słowa adresowe zawierające adresy odpowiednio poprzedniego i następnego elementu listy. Jest to pod względem dynamiki programu rozwiązanie prostsze, gdyż można już w trakcie konstrukcji listy wykryć przepełnienie pamięci, ale z punkt widzenia ogólnej pamięciochłonności rozwiązania problem jest jeszcze gorszy. Istnieje jednak wyjście polegające na konstrukcji listy dwukierunkowej za pomocą pojedynczych pól adresowych kosztem nieznacznej komplikacji programu.

Rozważmy dla uproszczenia dwukierunkową listę zorganizowaną jako niekoniecznie spójny ciąg słów przedstawiony na rys. 8.3. Należy umieścić w tych słowach informację umożliwiającą dostęp do poprzednich słów na liście, a także



Rys. 8.3. Lista dwukierunkowa

do słów następnych. Rozwiązanie polega na umieszczeniu w słowie różnicy adresów elementów następnego i poprzedniego. Tym razem, aby obliczyć adres $(i+1)$-go elementu listy musimy dysponować adresami elementów: i-go oraz $(i-1)$-go. Jeśli zawartość słowa o adresie $A(i)$ oznaczymy przez $P(i)$, mamy:

$$P(i) = A(i+1) - A(i-1),$$

skąd dysponując $A(i-1)$ oraz $A(i)$ obliczamy:

$$A(i+1) = P(i) + A(i-1)$$
$$A(i-2) = A(i) - P(i-1)$$

Niech $(i-1)$-szy element będzie adresowany przez rejestr DE, a element i-ty przez HL. Obliczenie adresu elementu $(i+1)$-go jest dane następującym ciągiem rozkazów:

```
LD A,(HL)
INC HL
LD H,(HL)
LD L,A
ADD HL,DE        ; HL zawiera wynik
```

a obliczenie adresu elementu $(i-2)$-go:

```
EX DE,HL
LD C,(HL)
INC HL
LD B,(HL)
EX DE,HL
AND A          ; wyzerowanie przeniesienia
SBC HL,BC      ; HL zawiera wynik
```

Jako samodzielne, bardzo polecane ćwiczenie zostawimy Czytelnikowi sprawdzenie, czy powyższa technika daje dobry wynik dla dowolnych adresów z pełnej przestrzeni 64K, czy nie trzeba ograniczyć pola pamięci przeznaczonego na listy, ze względu na to, że operacje arytmetyczne są wykonywane modulo 65536.

Uwagi zawarte w tym rozdziale poruszają znikomy ułamek zagadnienia struktur danych. Literatura na ten temat jest olbrzymia, jednak niezłym źródłem informacji na temat kodowania ich w języku asemblera są gotowe, sprawdzone programy.

Program deasemblera Z80

Program przedstawiony poniżej rozkodowuje i przedstawia w notacji asemblera jeden rozkaz mikroprocesora Z80, którego adres mieści się w słowie o adresie ADINSTR. Rozkodowany rozkaz zostanie zapisany w tablicy o nazwie DISPL, o długości 32 bajty. Zanim przejdziemy do samego programu opiszemy wykorzystywane struktury danych, algorytm działania i technikę implementacji. Ten deasembler nie rozpoznaje niepublikowanych rozkazów operujących rejestrami indeksowanymi. Z nieoficjalnych rozkazów rozpoznaje jedynie SLI.

Aby zrozumieć działanie programu należy przypomnieć sobie rys. 4.2, przedstawiający strukturę kodu rozkazu Z80. Tak więc dwubitowe pole *K* określa ogólną klasę operacji. Trójbitowe pola *M* i *N* mogą zawierać oznaczenia rejestrów bądź dokładnie specyfikować klasę operacji. Pole *M* dzieli się jeszcze na dwubitowy segment *P* i jednobitowy segment *Q*.

Będziemy potrzebowali kilku tablic zawierających nazwy rejestrów i mnemonikę używane w programie w różnych kontekstach.
Zapiszemy je następująco:

r(0) oznacza B, *r*(1) do *r*(7) oznaczają C, D, E, H, L i A.
ss(0) oznacza BC, *ss*(1): DE, *ss*(2): literę K, *ss*(3): SP.
qq(0) oznacza BC, *qq*(1): DE, *qq*(2): literę Q, *qq*(3): AF.
n(0) oznacza cyfrę 0, *n*(1) do *n*(7): cyfry 1, 2, 3, 4, 5, 6 i 7.
cc(0) oznacza NZ, *cc*(1): Z, *cc*(2): NC, *cc*(3): C, *cc*(4): PO.
cc(5): PE, *cc*(6): P i *cc*(7): M.
x(0) oznacza ADD A, *x*(1): ADC A, *x*(2): SUB_ , *x*(3): SBC A, *x*(4):
AND_ , *x*(5): XOR_ , *x*(6): OR_ i *x*(7): CP_ .
(Podkreślenia tu oznaczają spację.)

Program rozpoczyna działanie od wypełnienia tablicy DISPL spacjami i od wyzerowania zmiennych PREFIX i INDEX. Następnie pobiera bajt rozkazu i podejmuje decyzję:

A. Jeśli PREFIX = 0:

1. Jeśli kod rozkazu jest równy #76, to jest to HALT.
2. Jeśli kod = #CB, to PREFIX := 1 i zacznij od nowa.
3. Jeśli kod = #ED, to PREFIX := 2 i zacznij od nowa.
4. Jeśli kod = #DD, to INDEX := 1 i zacznij od nowa.
5. Jeśli kod = #FD, to INDEX := 2 i zacznij od nowa.
6. Jeśli $K = 0$.
   a. Jeśli $N = 0$·
      Jeśli $M > 3$, to DISPL := JR $cc(M-4)$,V.
      Jeśli $M < 4$, to wybierz M-ty element z listy:[1]
      NOP/EX AF,AF'/DJNZ V/JR V.
   b. Jeśli $N = 1$:
      Jeśli $Q = 0$, to DISPL := LD $ss(P)$,W,
      Jeśli $Q = 1$, to DISPL := ADD K,$ss(P)$.
   c. Jeśli $N = 2$:
      DISPL := LD plus M-ty element listy:
      (BC),A/A, (BC)/(DE),A/A, (DE)/(W),K/K, (W)/(W),A/A, (W).
   d. Jeśli $N = 3$:
      Jeśli $Q = 0$, to DISPL := INC $ss(P)$.
      Jeśli $Q = 1$, to DISPL := DEC $ss(P)$.
   e. Jeśli $N = 4$, to DISPL := INC $r(M)$.
   f. Jeśli $N = 5$, to DISPL := DEC $r(M)$.
   g. Jeśli $N = 6$, to DISPL := LD $r(M)$,V.
   h. Jeśli $N = 7$, wybierz M-ty element listy:
      RLCA/RRCA/RLA/RRA/DAA/CPL/SCF/CCF.
7. Jeśli $K = 1$, to DISPL := LD $r(M)$, $r(N)$.
8. Jeśli $K = 2$, to DISPL := $x(M)$, $r(N)$.
9. Jeśli $K = 3$:
   a. Jeśli $N = 0$, to DISPL := RET $cc(M)$.
   b. Jeśli $N = 1$:
      Jeśli $Q = 0$, to DISPL := POP $qq(P)$.
      Jeśli $Q = 1$, to wybierz P-ty element listy;
      RET/EXX/JP (K)/LD SP,K.
   c. Jeśli $N = 2$, to DISPL := JP $cc(M)$,W.
   d. Jeśli $N = 3$ wybierz M-ty element listy:
      JP W/*/OUT (V),A/IN A,(V)/EX (SP),K/EX DE,HL/DI/EI.
   e. Jeśli $N = 4$, to DISPL := CALL $cc(M)$,W.
   f. Jeśli $N = 5$:
      Jeśli $Q = 0$, to DISPL := PUSH $qq(P)$.
      Jeśli $Q = 1$, to DISPL := CALL W.

---

[1] Ukośna kreska „/" oddziela kolejne elementy listy.

g. Jeśli $N = 6$, to DISPL:= $x(M)$,V.
h. Jeśli $N = 7$, to DISPL:= RST plus M-ty element listy:
00/08/10/18/20/28/30/38.

B. Jeśli PREFIX = 1:
1. Jeśli $K = 0$, to wybierz M-ty element listy:
RLC/RRC/RL/RR/SLA/SRA/SLI/SRL i dopisz $r(N)$.
2. Jeśli $K = 1$, to DISPL:= BIT $n(M),r(N)$.
3. Jeśli $K = 2$, to DISPL:= RES $n(M),r(N)$.
4. Jeśli $K = 3$, to DISPL:= SET $n(M),r(N)$.

C. Jeśli PREFIX = 2:
K nie może się równać zeru.
1. Jeśli $K = 1$:
a. Jeśli $N = 0$, to DISPL:= IN $r(M)$,(C).
b. Jeśli $N = 1$, to DISPL:= OUT (C),$r(M)$.
c. Jeśli $N = 2$:
Jeśli $Q = 0$, to DISPL:= SBC HL,$ss(P)$.
Jeśli $Q = 1$, to DISPL:= ADC HL,$ss(P)$.
d. Jeśli $N = 3$:
Jeśli $Q = 0$, to DISPL:= LD (W),$ss(P)$.
Jeśli $Q = 1$, to DISPL:= LD $ss(P)$,(W).
e. Jeśli $N = 4$, to DISPL:= NEG.
f. Jeśli $N = 5$:
Jeśli $Q = 0$, to DISPL:= RETN.
Jeśli $Q = 1$, to DISPL:= RETI.
g. Jeśli $N = 6$, to wybierz M-ty element listy:
IM 0/*/IM 1/IM 2.
h. Jeśli $N = 7$, to wybierz M-ty element listy:
LD I,A/LD R,A/LD A,I/LD A,R/RRD/RLD.
2. Jeśli $K = 2$, to wybierz N-ty element listy: LD/CP/IN/OT i dopisz M-ty element listy: I/D/IR/DR.
K nie może się równać 3.

Teraz należy uzupełnić wynik. Każde V należy zastąpić jednym dodatkowo wczytanym bajtem, a W — dwoma, zamieniając ich kolejność zgodnie z konwencją umieszczania słów w pamięci. Każde K należy zastąpić HL, IX lub IY w zależności od wartości zmiennej INDEX. Każde Q należy zastąpić HL, IX + d lub IY + d. Przesunięcie d jest zawsze trzecim bajtem rozkazu. (W indeksowanych rozkazach RLC itp., oraz BIT itp. zostanie pobrane przed podstawowym kodem rozkazu.)

Przystąpimy teraz do konstrukcji programu. Paradoksalnie, powyższy opis algorytmu posłuży nam nie jako model programu w języku asemblera, lecz

jako dane do programu. Dane te będą jednobajtowymi kodami rozkazów wyspecjalizowanego interpretatora zawierającego 8 procedur wykonawczych, a także parametry i dane dla tych procedur. Kod rozkazu zajmuje trzy prawe bity bajtu. Następne trzy bity zawierają pierwszy parametr, bit nr 6 — drugi parametr, który zawsze oznacza, że należy przed następnym znakiem wstawić przecinek, a bit nr 7 — jeśli jest ustawiony — oznacza końcowy interpretowany rozkaz w ciągu. Procedury wykonawcze:

0. SWITCH. Po bajcie 0 następuje 8 bajtów zawierających adresy względne, które posłużą do 8-krotnego rozgałęzienia interpretowanego programu, jak to opisano w p. 6.4.

1. LITERAL. Po bajcie kodu rozkazu następuje ciąg znaków, który należy przepisać do DISPL. Ostatni znak ciągu ma ustawiony bit nr 7. Pierwszy parametr różny od zera oznacza, że na końcu należy dopisać spację.

2. LISTM. Wybiera *M*-ty element listy następującej za kodem. Lista składa się z ciągów znaków; każdy zakończony bajtem z ustawionym bitem nr 7. Pierwszy parametr zawiera długość listy.

3. LISTN. Podobnie jak wyżej, wybiera *N*-ty element.

4. SELM. Wybór *M*-tego elementu jednej z tablic: *r*, *ss*, *qq*, *n*, *cc*, *x*. Pierwszy parametr określa, która to tablica.

5. SELN. Podobnie jak wyżej, wybór *N*-tego elementu.

6. SKIPM. Zeruje bit nr 5 dekodowanego bajtu i, jeśli bit ten był ustawiony, pomija liczbę bajtów interpretowanego programu równą wartości pierwszego parametru, w przeciwnym razie interpretuje następny bajt.

7. SKIPQ. Przesuwa w prawo o jeden bit segment *M* dekodowanego bajtu, co jest równoważne $M := P$. Podobnie jak powyżej, pomija ciąg bajtów w zależności od poprzedniej wartości bitu nr 3 (czyli *Q*). Tym razem pominięcie następuje, gdy $Q = 0$.

Dla przykładu zakodujemy symbolicznie początkowy fragment interpretowanego programu (gdy PREFIX i *K* są równe zeru). Podkreślony znak tekstu oznacza, że do jego kodu zostało dodane 128.

Etykieta	Rozkaz	Parametr 1	Bit przecinka	Bit końca
	SWITCH			
	(Następuje 8 1-bajtowych adresów: H0, H1, H2 itd.)			
H0	SKIPM	6		
	LITERAL	1 (spacja)		
	JR			
	SELM	4		
	LITERAL		1	1
	V			
	LISTM	3		1
	NOPEX AF,AF' DJNZ VJR V			

Etykieta	Rozkaz	Parametr 1	Bit przecinka	Bit końca
H1	SKIPQ	6		
	LITERAL	1		
	LD			
	SELM	1		
	LITERAL		1	1
	W			
	LITERAL			
	ADD K			
	SELM	1	1	1
H2	LITERAL	1		
	LD			
	LISTM	7		1
	(BC),AA,(BC)(DE),AA,(DE)(W),KK,(W)(W),AA.(W)			
itd.				

W poniższym tekście dla zwartości cały ten program został zapisany szesnastkowo, opuszczając znak „#". Jest to specjalnie zredagowany wydruk nie odpowiadający konwencji asemblera.

Program rozpoczyna się od tablicy PFTAB będącej pierwszym przełącznikiem zawierającym 11 adresów względnych odpowiadających różnym wartościom zmiennych *K* oraz PREFIX.

```
;----------------------------------------------
;
; Dane interpretowane przez program deasemblera

PFTAB  DEFB
E0,0A,0E,0F,96,AF,B4,B9,BE,BD,C5,09,4C,C4,04,C5,2C,85,00,08,0C
25,2A,55,5C,69,6B,09,52,45,D4,A4,2F,09,50,4F,D0,94,9A,52,45,D4
45,58,D8,4A,50,20,28,4B,A9,4C,44,20,53,50,2C,CB,09,4A,D0,24,C1
D7,BA,4A,50,20,D7,A0,4F,55,54,20,28,56,29,2C,C1,49,4E,20,41,2C
28,56,A9,45,58,20,28,53,50,29,2C,CB,45,58,20,44,45,2C,48,CC,44
C9,45,C9,09,43,41,4C,CC,24,C1,D7,37,09,50,55,53,C8,94,81,43,41
4C,4C,20,D7,2C,81,D6,09,52,53,D4,BA,30,B0,30,B8,31,B0,31,B8,32
B0,32,B8,33,B0,33,B8,3A,52,4C,C3,52,52,C3,52,CC,52,D2,53,4C,C1
53,52,C1,53,4C,C9,53,52,CC,01,A0,85,09,42,49,D4,1C,C5,09,52,45
D3,1C,C5,09,53,45,D4,1C,C5,00,B4,BB,C3,D1,DE,E1,E9,F0,1B,4C,C4
43,D0,49,CE,4F,D4,06,9A,C9,C4,49,D2,44,D2,00,08,25,32,61,6B,6F
73,78,36,09,4A,D2,24,C1,D6,9A,4E,4F,D0,45,58,20,41,46,2C,41,46
A7,44,4A,4E,5A,20,D6,4A,52,20,D6,37,09,4C,C4,0C,C1,D7,01,41,44
44,20,CB,CC,09,4C,C4,BA,28,42,43,29,2C,C1,41,2C,28,42,43,A9,28
44,45,29,2C,C1,41,2C,28,44,45,A9,28,57,29,2C,CB,4B,2C,28,57,A9
28,57,29,2C,C1,41,2C,28,57,A9,2F,09,49,4E,C3,8C,09,44,45,C3,8C
09,49,4E,C3,84,09,44,45,C3,84,09,4C,C4,04,C1,D6,BA,52,4C,43,C1
52,52,43,C1,52,4C,C1,52,52,C1,44,41,C1,43,50,CC,53,43,C6,43,43
```

```
C6,09,49,CE,04,C1,28,43,A9,01,4F,55,54,20,28,43,A9,C4,3F,01,53
42,43,20,CB,CC,01,41,44,43,20,CB,CC,09,4C,C4,2F,01,28,57,A9,CC
0C,C1,28,57,A9,81,4E,45,C7,01,52,45,D4,17,81,CE,81,C9,09,49,CD
9A,B0,AD,B1,B2,3E,8A,52,52,C4,52,4C,C4,09,4C,C4,9A,49,2C,C1,52
2C,C1,41,2C,C9,41,2C,D2
;-----------------------------------------------------------
; Następuje tablica SELTAB rozpoczynająca się od
; przełącznika. Dla dokumentacji została wypisana w mnemonice
; asemblera, jednak również z pewną poprawką redakcyjną,
; mianowicie podkreślone znaki oznaczają odpowiedni kod ASCII
; plus 128.
;
SELTAB  DEFB SEL0-$
        DEFB SEL1-$
        DEFB SEL2-$
        DEFB SEL3-$
        DEFB SEL4-$
        DEFB SEL5-$
SEL0    DEFM "BCDEHL(Q)A"
SEL1    DEFM "BCDEKSP"
SEL2    DEFM "BCDEKAF"
SEL3    DEFM "01234567"
SEL4    DEFM "NZZNCCPOPEPM"
SEL5    DEFM "ADD      A,ADC A,SUB_
                   SBC A,AND_XOR_OR_CP_"
;
HLTX    DEFM "HALT"
REGTAB  DEFM "HLIXIY"
;-----------------------------------------------------------
ADINST  DEFW 0          ; zawiera adres dekodowanego rozkazu
; adres ten zostanie przekazany do BC'.
        DEFB " "        ; spacja
DISPL   DEFS 32         ; zawiera zdekodowany adres, kolejne
; jego bajty i mnemonikę. Adres DISPL zostanie przekazany do
; IX. Adres DISPL + 18 do HL'.
;-----------------------------------------------------------
; Początek programu interpretera
;
INIT
        EXX             ; alternatywny bank rejestrów
        LD HL,DISPL-1   ; adres spacji
        LD DE,DISPL
        LD BC,32        ; długość DISPL
        LDIR            ; wypełnienie spacjami
        LD BC,(ADINST)
        LD IX,DISPL
        LD HL,DISPL+18
        CALL HEX2       ; wydruk adresu
```

```
        EXX             ; główny bank rejestrów
        INC IX          ; wydruk spacji po adresie
        LD  DE,0000     ; inicjacja zmiennych INDEX i PREFIX
; D = 0 dla instrukcji używajacej HL
;     1 dla IX
;     2 dla IY
; E = 0 dla zwykłych kodów
;     1 dla prefiksowanych CB
;     2 dla prefiksowanych ED
        JR START
;-----------------------------------------------
; Procedura konwersji szesnastkowej
HEX2    LD A,B          ; dwa bajty w BC
        CALL HEX1       ; konwersja lewego
        LD A,C          ; ... i prawego
HEX1    PUSH AF         ; jeden bajt w A
        RRCA            ; obrót w prawo o 4 bity; najpierw
        RRCA            ;konwersja bardziej znaczącej
        RRCA            ;tetrady.
        RRCA            ; lewa połowa bajtu
        CALL TETRAD
        POP AF          ; teraz prawa tetrada
TETRAD  AND #0F         ; 1 cyfra
        ADD A,#30       ; -> ASCII
        CP #3A          ; czy > "9"?
        JR C,DIGIT
        ADD A,#07       ; litera
DIGIT   LD (IX),A       ; przesłanie do DISPL
        INC IX
        RET
;-----------------------------------------------
; pobranie jednego bajtu i jego konwersja
FETCH   EXX             ; alternatywny bank rejestrów
        LD A,(BC)       ; pobranie znaku
        CALL HEX1       ; zapisanie w DISPL
        LD A,(BC)       ; odtworzenie wartości A
        INC BC
        LD (ADINST),BC
        EXX             ; powrót do głównego banku rejestrów
        RET
;-----------------------------------------------
IND2    INC D           ; inkrementacja zmiennej INDEX
IND1    INC D
START                   ; początek dekodowania
        CALL FETCH
        CP #76
        JR Z,IHALT      ; dekodowanie rozkazu HALT
        CP #DD
```

```
        JR Z,IND1        ; INDEX := 1
        CP #FD
        JR Z,IND2        ; INDEX := 2
;   Rozkazy wykorzystujące rejestry indeksowe powtarzają
; pobieranie dekodowanego bajtu
        CP #CB           ; sprawdzenie prefiksów
        JR Z,PREF1       ; wtedy PREFIX := 1
        CP #ED
        JR NZ,START1
        INC E            ; tu PREFIX := 2
PREF1   INC E
        XOR A
        OR D             ; sprawdzenie czy nie jest to rozkaz
;                             DDCB/FDCB
        CALL NZ,FETCH ; jeśli tak, pobrać przesunięcie
        CALL FETCH       ; pobranie właściwego bajtu kodu
;-------------------------------------------------------
START1                   ; akumulator zawiera dekodowany bajt
        LD (SAVA),A      ; przechowanie akumulatora
        RLA
        RL E
        RLA
        RL E
        LD A,E
; teraz A zawiera 4 * PREFIX + wartość pola K
        LD HL,PFTAB      ; tablica zawierająca pocz.bloków
; sterujących (czyli interpretowany program)

        CALL SWIT1       ; przełącznik
; teraz w HL: adres odpowiedniego bloku sterującego
;-------------------------------------------------------
; Początek głównej pętli interpretera
MASTER
        LD A,(HL)        ; pobranie bajtu interpretowanego
                         ; rozkazu
        INC HL
        LD BC,MRETURN
        PUSH BC          ; w BC: adres powrotu z procedur
                         ; interpretera
        PUSH HL          ; adres następnego rozkazu
        PUSH AF          ; chwilowe zapamiętanie A
        LD HL,PROCTB     ; Tablica adresów (względnych)
                         ; procedur interpretera
        CALL SWITCH      ; przełącznik
        POP AF
        RRCA
        RRCA
        RRCA
```

```
; A zawiera teraz parametry procedury
        LD E,A          ; przekazane do E
        EX (SP),HL
; teraz na stosie adres procedury
; w HL: adres następnego rozkazu w bloku sterującym
; w A - dekodowany bajt
; w E - parametry
        LD A,(SAVA)
        RET             ; przekazanie sterowania
;---------------------------------------------------------
MRETURN
        BIT 4,E         ; czy koniec bloku sterującego?
        JR Z,MASTER     ; jeśli nie: nowy rozkaz
        RET             ; koniec dekodowania.
;---------------------------------------------------------
; Dekodowanie rozkazu HALT
IHALT   LD HL,HLTX      ; adres tekstu
        JR LITC         ; wewnątrz LITERAL
;---------------------------------------------------------
; Tablica procedur interpretera
PROCTB  DEFB SWITCH-$
        DEFB LITERAL-$
        DEFB LISTM-$
        DEFB LISTN-$
        DEFB SELM-$
        DEFB SELN-$
        DEFB SKIPM-$
        DEFB SKIPQ-$
;
SWITCH                  ; obliczenie adresu z przesunięcia
        AND #07         ; numer procedury zajmuje 3 bity
SWIT1   LD C,A
        LD B,00         ; BC zawiera przesunięcie
        ADD HL,BC       ; pozycja w tablicy
        LD C,(HL)       ; pobranie bajtu
        ADD HL,BC       ;i obliczenie adresu
        RET
;---------------------------------------------------------
LITERAL
        CALL LITC
        BIT 0,E         ; czy będzie drukowana spacja?
        RET Z
        EXX
        INC HL          ; przesunięcie o 1 pozycję w DISPL
        EXX
        RET
```

```
LITC
         BIT 3,E          ; czy wydrukować przecinek?
         RES 3,E
         LD A,","
         CALL NZ,INSCHAR; drukowanie jednego znaku z A
LITC1    LD A,(HL)        ; kolejny bajt literału
         BIT 7,A          ; czy końcowy?
         RES 7,A
         PUSH AF          ; zapamiętanie A i przeniesienia!
         CALL NC,INSERT; kolejny znak do DISPL
; bit przeniesienia w LITERAL jest równy zeru. Jeśli LITC
; jest wywoływane np. z SELM, trzeba opuścić część tablicy
         INC HL           ; adres następnego znaku w bloku
         POP AF
         JR Z,LITC1       ; czy ostatni
         RET
;--------------------------------------------------
LISTM
         RRA              ; operacja równoważna N := M
         RRA
         RRA
LISTN    AND #07          ; selekcja segmentu N
         LD C,A
         LD A,E
         AND #07          ; parametr
         LD B,A           ; liczba elementów listy
         INC B            ; + 1
LH0      XOR A
         CP C             ; wybranie C-tego; (przeniesienie!)
         CALL LITC        ; wydruk ciągu znaków
         DEC C
         DJNZ LH0
         RET
;--------------------------------------------------
SELM
         RRA              ; równoważne N := M
         RRA
         RRA
SELN
         PUSH HL          ; zapamiętanie
         LD HL,SELTAB
         PUSH AF          ; chwilowe zapamiętanie
         LD A,E
         CALL SWITCH      ; wybór listy
         POP AF           ; odtworzenie A
         AND #07          ; tylko N
         LD C,A
         CALL LH0-2
```

```
        POP HL          ; odtworzenie adresu w bloku
        RET
;-----------------------------------------------------------
SKIPQ
        RRCA
        XOR #04
        BIT 2,A         ; wartość segmentu Q
        JR SKIPM1
;-----------------------------------------------------------
SKIPM
        BIT 5,A
SKIPM1  RES 5,A
        LD (SAVA),A
; zmiana dekodowanego bajtu: M := M - 4  jeśli M > 3
        RET NZ          ; nic nie robić jeśli M było < 4
        LD A,E          ; pobranie parametru
        AND #07
        LD C,A
        LD B,0          ; do BC
        ADD HL,BC       ;i przeskok
        RET
;-----------------------------------------------------------
INSERT                  ; wprowadzanie znaku do DISPL (HL')
        CP "K"
        JR Z,REGIST     ; jest to HL, IX lub IY
        CP "Q"
        JR Z,INDIR      ; jest to (HL), (IX+d) lub (IY+d)
        CP "V"
        JR Z,LBYTE      ; jednobajtowa stała
        CP "W"
        JR Z,LWORD      ; dwubajtowa stała
INSCHAR                 ; wpisanie znaku
        EXX
        LD (HL),A
        INC HL
        EXX
        RET
LBYTE
        CALL FETCH      ; pobranie nowego bajtu
        LD A,(IX-2)     ; przepisanie 2 znaków z DISPL
        CALL INSCHAR
        LD A,(IX-1)
        JR   INSCHAR
;-----------------------------------------------------------
LWORD
        CALL FETCH      ; mniej znaczący bajt
        CALL LBYTE      ; bardziej znaczący
```

```
LWORD1
        LD A,(IX-4)       ; przepisanie 4 znaków tekstu
        CALL INSCHAR      ;  w odpowiedniej kolejności
        LD A,(IX-3)
        JR  INSCHAR
;------------------------------------------------------------
REGIST
        PUSH HL           ; chwilowe zapamiętanie
        PUSH BC
        LD HL,REGTAB      ; REGTAB symuluje blok sterujacy
        LD C,D            ; który rejestr: zależnie od INDEX
        LD B,#03          ; są trzy możliwosci
        CALL LH0
        POP BC
        POP HL
        RET
;------------------------------------------------------------
INDIR
        CALL LREG         ; na początku nazwa rejestru
        LD A,D
        OR A
        RET Z             ; powrót jeśli INDEX = 0
        LD A,"+"
        CALL INSCHAR
        LD A,(DISPL+11)
        CP " "
        JR Z,LBYTE        ; skok jeśli to nie DDCB/FDCB
        JR LWORD1         ; przesunięcie już jest
;------------------------------------------------------------
SAVA    DEFB
        END
```

Wybrane procedury arytmetyczne

Dodatek ten zawiera dwie kompletne procedury w języku asemblera i kilka użytecznych algorytmów dla liczb zmiennopozycyjnych.

```
; Procedura MULT44 mnożenia czterobajtowych liczb całkowitych.
; Opiera się na szybkim algorytmie podanym w rozdz. 7.3. Jest
; on cytowany jako metoda Karacuby (Dokł. Akad. Nauk ZSRR,
; 1962). Wykorzystuje 6 razy procedurę MULT21
; Parametry:
; MEM1, MEM2 - słowa zawierające adresy liczb umieszczonych
; w pamięci od najmniej znaczących adresów w górę.
; MULRES - adres ośmiobajtowego wyniku
;----------------------------------------------------------
MULT44
         LD HL,(MEM1)     ; adres mnożnej  do HL'
         LD DE,(MEM2)     ; adres mnożnika do DE'
         EXX
         LD HL,(MULRES)   ; adres wyniku
         LD (MEMRES),HL   ;    dodatkowo przechowany
         CALL ZERRES      ; zerowanie pola wynikowego
         LD A,04          ; początek głównej pętli
LOOP1    EXX
         EX AF,AF'        ; przechowanie licznika pętli
         LD A,(DE)        ; bajt mnożnika
         INC DE           ; przygotowanie następnego
         PUSH HL          ; adres odp. słowa mnożnej
         EXX
         POP HL           ; przesłany do drugiego banku
         LD E,(HL)
         INC HL
         LD D,(HL)        ; słowo mnożnej do DE
         CALL MULT21      ; mnożenie A*DE -> C,HL
```

```
        PUSH HL
        EX DE,HL        ; segment wyniku w C, DE
        PUSH BC
        LD HL,(MEMRES)
        CALL RESADD     ; dodanie segmentu wyniku
        LD HL,(MEMRES)
        INC HL
        INC HL
        POP BC
        POP DE
        CALL RESADD     ; dodanie (C,DE) * 2↑16
        CALL INSFIN     ; przesunięcie pozycji wyniku
        EX AF,AF'       ; licznik pętli
        DEC A
        JR Z,UCALC      ; koniec pętli?
        CP 2            ; po dwukrotnym przebiegu należy
        JR NZ,LOOP1     ;   pobrać następne słowo mnożnej
        EXX             ; od odpowiedniej pozycji wyniku
        INC HL          ; adres mnożnej
        INC HL          ; zwiększenie pozycji o 2
        JR LOOP1+1      ; ominięcie EXX
MEMRES  DEFW 0          ; pomocnicza zmienna
;---------------------------------------------------
; Druga część procedury mnożenia; obliczenie różnic słow
; mnożnej i bajtów wyniku i dodanie do wyniku na odp. pozycję.
UCALC
        LD HL,(MULRES)
        INC HL
        INC HL
        LD (MEMRES),HL  ; adres drugiego słowa wyniku
        LD HL,(MEM2)    ; adres pierwszego bajtu mnożnika
        CALL INSERT     ; obliczenie różnic i iloczynów
        LD HL,(MEM2)
        INC HL          ; adres drugiego bajtu mnożnika
INSERT  LD A,(HL)       ; bajt mnożnika
        INC HL
        INC HL
        SUB (HL)        ; odjęcie dalszego sąsiada
        EX AF,AF'       ; schowanie A, ale także F!
        LD HL,(MEM1)
        LD E,(HL)
        INC HL
        LD D,(HL)       ; słowo mnożnej do DE
        INC HL
        LD A,(HL)
        INC HL
        LD H,(HL)
```

```
        LD L,A          ; następne słowo do HL
        EX AF,AF'       ; obliczona wyżej różnica
        JR NC,APLUS     ; jeśli dodatnia, O.K.
        EX DE,HL        ; zamiana słów mnożnej
        NEG             ;  i zmiana znaku różnicy
APLUS   OR A            ; wyzerowanie przeniesienia
        SBC HL,DE       ; różnica słów mnożnej
        PUSH AF         ; jeszcze raz przechowanie F
        JR NC,UPLUS     ; jeśli ta różnica dodatnia, O.K.
        ADD HL,DE
        OR A
        EX DE,HL
        SBC HL,DE       ; zmiana znaku; teraz dodatnia
UPLUS   EX DE,HL
        CALL MULT21     ; iloczyn obu różnic
        POP AF          ; jeśli różnica słów była ujemna
        EX DE,HL
        LD HL,(MEMRES)  ;  od odpowiedniej pozycji wyniku
        JR NC,READY
        CALL RESSUB     ;  odjąć iloczyn
        JR INSFIN       ; i zakończyć
READY                   ; jeśli różnica była dodatnia
        CALL RESADD     ;  iloczyn należy dodać
INSFIN  LD HL,(MEMRES)
        INC HL          ; następna pozycja wyniku
        LD (MEMRES),HL
        RET
;--------------------------------------------------
; Pomocnicza procedura dodająca 3-bajtowy częściowy wynik
; w rejestrach C i DE do tablicy zawierającej wynik. Adres
; tablicy w HL.
RESADD  LD B,03         ; licznik pętli
ADLOOP  LD A,(HL)       ; pobranie bajtu z tabl. wynikowej
        ADC A,E         ; zawsze dodawanie tylko E
        LD (HL),A       ; wprowadzenie na poprzednie miejsce
        INC HL          ; przejście do następnego
        LD E,D
        LD D,C
        LD C,00         ; przesunięcie: 0 -> C -> D -> E
        DJNZ ADLOOP
        INC B           ; jeśli pozostanie przeniesienie
        JR C,ADLOOP     ; poprawienie następnego bajtu
        RET
;--------------------------------------------------
; Pomocnicza procedura odejmująca segment wyniku. Patrz RESADD
RESSUB  OR A
        LD B,03
```

```
SBLOOP  LD A,(HL)
        SBC A,E
        LD (HL),A
        INC HL
        LD E,D
        LD D,C
        LD C,00
        DJNZ SBLOOP
        INC B
        JR C,SBLOOP
        RET
;-----------------------------------------------------
ZERRES  LD B,08          ; procedura zerująca tablicę
        XOR A
ZERLOP  LD (HL),A
        INC HL
        DJNZ  ZERLOP
        RET
;-----------------------------------------------------
; Procedura mnożenia 8-bitowej liczby w A przez 16-bitową
; w DE. 24-bajtowy wynik w C (najbardziej znaczący bajt) i HL.
MULT21  LD HL,0000
        LD BC,#0800      ; B - licznik pętli
NXBIT   ADD HL,HL
        RLA              ; wysunięcie bitu mnożnika
        JR NC,FIN21
        ADD HL,DE        ; dodanie mnożnej jeśli równy 1
        ADC A,C          ; szybsze niż ADC A,00
FIN21   DJNZ NXBIT
        LD C,A           ; przesłanie wyniku
        RET
;-----------------------------------------------------

; Procedura DIVN obliczania przeskalowanej odwrotności
; 4-bajtowej liczby metodą iteracyjną Newtona
; Parametry:
; MEM - słowo    zawierające adres czterobajtowego argumentu.
; Liczba winna być tak znormalizowana by najbardziej znaczący
; bit był równy 1. (Będzie symbolicznie oznaczana jako Y)
; DIVRES - słowo zawierające adres 8-bajtowej tablicy. Wynik
; mieści się w czterech bardziej znaczących bajtach i
; będzie symbolicznie nazywany X.
; Najbardziej znaczący bit jest równy 1. Jeśli argument jest
; znormalizowaną mantysą, wynik zawiera się między 1 a 2.
```

```
DIVN
        LD HL,(DIVRES)  ; adres ośmiobajtowej tablicy
        INC HL
        INC HL
        INC HL
        INC HL
        LD (RES44),HL   ; adres czterobajtowego wyniku
        LD B,04
        CALL ZERRES+2   ; wyzerowanie 4 bajtów
        DEC HL
        PUSH HL
        LD HL,(MEM)     ; adres argumentu
        INC HL
        INC HL
        LD E,(HL)       ; dwa najbardziej znaczące bajty
        INC HL
        LD D,(HL)       ; w DE
        LD A,127
        SUB D
        LD (TEMP),A     ; zerowe przybliżenie. 2 bity O.K.
        POP HL
        LD (HL),A       ; górny bajt wyniku
        LD B,03         ; licznik pętli
ITER1   PUSH DE         ; początek pierwszej iteracji
        PUSH BC
        CALL MULT21     ; X * Y
        LD D,A
        LD E,H          ; przesłanie częściowego wyniku
        LD A,(TEMP)     ; jescze raz pobranie X
        CALL MULT21     ; X↑2 * Y
        EX DE,HL
        LD HL,(RES44)   ; adres wyniku
        INC HL
        CALL RESSUB     ; odjęcie iloczynu
        LD HL,(RES44)   ; adres wyniku
        LD B,03         ; przygotowanie mnożenia przez 2
SHFT2   INC HL
        RL (HL)
        DJNZ SHFT2      ; przesunięcie w lewo o 1 bit
        LD A,(HL)
        LD (TEMP),A     ; górny bajt kolejnego przybliżenia
        POP BC          ; licznik pętli
        POP DE          ; adres argumentu
        DJNZ ITER1      ; następna szybka iteracja
; teraz DIVRES zawiera ponad 1 bajt wyniku i poprawienie wymaga
; dłuższego mnożenia
        LD B,02         ; licznik pętli
```

```
NXITER PUSH BC
       LD HL,TEMP       ; adres pośredniego wyniku
       LD (MULRES),HL
       LD HL,(MEM)
       LD (MEM1),HL     ; przygotowanie mnożnej
       LD HL,(RES44)
       LD (MEM2),HL     ; przygotowanie mnożnika
       CALL MULT44      ; X * Y
       LD HL,TEMP+4     ; adres górnej połowy wyniku
       LD (MEM1),HL
       LD HL,TEMP1      ; kolejny pośredni wynik
       LD (MULRES),HL
       CALL MULT44      ; X↑2 * Y
       LD HL,TEMP1+4    ; adres górnej połowy wyniku
       LD DE,(RES44)    ; adres ostatecznego wyniku
       LD B,04          ; licznik pętli
SUBTR  LD A,(DE)        ; odjęcie 4 bajtów
       SBC A,(HL)
       LD (DE),A
       INC HL           ; następne pozycje odjemnika
       INC DE           ;   i odjemnej
       DJNZ SUBTR       ; końcowe przeniesienie nieważne
       LD B,05          ; licznik pętli
       LD HL,(RES44)
       DEC HL           ; adres 5 bajtów wyniku
MUL2   RL (HL)          ; pętla mnożenia przez 2
       INC HL
       DJNZ MUL2
       POP BC           ; licznik pętli
       DJNZ NXITER      ; następna iteracja
```

Procedury te mimo dość zwartego kodu są długie i jest to niestety cena, którą trzeba zapłacić. Znane są jeszcze szybsze metody mnożenia, patrz np. pozycja [6]: oparty na szybkiej transformacie Fouriera algorytm Schönhagego--Strassena oraz metoda Tooma-Cooka, które mogą mieć praktyczne znaczenie dla liczb o znacznie większej długości.

Poniżej zostały zebrane uwagi dotyczące złożonych procedur arytmetyki zmiennopozycyjnej. Ze względu na brak miejsca nie podajemy programów w języku asemblera, jednak prezentacja jest jawnie dostosowana do programowania na poziomie podstawowym. Zakładamy, że operujemy liczbami zmiennopozycyjnymi dwójkowymi, dodatnimi, o mantysie (znormalizowanej) równej m i wykładniku równym c.

Najczęściej spotykaną metodą implementacji takich funkcji jak logarytm, funkcje trygonometryczne itp. jest metoda szeregów potęgowych. Z reguły używa się szeregów Czebyszewa ze względu na to, że dokładność przybliżenia

funkcji szeregiem (obciętym) jest równomierna w całym przedziale argumentów, w którym to przybliżenie się stosuje. Ta metoda ma niestety kilka wad: zmusza do przechowywania dużych tablic współczynników; może być zbyt powolna, gdy wynik nie musi być podany z dużą dokładnością; gdy potrzebna jest dokładność większa niż założona, trzeba startować od nowej tablicy współczynników.

Przedstawimy teraz algorytmy obliczania funkcji oparte na innych metodach niż metoda szeregów potęgowych. Algorytmy te są uniwersalne, elastyczne i w wielu wypadkach bardzo sprawne.

● Iteracyjna metoda Newtona jest zdecydowanie najszybszą i najprostszą techniką obliczania wartości wielu funkcji dzięki temu, że każdy krok iteracyjny podwaja liczbę znaczących bitów. Typowym, wręcz demonstracyjnym przykładem jest obliczanie pierwiastka kwadratowego x z liczby y. Schemat iteracyjny ma postać:

$$x \leftarrow (x+y/x)/2$$

Dostosowanie tej metody do poziomu asemblera wygląda następująco:

1. Jeśli cecha jest nieparzysta — należy cechę zaokrąglić w górę i przesunąć mantysę w prawo o 1 bit. Podzielić cechę przez 2 otrzymując cechę wyniku.

2. Mantysa (ewentualnie przesunięta w prawo) jest zawarta między 1/4 a 1, co gwarantuje od razu normalizację wyniku. Aby obliczyć zerowe przybliżenie równe $(1+m)/2$, należy przesunąć mantysę w prawo i ustawić najbardziej znaczący bit.

3. Należy wykonać pętlę iteracyjną do żądanej dokładności. Przypominamy, że pierwsze dwa mnożenia można wykonywać na pojedynczych bajtach.

● Funkcję wykładniczą $\exp(x)$ obliczymy w inny sposób. Dla małych x bardzo dobrym przybliżeniem jest przybliżenie wymierne Padégo dane wzorem:

$$\exp(x) = (12+6x+x^2)/(12-6x+x^2)$$

Dla $x=1$ błąd tego przybliżenia wynosi ok. 0,15%.

Metoda postępowania jest następująca:

1. Jeśli mantysa jest większa od 1/2, odejmujemy od niej 1/2, a przechowujemy mnożnik $e^{1/2}$, przez który zostanie pomnożony końcowy wynik.

2. Teraz można dla mantysy zastosować przybliżenie Padégo, które daje już przynajmniej 5 znaczących cyfr dziesiętnych wyniku, lub powtórzyć powyższą operację zamieniając 1/2 na 1/4 itd.

3. Otrzymany wynik należy podnieść do potęgi 2^c przez kolejne mnożenia. Stosując podział binarny

$$x^{2^c} = (x^{2^{c-1}})^2$$

koszt tej operacji będzie proporcjonalny do wielkości cechy. Potęgowanie będzie obecne w każdym algorytmie obliczania funkcji wykładniczej.

4. Przy ujemnym argumencie żadne dodatkowe dzielenie nie jest potrzebne, należy tylko na początku zapamiętać znak mantysy i w odpowiednim momencie zamienić licznik i mianownik aproksymanty Padégo.

- Logarytm $x = \log(y)$ można obliczyć na dwa sposoby. W obu metodach należy zacząć od następującego kroku:

1. Jeśli $y = m \cdot 2^c$, to $\log(y) = c \cdot \log(2) + \log(m)$. Część całkowita logarytmu przy podstawie 2 jest więc dana od razu; logarytmu naturalnego czy dziesiętnego wymaga jednego mnożenia przez stałą ($\log_{10}2$ lub $\ln 2$).

Metoda Newtona dostarcza następującego schematu dla logarytmu naturalnego:

$$x \leftarrow x - 1 + y \cdot \exp(-x)$$

Jest to atrakcyjne rozwiązanie biorąc dlatego, że x jest tu zawarte w niewielkiem przedziale: od 0 do $\ln(1/2) = -0{,}693$.

Druga metoda, opisana przez Gospera w 1962 r. nadaje się dla logarytmów przy podstawie 2. Może mieć praktyczne zastosowanie, gdy potrzebne jest bardzo zgrubne oszacowanie logarytmu mantysy (tylko kilka bitów), gdyż ta metoda jest, obrazowo mówiąc, postawieniem metody Newtona „na głowie": jedno mnożenie jest potrzebne, aby otrzymać jeden bit wyniku (ale za to nie potrzeba wykonywać mnożenia pełnej długości). Polega ona na wykonaniu następujących kroków, przy użyciu pomocniczej zmiennej o nazwie *logm*:

2. Zainicjować wynik $logm = 0$, a następnie powtarzać następującą sekwencję kroków:

3. Pomnożyć *logm* przez 2. Podnieść mantysę do kwadratu. Jeśli wynik jest większy lub równy 1/2, do *logm* dodać 1, w przeciwnym razie wynikową mantysę pomnożyć przez 2. Powrócić do początku pętli.

4. Po zakończeniu pętli należy podzielić *logm* przez 2 do potęgi równej liczbie kroków pętli (co nie kosztuje nic, *logm* jako ciąg bitów jest ułamkiem o znanym położeniu kropki binarnej), a następnie odjąć 1.

- Przy obliczaniu funkcji trygonometrycznych wydaje się, że nie uniknie się szeregów, jednak i tu istnieją prostsze metody. Obliczanie $x = \sin(y)$ można oprzeć na następującym rekursywnym schemacie potrojonego kąta.

$$\sin(y) = 3 \cdot \sin(y/3) - 4 \cdot (\sin(y/3))^3$$

Na każdym poziomie rekursji (którą można zrealizować iteracyjnie) występują dwa mnożenia i jedno dzielenie przez 3, oprócz operacji znacznie mniej kosztownych. Rekursję można przerwać, gdy drugi człon jest dostatecznie mały

w porównaniu z pierwszym. Górne oszacowanie błędu otrzymamy zastępując dla małych liczb wartość funkcji sinus jej argumentem. Warunkiem przerwania rekursji jest $y \gg 4 \cdot (y/3)^3$, czyli $y \ll \sqrt{27/2} = 2{,}6$. Relację „dużo mniejsze" należy rozumieć tak, że y zawiera już tylko nieznaczące z punktu widzenia żądanej dokładności bity w porównaniu z 2,6. Metoda może posłużyć jako baza hierarchii metod pochodnych: redukcja kąta do $y/9$, $y/27$ itp. Oczywiście obliczanie funkcji cosinus czy tangens może korzystać z powyższej procedury oraz pierwiastka.

• Aby wyczerpać przegląd funkcji elementarnych i jednocześnie zaproponować metodę opartą na rozwinięciu potęgowym przejdziemy do funkcji cyklometrycznych. Rozwinięcie potęgowe arcsin czy arccos można łatwo znaleźć. Szybciej zbieżny jest jednak schemat rozwinięcia, w którym bierze się pod uwagę fakt, że funkcja cosinus w pobliżu zera może być bardzo dobrze przybliżona parabolą, a więc niezłym przybliżeniem funkcji $\arccos(1-y)$ jest pierwiastek z $2 \cdot y$. Stosunek wartości funkcji do jej przybliżenia pierwiastkiem wyraża się szybko zbieżnym szeregiem. Ostatecznie oznaczając $z = y/4$ mamy:

$$\arccos(1-y) = \sqrt{2y} \cdot \left(1 + \frac{z}{3} + \frac{z^2 \cdot 3}{5 \cdot 2!} + \frac{z^3 \cdot 3 \cdot 5}{7 \cdot 3!} + \ldots\right)$$

Struktura szeregu odpowiada następującemu schematowi:

```
sum := 1; coeff := 1;
for k := 1 to Nfinal do
    begin
        coeff := coeff * z * (2 * k - 1)/k;
        sum  := sum + coeff/(2 * k + 1)
    end;
```

W najgorszym przypadku ($y = 1$) rozwinięcie do członu kwadratowego włącznie daje błąd ok. 0,7%. Do otrzymania czterobajtowej dokładności trzeba wziąć ok. 8 członów szeregu (choć dla y bliskich 1 trzeba ich więcej), ale jest szybszy sposób. Korzystając ze wzoru na cosinus podwojonego kąta otrzymamy wzór:

$$\arccos(x) = 2 \cdot \arccos\left(\sqrt{(x+1)/2}\right)$$

co zmniejsza wartość y w szeregu z 1 do 0,3. Nic nie stoi na przeszkodzie, aby powtórzyć tę operację, dysponując szybką procedurą pierwiastkującą. Jako ćwiczenie proponujemy przeanalizowanie efektywności tej metody, biorąc pod uwagę, że każdy krok pierwiastkowania metodą Newtona zawiera jedno dzielenie.

Tablica kodów rozkazów mikroprocesora Z80 uszeregowana alfabetycznie

W mnemonicznym zapisie rozkazów N oznacza stałą jednobajtową, NN — stałą dwubajtową, a d — jednobajtową stałą przesunięcia w rozkazach indeksowanych. W kodzie rozkazu stałe te są zapisywane jako nn, nnnn i dd.

KOD	ROZKAZ	KOD	ROZKAZ	KOD	ROZKAZ
8E	ADC A,(HL)	29	ADD HL,HL	CB43	BIT 0,E
DD8Edd	ADC A,(IX+d)	39	ADD HL,SP	CB44	BIT 0,H
FD8Edd	ADC A,(IY+d)	DD09	ADD IX,BC	CB45	BIT 0,L
8F	ADC A,A	DD19	ADD IX,DE	CB4E	BIT 1,(HL)
88	ADC A,B	DD29	ADD IX,IX	DDCBdd4E	BIT 1,(IX+d)
89	ADC A,C	DD39	ADD IX,SP	FDCBdd4E	BIT 1,(IY+d)
8A	ADC A,D	FD09	ADD IY,BC	CB4F	BIT 1,A
8B	ADC A,E	FD19	ADD IY,DE	CB48	BIT 1,B
8C	ADC A,H	FD29	ADD IY,IY	CB49	BIT 1,C
8D	ADC A,L	FD39	ADD IY,SP	CB4A	BIT 1,D
CEnn	ADC A,N	A6	AND (HL)	CB4B	BIT 1,E
ED4A	ADC HL,BC	DDA6dd	AND (IX+d)	CB4C	BIT 1,H
ED5A	ADC HL,DE	FDA6dd	AND (IY+d)	CB4D	BIT 1,L
ED6A	ADC HL,HL	A7	AND A	CB56	BIT 2,(HL)
ED7A	ADC HL,SP	A0	AND B	DDCBdd56	BIT 2,(IX+d)
86	ADD A,(HL)	A1	AND C	FDCBdd56	BIT 2,(IY+d)
DD86dd	ADD A,(IX+d)	A2	AND D	CB57	BIT 2,A
FD86dd	ADD A,(IY+d)	A3	AND E	CB50	BIT 2,B
87	ADD A,A	A4	AND H	CB51	BIT 2,C
80	ADD A,B	A5	AND L	CB52	BIT 2,D
81	ADD A,C	E6nn	AND N	CB53	BIT 2,E
82	ADD A,D	CB46	BIT 0,(HL)	CB54	BIT 2,H
83	ADD A,E	DDCBdd46	BIT 0,(IX+d)	CB55	BIT 2,L
84	ADD A,H	FDCBdd46	BIT 0,(IY+d)	CB5E	BIT 3,(HL)
85	ADD A,L	CB47	BIT 0,A	DDCBdd5E	BIT 3,(IX+d)
C6nn	ADD A,N	CB40	BIT 0,B	FDCBdd5E	BIT 3,(IY+d)
09	ADD HL,BC	CB41	BIT 0,C	CB5F	BIT 3,A
19	ADD HL,DE	CB42	BIT 0,D	CB58	BIT 3,B

KOD	ROZKAZ	KOD	ROZKAZ	KOD	ROZKAZ
CB59	BIT 3,C	ECnnnn	CALL PE,NN	DBnn	IN A,(N)
CB5A	BIT 3,D	E4nnnn	CALL PO,NN	ED40	IN B,(C)
CB5B	BIT 3,E	CCnnnn	CALL Z,NN	ED48	IN C,(C)
CB5C	BIT 3,H	3F	CCF	ED50	IN D,(C)
CB5D	BIT 3,L	BE	CP (HL)	ED58	IN E,(C)
CB66	BIT 4,(HL)	DDBEdd	CP (IX+d)	ED60	IN H,(C)
DDCBdd66	BIT 4,(IX+d)	FDBEdd	CP (IY+d)	ED68	IN L,(C)
FDCBdd66	BIT 4,(IY+d)	BF	CP A	34	INC (HL)
CB67	BIT 4,A	B8	CP B	DD34dd	INC (IX+d)
CB60	BIT 4,B	B9	CP C	FD34dd	INC (IY+d)
CB61	BIT 4,C	BA	CP D	3C	INC A
CB62	BIT 4,D	BB	CP E	04	INC B
CB63	BIT 4,E	BC	CP H	03	INC BC
CB64	BIT 4,H	BD	CP L	0C	INC C
CB65	BIT 4,L	FEnn	CP N	14	INC D
CB6E	BIT 5,(HL)	EDA9	CPD	13	INC DE
DDCBdd6E	BIT 5,(IX+d)	EDB9	CPDR	1C	INC E
FDCBdd6E	BIT 5,(IY+d)	EDA1	CPI	24	INC H
CB6F	BIT 5,A	EDB1	CPIR	23	INC HL
CB68	BIT 5,B	2F	CPL	DD23	INC IX
CB69	BIT 5,C	27	DAA	FD23	INC IY
CB6A	BIT 5,D	35	DEC (HL)	2C	INC L
CB6B	BIT 5,E	DD35dd	DEC (IX+d)	33	INC SP
CB6C	BIT 5,H	FD35dd	DEC (IY+d)	EDAA	IND
CB6D	BIT 5,L	3D	DEC A	EDBA	INDR
CB76	BIT 6,(HL)	05	DEC B	EDA2	INI
DDCBdd76	BIT 6,(IX+d)	0B	DEC BC	EDB2	INIR
FDCBdd76	BIT 6,(IY+d)	0D	DEC C	E9	JP (HL)
CB77	BIT 6,A	15	DEC D	DDE9	JP (IX)
CB70	BIT 6,B	1B	DEC DE	FDE9	JP (IY)
CB71	BIT 6,C	1D	DEC E	DAnnnn	JP C,NN
CB72	BIT 6,D	25	DEC H	FAnnnn	JP M,NN
CB73	BIT 6,E	2B	DEC HL	D2nnnn	JP NC,NN
CB74	BIT 6,H	DD2B	DEC IX	C3nnnn	JP NN
CB75	BIT 6,L	FD2B	DEC IY	C2nnnn	JP NZ,NN
CB7E	BIT 7,(HL)	2D	DEC L	F2nnnn	JP P,NN
DDCBdd7E	BIT 7,(IX+d)	3B	DEC SP	EAnnnn	JP PE,NN
FDCBdd7E	BIT 7,(IY+d)	F3	DI	E2nnnn	JP PO,NN
CB7F	BIT 7,A	10nn	DJNZ N	CAnnnn	JP Z,NN
CB78	BIT 7,B	FB	EI	38nn	JR C,N
CB79	BIT 7,C	E3	EX (SP),HL	18nn	JR N
CB7A	BIT 7,D	DDE3	EX (SP),IX	30nn	JR NC,N
CB7B	BIT 7,E	FDE3	EX (SP),IY	20nn	JR NZ,N
CB7C	BIT 7,H	08	EX AF,AF'	28nn	JR Z,N
CB7D	BIT 7,L	EB	EX DE,HL	02	LD (BC),A
DCnnnn	CALL C,NN	D9	EXX	12	LD (DE),A
FCnnnn	CALL M,NN	76	HALT	77	LD (HL),A
D4nnnn	CALL NC,NN	ED46	IM 0	70	LD (HL),B
CDnnnn	CALL NN	ED56	IM 1	71	LD (HL),C
C4nnnn	CALL NZ,NN	ED5E	IM 2	72	LD (HL),D
F4nnnn	CALL P,NN	ED78	IN A,(C)	73	LD (HL),E

KOD	ROZKAZ
74	LD (HL),H
75	LD (HL),L
36nn	LD (HL),N
DD77dd	LD (IX+d),A
DD70dd	LD (IX+d),B
DD71dd	LD (IX+d),C
DD72dd	LD (IX+d),D
DD73dd	LD (IX+d),E
DD74dd	LD (IX+d),H
DD75dd	LD (IX+d),L
DD36ddnn	LD (IX+d),N
FD77dd	LD (IY+d),A
FD70dd	LD (IY+d),B
FD71dd	LD (IY+d),C
FD72dd	LD (IY+d),D
FD73dd	LD (IY+d),E
FD74dd	LD (IY+d),H
FD75dd	LD (IY+d),L
FD36ddnn	LD (IY+d),N
32nnnn	LD (NN),A
ED43nnnn	LD (NN),BC
ED53nnnn	LD (NN),DE
22nnnn	LD (NN),HL
DD22nnnn	LD (NN),IX
FD22nnnn	LD (NN),IY
ED73nnnn	LD (NN),SP
0A	LD A,(BC)
1A	LD A,(DE)
7E	LD A,(HL)
DD7Edd	LD A,(IX+d)
FD7Edd	LD A,(IY+d)
3Annnn	LD A,(NN)
7F	LD A,A
78	LD A,B
79	LD A,C
7A	LD A,D
7B	LD A,E
7C	LD A,H
ED57	LD A,I
7D	LD A,L
3Enn	LD A,N
ED5F	LD A,R
46	LD B,(HL)
DD46dd	LD B,(IX+d)
FD46dd	LD B,(IY+d)
47	LD B,A
40	LD B,B
41	LD B,C
42	LD B,D
43	LD B,E
44	LD B,H
45	LD B,L
06nn	LD B,N
ED4Bnnnn	LD BC,(NN)
01nnnn	LD BC,NN
4E	LD C,(HL)
DD4Edd	LD C,(IX+d)
FD4Edd	LD C,(IY+d)
4F	LD C,A
48	LD C,B
49	LD C,C
4A	LD C,D
4B	LD C,E
4C	LD C,H
4D	LD C,L
0Enn	LD C,N
56	LD D,(HL)
DD56dd	LD D,(IX+d)
FD56dd	LD D,(IY+d)
57	LD D,A
50	LD D,B
51	LD D,C
52	LD D,D
53	LD D,E
54	LD D,H
55	LD D,L
16nn	LD D,N
ED5Bnnnn	LD DE,(NN)
11nnnn	LD DE,NN
5E	LD E,(HL)
DD5Edd	LD E,(IX+d)
FD5Edd	LD E,(IY+d)
5F	LD E,A
58	LD E,B
59	LD E,C
5A	LD E,D
5B	LD E,E
5C	LD E,H
5D	LD E,L
1Enn	LD E,N
66	LD H,(HL)
DD66dd	LD H,(IX+d)
FD66dd	LD H,(IY+d)
67	LD H,A
60	LD H,B
61	LD H,C
62	LD H,D
63	LD H,E
64	LD H,H
65	LD H,L
26nn	LD H,N
2Annnn	LD HL,(NN)
21nnnn	LD HL,NN
ED47	LD I,A
DD2Annnn	LD IX,(NN)
DD21nnnn	LD IX,NN
FD2Annnn	LD IY,(NN)
FD21nnnn	LD IY,NN
6E	LD L,(HL)
DD6Edd	LD L,(IX+d)
FD6Edd	LD L,(IY+d)
6F	LD L,A
68	LD L,B
69	LD L,C
6A	LD L,D
6B	LD L,E
6C	LD L,H
6D	LD L,L
2Enn	LD L,N
ED4F	LD R,A
ED7Bnnnn	LD SP,(NN)
F9	LD SP,HL
DDF9	LD SP,IX
FDF9	LD SP,IY
31nnnn	LD SP,NN
EDA8	LDD
EDB8	LDDR
EDA0	LDI
EDB0	LDIR
ED44	NEG
00	NOP
B6	OR (HL)
DDB6dd	OR (IX+d)
FDB6dd	OR (IY+d)
B7	OR A
B0	OR B
B1	OR C
B2	OR D
B3	OR E
B4	OR H
B5	OR L
F6nn	OR N
EDBB	OTDR
EDB3	OTIR
ED79	OUT (C),A
ED41	OUT (C),B
ED49	OUT (C),C
ED51	OUT (C),D
ED59	OUT (C),E
ED61	OUT (C),H
ED69	OUT (C),L
D3nn	OUT (N),A
EDAB	OUTD

KOD	ROZKAZ	KOD	ROZKAZ	KOD	ROZKAZ
EDA3	OUTI	CB9C	RES 3,H	ED4D	RETI
F1	POP AF	CB9D	RES 3,L	ED45	RETN
C1	POP BC	CBA6	RES 4,(HL)	CB16	RL (HL)
D1	POP DE	DDCBddA6	RES 4,(IX+d)	DDCBdd16	RL (IX+d)
E1	POP HL	FDCBddA6	RES 4,(IY+d)	FDCBdd16	RL (IY+d)
DDE1	POP IX	CBA7	RES 4,A	CB17	RL A
FDE1	POP IY	CBA0	RES 4,B	CB10	RL B
F5	PUSH AF	CBA1	RES 4,C	CB11	RL C
C5	PUSH BC	CBA2	RES 4,D	CB12	RL D
D5	PUSH DE	CBA3	RES 4,E	CB13	RL E
E5	PUSH HL	CBA4	RES 4,H	CB14	RL H
DDE5	PUSH IX	CBA5	RES 4,L	CB15	RL L
FDE5	PUSH IY	CBAE	RES 5,(HL)	17	RLA
CB86	RES 0,(HL)	DDCBddAE	RES 5,(IX+d)	CB06	RLC (HL)
DDCBdd86	RES 0,(IX+d)	FDCBddAE	RES 5,(IY+d)	DDCBdd06	RLC (IX+d)
FDCBdd86	RES 0,(IY+d)	CBAF	RES 5,A	FDCBdd06	RLC (IY+d)
CB87	RES 0,A	CBA8	RES 5,B	CB07	RLC A
CB80	RES 0,B	CBA9	RES 5,C	CB00	RLC B
CB81	RES 0,C	CBAA	RES 5,D	CB01	RLC C
CB82	RES 0,D	CBAB	RES 5,E	CB02	RLC D
CB83	RES 0,E	CBAC	RES 5,H	CB03	RLC E
CB84	RES 0,H	CBAD	RES 5,L	CB04	RLC H
CB85	RES 0,L	CBB6	RES 6,(HL)	CB05	RLC L
CB8E	RES 1,(HL)	DDCBddB6	RES 6,(IX+d)	07	RLCA
DDCBdd8E	RES 1,(IX+d)	FDCBddB6	RES 6,(IY+d)	ED6F	RLD
FDCBdd8E	RES 1,(IY+d)	CBB7	RES 6,A	CB1E	RR (HL)
CB8F	RES 1,A	CBB0	RES 6,B	DDCBdd1E	RR (IX+d)
CB88	RES 1,B	CBB1	RES 6,C	FDCBdd1E	RR (IY+d)
CB89	RES 1,C	CBB2	RES 6,D	CB1F	RR A
CB8A	RES 1,D	CBB3	RES 6,E	CB18	RR B
CB8B	RES 1,E	CBB4	RES 6,H	CB19	RR C
CB8C	RES 1,H	CBB5	RES 6,L	CB1A	RR D
CB8D	RES 1,L	CBBE	RES 7,(HL)	CB1B	RR E
CB96	RES 2,(HL)	DDCBddBE	RES 7,(IX+d)	CB1C	RR H
DDCBdd96	RES 2,(IX+d)	FDCBddBE	RES 7,(IY+d)	CB1D	RR L
FDCBdd96	RES 2,(IY+d)	CBBF	RES 7,A	1F	RRA
CB97	RES 2,A	CBB8	RES 7,B	CB0E	RRC (HL)
CB90	RES 2,B	CBB9	RES 7,C	DDCBdd0E	RRC (IX+d)
CB91	RES 2,C	CBBA	RES 7,D	FDCBdd0E	RRC (IY+d)
CB92	RES 2,D	CBBB	RES 7,E	CB0F	RRC A
CB93	RES 2,E	CBBC	RES 7,H	CB08	RRC B
CB94	RES 2,H	CBBD	RES 7,L	CB09	RRC C
CB95	RES 2,L	C9	RET	CB0A	RRC D
CB9E	RES 3,(HL)	D8	RET C	CB0B	RRC E
DDCBdd9E	RES 3,(IX+d)	F8	RET M	CB0C	RRC H
FDCBdd9E	RES 3,(IY+d)	D0	RET NC	CB0D	RRC L
CB9F	RES 3,A	C0	RET NZ	0F	RRCA
CB98	RES 3,B	F0	RET P	ED67	RRD
CB99	RES 3,C	E8	RET PE	C7	RST 00
CB9A	RES 3,D	E0	RET PO	CF	RST 08
CB9B	RES 3,E	C8	RET Z	D7	RST 10

KOD	ROZKAZ	KOD	ROZKAZ	KOD	ROZKAZ
DF	RST 18	CBDE	SET 3,(HL)	DDCBdd26	SLA (IX+d)
E7	RST 20	DDCBddDE	SET 3,(IX+d)	FDCBdd26	SLA (IY+d)
EF	RST 28	FDCBddDE	SET 3,(IY+d)	CB27	SLA A
F7	RST 30	CBDF	SET 3,A	CB20	SLA B
FF	RST 38	CBD8	SET 3,B	CB21	SLA C
9E	SBC A,(HL)	CBD9	SET 3,C	CB22	SLA D
DD9Edd	SBC A,(IX+d)	CBDA	SET 3,D	CB23	SLA E
FD9Edd	SBC A,(IY+d)	CBDB	SET 3,E	CB24	SLA H
9F	SBC A,A	CBDC	SET 3,H	CB25	SLA L
98	SBC A,B	CBDD	SET 3,L	CB2E	SRA (HL)
99	SBC A,C	CBE6	SET 4,(HL)	DDCBdd2E	SRA (IX+d)
9A	SBC A,D	DDCBddE6	SET 4,(IX+d)	FDCBdd2E	SRA (IY+d)
9B	SBC A,E	FDCBddE6	SET 4,(IY+d)	CB2F	SRA A
9C	SBC A,H	CBE7	SET 4,A	CB28	SRA B
9D	SBC A,L	CBE0	SET 4,B	CB29	SRA C
DEnn	SBC A,N	CBE1	SET 4,C	CB2A	SRA D
ED42	SBC HL,BC	CBE2	SET 4,D	CB2B	SRA E
ED52	SBC HL,DE	CBE3	SET 4,E	CB2C	SRA H
ED62	SBC HL,HL	CBE4	SET 4,H	CB2D	SRA L
ED72	SBC HL,SP	CBE5	SET 4,L	CB3E	SRL (HL)
37	SCF	CBEE	SET 5,(HL)	DDCBdd3E	SRL (IX+d)
CBC6	SET 0,(HL)	DDCBddEE	SET 5,(IX+d)	FDCBdd3E	SRL (IY+d)
DDCBddC6	SET 0,(IX+d)	FDCBddEE	SET 5,(IY+d)	CB3F	SRL A
FDCBddC6	SET 0,(IY+d)	CBEF	SET 5,A	CB38	SRL B
CBC7	SET 0,A	CBE8	SET 5,B	CB39	SRL C
CBC0	SET 0,B	CBE9	SET 5,C	CB3A	SRL D
CBC1	SET 0,C	CBEA	SET 5,D	CB3B	SRL E
CBC2	SET 0,D	CBEB	SET 5,E	CB3C	SRL H
CBC3	SET 0,E	CBEC	SET 5,H	CB3D	SRL L
CBC4	SET 0,H	CBED	SET 5,L	96	SUB (HL)
CBC5	SET 0,L	CBF6	SET 6,(HL)	DD96dd	SUB (IX+d)
CBCE	SET 1,(HL)	DDCBddF6	SET 6,(IX+d)	FD96dd	SUB (IY+d)
DDCBddCE	SET 1,(IX+d)	FDCBddF6	SET 6,(IY+d)	97	SUB A
FDCBddCE	SET 1,(IY+d)	CBF7	SET 6,A	90	SUB B
CBCF	SET 1,A	CBF0	SET 6,B	91	SUB C
CBC8	SET 1,B	CBF1	SET 6,C	92	SUB D
CBC9	SET 1,C	CBF2	SET 6,D	93	SUB E
CBCA	SET 1,D	CBF3	SET 6,E	94	SUB H
CBCB	SET 1,E	CBF4	SET 6,H	95	SUB L
CBCC	SET 1,H	CBF5	SET 6,L	D6nn	SUB N
CBCD	SET 1,L	CBFE	SET 7,(HL)	AE	XOR (HL)
CBD6	SET 2,(HL)	DDCBddFE	SET 7,(IX+d)	DDAEdd	XOR (IX+d)
DDCBddD6	SET 2,(IX+d)	FDCBddFE	SET 7,(IY+d)	FDAEdd	XOR (IY+d)
FDCBddD6	SET 2,(IY+d)	CBFF	SET 7,A	AF	XOR A
CBD7	SET 2,A	CBF8	SET 7,B	A8	XOR B
CBD0	SET 2,B	CBF9	SET 7,C	A9	XOR C
CBD1	SET 2,C	CBFA	SET 7,D	AA	XOR D
CBD2	SET 2,D	CBFB	SET 7,E	AB	XOR E
CBD3	SET 2,E	CBFC	SET 7,H	AC	XOR H
CBD4	SET 2,H	CBFD	SET 7,L	AD	XOR L
CBD5	SET 2,L	CB26	SLA (HL)	EEnn	XOR N

Tablica kodów rozkazów mikroprocesora Z80 uszeregowana według kodów

(Rozkazy prefiksowane zostały zebrane razem.)

Oznaczenia takie jak w dodatku C.1.

KOD	ROZKAZ
00	NOP
01nnnn	LD BC,NN
02	LD (BC),A
03	INC BC
04	INC B
05	DEC B
06nn	LD B,N
07	RLCA
08	EX AF,AF'
09	ADD HL,BC
0A	LD A,(BC)
0B	DEC BC
0C	INC C
0D	DEC C
0Enn	LD C,N
0F	RRCA
10nn	DJNZ N
11nnnn	LD DE,NN
12	LD (DE),A
13	INC DE
14	INC D
15	DEC D
16nn	LD D,N
17	RLA
18nn	JR N
19	ADD HL,DE
1A	LD A,(DE)
1B	DEC DE
1C	INC E
1D	DEC E
1Enn	LD E,N
1F	RRA
20nn	JR NZ,N
21nnnn	LD HL,NN
22nnnn	LD (NN),HL
23	INC HL
24	INC H
25	DEC H
26nn	LD H,N
27	DAA
28nn	JR Z,N
29	ADD HL,HL
2Annnn	LD HL,(NN)
2B	DEC HL
2C	INC L
2D	DEC L
2Enn	LD L,N
2F	CPL
30nn	JR NC,N
31nnnn	LD SP,NN
32nnnn	LD (NN),A
33	INC SP
34	INC (HL)
35	DEC (HL)
36nn	LD (HL),N
37	SCF
38nn	JR C,N
39	ADD HL,SP
3Annnn	LD A,(NN)
3B	DEC SP
3C	INC A
3D	DEC A
3Enn	LD A,N
3F	CCF
40	LD B,B
41	LD B,C
42	LD B,D
43	LD B,E
44	LD B,H
45	LD B,L
46	LD B,(HL)
47	LD B,A
48	LD C,B
49	LD C,C
4A	LD C,D
4B	LD C,E
4C	LD C,H
4D	LD C,L
4E	LD C,(HL)
4F	LD C,A
50	LD D,B
51	LD D,C
52	LD D,D
53	LD D,E

KOD	ROZKAZ	KOD	ROZKAZ	KOD	ROZKAZ
54	LD D,H	87	ADD A,A	BA	CP D
55	LD D,L	88	ADC A,B	BB	CP E
56	LD D,(HL)	89	ADC A,C	BC	CP H
57	LD D,A	8A	ADC A,D	BD	CP L
58	LD E,B	8B	ADC A,E	BE	CP (HL)
59	LD E,C	8C	ADC A,H	BF	CP A
5A	LD E,D	8D	ADC A,L	C0	RET NZ
5B	LD E,E	8E	ADC A,(HL)	C1	POP BC
5C	LD E,H	8F	ADC A,A	C2nnnn	JP NZ,NN
5D	LD E,L	90	SUB B	C3nnnn	JP NN
5E	LD E,(HL)	91	SUB C	C4nnnn	CALL NZ,NN
5F	LD E,A	92	SUB D	C5	PUSH BC
60	LD H,B	93	SUB E	C6nn	ADD A,N
61	LD H,C	94	SUB H	C7	RST 00
62	LD H,D	95	SUB L	C8	RET Z
63	LD H,E	96	SUB (HL)	C9	RET
64	LD H,H	97	SUB A	CAnnnn	JP Z,NN
65	LD H,L	98	SBC A,B	CCnnnn	CALL Z,NN
66	LD H,(HL)	99	SBC A,C	CDnnnn	CALL NN
67	LD H,A	9A	SBC A,D	CEnn	ADC A,N
68	LD L,B	9B	SBC A,E	CF	RST 08
69	LD L,C	9C	SBC A,H	D0	RET NC
6A	LD L,D	9D	SBC A,L	D1	POP DE
6B	LD L,E	9E	SBC A,(HL)	D2nnnn	JP NC,NN
6C	LD L,H	9F	SBC A,A	D3nn	OUT (N),A
6D	LD L,L	A0	AND B	D4nnnn	CALL NC,NN
6E	LD L,(HL)	A1	AND C	D5	PUSH DE
6F	LD L,A	A2	AND D	D6nn	SUB N
70	LD (HL),B	A3	AND E	D7	RST 10
71	LD (HL),C	A4	AND H	D8	RET C
72	LD (HL),D	A5	AND L	D9	EXX
73	LD (HL),E	A6	AND (HL)	DAnnnn	JP C,NN
74	LD (HL),H	A7	AND A	DBnn	IN A,(N)
75	LD (HL),L	A8	XOR B	DCnnnn	CALL C,NN
76	HALT	A9	XOR C	DEnn	SBC A,N
77	LD (HL),A	AA	XOR D	DF	RST 18
78	LD A,B	AB	XOR E	E0	RET PO
79	LD A,C	AC	XOR H	E1	POP HL
7A	LD A,D	AD	XOR L	E2nnnn	JP PO,NN
7B	LD A,E	AE	XOR (HL)	E3	EX (SP),HL
7C	LD A,H	AF	XOR A	E4nnnn	CALL PO,NN
7D	LD A,L	B0	OR B	E5	PUSH HL
7E	LD A,(HL)	B1	OR C	E6nn	AND N
7F	LD A,A	B2	OR D	E7	RST 20
80	ADD A,B	B3	OR E	E8	RET PE
81	ADD A,C	B4	OR H	E9	JP (HL)
82	ADD A,D	B5	OR L	EAnnnn	JP PE,NN
83	ADD A,E	B6	OR (HL)	EB	EX DE,HL
84	ADD A,H	B7	OR A	ECnnnn	CALL PE,NN
85	ADD A,L	B8	CP B	EEnn	XOR N
86	ADD A,(HL)	B9	CP C	EF	RST 28

KOD	ROZKAZ	KOD	ROZKAZ	KOD	ROZKAZ
F0	RET P	CB24	SLA H	CB5F	BIT 3,A
F1	POP AF	CB25	SLA L	CB60	BIT 4,B
F2nnnn	JP P,NN	CB26	SLA (HL)	CB61	BIT 4,C
F3	DI	CB27	SLA A	CB62	BIT 4,D
F4nnnn	CALL P,NN	CB28	SRA B	CB63	BIT 4,E
F5	PUSH AF	CB29	SRA C	CB64	BIT 4,H
F6nn	OR N	CB2A	SRA D	CB65	BIT 4,L
F7	RST 30	CB2B	SRA E	CB66	BIT 4,(HL)
F8	RET M	CB2C	SRA H	CB67	BIT 4,A
F9	LD SP,HL	CB2D	SRA L	CB68	BIT 5,B
FAnnnn	JP M,NN	CB2E	SRA (HL)	CB69	BIT 5,C
FB	EI	CB2F	SRA A	CB6A	BIT 5,D
FCnnnn	CALL M,NN	CB38	SRL B	CB6B	BIT 5,E
FEnn	CP N	CB39	SRL C	CB6C	BIT 5,H
FF	RST 38	CB3A	SRL D	CB6D	BIT 5,L
CB00	RLC B	CB3B	SRL E	CB6E	BIT 5,(HL)
CB01	RLC C	CB3C	SRL H	CB6F	BIT 5,A
CB02	RLC D	CB3D	SRL L	CB70	BIT 6,B
CB03	RLC E	CB3E	SRL (HL)	CB71	BIT 6,C
CB04	RLC H	CB3F	SRL A	CB72	BIT 6,D
CB05	RLC L	CB40	BIT 0,B	CB73	BIT 6,E
CB06	RLC (HL)	CB41	BIT 0,C	CB74	BIT 6,H
CB07	RLC A	CB42	BIT 0,D	CB75	BIT 6,L
CB08	RRC B	CB43	BIT 0,E	CB76	BIT 6,(HL)
CB09	RRC C	CB44	BIT 0,H	CB77	BIT 6,A
CB0A	RRC D	CB45	BIT 0,L	CB78	BIT 7,B
CB0B	RRC E	CB46	BIT 0,(HL)	CB79	BIT 7,C
CB0C	RRC H	CB47	BIT 0,A	CB7A	BIT 7,D
CB0D	RRC L	CB48	BIT 1,B	CB7B	BIT 7,E
CB0E	RRC (HL)	CB49	BIT 1,C	CB7C	BIT 7,H
CB0F	RRC A	CB4A	BIT 1,D	CB7D	BIT 7,L
CB10	RL B	CB4B	BIT 1,E	CB7E	BIT 7,(HL)
CB11	RL C	CB4C	BIT 1,H	CB7F	BIT 7,A
CB12	RL D	CB4D	BIT 1,L	CB80	RES 0,B
CB13	RL E	CB4E	BIT 1,(HL)	CB81	RES 0,C
CB14	RL H	CB4F	BIT 1,A	CB82	RES 0,D
CB15	RL L	CB50	BIT 2,B	CB83	RES 0,E
CB16	RL (HL)	CB51	BIT 2,C	CB84	RES 0,H
CB17	RL A	CB52	BIT 2,D	CB85	RES 0,L
CB18	RR B	CB53	BIT 2,E	CB86	RES 0,(HL)
CB19	RR C	CB54	BIT 2,H	CB87	RES 0,A
CB1A	RR D	CB55	BIT 2,L	CB88	RES 1,B
CB1B	RR E	CB56	BIT 2,(HL)	CB89	RES 1,C
CB1C	RR H	CB57	BIT 2,A	CB8A	RES 1,D
CB1D	RR L	CB58	BIT 3,B	CB8B	RES 1,E
CB1E	RR (HL)	CB59	BIT 3,C	CB8C	RES 1,H
CB1F	RR A	CB5A	BIT 3,D	CB8D	RES 1,L
CB20	SLA B	CB5B	BIT 3,E	CB8E	RES 1,(HL)
CB21	SLA C	CB5C	BIT 3,H	CB8F	RES 1,A
CB22	SLA D	CB5D	BIT 3,L	CB90	RES 2,B
CB23	SLA E	CB5E	BIT 3,(HL)	CB91	RES 2,C

KOD	ROZKAZ	KOD	ROZKAZ	KOD	ROZKAZ
CB92	RES 2,D	CBC5	SET 0,L	CBF8	SET 7,B
CB93	RES 2,E	CBC6	SET 0,(HL)	CBF9	SET 7,C
CB94	RES 2,H	CBC7	SET 0,A	CBFA	SET 7,D
CB95	RES 2,L	CBC8	SET 1,B	CBFB	SET 7,E
CB96	RES 2,(HL)	CBC9	SET 1,C	CBFC	SET 7,H
CB97	RES 2,A	CBCA	SET 1,D	CBFD	SET 7,L
CB98	RES 3,B	CBCB	SET 1,E	CBFE	SET 7,(HL)
CB99	RES 3,C	CBCC	SET 1,H	CBFF	SET 7,A
CB9A	RES 3,D	CBCD	SET 1,L	DD09	ADD IX,BC
CB9B	RES 3,E	CBCE	SET 1,(HL)	DD19	ADD IX,DE
CB9C	RES 3,H	CBCF	SET 1,A	DD21nnnn	LD IX,NN
CB9D	RES 3,L	CBD0	SET 2,B	DD22nnnn	LD (NN),IX
CB9E	RES 3,(HL)	CBD1	SET 2,C	DD23	INC IX
CB9F	RES 3,A	CBD2	SET 2,D	DD29	ADD IX,IX
CBA0	RES 4,B	CBD3	SET 2,E	DD2Annnn	LD IX,(NN)
CBA1	RES 4,C	CBD4	SET 2,H	DD2B	DEC IX
CBA2	RES 4,D	CBD5	SET 2,L	DD34dd	INC (IX+d)
CBA3	RES 4,E	CBD6	SET 2,(HL)	DD35dd	DEC (IX+d)
CBA4	RES 4,H	CBD7	SET 2,A	DD36ddnn	LD (IX+d),N
CBA5	RES 4,L	CBD8	SET 3,B	DD39	ADD IX,SP
CBA6	RES 4,(HL)	CBD9	SET 3,C	DD46dd	LD B,(IX+d)
CBA7	RES 4,A	CBDA	SET 3,D	DD4Edd	LD C,(IX+d)
CBA8	RES 5,B	CBDB	SET 3,E	DD56dd	LD D,(IX+d)
CBA9	RES 5,C	CBDC	SET 3,H	DD5Edd	LD E,(IX+d)
CBAA	RES 5,D	CBDD	SET 3,L	DD66dd	LD H,(IX+d)
CBAB	RES 5,E	CBDE	SET 3,(HL)	DD6Edd	LD L,(IX+d)
CBAC	RES 5,H	CBDF	SET 3,A	DD70dd	LD (IX+d),B
CBAD	RES 5,L	CBE0	SET 4,B	DD71dd	LD (IX+d),C
CBAE	RES 5,(HL)	CBE1	SET 4,C	DD72dd	LD (IX+d),D
CBAF	RES 5,A	CBE2	SET 4,D	DD73dd	LD (IX+d),E
CBB0	RES 6,B	CBE3	SET 4,E	DD74dd	LD (IX+d),H
CBB1	RES 6,C	CBE4	SET 4,H	DD75dd	LD (IX+d),L
CBB2	RES 6,D	CBE5	SET 4,L	DD77dd	LD (IX+d),A
CBB3	RES 6,E	CBE6	SET 4,(HL)	DD7Edd	LD A,(IX+d)
CBB4	RES 6,H	CBE7	SET 4,A	DD86dd	ADD A,(IX+d)
CBB5	RES 6,L	CBE8	SET 5,B	DD8Edd	ADC A,(IX+d)
CBB6	RES 6,(HL)	CBE9	SET 5,C	DD96dd	SUB (IX+d)
CBB7	RES 6,A	CBEA	SET 5,D	DD9Edd	SBC A,(IX+d)
CBB8	RES 7,B	CBEB	SET 5,E	DDA6dd	AND (IX+d)
CBB9	RES 7,C	CBEC	SET 5,H	DDAEdd	XOR (IX+d)
CBBA	RES 7,D	CBED	SET 5,L	DDB6dd	OR (IX+d)
CBBB	RES 7,E	CBEE	SET 5,(HL)	DDBEdd	CP (IX+d)
CBBC	RES 7,H	CBEF	SET 5,A	DDE1	POP IX
CBBD	RES 7,L	CBF0	SET 6,B	DDE3	EX (SP),IX
CBBE	RES 7,(HL)	CBF1	SET 6,C	DDE5	PUSH IX
CBBF	RES 7,A	CBF2	SET 6,D	DDE9	JP (IX)
CBC0	SET 0,B	CBF3	SET 6,E	DDF9	LD SP,IX
CBC1	SET 0,C	CBF4	SET 6,H	DDCBdd06	RLC (IX+d)
CBC2	SET 0,D	CBF5	SET 6,L	DDCBdd0E	RRC (IX+d)
CBC3	SET 0,E	CBF6	SET 6,(HL)	DDCBdd16	RL (IX+d)
CBC4	SET 0,H	CBF7	SET 6,A	DDCBdd1E	RR (IX+d)

KOD	ROZKAZ	KOD	ROZKAZ	KOD	ROZKAZ
DDCBdd26	SLA (IX+d)	ED5E	IM 2	FD71dd	LD (IY+d),C
DDCBdd2E	SRA (IX+d)	ED5F	LD A,R	FD72dd	LD (IY+d),D
DDCBdd3E	SRL (IX+d)	ED60	IN H,(C)	FD73dd	LD (IY+d),E
DDCBdd46	BIT 0,(IX+d)	ED61	OUT (C),H	FD74dd	LD (IY+d),H
DDCBdd4E	BIT 1,(IX+d)	ED62	SBC HL,HL	FD75dd	LD (IY+d),L
DDCBdd56	BIT 2,(IX+d)	ED67	RRD	FD77dd	LD (IY+d),A
DDCBdd5E	BIT 3,(IX+d)	ED68	IN L,(C)	FD7Edd	LD A,(IY+d)
DDCBdd66	BIT 4,(IX+d)	ED69	OUT (C),L	FD86dd	ADD A,(IY+d)
DDCBdd6E	BIT 5,(IX+d)	ED6A	ADC HL,HL	FD8Edd	ADC A,(IY+d)
DDCBdd76	BIT 6,(IX+d)	ED6F	RLD	FD96dd	SUB (IY+d)
DDCBdd7E	BIT 7,(IX+d)	ED72	SBC HL,SP	FD9Edd	SBC A,(IY+d)
DDCBdd86	RES 0,(IX+d)	ED73nnnn	LD (NN),SP	FDA6dd	AND (IY+d)
DDCBdd8E	RES 1,(IX+d)	ED78	IN A,(C)	FDAEdd	XOR (IY+d)
DDCBdd96	RES 2,(IX+d)	ED79	OUT (C),A	FDB6dd	OR (IY+d)
DDCBdd9E	RES 3,(IX+d)	ED7A	ADC HL,SP	FDBEdd	CP (IY+d)
DDCBddA6	RES 4,(IX+d)	ED7Bnnnn	LD SP,(NN)	FDE1	POP IY
DDCBddAE	RES 5,(IX+d)	EDA0	LDI	FDE3	EX (SP),IY
DDCBddB6	RES 6,(IX+d)	EDA1	CPI	FDE5	PUSH IY
DDCBddBE	RES 7,(IX+d)	EDA2	INI	FDE9	JP (IY)
DDCBddC6	SET 0,(IX+d)	EDA3	OUTI	FDF9	LD SP,IY
DDCBddCE	SET 1,(IX+d)	EDA8	LDD	FDCBdd06	RLC (IY+d)
DDCBddD6	SET 2,(IX+d)	EDA9	CPD	FDCBdd0E	RRC (IY+d)
DDCBddDE	SET 3,(IX+d)	EDAA	IND	FDCBdd16	RL (IY+d)
DDCBddE6	SET 4,(IX+d)	EDAB	OUTD	FDCBdd1E	RR (IY+d)
DDCBddEE	SET 5,(IX+d)	EDB0	LDIR	FDCBdd26	SLA (IY+d)
DDCBddF6	SET 6,(IX+d)	EDB1	CPIR	FDCBdd2E	SRA (IY+d)
DDCBddFE	SET 7,(IX+d)	EDB2	INIR	FDCBdd3E	SRL (IY+d)
ED40	IN B,(C)	EDB3	OTIR	FDCBdd46	BIT 0,(IY+d)
ED41	OUT (C),B	EDB8	LDDR	FDCBdd4E	BIT 1,(IY+d)
ED42	SBC HL,BC	EDB9	CPDR	FDCBdd56	BIT 2,(IY+d)
ED43nnnn	LD (NN),BC	EDBA	INDR	FDCBdd5E	BIT 3,(IY+d)
ED44	NEG	EDBB	OTDR	FDCBdd66	BIT 4,(IY+d)
ED45	RETN	FD09	ADD IY,BC	FDCBdd6E	BIT 5,(IY+d)
ED46	IM 0	FD19	ADD IY,DE	FDCBdd76	BIT 6,(IY+d)
ED47	LD I,A	FD21nnnn	LD IY,NN	FDCBdd7E	BIT 7,(IY+d)
ED48	IN C,(C)	FD22nnnn	LD (NN),IY	FDCBdd86	RES 0,(IY+d)
ED49	OUT (C),C	FD23	INC IY	FDCBdd8E	RES 1,(IY+d)
ED4A	ADC HL,BC	FD29	ADD IY,IY	FDCBdd96	RES 2,(IY+d)
ED4Bnnnn	LD BC,(NN)	FD2Annnn	LD IY,(NN)	FDCBdd9E	RES 3,(IY+d)
ED4D	RETI	FD2B	DEC IY	FDCBddA6	RES 4,(IY+d)
ED4F	LD R,A	FD34dd	INC (IY+d)	FDCBddAE	RES 5,(IY+d)
ED50	IN D,(C)	FD35dd	DEC (IY+d)	FDCBddB6	RES 6,(IY+d)
ED51	OUT (C),D	FD36ddnn	LD (IY+d),N	FDCBddBE	RES 7,(IY+d)
ED52	SBC HL,DE	FD39	ADD IY,SP	FDCBddC6	SET 0,(IY+d)
ED53nnnn	LD (NN),DE	FD46dd	LD B,(IY+d)	FDCBddCE	SET 1,(IY+d)
ED56	IM 1	FD4Edd	LD C,(IY+d)	FDCBddD6	SET 2,(IY+d)
ED57	LD A,I	FD56dd	LD D,(IY+d)	FDCBddDE	SET 3,(IY+d)
ED58	IN E,(C)	FD5Edd	LD E,(IY+d)	FDCBddE6	SET 4,(IY+d)
ED59	OUT (C),E	FD66dd	LD H,(IY+d)	FDCBddEE	SET 5,(IY+d)
ED5A	ADC HL,DE	FD6Edd	LD L,(IY+d)	FDCBddF6	SET 6,(IY+d)
ED5Bnnnn	LD DE,(NN)	FD70dd	LD (IY+d),B	FDCBddFE	SET 7,(IY+d)

Literatura

1. Aho A. V., Hopcroft J. E., Ullman J. D.: *Projektowanie i analiza algorytmów komputerowych*. Warszawa, PWN 1983.
2. Baker T.: *Mastering machine code on your ZX81*. Londyn, Interface 1980.
3. Coffron J. W.: *Z80 Applications*. Berkeley, SYBEX 1983.
4. Flores I.: *Arytmetyka maszyn cyfrowych*. Warszawa, WNT 1970.
5. Grabowski J., Koślacz S.: *Podstawy i praktyka programowania mikroprocesorów*. Warszawa, WNT 1980.
6. Knuth D. E.: *The art of computer programming*. Vol. 2: *Seminumerical algorithms*. Reading, Mass., Addison-Wesley 1969.
7. Lipowski J. i in.: *Modułowe systemy mikrokomputerowe*. Warszawa, WNT 1984.
8. Maćków P.: Niepublikowane rozkazy mikroprocesora Z80. *Informatyka*, 7, 1984, s. 15.
9. Misiurewicz P.: *Układy mikroprocesorowe*. Warszawa, WNT 1983.
10. Nichols E. A., Nichols J. C., Rony P. R.: *Z-80 Microprocessor*. Indianapolis, Howard Sams & Co. 1979.
11. Sacha K., Rydzewski A.: *Mikroprocesor w pytaniach i odpowiedziach*. Warszawa, WNT 1985.
12. Zaks R.: *Programming the Z80*. Berkeley, SYBEX 1982.
13. Zilog Z80 — CPU Product Specification, 1979.
14. Zilog Z80 Technical Manual, 1979.

Skorowidz

Adres odświeżania 10, 18
akumulator 21, 37
ALU 20
arcus cosinus 160
arytmetyka zmiennopozycyjna 122, 157
arytmometr 20
ASCII 41, 110
asembler 38

Bajt 12
bank rejestrów 23, 56
BCD 12, 14, 63, 117
bit 12, 108
blokowanie przerwań 31, 32, 75, 83

Cecha liczby 122
CP/M 9, 90
cykl maszynowy 26
– odświeżania 24
– rozkazowy 26

Deasembler 107, 140
dekodowanie rozkazu 19, 26, 29
dekrementacja 61, 70
dyrektywy asemblera 40
dzielenie 112, 114, 120, 154

Etykieta rozkazu 39

FORTH 106
funkcja wykładnicza 158

Indeksowanie tablic 133
inkrementacja 61, 70
interpretacja 90, 105, 143
interpretator 90, 105, 143

Kasowanie procesora 19, 34
klasa operacji 43
kolejka 99, 128
konwersja 110, 114, 146
koprocedura 97

Liczby 12
– losowe 25, 112
– szesnastkowe 39
– wielobajtowe 116
– zmiennopozycyjne 122, 157
licznik odświeżania 21, 24
– rozkazów 21, 24, 79
lista (struktura danych) 119, 133
– dwukierunkowa 138
logarytm 159

Magistrala 20
makroinstrukcja (MACRO) 42, 96
mantysa liczby 122
maskowanie 60, 109

metoda Newtona 120, 158
mnożenie 113, 119, 152
modularyzacja programów 91

Nadmiar 14, 22, 47
napięcia zasilania 10, 16

Odczyt odświeżający 29
– pamięci 29
odwracanie tablic 125
optymalizacja 92, 101, 119, 123

Parametr procedury 93
parzystość 15, 22, 47
pętla 102
pierwiastek kwadratowy 158
podprogram 80, 91
port wejścia/wyjścia 11, 30, 85
praca jałowa 18, 29, 75
prąd zasilania procesora 16
prefiks rozkazu 42
procesor Intel 8080 9, 47
przemieszczalność 95
przeniesienie 13, 22
przerwanie 31
– maskowalne 19, 31, 75
– niemaskowalne 19, 31
przestrzeń adresowa 36, 85

Rejestr 10, 21
– indeksowy 11, 21, 23
– stosu 11, 21, 24, 37, 54, 56
– wektora przerwań 11, 25
– wirtualny M 43
– wskaźników stanu 21
– WZ 24, 78
rejestry HX, LX, HY i LY 45, 47, 49
– IFF1, IFF2 31, 52, 83
rekursja 82, 92, 159
rozgałęzienie 99
rozkaz HALT 29, 42, 48, 75
– korekcji dziesiętnej DAA 14, 63, 117
rozkazy indeksowane 23, 49, 51
– nieoficjalne 44, 49, 65, 85
rozkazy prefiksowane 42
— zamiany 23, 56

Sinus 159
skok 77
— względny 78
słowo maszynowe 12
sortowanie 132
standard TTL 17
stos 94, 122
stóg 128
sygnał $\overline{\text{HALT}}$ 18, 29
— $\overline{\text{INT}}$ 19, 31
— $\overline{\text{IORQ}}$ 18, 32
— $\overline{\text{MI}}$ 18, 28
— $\overline{\text{MREQ}}$ 18
— $\overline{\text{NMI}}$ 19, 31
— $\overline{\text{RESET}}$ 19, 24
— $\overline{\text{RFSH}}$ 18
— $\overline{\text{WAIT}}$ 19, 28
sygnały $\overline{\text{BUSAK}}$, $\overline{\text{BUSRQ}}$ 19, 34
— $\overline{\text{RD}}$, $\overline{\text{WR}}$ 18
— sterujące 18
szyna adresów 20
— danych 20
— sterowania 20

Tablica 125
– wielowymiarowa 133
takt zegarowy 27
taktowanie 10, 17
tetrada 14, 63, 67
tryb adresowania 36
– obsługi przerwania 33, 75

Wektor przerwań 33
współprogram 97

Zapis dwójkowy uzupełnieniowy 13
– pamięci 29
zawieszenie szyn 10, 19, 34
zegar 10, 17
zerowanie procesora 19, 34
Zilog Inc. 9, 11
znak liczby 13, 22, 117

Mikrokomputery

Wydano

J. Bielecki — Język C — interpretacja standardu
J. Bielecki — System operacyjny ISIS-II
M. Kalinowska-Iszkowska, W. Iszkowski — Klucze do Basicu
J. Karczmarczuk — Mikroprocesor Z80

W przygotowaniu

J. Bielecki — Fortran 77
J. Bielecki — Język Forth
J. Bielecki — Turbo Pascal wersja 3.0
J. Bielecki — Turbo Pascal z grafiką dla IBM PC
J. Bielecki — Wprowadzenie do języka C
J. Boisgontier, S. Brébion — Basic dla wszystkich
B. Frelek — Commodore 64
D. Hearn, M. P. Baker — Grafika mikrokomputerowa
W. Iszkowski — Nauka programowania w języku BASIC dla początkujących
W. Link — Jak mierzyć, sterować i regulować za pomocą Basicu?
R. Świniarski — CP/M — system operacyjny mikrokomputerów

Wydawnictwa Naukowo-Techniczne informują wszystkich użytkowników i entuzjastów mikrokomputerów, że ukaże się książka autorstwa K. Sachy, P. Misiurewicza i T. Kręglewskiego pt.:

„Przewodnik po technice mikrokomputerowej".

Książka stanowi zbiór informacji — w postaci haseł ułożonych alfabetycznie — dotyczących tej żywiołowo rozwijającej się dziedziny. Opisano w niej zarówno pojęcia podstawowe, jak i najnowsze tendencje. Omówiono podstawowe zagadnienia związane z budową, zasadami działania i oprogramowaniem mikrokomputerów: mikroprocesory i ich elementy, pamięci mikrokomputerów, układy sprzęgające, urządzenia zewnętrzne, systemy operacyjne, programy usługowe, wybrane języki programowania, a także zastosowania mikrokomputerów m.in. w systemach czasu rzeczywistego i do sterowania urządzeń.
Książka zawiera również skorowidz większości powszechnie spotykanych terminów z zakresu techniki mikrokomputerowej oraz ich odpowiedniki w czterech językach obcych.

INTERAMS

BIURO USŁUG KOMPUTEROWYCH

Spółka z o.o.

00-867 Warszawa, ul. Chłodna 35/37, tel. 247823

Nr konta: BSRz Warszawa, Podwale 17a, 136-61-132118 Identyfikator 901235

---

oferuje

Zintegrowany pakiet oprogramowania, umożliwiający kompleksowe zarządzanie każdym przedsiębiorstwem

PAKIETY PODSTAWOWE

- System Informacji Kierownictwa
- Płace
- Kadry
- System Finansowo-Księgowy
- Gospodarka Materiałowa
- Gospodarka Środkami Trwałymi
- Gospodarka Przedm. Nietrwałymi
- Kalkulacja Podatku od Ponadnormatywnych Wynagrodzeń

PRZEDSIĘBIORSTWA HANDLOWE

Specjalistyczne Oprogramowanie dla Przedsiębiorstw Handlowych

PRZEDSIĘBIORSTWA TRANSPORTOWE

- Ewidencja Kart Drogowych
- Kalkulacja Cen Środków Transportu

PRZEDSIĘBIORSTWA BUDOWLANE

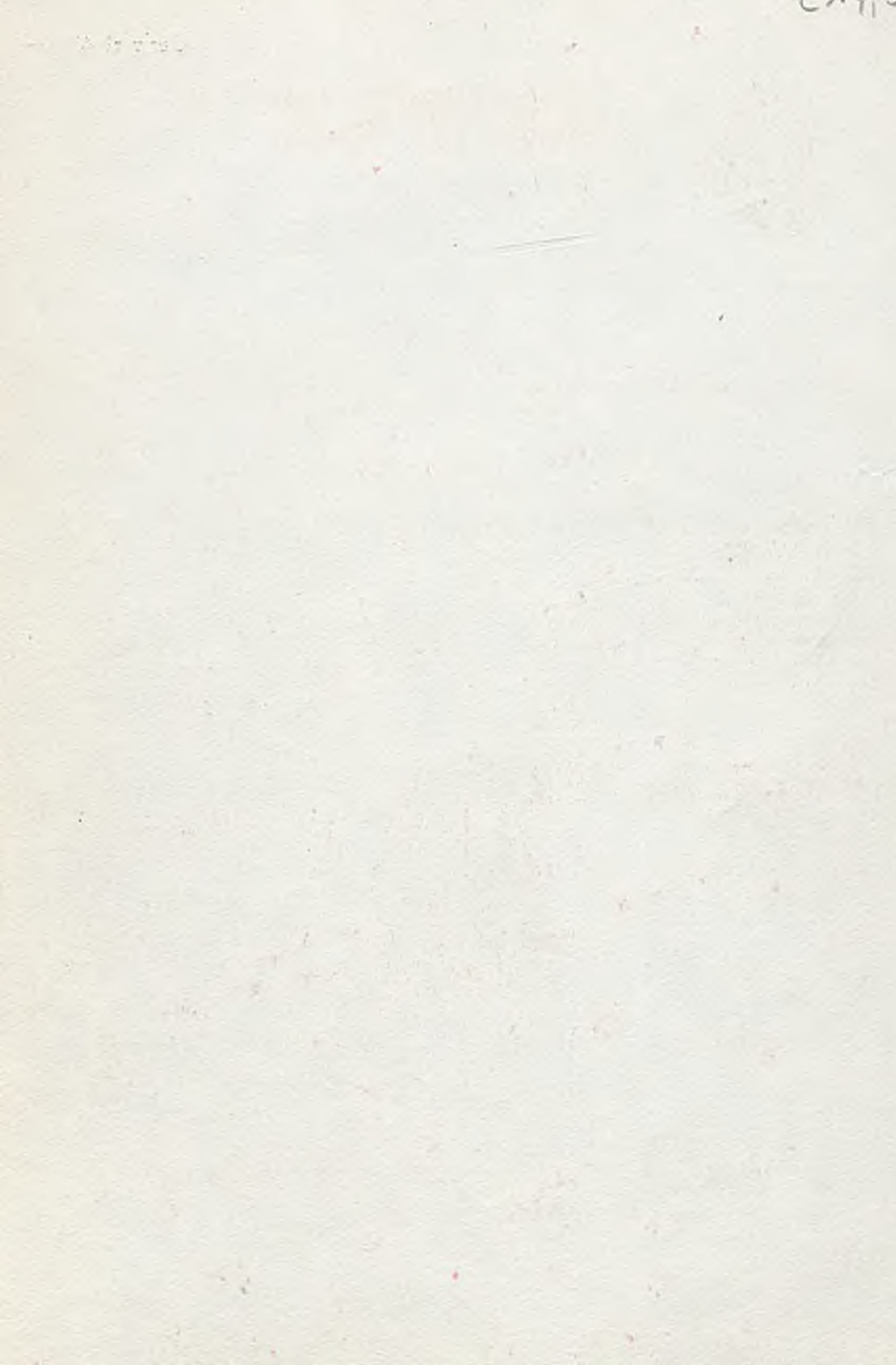
- Kosztorysowanie
- Planowanie Inwestycji

PRZEDSIĘBIORSTWA PRODUKCYJNE

- Techniczne Przygotowanie Produkcji
- Sterowanie Produkcją
- Planowanie Produkcji

Zapraszamy na pokazy systemów i sprzętu do salonu przy ul.Chłodnej 35/37

UWAGA! *Prowadzimy kursy dla Menadżerów*



BG Politechniki Śląskiej
nr inw.: 105 - 102869
Mg S.102869